x86-64 Programming I
CSE 351 Summer 2021

Instructor:
Mara Kirdani-Ryan

Teaching Assistants:
Kashish Aggarwal
Nick Durand
Colton Jobes
Tim Mandzyuk

http://xkcd.com/409/
Gentle and Loving Reminders!

- hw6 & hw7 due Friday (7/9) – 8pm
- hw8 due Monday (7/12) – 8pm

- Lab 1b due Friday at 8pm (7/9)
  - Submit aisle_manager.c, store_client.c, and lab1Breflect.txt
Gentle and Loving Reminders!

- Unit Summary 1 Due Monday 7/12!
  - Submitted via Gradescope
  - We’re here to help! Especially if you’re feeling stuck!
  - Task 3 is going out today

- We want to give you an opportunity to reflect and synthesize the material!
  - Exams are pretty terrible for this!
How are y’all feeling today?
Second Floor! Programs!

- Values in modern processors
- Critical Analysis
- Accessibility, agency and support
- Establishing and extending structures
- How programs are executed by a processor
Learning Objectives for Today!

- You should be able to:
  - Explain what an ISA is, in plain language
  - Explain the difference between registers and memory, and the tradeoffs between using each
  - Explain the effects of mov and arithmetic x86 instructions
  - Translate single lines of arithmetic C code (memory accesses and math) into assembly, and vice versa
  - Explain the growth of monopolies in every industry, across the last 50 years
  - Explain the conditions that the x86 architecture and the IBM PC were created in, and how that affected their implementations
Architecture: the HW/SW 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

Clang

x86-64

ARMv8 (AArch64/A64)

Intel Pentium 4
Intel Core 2
Intel Core i7
AMD Opteron
AMD Athlon
ARM Cortex-A53
Apple A7
Definitions

- **Architecture (ISA):** The parts of a processor design that one needs to understand to write assembly code
  - “What is directly visible to software”
  - “Interface contract between HW and SW”

- **Microarchitecture:** Implementation of the architecture
  - CSE/EE 469
Instruction Set Architectures

- 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

- **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
## Dominant ISAs

<table>
<thead>
<tr>
<th></th>
<th>x86</th>
<th>ARM architectures</th>
<th>RISC-V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Designer</td>
<td>Intel, AMD</td>
<td>ARM Holdings</td>
<td>University of California, Berkeley</td>
</tr>
<tr>
<td>Bits</td>
<td>16-bit, 32-bit and 64-bit</td>
<td>32-bit, 64-bit</td>
<td>32 · 64 · 128</td>
</tr>
<tr>
<td>Introduced</td>
<td>1978 (16-bit), 1985 (32-bit), 2003 (64-bit)</td>
<td>1985; 31 years ago</td>
<td>2010</td>
</tr>
<tr>
<td>Design</td>
<td>CISC</td>
<td>RISC</td>
<td>RISC</td>
</tr>
<tr>
<td>Type</td>
<td>Register-memory</td>
<td>Register-Register</td>
<td>Load-store</td>
</tr>
<tr>
<td>Encoding</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.</td>
<td>Variable</td>
</tr>
<tr>
<td>Endianness</td>
<td>Little</td>
<td>Bi (little as default)</td>
<td>Little[1][3]</td>
</tr>
</tbody>
</table>

- **Macbooks & PCs** (Core i3, i5, i7, i9)
- **x86-64 Instruction Set**
- **Smartphone-like devices** (iPhone, iPad, Raspberry Pi)
- **ARM Instruction Set**
- **Mostly research, though some footholds in industry, especially in embedded**
Writing Assembly Code? In 2021???

- You probably won’t 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
    - Which optimizations are done by the compiler?
    - Understanding sources of program inefficiency
  - Implementing systems software
    - The “states” of processes that the OS must manage
    - Special units (timers, I/O, etc.) inside processor!
  - Fighting malicious software
    - Distributed software is in binary form
Assembly Programmer’s View

- Programmer-visible state
  - PC: 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, …
  - “Intel”: used by Intel documentation, Intel tools, …
  - Must know which you’re reading
What is a Register?

- 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
### x86-64 Integer Registers – 64 bits wide

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
</tr>
<tr>
<td>%r8</td>
<td>%r8d</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>
Some History: IA32 Registers – 32 bits wide

<table>
<thead>
<tr>
<th>Register</th>
<th>Bit</th>
<th>Purpose</th>
<th>Name Origin</th>
<th>16-bit virtual registers (backwards compatibility)</th>
<th>Name Origin (mostly obsolete)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ax</td>
<td>accumulate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ecx</td>
<td>%cx</td>
<td>counter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edx</td>
<td>%dx</td>
<td>data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx</td>
<td>base</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
<td>source index</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
<td>destination index</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td>stack pointer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td>base pointer</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Memory vs. Registers

- Addresses vs. Names
  - 0x7FFFFD024C3DC %rdi

- Big vs. Small
  - ~8 GiB
  - (16 x 8 B) = 128 B

- Slow vs. Fast
  - ~50-100 ns
  - sub-nanosecond timescale

- Dynamic vs. Static
  - Allocate as needed
  - Fixed hardware allocation
Three Basic Kinds of Instructions

1) Transfer data between memory and register
   • **Load** data from memory into register
     • \%reg = Mem[address]
   • **Store** register data into memory
     • Mem[address] = \%reg

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

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

- **Immediate**: Constant integer data
  - Examples: $0x400$, $-533$
  - Like C literal, but prefixed with `$`
  - Encoded with 1, 2, 4, or 8 bytes depending on the instruction

- **Register**: 1 of 16 integer registers
  - Examples: %rax, %r13
  - But %rsp reserved for special use
  - Others have special uses for particular instructions

- **Memory**: Consecutive bytes of memory at a computed address
  - Simplest example: (%rax)
  - Various other “address modes”
x86-64 Introduction

- Data transfer instruction (mov)
- Arithmetic operations
- Memory addressing modes
  - swap example
- Address computation instruction (lea)
Moving Data

- General form: `mov_ source, destination`
  - Missing letter (_) specifies size of operands
  - Note that due to backwards-compatible support for 8086 programs (16-bit machines!), “word” means 16 bits = 2 bytes in x86 instruction names
  - Lots of these in typical code

- `movb src, dst`
  - Move 1-byte “byte”
- `movw src, dst`
  - Move 2-byte “word”
- `movl src, dst`  
  - Move 4-byte “long word”
- `movq src, dst`  
  - Move 8-byte “quad word”
## Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src, Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td>Reg</td>
<td><code>movq $0x4, %rax</code></td>
<td><code>var_a = 0x4;</code></td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td><code>movq $-147, (%rax)</code></td>
<td><code>*p_a = -147;</code></td>
</tr>
<tr>
<td>Mem</td>
<td>Mem</td>
<td><code>movq %rax, %rdx</code></td>
<td><code>var_d = var_a;</code></td>
</tr>
<tr>
<td>Mem</td>
<td>Mem</td>
<td><code>movq %rax, (%rdx)</code></td>
<td><code>*p_d = var_a;</code></td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td><code>movq (%rax), %rdx</code></td>
<td><code>var_d = *p_a;</code></td>
</tr>
</tbody>
</table>

- **Cannot do memory-memory transfer with a single instruction**
  - How would you do it?
Some Arithmetic Operations

- Binary (two-operand) Instructions:
  
<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq src, dst</td>
<td>dst = dst + src</td>
</tr>
<tr>
<td>subq src, dst</td>
<td>dst = dst - src</td>
</tr>
<tr>
<td>imulq src, dst</td>
<td>dst = dst * src</td>
</tr>
<tr>
<td>sarq src, dst</td>
<td>dst = dst &gt;&gt; src</td>
</tr>
<tr>
<td>shrq src, dst</td>
<td>dst = dst &gt;&gt; src</td>
</tr>
<tr>
<td>shlq src, dst</td>
<td>dst = dst &lt;&lt; src</td>
</tr>
<tr>
<td>xorq src, dst</td>
<td>dst = dst ^ src</td>
</tr>
</tbody>
</table>

- Beware argument order!
- No distinction between signed and unsigned
  - Only arithmetic vs. logical shifts
- “r3 = r1 + r2”? (maximum of one memory operand)
Just to check in!

Which of the following would implement:

\[ \%rcx = \%rax + \%rbx \]

❤️  `addq %rax,%rbx,%rcx`
✔  `addq %rcx,%rax,%rbx`
💚  `movq %rax,%rcx; addq %rbx, %rcx`
💙  `movq (%rbx),%rcx ;addq (%rax),%rcx`
_cookies We’re lost...
Some Arithmetic Operations

- **Unary (one-operand) Instructions:**
  
<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>incq dst</code></td>
<td><code>dst = dst + 1</code></td>
<td>increment</td>
</tr>
<tr>
<td><code>decq dst</code></td>
<td><code>dst = dst – 1</code></td>
<td>decrement</td>
</tr>
<tr>
<td><code>negq dst</code></td>
<td><code>dst = –dst</code></td>
<td>negate</td>
</tr>
<tr>
<td><code>notq dst</code></td>
<td><code>dst = ~dst</code></td>
<td>bitwise complement</td>
</tr>
</tbody>
</table>

- See CSPP Section 3.5.5 for more instructions: `mulq, cqto, idivq, divq`
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>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>

```assembly
simple_arith:
    addq %rdi, %rsi
    imulq $3, %rsi
    movq %rsi, %rax
    ret
```

```assembly
y += x;
y *= 3;
long r = y;
return r;
```
Example of Basic Addressing Modes

```c
void swap(long *xp, long *yp) {
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

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

```c
void swap(long *xp, long *yp) {
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

**Registers**

- `%rdi`<br>- `%rsi`<br>- `%rax`<br>- `%rdx`<br>

**Memory**

**Register** | **Variable**
--- | ---
 `%rdi` | `xp`
 `%rsi` | `yp`
 `%rax` | `t0`
 `%rdx` | `t1`
Understanding `swap()`

### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Word Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td>456</td>
</tr>
</tbody>
</table>

### Code Snippet

```
swap:
movq (%rdi), %rax  # t0 = *xp
movq (%rsi), %rdx  # t1 = *yp
movq %rdx, (%rdi)  # *xp = t1
movq %rax, (%rsi)  # *yp = t0
ret
```
Understanding `swap()`

### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td>456</td>
</tr>
</tbody>
</table>

### swap:

```assembly
movq (%rdi), %rax  # t0 = *xp
movq (%rsi), %rdx  # t1 = *yp
movq %rdx, (%rdi)  # *xp = t1
movq %rax, (%rsi)  # *yp = t0
ret
```
# Understanding `swap()`

## Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td>0x120</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>0x100</td>
</tr>
<tr>
<td><code>%rax</code></td>
<td>123</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>456</td>
</tr>
</tbody>
</table>

## Memory

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>0x120</td>
</tr>
<tr>
<td>0x118</td>
<td>0x110</td>
</tr>
<tr>
<td>0x108</td>
<td>0x100</td>
</tr>
</tbody>
</table>

## Swap Function

```assembly
swap:
    movq (%rdi), %rax  # t0 = *xp
    movq (%rsi), %rdx  # t1 = *yp
    movq %rdx, (%rdi)  # *xp = t1
    movq %rax, (%rsi)  # *yp = t0
    ret
```
Understanding swap()

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Word Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x100</td>
</tr>
</tbody>
</table>

swap:

```assembly
movq (%rdi), %rax  # t0 = *xp
movq (%rsi), %rdx  # t1 = *yp
movq %rdx, (%rdi)  # *xp = t1
movq %rax, (%rsi)  # *yp = t0
ret
```
Understanding `swap()`

### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

### Memory

Word Address

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x100</td>
</tr>
</tbody>
</table>

### swap:

```assembly
movq (%rdi), %rax  # t0 = *xp
movq (%rsi), %rdx  # t1 = *yp
movq %rdx, (%rdi)  # *xp = t1
movq %rax, (%rsi)  # *yp = t0
ret
```
How are we feeling about swap()?
Memory Addressing Modes: Basic

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

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

- **General:**
  - \( D(Rb, Ri, S) \)  \( Mem[Reg[Rb]+Reg[Ri]*S+D] \)
    - \( Rb \): 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)
  - \( D(Rb, Ri) \)  \( Mem[Reg[Rb]+Reg[Ri]+D] \)  \( (S=1) \)
  - \( (Rb, Ri, S) \)  \( Mem[Reg[Rb]+Reg[Ri]*S] \)  \( (D=0) \)
  - \( (Rb, Ri) \)  \( Mem[Reg[Rb]+Reg[Ri]] \)  \( (S=1, D=0) \)
  - \( (,Ri,S) \)  \( Mem[Reg[Ri]*S] \)  \( (Rb=0, D=0) \)
How are we feeling about addressing modes?

We’ll do more on Friday!
Summary

- We’re learning about x86-64 here!
  - 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

- 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
Breakouts!
Floorplan Critique!
Giving and Receiving Critique

- Mandatory compliment sandwiches!
  - One thing you like
  - One thing you’d like to improve
  - One thing you enjoy or you’re excited about

- Our goal is to help each other improve!
  - We’re here to help you!
  - Be here to help each other!
Breakouts!
Floorplan Critique!