Machine-Level Programming I: Basics

15-213/18-213/14-513/15-513/18-613: Introduction to Computer Systems
5th Lecture, September 15, 2020
Assignment Announcements

- **Written Assignment 1 available on Canvas**
  - Due Wed, Sept. 16, 11:59pm ET
  - Peer grading due Wed, Sept. 23, 11:59pm ET
  - You will grade 3 submissions using the rubric posted on Canvas

- **Lab 1 available via Autolab**
  - Due Thurs, Sept. 17, 11:59pm ET
  - Read instructions carefully: writeup, bits.c, tests.c
    - Quirky software infrastructure
  - Based on lectures 2, 3, and 4 (CS:APP Chapter 2)

- **Recitations**
  - In person: you have been contacted with your recitation info
  - Online: use the zoom links provided on the Canvas homepage
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code
Intel x86 Processors

- Dominate laptop/desktop/server market
- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
- x86 is a Complex Instruction Set Computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
- Compare: Reduced Instruction Set Computer (RISC)
  - RISC: *very few* instructions, with *very few* modes for each
  - RISC can be quite fast (but Intel still wins on speed!)
  - Current RISC renaissance (e.g., ARM, RISCV), especially for low-power
# Intel x86 Evolution: Milestones

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
<th>MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
<td>5-10</td>
</tr>
<tr>
<td>▪ First 16-bit Intel processor. Basis for IBM PC &amp; DOS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ 1MB address space</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td>▪ First 32 bit Intel processor, referred to as IA32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>▪ Added “flat addressing”, capable of running Unix</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium 4E</td>
<td>2004</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td>▪ First 64-bit Intel x86 processor, referred to as x86-64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core 2</td>
<td>2006</td>
<td>291M</td>
<td>1060-3333</td>
</tr>
<tr>
<td>▪ First multi-core Intel processor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td>1600-4400</td>
</tr>
<tr>
<td>▪ Four cores (our shark machines)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Intel x86 Processors, cont.

■ Machine Evolution
- 386  1985  0.3M
- Pentium  1993  3.1M
- Pentium/MMX  1997  4.5M
- PentiumPro  1995  6.5M
- Pentium III  1999  8.2M
- Pentium 4  2000  42M
- Core 2 Duo  2006  291M
- Core i7  2008  731M
- Core i7 Skylake  2015  1.9B

■ Added Features
- Instructions to support multimedia operations
- Instructions to enable more efficient conditional operations
- Transition from 32 bits to 64 bits
- More cores
## Intel x86 Processors, cont.

### Past Generations

<table>
<thead>
<tr>
<th>Generation</th>
<th>Year</th>
<th>Process Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Pentium Pro</td>
<td>1995</td>
<td>600 nm</td>
</tr>
<tr>
<td>1st Pentium III</td>
<td>1999</td>
<td>250 nm</td>
</tr>
<tr>
<td>1st Pentium 4</td>
<td>2000</td>
<td>180 nm</td>
</tr>
<tr>
<td>1st Core 2 Duo</td>
<td>2006</td>
<td>65 nm</td>
</tr>
</tbody>
</table>

### Recent & Upcoming Generations

1. Nehalem         | 2008 | 45 nm             |
2. Sandy Bridge    | 2011 | 32 nm             |
3. Ivy Bridge      | 2012 | 22 nm             |
4. Haswell         | 2013 | 22 nm             |
5. Broadwell       | 2014 | 14 nm             |
6. Skylake         | 2015 | 14 nm             |
7. Kaby Lake       | 2016 | 14 nm             |
8. Coffee Lake     | 2017 | 14 nm             |
9. Cannon Lake     | 2018 | 10 nm             |
10. Ice Lake       | 2019 | 10 nm             |
11. Tiger Lake     | 2020?| 10 nm             |

*Process technology dimension = width of narrowest wires (10 nm ≈ 100 atoms wide)*
2018 State of the Art: Coffee Lake

- **Mobile Model: Core i7**
  - 2.2-3.2 GHz
  - 45 W

- **Desktop Model: Core i7**
  - Integrated graphics
  - 2.4-4.0 GHz
  - 35-95 W

- **Server Model: Xeon E**
  - Integrated graphics
  - Multi-socket enabled
  - 3.3-3.8 GHz
  - 80-95 W
x86 Clones: Advanced Micro Devices (AMD)

- Historically
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper
- Then
  - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64, their own extension to 64 bits
- Recent Years
  - Intel got its act together
    - 1995-2011: Lead semiconductor “fab” in world
    - 2018: #2 largest by $$ (#1 is Samsung)
    - 2019: reclaimed #1
  - AMD fell behind
    - Relies on external semiconductor manufacturer GlobalFoundaries
    - ca. 2019 CPUs (e.g., Ryzen) are competitive again
Intel’s 64-Bit History

- **2001: Intel Attempts Radical Shift from IA32 to IA64**
  - Totally different architecture (Itanium, AKA “Itanic”)
  - Executes IA32 code only as legacy
  - Performance disappointing

- **2003: AMD Steps in with Evolutionary Solution**
  - x86-64 (now called “AMD64”)

- **Intel Felt Obligated to Focus on IA64**
  - Hard to admit mistake or that AMD is better

- **2004: Intel Announces EM64T extension to IA32**
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64!

- **Virtually all modern x86 processors support x86-64**
  - But, lots of code still runs in 32-bit mode
Our Coverage

- **IA32**
  - The traditional x86
  - For 15/18-213: RIP, Summer 2015

- **x86-64**
  - The standard
  - `shark> gcc hello.c`
  - `shark> gcc -m64 hello.c`

- **Presentation**
  - Book covers x86-64
  - Web aside on IA32
  - We will only cover x86-64
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code
Levels of Abstraction

C program

```
#include <stdio.h>
int main()

i n = 10, t1 = 0, t2 = 1, nxt;
for (i = 1; i <= n; ++i){
    printf("%d, ", t1);
    nxt = t1 + t2;
    t1 = t2;
    t2 = nxt; }
return 0; }
```

Nice clean layers, but beware...

Assembly programmer

Computer Designer

Gates, clocks, circuit layout, ...
Definitions

- **Architecture**: (also ISA: instruction set architecture) The parts of a processor design that one needs to understand for writing correct machine/assembly code
  - Examples: instruction set specification, registers
  - **Machine Code**: The byte-level programs that a processor executes
  - **Assembly Code**: A text representation of machine code

- **Microarchitecture**: Implementation of the architecture
  - Examples: cache sizes and core frequency

- **Example ISAs:**
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: Used in almost all mobile phones
  - RISC V: New open-source ISA
Assembly/Machine Code View

Programmer-Visible State

- **PC**: Program counter
  - Address of next instruction
  - Called “RIP” (x86-64)

- **Register file**
  - Heavily used program data

- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures
Assembly Characteristics: Data Types

- "Integer" data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)

- Floating point data of 4, 8, or 10 bytes

- (SIMD vector data types of 8, 16, 32 or 64 bytes)

- Code: Byte sequences encoding series of instructions

- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
### x86-64 Integer Registers

<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>

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
- Not part of memory (or cache)
Some History: IA32 Registers

<table>
<thead>
<tr>
<th>General Purpose Registers</th>
<th>Origin (mostly obsolete)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax %ax %ah %al</td>
<td>accumulate</td>
</tr>
<tr>
<td>%ecx %cx %ch %cl</td>
<td>counter</td>
</tr>
<tr>
<td>%edx %dx %dh %dl</td>
<td>data</td>
</tr>
<tr>
<td>%ebx %bx %bh %bl</td>
<td>base</td>
</tr>
<tr>
<td>%esi %si</td>
<td>source index</td>
</tr>
<tr>
<td>%edi %di</td>
<td>destination index</td>
</tr>
<tr>
<td>%esp %sp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>%ebp %bp</td>
<td>base pointer</td>
</tr>
</tbody>
</table>

16-bit virtual registers (backwards compatibility)
Assembly Characteristics: Operations

- **Transfer data between memory and register**
  - Load data from memory into register
  - Store register data into memory

- **Perform arithmetic function on register or memory data**

- **Transfer control**
  - Unconditional jumps to/from procedures
  - Conditional branches
  - Indirect branches
Moving Data

Moving Data

mov q Source, Dest

Operand Types

- **Immediate**: Constant integer data
  - Example: $0x400, $-533
  - Like C constant, but prefixed with `$`
  - Encoded with 1, 2, or 4 bytes

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

- **Memory**: 8 consecutive bytes of memory at address given by register
  - Simplest example: (%rax)
  - Various other “addressing modes”

Warning: Intel docs use mov Dest, Source
### movq Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src,Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Reg</strong></td>
<td>movq $0x4,%rax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td><strong>Imm</strong></td>
<td><strong>Reg</strong></td>
<td>movq $-147,(%rax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>Mem</td>
<td>Mem</td>
<td>movq %rax,%rdx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td></td>
<td>Reg</td>
<td>movq %rax,(%rdx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq (%rax),%rdx</td>
<td>temp = *p;</td>
</tr>
</tbody>
</table>

*Cannot do memory-memory transfer with a single instruction*
Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

  ```
  movq (%rcx), %rax
  ```

- Displacement D(R) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

  ```
  movq 8(%rbp), %rdx
  ```
Example of Simple Addressing Modes

```c
void whatAmI(<type> a, <type> b)
{
    ???
}
```

whatAmI:
```
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```
Example of Simple 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;
}
```

Memory

Registers

- %rdi
- %rsi
- %rax
- %rdx

Memory

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>xp</td>
</tr>
<tr>
<td>%rsi</td>
<td>yp</td>
</tr>
<tr>
<td>%rax</td>
<td>t0</td>
</tr>
<tr>
<td>%rdx</td>
<td>t1</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()

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi 0x120</td>
<td>123 0x120</td>
</tr>
<tr>
<td>%rsi 0x100</td>
<td>0x118</td>
</tr>
<tr>
<td>%rax</td>
<td>0x110</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td>456 0x100</td>
</tr>
</tbody>
</table>

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

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

Registers

| %rdi | 0x120 |
| %rsi | 0x100 |
| %rax | 123   |
| %rdx |       |

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th></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>
Understanding Swap()

<table>
<thead>
<tr>
<th>Registers</th>
<th></th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
<td>0x118</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
<td>0x110</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
<td>0x108</td>
</tr>
</tbody>
</table>

Address

0x120
0x118
0x110
0x108
0x100

\[
\text{swap:} \\
\text{movq} \quad (%\text{rdi}), \quad %\text{rax} \quad \# \ t0 = *xp \\
\text{movq} \quad (%\text{rsi}), \quad %\text{rdx} \quad \# \ t1 = *yp \\
\text{movq} \quad %\text{rdx}, \quad (%\text{rdi}) \quad \# \ *xp = t1 \\
\text{movq} \quad %\text{rax}, \quad (%\text{rsi}) \quad \# \ *yp = t0 \\
\text{ret} \\
\]

Understanding Swap()

**Registers**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</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>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>456</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></td>
</tr>
</tbody>
</table>

**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>Value</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>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>456</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>123</td>
</tr>
</tbody>
</table>

swap:

```
movq (%rdi), %rax  # t0 = *xp
movq (%rsi), %rdx  # t1 = *yp
movq %rdx, (%rdi)  # *xp = t1
movq %rax, (%rsi)  # *yp = t0
ret
```
Simple Memory Addressing Modes

- **Normal** 
  \((R)\) \quad Mem[Reg[R]]
  - Register \(R\) specifies memory address
  - Aha! Pointer dereferencing in C

  \[
  \text{movq} \ (\%rcx),\%rax
  \]

- **Displacement** 
  \(D(R)\) \quad Mem[Reg[R]+D]
  - Register \(R\) specifies start of memory region
  - Constant displacement \(D\) specifies offset

  \[
  \text{movq} \ 8(\%rbp),\%rdx
  \]
Complete Memory Addressing Modes

Most General Form

\[ D(R_b, R_i, S) \quad \text{Mem}[\text{Reg}[R_b]+S*\text{Reg}[R_i]+D] \]

- **D**: Constant “displacement” 1, 2, or 4 bytes
- **R_b**: Base register: Any of 16 integer registers
- **R_i**: Index register: Any, except for %rsp
- **S**: Scale: 1, 2, 4, or 8 (*why these numbers?*)

Special Cases

- \((R_b, R_i)\) \quad \text{Mem}[\text{Reg}[R_b]+\text{Reg}[R_i]]
- \(D(R_b, R_i)\) \quad \text{Mem}[\text{Reg}[R_b]+\text{Reg}[R_i]+D]
- \((R_b, R_i, S)\) \quad \text{Mem}[\text{Reg}[R_b]+S*\text{Reg}[R_i]]
# Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0x8(%rdx))</td>
<td>0xf000</td>
<td>0xf008</td>
</tr>
<tr>
<td>(0xf000 + 0x100)</td>
<td>0xf100</td>
<td></td>
</tr>
<tr>
<td>(0xf000 + 4 \times 0x100)</td>
<td>0xf400</td>
<td></td>
</tr>
<tr>
<td>(2 \times 0xf000 + 0x80)</td>
<td>0x1e080</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D(Rb,Ri,S)</th>
<th>Mem[Reg[Rb]+S*Reg[Ri]+ D]</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Constant “displacement” 1, 2, or 4 bytes</td>
</tr>
<tr>
<td>Rb</td>
<td>Base register: Any of 16 integer registers</td>
</tr>
<tr>
<td>Ri</td>
<td>Index register: Any, except for %rsp</td>
</tr>
<tr>
<td>S</td>
<td>Scale: 1, 2, 4, or 8 (why these numbers?)</td>
</tr>
</tbody>
</table>
## Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%rdx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code
Address Computation Instruction

- **leaq $Src$, $Dst$**
  - $Src$ is address mode expression
  - Set $Dst$ to address denoted by expression

- **Uses**
  - Computing addresses without a memory reference
    - E.g., translation of $p = &x[i]$;
  - Computing arithmetic expressions of the form $x + k*y$
    - $k = 1, 2, 4, \text{ or } 8$

- **Example**

  ```c
  long m12(long x)
  {
      return x*12;
  }
  ```

  Converted to ASM by compiler:

  ```asm
  leaq (%rdi,%rdi,2), %rax  # t = x+2*x
  salq $2, %rax             # return t<<2
  ```
Some Arithmetic Operations

- **Two Operand Instructions:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addq</code></td>
<td><code>Src, Dest</code></td>
</tr>
<tr>
<td><code>subq</code></td>
<td><code>Src, Dest</code></td>
</tr>
<tr>
<td><code>imulq</code></td>
<td><code>Src, Dest</code></td>
</tr>
<tr>
<td><code>shlq</code></td>
<td><code>Src, Dest</code></td>
</tr>
<tr>
<td><code>sarq</code></td>
<td><code>Src, Dest</code></td>
</tr>
<tr>
<td><code>shrq</code></td>
<td><code>Src, Dest</code></td>
</tr>
<tr>
<td><code>xorq</code></td>
<td><code>Src, Dest</code></td>
</tr>
<tr>
<td><code>andq</code></td>
<td><code>Src, Dest</code></td>
</tr>
<tr>
<td><code>orq</code></td>
<td><code>Src, Dest</code></td>
</tr>
</tbody>
</table>

- Watch out for argument order! `Src, Dest` (Warning: Intel docs use “op `Dest, Src`”)
- No distinction between signed and unsigned int (why?)
Quiz Time!

Check out:

https://canvas.cmu.edu/courses/17808
Some Arithmetic Operations

- **One Operand Instructions**

  - `incq` \( \text{Dest} \) \( \text{Dest} = \text{Dest} + 1 \)
  - `decq` \( \text{Dest} \) \( \text{Dest} = \text{Dest} - 1 \)
  - `negq` \( \text{Dest} \) \( \text{Dest} = -\text{Dest} \)
  - `notq` \( \text{Dest} \) \( \text{Dest} = \sim\text{Dest} \)

- **See book for more instructions**

  - Depending how you count, there are 2,034 total x86 instructions
  - (If you count all addr modes, op widths, flags, it’s actually 3,683)
Arithmetic Expression Example

```c
long arith (long x, long y, long z)
{
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

Arith:
- `leaq`: address computation
- `salq`: shift
- `imulq`: multiplication
  - Curious: only used once...

Interesting Instructions
Understanding Arithmetic Expression

Example

```c
long arith
(long x, long y, long z)
{
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

### arith:

```c
leaq (%rdi,%rsi), %rax  # t1
addq %rdx, %rax          # t2
leaq (%rsi,%rsi,2), %rdx # t4
salq $4, %rdx
leaq 4(%rdi,%rdx), %rcx # t5
imulq %rcx, %rax         # rval
ret
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z, t4</td>
</tr>
<tr>
<td>%rax</td>
<td>t1, t2, rval</td>
</tr>
<tr>
<td>%rcx</td>
<td>t5</td>
</tr>
</tbody>
</table>

Register Use(s)
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code
Turning C into Object Code

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Use basic optimizations (`-Og`) [New to recent versions of GCC]
  - Put resulting binary in file `p`

```
C program (p1.c  p2.c)

Compiler (gcc  -Og  -S)

Asm program (p1.s  p2.s)

Assembler (gcc or as)

Object program (p1.o  p2.o)

Linker (gcc or  ld)

Executable program (p)

Static libraries (.a)
```


44
## Compiling Into Assembly

### C Code (sum.c)

```c
long plus(long x, long y);

void sumstore(long x, long y, long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

### Generated x86-64 Assembly

```
sumstore:
    pushq  %rbx
    movq  %rdx, %rbx
    call  plus
    movq  %rax, (%rbx)
    popq  %rbx
    ret
```

### Obtain (on shark machine) with command

```
gcc -Og -S sum.c
```

### Produces file sum.s

**Warning**: Will get very different results on non-Shark machines (Andrew Linux, Mac OS-X, ...) due to different versions of gcc and different compiler settings.
What it really looks like

```assembly
.globl sumstore
.type sumstore, @function

sumstore:
.LFB35:
    .cfi_startproc
    pushq %rbx
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    movq %rdx, %rbx
    call plus
    movq %rax, (%rbx)
    popq %rbx
    .cfi_def_cfa_offset 8
    ret
    .cfi_endproc

.LFE35:
    .size sumstore, .-_sumstore
```
What it really looks like

```
.globl sumstore
.type sumstore, @function
sumstore:
.LFB35:
  .cfi_startproc
  pushq %rbx
  .cfi_def_cfa_offset 16
  .cfi_offset 3, -16
  movq %rdx, %rbx
  call plus
  movq %rax, (%rbx)
  popq %rbx
  .cfi_def_cfa_offset 8
  ret
  .cfi_endproc
.LFE35:
  .size sumstore, .-sumstore
```

Things that look weird and are preceded by a ‘.’ are generally directives.
Assembly Characteristics: Data Types

- "Integer" data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)

- Floating point data of 4, 8, or 10 bytes

- (SIMD vector data types of 8, 16, 32 or 64 bytes)

- Code: Byte sequences encoding series of instructions

- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
Assembly Characteristics: Operations

- **Transfer data between memory and register**
  - Load data from memory into register
  - Store register data into memory

- **Perform arithmetic function on register or memory data**

- **Transfer control**
  - Unconditional jumps to/from procedures
  - Conditional branches
Object Code

Code for sumstore

0x0400595:
  0x53
  0x48
  0x89
  0xd3
  0xe8
  0xf2
  0xff
  0xff
  0xff
  0xff
  0x48
  0x89
  0x03
  0xc3

- Total of 14 bytes
- Each instruction 1, 3, or 5 bytes
- Starts at address 0x0400595

- Assembler
  - Translates .s into .o
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files

- Linker
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for malloc, printf
  - Some libraries are dynamically linked
    - Linking occurs when program begins execution
Machine Instruction Example

- **C Code**
  - \texttt{*dest = t;}

- **Assembly**
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:
    - \(t\): Register \texttt{%rax}
    - \(\text{dest}\): Register \texttt{%rbx}
    - \(*\text{dest}\): Memory \(M[\%rbx]\)

- **Object Code**
  - 3-byte instruction
  - Stored at address \texttt{0x40059e}
Disassembling Object Code

Disassembled

```
0000000000400595 <sumstore>:
400595:  53    push %rbx
400596:  48 89 d3 mov %rdx,%rbx
400599:  e8 f2 ff ff ff callq 400590 <plus>
40059e:  48 89 03 mov %rax,(%rbx)
4005a1:  5b    pop %rbx
4005a2:  c3    retq
```

- **Disassembler**

  `objdump -d sum`

  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either `a.out` (complete executable) or `.o` file
**Alternate Disassembly**

**Disassembled**

<table>
<thead>
<tr>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000000400595 &lt;+0&gt;: push %rbx</td>
<td>Push %rbx register</td>
</tr>
<tr>
<td>0x00000000000400596 &lt;+1&gt;: mov %rdx,%rbx</td>
<td>Move %rdx to %rbx</td>
</tr>
<tr>
<td>0x00000000000400599 &lt;+4&gt;: callq 0x400590 &lt;plus&gt;</td>
<td>Call function %plus</td>
</tr>
<tr>
<td>0x0000000000040059e &lt;+9&gt;: mov %rax,(%rbx)</td>
<td>Move %rax to memory</td>
</tr>
<tr>
<td>0x000000000004005a1 &lt;+12&gt;: pop %rbx</td>
<td>Pop %rbx register</td>
</tr>
<tr>
<td>0x000000000004005a2 &lt;+13&gt;: retq</td>
<td>Return from function</td>
</tr>
</tbody>
</table>

- **Within gdb Debugger**
  - Disassemble procedure
    - `gdb sum`
    - `disassemble sumstore`
Alternate Disassembly

Object Code

0x0400595:
0x53
0x48
0x89
0xd3
0xe8
0xff
0xff
0xff
0x48
0x03
0x5b
0xc3

Disassembled

Dump of assembler code for function sumstore:

0x00000000000400595 <+0>: push %rbx
0x00000000000400596 <+1>: mov %rdx,%rbx
0x00000000000400599 <+4>: callq 0x400590 <plus>
0x0000000000040059e <+9>: mov %rax,(%rbx)
0x000000000004005a1 <+12>: pop %rbx
0x000000000004005a2 <+13>: retq

Within gdb Debugger

- Disassemble procedure
  gdb sum
disassemble sumstore
- Examine the 14 bytes starting at sumstore
  x/14xb sumstore
What Can be Disassembled?

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

% objdump -d WINWORD.EXE

WINWORD.EXE: file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:
30001001:
30001003:
30001005:
3000100a:

Reverse engineering forbidden by Microsoft End User License Agreement
Machine Programming I: Summary

- History of Intel processors and architectures
  - Evolutionary design leads to many quirks and artifacts

- C, assembly, machine code
  - New forms of visible state: program counter, registers, ...
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences

- Assembly Basics: Registers, operands, move
  - The x86-64 move instructions cover wide range of data movement forms

- Arithmetic
  - C compiler will figure out different instruction combinations to carry out computation