Machine-Level Programming I: Basics

15-213/18-213/15-213: Introduction to Computer Systems
5th Lecture, September 12, 2017

Today’s Instructor:
Phil Gibbons
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

- Complex instruction set computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!
    - In terms of speed. Less so 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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></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

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>Model</th>
<th>Year</th>
<th>Process Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>1&lt;sup&gt;st&lt;/sup&gt; Pentium Pro</td>
<td>1995</td>
<td>600 nm</td>
</tr>
<tr>
<td>1&lt;sup&gt;st&lt;/sup&gt; Pentium III</td>
<td>1999</td>
<td>250 nm</td>
</tr>
<tr>
<td>1&lt;sup&gt;st&lt;/sup&gt; Pentium 4</td>
<td>2000</td>
<td>180 nm</td>
</tr>
<tr>
<td>1&lt;sup&gt;st&lt;/sup&gt; 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. Cannonlake     | 2018 | 10 nm              

**Process technology dimension**

= width of narrowest wires

(10 nm ≈ 100 atoms wide)
2017 State of the Art: Skylake

■ Mobile Model: Core i7
  ▪ 2.6-2.9 GHz
  ▪ 45 W

■ Desktop Model: Core i7
  ▪ Integrated graphics
  ▪ 2.8-4.0 GHz
  ▪ 35-91 W

■ Server Model: Xeon
  ▪ Integrated graphics
  ▪ Multi-socket enabled
  ▪ 2-3.7 GHz
  ▪ 25-80 W

Figure 1: Architecture components layout for an Intel® Core™ i7 processor 6700K for desktop systems. This SoC contains 4 CPU cores, outlined in blue dashed boxes. Outlined in the red dashed box, is an Intel® HD Graphics 530. It is a one-slice instantiation of Intel processor graphics gen9 architecture.
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
  - Leads the world in semiconductor technology
- AMD has fallen behind
  - Relies on external semiconductor manufacturer
Intel’s 64-Bit History

- 2001: Intel Attempts Radical Shift from IA32 to IA64
  - Totally different architecture (Itanium)
  - 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!

- All but low-end 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 programmer

Assembly programmer

Computer Designer

C code

Nice clean layers, but beware...

Caches, clock freq, layout, ...

Of course, you know that: It’s why you are taking this course.
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><code>%eax</code> %<code>ax</code> %<code>ah</code> %<code>al</code></td>
<td>accumulate</td>
</tr>
<tr>
<td>%<code>ecx</code> %<code>cx</code> %<code>ch</code> %<code>cl</code></td>
<td>counter</td>
</tr>
<tr>
<td>%<code>edx</code> %<code>dx</code> %<code>dh</code> %<code>dl</code></td>
<td>data</td>
</tr>
<tr>
<td>%<code>ebx</code> %<code>bx</code> %<code>bh</code> %<code>bl</code></td>
<td>base</td>
</tr>
<tr>
<td>%<code>esi</code> %<code>si</code></td>
<td>source index</td>
</tr>
<tr>
<td>%<code>edi</code> %<code>di</code></td>
<td>destination index</td>
</tr>
<tr>
<td>%<code>esp</code> %<code>sp</code></td>
<td>stack pointer</td>
</tr>
<tr>
<td>%<code>ebp</code> %<code>bp</code></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**
  - `movq 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>Imm</td>
<td>Reg</td>
<td>movq $0x4, %rax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq $-147, (%rax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>Reg</td>
<td>Mem</td>
<td>movq %rax, %rdx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq (%rax), %rdx</td>
<td>*p = temp;</td>
</tr>
</tbody>
</table>

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

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

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

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

\[
\text{movq} \ 8(\%\text{rbp}),\%\text{rdx}
\]
Example of Simple Addressing Modes

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

whatAmI:

```c
    movq (%rdi), %rax
    movq (%rsi), %rdx
    movq %rdx, (%rdi)
    movq %rax, (%rsi)
    ret
```

%rdi %rsi
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;
}
```

### Registers

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

### Memory

### Register | Value
---|---
%rdi | xp
%rsi | yp
%rax | t0
%rdx | t1

### 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></td>
</tr>
<tr>
<td>%rdx</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`

Memory

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

**Memory**

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

Address

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

```assembly
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**  \( (R) \)  \( \text{Mem}[\text{Reg}[R]] \)
  - Register \( R \) specifies memory address
  - Aha! Pointer dereferencing in C

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

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

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

**Most General Form**

\[ D(R_b, R_i, S) \rightarrow Mem[Reg[R_b] + S*Reg[R_i] + D] \]

- **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**

\[
\begin{align*}
D(R_b, R_i) & \rightarrow Mem[Reg[R_b] + Reg[R_i]] \\
D(R_b, R_i) & \rightarrow Mem[Reg[R_b] + Reg[R_i] + D] \\
(R_b, R_i, S) & \rightarrow Mem[Reg[R_b] + S*Reg[R_i]]
\end{align*}
\]
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></td>
<td></td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80(%rdx,2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+D]

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

<table>
<thead>
<tr>
<th>%rdx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rcx</td>
<td>0x0100</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, or 8`

- **Example**

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

Converted to ASM by compiler:

```
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>addq</td>
<td>( Src, Dest )</td>
</tr>
<tr>
<td>subq</td>
<td>( Src, Dest )</td>
</tr>
<tr>
<td>imulq</td>
<td>( Src, Dest )</td>
</tr>
<tr>
<td>salq</td>
<td>( Src, Dest )</td>
</tr>
<tr>
<td>sarq</td>
<td>( Src, Dest )</td>
</tr>
<tr>
<td>shrq</td>
<td>( Src, Dest )</td>
</tr>
<tr>
<td>xorq</td>
<td>( Src, Dest )</td>
</tr>
<tr>
<td>andq</td>
<td>( Src, Dest )</td>
</tr>
<tr>
<td>orq</td>
<td>( Src, Dest )</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/1221
Some Arithmetic Operations

- **One Operand Instructions**
  
  - `incq`  
    - `Dest`  
    - `Dest = Dest + 1`
  
  - `decq`  
    - `Dest`
    - `Dest = Dest − 1`
  
  - `negq`  
    - `Dest`
    - `Dest = − Dest`
  
  - `notq`  
    - `Dest`
    - `Dest = ~Dest`

- **See book for more instructions**
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;
}
```

### Interesting Instructions

- **leaq**: address computation
- **salq**: shift
- **imulq**: multiplication
  - But, only used once
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;
}
```

```assembly
arith:
    leaq (%rdi,%rsi), %rax      # t1
    addq %rdx, %rax             # t2
    leaq (%rsi,%rsi,2), %rdx
    salq $4, %rdx               # t4
    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>
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)
\textit{text} \downarrow

Asm program (p1.s p2.s)         Assembler (gcc or as)
\textit{text} \downarrow

Object program (p1.o p2.o)  Linker (gcc or ld)
\textit{binary} \downarrow

Executable program (p)  Static libraries (.a)
\textit{binary}
```

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition
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

```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

.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

```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
```

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  
  0xff  
  0xff  
  0xff  
  0x48  
  0x89  
  0x03  
  0x5b  
  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**

*dest = t;

**Assembly**

- Move 8-byte value to memory
  - Quad words in x86-64 parlance
- Operands:
  - `t`: Register `%rax`
  - `dest`: Register `%rbx`
  - `*dest`: Memory `M[%rbx]`

**Object Code**

- 3-byte instruction
- Stored at address `0x40059e`
Disassembling Object Code

Disassembled

0000000000400595 <sumstore>:  
000595:  53               push %rbx  
000596:  48 89 d3          mov %rdx,%rbx  
000599:  e8 f2 ff ff ff     callq 400590 <plus>  
00059e:  48 89 03          mov %rax,(%rbx)  
0005a1:  5b               pop %rbx  
0005a2:  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

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
disable sumstore
Alternate Disassembly

Disassembled

Dump of assembler code for function sumstore:
0x000000000000400595 <+0>: push %rbx
0x000000000000400596 <+1>: mov %rdx,%rbx
0x000000000000400599 <+4>: callq 0x400590 <plus>
0x00000000000040059e <+9>: mov %rax,(%rbx)
0x0000000000004005a1 <+12>: pop %rbx
0x0000000000004005a2 <+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
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