CSC 252: Computer Organization
Spring 2018: Lecture 5

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Action Items:
• Assignment 1 is due tomorrow, midnight
• Assignment 2 is out
• Trivia 2 is due on the coming Tues, noon
Announcement
Announcement

- Programming Assignment 2 is out
  - Due on **Feb 16, 11:59 PM**
  - Trivia due Feb 6, noon
  - You have 3 slip days

![Calendar showing due dates](image)
Announcement

• Programming Assignment 2 is out
  • Due on **Feb 16, 11:59 PM**
  • Trivia due Feb 6, noon
  • You have 3 slip days
• Read the instructions before getting started!!!
  • You get 1/4 point off for every wrong answer
  • Maxed out at 10
Floating Point Review

\[ v = (-1)^s \times 1.frac \times 2^E \]
Floating Point Review

\[ v = (-1)^s \times 1.frac \times 2^E \]

- Denormalized
  - \( E = (exp + 1) - \text{bias} \)
  - \( M = 0.frac \)

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>00</td>
<td>0.00 x 2^{-2}</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
<td>11</td>
<td>0.11 x 2^{-2}</td>
<td>3/16</td>
</tr>
</tbody>
</table>
Floating Point Review

$$v = (-1)^s \times 1.frac \times 2^E$$

- **Denormalized**
  - $E = (\text{exp} + 1) - \text{bias}$
  - $M = 0.frac$

- **Normalized**
  - $E = \text{exp} - \text{bias}$
  - $M = 1.frac$

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>00</td>
<td>$0.00 \times 2^{-2}$</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
<td>11</td>
<td>$0.11 \times 2^{-2}$</td>
<td>3/16</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td>00</td>
<td>$1.00 \times 2^{-2}$</td>
<td>1/4</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td>11</td>
<td>$1.11 \times 2^{-2}$</td>
<td>7/16</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>00</td>
<td>$1.00 \times 2^{-1}$</td>
<td>1/2</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>11</td>
<td>$1.11 \times 2^{-1}$</td>
<td>7/8</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>00</td>
<td>$1.00 \times 2^0$</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>11</td>
<td>$1.11 \times 2^0$</td>
<td>1 3/4</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>00</td>
<td>$1.00 \times 2^1$</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>11</td>
<td>$1.11 \times 2^1$</td>
<td>3 1/2</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>00</td>
<td>$1.00 \times 2^2$</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>11</td>
<td>$1.11 \times 2^2$</td>
<td>7</td>
</tr>
</tbody>
</table>
Floating Point Review

\[ v = (-1)^s \times 1 \cdot \text{frac} \times 2^E \]

- **Denormalized**
  - \( E = (\text{exp} + 1) - \text{bias} \)
  - \( M = 0.\text{frac} \)

- **Normalized**
  - \( E = \text{exp} - \text{bias} \)
  - \( M = 1.\text{frac} \)

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>00</td>
<td>0.00 x 2^{-2}</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
<td>11</td>
<td>0.11 x 2^{-2}</td>
<td>3/16</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td>00</td>
<td>1.00 x 2^{-2}</td>
<td>1/4</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td>11</td>
<td>1.11 x 2^{-2}</td>
<td>7/16</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>00</td>
<td>1.00 x 2^{-1}</td>
<td>1/2</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>11</td>
<td>1.11 x 2^{-1}</td>
<td>7/8</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>00</td>
<td>1.00 x 2^{0}</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>11</td>
<td>1.11 x 2^{0}</td>
<td>1 3/4</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>00</td>
<td>1.00 x 2^{1}</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>11</td>
<td>1.11 x 2^{1}</td>
<td>3 1/2</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>00</td>
<td>1.00 x 2^{2}</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>11</td>
<td>1.11 x 2^{2}</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>00</td>
<td>infinite</td>
<td>infinite</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>11</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
## Floating Point Review

### Denormalized

<table>
<thead>
<tr>
<th>s exp frac</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000 00</td>
<td>0.00 x 2^-2</td>
<td>0</td>
</tr>
<tr>
<td>0 000 11</td>
<td>0.11 x 2^-2</td>
<td>3/16</td>
</tr>
<tr>
<td>0 001 00</td>
<td>1.00 x 2^-2</td>
<td>1/4</td>
</tr>
<tr>
<td>0 001 11</td>
<td>1.11 x 2^-2</td>
<td>7/16</td>
</tr>
<tr>
<td>0 010 00</td>
<td>1.00 x 2^-1</td>
<td>1/2</td>
</tr>
<tr>
<td>0 010 11</td>
<td>1.11 x 2^-1</td>
<td>7/8</td>
</tr>
<tr>
<td>0 100 00</td>
<td>1.00 x 2^0</td>
<td>1</td>
</tr>
<tr>
<td>0 100 11</td>
<td>1.11 x 2^0</td>
<td>1 3/4</td>
</tr>
<tr>
<td>0 101 00</td>
<td>1.00 x 2^1</td>
<td>2</td>
</tr>
<tr>
<td>0 101 11</td>
<td>1.11 x 2^1</td>
<td>3 1/2</td>
</tr>
<tr>
<td>0 110 00</td>
<td>1.00 x 2^2</td>
<td>4</td>
</tr>
<tr>
<td>0 110 11</td>
<td>1.11 x 2^2</td>
<td>7</td>
</tr>
<tr>
<td>0 111 00</td>
<td>infinite</td>
<td>infinite</td>
</tr>
<tr>
<td>0 111 11</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

### Normalized

### Special Value
If you do an integer increment on a positive FP number, you get the next larger FP number.
Floating Point Review

- If you do an integer increment on a positive FP number, you get the next larger FP number.
- Bit patterns representing non-negative numbers are ordered the same way as integers, so could use regular integer comparison.

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>00</td>
<td>0.00 x 2^-2</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
<td>11</td>
<td>0.11 x 2^-2</td>
<td>3/16</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td>00</td>
<td>1.00 x 2^-2</td>
<td>1/4</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td>11</td>
<td>1.11 x 2^-2</td>
<td>7/16</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>00</td>
<td>1.00 x 2^-1</td>
<td>1/2</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>11</td>
<td>1.11 x 2^-1</td>
<td>7/8</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>00</td>
<td>1.00 x 2^0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>11</td>
<td>1.11 x 2^0</td>
<td>1.3/4</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>00</td>
<td>1.00 x 2^1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>11</td>
<td>1.11 x 2^1</td>
<td>3.1/2</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>00</td>
<td>1.00 x 2^2</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>11</td>
<td>1.11 x 2^2</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>00</td>
<td>infinite</td>
<td>infinite</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>11</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
Floating Point Review

- If you do an integer increment on a positive FP number, you get the next larger FP number.
- Bit patterns representing non-negative numbers are ordered the same way as integers, so could use regular integer comparison.
- You don’t get this property if:
  - $exp$ is interpreted as signed
  - $exp$ and $frac$ are swapped

<table>
<thead>
<tr>
<th></th>
<th>exp</th>
<th>frac</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>00</td>
<td>0.00 x $2^{-2}$</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
<td>11</td>
<td>0.11 x $2^{-2}$</td>
<td>3/16</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td>00</td>
<td>1.00 x $2^{-2}$</td>
<td>1/4</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td>11</td>
<td>1.11 x $2^{-2}$</td>
<td>7/16</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>00</td>
<td>1.00 x $2^{-1}$</td>
<td>1/2</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>11</td>
<td>1.11 x $2^{-1}$</td>
<td>7/8</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>00</td>
<td>1.00 x $2^{0}$</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>11</td>
<td>1.11 x $2^{0}$</td>
<td>3/4</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>00</td>
<td>1.00 x $2^{1}$</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>11</td>
<td>1.11 x $2^{1}$</td>
<td>3 1/2</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>00</td>
<td>1.00 x $2^{2}$</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>11</td>
<td>1.11 x $2^{2}$</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>00</td>
<td>infinite</td>
<td>infinite</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>11</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
# Floating Point in C

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Bits</th>
<th>Max Value</th>
<th>Max Value (Decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
<td>$2^7 - 1$</td>
<td>127</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>$2^{15} - 1$</td>
<td>32767</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>$2^{31} - 1$</td>
<td>2147483647</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>$2^{31} - 1$</td>
<td>$\approx 9.2 \times 10^{18}$</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>$(2 - 2^{-23}) \times 2^{127}$</td>
<td>$\approx 3.4 \times 10^{38}$</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>$(2 - 2^{-52}) \times 2^{1023}$</td>
<td>$\approx 1.8 \times 10^{308}$</td>
</tr>
</tbody>
</table>

Fixed point (implicit binary point)

SP floating point

DP floating point
Floating Point in C

### Fixed point (implicit binary point)

- **SP floating point**
- **DP floating point**

### C Data Type

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Bits</th>
<th>Max Value (Decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
<td>$2^7 - 1$</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>$2^{15} - 1$</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>$2^{31} - 1$</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>$2^{31} - 1$</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>$(2 - 2^{-23}) \times 2^{127}$</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>$(2 - 2^{-52}) \times 2^{1023}$</td>
</tr>
</tbody>
</table>

- To represent $2^{31}$ in fixed-point, you need at least 32 bits
  - Because fixed-point is a *weighted positional* representation

- In floating-point, we directly encode the exponent
  - Floating point is based on scientific notation
  - Encoding 31 only needs 7 bits in the exp field
Floating Point Conversions/Casting in C

- **double/float → int**
  - Truncates fractional part
  - Like rounding toward zero
  - Not defined when out of range or NaN: Generally sets to TMin
Floating Point Conversions/Casting in C

- **double/float → int**
  - Truncates fractional part
  - Like rounding toward zero
  - Not defined when out of range or NaN: Generally sets to Tmin

- **int → float**
  - Can’t guarantee exact casting. Will round according to rounding mode

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8-bit</td>
<td>23-bit</td>
</tr>
</tbody>
</table>
Floating Point Conversions/Casting in C

• **double/float → int**
  - Truncates fractional part
  - Like rounding toward zero
  - Not defined when out of range or NaN: Generally sets to TMin

• **int → float**
  - Can’t guarantee exact casting. Will round according to rounding mode

• **int → double**
  - Exact conversion
So far in 252...

C Program

int, float
if, else
+, -, >>
So far in 252...

C Program

Machine Code

int, float
if, else
+, -, >>

00001111
01010101
11110000
So far in 252...

C Program

Machine Code

Processor

int, float
if, else
+, -, >>

00001111
01010101
11110000

Ripple-carry Adder
So far in 252...

C Program

Machine Code

Processor

Transistor

int, float
if, else
+, -, >>

00001111
01010101
11110000

Ripple-carry Adder

NAND Gate
NOR Gate
So far in 252...

C Program

Compiler

Assembly Program

Machine Code

Processor

Transistor

int, float
if, else
+, -, >>
ret, call
movq, addq
jmp, jne

00001111
01010101
11110000

Ripple-carry Adder

NAND Gate
NOR Gate
So far in 252…

C Program

Compiler

Assembly Program

Semantically Equivalent

int, float
if, else
+, -, >>

ret, call
movq, addq
jmp, jne

Machine Code

00001111
01010101
11110000

Processor

Ripple-carry Adder

Transistor

NAND Gate
NOR Gate
So far in 252...

C Program

Compilation

Assembly Program

Semantically Equivalent

Machine Code

Processor

Transistor

int, float

if, else

+, -, >>

ret, call

movq, addq

jmp, jne

00001111
01010101
11110000

Ripple-carry
Adder

NAND Gate

NOR Gate
So far in 252…

C Program

Compiler

Assembly Program

Assembler

Machine Code

Processor

Transistor

Semantically Equivalent

int, float
if, else
+,-,>>
ret, call
movq, addq
jmp, jne

00001111
01010101
11110000

Ripple-carry Adder

NAND Gate
NOR Gate
So far in 252…

High-Level Language

C Program

Assembly Program

Machine Code

Processor

Transistor
So far in 252…

High-Level Language

Instruction Set Architecture (ISA)

- ISA: Assembly programmers’ view of a computer
  - Provide all info for someone wants to write assembly/machine code
  - “Contract” between assembly/machine code and processor
So far in 252...

- **High-Level Language**
- **Instruction Set Architecture (ISA)**

C Program ➔ Assembly Program ➔ Machine Code ➔ Processor ➔ Transistor

- **ISA**: Assembly programmers’ view of a computer
  - Provide all info for someone wants to write assembly/machine code
  - “Contract” between assembly/machine code and processor

- Processors execute machine code (binary). Assembly program is merely a text representation of machine code
So far in 252...

High-Level Language

Instruction Set Architecture (ISA)

Microarchitecture

Circuit

- **ISA**: Assembly programmers’ view of a computer
  - Provide all info for someone wants to write assembly/machine code
  - “Contract” between assembly/machine code and processor

- Processors execute machine code (binary). Assembly program is merely a text representation of machine code

- **Microarchitecture**: Hardware implementation of the ISA (with the help of circuit technologies)
This Module (4 Lectures)

- High-Level Language
  - C Program
  - Assembly Program
  - Machine Code
  - Processor
  - Transistor

- Instruction Set Architecture (ISA)
  - Assembly Programming
    - Explain how various C constructs are implemented in assembly code
    - Effectively translating from C to assembly program manually
    - Helps us understand how compilers work
    - Helps us understand how assemblers work

- Microarchitecture
  - Microarchitecture is the topic of the next module
Today: Assembly Programming I: Basics

• Different ISAs and history behind them
• C, assembly, machine code
• Move operations (and addressing modes)
Instruction Set Architecture
Instruction Set Architecture

• There used to be many ISAs
  • x86, ARM, Power/PowerPC, Sparc, MIPS, IA64, z
  • Very consolidated today: ARM for mobile, x86 for others
Instruction Set Architecture

• There used to be many ISAs
  • x86, ARM, Power/PowerPC, Sparc, MIPS, IA64, z
  • Very consolidated today: ARM for mobile, x86 for others

• There are even more microarchitectures
  • Apple/Samsung/Qualcomm have their own microarchitecture (implementation) of the ARM ISA
  • Intel and AMD have different microarchitectures for x86
Instruction Set Architecture

• There used to be many ISAs
  • x86, ARM, Power/PowerPC, Sparc, MIPS, IA64, z
  • Very consolidated today: ARM for mobile, x86 for others

• There are even more microarchitectures
  • Apple/Samsung/Qualcomm have their own microarchitecture (implementation) of the ARM ISA
  • Intel and AMD have different microarchitectures for x86

• ISA is lucrative business: ARM’s Business Model
  • Patent the ISA, and then license the ISA
  • Every implementer pays a royalty to ARM
  • Apple/Samsung pays ARM whenever they sell a smartphone

Intel x86 ISA

• Dominate laptop/desktop/cloud market
Intel x86 ISA

- Dominate laptop/desktop/cloud market
Intel x86 ISA

- Dominate laptop/desktop/cloud market
Intel x86 ISA Evolution (Milestones)

- Evolutionary design: Added more features as time goes on
Intel x86 ISA Evolution (Milestones)

- Evolutionary design: Added more features as time goes on

<table>
<thead>
<tr>
<th>Date</th>
<th>Feature</th>
<th>Notable Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1974</td>
<td>8-bit ISA</td>
<td>8080</td>
</tr>
<tr>
<td>1978</td>
<td>16-bit ISA (Basis for IBM PC &amp; DOS)</td>
<td>8086</td>
</tr>
<tr>
<td>1980</td>
<td>Add Floating Point instructions</td>
<td>8087</td>
</tr>
<tr>
<td>1985</td>
<td>32-bit ISA (Refer to as IA32)</td>
<td>386</td>
</tr>
<tr>
<td>1997</td>
<td>Add Multi-Media eXtension (MMX)</td>
<td>Pentium/MMX</td>
</tr>
<tr>
<td>1999</td>
<td>Add Streaming SIMD Extension (SSE)</td>
<td>Pentium III</td>
</tr>
<tr>
<td>2001</td>
<td>Intel’s first attempt at 64-bit ISA (IA64, failed)</td>
<td>Itanium</td>
</tr>
<tr>
<td>2004</td>
<td>Implement AMD’s 64-bit ISA (x86-64, AMD64)</td>
<td>Pentium 4E</td>
</tr>
<tr>
<td>2008</td>
<td>Add Advanced Vector Extension (AVE)</td>
<td>Core i7 Sandy Bridge</td>
</tr>
</tbody>
</table>
### Intel x86 ISA Evolution (Milestones)

- Evolutionary design: Added more features as time goes on

<table>
<thead>
<tr>
<th>Date</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1974</td>
<td>8-bit ISA 8080</td>
</tr>
<tr>
<td>1978</td>
<td>16-bit ISA (Basis for IBM PC &amp; DOS) 8086</td>
</tr>
<tr>
<td>1980</td>
<td>Add Floating Point instructions 8087</td>
</tr>
<tr>
<td>1985</td>
<td>32-bit ISA (Refer to as IA32) 386</td>
</tr>
<tr>
<td>1997</td>
<td>Add Multi-Media eXtension (MMX) Pentium/MMX</td>
</tr>
<tr>
<td>1999</td>
<td>Add Streaming SIMD Extension (SSE) Pentium III</td>
</tr>
<tr>
<td>2001</td>
<td>Intel's first attempt at 64-bit ISA (IA64, failed) Itanium</td>
</tr>
<tr>
<td>2004</td>
<td>Implement AMD's 64-bit ISA (x86-64, AMD64) Pentium 4E</td>
</tr>
<tr>
<td>2008</td>
<td>Add Advanced Vector Extension (AVE) Core i7 Sandy Bridge</td>
</tr>
</tbody>
</table>
Backward Compatibility

- Binary executable generated for an older processor can execute on a newer processor
- Allows legacy code to be executed on newer machines
  - Buy new machines without changing the software
- x86 is backward compatible up until 8086 (16-bit ISA)
  - i.e., an 8086 binary executable can be executed on any of today’s x86 machines
- Great for users, nasty for processor implementers
  - Every instruction you put into the ISA, you are stuck with it FOREVER
x86 Clones: Advanced Micro Devices (AMD)

• Historically
  • AMD build processors for x86 ISA
  • A little bit slower, a lot cheaper

• Then
  • Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  • Developed x86-64, their own 64-bit x86 extension to IA32
  • Built first 1 GHz CPU

• Intel felt hard to admit mistake or that AMD was better

• 2004: Intel Announces EM64T extension to IA32
  • Almost identical to x86-64!
  • Today’s 64-bit x86 ISA is basically AMD’s original proposal
x86 Clones: Advanced Micro Devices (AMD)

• Today: Holding up not too badly
x86 Clones: Advanced Micro Devices (AMD)

• Today: Holding up not too badly
x86 Clones: Advanced Micro Devices (AMD)

• Today: Holding up not too badly
Our Coverage

- IA32
  - The traditional x86
  - 2\textsuperscript{nd} edition of the textbook

- x86-64
  - The standard
  - CSUG machine
  - 3\textsuperscript{rd} edition of the textbook
  - Our focus
Aside: Moore’s Law

- More instructions require more transistors to implement
Aside: Moore’s Law

- More instructions require more transistors to implement
Aside: Moore’s Law

• More instructions require more transistors to implement
Aside: Moore’s Law

- More instructions require more transistors to implement
Aside: Moore’s Law

- More instructions require more transistors to implement
- Gordon Moore in 1965 predicted that the number of transistors doubles every year
Aside: Moore’s Law

- More instructions require more transistors to implement
- Gordon Moore in 1965 predicted that the number of transistors doubles every year
Aside: Moore’s Law

• More instructions require more transistors to implement
• Gordon Moore in 1965 predicted that the number of transistors doubles every year
Aside: Moore’s Law

- More instructions require more transistors to implement
- Gordon Moore in 1965 predicted that the number of transistors doubles every year
Aside: Moore’s Law

• More instructions require more transistors to implement
• Gordon Moore in 1965 predicted that the number of transistors doubles every year
• In 1975 he revised the prediction to doubling every 2 years
Aside: Moore’s Law

• More instructions require more transistors to implement
• Gordon Moore in 1965 predicted that the number of transistors doubles every year
• In 1975 he revised the prediction to doubling every 2 years
• Today’s widely-known Moore’s Law: number of transistors double about every 18 months
  • Moore never used the number 18…
Aside: Moore’s Law
Aside: Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
Aside: Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
Aside: Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • \(~1.4x\) smaller each dimension\((1.4^2 \sim 2)\)
Aside: Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • ~1.4x smaller each dimension ($1.4^2 \sim 2$)

• Moore’s Law is:
Aside: Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • ~1.4x smaller each dimension\(1.4^2 \sim 2\)

• Moore’s Law is:
  • A law of physics?
Aside: Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • ~1.4x smaller each dimension ($1.4^2 \sim 2$)

• Moore’s Law is:
  • A law of physics?  No
Aside: Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • ~1.4x smaller each dimension\((1.4^2 \sim 2)\)

• Moore’s Law is:
  • A law of physics? \textbf{No}
  • A law of circuits?
Aside: Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • ~1.4x smaller each dimension (1.4^2 ~ 2)

• Moore’s Law is:
  • A law of physics?  No
  • A law of circuits?  No
Aside: Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • ~1.4x smaller each dimension\( (1.4^2 \sim 2) \)

• Moore’s Law is:
  • A law of physics? \textit{No}
  • A law of circuits? \textit{No}
  • A law of economy?
Aside: Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • ~1.4x smaller each dimension (1.4^2 ~ 2)

• Moore’s Law is:
  • A law of physics? No
  • A law of circuits? No
  • A law of economy? Yes
Aside: Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?

Transistors will stop shrinking in 2021, but Moore’s law will live on

Final semiconductor industry roadmap says the future is 3D packaging and cooling.

The first problem has been known about for a long while. Basically, starting at around the 65nm node in 2006, the economic gains from moving to smaller transistors have been slowly dribbling away. Previously, moving to a smaller node meant you could cram tons more chips onto a single silicon wafer, at a reasonably small price increase. With recent nodes like 22 or 14nm, though, there are so many additional steps required that it costs a lot more to manufacture a completed wafer—not to mention additional costs for things like package-on-package (PoP) and through-silicon vias (TSV) packaging.
Aside: Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?

Transistors will stop shrinking in 2021, but Moore’s law will live on

Final semiconductor industry roadmap says the future is 3D packaging and cooling.

The first problem has been known about for a long while. Basically, starting at around the 65nm node in 2006, the economic gains from moving to smaller transistors have been slowly dribbling away. Previously, moving to a smaller node meant you could cram tons more chips onto a single silicon wafer, at a reasonably small price increase. With recent nodes like 22 or 14nm, though, there are so many additional steps required that it costs a lot more to manufacture a completed wafer—not to mention additional costs for things like package-on-package (PoP) and through-silicon vias (TSV) packaging.
Aside: Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • ~1.4x smaller each dimension (1.4^2 ~ 2)

• Moore’s Law is:
  • A law of physics?  No
  • A law of circuits?  No
  • A law of economy?  Yes
  • A law of psychology?
Aside: Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • $\sim 1.4^x$ smaller each dimension ($1.4^2 \sim 2$)

• Moore’s Law is:
  • A law of physics?  No
  • A law of circuits?  No
  • A law of economy?  Yes
  • A law of psychology?  Yes
Aside: Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • ~1.4x smaller each dimension (1.4^2 ~ 2)

• Moore’s Law is:
  • A law of physics?  No
  • A law of circuits?  No
  • A law of economy?  Yes
  • A law of psychology?  Yes

Questions?
Today: Assembly Programming I: Basics

- Different ISAs and history behind them
- C, assembly, machine code
- Move operations (and addressing modes)
Assembly Code’s View of Computer: ISA
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

CPU

Memory
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- (Byte Addressable) Memory
  - Code: instructions
  - Data
  - Stack to support function call

CPU

Memory
- Code
- Data
- Stack
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

• (Byte Addressable) Memory
  • Code: instructions
  • Data
  • Stack to support function call
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- (Byte Addressable) Memory
  - Code: instructions
  - Data
  - Stack to support function call

CPU

Memory

Code
Data
Stack

Data

0x53
0x48
0x89
0xd3
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- (Byte Addressable) Memory
  - Code: instructions
  - Data
  - Stack to support function call

Instruction is the fundamental unit of work.
All instructions are coded as bits (just like data!)
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- (Byte Addressable) Memory
  - Code: instructions
  - Data
  - Stack to support function call

![Diagram of CPU and Memory with Code (Instructions), Data, Stack sections and address values: 0x53, 0x48, 0x89, 0xd3]
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- (Byte Addressable) Memory
  - Code: instructions
  - Data
  - Stack to support function call

- Register file
  - Faster memory (e.g., 0.5 ns vs. 15 ns)
  - Small memory (e.g., 128 B vs. 16 GB)
  - Heavily used program data
## x86-64 Integer Register File

<table>
<thead>
<tr>
<th>8 Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
</tr>
<tr>
<td>%rbx</td>
</tr>
<tr>
<td>%rcx</td>
</tr>
<tr>
<td>%rdx</td>
</tr>
<tr>
<td>%rsi</td>
</tr>
<tr>
<td>%rdi</td>
</tr>
<tr>
<td>%rsp</td>
</tr>
<tr>
<td>%rbp</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%r8</td>
</tr>
<tr>
<td>%r9</td>
</tr>
<tr>
<td>%r10</td>
</tr>
<tr>
<td>%r11</td>
</tr>
<tr>
<td>%r12</td>
</tr>
<tr>
<td>%r13</td>
</tr>
<tr>
<td>%r14</td>
</tr>
<tr>
<td>%r15</td>
</tr>
</tbody>
</table>
x86-64 Integer Register File

- Lower-half of each register can be independently addressed (until 8 bytes)
x86-64 Integer Register File

- Lower-half of each register can be independently addressed (until 8 bytes)
x86-64 Integer Register File

• Lower-half of each register can be independently addressed (until 8 bytes)
x86-64 Integer Register File

• Lower-half of each register can be independently addressed (until 8 bytes)
x86-64 Integer Register File

• Lower-half of each register can be independently addressed (until 8 bytes)
x86-64 Integer Register File

- Lower-half of each register can be independently addressed (until 8 bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
</tr>
<tr>
<td>Pointer</td>
<td>8</td>
</tr>
</tbody>
</table>
x86-64 Integer Register File

- Lower-half of each register can be independently addressed (until 8 bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
</tr>
<tr>
<td>Pointer</td>
<td>8</td>
</tr>
</tbody>
</table>

Floating point data is stored in a separate set of register file (in 3 lectures…).
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- (Byte Addressable) Memory
  - Code: instructions
  - Data
  - Stack to support function call

- Register file
  - Faster memory (e.g., 0.5 ns vs. 15 ns)
  - Small memory (e.g., 128 B vs. 16 GB)
  - Heavily used program data
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- (Byte Addressable) Memory
  - Code: instructions
  - Data
  - Stack to support function call

- Register file
  - Faster memory (e.g., 0.5 ns vs. 15 ns)
  - Small memory (e.g., 128 B vs. 16 GB)
  - Heavily used program data

- PC: Program counter
  - A special register containing address of next instruction
  - Called “RIP” in x86-64
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- **CPU**
  - **PC**: Program counter
    - A special register containing address of next instruction
    - Called “RIP” in x86-64

- **Register file**
  - Faster memory (e.g., 0.5 ns vs. 15 ns)
  - Small memory (e.g., 128 B vs. 16 GB)
  - Heavily used program data

- **Memory**
  - Code
  - Data
  - Stack to support function call

- **(Byte Addressable) Memory**
  - Code: instructions
  - Data

- **PC**: Program counter
  - A special register containing address of next instruction
  - Called “RIP” in x86-64
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- **CPU**
  - PC: Program counter
  - A special register containing address of next instruction
  - Called “RIP” in x86-64
  - Register file
    - Faster memory (e.g., 0.5 ns vs. 15 ns)
    - Small memory (e.g., 128 B vs. 16 GB)
    - Heavily used program data

- **Memory**
  - Code
  - Data
  - Stack to support function call

- **(Byte Addressable) Memory**
  - Code: instructions
  - Data

- **Instructions**

- **Addresses**
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- (Byte Addressable) Memory
  - Code: instructions
  - Data
  - Stack to support function call

- Register file
  - Faster memory (e.g., 0.5 ns vs. 15 ns)
  - Small memory (e.g., 128 B vs. 16 GB)
  - Heavily used program data

- PC: Program counter
  - A special register containing address of next instruction
  - Called “RIP” in x86-64
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

• (Byte Addressable) Memory
  • Code: instructions
  • Data
  • Stack to support function call

• Register file
  • Faster memory (e.g., 0.5 ns vs. 15 ns)
  • Small memory (e.g., 128 B vs. 16 GB)
  • Heavily used program data

• PC: Program counter
  • A special register containing address of next instruction
  • Called “RIP” in x86-64

• Arithmetic logic unit (ALU)
  • Where computation happens
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- **(Byte Addressable) Memory**
  - Code: instructions
  - Data
  - Stack to support function call

- **Register file**
  - Faster memory (e.g., 0.5 ns vs. 15 ns)
  - Small memory (e.g., 128 B vs. 16 GB)
  - Heavily used program data

- **PC: Program counter**
  - A special register containing address of next instruction
  - Called “RIP” in x86-64

- **Arithmetic logic unit (ALU)**
  - Where computation happens

- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branch
Assembly Program Instructions

Assembly Programmer’s Perspective of a Computer
Assembly Program Instructions

Assembly Programmer’s Perspective of a Computer

- **Compute Instruction**: Perform arithmetics on register or memory data
  - `addq %eax, %ebx`
  - C constructs: +, -, >>, etc.
Assembly Program Instructions

Assembly Programmer’s Perspective of a Computer

- **Compute Instruction**: Perform arithmetics on register or memory data
  - `addq %eax, %ebx`
  - C constructs: +, -, >>, etc.
- **Data Movement Instruction**: Transfer data between memory and register
  - `movq %eax, (%ebx)`
Assembly Program Instructions

Assembly Programmer’s Perspective of a Computer

- **Compute Instruction**: Perform arithmetics on register or memory data
  - `addq %eax, %ebx`
  - C constructs: +, -, >>, etc.

- **Data Movement Instruction**: Transfer data between memory and register
  - `movq %eax, (%ebx)`

- **Control Instruction**: Alter the sequence of instructions (by changing PC)
  - `jmp`, `call`
  - C constructs: *if-else, do-while*, function call, etc.
Turning C into Object Code

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;
}
```
Turning C into Object Code

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
```
Turning C into Object Code

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 CSUG machine) with command

```
gcc -Og -S sum.c -o sum.s
```
Turning C into Object Code

Generated x86-64 Assembly

```assembly
sumstore:
pushq   %rbx
movq    %rdx, %rbx
call    plus
movq    %rax, (%rbx)
popq    %rbx
ret
```
Turning C into Object Code

Generated x86-64 Assembly

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

Binary Code for `sumstore`

Memory

```
0x53
0x48
0x89
0xd3
0xe8
0xf2
0xff
0xff
0xff
0x48
0x89
0x03
0x5b
0xc3
```
Turning C into Object Code

Generated x86-64 Assembly

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

Binary Code for `sumstore`

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0400595</td>
<td>0x53</td>
</tr>
<tr>
<td>0x48</td>
<td>0x48</td>
</tr>
<tr>
<td>0x89</td>
<td>0x89</td>
</tr>
<tr>
<td>0xd3</td>
<td>0xd3</td>
</tr>
<tr>
<td>0xe8</td>
<td>0xe8</td>
</tr>
<tr>
<td>0xf2</td>
<td>0xf2</td>
</tr>
<tr>
<td>0xff</td>
<td>0xff</td>
</tr>
<tr>
<td>0xff</td>
<td>0xff</td>
</tr>
<tr>
<td>0xff</td>
<td>0xff</td>
</tr>
<tr>
<td>0x48</td>
<td>0x48</td>
</tr>
<tr>
<td>0x89</td>
<td>0x89</td>
</tr>
<tr>
<td>0x03</td>
<td>0x03</td>
</tr>
<tr>
<td>0x5b</td>
<td>0x5b</td>
</tr>
<tr>
<td>0xc3</td>
<td>0xc3</td>
</tr>
</tbody>
</table>
Turning C into Object Code

Generated x86-64 Assembly

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

Binary Code for `sumstore`

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0400595</td>
<td>0x53</td>
</tr>
<tr>
<td></td>
<td>0x48</td>
</tr>
<tr>
<td></td>
<td>0x89</td>
</tr>
<tr>
<td></td>
<td>0xd3</td>
</tr>
<tr>
<td></td>
<td>0xe8</td>
</tr>
<tr>
<td></td>
<td>0xf2</td>
</tr>
<tr>
<td></td>
<td>0xff</td>
</tr>
<tr>
<td></td>
<td>0xff</td>
</tr>
<tr>
<td></td>
<td>0xff</td>
</tr>
<tr>
<td></td>
<td>0x48</td>
</tr>
<tr>
<td></td>
<td>0x89</td>
</tr>
<tr>
<td></td>
<td>0x03</td>
</tr>
<tr>
<td></td>
<td>0x5b</td>
</tr>
<tr>
<td></td>
<td>0xc3</td>
</tr>
</tbody>
</table>

Obtain (on CSUG machine) with command

```
gcc -c sum.s -o sum.o
```
Turning C into Object Code

Generated x86-64 Assembly

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

Binary Code for `sumstore`

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0400595</td>
<td>0x53 0x48 0x89 0xd3 0xe8 0xf2 0xff 0xff 0xff 0x48 0x89 0x03 0x5b 0xc3</td>
</tr>
</tbody>
</table>

Obtain (on CSUG machine) with command

```
gcc -c sum.s -o sum.o
```

- Total of 14 bytes
- Instructions have variable lengths: e.g., 1, 3, or 5 bytes
- Code starts at memory address 0x0400595
Machine Instruction Example
Machine Instruction Example

```c
long t;
long *dest;
t += *dest;
```

- C Code
  - Add value `t` with value in memory location whose address is `dest` and store the result back to `t`
Machine Instruction Example

• C Code
  • Add value `t` with value in memory location whose address is `dest` and store the result back to `t`

• Assembly Instruction
  • Operator: Add two 8-byte values
    • Quad words in x86-64 parlance
  • Operands:
    
    |   |   |
    |---|---|
    | `t` | Register %rax |
    | `dest` | Register %rbx |
    | `*dest` | Memory M[|%rbx|] |
Machine Instruction Example

C Code
- Add value `t` with value in memory location whose address is `dest` and store the result back to `t`.

Assembly Instruction
- Operator: Add two 8-byte values
  - Quad words in x86-64 parlance
- Operands:
  - `t`: Register `%rax`
  - `dest`: Register `%rbx`
  - `*dest`: Memory `M[%rbx]`
Machine Instruction Example

C Code

- Add value `t` with value in memory location whose address is `dest` and store the result back to `t`

Assembly Instruction

- Operator: Add two 8-byte values
  - Quad words in x86-64 parlance
- Operands:
  - `t`: Register `%rax`
  - `dest`: Register `%rbx`
  - `*dest`: Memory `M[%rbx]`

```c
long t;
long *dest;
t += *dest;
```
Machine Instruction Example

C Code
- Add value \( t \) with value in memory location whose address is \( \text{dest} \) and store the result back to \( t \)

Assembly Instruction
- Operator: Add two 8-byte values
  - Quad words in x86-64 parlance
- Operands:
  - \( t \): Register \( \%rax \)
  - \( \text{dest} \): Register \( \%rbx \)
  - \( *\text{dest} \): Memory \( M[\%rbx] \)
Machine Instruction Example

- **C Code**
  - Add value \( t \) with value in memory location whose address is \( \text{dest} \) and store the result back to \( t \)

- **Assembly Instruction**
  - Operator: Add two 8-byte values
    - Quad words in x86-64 parlance
  - Operands:
    - \( t: \) Register \%rax
    - \( \text{dest}: \) Register \%rbx
    - \( *\text{dest}: \) Memory \( M[\%rbx] \)

```c
long t;
long *dest;
t += *dest;
```
Machine Instruction Example

- **C Code**
  - Add value `t` with value in memory location whose address is `dest` and store the result back to `t`.

- **Assembly Instruction**
  - Operator: Add two 8-byte values.
    - Quad words in x86-64 parlance.
  - Operands:
    - `t`: Register `%rax`
    - `dest`: Register `%rbx`
    - `*dest`: Memory `M[rbx]`
Machine Instruction Example

### C Code
- Add value `t` with value in memory location whose address is `dest` and store the result back to `t`

```c
long t;
long *dest;
t += *dest;
```

### Assembly Instruction
- Operator: Add two 8-byte values
- Quad words in x86-64 parlance
- Operands:
  - `t`: Register `%rax`
  - `dest`: Register `%rbx`
  - `*dest`: Memory `M[%rbx]`

```assembly
addq %rax, (%rbx)
```

### Object Code
- 3-byte instruction
- Stored at address `0xf0059e`

```
0x 48 01 d8
```
Instruction Processing Sequence

Assembly Programmer’s Perspective of a Computer

Fetch Instruction (According to PC)
Instruction Processing Sequence

Assembly Programmer’s Perspective of a Computer

Fetch Instruction (According to PC)

0x4801d8
Instruction Processing Sequence

Assembly Programmer’s Perspective of a Computer

CPU
- PC
- ALU
- Register File
- Condition Codes

Memory
- Addresses
- Data
- Code
- Data
- Stack

Fetch Instruction (According to PC) → Decode Instruction

```
addq %rax, (%rbx)
```
Instruction Processing Sequence

Assembly Programmer’s Perspective of a Computer

CPU
- PC
- ALU
- Register File
- Condition Codes

Memory
- Code
- Data
- Stack

Fetch Instruction (According to PC) → Decode Instruction → Fetch Operands
Instruction Processing Sequence

Assembly Programmer’s Perspective of a Computer

CPU
- PC
- ALU
- Register File
- Condition Codes

Memory
- Code
- Data
- Stack

Addresses

Data

Instructions

Fetch Instruction (According to PC) → Decode Instruction → Fetch Operands → Execute Instruction
Instruction Processing Sequence

Assembly Programmer’s Perspective of a Computer

CPU
- PC
- ALU
  - Register File
  - Condition Codes

Memory
- Code
- Data
- Stack

Fetch Instruction (According to PC) → Decode Instruction → Fetch Operands → Execute Instruction → Update Condition Codes
Assembly Programmer’s Perspective of a Computer

Instruction Processing Sequence

CPU
- PC
- ALU
- Register File
- Condition Codes

Memory
- Code
- Data
- Stack

Addresses
Data
Instructions

Fetch Instruction (According to PC) → Decode Instruction → Fetch Operands → Execute Instruction → Store Results

Update Condition Codes
Instruction Processing Sequence

Assembly Programmer’s Perspective of a Computer

Fetch Instruction (According to PC) → Decode Instruction → Fetch Operands → Execute Instruction → Store Results

Memory

- Code
- Data
- Stack

CPU

- PC
- ALU
- Register File
- Condition Codes

Addresses

Data

Instructions

Update Condition Codes

Adjust PC
Instruction Processing Sequence

Assembly Programmer’s Perspective of a Computer

CPU
- PC
- ALU
- Register File
- Condition Codes

Memory
- Code
- Data
- Stack

Addresses
Data
Instructions

Fetch Instruction (According to PC) → Decode Instruction → Fetch Operands → Execute Instruction → Store Results

Update Condition Codes

Adjust PC
Instruction Processing Sequence

Assembly Programmer’s Perspective of a Computer

Questions?

CPU
- PC
- ALU
- Register File
- Condition Codes

Memory
- Code
- Data
- Stack

Fetch Instruction (According to PC) → Decode Instruction → Fetch Operands → Execute Instruction → Store Results

Update Condition Codes → Adjust PC
Today: Assembly Programming I: Basics

- Different ISAs and history behind them
- C, assembly, machine code
- Move operations (and addressing modes)
Data Movement Instructions

\texttt{movq} Source, Dest
Data Movement Instructions

\textbf{movq} \textit{Source, Dest}

Operator  Operands
Data Movement Instructions

\[ \text{movq} \ Source, \ Dest \]

- **Register**: One of 16 integer registers
  - Example: \%rax, \%r13
  - But \%rsp reserved for special use
Data Movement Instructions

\[ \text{movq } \text{Source, Dest} \]

- **Register:** One of 16 integer registers
  - Example: \%rax, \%r13
  - But \%rsp reserved for special use
- **Immediate:** Constant integer data
  - Example: \$0x400, \$–533; like C constant, but prefixed with ‘$’
  - Encoded with 1, 2, or 4 bytes; can only be source
Data Movement Instructions

\textbf{movq} Source, Dest

\textbf{Operator} \quad \textbf{Operands}

- **Register:** One of 16 integer registers
  - Example: \%rax, \%r13
  - But \%rsp reserved for special use

- **Immediate:** Constant integer data
  - Example: $0\times400$, $-533$; like C constant, but prefixed with ‘$’
  - Encoded with 1, 2, or 4 bytes; can only be source

- **Memory:** 8 consecutive bytes in memory at given \texttt{address}
  - Simplest example: (\%rax)
  - How to obtain the address is called “addressing mode” (later…).
## Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src,Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Imm</em></td>
<td><em>Reg</em></td>
<td><em>Reg,Mem</em></td>
<td><em>C Analog</em></td>
</tr>
<tr>
<td><em>Reg</em></td>
<td><em>Reg</em></td>
<td><em>Reg,Mem</em></td>
<td><em>C Analog</em></td>
</tr>
<tr>
<td><em>Mem</em></td>
<td><em>Reg</em></td>
<td><em>Reg,Mem</em></td>
<td><em>C Analog</em></td>
</tr>
</tbody>
</table>

*movq*

*Cannot do memory-memory transfer with a single instruction in x86.*
### Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src, Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Imm} )</td>
<td>( \text{Reg} )</td>
<td>( \text{Reg} )</td>
<td>\text{movq} $0x4, %rax$</td>
</tr>
<tr>
<td>( \text{Reg} )</td>
<td>( \text{Mem} )</td>
<td>( \text{Mem} )</td>
<td>\text{C Analog}</td>
</tr>
<tr>
<td>( \text{Mem} )</td>
<td>( \text{Reg} )</td>
<td>\text{C Analog}</td>
<td>\text{C Analog}</td>
</tr>
</tbody>
</table>

**Cannot do memory-memory transfer with a single instruction in x86.**
### 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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Example**: `movq $0x4,%rax`  
  - temp = 0x4;

**Cannot do memory-memory transfer with a single instruction in x86.**
### 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><strong>Imm</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reg</td>
<td>Mem</td>
<td>movq $0x4, %rax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td></td>
<td>movq $-147, (%rax)</td>
<td></td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Mem</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Cannot do memory-memory transfer with a single instruction in x86.*
**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></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reg</td>
<td>movq $0x4, %rax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td></td>
<td>movq $-147, (%rax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**movq**

*Cannot do memory-memory transfer with a single instruction in x86.*
Cannot do memory-memory transfer with a single instruction in x86.

**Operand Combinations**

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

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

**Cannot do memory-memory transfer with a single instruction in x86.**
### 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><code>movq $0x4,%rax</code></td>
<td><code>temp = 0x4;</code></td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td><code>movq $-147,(%rax)</code></td>
<td><code>*p = -147;</code></td>
</tr>
<tr>
<td>Reg</td>
<td>Reg</td>
<td><code>movq %rax,%rdx</code></td>
<td><code>temp2 = temp1;</code></td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td><code>movq %rax,(%rdx)</code></td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Cannot do memory-memory transfer with a single instruction in x86.*
## 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><strong>Imm</strong></td>
<td>Reg</td>
<td>movq $0x4,%rax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movq $-147,(%rax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td>Reg</td>
<td>movq %rax,%rdx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movq %rax,(%rdx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Cannot do memory-memory transfer with a single instruction in x86.**
### 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></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 in x86.
### Operan Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src, Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg</td>
<td>Mem</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 in x86.**
Memory Addressing Modes

• An addressing mode specifies:
  • how to calculate the effective memory address of an operand
  • by using information held in registers and/or constants
Memory Addressing Modes

• An addressing mode specifies:
  • how to calculate the effective memory address of an operand
  • by using information held in registers and/or constants

• Normal: (R)
  • Memory address: content of Register R (Reg[R])
  • Pointer dereferencing in C

\[
\text{movq } (%rcx),%rax; \ // \ adddress \ = \ %rcx
\]
Memory Addressing Modes

• An addressing mode specifies:
  • how to calculate the effective memory address of an operand
  • by using information held in registers and/or constants

• Normal: (R)
  • Memory address: content of Register R (Reg[R])
  • Pointer dereferencing in C

\[
\text{movq } (%\text{rcx}),\%rax; \ // \text{ address } = \%rcx
\]

• Displacement: D(R)
  • Memory address: Reg[R]+D
  • Register R specifies start of memory region
  • Constant displacement D specifies offset

\[
\text{movq } 8(%\text{rbp}),\%rdx; \ // \text{ address } = \%rbp + 8
\]
Example of Simple Addressing Modes

```c
void swap( long *xp, long *yp )
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```
Example of Simple Addressing Modes

```c
void swap (long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```
Example of Simple Addressing Modes

```c
void swap
    (long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```
Example of Simple Addressing Modes

```c
void swap (long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```
Example of Simple Addressing Modes

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

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi xp</td>
<td>*xp xp</td>
<td>xp</td>
</tr>
<tr>
<td>%rsi yp</td>
<td>*yp yp</td>
<td>yp</td>
</tr>
</tbody>
</table>
Example of Simple Addressing Modes

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

swap:
  movq (%rdi), %rax  # t0 = *xp
  movq (%rsi), %rdx  # t1 = *yp
  movq %rdx, (%rdi)  # *xp = t1
  movq %rax, (%rsi)  # *yp = t0
  ret
Example of Simple Addressing Modes

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

**How Does This Work?**

```
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></td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>xp</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>yp</td>
</tr>
</tbody>
</table>

### Addr

<table>
<thead>
<tr>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td></td>
</tr>
<tr>
<td>456</td>
<td></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**
- `%rdi`: 0x120
- `%rsi`: 0x100
- `%rax`: 
- `%rdx`: 

**Memory**
- **Addr**: 
  - 0x120  `xp`
  - 0x118
  - 0x110
  - 0x108
  - 0x100  `yp`

<table>
<thead>
<tr>
<th>swap:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>movq   (%rdi), %rax</code></td>
<td># <code>t0 = *xp</code></td>
</tr>
<tr>
<td><code>movq   (%rsi), %rdx</code></td>
<td># <code>t1 = *yp</code></td>
</tr>
<tr>
<td><code>movq   %rdx, (%rdi)</code></td>
<td># <code>*xp = t1</code></td>
</tr>
<tr>
<td><code>movq   %rax, (%rsi)</code></td>
<td># <code>*yp = t0</code></td>
</tr>
<tr>
<td><code>ret</code></td>
<td></td>
</tr>
</tbody>
</table>
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></td>
</tr>
</tbody>
</table>

**Memory**

<table>
<thead>
<tr>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x100</td>
<td>456</td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x118</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 \texttt{Swap()}

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>123</td>
<td>0x120 (xp)</td>
</tr>
<tr>
<td>%rsi</td>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td>%rax</td>
<td></td>
<td>0x110</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
<td>0x100 (yp)</td>
</tr>
</tbody>
</table>

\texttt{swap:}

\begin{align*}
\text{movq} & \quad (%rdi), \ %rax \quad \# \ t0 = *xp \\
\text{movq} & \quad (%rsi), \ %rdx \quad \# \ t1 = *yp \\
\text{movq} & \quad %rdx, \ (%rdi) \quad \# \ *xp = t1 \\
\text{movq} & \quad %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>Addr</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>

### Example Code

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

\begin{tabular}{|c|c|c|c|}
\hline
\textbf{Registers} & \textbf{Memory} & \textbf{Addr} \\
\hline
%rdi & 0x120 & 123 & 0x120 \ xp \\
%rsi & 0x100 & & 0x118 \\
%rax & 123 & & 0x110 \\
%rdx & 456 & & 0x108 \\
\hline
\end{tabular}

\texttt{swap:}

\begin{align*}
\text{movq} & \quad (%\text{rdi}), \ %\text{rax} \quad \# \ t0 = \ *\text{xp} \\
\text{movq} & \quad (%\text{rsi}), \ %\text{rdx} \quad \# \ t1 = \ *\text{yp} \\
\text{movq} & \quad %\text{rdx}, \ (%\text{rdi}) \quad \# \ *\text{xp} = \ t1 \\
\text{movq} & \quad %\text{rax}, \ (%\text{rsi}) \quad \# \ *\text{yp} = \ t0 \\
\text{ret} & \quad \text{} \\
\end{align*}
### 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>Addr</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>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>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>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>456</td>
</tr>
<tr>
<td>0x118</td>
<td>0x120</td>
</tr>
<tr>
<td>0x110</td>
<td>0x118</td>
</tr>
<tr>
<td>0x108</td>
<td>0x110</td>
</tr>
<tr>
<td>0x100</td>
<td>456</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
```