Floating Point II, x86-64 Intro
CSE 351 Autumn 2017

Instructor:
Justin Hsia

Teaching Assistants:
Lucas Wotton    Michael Zhang    Parker DeWilde    Ryan Wong
Sam Gehman     Sam Wolfson     Savanna Yee     Vinny Palaniappan
Administrivia

- Lab 1 due on Friday (10/13)
  - Submit `bits.c, pointer.c, lab1reflect.txt`
- Homework 2 due next Friday (10/20)
  - On Integers, Floating Point, and x86-64
- Section tomorrow on Integers and Floating Point
- Peer Instruction Questions are for your benefit!
  - TAs are scattered about as well to help
Floating point topics

- Fractional binary numbers
- IEEE floating-point standard
- **Floating-point operations and rounding**
- Floating-point in C

- There are many more details that we won’t cover
  - It’s a 58-page standard...
## Floating Point Encoding Summary

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Mantissa</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0</td>
<td>± 0</td>
</tr>
<tr>
<td>0x00</td>
<td>non-zero</td>
<td>± denorm num</td>
</tr>
<tr>
<td>0x01 – 0xFE</td>
<td>anything</td>
<td>± norm num</td>
</tr>
<tr>
<td>0xFF</td>
<td>0</td>
<td>± ∞</td>
</tr>
<tr>
<td>0xFF</td>
<td>non-zero</td>
<td>NaN</td>
</tr>
</tbody>
</table>
Distribution of Values

- What ranges are NOT representable?
  - Between largest norm and infinity \textbf{Overflow} (Exp too large)
  - Between zero and smallest denorm \textbf{Underflow} (Exp too small)
  - Between norm numbers? \textbf{Rounding}

- Given a FP number, what’s the bit pattern of the next largest representable number?
  - What is this “step” when \text{Exp} = 0?
  - What is this “step” when \text{Exp} = 100?

- Distribution of values is denser toward zero
Floating Point Operations: Basic Idea

Value = \((-1)^S \times \text{Mantissa} \times 2^\text{Exponent}\)

- \(x +_f y = \text{Round}(x + y)\)
- \(x \times_f y = \text{Round}(x \times y)\)

Basic idea for floating point operations:
- First, compute the exact result
- Then *round* the result to make it fit into desired precision:
  - Possibly over/underflow if exponent outside of range
  - Possibly drop least-significant bits of mantissa to fit into M bit vector
Floating Point Addition

- \((-1)^{S_1} \times Man_1 \times 2^{Exp_1} + (-1)^{S_2} \times Man_2 \times 2^{Exp_2}\)
  - Assume \(Exp_1 > Exp_2\)

- Exact Result: \((-1)^{S} \times Man \times 2^{Exp}\)
  - Sign \(S\), mantissa \(Man\):
    - Result of signed align & add
  - Exponent \(E\): \(E_1\)

- Adjustments:
  - If \(Man \geq 2\), shift \(Man\) right, increment \(Exp\)
  - If \(Man < 1\), shift \(Man\) left \(k\) positions, decrement \(Exp\) by \(k\)
  - Over/underflow if \(Exp\) out of range
  - Round \(Man\) to fit mantissa precision

\[
\begin{align*}
1.010 \times 2^2 & \quad + \quad 1.0100 \times 2^2 \\
+ 1.000 \times 2^{-1} & \quad + \quad 0.0010 \times 2^2 \\
\text{???} & \quad \text{??}
\end{align*}
\]
Floating Point Multiplication

\[ (-1)^{S_1} \times \text{Man}_1 \times 2^{\text{Exp}_1} \times (-1)^{S_2} \times \text{Man}_2 \times 2^{\text{Exp}_2} \]

- **Exact Result:** \((-1)^S \times \text{M} \times 2^E\)
  - **Sign** \(S\): \(S_1 \land S_2\)
  - **Mantissa** \(\text{Man}\): \(\text{Man}_1 \times \text{Man}_2\)
  - **Exponent** \(\text{Exp}\): \(\text{Exp}_1 + \text{Exp}_2\)

- **Adjustments:**
  - If \(\text{Man} \geq 2\), shift \(\text{Man}\) right, increment \(\text{Exp}\)
  - Over/underflow if \(\text{Exp}\) out of range
  - Round \(\text{Man}\) to fit mantissa precision
Mathematical Properties of FP Operations

- Exponent overflow yields $+\infty$ or $-\infty$
- Floats with value $+\infty$, $-\infty$, and NaN can be used in operations
  - Result usually still $+\infty$, $-\infty$, or NaN; but not always intuitive
- Floating point operations do not work like real math, due to **rounding**
  - Not associative: $(3.14+1e100)–1e100 \neq 3.14+(1e100–1e100)$
    - $0 \neq 3.14$
  - Not distributive: $100*(0.1+0.2) \neq 100*0.1+100*0.2$
    - $30.000000000000003553 \neq 30$
- Not cumulative
  - Repeatedly adding a very small number to a large one may do nothing
Floating point topics

- Fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- Floating-point in C

- There are many more details that we won’t cover
  - It’s a 58-page standard...
Floating Point in C

- C offers two (well, 3) levels of precision
  - `float 1.0f` single precision (32-bit)
  - `double 1.0` double precision (64-bit)
  - `long double 1.0L` (“double double” or quadruple) precision (64-128 bits)

- `#include <math.h>` to get `INFINITY` and `NAN` constants

- Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results, so just avoid them!
Floating Point Conversions in C

- **Casting between int, float, and double changes the bit representation**
  - **int → float**
    - May be rounded (not enough bits in mantissa: 23)
    - Overflow impossible
  - **int or float → double**
    - Exact conversion (all 32-bit ints representable)
  - **long → double**
    - Depends on word size (32-bit is exact, 64-bit may be rounded)
  - **double or float → int**
    - Truncates fractional part (rounded toward zero)
    - “Not defined” when out of range or NaN: generally sets to Tmin (even if the value is a very big positive)
Floating Point and the Programmer

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    float f1 = 1.0;
    float f2 = 0.0;
    int i;
    for (i = 0; i < 10; i++)
        f2 += 1.0/10.0;
    printf("0x%08x 0x%08x\n", *(int*)&f1, *(int*)&f2);
    printf("f1 = %10.9f\n", f1);
    printf("f2 = %10.9f\n\n", f2);

    f1 = 1E30;
    f2 = 1E-30;
    float f3 = f1 + f2;
    printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );

    return 0;
}
```

$ ./a.out
0x3f800000 0x3f800001
f1 = 1.000000000
f2 = 1.000000119
f1 == f3? yes
Floating Point Summary

- Floats also suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow
  - “Gaps” produced in representable numbers means we can lose precision, unlike ints
    - Some “simple fractions” have no exact representation (e.g. 0.2)
    - “Every operation gets a slightly wrong result”
- Floating point arithmetic not associative or distributive
  - Mathematically equivalent ways of writing an expression may compute different results
- Never test floating point values for equality!
- Careful when converting between ints and floats!
Number Representation Really Matters

- **1991**: Patriot missile targeting error
  - clock skew due to conversion from integer to floating point
- **1996**: Ariane 5 rocket exploded ($1 billion)
  - overflow converting 64-bit floating point to 16-bit integer
- **2000**: Y2K problem
  - limited (decimal) representation: overflow, wrap-around
- **2038**: Unix epoch rollover
  - Unix epoch = seconds since 12am, January 1, 1970
  - signed 32-bit integer representation rolls over to Tmin in 2038
- **Other related bugs:**
  - 1982: Vancouver Stock Exchange 10% error in less than 2 years
  - 1994: Intel Pentium FDIV (floating point division) HW bug ($475 million)
  - 1997: USS Yorktown “smart” warship stranded: divide by zero
  - 1998: Mars Climate Orbiter crashed: unit mismatch ($193 million)
**Roadmap**

C:

```c
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();
```

- **Assembly language:**
  ```assembly
get_mpg:
  pushq  %rbp
  movq   %rsp, %rbp
  ...
  popq   %rbp
  ret
  ```

- **Machine code:**

  ```
  0111010000011000
  100011010000010000000010
  1000100111000010
  110000011111101000001111
  ```

- **Computer system:**
  - **OS:** Windows 10, OS X Yosemite
  - **x86 assembly**
  - **Procedures & stacks**
  - **Executables**
  - **Arrays & structs**
  - **Memory & caches**
  - **Processes**
  - **Virtual memory**
  - **Memory allocation**
  - **Java vs. C**
Translation

What makes programs run fast(er)?
HW Interface Affects Performance

Source code
Different applications or algorithms

Compiler
Perform optimizations, generate instructions

Architecture
Instruction set

Hardware
Different implementations

C Language

Program A

Program B

Your program

GCC

Clang

x86-64

ARMv8 (AArch64/A64)

Intel Pentium 4

Intel Core 2

Intel Core i7

AMD Opteron

AMD Athlon

ARM Cortex-A53

Apple A7
Instruction Set Architectures

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

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

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

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

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

- **Memory**
  - How do you specify a memory location?
# Mainstream ISAs

## x86
- **Designer**: Intel, AMD
- **Bits**: 16-bit, 32-bit and 64-bit
- **Introduced**: 1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
- **Design**: CISC
- **Type**: Register-memory
- **Encoding**: Variable (1 to 15 bytes)
- **Endianness**: Little

## ARM architectures
- **Designer**: ARM Holdings
- **Bits**: 32-bit, 64-bit
- **Introduced**: 1985; 31 years ago
- **Design**: RISC
- **Type**: Register-Register
- **Encoding**: AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility.[1]
- **Endianness**: Bi (little as default)

## MIPS
- **Designer**: MIPS Technologies, Inc.
- **Bits**: 64-bit (32→64)
- **Introduced**: 1981; 35 years ago
- **Design**: RISC
- **Type**: Register-Register
- **Encoding**: Fixed
- **Endianness**: Bi

---

**Macbooks & PCs**  
(Core i3, i5, i7, M)  
**x86-64 Instruction Set**

**Smartphone-like devices**  
(iPhone, iPad, Raspberry Pi)  
**ARM Instruction Set**

**Digital home & networking equipment**  
(Blu-ray, PlayStation 2)  
**MIPS Instruction Set**
Definitions

- **Architecture (ISA):** The parts of a processor design that one needs to understand to write assembly code
  - “What is directly visible to software”
- **Microarchitecture:** Implementation of the architecture
  - CSE/EE 469, 470

- Are the following part of the architecture?
  - Number of registers?
  - How about CPU frequency?
  - Cache size? Memory size?
Assembly Programmer’s View

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

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

- Integral data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, 10 or 2x8 or 4x4 or 8x2
  - Different registers for those (e.g. %xmm1, %ymm2)
  - Come from extensions to x86 (SSE, AVX, ...)
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
- Two common syntaxes
  - “AT&T”: used by our course, slides, textbook, gnu tools, ...
  - “Intel”: used by Intel documentation, Intel tools, ...
  - Must know which you’re reading

Not covered In 351
What is a Register?

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

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

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

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Register</th>
<th>Register</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
<td>%r8</td>
<td>%r8d</td>
</tr>
<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>
Some History: IA32 Registers – 32 bits wide

<table>
<thead>
<tr>
<th>Register</th>
<th>Origin</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>accumulate</td>
<td>general purpose</td>
</tr>
<tr>
<td>%edx</td>
<td>data</td>
<td>general purpose</td>
</tr>
<tr>
<td>%ebx</td>
<td>base</td>
<td>general purpose</td>
</tr>
<tr>
<td>%esi</td>
<td>source index</td>
<td>general purpose</td>
</tr>
<tr>
<td>%edi</td>
<td>destination index</td>
<td>general purpose</td>
</tr>
<tr>
<td>%esp</td>
<td>stack pointer</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>base pointer</td>
<td></td>
</tr>
</tbody>
</table>

16-bit virtual registers (backwards compatibility)
Name Origin (mostly obsolete)
## Memory vs. Registers

<table>
<thead>
<tr>
<th>Memory</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addresses</td>
<td><strong>vs.</strong> Names</td>
</tr>
<tr>
<td>0x7FFFD024C3DC</td>
<td>%rdi</td>
</tr>
<tr>
<td>Big</td>
<td><strong>vs.</strong> Small</td>
</tr>
<tr>
<td>~ 8 GiB</td>
<td>(16 x 8 B) = 128 B</td>
</tr>
<tr>
<td>Slow</td>
<td><strong>vs.</strong> Fast</td>
</tr>
<tr>
<td>~ 50-100 ns</td>
<td>sub-nanosecond timescale</td>
</tr>
<tr>
<td>Dynamic</td>
<td><strong>vs.</strong> Static</td>
</tr>
<tr>
<td>Can “grow” as needed while program runs</td>
<td>fixed number in hardware</td>
</tr>
</tbody>
</table>
Three Basic Kinds of Instructions

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

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

3) Control flow: what instruction to execute next
   - Unconditional jumps to/from procedures
   - Conditional branches

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

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

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

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

- Converting between integral and floating point data types *does* change the bits
  - Floating point rounding is a HUGE issue!
    - Limited mantissa bits cause inaccurate representations
    - Floating point arithmetic is NOT associative or distributive
- x86-64 is a complex instruction set computing (CISC) architecture
- **Registers** are named locations in the CPU for holding and manipulating data
  - x86-64 uses 16 64-bit wide registers
- Assembly operands include immediates, registers, and data at specified memory locations