Floating Point II, x86-64 Intro
CSE 351 Autumn 2018

Instructor: Justin Hsia
Teaching Assistants: Akshat Aggarwal
                     An Wang
                     Brian Dai
                     Britt Henderson
                     Kevin Bi
                     Kory Watson
                     Sophie Tian
                     Teagan Horkan
                     Andrew Hu
                     James Shin
                     Riley Germundson

http://xkcd.com/899/
Administrivia

- Lab 1b due Friday (10/12)
  - Submit `bits.c` and `lab1Breflect.txt`

- Homework 2 due next Friday (10/19)
  - On Integers, Floating Point, and x86-64

- Section tomorrow on Integers and Floating Point
Denorm Numbers

- Denormalized numbers \((E = 0x00)\)
  - No leading 1
  - Uses implicit exponent of \(-126\)

- Denormalized numbers close the gap between zero and the smallest normalized number
  - Smallest norm: \(\pm 1.0...0_{\text{two}} \times 2^{-126} = \pm 2^{-126}\)
  - Smallest denorm: \(\pm 0.0...01_{\text{two}} \times 2^{-126} = \pm 2^{-149}\)
    - There is still a gap between zero and the smallest denormalized number

This is extra (non-testable) material
Other Special Cases

- **E = 0xFF, M = 0:** ±∞
  - *e.g.* division by 0
  - Still work in comparisons!

- **E = 0xFF, M ≠ 0:** Not a Number (**NaN**)
  - *e.g.* square root of negative number, 0/0, ∞–∞
  - NaN propagates through computations
  - Value of **M** can be useful in debugging

- New largest value (besides ∞)?
  - **E = 0xFF** has now been taken!
  - **E = 0xFE** has largest: $1.1\ldots1_2 \times 2^{127} = 2^{128} - 2^{104}$
## Floating Point Encoding Summary

<table>
<thead>
<tr>
<th>E</th>
<th>M</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>
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...
Tiny Floating Point Representation

- We will use the following 8-bit floating point representation to illustrate some key points:
  - Assume that it has the same properties as IEEE floating point:
    - bias =
    - encoding of $-0 =$
    - encoding of $+\infty =$
    - encoding of the largest (+) normalized # =
    - encoding of the smallest (+) normalized # =
Peer Instruction Question

- Using our 8-bit representation, what value gets stored when we try to encode \(2.625 = 2^1 + 2^{-1} + 2^{-3}\)?


A. + 2.5
B. + 2.625
C. + 2.75
D. + 3.25
E. We’re lost...
Peer Instruction Question

- Using our 8-bit representation, what value gets stored when we try to encode 384 = 2^8 + 2^7?

\[
\begin{array}{|c|c|c|}
\hline
S & E & M \\
\hline
1 & 4 & 3 \\
\hline
\end{array}
\]


A. + 256

B. + 384

C. + ∞

D. NaN

E. We’re lost...
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 Exp = 0?
  - What is this “step” when Exp = 100?

- Distribution of values is denser toward zero
Floating Point Rounding

- The IEEE 754 standard actually specifies different rounding modes:
  - Round to nearest, ties to nearest even digit
  - Round toward $+\infty$ (round up)
  - Round toward $-\infty$ (round down)
  - Round toward 0 (truncation)

- In our tiny example:
  - Man = 1.001 01 rounded to M = 0b001
  - Man = 1.001 11 rounded to M = 0b010
  - Man = 1.001 10 rounded to M = 0b010

This is extra (non-testable) material
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 the specified precision (width of M)
  - Possibly over/underflow if exponent outside of range
Mathematical Properties of FP Operations

- **Overflow** yields $\pm \infty$ and **underflow** yields 0
- **Floats with value $\pm \infty$ and NaN** can be used in operations
  - Result usually still $\pm \infty$ 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:** $100\times(0.1+0.2) \neq 100\times0.1+100\times0.2$
  - $30.000000000000003553 \neq 30$
  - **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

- Two common levels of precision:
  - `float 1.0f` single precision (32-bit)
  - `double 1.0` double precision (64-bit)

- `#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)
Peer Instruction Question

- We execute the following code in C. How many bytes are the same (value and position) between `i` and `f`?
  - No voting.

```c
int i = 384;  // 2^8 + 2^7
float f = (float) i;
```

A. 0 bytes
B. 1 byte
C. 2 bytes
D. 3 bytes
E. We’re lost...
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)
Car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

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

Assembly language:
get_mpg:
    pushq %rbp
    movq %rsp, %rbp
    ...
    popq %rbp
    ret

Machine code:
0111010000011000
1000110100000100
1000100111000010
110000011111101000011111

OS:
Windows 10
OS X Yosemite

Computer system:

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Processes
Virtual memory
Memory allocation
Java vs. C
Architecture Sits at the Hardware Interface

**Source code**
Different applications or algorithms

**Compiler**
Perform optimizations, generate instructions

**Architecture**
Instruction set

**Hardware**
Different implementations

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

---

C Language

- Program A
- Program B
- Your program

Compiler

- GCC
- Clang

Architecture

- x86-64
- ARMv8 (AArch64/A64)
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
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**
Summary

- Floating point encoding has many limitations
  - Overflow, underflow, rounding
  - Rounding is a HUGE issue due to limited mantissa bits and gaps that are scaled by the value of the exponent
  - Floating point arithmetic is NOT associative or distributive

- Converting between integral and floating point data types does change the bits

- x86-64 is a complex instruction set computing (CISC) architecture