

# CSC2/452 Computer Organization

## Floating Point

Sreepathi Pai

URCS

September 19, 2022

# Outline

Administrivia

Real Numbers

The IEEE Floating Point Standards

Arbitrary Precision

# Outline

Administrivia

Real Numbers

The IEEE Floating Point Standards

Arbitrary Precision

# Announcements

- ▶ Homework #2 is out
  - ▶ Due this Wednesday IN CLASS.
- ▶ Assignment #1 out
  - ▶ Due next week, Monday Sep 26 at 7PM.
  - ▶ This assignment builds in time for getting your environment sorted out: CSUG accounts, etc.

# Outline

Administrivia

Real Numbers

The IEEE Floating Point Standards

Arbitrary Precision

# Real Numbers

- ▶  $\mathbb{R}$ 
  - ▶ infinite (just like integers)
  - ▶ but they are different infinity (uncountable)
- ▶ There are infinite real numbers between any two real numbers
- ▶ How do we represent these using a finite, fixed number of bits?
  - ▶ Say, 32 bits

# The problem

- ▶ Assume 5 bits are available
- ▶ Consider 17: 10001
- ▶ Consider 18: 10010
- ▶ Where shall we put 17.5?
  - ▶ No bit pattern "halfway" between 10001 and 10010

## One option

- ▶ Consider only deltas of 0.25, 0.5, 0.75
- ▶ Then
  - ▶ 17.00: 10001
  - ▶ 17.25: 10010
  - ▶ 17.50: 10011
  - ▶ 17.75: 10100
  - ▶ 18.00: 10101
- ▶ This is the basis of the idea of *fixed point*
  - ▶ Can't represent all numbers
  - ▶ Fixed accuracy
- ▶ Used widely in tiny computers

# Representing Real Numbers

- ▶ We cannot represent real numbers accurately using a finite, fixed number of bits
  - ▶ But do we need infinite accuracy?
- ▶ How many (decimal) digits of precision do we use?
  - ▶ In our bank accounts (before and after the decimal point?)
  - ▶ In engineering?
  - ▶ In science?

# On magnitudes

- ▶ Smallest length
  - ▶ Planck length, on the order of  $10^{-35}$  (would require 35 decimal digits)
- ▶ Smallest time
  - ▶ Planck time, on the order of  $10^{-44}$
- ▶ Width of visible universe
  - ▶ On the order of  $10^{24}$
  - ▶ Lower bound on radius of universe:  $10^{27}$

# On precision

- ▶ Avogadro's number:  $6.02214076 \times 10^{23}$ 
  - ▶ So, actually: 602214076000000000000000
- ▶  $\pi = 3.1415... \times 10^0$ 
  - ▶ NASA requires about 16 decimal digits of  $\pi$ <sup>1</sup>
  - ▶ We know about a trillion

---

<sup>1</sup><https://blogs.scientificamerican.com/observations/how-much-pi-do-you-need/>

# Scientific notation for numbers

- ▶ The scientific notation allows us to represent real numbers as:

$$\text{significant} \times \text{base}^{\text{exponent}}$$

- ▶ For Avogadro's number:
  - ▶ Significant: 6.02214076
  - ▶ Significant is scaled so always only one digit before the decimal point
  - ▶ Base: 10
  - ▶ Exponent: 23

# Binary Scientific Notation

- ▶ We can use scientific notation for binary numbers too:

$$1.011 \times 2^3$$

- ▶ Here, the number is:

- ▶  $(1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}) \times 2^3$

- ▶  $(1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0) = 11_{10}$

- ▶ Components:

- ▶ Significand: 1.011

- ▶ Base: 2

- ▶ Exponent: 3

## Binary Scientific Notation: Example #2

- ▶ Now with a negative exponent:

$$1.011 \times 2^{-3}$$

- ▶ Here, the number is:

- ▶  $(1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}) \times 2^{-3}$

- ▶  $(1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + 1 \times 2^{-6})$

- ▶  $(0.125_{10} + 0 + 0.0625_{10} + 0.03125_{10}) = 0.171875$

- ▶ Components:

- ▶ Significand: 1.011

- ▶ Base: 2

- ▶ Exponent: -3

## Some design notes

- ▶ Significand contains a radix point (i.e. decimal point or binary point)
  - ▶ But it's position is fixed: only one digit before the radix point
  - ▶ In binary scientific notation, this is always 1 (why?)
  - ▶ We don't need to store the radix point
  - ▶ So significand can be treated as an integer with an implicit radix point
- ▶ Base is always 2 for binary numbers
  - ▶ No need to store this
- ▶ Exponent is also an integer
  - ▶ Could be negative or positive or zero

## Design notes (continued)

- ▶ So (binary) real numbers can be expressed as a combination of two fields:
  - ▶ significand (possibly a large number, say upto 10 decimal digits)
  - ▶ exponent (possibly a smallish number, say upto  $44_{10}$ )
  - ▶ would allow us to store numbers with at least 10 decimal digits of precision, upto 44 decimal digits long
- ▶ We'll also need to store sign information for the significand and the exponent
- ▶ How many bits?
  - ▶ for 10 significant decimal digits? e.g. 9,999,999,999
  - ▶ for max. exponent  $50_{10}$ ?
  - ▶ plus two bits for sign (one for significand, one for exponent)

## Design notes (continued)

- ▶ How many bits?
  - ▶ for 10 significant decimal digits? e.g. 9,999,999,999: about 34 bits
  - ▶ for max. exponent 50? about 6 bits
  - ▶ plus two bits for sign (one for significand, one for exponent)
- ▶ Total:  $34 + 6 + 2 = 42$  bits
  - ▶ **Could be implemented as a bitfield**
  - ▶ But 42 is between 32 and 64, not efficient to manipulate
- ▶ What format should we use to store negative significands and exponents?
  - ▶ sign/magnitude
  - ▶ one's complement
  - ▶ two's complement
  - ▶ other?

# Bitfield Design Constraints

- ▶ Ideally should fit sign, significand and exponent in 32 bits or 64 bits
  - ▶ Easier to manipulate on modern systems
- ▶ Arithmetic operations should be fast and “easy”
- ▶ Comparison operations should be fast and “easy”
  - ▶ e.g. should not need to extract fields and compare separately
  - ▶ useful for sorting numbers
- ▶ Should satisfy application requirements
  - ▶ esp. with accuracy, precision and rounding
  - ▶ should probably be constraint #1

# Outline

Administrivia

Real Numbers

**The IEEE Floating Point Standards**

Arbitrary Precision

# IEEE 754 32-bit floating point standard

- ▶ Total size: 32-bits
  - ▶ Also called “single-precision”
  - ▶ On most systems, the C type `float` is single-precision
- ▶ Significand: 24 bits, roughly 7 significant (decimal) digits of accuracy
  - ▶ Sometimes called (wrongly) the Mantissa
- ▶ Exponent: 8 bits, from  $2^{-126}$  to  $2^{127}$  (roughly  $10^{-38}$  to  $10^{38}$  (decimal))
- ▶ Sign bit: 1 sign bit for the significand
  - ▶ What about sign bit for the exponent?
- ▶ Also supports special representations:
  - ▶ for  $+\infty$  and  $-\infty$
  - ▶ For “not-a-number” *NaN*, e.g. for representing (0/0)
  - ▶ “denormals”
- ▶ Note:  $24 + 8 + 1 = 33$ , not 32

## Representing the significand

1.100 1001 0000 1111 1101 1011

- ▶ 24 bits of significand
- ▶ *Normalized* form, only one digit before the radix point
  - ▶ Change the exponent until this is achieved (normalization)
  - ▶ That digit must be non-zero
  - ▶ Always 1
- ▶ Hence, do not need to store it!
  - ▶ Only use 23 bits for the magnitude
  - ▶ In example, only 100 1001 0000 1111 1101 1011 is stored
- ▶ Uses sign/magnitude notation (not one's or two's complement)
  - ▶ 1 bit for sign (0 for +, 1 for -)
  - ▶ 23 bits for magnitude + one always 1 implicit bit (not stored)

# Appreciating Precision

One weird trick to make money from banks:

```
#include <stdio.h>

int main(void) {
    float f;
    int i;

    f = 16777216.0;
    f = f + 3.0;

    printf("%f\n", f);
}
```

- ▶ Note that 16777216 is  $2^{24}$
- ▶ What is the value of `f` that is printed?
  - ▶ A: 16777216.0
  - ▶ B: 16777219.0
  - ▶ C: 16777220.0
  - ▶ D: something else
  - ▶ E: undefined

## More Surprises

```
#include <stdio.h>

int main(void) {
    float f;
    int i;

    f = 16777216.0;

    for(i = 0; i < 2000; i++) {
        f = f + 1.0;
        // printf("%f\n", f) // uncomment to see what is happening
    }

    printf("%f\n", f);
}
```

- ▶ What is the value of `f` that is printed?
  - ▶ A: 16777216.0
  - ▶ B: 16779216.0
  - ▶ C: something else
  - ▶ D: undefined

## This is not C specific!

- ▶ Integers in Python behave differently from C
  - ▶ Don't overflow, are always signed
  - ▶ Python's integers are like mathematical integers
- ▶ Very few languages implement mathematical reals though.
  - ▶ Most use IEEE 754
  - ▶ Python, Javascript, Java, etc.

# Rounding

- ▶ IEEE floating point rounds numbers that cannot be exactly represented
- ▶ For an operation  $\oplus$  (where  $\oplus$  could be any of *mathematical*  $+$ ,  $-$ ,  $/$ ,  $\times$ )
  - ▶ the standard says  $x \oplus y \rightarrow \text{Round}(x \oplus y)$
- ▶ Four rounding modes
  - ▶ Round towards nearest (also known as round towards even, and default)
  - ▶ Round towards zero
  - ▶ Round towards  $+\infty$
  - ▶ Round towards  $-\infty$

# What's happening

- ▶  $16777216.0 + 1.0$  is unrepresentable
  - ▶ By default, rounding mode is round to nearest
  - ▶ Nearest is  $16777216.0$
  - ▶ No change!
- ▶ Why it is also called round to even
  - ▶ If an unrepresentable value is equidistant between two representable values
  - ▶ It is not possible to say which is “nearest”
  - ▶ IEEE standard picks the *even* value between the two representable values
- ▶ This makes floating point arithmetic *non-associative*
  - ▶  $(a + b) + c \neq a + (b + c)$
  - ▶  $((a + 1.0) + 1.0) \neq (a + (1.0 + 1.0))$

# Representing the Exponent

- ▶ 8-bit wide bitfield
  - ▶ Can store 256 values
  - ▶ Must store values from -126 to 127 (that's 254 values)
- ▶ Uses *biased* representation
  - ▶ To store  $x$ , we actually store  $x + 127$  in 8 bits
  - ▶ So 127 is stored as 254
  - ▶ And -126 is stored as 1
  - ▶ No sign bit required!
  - ▶ So field actually contains values from 1 to 254 to represent -126 to 127
- ▶ The biased values 0 and 255 are used to indicate special numbers

## Why biased? Comparing exponents

Which is greater?

$$1.011 \times 2^{-3}$$

Or:

$$1.011 \times 2^{+3}$$

- ▶ Note -3 in biased notation is  $-3 + 127 = 124 = 0111\ 1100_2$
- ▶ Note 3 in biased notation is  $+3 + 127 = 130 = 1000\ 0010_2$

## Putting it altogether

- ▶ Three bit fields
  - ▶  $s$ : Significand Sign (1 bit)
  - ▶  $M$ : Significand (23 bits)
  - ▶  $E$ : Biased Exponent (8 bits)
- ▶ 6 possible ways to order them
  - ▶  $s, M, E$
  - ▶  $s, E, M$
  - ▶  $M, s, E$
  - ▶  $M, E, s$
  - ▶  $E, s, M$
  - ▶  $E, M, s$
- ▶ Out of familiarity, let's only consider those where  $s$  occupies higher bits than  $M$

# Comparing Three Formats

- ▶ Suppose you have two numbers:
  - ▶  $a = 1.100... \times 2^3$
  - ▶  $b = 1.010... \times 2^5$
  - ▶ Which is greater?
- ▶ Representation
  - ▶ Significand:  $100..._2$  for  $a$  and  $010..._2$  for  $b$
  - ▶ Exponent:  $3 + 127 = 130 = 1000\ 0010_2$  and  $5 + 127 = 132 = 1000\ 0100_2$
  - ▶ Sign is 0 for both

## Comparing Three formats (contd.)

- ▶  $s, M, E$

- ▶ 0 | 100 000 ... | 1000 0010

- ▶ 0 | 010 000 ... | 1000 0100

- ▶  $s, E, M$

- ▶ 0 | 1000 0010 | 100 000 ...

- ▶ 0 | 1000 0100 | 010 000 ...

- ▶  $E, s, M$

- ▶ 1000 0010 | 0 | 100 000 ...

- ▶ 1000 0100 | 0 | 010 000 ...

# IEEE 754 Single Precision Format

- ▶ Uses  $s, E, M$  format
- ▶ If a number  $x > y$ , then its bitwise representation  $x > y$ 
  - ▶ When sign bit is same, scan from bit 30 to 0, looking for first different bit
  - ▶ When sign bit is different, 1 in sign bit indicates less than 0 (exceptions +0 and -0)
- ▶ Can thus compare floating point numbers without having to extract bitfields!

# Representing Zero

$$0 \times 2^x$$

- ▶ Has no leading 1
- ▶ Special representation
  - ▶ Sign bit can be 0 or 1
  - ▶ Exponent is all zeroes (i.e. it appears to be  $-127$  stored biased, hence  $-126$  is lower limit)
  - ▶ Magnitude is all zeroes
- ▶ Hence:
  - ▶  $+0$ : all 32 bits are zero
  - ▶  $-0$ : sign bit is 1, but all other bits are zero



# Let's make it zero!

- ▶ Default behaviour on many systems before IEEE754
  - ▶ Underflow to zero
- ▶  $a = 1.000\ 0000\ 0000\ 0000\ 0000 \times 2^{-126}$
- ▶  $b = 1.000\ 0000\ 0000\ 0000\ 0001 \times 2^{-126}$
- ▶ What is  $a - b$ ?
  - ▶ Remember,  $a \neq b$
- ▶ What would  $x/(a - b)$ ?

# Denormals

- ▶  $a = 1.000\ 0000\ 0000\ 0000\ 0000 \times 2^{-126}$
- ▶  $b = 1.000\ 0000\ 0000\ 0000\ 0001 \times 2^{-126}$
- ▶  $a - b = 0.111\ 11111\ 11111\ 11111\ 1111 \times 2^{-126}$ 
  - ▶ Numbers of this form are called *denormals* or *subnormals*
  - ▶ They have a 0 before the radix point
- ▶ IEEE 754 specifies how to store denormals:
  - ▶  $s$ , sign as usual
  - ▶  $E$ , exponent is zero
  - ▶  $M$ , the significand is non-zero
- ▶ This allows “gradual underflow” to zero
  - ▶ Some systems detect denormals and perform arithmetic in software
  - ▶ Slow!

# Representing Infinities

0 1111 1111 000 0000 0000 0000 0000

- ▶ In the above representation,
  - ▶ Sign: 0
  - ▶ Exponent: 255
  - ▶ Significand: 0
- ▶ Exponent
  - ▶ 0 indicates either zero or a subnormal
  - ▶ 1 to 254 indicates normalized exponent -126 to 127
  - ▶ 255 indicates either infinity or NaN
- ▶ With significand zero:
  - ▶ Exponent 255 indicates  $+\infty$  or  $-\infty$  (depending on sign)
  - ▶  $-\infty < x < +\infty$  where  $x$  is any representable number

# Representing NaNs

0 1111 1111 xxx xxxx xxxx xxxx

- ▶ In the above representation,
  - ▶ Sign: 0
  - ▶ Exponent: 255
  - ▶ Significand:  $\neq 0$  (i.e. the x bits are not all zero)
- ▶ With significand non-zero:
  - ▶ Exponent 255 indicates *NaN* (not-a-number)
  - ▶ Produced by operations like  $0/0$ ,  $\infty/\infty$ , etc.
- ▶ *NaNs* propagate:
  - ▶ Any operation involving a *NaN* results in a *NaN*

# Addition in Floating Point

Add

$$a = 1.000\ 0000\ 0000\ 0000\ 0000 \times 2^3$$

to:

$$b = 1.000\ 0000\ 0000\ 0000\ 0000 \times 2^4$$

## Equalizing exponents

- ▶ Exponents for  $a$  and  $b$  are different, so equalize them
  - ▶ Shift one of them
  - ▶ The shifted representation is *internal*
  - ▶ Only the result after addition is visible
- ▶  $b = 10.00\ 0000\ 0000\ 0000\ 0000 \times 2^3$
- ▶  $a + b = 11.00\ 0000\ 0000\ 0000\ 0000 \times 2^3$
- ▶ Normalized,  $1.100\ 0000\ 0000\ 0000\ 0000 \times 2^4$

# Double-precision

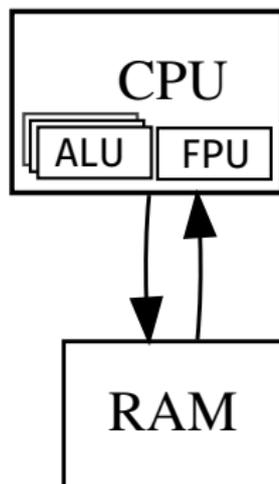
- ▶ 64-bit floating point format
  - ▶ Significand: 53 bits (52 stored), around 17 decimal digits of precision
  - ▶ Exponent: 11 bits (biased by 1023)
  - ▶ Sign: 1 bit
- ▶ In C, usually `double`
- ▶ Range:  $2^{-1022}$  to  $2^{1023}$  for normalized numbers
  - ▶ Roughly  $10^{-308}$  to  $10^{308}$

# A Programmer's View of Floating Point

- ▶ When translating algorithms from math to code, be wary
  - ▶ Computers use floats, not real numbers!
- ▶ Two major problems:
  - ▶ Non-termination (usually because exact `==` is not possible)
  - ▶ Numerical instability (approximation errors are “magnified”)
- ▶ If you deal with *computational* science or use numerics extensively, educate yourself
  - ▶ Resources at the end
  - ▶ Or take a Numerical Analysis class (primer at the end)

# How the machine supports floating point

- ▶ A math co-processor called the “floating point unit”
  - ▶ Back in the day, a separate processor
  - ▶ The Intel 387 is a classic
- ▶ For machines without a coprocessor, everything was done in software
  - ▶ Sometimes called “softfloat”
  - ▶ Still used to handle denormals on some processors
- ▶ These days, integrated into the CPU as FPU



## Some excitement these days

- ▶ Intel, NVIDIA and AMD recently announced support for FP8
  - ▶ 8-bit floating point numbers
- ▶ Follows the introduction of FP16 a few years ago
  - ▶ 16-bit floating point numbers
- ▶ Driven by the needs of deep learning programs and memory constraints
  - ▶ Also, “correctness” is not number #1 priority for DL

# Outline

Administrivia

Real Numbers

The IEEE Floating Point Standards

Arbitrary Precision

# Python Integers

- ▶ Python only has signed integers (like Java)

```
v = 1
for i in range(256):
    v = v * 2

print(v)
```

- ▶ What is the value of `v` that is printed?
  - ▶ A: Undefined
  - ▶ B:  $2^{256} \bmod 2^{64}$  (assuming 64-bit integers)
  - ▶ C:  $2^{256}$
- ▶ Reference: Python Numeric Types

# Arbitrary Precision Floating Point

The bc calculator in Linux:

```
bc 1.07.1
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017
This is free software with ABSOLUTELY NO WARRANTY.
For details type 'warranty'.
16777216.0+1.0
16777217.0

16777216.0+3.0
16777219.0

f = 16277216.0
for(i = 0; i < 2000; i++) { f += 1.0; }

f
16279216.0
```

# Summary

- ▶ Take away: floating point numbers are NOT real numbers
- ▶ Reference: Chapter 2

For further study:

- ▶ Link to An Interview with the Old Man of Floating-Point
  - ▶ IEEE754 won William Kahan the Turing Award
- ▶ Definitely read:
  - ▶ Goldberg, What Every Computer Scientist should Know about Floating-Point Arithmetic, ACM 1991
  - ▶ Stadherr, High Performance Computing, Are we just getting wrong answers faster?
  - ▶ Trefethen, Numerical Analysis