

Introduction to Computer Systems More on Numbers:

Representing Real Numbers



Lecture Objectives

- ◆ To introduce how real numbers are represented in computer systems.
- ◆ To explain some of the limitations of floating point representations



Lecture Outline

More on Numbers:

- ♦ Binary Arithmetic using 2's Complement
- ◆ Fixed Point Decimal to Binary
- ◆ Fixed Point Binary to Decimal
- ◆ Fixed Point Arithmetic
- ◆ Floating Point Numbers
- Numerical Precision

Binary Arithmetic - Observations

- ◆ Add a "sign" bit assume on the left
- ◆ Then +5 = 0101 and -5 = 1101
- ◆ We know that +5 + -5 = 0, so this should hold in binary!
- But:

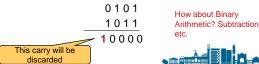


We need to be a bit smarter about this ...



Binary Arithmetic - 2's Complement

- In 2's Complement the convention is:
- All positive numbers start with 0
- All negative numbers start with 1
- Negation is achieved by:
 - Flipping all the bits
 - Adding 1 to the least significant (right-most)
- Lets see the same example again:
- +5 = 0101 and -5 = 1011 (in 2C notation)



How Real Numbers are Represented?

How do we represent real numbers in computer systems?

Two of the common ways to achieve this are:

- ◆ Fixed Point: binary point is fixed e.g. 1101101.0001001
- ◆ Floating Point: binary point floats to the right of the most significant 1 and an exponent is used e.g. 1.1011010001001 x 26

•IEEE 754 - https://en.wikipedia.org/wiki/IEEE 754



Fixed Point Decimal-to-Binary

- ◆ Integer part convert as before (repeated division by 2)
- Non-integer part follows opposite process
- Repeated multiplication by 2, keeping integer part:

0.537 x 2 = 1.074 0.074 x 2 = 0.148 0.148 x 2 = 0.296 0.296 x 2 = 0.592 0.592 x 2 = 1.184 0.184 x 2 = 0.368 So 0.537₁₀ = 0.100010₂

- 0.625?
- **♦** 0.512?

Fixed Point Binary-to-Decimal

- Allocate subset of bits to integer part, and the remainder to the non-integer part.
- ◆ For example, 4+4 bits:

1101.0101 $1x2^{3} + 1x2^{2} + 0x2^{1} + 1x2^{0} + 0x2^{-1} + 1x2^{-2} + 0x2^{-3} + 1x2^{-4}$ = 8 + 4 + 0 + 1 + 0.0 + 0.25 + 0.0 + 0.0625 = 13.3125

- **101.110?**
- **•** 010.001?

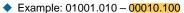


Slide #7 of 25

2's Complemen 11101.011

Fixed Point Arithmetic

Everything is the same as for whole numbers



◆ Take 2C and add:

• 00110.101 – 10110.010?

Decimal Fractions (Base 10)



Slide #9 of 25

Infinite Decimal Fractions (Base 10)

e.g.
$$\frac{1}{3} = 0.3333333333$$
If only fixed number of decimal places allowed, the rest is lost

e.g. four places:





Binary Fractions (Base 2)

Similar. e.g. $\frac{1}{12} = \frac{5}{8} = \frac{10 \cdot 100 \cdot 100}{32 \cdot 16 \cdot 8421} = \frac{101010101 \times 2^{-3}}{1000}$ $\frac{1}{100} = 0.000110011001100...$ decimal infinite binary fraction



Floating Point Representation (for fractions)

General principle

- like "scientific notation", but in binary

Numbers represented as: ±m*2e

Sign (±)	Mantissa (M)	Exponent (e)
1 bit	Actual significant digits	'Signed' binary
0 for +		'Shows' where binary point goes
1 for -		



Choice of Representations

$$\frac{425}{3} = 101010 \cdot 101 = 101010101 \times 2^{-3}$$

$$= 101010101010 \times 2^{-4}$$

$$= 10101010100 \times 2^{-5}$$

$$= 10101010101 \times 2^{0}$$

$$= 10101010101 \times 2^{0}$$

$$= 10101010101 \times 2^{5}$$

Floating Point Representation in Java

S	Offset e	Mantissa m
Sign	Exponent:	Mantissa:
0. 1 for +	Biased binary	Leading mantissa bit (1.) left out

- ◆ take bits of m
- ◆ put "1." at start, so normalized
- move binary point right e places
- (or left for negative e)
- ◆ note e is stored with an "offset" added to it



Java Types for Floating Point

No of Bits					
Туре	Sign	Mantissa	Exponent	Total	Bytes
float	1	23	8	32	4
double	1	52	11	64	8

1 extra for hidden bit

52 bit mantissa : 2⁵³ ≈ 8 x 10¹⁵

We get 15 significant decimal digits in double data type.



Offset: 2^(n-1) -1

Java Types for Floating Point $2^{(8-1)} - 1 = 127$

 $42^{5}/_{8} = 101010.101 = 1.01010101 \times 2^{5}$

	0	1000 0100	010 1010 1000 0000 0000 0000
Sign 0 for +	Sign 5 + offset 127 = 132 of for + e + 127		Mantissa Without the leading 1.

¹/₁₀ = 0.000110011001100... = 1.10011001100... x 2⁻⁴

100 1100 1100 1100 1100 1101 0111 1011 0 Sign 0 for + -4 + offset 127 = 123 Mantissa Rounded Up e + 127 Without the leading 1. Rounding Error!



Money as Floating Point?

Floating Point value for amount in pounds with pence as fraction?

Not a good practice!

Pence need infinite binary fractions e.g. 10p is 0.0001100110011001100... so we get rounding errors

Always use int or long (or BigInteger) for money.



Factorial with Double

```
* Calculate factorial.
* requires: 0 <= n
* @param n number whose factorial is to be calculated
* @return factorial of n
                                                 165, 5.423910666131586E295
public static double dfact(int n){
                                                 166 9 003691705778433E297
          double a = 1;
                                                 167, 1.5036165148649983E300
          for (int i = 1; i \le n; i++){
                                                 168, 2.526075744973197E302
169, 4.2690680090047027E304
                                                 170, 7.257415615307994E306
          return a;
                                                 171, Infinity
172, Infinity
                                                 173, Infinity
                                                 174. Infinity
```

Accuracy Issues - Even in Double Floating Point

```
n = 170

n! = 7.257415615307994E306

= 7257415615307994E291

This digit is wrong! 291 more digits after it
```

Windows calculator says:

7.2574156153079989673967282111293e+306

- Floating point arithmetic loses accuracy in least significant digits.
- Most significant, and overall size, are OK.



Why is 171! too big?

170!
$$\approx$$
 7.2 \times 10³⁰⁶

171! \approx 171 \times 7.2 \times 10³⁰⁶
 \approx 1231 \times 10³⁰⁶
 \approx 10³⁰⁹ = (10³)¹⁰³ \approx 2¹⁰³⁰

Needs binary exponent \approx 1030 $>$ 1024 = 2¹⁰

Exponent too big to fit in 11 bits (signed) for Java double



Floating Point Overflow

More precisely:
In Java: Float.POSITIVE_INFINITY
or Double.POSITIVE_INFINITY
it's a special value POSITIVE_INFINITY

Other special values:

If too big but negative: NEGATIVE_INFINITY

Indistinguishable from 0 but known to be negative: -0.0 Impossible number (e.g. Sqrt(-1)) NaN "Not a Number"

These special values allow you to check for overflow in a program

- unlike the case for integer arithmetic



Summary – Java Floating Point

- ◆ It normalizes where possible
- Unnormalised for smallest numbers (offset e = o)
- Special representations for NaN etc.
- Details in API for:
- java.lang.Float.intBitsToFloat
- java.lang.Double.longBitsToDouble

Numerical Precision

 Fixed point is convenient and intuitive but has two problems

1)Numerical precision

•Only values that are an integer multiple of the smallest power of two can be represented exactly

2)Numerical range

•Increased precision of non-integer part is at the expense of numerical range

 Floating point representation effectively addresses these issues.





Summary - Numbers

- Representing numbers in the computer
- •Whole numbers in binary and hexadecimal notation
- •Positive real numbers in fixed-point binary
- Binary arithmetic is like decimal arithmetic
- •Our lack of binary practice makes it hard!
- Negative numbers are tricky things
- ◆But we can use a few of our own tricks 2C
- Floating point is an alternative, but is very unnatural for us!

