Introduction to Computer Systems
More on Numbers:
Representing Real Numbers

Lecture Objectives

To introduce how **real numbers** are **represented** in computer systems, and 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

$+5 = 0101$

$-5 = 1101$

10010

- 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:

$$\begin{array}{r} 0\,1\,0\,1 \\ 1\,1\,0\,1 \\ \hline \mathbf{1}\,0\,0\,\mathbf{1}\,0 \end{array}$$

We need to Fix It!

$+2$

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

Binary Arithmetic – 2’s Complement

$-5 = 1011$

- 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)

$$
\begin{array}{r}
0101 \\
1010 \\
+1 \\
\hline
1011
\end{array}
= -5
$$

$$
\begin{array}{rr}
+5 \rightarrow & 0\,1\,0\,1 \\
-5 \rightarrow & 1\,0\,1\,1 \\
\hline
 & \mathbf{1}\,\underbrace{0\,0\,0\,0}
\end{array}
$$

This carry will be discarded

How about Binary Arithmetic? Subtraction etc.

How Real Numbers are Represented?

How do we represent real numbers in computer systems?

Two of the common ways to achieve this are:

7.9

- **Fixed Point**: binary point is fixed e.g. 1101101.000100101
- **Floating Point**: binary point floats to the right of the most significant 1 and an exponent is used e.g. $1.101101000100101 \times 2^{6}$

• IEEE 754 - https://en.wikipedia.org/wiki/IEEE_754

floating after the first bit

Fixed Point Decimal-to-Binary

$(5.537)_{10}$

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

$$0.537 \times 2 = \mathbf{1}.074$$
$$0.074 \times 2 = \mathbf{0}.148$$
$$0.148 \times 2 = \mathbf{0}.296$$
$$0.296 \times 2 = \mathbf{0}.592$$
$$0.592 \times 2 = \mathbf{1}.184$$
$$0.184 \times 2 = \mathbf{0}.368$$

So $0.537_{10} = 0.100010_2$

$(101.100010)_2$

~~3~~.6

$\boxed{10.6}$

- 0.625?
- 0.512?

$(0.100010)_2 \rightarrow (?)_{10}$

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:

$\frac{1}{2}$ $\frac{1}{4}$ $\frac{1}{8}$

$$1101.0101$$

$$1 \times 2^{3} + 1 \times 2^{2} + 0 \times 2^{1} + 1 \times 2^{0} + 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4}$$

$$= 8 + 4 + 0 + 1 + 0.0 + 0.25 + 0.0 + 0.0625$$

$$= 13.3125$$

- 101.110?
- 010.001?

00010.100

11101.011

+1

Fixed Point Arithmetic

- Everything is the same as for whole numbers
- Example: 01001.010 – 00010.100
- Take 2C and add:

0 1 0 0 1 . 0 1 0	(9.25)
+ 1 1 1 0 1 . 1 0 0	(-2.5)
(1) 0 0 1 1 0 . 1 1 0	(6.75)

- 10110.101 – 00110.010 ?

Decimal Fractions (Base 10)

e.g. $\rightarrow$ 42.625

$$\begin{array}{cc} 4 & 2 \\ \downarrow & \downarrow \\ 10 & 1 \end{array} . \begin{array}{ccc} 6 & 2 & 5 \\ \frac{1}{10} & \frac{1}{100} & \frac{1}{1000} \end{array}$$

$$4 \times 10 + 2 \times 1 + \frac{6}{10} + \frac{2}{100} + \frac{5}{1000}$$

$$= 42625 \times 10^{-3}$$

$$= 42\frac{625}{1000} = 42\frac{25}{40} = 42\frac{5}{8}$$

Infinite Decimal Fractions (Base 10)

e.g. $\frac{1}{3} = 0.3333|333333\ldots$

If only fixed number of decimal places allowed, the rest is lost

e.g. four places:

$\boxed{0.3333}\underbrace{333333\ldots}_{\text{rounding error}}$

Binary Fractions (Base 2)

Similar. e.g.

$$42\frac{5}{8} = \boxed{101010.101} = \underbrace{101010101}_{\text{binary}} \times 2^{-3}$$

32 16 8 4 2 1 $\frac{1}{2}$ $\frac{1}{4}$ $\frac{1}{8}$

$$\frac{1}{10} = \underbrace{0.000\boxed{1100}\boxed{1100}\boxed{1100}\ldots}_{\text{infinite binary fraction}}$$

decimal

Floating Point Representation (for fractions)

General principle

- like "scientific notation", but in binary

Numbers represented as: $\pm m*2^e$

Sign (±)	**Mantissa (m)**	**Exponent (e)**
1 bit 0 for + 1 for –	Actual significant digits	2s complement signed binary Shows where binary point goes

Choice of Representations

$$42\frac{5}{8} = 101010.101 = 101010101 \times 2^{-3}$$

$$= 1010101010 \times 2^{-4}$$

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

$$= \dots$$

or

$$= 101010.101 \times 2^{0}$$

$$= 10101.0101 \times 2^{1}$$

$$= \dots = 1.01010101 \times 2^{5}$$

Floating Point Representation in Java

S	Offset e	Mantissa m

Sign
0, 1 for +, -

Exponent
2's complement
signed binary

Mantissa
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					
Type	Sign	Mantissa	Exponent	Total	Bytes
float	1	23	8	$\boxed{32}$	4
double	1	52	11	64	8

1 extra for hidden bit

52 bit mantissa : $2^{53} \approx 8 \times 10^{15}$

$2^{53} \Rightarrow 2^3 * 2^{50}$

We get 15 significant decimal digits in double data type.

$8 \times (2^{10})^5$

$10^3 \rightarrow 8 \times 10^{15}$

5+127=132

Java Types for Floating Point

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

0	**1000 0100**	**010 1010 1000 0000 0000 0000**

Sign
0 for +

5 + offset 127 = 132
e + 127

Mantissa
Without the leading 1.

$${}^{1}/_{10} = 0.000110011001100...$$
$$= 1.10011001100... \times 2^{-4}$$

0	**0111 1011**	**100 1100 1100 1100 1100 1101**

Sign
0 for +

-4 + offset 127 = 123
e + 127

Mantissa
Without the leading 1.

Rounded Up
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 0.00011001100110011100...

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
*/
public static double dfact(int n){
        double a = 1;
        for (int i = 1; i <= n; i++){
                a = a * i;
        }
        return a;
}
```

n, n!
165, 5.423910666131586E295
166, 9.003691705778433E297
167, 1.5036165148649983E300
168, 2.526075744973197E302
169, 4.2690680090047027E304
170, 7.257415615307994E306
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:

7.257415615307998**8**967396728211129**3**e+306

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

Why is 171! too big?

2^{10}

$170! \approx 7.2 \times 10^{306}$

$171! \approx 171 \times 7.2 \times 10^{306}$

$\approx 1231 \times 10^{306}$

$\approx 10^3$

$\approx 10^{309} = (10^3)^{103} \approx 2^{1030}$

Needs binary exponent $\approx 1030 > 1024 = 2^{10}$

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

Floating Point Overflow

More precisely:
Float.POSITIVE_INFINITY
or
Double.POSITIVE_INFINITY

In Java:

If result is too big for datatype,

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

IEEE 754

- 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!