CSC207 Week 11

Larry Zhang

Logistics

Make A3 groups

Floating Point

Based on materials from CSC207 Fall 2016 St George

DEMO #1:

Adding.java

Brace Yourself

What just happened?

- To understand it, we need to have clear picture of how numbers are really stored inside a computer.
- Integers (like 207, 42): easy
- numbers with fractional parts (like 3.14159): trickier

Integers

- Integers are stored as **binary** numbers in a computer.
  - byte (8-bit binary)
  - short, char (16-bit binary)
  - int (32-bit binary)
  - long (64-bit binary)
- The range of values depends on the sizes of the integer (# of bits)
- What's the binary representation of `short x = 207` ?
- `0000 0000 1100 1111`

$$207 = 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

Fractions

- Everything is still binary in a computer.
- What is the decimal value of binary number **0.0111**?

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

$$= 0/2 + 1/4 + 1/8 + 1/16$$

$$= 0.25 + 0.125 + 0.0625$$

$$= 0.4375$$

Finite representation or not: **Decimal**

- Recall in decimal, not all fractions can be represented using a finite number of decimal digits. Example?
  - 1/3 = 0.3333333333….
  - 1/7 = 0.1428571428571428….
  - What’s the necessary and sufficient condition for a fraction to be representable with a finite number of decimal digits?
  - if and only if the value can be written as a sum like the following

$$v = \sum_{i=1}^{n} d_i \cdot 10^{-i}$$

Finite representation or not: **Binary**

It's very similar to decimal numbers, except that the base is changed from 10 to 2

A fraction has a finite binary representation iff it can be written as the following sum.

$$v = \sum_{i=1}^{n} d_i \cdot 2^{-i}$$

Examples:

- 0.5 in binary, finite or not?
- 0.25 in binary, finite or not?
- 0.75 in binary, finite or not?
  - 0.75 = ½ + ¼, (0.11)
- 0.8125 in binary, finite or not?
  - 0.8125 = ½ + ¼ + 1/16, (0.1101)

- 0.1, i.e., 1/10, finite or not?
  - **infinite!**
  - **0.0001100110011001100110011001...**

A very sad fact about computers ...

is that all of them can only have a **finite** amount of memory.

so it is **impossible** to for a computer to store the **accurate value** of a number with infinite binary representation.

we have only a certain number of bits we can use (like 32, 64) store an **approximation** of the value with a **certain precision**.

Say we have **32 bits** that we can use to represent this number,

how do we use these 32 bits?

Idea #1

Among the 32 bits, we use the higher 16 bits for the **whole-number part** and 16 bits for the **fractional** part. For example,

`0101 1111 0001 0010 . 0101 0111 1111 0001`

`(decimal value: 24338.3435211181640625)`

What's the smallest fraction we can represent using this format?

0. 0000 0000 0000 0001 (1/2^16 = 0.000015258789063)

Want to store 0.000000001 ? No can do! And this does not make sense because this number is so simple!

This must be a better way!

Back to decimal ...

When we want to write a number like 0.000000000000000000000000000217, we usually don't write it like this. We often use a more **compact** form …

Scientific Notation

2.17 x 10^(-33) or **2.17e-33**

Exponent

Mantissa: 1.0 <= v < 10.0, with exactly one digit to the left of the decimal point.
(Note: the first digit must be between 1 and 9)

Back to binary ...

If we use “scientific notation” for binary numbers, we will be able to represent much smaller numbers than what we had in Idea #1.

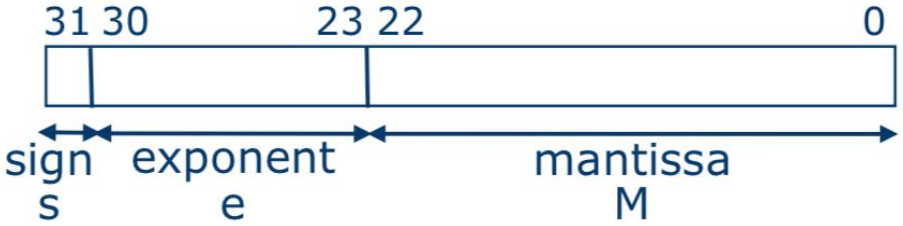
-6.84 is written as -1.71×2^{2}

0.05 is written as 1.6×2^{-5}

Note: The integer part (first digit) of the mantissa is always 1. Why?

IEEE-754 Floating Point Format

Use the 32 bits like this:



- **1 bit** for the **sign**: 1 for negative and 0 for positive
- **8 bits** for the **exponent** e
  - To allow for negative exponents, 127 is added to that exponent to get the representation. We say that the exponent is "biased" by 127.
  - So the range of possible exponents is not 0 to 2^8-1 = 0 to 255, but (0-127) to (255-127) = -127 to 128.
- **23 bits** for the **mantissa** M
  - Since the first bit must be 1, we don't waste space storing it!

$$(-1)^s * (1 + M) * 2^{e-127}$$

Single Precision vs Double precision

In Java, this data type is called **float**.

In Java, this data type is called **double**.

A bit history

- 30 years ago, computer manufacturers each had their own standard for floating point.
- Problem? Writing portable software!
- Advantage to manufacturers? Customers got locked in to their particular computers.
- In the late 1980s, the IEEE produced the standard that now virtually all follow.
- William Kahan spearheaded the effort, and won the 1989 Turing Award for it.

William Kahan

Bachelor, Master, PhD

from University of Toronto

Example

conversion from binary to decimal

$$(-1)^{\text{sign}} \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right) \times 2^{(e-127)}$$

- $\text{sign} = 0$
- $1 + \sum_{i=1}^{23} b_{23-i} 2^{-i} = 1 + 2^{-2} = 1.25$
- $2^{(e-127)} = 2^{124-127} = 2^{-3}$

thus:

- $\text{value} = 1.25 \times 2^{-3} = 0.15625$

Example: Conversion from decimal to binary

Let's convert 0.085 to IEEE-754 single precision floating point binary number

- Write 0.085 in base-2 scientific notation
  - 0.085 = 1.36 / 16 = 1.36 x 2^(-4)
- Determine the sign bit
  - it's 0 because positive number
- Determine the exponent
  - -4 = 123 - 127, so it's 8-bit binary for 123: 01111011
- Determine the mantissa
  - convert 0.36 to binary by repeatedly multiplying by 2
  - keep 23 bits, we get 0.01011100001010001111011

Finally, the result is: 0[01111011]01011100001010001111011

```
0.36 x 2 = 0.72
0.72 x 2 = 1.44
0.44 x 2 = 0.88
0.88 x 2 = 1.76
0.76 x 2 = 1.52
0.52 x 2 = 1.04
0.04 x 2 = 0.08
0.08 x 2 = 0.16
0.16 x 2 = 0.32
0.32 x 2 = 0.64
0.64 x 2 = 1.28
0.28 x 2 = 0.56
0.56 x 2 = 1.12
0.12 x 2 = 0.24
0.24 x 2 = 0.48
0.48 x 2 = 0.96
0.96 x 2 = 1.92
0.92 x 2 = 1.84
0.84 x 2 = 1.68
0.68 x 2 = 1.36
0.36 x 2 =  ...
```

Rounding

- If we have to lose some digits, we don't just truncate, we round.
- In rounding a decimal to a whole number, an issue arises: If we have a 0.5, do we round up or down?
- If we always round up, we are biasing towards higher values.
- "Proper" rounding: round to the nearest even number.
  E.g., 17.5 is rounded up to 18 but 16.5 is rounded down to 16.
- The IEEE standard uses proper rounding also.

Some special values

```
0[00000000]00000000000000000000000: Zero

0[11111111]00000000000000000000000: Positive infinity

1[11111111]00000000000000000000000: Negative infinity

*[11111111]-anything-but-all-zero-: Not a Number (NaN)
```

Overflow

Definition: **overflow** is the largest representable number

For IEEE-754 single precision, it is

0[11111110]11111111111111111111111

= +1.11111111111111111111111 x 2^(127)

= overflow

Underflow

Def: Underflow is the smallest positive representable number

Example: For single precision, it appears that underflow is

0[00000001]00000000000000000000

= 1.00000000000000000000 x 2^(-126)

But not really, IEEE-754 allows **denormalized** numbers (numbers that do not follow scientific notation), so the **real underflow** is:

0[00000000]00000000000000000001

= 0.00000000000000000001 x 2^(-126)
= 1 x 2^(-126-23) = 1 x 2^(-149)

From IEEEout.txt (on course website)

```
0[00000011]00000000000000000000000=+1.00000000000000000000000x2^(-124)=4.7019774E-38

0[00000010]00000000000000000000000=+1.00000000000000000000000x2^(-125)=2.3509887E-38

0[00000001]00000000000000000000000=+1.00000000000000000000000x2^(-126)=1.17549435E-38

0[00000000]10000000000000000000000=+0.10000000000000000000000x2^(-126)=5.877472E-39

0[00000000]01000000000000000000000=+0.01000000000000000000000x2^(-126)=2.938736E-39

0[00000000]00100000000000000000000=+0.00100000000000000000000x2^(-126)=1.469368E-39

denormalized
```

Back to DEMO #1:

`Adding.java`

Machine Epsilon

Definition: **Machine Epsilon** (eps) is such that 1 + eps is the smallest possible mantissa you can get that is > 1

Machine Epsilon is the best precision you can have in the mantissa.

For single precision, $eps = 1 \times 2^{-23} \approx 1.19e-7$

i.e., if you add 1.0 by 1e-7, nothing's gonna change.

For double precision, $eps = 1 \times 2^{-52} \approx 2.22e-16$

DEMO #2:

`Totalling.java`

Takeaways

- 0.1 cannot be represented exactly in binary so we just an approximated value, that leads to the unexpected result.
- And adding a very small quantity to a very large quantity can mean the smaller quantity falls off the end of the mantissa.
- But if we add small quantities to each other, this doesn’t happen. And if they accumulate into a larger quantity, they may not be lost when we finally add the big quantity in.

DEMO #3:

`ArrayTotal.java`

Takeaways

- When adding floating point numbers, add the smallest first.
- More generally, try to avoid adding dissimilar quantities.
- Specific scenario: When adding a list of floating point numbers, **sort** them first.

DEMO #4:

`LoopCounter.java`

Takeaways

- Don’t use floating point variables to control what is essentially a counted loop.
- Also, use fewer arithmetic operations where possible.
  - fewer operations means less error being accumulated
- Avoid checking equality between two numbers using “==”
  - don’t check this condition: `x == 0.207`
  - check this: `(x >= 0.207-0.0001) && (x <= 0.207+0.0001)`
  - `or check this: abs(x - 0.207) <= 0.0001`

DEMO #5:

`Examine.java`

Takeaway

- What are all these extra digits??
- `4/5 = 1.10011001 10011001 10011001 1001100...x 2^(-1)`
- This gets rounded to `1.10011001 10011001 1001101 x 2^(-1)`
- When we print, it gets converted back to decimal, which is:
  `0.8000000119209289550781250000`
- However, the best precision you have here is just

  `2^{-23}*2^{-1} ≈ 6e-8`
- Only the 7 blue digits are significant
- **Don’t print more precision in your output than you are holding.**

A General Takeaway

Use **double** instead of **float** whenever possible.

Why does this matter?

Patriot missile accident

- In 1991, an American missile failed to track and destroy an incoming missile. Instead it hit a US Army barracks, killing 28.
- The system tracked time in tenths of seconds. The error in approximating 0.1 with 24 bits was magnified in its calculations.
- At the time of the accident, the error corresponded to 0.34 seconds. A Patriot missile travels about half a km in that time.

http://www-users.math.umn.edu/~arnold/disasters/patriot.html

Ariane 5 rocket explosion

- In 1996, the European Space Agency's Ariane 5 rocket exploded 40 seconds after launch.
- During conversion of a 64-bit to a 16-bit format, overflow occurred: the number was too big to store in 16 bits.
- This hadn't been expected because the data (acceleration reported by sensors) had never been this large before. But this new rocket was faster than its predecessor.
- $7 billion of R&D had been invested in this rocket.

https://around.com/ariane.html

Sinking of an oil rig

- In 1992, the Sleipner A oil and gas platform sank in the North Sea near Norway.
- Numerical issues in modelling the structure caused shear stresses to be underestimated by 47%.
- As a result, concrete walls were not built thick enough.
- Cost: $700 million

http://www-users.math.umn.edu/~arnold/disasters/sleipner.html

“95% of folks out there are completely clueless about floating-point.”

- *James Gosling*

If you’re interested in this kind of stuff

Consider taking CSC336 / CSC338: Numerical Methods

Reference

http://www.oxfordmathcenter.com/drupal7/node/43

https://www.youtube.com/watch?v=PZRI1IfStY0

https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Data/float.html