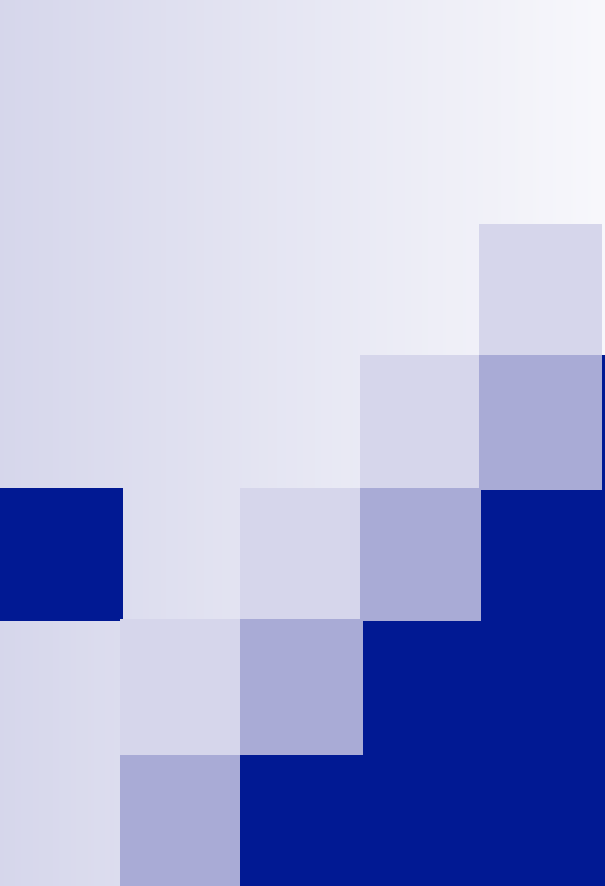




Floating Point Numbers

ICS312
Machine-Level and
Systems Programming

Henri Casanova (henric@hawaii.edu)

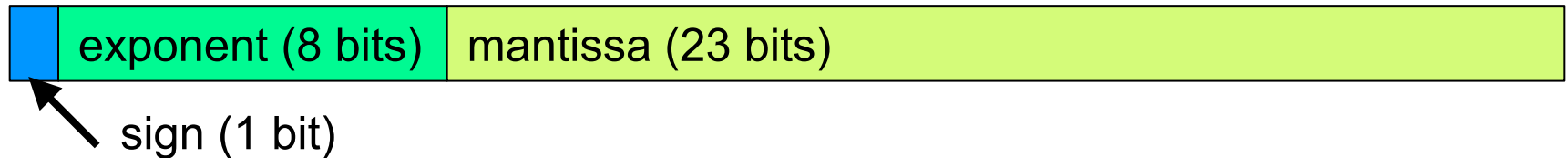
Why Floating Point Numbers?

- You cannot do everything (or at least conveniently) with integers because they can't represent very small/large values
- In many fields you need real numbers
 - Scientific computing, graphics, machine learning, etc.
- But we cannot represent real numbers in finite precision arithmetic
 - Some numbers are irrational, like $\sqrt{2}$, or even transcendent, like π
 - Some numbers are rational but cannot be represented with finite numbers of digits, like $1/3$
- So, although we want real numbers, instead, we get an **approximation**: floating point numbers

The IEEE 754 Format

- About 40 years ago IEEE defined a format for encoding floating point numbers
 - Implemented by all CPUs
 - Some variations on GPUs
 - ML accelerators like TPUs deviate (more range, less precisions), but the overall structure is the same
- There is a format for 32-bit FP numbers (aka “float”) and for 64-bit FP numbers (aka “double”)
- Since in this course we assume a 32-bit architecture I’ll only consider 32-bit FP numbers
 - The format is the same for 64-bit numbers, just “bigger”

The 32-bit IEEE 754 Format



$$-1^{\text{sign}} \times 1.\text{mantissa} \times 2^{\text{exponent} - 127}$$

- Note the leading 1 before the mantissa
 - Saves 1 bit of encoding by writing all numbers in the form 1.xxxxx
- Both mantissa and exponent are, of course, in binary!
- The exponent is weirdly “biased” by the -127 term
 - The exponent is ALWAYS stored as a non-negative integer between 0 and 255 (and then 127 is subtracted from it so that we can have negative/positive exponents)
 - This is a “hack” to allow fast comparison of FP numbers: comparing FP numbers of the same sign as if they were integers does the right thing!

Example

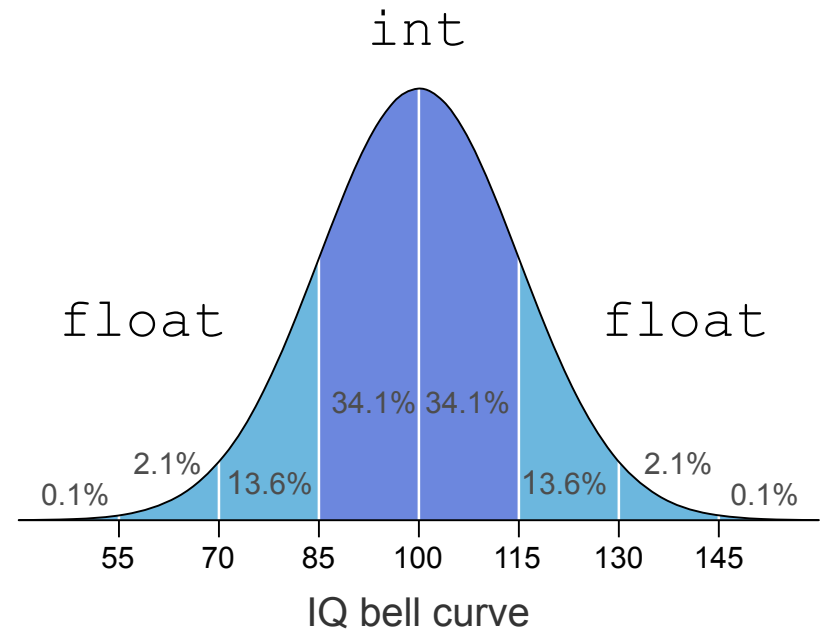
- Let's come up with the representation of -6.75
- The sign bit is 1, since the number is negative
- Let's convert 6.75 to binary:
 - $6_{10} = 110_2$
 - $0.75_{10} = 0.11_2$
 - Like the systematic algorithm to convert base-10 integers to binary, but multiply by 2 instead of dividing by 2
 - $0.75 * 2 = 1.5$; $0.5 * 2 = 1.0$; $0 * 2 = 0 \implies 0.11$
 - Therefore: $6.75_{10} = 110.11_2$
- "Normalizing": $110.11_2 = 1.1011_2 * 2^2$
- Exponent = $2 + 127 = 129 = 10000001_2$
- Mantissa = 1011 followed by 19 zeros
- Result: 1 10000001 101100000000000000000000

Special Values

Value	Representation	Meaning
± 0	Sign = 0 or 1 Exponent = 0...0 Mantissa = 0...0	Two representations of zero, but they compare equal
$\pm \infty$	Sign = 0 or 1 Exponent = 1...1 Mantissa = 0...0	Produced as a result for, e.g., a division by zero
NaN	Exponent = 1...1 Mantissa != 0...0	Produced as a result for $\sqrt{-1}$, $0/0$, etc

FP Numbers are Not Uniform

- Integers are evenly spaced
- FP numbers are denser near zero, and pretty sparse for large magnitudes
 - The gap between two consecutive numbers grows as numbers get large



- **Consequence:** Adding a very small number to a large number may do nothing!

FP “Math” is Scary

- The fact that FP numbers are just an approximation of real numbers leads to all kinds of problems
- The well-known example: 0.1_{10} cannot be represented exactly as an FP number
 - Just like $1/3$ cannot be represented exactly in decimal
- So, when in your code you have something like:

```
float x = 0.1;
```

- turns out the value is “a little bit off”!!!
- Let's try this in python:

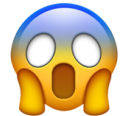
```
x = 0.1  
format(x, '.55f')
```

- The number you wrote is not the one you get!

FP “Math” is Scary

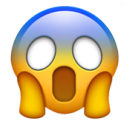
- What we saw in the previous slide has enormous implications
- Equality tests can fail:

```
>>> 0.1 + 0.2 == 0.3  
False
```



- Larger numbers are lesss “reliable”

```
>>> x = 10000000000000000  
>>> math.sqrt(x) == math.sqrt(x-1)  
True
```



FP “Math” is Scary

- Even associativity can fail:
- In C:

```
#include <stdio.h>
int main() {
    float a = 1e8f, b=-1e8f, c=1.0f;
    printf("--> %f\n", (a+b)+c);
    printf("--> %f\n", a+(b+c));
}
```

- The above program outputs:

```
--> 1.000000
--> 0.000000
```





Bottom-Line

FP numbers are NOT real numbers

- There has been a lot of work about coping with the limitations of FP numbers
 - Some of it very theoretical/difficult
- Let's look at common techniques you should know...

Comparison: use an epsilon

■ Never do: `if (a == b) . . .`

but instead: `if (|a - b| < epsilon) . . .`

- Choosing the right epsilon is not easy
 - Should be larger for larger numbers :(
- Some languages provide built-in ways to do this

Python `math.isclose(0.1 + 0.2, 0.3)`

Java `Double.compare(0.1 + 0.2, 0.3);`



Re-order Computations

- There is a lot of “lore” there
- For instance, if summing a list of FP numbers it’s much more accurate to start with the smaller numbers
- There are some well-known algorithms to minimize error (and to have guarantees about the maximum errors)
- Bottom-line: if you get weird results with FP numbers, you may be able to “fix it” by re-ordering computations
 - We could have a whole semester of content here :)



Infinite Precision?

- Perhaps you MUST have no error at all
 - Finance, Cryptography, symbolic math
- The way to go is to use arbitrary/infinite precision arithmetic
- But this is implemented in software, and thus its very, very slow (could be 1000x slower than hardware FP arithmetic)
- Available in most languages:
 - Python's `decimal` package
 - Java's `BigDecimal` class
 - etc.

Horror Stories

- 1982: Vancouver Stock Exchange
 - Accumulated error lead to an index having “lost” half its value!
- 1991: US Patriot missile
 - Accumulation of FP error after hours of operation lead to a US barrack being hit with loss of life
 - This was because of 0.1 being non-exact!
- 1996: Ariane crash
 - FP numbers cast to an integer, leading to overflow, leading to a rocket self-destruct (millions of \$ lost)
- And so on....



Takeways

- Floating points numbers are approximations of real numbers
- The format is: sign / exponent / mantissa
- Math based on FP numbers can be wrong:
 - Some values are only approximations (0.1)
 - The number of wrote is not always the number you get
 - Some results are simply wrong
- Solutions:
 - Epsilon-like techniques, operation re-ordering
 - Use your language's built-in features if any
 - Infinite precision, but it's slooooooow



Conclusion

- Almost every developer/system hits some FP bug at some point
- It's good to be warned/aware
- We'll have an **in-class quiz** on this module and the previous Control Structures module
- This completes our material for **Midterm #2**