SPECTRAL:
TECHNOLOGIES

WEEK 6

Floating-Point Numbers

Why bother

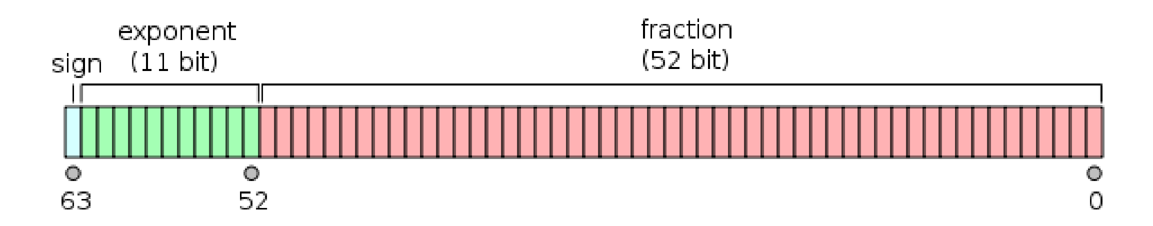
- There are a few tricky spots you need to be aware of
- Some properties of floating-point numbers are unintuitive
- So there are a lot of myths around fp

SPECTRAL:
TECHNOLOGIES

Floating-Point Number

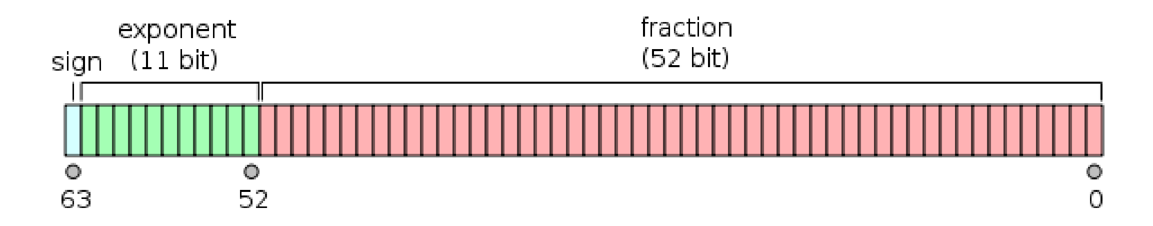
- The general form is $(-1)^s \cdot F \cdot B^e$
- On real machines $B = 2$, so any fp is $(-1)^s \cdot F \cdot 2^e$
- s repsents sign (1 or 0) and $F = \pm f_0. f_1 \dots f_{p-1}$, each f_i is 0 or 1

  F is called *mantissa*
- Note, that numbers may have multiple representations in such form, thats why its *normalized*, i.e $f_0 = 1$
- Some numbers cannot be represented exactly, i.e. 0.1
- Since numbers are normalized, there is no need to store f_0. Exponent is *biased*, since want to be able to store both positive and negative exponents
- So the actual form (for double) is $(-1)^s(1.f_{51}f_{50}\dots f_0)\cdot 2^{e-1023}$



Special cases

- Note, that 0 cannot be represented, so $e = 0$ is reserved and is a special case for 0. So the smallest exponent for normal numbers is 2^{-1022}. And $e = 0$ and $F = 0$ for 0
- $e = 0$ is also reserved for *denormalized* numbers (lets omit them for now)
- $e = 11111111111_2$ with $F = 0$ is reserved for ∞
- $e = 11111111111_2$ with $F \neq 0$ is reserved for NaN



Special cases

- There is positive and negative 0 (treated as equal, but the sign carries over computations)
- You can get $\pm\infty$ as a result of $\frac{x}{0}$ overflow or with $x \neq 0$
- There are a few ways to get NaN, e.g. $\frac{0}{0}$ or $\frac{\pm\infty}{\pm\infty}$. Any operation with NaN produces NaN and NaN $\neq$ NaN
- Its possible to setup traps to handle some of these situations as errors

SPECTRAL:
TECHNOLOGIES

Rounding rules

- The result of any computation with fp is calculated as accurate as possible internally (additional bits are used)
- Then its rounded to the nearest representable number (in case of a tie preferring the number that ends with a zero)
- We can use units in the last place (ulps) to measure errors: distance between two numbers in terms of how many representable numbers can fit between the precise real value and the actual result of the computation
- On practice its more convinient to use relative error

SPECTRAL:
TECHNOLOGIES

Rounding rules

- It can be shown, that the worst relative error for any fp computation happens when rounding is applied to numbers of the form $1 + a$, where $a \in [0, 2^{-p})$, where is a width of mantissa
- This number is denoted as *machine epsilon*, so for doubles $\epsilon = 2^{-53}$
- So for any arithmetic operation with result , the real value lies in the interval $[x \cdot (1 - \epsilon), x \cdot (1 + \epsilon)]$
- Now this interval can be applied to bound the relative error of any complex computation
- Lets calculate $f(x, y) = x^2 - y^2$ (assume $x > y$)
  - direct computation gives a relative error $O(\epsilon|x|)$
  - $(x - y)(x + y)$ gives relative error $O(\epsilon|x - y|)$

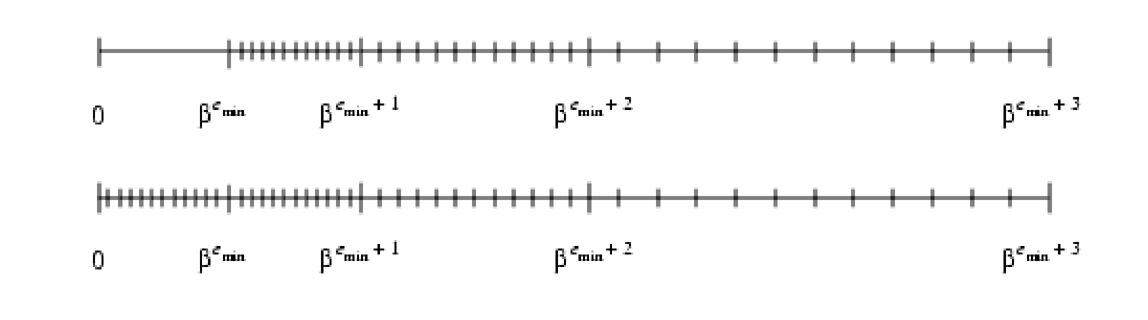
SPECTRAL:
TECHNOLOGIES

Rounding rules

- Multiplication and division are much safer in terms of accuracy
  - The relative error of product of n numbers can be bound by $O(n\epsilon)$
- Avoid summation and substraction of numbers of different magnitudes
- This is the worst case, summing n numbers may lead to huge errors
  - *Kahan Summation* may be used to get more accurate results
- Due to the rounding operations are not associative or distributive
  - $a + (b + c) \neq (a + b) + c$
  - $a(b + c) \neq ab + ac$
  - and so on
- The calculations are always deterministic, but if you have any parallelism, the order of threads executing is usually not determined

Denormalized numbers

- With the current normal form $(-1)^s 1.F \cdot 2^{e-bias}$ some calculations with close numbers will be flushed to zero
  - In fact $x = y \Leftrightarrow x - y = 0$ doesn't hold
- The gap between 0 and the smallest normalized number is bigger, then the gap after it
- Denormalized numbers has the form $(-1)^s 0.F \cdot 2^{-bias}$ with $F \neq 0$



Denormalized numbers

- FPU is optimized for handling normal values
- On x86-64 denormalized values are handled by hardware, but still quite slow (~ **x100** times slower)
- On some CPUs its left for software to emulate them, which is even slower
- There are ways to flush them to 0

Comparing fp numbers

- We can't compare `double x` and double y as `x != y` due to the rounding errors
- One way is to use absolute error `abs(x - y) < eps`, where `eps` is some small number
- Another way is to use relative error
- Sometimes integer representation may be used

SPECTRAL:
TECHNOLOGIES

FP and integers

- The binary form we've described before has an interesting property $x < y \Leftrightarrow bp(x) < bp(y)$, where $bp(x)$ is a binary form of x
- So for `double x`

```
double NextDouble(double x) {
    uint64_t val = *reinterpret_cast<uint64_t*>(&x) + 1;
    return *reinterpret_cast<double*>(&val);
}
```

gives the next representable floating-point number in doubles

- This can be used to compare numbers based on ulps
- You can convert binary search over the real interval $[l, r]$ to the integer binary search over $[bp(l), bp(r)]$

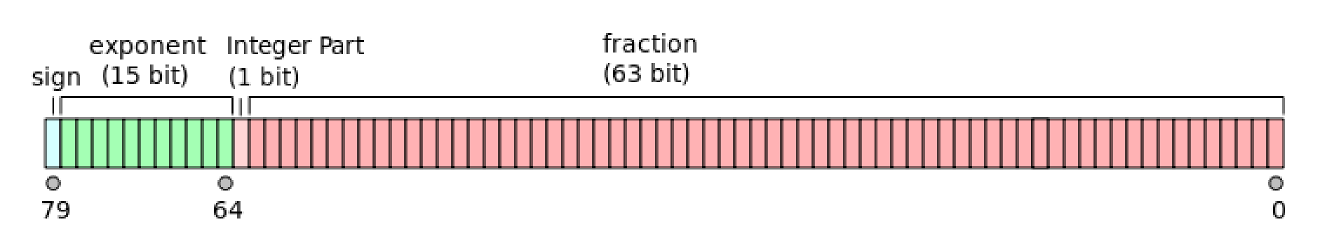
FP and integers

- For double integers between -2^{53} and 2^{53} are represented exactly
- long double (where supported) can represent all 64-bit integers

SPECTRAL:
TECHNOLOGIES

long double

- On x86-64 all fp arithmetic (both float and double) is usually performed with xmm registers, which are registers used for SSE. So SSE instructions may be used even for single operations
- But x86-64 also supports extended 80-bit precision numbers via old x87-registers
- You can use these numbers via `long double` type, but its not a standart
  - works on x86_64/gcc/clang though
- Operations are slower (but more precise), cannot be vectorized



SPECTRAL:
TECHNOLOGIES

C++

- Considering various properties of fp numbers, compilers can't perform some
- optimizations with reorderings, vectorizations and such
- `-ffast-math` can be used to relax a lot of constraints. It includes:
  - -fno-trapping-math, -fno-signaling-nans
  - -fno-rounding-math
  - -funsafe-math-optimizations
  - -ffinite-math-only, -fno-signed-zeros
  - denormal values are flushed to 0
- Can greatly speed up your program

SPECTRAL:
TECHNOLOGIES

Common tips

- Use `double` by default
  - `float` is usually only faster, when your code is vectorized enough
  - `long double` is rarely needed in real cases
- Never compare fp numbers directly, check for absolute or relative error. For the same reason avoid using fp as keys in maps or hashmaps
- Avoid adding/substracting numbers of different magnitudes
- Be aware of denormalized values, as you can get a slowdown out of nowhere
  - Use `-ffast-math` to effectively disable them (and improve the performance in general)
- `double` isn't strictly better than `float` due to errors cancelling

Errors cancelling

- Assume we have 2 types sf2 and sf3, both are base-10 numbers, with 2 and 3 significant digits, respectively
- Consider following calculation

```
// Calculate (5 / 4) * 8. Expected result: 10
sf2 x_2 = 5.0 / 4.0; // 1.3
sf2 y_2 = x_2 * 8.0; // 10
sf3 x_3 = x_2; // 1.30
sf3 y_3 = x_3 * 8.0; // 10.4
```

- Surprisingly sf2 gives more precise answer, since errors of x_2 and y_2 cancelled each other out
- One of many unintuitive quirks of fp numbers

Additional materials

- Classic https://www.itu.dk/~sestoft/bachelor/IEEE754_article.pdf
  (What Every Computer Scientist Should Know About Floating-Point Arithmetic)
- https://gcc.gnu.org/wiki/FloatingPointMath
- https://en.algorithmica.org/hpc