CSO: Recitation 6

Goktug Saatcioglu

10.10.2019

1 Floating point numbers

We saw in class that fractional binary numbers cannot be used to represent very large of very small numbers (close to 0). Thus, we use floating point numbers to represent fractional numbers. The IEEE Standard 754 governs the rules regarding floating point hardware implementations and arithmetic behavior. The main goals of floating point is:

- Consistent representation of floating point numbers by all machines.
- Correctly rounded floating point operations.
- Consistent treatment of exceptional situations such as division by zero.

1.1 Representation

A floating point number is represented as:

$$(-1)^s(1/0).M \times 2^E$$

where

- s is the sign bit which determines whether a number is positive or negative,
- M is the significand which is a fractional value,
- E is the exponent which weights the value by a power of two.

We encode the floating point values as

- MSB s is sign bit,
- exp encodes E,
- $frac$ encodes M.

We have the following precisions:

- Single precision: 32 bits: 1 for sign, 8 for exponent and 23 for mantissa.
- Double precision: 64 bits: 1 for sign, 11 for exponent and 52 for mantissa.
- Extended precision: 80 bits: 1 for sign, 15 for exponent and 63 or 64 for mantissa. This is an Intel only format.

1.2 Normalized encoding

- Condition: $exp \neq 000\ldots0$ and $exp \neq 111\ldots1$.
- Exponent is: $E = Exp - (2^{k-1} - 1)$, k is the # of exponent bits. $(2^{k-1} - 1)$ is usually referred to as the bias.
  - Single precision: $E = exp - 127$
  - $Range(SingleE) = [-126, 127]$
  - Double precision: $E = exp - 1023$

  - $Range(DoubleE) = [-1023, 1022]$

- Significand is: $M = 1.(xxx...x)_2$
  - $Range(M) = [1.0, 2.0 - \epsilon)$, ϵ is the smallest possible floating point number.
  - Get extra leading bit for free, we refer to this as the hidden bit.

1.3 Denormalized encoding

- Condition: $exp = 000\ldots0$.
- Exponent is: $E = 1 - Bias$ (instead of $E = 0 - Bias$)
- Significand is: $M = 0.(xxx...x)_2$ (instead of $M = 1.(xxx...x)_2$)
- Cases:
  - $exp = 000\ldots0$, $frac = 000\ldots0$ represents zero, note that there are two distinct values: $+0$ and -0,
  - $exp = 000\ldots0$, $frac \neq 000\ldots0$ represents numbers very close to 0.0.

1.4 Special values encoding

- Condition: $exp = 111\ldots1$
- Case: $exp = 111\ldots1$, $frac = 000\ldots0$
  - Represents value ∞ (infinity)
  - Note that there are two infinities: $+\infty$ and $-\infty$
  - Used for operations that overflow
  - E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$
- Case: $exp = 111\ldots1$, $frac \neq 000\ldots0$
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined. E.g., $sqrt(-1)$,$\infty - \infty$,$\infty \times 0$

1.5 Rounding modes

IEEE 754 supports five rounding modes:

- Round to nearest even (default)
  - if fractional part $< .5$, round to 0
  - if fractional part $> .5$, round away from 0
  - if fractional part $= .5$, round to nearest even digit
- Round to nearest (tie: round away from 0)
- Round to 0
- Round down (to -1)
- Round up (to $+1$)

2 Exercises

For these exercises we define our floating point format as 1 sign bit, 3 exponent bits and 4 mantissa bits. So the bias is $2^{3-1} - 1 = 3$.

2.1 Encoding

Using the above floating point format, encode the number 2.625 to floating point.

- The integral part is easy, $(2)_{10} = (10)_2$.
- The fractional part can be figured out as follows:
  - $0.625 \times 2 = 1.25$, therefore generate 1 and continue with 0.25,
  - $0.25 \times 2 = 0.50$, therefore generate 0 and continue with 0.50,
  - $0.50 \times 2 = 1.00$, therefore generate 1 and stop as nothing remains.
- So we have: 10.101×2^0.
- Normalize, i.e. shift to the right by one. Now we have 1.0101×2^1.
- The exponent is $E = 1$ and we want $exp = 1$ which we can get from $(x - 3) = 1 \implies 4$ which means $exp = 100$.
- The sign bit is 0.
- The result is `0|100|0101`.

Using the above floating point format, encode the number 1.7 to floating point.

- The integral part is easy, $(1)_{10} = (1)_2$.
- The fractional part can be figured out as follows:
  - $0.7 \times 2 = 1.4$, therefore generate 1 and continue with 0.4,
  - $0.4 \times 2 = 0.8$, therefore generate 0 and continue with 0.8,
  - $0.8 \times 2 = 1.6$, therefore generate 1 and continue with 0.6,
  - $0.6 \times 2 = 1.2$, therefore generate 1 and continue with 0.2,
  - $0.2 \times 2 = 0.4$, therefore generate 0 and continue with 0.4,
  - $0.4 \times 2 = 0.8$, therefore generate 0 and continue with 0.8,
  - $0.8 \times 2 = 1.6$, therefore generate 1 and continue with 0.6,
  - $0.6 \times 2 = 1.2$, therefore generate 1 and continue with 0.6,
  - ...
  - Notice that this process continues on forever. The reason is the he number 7/10, which makes a perfectly reasonable decimal fraction, is a repeating fraction in binary, just as the faction 1/3 is a repeating fraction in decimal (it repeats in binary as well). We cannot represent this exactly as a floating point number. The closest we can come in four bits is .1011.
- So we have: 1.1011×2^0.
- Normalize. Now we have 1.1011×2^0.
- The exponent is $E = 0$ and we want $exp = 0$ which we can get from $(x - 3) = 0 \implies 3$ which means $exp = 011$.
- The sign bit is 0.
- The result is `0|011|1011`.

2.2 Decoding

Using the above floating point format, decode the floating point number `11010011` to decimal.

- Let's split the input: `1|101|0011`.
- The mantissa (with the hidden bit) is 1.0011 which is equivalent to $1 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4} = 1.1875$.
- The exponent is 101 and the bias is 3 so $E = (101)_2 - 3_{10} = 5 - 3 = 2$.
- The sign bit is 1 so we have a negative number.
- Bringing everything together, we have $(-1)^1(1.1875) \times 2^2 = -4.75$.

Using the above floating point format, decode the floating point number `00111011` to decimal.

- Let's split the input: `0|011|1011`.
- The mantissa (with the hidden bit) is 1.1011 which is equivalent to $1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4} = 1.6875$.
- The exponent is 011 and the bias is 3 so $E = (011)_2 - 3_{10} = 3 - 3 = 0$.
- The sign bit is 0 so we have a positive number.
- Bringing everything together, we have $(-1)^0(1.6875) \times 2^0 = 1.6875$.
- Notice how we do not get back 1.7! This phenemenon is known as the representation error of floating point numbers. Because we only have a finite amount of precision (finite number of digits) we cannot represent every possible real value in floating point.

2.3 Arithmetic

Let's try doing floating point addition. What is the result of doing $2.625 + 1.7$ in floating point and decimal?

- We know that 2.625 =`01000101` and 1.7 =`00111011`.
- Splitting the inputs into parts gets us 2.625 =`0|100|0101` and 1.7 =`0|011|1011`.
- To perform addition we must align the exponents. $E_1 = (100)_2 - (bias) = 4 - 3 = 1$ and $E_2 = (011)_2 - (bias) = 3 - 3 = 0$. Since $E_1 > E_2$ we increment E_2 by one to get $E_{res} = 100$.
- Since we incremented the exponent of 1.7 by one we must shift the mantissa to the right by one (why?). Thus, the new mantissa (with the hidden bit) becomes 0.1101 (notice how we lost a bit due to this shift, this is known as rounding error, and the hidden bit is part of this shift too).
- Since both signs are positive we do addition.
- We add the two mantissas (with the hidden bits) together to get: $1.0101 + 0.1101 = 10.0010$. Notice that the result is too large to fit into the mantissa as there is an overflow in the hidden bit. We must again shift by 1 and increment the resulting exponent by one, aka normalize the result. This gets us the mantissa 1.0001 and the exponent 2. (This is known as the rounding error associated with a floating point addition and, in general, each floating point arithmetic operation has a resulting floating point rounding error.)
- The floating point result is `0|101|1001`.
- Now we convert to decimal. The exponent, from above, is 2. The mantissa is $2^0 + 2^{-4} = 1.0625$. The sign bit is 0 so the number is positive. The resulting number is $1.0625 \times 2^2 = 4.25$. Notice how this result is way off compared to what the actual result should be! Round-off error, in general, can be weird and unpredictable. The more precision we have, in general, the more accurate our program is but this does not always hold.

2.4 Resources

- Float to decimal: click here.
- Decimal to float: click here.
- Floating point arithmetic: click here.

3 Catastrophic cancellation

Consider the condition `0.1+0.2==0.3` done in floating point arithmetic. In real semantics, we expect this result to be true but in floating point this is always false. The reason is that 0.1, 0.2 and 0.3 cannot be exactly represented in floating point and floating point arithmetic is not exact in this case.

Due to finiteness of precision, floating point addition can suffer from catastrophic cancellation or swamping. Suppose we have two floating point numbers $a = 10^5$ and $b = 10^{-12}$. The quantity $c = a + b$ is equal to a, since a and b differ by many orders of magnitude. This means that b is rounded 0 in the addition which causes $a + b$ to equal 0. This is referred to as catastrophic cancellation.

3.1 Rectifying cancellation

Consider, for example, for values of x very near 0, the expression

$$\sqrt{x+1} - 1$$

suffers cancellation, as 1 swamps x in the computation of $x + 1$, and the subsuquent subtraction results in 0. We can rewrite the computation in an equivalent form that avoids the cancellation. For example,

$$\sqrt{x+1} - 1 = \frac{x}{\sqrt{x+1}+1}$$

avoids the cancellation for values of x near zero. Now, the only value of x that results in a zero output is 0 itself.

Note: Not all cancellation can be avoided, and not all can- cellation is bad! (See Kahan summation algorithm as an example.)

3.2 Notable floating point cancellation accidents

See here for a famous example of floating point roundoff error and its effect on the patriot missile system during the Gulf War. The Vancouver Stock Exchange caused a lot of confusion with its index value due to subtle but slowly accumulating floating point roundoff error, see here. The Ariane 5 rocket exploded in mid-air due to a floating point conversion and you can read about the details here. A few more can be found here.