CS221
Mock
Floating Point Number Representation

So far we have discussed signed and unsigned number representations. But how do we represent fractions and floating point numbers? For example, we also need a way to represent a number like 409.331. This document will discuss a binary representation for floating point numbers, as well as the IEEE 754 floating point standard.

Converting from Decimal to Binary

Let's say that we want to convert 252.390625 into binary. The first task is to convert the number 252 into binary. We already know how to do this, we just divide 252 by 2 and keep the remainders, repeating the process with the non-fractional part. 252 = 11111100

The next step is to convert 0.390625 into binary. To do this, **instead of dividing by 2, we multiply by 2**. Each time we record whatever is to the left of the decimal place after the operation. The first number becomes the leftmost bit, and the final number will be the rightmost bit. We then repeat this process using whatever is to the right of the decimal place.

0.390625 * 2 =	0.78125	0 as leftmost bit
0.78125 * 2 =	1.5625	1 as the next bit
0.5625 * 2 =	1.125	1
0.125 * 2 =	0.25	0
0.25 * 2 =	0.5	0
0.5 * 2 =	1.0	1
0		

Upon hitting 0, we're finished. The binary representation of this number is then:

11111100.011001

Exercise:

What is 3.625 in binary?
What is 0.1 in binary?

Note that with the last example, we could continue forever. In practice, we continue the process until we reach the precision desired to represent the number. Also note that if we wanted to use this process on something other than base 2, we would just multiply by whatever base we were interested in (e.g., base 16).

Converting Binary to Decimal

Given a floating point number in binary like 1100.011001, how do we convert this back to decimal? The process is almost identical to the process for unsigned binary. The stuff to the left of the decimal point is the same:

$$
\begin{aligned}
1100 &= \\
&\quad 0 * 2^0 + 0 * 2^1 + 1*2^2 + 1*2^3 \\
&= \quad 4+8 \\
&= \quad 12
\end{aligned}
$$

For the fractional part, .011001 we multiply and sum each bit, but starting with 2^{-1} power and continuing up to 2^{-2}, 2^{-3}, etc.

.	0	1	1	0	0	1
	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}

Recall that 2^{-1} is just 1/2, 2^{-2} is 1/4, 2^{-3} is 1/8, etc.

Summing this up gives us:

$$
\begin{aligned}
&\quad 0 * 2^{-1} + 1*2^{-2} + 1*2^{-3} + 0*2^{-4} + 0*2^{-5} + 1*2^{-6} \\
&= \quad 1 / 4 \quad + \; 1 / 8 \quad + \; 1 / 64 \\
&= \quad .25 \; + .125 + .015625 \\
&= \quad .390625
\end{aligned}
$$

Putting these together gives us 12.390625.

Exercise: What is 100.1111 in decimal?

Scientific Notation

What we've described is conceptually how to convert fractions from decimal to binary, but typically the data isn't stored in the computer in the same format. First we need to normalize the data, using scientific notation.

Consider the decimal number 0201.0900 . First we need to determine which digits are significant. The formal rules for significant digits is:

1. A nonzero digit is always significant
2. The digit 0 is never significant when it precedes nonzero digits

Using these rules, we can discard the initial 0 leaving us with 201.0900. Notice that we kept the trailing zeros; they may or may not be significant. Without more information we can't tell if we want precisely 201.0900, or if perhaps 201.09 is all we knew and the extra zeros are padding.

To represent this in scientific notation, we move the decimal point to the position immediately to the right of the leftmost significant digit, and multiply by the correct factor of 10 to get back the original value:

$$2.010900 * 10^2$$

In this case, we moved the decimal point two places to the left, so we multiply by 10^2. If we had a fraction we would do the opposite and multiply by a fraction of 10:

$$0.0020109 = 2.0109 * 10^{-3}$$

We can apply the same process to binary fractions, but use powers of 2 instead of powers of 10. Say that we have the binary value 100.1011. Converting this to scientific notation results in:

$$1.001011 * 2^2$$

Similarly, 0.00100 converted to scientific notation results in:

$$1.00 * 2^{-3}$$

Exercise: What is 101011.101 in scientific notation?

IEEE 754 Representation

To represent a number using the IEEE 754 format, first convert it to binary scientific notation. For example, let's say that we end up with $1.001011 * 2^3$. In general terms, this value is: (Sign) * (Mantissa) * $2^{(\text{exponent})}$

In storing this value, by default, we will assume that the power is 2. To get this value back, we will need to store the "1.001011", the sign using a sign bit, and then the exponent 3.

The pieces that we must store are the:

- Sign
- Exponent (the 3)
- Mantissa (the 1.001011 part)

In a single precision floating point number using the IEEE 754 format, 32 bits are used to store all of these values. The format is:

```
S EEEEEEEE MMMMMMMMMMMMMMMMMMMMMMM
1        8                  23
```

1 bit is allocated to the sign field (the leftmost bit).
8 bits are allocated to store the exponent field.
23 bits are allocated to store the mantissa field.

Sign Field

This is just a sign bit, like we used with signed binary numbers. It is either 0 or 1. 0 indicates a positive number, and 1 indicates a negative number. For our example number of $1.001011 * 2^3$, this is positive so the sign bit would hold a 0.

Exponent Field

The exponent section is eight bits long and is also called the *characteristic*. Since we are using 8 bits, this means we have values from 0 to 255. However, how would we handle negative exponents? To address negative values, the exponent is **biased** by a value of 127. In other words, the value 127 is added to the actual exponent we want to represent:

BiasedExponent = 127 + ActualExponent

A short listing of values for the exponent is:

Decimal	Biased Decimal	Biased Binary
0	127 + 0 = 127	01111111
1	127 + 1 = 128	10000000
2	127 + 2 = 129	10000001
128	127 + 128 = 255	11111111*
-1	127 – 1 = 126	01111110
-127	127 – 127 = 0	00000000*

For our example number of $1.001011 * 2^3$, we want to represent 3. Using the bias of 127, we store 127+3 = 130 or 10000010 for the exponent field.

Mantissa Field

The mantissa section is twenty-three bits long and is sometimes called the *significand*. Since the power of 2 is implicit, all that is left to store are the significant digits of the

* We actually can't represent these values in a single precision IEEE 754 floating point value. These are special cases, described later

number to represent. In the case of our example of $1.001011 * 2^3$ this corresponds to the 1.001011 part.

Initially, if we have 23 bits to use, you might think that the mantissa would hold 10010110000000000000000 to hold our significant digits. However, it does not. The reason is that in scientific notation there will never be a leading 0. For example, we would never have $0.345 * 10^2$. To be in scientific notation, this would have to be expressed as $3.45 * 10^3$. We have the same issue in binary scientific notation. However, since we can't have a leading 0, this means there is only one alternative: there **must** be a leading 1! Since 0 and 1 are the only digits, the leading digit has to be a 1. So, we really don't need to represent the leading 1 in the binary scientific notation. We can just assume it is there. This is referred to as the *hidden bit*. This scheme has the advantage that it gives us one additional bit worth of precision.

The mantissa for $1.001011 * 2^3$ would then be
00101100000000000000000
This long stream of 0's has one additional zero than the initial "guess" at the mantissa's value.

Putting all the pieces together for this example gives us:
Sign bit = 0
Exponent = 10000010
Mantissa = 00101100000000000000000
Or: 0100 0001 0001 0110 0000 0000 0000 0000
Or: 41160000 in hex

Exercises:
Represent –10.4375 in IEEE 754 representation and express as a hex number.

Given the hex number: 3EA80000 what is this in decimal?

How would you represent 0 in IEEE 754?

Double Precision

In addition to the single precision floating point described here, there is also double precision floating point. These have 64 bits instead of 32, and instead of field lengths of 1, 8, and 23 as in single precision, have field lengths of 1, 11, and 52. The exponent field contains a value that is actually 1023 larger than the "true" exponent, rather than being larger by 127 as in single precision. Otherwise, it is exactly the same. The advantage of double precision floating point is the ability to represent larger numbers and also numbers with more precision (more decimal points). However, generally it is slower for the computer to operate upon the 64 bits than it is upon the 32 bits. Chapter 8.5 of Stallings has a table illustrating the range of values that can be represented using double precision.

Special Numbers

There are a couple of numbers that are special. Generally, these are numbers reserved for cases when some kind of error occurs. The most common such case is when you might attempt to divide by zero. Or you might want to represent the number 0 exactly! These special cases are assigned codes, where the exponent field is all 1's or 0's.

The first special case refers to *denormalized* numbers. Denormalized numbers occur when the exponent field is all 0's. Earlier we said that this would generally be 0 – 127 or –127 as the exponent. However, that is not the case. Instead, it is a special case to represent 0.mantissa * 2^{-126} instead of 1.mantissa * 2^{-127}. Why? One reason is this lets us encode the value 0 because we no longer have an implied 1 bit. If the mantissa is all 0's and the exponent is all 0's, then we have 0 * 2^{-126} or 0. This format also lets us represent really small numbers between 0 and 1*2^{-126}. This is referred to as *gradual underflow*. It is possible to have values that are too small to represent. By having denormalized numbers, we can at least extend the lower range of representable numbers. Eventually though, we will have an *underflow* condition where a number is too small to represent, and we end up with 0.

The next special case occurs when the exponent contains all 1's. Again, we said earlier that this would normally be 255 – 127 or an exponent of 128. However, there is a special case encoded for *infinity*. Infinity is encoded when the exponent field is all 1's and the mantissa is all 0's. Note that there can be + and – infinity. The special value of infinity is what happens in an *overflow* condition – when we try to represent a number that is too large for the allocated space.

Another special case also occurs when the exponent field is all 1's. If the exponent contains all 1's but the mantissa does not contain all 0's, then this is referred to as *NaN* or *Not a Number*. The sign bit is unused.

You will encounter NaN if you attempt to divide 0 by 0 or perform some other ambiguous computation. If you divide a number by 0, you will get infinity. Note that some systems will return NaN if you attempt to divide by 0, instead of infinity. Also note that not all CPUs will check for these conditions. Adding hardware to check for these special cases can be expensive and actually slow the system down as more transistors are utilized. For this reason, some CPUs will simply generate an *exception* (an error) and refuse to complete the operation.

Computing with Floating Point

Due to truncation, precision, and rounding errors in the exponent and mantissa you are not guaranteed that floating point numbers will be identical! As we noticed with the number 0.2, we need an infinite number of bits if we want to represent it exactly.

For example, if you compute add 0.2 to a number 100,000 times, you won't get a value that is exactly 20,000 larger. Due to rounding errors, you will get a slightly different value. For these reasons, it is best to check for a range of numbers when dealing with floating point (e.g., >=19999.99 and <= 20000.01) instead of strict equality.

Floating Point Inaccuracies

Here is an example illustrating how a floating point representation can lose accuracy compared to regular unsigned integers, especially as we increase the magnitude of the value.

Informally, if we have the same number of bits to represent both a floating point value and an integer, in the floating point format we have to use some bits to represent the exponent and sign. This leaves fewer bits for the mantissa. An unsigned integer is able to use all bits to represent mantissa. This means the floating point format will have less precision than the integer format.

Consider the following floating point format that uses 8 bits:

EEE MMMMM
3 5

The exponent is unbiased, so the exponent field can represent exponents from 0 to 7. The mantissa uses the hidden one bit. There are no special cases as with IEEE 754.

If the mantissa is all zero's and the exponent is 0, then the smallest number we can represent is $1.00000 * 2^0$ which equals 1.

If the mantissa is all one's and the exponent is 111, then the largest number we can represent is $1.11111 * 2^7$ which equals 11111100 or 252.

With all eight bits as an unsigned integer, we can represent 2^8 patterns, or the integers 0 to 255.

Right off the bat, we can see that our floating point format can't represent the numbers 253, 254, or 255, while the integer format can. Of course, the floating point format can represent lots of fractional numbers that the integer format can't represent.

Here is an example of error with the floating point format. If the biggest number we can represent is 252, what is the next smallest number representable? In binary that would be the value:

111 11110

This is $1.11110 * 2^7$ which equals 11111000 or 248. We're missing four whole integers in between this and the largest number representable.

The next smallest value in binary is:

111 11101

This is $1.11101 * 2^7$ which equals 11110100 or 244. We dropped another four values.

The amount of error is worst for the largest values, since any lack of precision in the mantissa is amplified by the exponent. For small values, we lose little to no precision. For example, we said that the smallest value representable was 1. What is the next smallest value? It is:

000 00001

This is $1.00001 * 2^0$ which is just 1.03125. No loss of precision here compared to integers!

We can also represent exactly the number two:

001 00000

This is $1.00000 * 2^1$ which is two.

In summary, the floating point representation suffers in precision for large numbers, but is more accurate for small values. This is something to take into account whenever you are converting integer numbers to floating point format, especially if the integer is large. The resulting floating point number may not be particularly close to the actual integer and could cause an error in your program.

Floating Point Arithmetic

To briefly describe floating point arithmetic, all numbers are stored and processed in exponential form.

Addition and Subtraction: Unlike twos complement, addition and subtraction are more complicated than multiplication and division when operating with floating point values. As with two's complement, subtraction is performed by changing the sign of the subtrahend and then performing an addition.

1. The first step is to check for zeros. If either operand is zero, the other is reported as the result.

2. The second step is to align the mantissa / significand. For example, given:

$$123 * 10^{0} + 456 * 10^{-2}$$

we can't just add the 123 and the 456. The digits have to be set to an equivalent exponent.

$$123 * 10^{0} + 4.56 * 10^{0} = 127.56 * 10^{0}$$

We could have aligned the digits by either shifting one number right or another number left. Since either operation can result in the loss of digits and precision, it is the smaller number that is shifted. Therefore, any lost digits are of smaller significance. For each shift right the exponent is increased by one, and for each shift left the exponent is decreased by one.

3. Add or subtract the mantissas

4. Normalize the result. After performing the addition we may end up with a non-normalized number (e.g., $127.56 * 10^{0}$) which will need to be re-normalized and put back into the IEEE 754 format. We might also have the unfortunate case that the new value results in overflow or underflow of the exponent, as well.

Conceptually, multiplication is easier: add the exponents and multiply the mantissas.

For division, the exponents are subtracted and the mantissas are divided.

Both might require reorganizing the results into normal form and watching out for the signs and for overflow/underflow, and division by zero. Consult the Stallings textbook for detailed flowcharts on performing addition, multiplication, or division using floating point.

Other resources

For a comprehensive discussion about floating point representations, see Goldberg's article "What Every Computer Scientist Should Know About Floating Point Arithmetic" at http://cch.loria.fr/documentation/IEEE754/ACM/goldberg.pdf (warning, 3Mb large file).

Sun Microsystems also has a good discussion of underflow available at http://docs.sun.com/htmlcoll/coll.648.2/iso-8859-1/NUMCOMPGD/ncg_math.html