Floating-point representation

by Carl Burch, Hendrix College, September 2011

Contents

Representing numbers as integers in a fixed number of bits has some notable limitations. It can't handle numbers that have a fraction, like 3.14, and it isn't suitable for very large numbers that don't fit into 32 bits, like 6.02×10^{23}. In this document, we'll study *floating-point representation* for handling numbers like these — and we'll look particularly at the IEEE format, which is used universally today.

1. Fixed-point

One possibility for handling numbers with fractional parts is to add bits after the decimal point: The first bit after the decimal point is the halves place, the next bit the quarters place, the next bit the eighths place, and so on.

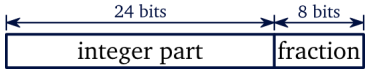
$$\overline{4}\ \overline{2}\ \overline{1}\ .\ \overline{\frac{1}{2}}\ \overline{\frac{1}{4}}\ \overline{\frac{1}{8}}$$

Suppose that we want to represent $1.625_{(10)}$. We would want 1 in the ones place, leaving us with 0.625. Then we want 1 in the halves place, leaving us with 0.625 − 0.5 = 0.125. No quarters will fit, so put a 0 there. We want a 1 in the eighths place, and we subtract 0.125 from 0.125 to get 0.

$$\frac{0}{4}\ \frac{0}{2}\ \frac{1}{1}.\frac{1}{1/2}\ \frac{0}{1/4}\ \frac{1}{1/8}$$

So the binary representation of 1.625 would be $1.101_{(2)}$.

The idea of **fixed-point representation** is to split the bits of the representation between the places to the left of the decimal point and places to the right of the decimal point. For example, a 32-bit fixed-point representation might allocate 24 bits for the integer part and 8 bits for the fractional part.



To represent 1.625, we would use the first 24 bits to indicate 1, and we'd use the remaining 8 bits to represent 0.625. Thus, our 32-bit representation would be:

00000000 00000000 00000001 10100000.

Fixed-point representation works reasonably well as long as you work with numbers within the supported range. The 32-bit fixed-point representation described above can represent any multiple of 1/256 from 0 up to $2^{24} \approx 16.7$ million. But programs frequently need to work with numbers from a much broader range. For this reason, fixed-point representation isn't used very often in today's computing world.

A notable exception is financial software. Here, all computations must be represented exactly to the penny, and in fact further precision is rarely desired, since results are always rounded to the penny. Moreover, most applications have no requirement for large amounts (like trillions of dollars), so the limited range of fixed-point numbers isn't an issue. Thus, programs typically use a variant of fixed-point representation that represents each amount as an integer multiple of 1/100, just as the fixed-point representation described above represents each number as a multiple of 1/256.

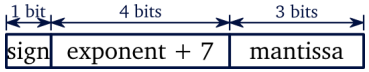
2. Normalized floating-point

Floating-point representation is an alternative technique based on scientific notation.

2.1. Floating-point basics

Though we'd like to use scientific notation, we'll base our scientific notation on powers of 2, not powers of 10, because we're working with computers that prefer binary. For example, $5.5_{(10)}$ is $101.1_{(2)}$ in binary, and it converts a to binary scientific notation of $1.011_{(2)} \times 2^2$. In converting to binary scientific notation here, we moved the decimal point to the left two places, just as we would in converting $101.1_{(10)}$ to scientific notation: It would be $1.011_{(10)} \times 10^2$.)

Once we have a number in binary scientific notation, we still must have a technique for mapping that into a set of bits. First, let us define the two parts of scientific representation: In $1.011_{(2)} \times 10^2$, we call $1.011_{(2)}$ the **mantissa** (or the **significand**), and we call 2 the **exponent**. In this section we'll use 8 bits to store such a number.



We use the first bit to represent the sign (1 for negative, 0 for positive), the next four bits for the sum of 7 and the actual exponent (we add 7 to allow for negative exponents), and the last three bits for the mantissa's fractional part. Note that we omit the integer part of the mantissa: Since the mantissa must have exactly one nonzero bit to the left of its decimal point, and the only nonzero bit is 1, we know that the bit to the left of the decimal point must be a 1. There's no point in wasting space in inserting this 1 into our bit pattern, so we include only the bits of the mantissa to the right of the decimal point.

For our example of $5.5_{(10)} = 1.011_{(2)} \times 2^2$, we add 7 to 2 to arrive at $9_{(10)} = 1001_{(2)}$ for the exponent bits. Into the mantissa bits we place the bits following the decimal point of the scientific notation, 011. This gives us 0 1001 011 as the 8-bit representation of $5.5_{(10)}$.

We call this *floating-point representation* because the values of the mantissa bits “float” along with the decimal point, based on the exponent's given value. This is in contrast to *fixed-point* representation, where the decimal point is always in the same place among the bits given.

Suppose we want to represent $-96_{(10)}$.

a. First we convert our desired number to binary: $-1100000_{(2)}$.
b. Then we convert this to binary scientific notation: $-1.100000_{(2)} \times 2^6$.
c. Then we fit this into the bits.
    1. We choose 1 for the sign bit since the number is negative.
    2. We add 7 to the exponent and place the result into the four exponent bits. For this example, we arrive at $6 + 7 = 13_{(10)} = 1101_{(2)}$.

3. The three mantissa bits are the first three bits following the leading 1: 100. If it happened that there were 1 bits beyond the 1/8's place, we would need to round the mantissa to the nearest eighth.)

Thus we end up with 1 1101 100.

Conversely, suppose we want to decode the number 0 0101 100.

1. We observe that the number is positive, and the exponent bits represent $0101_{(2)} = 5_{(10)}$. This is 7 more than the actual exponent, and so the actual exponent must be −2. Thus, in binary scientific notation, we have $1.100_{(2)} \times 2^{-2}$.
2. We convert this to binary: $1.100_{(2)} \times 2^{-2} = 0.011_{(2)}$.
3. We convert the binary into decimal: $0.011_{(2)} = 1/4 + 1/8 = 3/8 = 0.375_{(10)}$.

2.2. Representable numbers

This 8-bit floating-point format can represent a wide range of both small numbers and large numbers. To find the smallest possible positive number we can represent, we would want the sign bit to be 0, we would place 0 in all the exponent bits to get the smallest exponent possible, and we would put 0 in all the mantissa bits. This gives us 0 0000 000, which represents

$$1.000_{(2)} \times 2^{0-7} = 2^{-7} \approx 0.0078_{(10)}.$$

To determine the largest positive number, we would want the sign bit still to be 0, we would place 1 in all the exponent bits to get the largest exponent possible, and we would put 1 in all the mantissa bits. This gives us 0 1111 111, which represents

$$1.111_{(2)} \times 2^{15-7} = 1.111_{(2)} \times 2^{8} = 111100000_{(2)} = 480_{(10)}.$$

Thus, our 8-bit floating-point format can represent positive numbers from about $0.0078_{(10)}$ to $480_{(10)}$. In contrast, the 8-bit two's-complement representation can only represent positive numbers between 1 and 127.

But notice that the floating-point representation can't represent all of the numbers in its range — this would be impossible, since eight bits can represent only $2^8 = 256$ distinct values, and there are infinitely many real numbers in the range to represent. What's going on? Let's consider how to represent $51_{(10)}$ in this scheme. In binary, this is $110011_{(2)} = 1.10011_{(2)} \times 2^5$. When we try to fit the mantissa into the 3-bit portion of our scheme, we find that the last two bits won't fit: We would be forced to round to $1.101_{(2)} \times 2^5$, and the resulting bit pattern

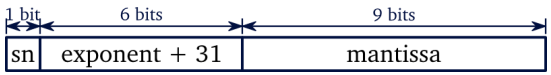
would be 0 1100 101. That rounding means that we're not representing the number precisely. In fact, 1 1100 101 translates to

$$1.101_{(2)} \times 2^{12-7} = 1.101_{(2)} \times 2^5 = 110100_{(2)} = 52_{(10)}.$$

Thus, in our 8-bit floating-point representation, 51 equals 52! That's pretty irritating, but it's a price we have to pay if we want to be able to handle a large range of numbers with such a small number of bits.

(By the way, in rounding numbers that are exactly between two possibilities, the typical policy is to round so that the final mantissa bit is 0. For example, taking the number 19, we end up with $1.0011_{(2)} \times 2^4$, and we would round this up to $1.010_{(2)} \times 2^4 = 20_{(10)}$. On the other hand, rounding up the number $21 = 1.0101_{(2)} \times 2^4$ would lead to $1.011_{(2)} \times 2^4$, leaving a 1 in the final bit of the mantissa, which we want to avoid; so instead we round *down* to $1.010_{(2)} \times 2^4 = 20_{(10)}$. Doubtless you were taught in grade school to “round up” all the time; computers don't do this because if we consistently round up, all those roundings will bias the total of the numbers upward. Rounding so the final bit is 0 ensure that exactly half of the possible numbers round up and exactly half round down.)

While a floating-point representation can't represent all numbers precisely, it *does* give us a guaranteed number of significant digits. For this 8-bit representation, we get a single digit of precision, which is pretty limited. To get more precision, we need more mantissa bits. Suppose we defined a similar 16-bit representation with 1 bit for the sign bit, 6 bits for the exponent plus 31, and 9 bits for the mantissa.



This representation, with its 9 mantissa bits, happens to provide three significant digits. Given a limited length for a floating-point representation, we have to compromise between more mantissa bits (to get more precision) and more exponent bits (to get a wider range of numbers to represent). For 16-bit floating-point numbers, the 6-and-9 split is a reasonable tradeoff of range versus precision.

3. IEEE standard

Nearly all computers today follow the the **IEEE 754 standard** for representing floating-point numbers. This standard was largely developed by 1980 and it was formally adopted in 1985, though several manufacturers continued to use their own formats throughout the 1980's. This standard is similar to the 8-bit and 16-bit formats we've explored already, but the standard deals with longer bit lengths to gain more precision and range; and it incorporates two special cases to deal with very small and very large numbers.

3.1. IEEE formats

There are two major varieties of the standard, for 32 bits and 64 bits, although it does define other lengths, like 128 bits.

format	sign bit	exponent bits	mantissa bits	exponent excess	significant digits
Our 8-bit	1	4	3	7	1
Our 16-bit	1	6	9	31	3
IEEE 32-bit	1	8	23	127	6
IEEE 64-bit	1	11	52	1,023	15
IEEE 128-bit	1	15	112	16,383	34

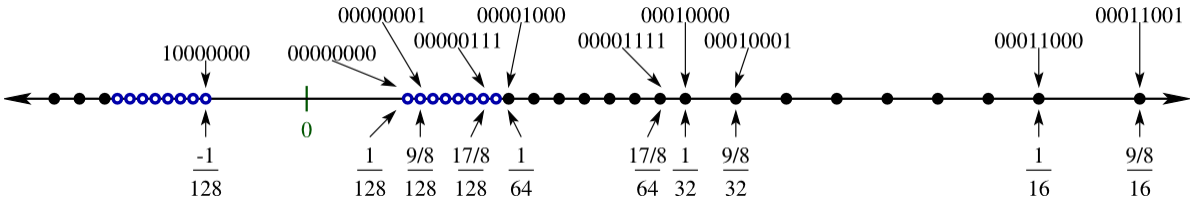
All formats use an offset for the exponent that is halfway up the range of integers that can fit into the exponent bits; this is called the **excess**. For the 8-bit format, we had 4 exponent bits; the largest number that can fit into 4 bits is $2^4 - 1 = 15$, and so the excess is $7 \approx 15/2$. The IEEE 32-bit format has 8 exponent bits, and so the largest number that fits is 255, and the excess is $127 \approx 255/2$.

In C programming, the `float` type corresponds to 32 bits, and `double` corresponds to 64 bits. (Actually, C doesn't specify the length or representation for these types. But this is most common. Java, incidentally, requires that these types correspond to the respective IEEE formats.)

The IEEE standard formats *generally* follow the rules we've outlined so far, but there are two exceptions: the denormalized numbers and the nonnumeric values. We'll look at these next.

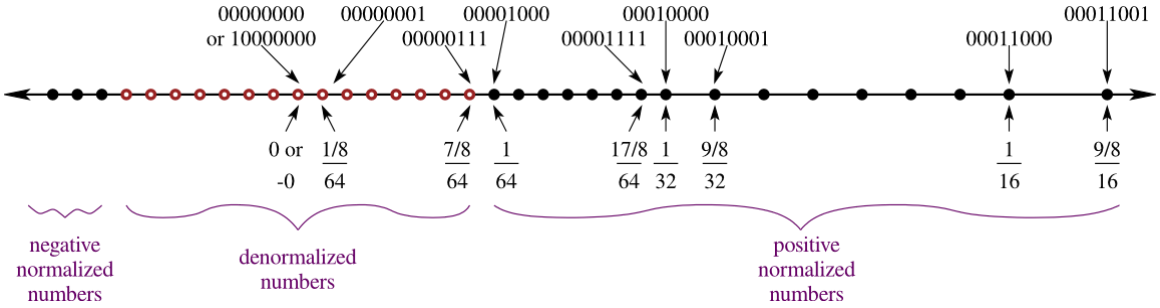
3.2. Denormalized numbers

The first special case is for dealing with very small values. Let's go back to the 8-bit representation we've been studying. If we plot the small numbers that can be represented on the number line, we get the distribution illustrated below.



The smallest representable positive number is 2^{-7} = 1/128 (bit pattern 00000000), and the largest representable negative number is -2^{-7} = −1/128 (bit pattern 10000000). These are small numbers, but when we look at the above number line, we see an anomaly: They're separated by a weirdly large gap. And, crucially, this gap skips over one of the most important numbers of all: zero!

To deal with this, the IEEE standard redefines how to interpret the bit patterns corresponding to the most closely spaced points (drawn as hollow blue circles above). Rather than have them defined as in the simple floating-point format, we instead spread them evenly across the range, yielding the hollow red points below.



Each of the hollow blue points in the first number line correspond to bit patterns where the exponent bits that are all 0. To arrive at the hollow red numbers instead, the meaning of such numbers changes as follows: When all exponent bits are 0, then the exponent is −6, and the mantissa has an implied 0 before it. Consider the bit pattern 0 0000 010: In IEEE-style floating-point (using the red points), this would represent $0.010_{(2)} \times 2^{-6}$. By contrast, in a simple floating-point format (using the blue points), this same bit pattern would correspond to $1.010_{(2)} \times 2^{-7}$; the changes are in the bit before the mantissa's decimal point and in the exponent of −7.

These red points are called **denormalized numbers**. The word *denormalized* comes from the fact that the mantissa is not in its *normal* form, where a nonzero digit is to the left of the decimal point.

Suppose we want to represent $0.005_{(10)}$ in our 8-bit floating-point format with a denormalized case. We first convert our number into the form $x \times 2^{-6}$. In this case, we would get $0.320_{(10)} \times 2^{-6}$. Converting $0.320_{(10)}$ to binary, we get approximately $0.0101001_{(2)}$. In the 8-bit format, however, we have only three mantissa bits, and so we would round this to $0.011_{(2)}$. Thus, we have $0.011_{(2)} \times 2^{-6}$, and our bit representation would be 0 0000 011. This is just an approximation to the original number of $0.005_{(10)}$: It is about $0.00586_{(10)}$. That may seem a crude approximation, but it's better than we could do in our simple format of before, where the closest we could get would be $0.00781_{(10)}$.

How would we represent 0? We go through the same process: Converting this into the form $x \times 2^{-6}$, we get 0.0×2^{-6}. This translates into the bit representation 0 0000 000. Notice that 1 0000 000 also represents 0 (technically, −0). Having two representations of 0 is an irritating anomaly, but the IEEE committee decided that it was benign compared to the complexity of any techniques they considered for avoiding it.

Why −6 for the exponent? It would make more intuitive sense to use −7, since this is what the all-zeroes exponent is normally. We use −6, however, because we want a smooth transition between the normalized values and the denormalized values. The smallest positive normalized value is 1×2^{-6} (bit pattern 0 0001 000). If we used −7 for the denormalized exponent, then the largest denormalized value would be $0.111_{(2)} \times 2^{-7}$, which is roughly half of the smallest positive normalized value. By using the same exponent as for the smallest normalized case, the standard spreads the denormalized numbers evenly from the smallest positive normalized number to 0. The second number line diagrammed above illustrates this: The hollow red circles, representing values handled by the denormalized case, are spread evenly between the solid black circles, representing the numbers handled by the normalized case.

Our discussion has focused on the 8-bit representation that in fact is quite impractical. But the same concept works the same for the IEEE standard floating-point formats. The major difference is that the exponent varies based on the format's excess. In the 32-bit standard, for example, the denormalized case still kicks in when all exponent bits are zero, but the exponent it represents is −126. That's because the normalized case involves an excess-127 exponent, and so the lowest exponent for normalized numbers is 1 − 127 = −126.

3.3. Nonnumeric values

The IEEE standard's designers were concerned with some special cases — particularly, computations where the answer doesn't fit into the range of defined numbers. To address such possibilities, they reserved the all-ones exponent for the **nonnumeric values**. They designed two types of nonnumeric values into the IEEE standard.

- If the exponent is all ones and the mantissa is all zeroes, then the number represents infinity or negative infinity, depending on the sign bit. Essentially, these two values represent numbers that have gone out of bounds. This value often results from an overflow; for example, if you added the largest positive value to itself, you would get infinity. Or if you divide 1 by a tiny number, you would get either infinity or negative infinity.

- If the exponent is all ones, and the mantissa has some non-zero bits, then the number represents “not a number,” written as NaN. This represents an error condition, such as the following.

  the square root of −1
  arcsine of 2

$\infty + -\infty$

$0 \div 0$

4. Arithmetic

You might wish that the standard laws of arithmetic would apply to numeric representations. For example, it would be convenient if the commutative law ($x + y = y + x$) applied to addition.

With integer arithmetic using two's-complement representation, all the standard laws apply except those that involve division. Identities involving division fail, of course, because division often results in non-integers, which can only be approximated in a two's-complement representation.

With IEEE floating-point numbers, however, many laws fall apart. The commutative law still holds for both addition and multiplication. But the associative law of addition ($x + (y + z) = (x + y) + z$) does not: Suppose x and y were the largest possible numeric value, and z were the most negative numeric value. Then the value of $x + (y + z)$ would be x, since $y + z$ is 0 and $x + 0$ is x. But the value of $(x + y) + z$ would be positive infinity, since $x + y$ results in overflow, which is equivalent to infinity in the IEEE format, and any finite number added to infinity results in infinity again.

The associative law fails for addition even without resorting to such special cases. Going back to our 8-bit representation, consider $x = 1$, $y = 52$, $z = -52$. If you evaluate $x + (y + z)$, you first notice that $y + z = 0$ and so $x + (y + z)$ is 1. But $(x + y) + z$ is 0, since the value 1 + 52 is still 52 due to rounding.

Another example of something that should be true but isn't in IEEE floating-point arithmetic is the following equation.

$$1/6 + 1/6 + 1/6 + 1/6 + 1/6 + 1/6 = 1.$$

The reason this fails is that 1/6 has no precise representation with a fixed number of mantissa bits, and so it must be approximated with the IEEE format. As it happens, it underestimates slightly. When we add this underestimation to itself 6 times, the total ends up being less than 1.

Such oddities are problems, and good programmers are very aware of them. Nobody has really come up with an alternative that is convincingly better than the IEEE approach to representing numbers: All have some odd anomalies, and the IEE approach avoids these better than most. Consequently, all modern computers use this representation for non-integral numbers.