(https://andybargh.com)

Understanding Fixed Point and Floating Point Number Representations

In my previous post (/2014/03/25/binary-fractions/) we learnt the fundamental concepts of how binary could be used to represent real numbers (i.e. numbers with a fractional component). When it comes to storing these numbers though there are two major approaches in modern computing. These are **Fixed Point Notation** and **Floating Point Notation**.

In today's post, we continue to build our background computer science knowledge and look at the details of these storage formats. This knowledge will be useful in upcoming posts when we dive into **Data Types** the fundamental building blocks of an iOS application.

As we learnt in my last post, fractional binary numbers have two parts, the bits that represent the integer number (the part before the radix point) and the bits that represent the fractional part (the part after the radix point).

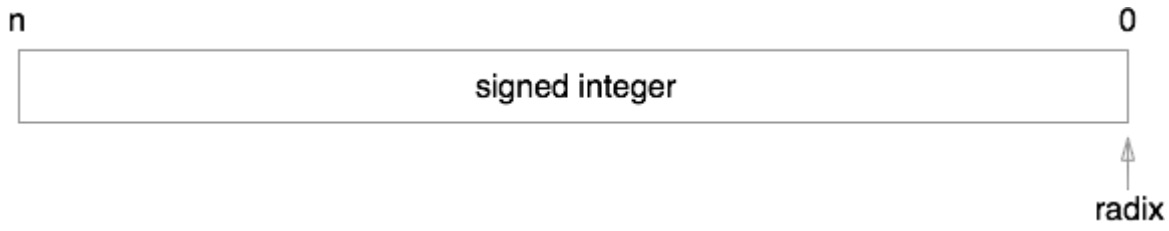
Think about this! What if we had only a limited number of binary bits in which to store our fractional binary number? This is common in many modern computers systems, How would we know how many bits to use for the integer part and how many bits to use for the fractional part?

This is the problem that both Fixed Point and Floating Point Notations representations attempt to solve.

To get started then, let's take a look at Fixed Point notation. It is the simpler one of the two representations.

Fixed Point Representation

Fixed Point Notation is a representation of our fractional number as it is stored in memory. In Fixed Point Notation, the number is stored as a signed integer in two's complement format (/2014/03/11/signed-numbers-in-binary/).



On top of this, we apply a notional split, locating the radix point (the separator between integer and fractional parts) a fixed number of bits to the left of its notational starting position to the right of the least significant bit. I've illustrated this in the diagram below.

When we interpret the bits of the signed integer stored in memory we reposition the radix point by multiplying the stored integer by a fixed scaling factor. The scaling factor in binary is always 2 raised to a fixed exponent. As the scaling factor is a power of 2 it relocates the radix point some number of places to

the left or right of its starting position.

During this conversion there are three directions that the radix point can be moved:

- **The radix point is moved to the right:** This is represented by a scaling factor whose exponent is 1 or more. In this case additional zeros are appended to the right of the least-significant bit and means that the actual number being represented is larger than the binary integer that was stored.

- **The radix point remains where it is:** This is represented by a scaling factor whose exponent is 0 and means that the integer value stored is exactly the same as the integer value being represented.

- **The radix point is moved to the left:** This is represented by a scaling factor whose exponent is negative. This means that the number being represented is smaller than the integer number that was stored and means that the number being represented has a fractional component.

Let's take a look at a couple of examples.

Examples of Fixed Point Numbers

Lets assume we have an 8-bit signed binary number 00011011_2 that is stored in memory using 8-bits of storage (hence the leading zeros).

In our first scenario, lets also assume this number was stored as a signed fixed-point representation with a scale factor of 2^2.

As our scale factor is greater than 1, when we translated the bits stored in memory into the number we are actually representing, we move the radix point two places to the right. This gives us the number: 1101100_2 (Note the additional zeros that are appended to the right of the least significant bit).

In our second scenario, let us assume that we start off with the same binary number in memory but this time we'll assume that it is stored as a signed fixed-point representation with a scale factor of 2^{-3}. As the exponent is negative we move the radix point three places to the left. This gives us the number 00011.011_2

Advantages and Disadvantages of Fixed Point Representation

The major advantage of using a fixed-point representation is performance. As the value stored in memory is an integer the CPU can take advantage of many of the optimizations that modern computers have to perform integer arithmetic without having to rely on additional hardware or software logic. This in turn can lead to increases in performance and when writing your apps, can therefore lead to an improved experience for your users.

However, there is a downside! Fixed Point Representations have a relatively limited range of values that they can represent.

So how do we work out the maximum and minimum numbers that can be stored in a fixed-point representation and determine whether it is suitable for our needs? All we do is take the largest and smallest integer values that can be stored in the given number of bits and multiply that by the scale factor associated with our fixed-point representation. For a given signed binary number using b bits of storage with a scale factor of f the maximum and minimum values that can be stored are:

Minimum: $-2^{b-1}/2^{f}$

Maximum: $(2^{b-1}-1)/2^{f}$

If the number you want to represent fits into this range then things are great. If it doesn't though, you have to look for an alternative! This is where Floating Point Notation comes in.

Floating Point Notation

Floating Point Notation is an alternative to the Fixed Point notation and is the representation that most modern computers use when storing fractional numbers in memory. Floating Point Notation is a way to represent very large or very small numbers precisely using scientific notation in binary. In doing so, Floating Point Representation provides a varying degrees of precision depending on the scale of the numbers that you are using.

For example, the level of precision we need when we are talking about the distance between atoms (10^{-10} m) is very different from the precision we need when we're talking about the distance between the earth and the sun (10^{11} m). This is a major benefit and allows a much wider range of numbers to be represented than is possible in Fixed Point Notation.

Floating Point Representation is based on **Scientific Notation**. You may have used Scientific Notation in school.

When we use Scientific Notation in decimal (the form you're probably most familiar with), we write numbers in the following form:

+/- mantissa x $10^{exponent}$

In this form, there is an optional sign indicating whether the overall number is positive or negative, followed by a mantissa (also known as a significand) which is a real (fractional) number which in turn is multiplied by a number base (or radix) raised by an exponent. As we know, in decimal this number base is 10.

Floating Point Representation is essentially Scientific Notation applied to binary numbers. In binary, the only real difference is that the number base is 2 instead of 10. We would therefore write Floating Point Numbers in the following form:

+/- mantissa x $2^{exponent}$

Now, you may not have realised it but when we write numbers in scientific notation (whether they be binary or decimal) we can write them in a number of different ways.

In decimal we could write 1.5 x 10^2, 15 x 10^1 and 150 x 10^0 and yet all these numbers have exactly the same value.

This provides flexibility but with this flexibility also comes confusion. To try and address this confusion a common set of rules known as normalized scientific notation are used to define how numbers in scientific notation are normally written.

Normalized Scientific Notation

Normalised Scientific Notation is a nomenclature that standardises the way we write numbers in scientific notation. In the normalized form we have a single key rule:

"We choose an exponent so that the absolute value of the mantissa remains greater than or equal to 1 but less than the number base."

Let's look at a couple of examples!

If we had the decimal number 50010 and wanted to write it in scientific notation we could write it as either 500×10^0 or 50×10^1.

In normalized form though, we would apply the rule above and move the radix point so that only a single digit, greater than or equal to 1 and less than (in this case) 10 were to the left of the radix point.

In this case this would mean moving our radix point two places to the left so we had $5.0 \times 10^?$.

We would then need to work out our exponent. To get back to our original number we would need to move our radix point two places to the right. Remember what we learnt earlier? If we have to move our radix point to the right to get back to our original number that means the exponent is positive. This gives us: 5.0×10^2.

Lets look at a slightly more complicated example, this time in binary.

What if we had the binary number 10.1_2? What would this be in scientific notation? Again we apply the rules: We need to have a mantissa that is greater than or equal to 1 and less than our number base (which this time is 2).

That would mean our mantissa would need to be $1.01 \times 2^?$. To get back to our original number we would need to move our radix point 1 place to the right. What does right mean? That means the exponent is positive.

Last example. This time one that is a little more tricky!

Imagine I had the number 0.111_2 and wanted to write it in normalized scientific notation? Again, we apply the rules. We need a mantissa greater than or equal to 1 and less than 2.

That means we want to write our mantissa as $1.11 \times 2^?$.

Now, to get back to our original number we would need to move our radix point 1 place to the... left. What did we learn about moving to the left? That means our exponent is negative. That gives us: 1.11×2^{-1}.

IEEE 754 Representations

As you've probably worked out by now, Floating Point numbers are used everywhere in modern computing. Whether it be the percentage of the market that have upgraded to the latest version of iOS, the current position and orientation of your iPhone in space or the amount of money flowing into your your bank account following the release of your latest blockbuster app!

Because of its wide use, the format used to store Floating Point numbers in memory has been standardized by the Institute of Electrical and Electronic Engineers in something called **IEEE 754**. This standard defines a number of different binary representations that can be used when storing Floating Point Numbers in memory:

- **Half Precision** – Uses 16-bits of storage in total.
- **Single Precision** – Uses 32-bits of storage in total.
- **Double Precision** – Uses 64-bits of storage in total.
- **Quadruple Precision** – Uses 128-bits of storage in total.

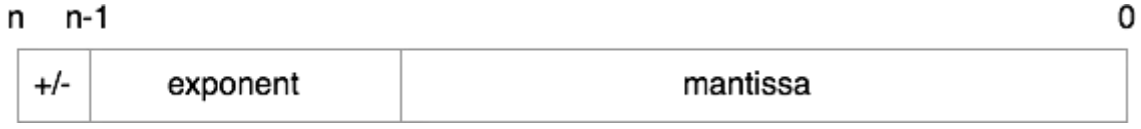
In each of these cases, their basic structure is as follows:

$(-1)^{sign} \times mantissa \times 2^{exponent}$

When it comes to storing Floating Point Numbers in memory, only three critical parts of that basic structure are stored:

- **Sign**
- **Exponent**
- **Mantissa**

The diagram below shows these parts are stored in memory:



The Sign

As i hinted at above, all four binary representations defined in the IEEE 754 standard, have the most significant bit as a sign bit and use it to store the sign of the overall number. In similar vein to what we have seen in previous posts, if the sign bit is clear (a value of 0) the overall number is positive. If the bit is set (a value of 1) the number is negative.

Exponent

The Exponent represent the power to which the mantissa will be raised. There are always a fixed number of exponent bits when storing a floating point representation in memory and the exact number of bits that are used is defined by the particular IEEE 754 representation (Single Precision, Double Precision etc). We'll take a look at this shortly.

In all cases, the exponents in each of these representations need to be able to represent both positive exponents (in order to represent very large numbers) and negative exponents (in order to represent very small numbers). To avoid the complications of having to store the exponents in two's complement format, something called an exponent bias is used.

Exponent Bias

Exponent Bias is where the value stored for the exponent is offset from the actual exponent value by a bias. The bias is simply a number that is added to the exponent to ensure that the value that is stored is always positive. The table below shows the number of bits used for the exponent in each of the formats, the allowed range of values the different exponents can have before applying the bias along with the allowed values after applying the bias:

Representation	Bits	Normal Range (Pre Bias)	Bias	Modified Range (Post Bias)	Notes
Half Precision	5	-14 to +15	+15	+1 to +30	Biased values of 0 (all bits clear) and 31 (all bits set) have special meaning.

Single Precision	8	-126 to +127	+127	+1 to +254	Biased values of 0 (all bits clear) and 255 (all bits set) have special meaning.
Double Precision	11	-1022 to +1023	+1023	+1 to +2046	Biased values of 0 (all bits clear) and 2047 (all bits set) have special meaning.
Quadrupal Precision	15	-16382 to +16383	+16383	+1 to +32766	Biased values of 0 (all bits clear) and 32767 (all bits set) have special meaning.

Mantissa (a.k.a. Signficand) Bits

In the IEEE 754 representations, the mantissa is expressed in normalized form. The formats follow the same rules for normalization as we saw with Scientific Notation, and puts the radix point after the first non-zero digit.

In binary though, we also get a nice little bonus!

As we are expressing our numbers in binary, we get the benefit of knowing that the first non-zero digit will always be a 1 (after all we can only have 1's or 0's). Given this, we are therefore able to drop that first bit, simply assuming it is there, and instead gain an additional (implicit) bit of precision.

When numbers are stored, we only store the part of the mantissa that represents the fractional part of the number, the part to the right of the radix point. The table below shows the effect of the implicit integer bit and the effect that it has on the overall precision:

Representation	Precision Bits	Effective Precision
Half Precision	1 bit (implicit) + 10 bits (explicit)	11 bits
Single Precision	1 bit (implicit) + 23 bits (explicit)	24 bits
Double Precision	1 bit (implicit) + 52 bits (explicit)	53 bits
Quadrupal Precision	1 bit (implicit) + 112 bits (explicit)	112 bits

Summary of IEEE 754 Representations

In summary then, the IEEE 754 standard defines four main formats for the representation of binary floating point numbers in memory:

Representation	Total Bits	Sign Bit	Mantissa Bits	Exponent Bits
Half Precision	16	1	1 implicit + 10	5
Single Precision	32	1	1 implicit + 23	8
Double Precision	64	1	1 implicit + 52	11
Quadrupal Precision	128	1	1 implicit + 112	15

In addition to these formats, the IEEE 754 standard also defines a number of numerical symbols that it is also worth knowing about. We'll explore these briefly in the next section.

Special Values

Representing Zero

As we have seen, when representing numbers in Floating Point and storing them in memory, we write our numbers in a normalized form before dropping the implicit set bit before the radix point. When the numbers in memory are interpreted, the implicit bit is re-instated. This implicit assumption that the bit immediately to the left of the radix point is set to 1 causes problems though. What if we wanted to represent zero?

To get around that problem, the IEEE 754 standard defines zero as a special case and represents it by using an exponent of 0 and a mantissa of 0. Due to the sign bit still being available this leads to values of -0 and +0 the standard defines that they must compared as equal.

Denormalized Form

The IEEE 754 also allows representation of numbers in a denormalized form. If the bits in the exponent are all zeros but the mantissa is non-zero value the number is said to be stored in a denormalized form. In this case, when the number in memory is interpreted, the assumption that there is a bit set to the left of the radix point is again ignored. This leads to numbers in the form:

Representation	
Half Precision	$(-1)^s \times 0.f \times 2^{-14}$
Single Precision	$(-1)^s \times 0.f \times 2^{-126}$
Double Precision	$(-1)^s \times 0.f \times 2^{-1022}$
Quadrupal Precision	$(-1)^s \times 0.f \times 2^{-16382}$

Where s is the sign and f is the fractional part of the mantissa.

Infinity

IEEE standard also defines a mechanism for representing infinity. Infinity is represented by an exponent with all the bits set and a mantissa with all the bits cleared. Again, the sign bit remains in effect which leads to the concept of +infinity and -infinity.

Not a Number (NaN)

The final thing that is of interest in the IEEE standard is the concept of Not A Number (NaN). This is used to represent a number that is not a real number. This is represented in memory by an exponent with all bits set and a non-zero mantissa. Most commonly you will see this reported by your compiler, usually when you've tried to divide something by zero.

Summary of Special Values

In summary then, the table below shows the different values of the exponent and mantissa and the special values that are being represented as defined by the IEEE 754 standard:

Exponent	Mantissa	Object Represented
Value of 0 (i.e. Stored Value == Bias)	All Bits Set to 0	Zero
Value of 0 (i.e. Stored Value == Bias)	Non-Zero	+/- Denormalized number
All Bits Set to 1	All Bits Set to 0	+/- Infinity
All Bits Set to 1	Non-Zero	NaN (Not a Number)

Summary

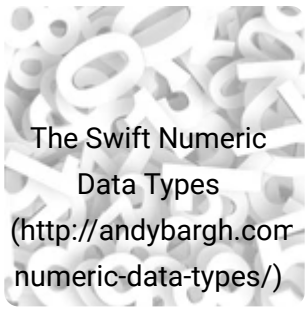
That about wraps it up for Fixed Point and Floating Point numbers for today. By reading this post I hope you will have gained a deep understanding of the differences between Fixed Point and Floating Point representations. That will allow you to make a much more informed choice about how to store

information within your apps in future. In forthcoming posts you will see that this provides a solid foundation on which to build your iOS development knowledge. Your knowledge should continue to grow if you work through my next post!

For now though, thanks for joining me and if you have any questions, please don't hesitate to leave them in the comments below.

Image credit: https://flic.kr/p/iCc7qY (https://flic.kr/p/iCc7qY)

Related Posts:



April 8, 2014
File Under: Computer Science (https://andybargh.com/category/computer-science/)
Tagged With: Binary (https://andybargh.com/tag/binary/), Computer Science (https://andybargh.com/tag/compsci/)

ALSO ON ANDYBARGH.COM

Pattern Matching in Swift

11 comments - 6 months ago

Optionals in Swift

2 comments - 9 months ago