Floating Point

Thomas Finley, April 2000

Contents and Introduction

This document explains the IEEE 754 floating-point standard. It explains the binary representation of these numbers, how to convert to decimal from floating point, how to convert from floating point to decimal, discusses special cases in floating point, and finally ends with some C code to further one's understanding of floating point. This document does not cover operations with floating point numbers.

I wrote this document so that if you know how to represent, you can skip the representation section, and if you know how to convert to decimal from single precision, you can skip that section, and if you know how to convert to single precision from decimal, you can skip that section.

Representation

First, know that binary numbers can have, if you'll forgive my saying so, a decimal point. It works more or less the same way that the decimal point does with decimal numbers. For example, the decimal 22.589 is merely 22 and $5*10^{-1} + 8*10^{-2} + 9*10^{-3}$. Similarly, the binary number 101.001 is simply $1*2^2 + 0*2^1 + 1*2^0 + 0*2^{-1} + 0*2^{-2} + 1*2^{-3}$, or rather simply $2^2 + 2^0 + 2^{-3}$ (this particular number works out to be 9.125, if that helps your thinking).

Second, know that binary numbers, like decimal numbers, can be represented in scientific notation. E.g., The decimal 923.52 can be represented as $9.2352 * 10^2$. Similarly, binary numbers can be expressed that way as well. Say we have the binary number 101011.101 (which is 43.625). This would be represented using scientific notation as $1.01011101 * 2^5$.

Now that I'm sure the understanding is perfect, I can finally get into representation. The single precision floating point unit is a packet of 32 bits, divided into three sections one bit, eight bits, and twenty-three bits, in that order. I will make use of the previously mentioned binary number $1.01011101 * 2^5$ to illustrate how one would take a binary number in scientific notation and represent it in floating point notation.

x	xxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxx
Sign 1 bit	**Exponent** 8 bits	**Mantissa** 23 bits

Sign Field

The first section is one bit long, and is the sign bit. It is either 0 or 1; 0 indicates that the number is positive, 1 negative. The number $1.01011101 * 2^5$ is positive, so this field would have a value of 0.

0	xxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxx

Exponent Field

The second section is eight bits long, and serves as the "exponent" of the number as it is expressed in scientific notation as explained above (there is a caveat, so stick around). A field eight bits long can have values ranging from 0 to 255. How would the case of a negative exponent be covered? To cover the case of negative values, this "exponent" is actually 127 greater than the "real" exponent a of the 2^a term in the scientific notation. Therefore, in our 1.01011101×2^5 number, the eight-bit exponent field would have a decimal value of 5 + 127 = 132. In binary this is 10000100.

0	10000100	xxxxxxxxxxxxxxxxxxxxxxx

Mantissa Field

The third section is twenty-three bits long, and serves as the "mantissa." (The mantissa is sometimes called the significand.) The mantissa is merely the "other stuff" to the left of the 2^a term in our binary number represented in scientific notation. In our $1.01011101 * 2^5$ number, you would think that the mantissa, in all its 23 bit glory, would take the value 10101110100000000000000, but it does not. If you think about it, all numbers expressed in binary notation would have a leading 1. Why? In decimal scientific notation there should never be expressed a value with a leading 0, like $0.2392 * 10^3$ or something. This would be expressed instead as $2.392 * 10^2$. The point is, there is never a leading 0 in scientific notation, and it makes no difference whether we're talking about binary or decimal or hexadecimal or whatever. The advantage of binary, though, is that if a digit can't be 0, it must be 1! So, the 1 is assumed to be there and is left out to give us just that much more precision. Thus, our mantissa for our number would in fact be 01011101000000000000000. (The long stream of 0s at the end has one more zero than the alternative number at the beginning of this paragagh.)

0	10000100	01011101000000000000000

Double Precision vs. Single Precision

In addition to the single precision floating point described here, there are also double precision floating point units. 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 44. 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.

Conversion from Floating Point Representation to Decimal

Now I'll talk about conversation.

Conversion to decimal is very simple if you know how these numbers are represented. Let's take the hexadecimal number 0xC0B40000, and suppose it is actually a single precision floating point unit that we want to make into a decimal number. First convert each digit to binary.

Hex	C	0	B	4	0	0	0	0
Binary	1100	0000	1011	0100	0000	0000	0000	0000

Then reorganize this number into packets of 1, 8, and 23 bits long.

1	10000001	01101000000000000000000

The sign field is one, which means this number is negative. The exponent field has a value of 129, which signifies a real exponent of 2 (remember the real exponent is the value of the exponent field minus 127). The mantissa has a value of 1.01101 (once we stick in the implied 1). So, our number is the following:

$$
\begin{aligned}
& -1.01101 * 2^2 \\
=\ & -(2^0 + 2^{-2} + 2^{-3} + 2^{-5}) \quad * \ 2^2 \\
=\ & -(2^2 + 2^0 + 2^{-1} + 2^{-3}) \\
=\ & -(4 + 1 + .5 + 0.125) \\
=\ & -5.625
\end{aligned}
$$

It's almost fun, yeah?

Conversion from Decimal to Floating Point Representation

Say we have the decimal number 329.390625 and we want to represent it using floating point numbers. The method is to first convert it to binary scientific notation, and then use what we know about the representation of floating point numbers to show the 32 bits that will represent it.

If you know how to put a decimal into binary scientific notation, you'll get no benefit from reading this. If you don't, read this.

The first step is to convert what there is to the left of the decimal point to binary. 329 is equivalent to the binary 101001001. Then, leave yourself with what is to the right of the decimal point, in our example 0.390625.

Yes, I deliberately chose that number to be so convoluted that it wasn't perfectly obvious what the binary representation would be. There is an algorithm to convert to different bases that is simple, straightforward, and largely foolproof. I'll illustrate it for base two. Our base is 2, so we multiply this number times 2. We then record whatever is to the left of the decimal place after this operation. We then take this number and discard whatever is to the left of the decimal place, and continue with this progress on the resulting number. This is how it would be done with 0.390625.

0.390625	* 2 = 0.78125	0
0.78125	* 2 = 1.5625	1
0.5625	* 2 = 1.125	1
0.125	* 2 = 0.25	0
0.25	* 2 = 0.5	0
0.5	* 2 = 1	1
0		

Since we've reached zero, we're done with that. The binary representation of the number beyond the decimal point can be read from the right column, from the top number downward. This is 0.011001.

As an aside, it is important to note that not all numbers are resolved so conveniently or quickly as sums of lower and lower powers of two (a number as simple as 0.2 is an example). If they are not so easily resolved, we merely continue on this process until we have however many bits we need to fill up the mantissa.

As another aside, to the more ambitious among you that don't know already, since this algorithm works similarly for all bases you could just as well use this for any other conversion you have to attempt. This can be used to your advantage in this process by converting using base 16.

0.390625	* 16 = 6.25	6
0.25	* 16 = 4	4
0		

If we convert simply from hex to binary, 0x64 is 0110 0100, which is the same result as the 011001 yielded above. This method is much faster.

Anyway! We take those numbers that we got, and represent them as .011001, placing them in the order we acquired them. Put in sequence with our binary representation of 329, we get 101001001.011001. In our binary scientific notation, this is $1.01001001011001 * 2^8$. We then use what we know about how single precision numbers are represented to complete this process.

The sign is positive, so the sign field is 0.

The exponent is 8. 8 + 127 = 135, so the exponent field is 10000111.

The mantissa is merely 01001001011001 (remember the implied 1 of the mantissa means we don't include the leading 1) plus however many 0s we have to add to the right side to make that binary number 23 bits long.

Since one of the homework problems involves representing this as hex, I will finish with a hex number.

0	10000111	01001001011001000000000

Then we break it into four bit pieces (since each hexadecimal digit is the equivalent of 4 bits) and then convert each four bit quantity into the corresponding hexadecimal digit.

Binary	0100	0011	1010	0100	1011	0010	0000	0000
Hex	4	3	A	4	B	2	0	0

So, in hexadecimal, this number is 0x43A4B200.

Special Numbers

Sometimes the computer feels a need to put forth a result of a calculation that reflects that some error was made. Perhaps the magnitude of the result of a calculation was larger or smaller than this format would seem to be able to support. Perhaps you attempted to divide by zero. Perhaps you're trying to represent zero! How does one deal with these issues? The answer is that there are special cases of floating point numbers, specifically when the exponent field is all 1 bits (255) or all 0 bits (0).

Denormalized Numbers

If you have an exponent field that's all zero bits, this is what's called a denormalized number. With the exponent field equal to zero, you would think that the real exponent would be -127, so this number would take the form of $1.MANTISSA * 2^{-127}$ as described above, but it does not. Instead, it is $0.MANTISSA * 2^{-126}$. Notice that the exponent is no longer the value of the exponent field minus 127. It is simply -126. Also notice that we no longer include an implied one bit for the mantissa.

As an example, take the floating point number represented as 0x80280000. First, convert this to binary.

Hex	8	0	2	8	0	0	0	0
Binary	1000	0000	0010	1000	0000	0000	0000	0000

We put this into the three 1 bit, 8 bits, and 23 bits packets that we're now familiar with.

1	00000000	01010000000000000000000

Our sign bit is 1, so this number is negative. Our exponent is 0, so we know this is a denormalized number. Our mantissa is 0101, which reflects a real mantissa of 0.0101; remember we don't include what was previously an implied one bit for an exponent of zero. So, this means we have a number $-0.0101_2 * 2^{-126} = -0.3125_{10} * 2^{-126} = -1.25_{10} * 2^{-128}$.

Zero

You can think of zero as simply another denormalized number. Zero is represented by an exponent of zero and a mantissa of zero. From our understanding of denormalized numbers, this translates into $0 * 2^{-126} = 0$. This sign bit can be either positive (0) or negative (1), leading to either a positive or negative zero. This doesn't make very much sense mathematically, but it is allowed.

Infinity

Just as the case of all zero bits in the exponent field is a special case, so is the case of all one bits. If the exponent field is all ones, and the mantissa is all zeros, then this number is an infinity. There can be either positive or negative infinities depending on the sign bit. For example, 0x7F800000 is positive infinity, and 0xFF800000 is negative infinity.

NaN (Not a Number)

These special quantities have an exponent field of 255 (all one bits) like infinity, but differ from the representation of infinity in that the mantissa contains some one bits. It doesn't matter where they are or how many of them there are, just so long as there are some. The sign bit appears to have no bearing on this. Examples of this special quantity include 0x7FFFFFFF, 0xFF81ABD0, 0x7FAA12F9, and soforth.

Summary of Special Cases

A summary of special cases is shown in the below table. It is more or less a copy of the table found on page 301 of the second edition of Computer Organization and Design, the Hardware Software Interface" by Patterson and Hennessy, the textbook for Computer Science 104 in the Spring 2000 semester. Even though only single precision was covered in the above text, I include double precision for the sake of completeness.

Single Precision		**Double Precision**		**Object Represented**
Exponent	Mantissa	Exponent	Mantissa	
0	0	0	0	zero
0	nonzero	0	nonzero	± denormalized number
1-254	anything	1-2046	anything	± normalized number
255	0	2047	0	± infinity
255	nonzero	2047	nonzero	NaN (Not a Number)

When, Where, and Where Not

When you have operations like 0/0 or subtracting infinity from infinity (or some other ambiguous computation), you will get NaN. When you divide a number by zero, you will get an infinity.

However, accounting for these special operations takes some extra effort on the part of the designer, and can lead to slower operations as more transistors are utilized in chip design. For this reason sometimes CPUs do not account for these operations, and instead generate an exception. For example, when I try to divide by zero or do operations with infinity, my computer generates exceptions and refuses to complete the operation (my computer has a G3 processor, or MPC750).

Helper Software

If you're interested in investigating further, I include two programs for which I provide the C code that you can run to gain a greater understanding of how floating point works, and also to check your work on various assignments.

Hex 2 Float

```
#include <stdio.h>int main(){  float theFloat;  while (1) {    scanf("%x", (int *) &theFloat);    printf("0x%08X, %f\n", *(int *)&theFloat, theFloat);  }  return 0;}
```

This program accepts as input a hexadecimal quantity and reads it as raw data into the variable "theFloat." The program then outputs the hexadecimal representation of the data in "theFloat" (repeating the input), and prints alongside it the floating point quantity that it represents.

I show here a sample run of the program. Notice the special case floating point quantities (0, infinity, and not a number).

For the denormalized but nonzero numbers, this program will display zero even though the number is not really zero. If you want to get around this problem, replace the %f in the formatting string of the printf function with %e, which will deplay the number to great precision with scientific notation. I did not have it as %e because I find scientific notation extremely annoying.

```
C0B400000xC0B40000, -5.62500043A4B2000x43A4B200, 329.390625000000000x00000000, 0.000000800000000x80000000, -0.0000007f8000000x7F800000, infff8000000xFF800000, -inf7fffffff0x7FFFFFFF, NaNffffffff0xFFFFFFFF, NaN7f81A0230x7F81A023, NaN
```

Float 2 Hex

```
#include <stdio.h>int main(){  float theFloat;  while (1) {    scanf("%f", &theFloat);    printf("0x%08X, %f\n", *(int *)&theFloat, theFloat);  }  return 0;}
```

This is a slight modification of the "Hex 2 Float" program. The exception is it reads in a floating point number. Just like and outputs the hexadecimal form plus the floating point number. Again I include a sample run of this program, confirming the results of the example problems I covered earlier in this text, along with some other simple cases. Notice the hexadecimal representation of 0.2.

```
-5.6250xC0B40000, -5.625000329.3906250x43A4B200, 329.39062500x00000000, 0.000000-00x80000000, -0.000000.20x3E4CCCCD, 0.200000.50x3F000000, 0.50000010x3F800000, 1.000000
```

And that's the end of that chapter.

Thomas Finley 2000