Floating-point Addition

In order to understand how the computer adds 2 floating point numbers, we need to recall that a floating-point number has two parts:

1. an 8-bit exponent expressed in biased-127
2. a 25-bit mantissa expressed in sign-magnitude

Here we are assuming single precision (32-bit representation). The way to add double precision numbers is the same, only involving more bits. It may sound odd that the mantissa is 25 bits, but remember the first bit is the sign bit, the 2^{nd} bit is the hidden bit, and the other 23 bits are from the fractional part of the mantissa.

The steps for performing floating-point addition are straightforward:

1. Make sure the exponents match. If they don't, make the smaller number match the exponent of the larger.
2. Add the mantissas. This specifically entails:
   (a) Convert the mantissas from sign-magnitude to 2's complement
   (b) Add the mantissas in 2's complement
   (c) Convert the mantissas back from 2's complement into sign-magnitude
3. Make sure the answer is normalized.

Note that steps 2(a) and 2(c) above, the conversions between sign-magnitude and 2's complement, are trivial if you are dealing with a positive number – in this case the representations are the same.

EXAMPLE #1
Let's try some examples. First, let's look at 8.75 + 1.25

For a human being, a preliminary step would be to figure out the correct IEEE FPS notation for these numbers. 8.75 in base 10 = 1000.11 in base 2. To convert this to binary scientific notation, we need an exponent of 3 and a mantissa of 1.00011. The number 1.25 in base 10 equals 1.01 in base 2, and in binary scientific notation we would need an exponent of 0 and a mantissa of 1.01.

So our two operands are

0 10000010 (1)00011
0 01111111 (1)01000

Note that both these numbers are positive, so that is why the sign bits are zero. The exponent 3 became "10000010" in biased-127 and 0 became "01111111". The figure (1) in each number represents the hidden bit. I only wrote 5 bits of the fractional mantissa; the other 18 to the right that I omitted would be all zeros.

Okay, time for step 1 of the addition: Make sure the exponents match. Here, they do not. The smaller number needs to be increased by 3 powers of 2. To increase the exponent while not changing the value of the number, we need to shift the mantissa to the right to compensate.

After matching exponents, we have:

0 10000010 (1)00011
0 10000010 (0)00101

You should compare the 2nd operand with its original representation to see what just happened: we added 3 to the exponent and shifted the exponent 3 to the right.

In step 2 we concern ourselves only with the mantissas, so we ignore the exponents for a while. Looking only at the mantissas we have

0 (1)00011
0 (0)00101

Step 2(a) says we need to convert these mantissas from sign-magnitude into 2's complement. But since they are positive there is nothing really to do in this case. Now step 2(b) does the actual addition:

```
0 (1)00011
0 (0)00101
-----------
0 (1)01000
```

Step 2(c) says to convert this 2's complement mantissa back into sign-magnitude. But this is easy again because the number is positive – do nothing.

In step 3 we write our final answer, inserting the exponent.

0 10000010 (1)01000

This is the legitimate IEEE notation for the number 10.0 (except for 0 mantissa bits I've been omitting off to the right.)

EXAMPLE #2
Now let's try other examples that illustrate more interesting situations. Next we will try 14.5 + 6.75.

14.5 in base 10 = 1110.1 in base 2 → exponent = 3, mantissa = 1.1101
6.75 in base 10 = 110.11 in base 2 → exponent = 2, mantissa = 1.1011

In step 1, we need to make the exponents match.

0 10000010 (1)1101
0 10000001 (1)1011

For the second number, which is the smaller of the two, we add 1 to the exponent and shift its mantissa one place to the right. For clarity, I'm going to write one additional zero at the end of the 1st number's mantissa so all the bits line up:

0 10000010 (1)11010
0 10000010 (0)11011

Step 2(a) tells us to convert the mantissas from sign-magnitude to 2's complement, but the numbers are positive so we don't need to change anything. Now step 2(b) tells us to add:

```
  0 (1)11010
  0 (0)11011
  ----------
 0 (10)10101
```

Note that here we have a carry out of the hidden bit, so now we are (for the moment) going to have 2 hidden bits. We cannot let this carry go over into the sign bit because this would actually be a sign of overflow. Overflow for floating-point addition cannot take place until we actually get to step 3 and look at the exponent of our final answer. So what we are doing here is a slightly modified version of the usual 2's complement addition algorithm.

Step 2(c) tells us to convert back from 2's complement to sign-magnitude, but again the number is positive so we do nothing.

Now time for step 3, where we need to resolve the issue of the 2 hidden bits. When we put the exponent and mantissa together in our answer we have:

0 10000010 (10)10101

When there are 2 hidden bits (hiding a value greater than 1), all we need to do is shift the mantissa to the right 1 place, and add 1 to the exponent to compensate. Now we have:

0 10000011 (1)011101

EXAMPLE #3: 12.0 – 3.5

12.0 in base 10 equals 1100.0 in base 2 → exponent = 3, mantissa = 1.1
3.5 in base 10 equals 11.1 in base 2 → exponent = 1, mantissa = 1.11

0 10000010 (1)10
1 10000000 (1)11

step 1 – make the exponents match

0 10000010 (1)1000
1 10000010 (0)0111

step 2 – perfrom 2's complement addition:

```
      0 (1)1000
+     1 (0)0111
```

becomes

```
      0 (1)1000
+     1 (1)1001
--------------------
      0 (1)0001
```

and we convert back to sign-magnitude, but answer is already positive,
so our final answer is

0 10000010 (1)0001

which looks like 8.5.

EXAMPLE #4: 2.5 – 0.75

2.5 in base 10 equals 10.1 in base 2 → exponent = 1, mantissa = 1.01
0.75 in base 10 equals .11 in base 2 → exponent = -1, mantissa = 1.1

0 10000000 (1)01
1 01111110 (1)1

step 1 – make exponents match

0 10000000 (1)010
1 10000000 (0)011

step 2 – perform addition

```
      0 (1)010
+     1 (0)011
```

becomes

```
       0 (1)010
+      1 (1)101
-----------------
       0 (0)111
```

Step 3 – put answer together and NORMALIZE!

0 10000000 (0)111

becomes

0 01111111 (1)11

which looks like 1.75