# Floating Point Numbers notes and example

We have seen how to represent fractions and non-integers in binary (base 2) notation. For instance:
`23 5/8` can become `10111.101`

However, we encounter a problem when we try to represent this on a computer. We have talked about physical ways to represent 0s and 1s. However, with only two options, we have no way of representing the "binary point". Floating point representations are a way of dealing with this. For floating point, we need a *sign*, an *exponent* and a *fractional part*.

There are a couple steps.

**1)** Convert the binary number to "normal form".

a) This means to move the binary point to just after the first 1 in the representation, and keep track of how many places it had to move and in which direction. For instance:

`10111.101` becomes `1.0111101` by moving the binary point 4 places to the left.

b) This number is then written as

$1.0111101 * 2^4$

This is called `"normal form".`

**2)** a) Now, for the number fill in the three following pieces of information:

Example:

| Sign | exponent | fractional part |
|---|---|---|
| + | +4 | 0111101 |

Note that the "leading" 1 has been omitted from the fractional part. Why can we do this?

b) Now, represent the sign as a bit, the exponent in sign-magnitude binary notation and leave the fractional part.

Example:

| Sign | exponent | fractional part |
|---|---|---|
| 0 | 0100 | 0111101 |

c) End result:

`001000111101`

ASSUMPTIONS: This assumes 12 bits for one number, 4 used for the exponent and 7 used for the fractional part.

Challenge: Write this number in Hexadecimal.

**CHANGE ASSUMPTIONS:**

Now, lets assume 16 bits for one number, 6 bits for the exponent. (leaving 9 for the fractional part)

Revisit the Example:

| Sign | exponent | fractional part |
|---|---|---|
| 0 | 0100 | 0111101 |

We now need two more bits for the exponent and two more for the fractional part. We add 0's. Where do we add them?

Revisit the Example:

| Sign | exponent | fractional part |
|---|---|---|
| 0 | 000100 | 011110100 |

Result:

0000100011110100

Challenge: Write this number in Hexadecimal.