- The essential idea of floating point representation is that a fixed number of bits are used (usually 32 or 64) and that the binary point "floats" to where it is needed in those bits. Of course, the computer only holds bit patterns. Some of the bits of a floating point representation must be used to say where the binary point lies. Floating point expressions can represent numbers that are very small and numbers that are very large. When a floating point calculation is performed, the binary point floats to the correct position in the result. The programmer does not need to explicitly keep track of it.

- How's it stored?
    - The method that the developers of IEEE 754 Form finally hit upon uses the idea of scientific notation. Scientific notation is a standard way to express numbers; it makes them easy to read and compare. We just factor our number into two parts: a value whose magnitude is in the range of $1 \le n < 10$, and a power of 10. For example:

$$3498523 \quad \text{is written as} \quad 3.498523 \times 10^{6}$$

$$-0.0432 \quad \text{is written as} \quad -4.32 \times 10^{-2}$$

    - The same idea applies here, except that we need to use powers of 2 because the computer works efficiently with binary numbers. Just factor the number into a value whose magnitude is in the range $1 \le n < 2$, and a power of 2.

$$-6.84 \quad \text{is written as} \quad -1.71 \times 2^{2}$$

$$0.05 \quad \text{is written as} \quad 1.6 \times 2^{-5}$$

    - To create the bitstring, we need to massage this product so that it takes the following form:

$$(-1)^{\text{sign bit}}(1 + \text{fraction}) \times 2^{\text{exponent - bias}}$$

Once this is done, we will have three key pieces of information (shown in color above) that, when taken together, identify the number:

- *First Piece* -- If the sign bit is a 0, then the number is positive; $(-1)^0 = 1$. If the sign bit is a 1, the number is negative; $(-1)^1 = -1$.
- *Second Piece* -- We always factor so that the number in parentheses equals (1+ some fraction)(1+ some fraction). Since we know that the 1 is there, the only important thing is the fraction, which we will write as a binary string.

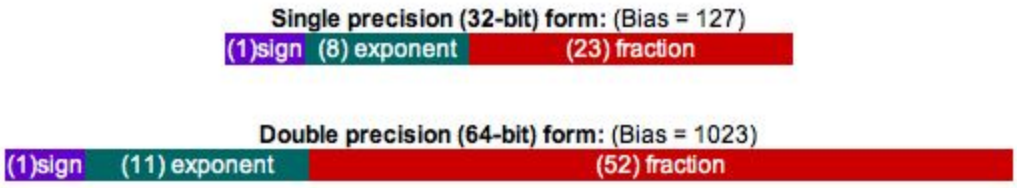
$$0.1_{binary} = 2^{-1} = 0.5$$

$$0.01_{binary} = 2^{-2} = 0.25$$

$$0.101_{binary} = 2^{-1} + 2^{-3} = 0.625$$

- *Third Piece* -- The power of 2 that we got in the last step is simply an integer. Note, this integer may be positive or negative, depending on whether the original value was large or small, respectively. We'll need to store this exponent -- however, using the two's complement, the usual representation for signed values, makes comparisons of these values more difficult. As such, we add a constant value, called a *bias*, to the exponent. By biasing the exponent before it is stored, we put it within an unsigned range more suitable for comparison.
    a. For single-precision floating-point, exponents in the range of -126 to + 127 are biased by adding 127 to get a value in the range 1 to 254 (0 and 255 have special meanings).
    b. For double-precision, exponents in the range -1022 to +1023 are biased by adding 1023 to get a value in the range 1 to 2046 (0 and 2047 have special meanings).

- The sum of the bias and the power of 2 is the exponent that actually goes into the IEEE 754 string. Remember, the exponent = power + bias. (Alternatively, the power = exponent-bias). This exponent must itself ultimately be expressed in binary form -- but given that we have a positive integer after adding the bias, this can now be done in the normal way.



(*The numbers in parentheses show how many bits are required in each field.*)

- By arranging the fields in this way, so that the sign bit is in the most significant bit position, the biased exponent in the middle, then the mantissa in the least significant bits -- the resulting value will actually be ordered properly for comparisons, whether it's interpreted as a floating point or integer value. This allows high speed comparisons of floating point numbers using fixed point hardware.

There are some special cases:

- *Zero*
    - Sign bit = 0; biased exponent = all 0 bits; and the fraction = all 0 bits;
- *Positive and Negative Infinity*
    - Sign bit = 0 for positive infinity, 1 for negative infinity; biased exponent = all 1 bits; and the fraction = all 0 bits;
- *NaN (Not-A-Number)*
    - Sign bit = 0 or 1; biased exponent = all 1 bits; and the fraction is anything but all 0 bits. (NaN's pop up when one does an invalid operation on a floating point value, such as dividing by zero, or taking the square root of a negative number.)