Anatomy of a floating point number

Posted on **6 April 2009** by **John**

In my previous post, I explained that floating point numbers are a leaky abstraction. Often you can pretend that they are mathematical real numbers, but sometimes you cannot. This post peels back the abstraction and explains exactly what a floating point number is. (Technically, this post describes an IEEE 754 double precision floating point number, by far the most common kind of floating point number in practice.)

A floating point number has 64 bits that encode a number of the form $\pm p \times 2^e$. The first bit encodes the sign, 0 for positive numbers and 1 for negative numbers. The next 11 bits encode the exponent e, and the last 52 bits encode the precision p. The encoding of the exponent and precision require some explanation.

The exponent is stored with a bias of 1023. That is, positive and negative exponents are all stored in a single positive number by storing e + 1023 rather than storing e directly. Eleven bits can represent integers from 0 up to 2047. Subtracting the bias, this corresponds to values of e from -1023 to +1024. Define $e_{min} = -1022$ and $e_{max} = +1023$. The values $e_{min} - 1$ and $e_{max} + 1$ are reserved for special use. More on that below.

Floating point numbers are typically stored in *normalized* form. In base 10, a number is in normalized scientific notation if the significand is ≥ 1 and < 10. For example, 3.14×10^2 is in normalized form, but 0.314×10^3 and 31.4×10^2 are not. In general, a number in base β is in normalized form if it is of the form $p \times \beta^e$ where $1 \leq p < \beta$. This says that for binary, i.e. $\beta = 2$, the first bit of the significand of a normalized number is always 1. Since this bit never changes, it doesn't need to be stored. Therefore we can express 53 bits of precision in 52 bits of storage. Instead of storing the significand directly, we store f, the fractional part, where the significand is of the form 1.f.

The scheme above does not explain how to store 0. Its impossible to specify values of f and e so that $1.f \times 2^e = 0$. The floating point format makes an exception to the rules stated above. When $e = e_{min} - 1$ and $f = 0$, the bits are interpreted as 0. When $e = e_{min} - 1$ and $f \neq 0$, the result is a denormalized number. The bits are interpreted as $0.f \times 2^{e_{min}}$. In short, the special exponent reserved below e_{min} is used to represent 0 and denormalized floating point numbers.

The special exponent reserved above e_{max} is used to represent ∞ and `NaN`. If $e = e_{max} + 1$ and $f = 0$, the bits are interpreted as ∞. But if $e = e_{max} + 1$ and $f \neq 0$, the bits are interpreted as a `NaN` or "not a number." See IEEE floating point exceptions for more information about ∞ and `NaN`.

Since the largest exponent is 1023 and the largest significant is 1.f where f has 52 ones, the largest floating point number is $2^{1023}(2 - 2^{-52}) = 2^{1024} - 2^{971} \approx 2^{1024} \approx 1.8 \times 10^{308}$. In C, this constant is defined as `DBL_MAX`, defined in `<float.h>`.

Since the smallest exponent is -1022, the smallest positive *normalized* number is $1.0 \times 2^{-1022} \approx 2.2 \times 10^{-308}$. In C, this is defined as `DBL_MIN`. However, it is not the smallest positive number representable as a floating point number, only the smallest normalized floating point number. Smaller numbers can be expressed in denormalized form, albeit at a loss of significance. The smallest denormalized positive number occurs with f has 51 0's followed by a single 1. This corresponds to $2^{-52}*2^{-1022} = 2^{-1074} \approx 4.9 \times 10^{-324}$. Attempts to represent any smaller number must underflow to zero.

C gives the name `DBL_EPSILON` to the smallest positive number ε such that $1 + \varepsilon \neq 1$ to machine precision. Since the significant has 52 bits, it's clear that `DBL_EPSILON` $= 2^{-52} \approx 2.2 \times 10^{-16}$. That is why we say a floating point number has between 15 and 16 significant (decimal) figures.

For more details see What Every Computer Scientist Should Know About Floating-Point Arithmetic.

First post in this series: Floating point numbers are a leaky abstraction