IEEE Standard 754
Floating Point Numbers

Steve Hollasch / Last update 2005-Feb-24

IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macintoshes, and most Unix platforms. This article gives a brief overview of IEEE floating point and its representation. Discussion of arithmetic implementation may be found in the book mentioned at the bottom of this article.

What Are Floating Point Numbers?

There are several ways to represent real numbers on computers. Fixed point places a radix point somewhere in the middle of the digits, and is equivalent to using integers that represent portions of some unit. For example, one might represent 1/100ths of a unit; if you have four decimal digits, you could represent 10.82, or 00.01. Another approach is to use rationals, and represent every number as the ratio of two integers.

Floating-point representation - the most common solution - basically represents reals in scientific notation. Scientific notation represents numbers as a base number and an exponent. For example, 123.456 could be represented as 1.23456×10^2. In hexadecimal, the number 123.abc might be represented as $1.23abc \times 16^2$.

Floating-point solves a number of representation problems. Fixed-point has a fixed window of representation, which limits it from representing very large or very small numbers. Also, fixed-point is prone to a loss of precision when two large numbers are divided.

Floating-point, on the other hand, employs a sort of "sliding window" of precision appropriate to the scale of the number. This allows it to represent numbers from 1,000,000,000,000 to 0.0000000000000001 with ease.

Storage Layout

IEEE floating point numbers have three basic components: the sign, the exponent, and the mantissa. The mantissa is composed of the *fraction* and an implicit leading digit (explained below). The exponent base (2) is implicit and need not be stored.

The following figure shows the layout for single (32-bit) and double (64-bit) precision floating-point values. The number of bits for each field are shown (bit ranges are in square brackets):

	Sign	Exponent	Fraction	Bias
Single Precision	1 [31]	8 [30-23]	23 [22-00]	127
Double Precision	1 [63]	11 [62-52]	52 [51-00]	1023

The Sign Bit

The sign bit is as simple as it gets. 0 denotes a positive number; 1 denotes a negative number. Flipping the value of this bit flips the sign of the number.

The Exponent

The exponent field needs to represent both positive and negative exponents. To do this, a *bias* is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73. For reasons discussed later, exponents of -127 (all 0s) and +128 (all 1s) are reserved for special numbers.

For double precision, the exponent field is 11 bits, and has a bias of 1023.

The Mantissa

The *mantissa*, also known as the *significand*, represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits.

To find out the value of the implicit leading bit, consider that any number can be expressed in scientific notation in many different ways. For example, the number five can be represented as any of these:

$$5.00 \times 10^{0}$$
$$0.05 \times 10^{2}$$
$$5000 \times 10^{-3}$$

In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in *normalized* form. This basically puts the radix point after the first non-zero digit. In normalized form, five is represented as 5.0×10^{0}.

A nice little optimization is available to us in base two, since the only possible non-zero digit is 1. Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa has effectively 24 bits of resolution, by way of 23 fraction bits.

Putting it All Together

So, to sum up:

1. The sign bit is 0 for positive, 1 for negative.

2. The exponent's base is two.
3. The exponent field contains 127 plus the true exponent for single-precision, or 1023 plus the true exponent for double precision.
4. The first bit of the mantissa is typically assumed to be 1.*f*, where *f* is the field of fraction bits.

Ranges of Floating-Point Numbers

Let's consider single-precision floats for a second. Note that we're taking essentially a 32-bit number and re-jiggering the fields to cover a much broader range. Something has to give, and it's precision. For example, regular 32-bit integers, with all precision centered around zero, can precisely store integers with 32-bits of resolution. Single-precision floating-point, on the other hand, is unable to match this resolution with its 24 bits. It does, however, approximate this value by effectively truncating from the lower end. For example:

```
    11110000 11001100 10101010 00001111  // 32-bit integer
= +1.1110000 11001100 10101010 x 2^31     // Single-Precision Float
=   11110000 11001100 10101010 00000000  // Corresponding Value
```

This approximates the 32-bit value, but doesn't yield an exact representation. On the other hand, besides the ability to represent fractional components (which integers lack completely), the floating-point value can represent numbers around 2^{127}, compared to 32-bit integers maximum value around 2^{32}.

The range of positive floating point numbers can be split into normalized numbers (which preserve the full precision of the mantissa), and *denormalized* numbers (discussed later) which use only a portion of the fractions's precision.

	Denormalized	**Normalized**	**Approximate Decimal**
Single Precision	$\pm 2^{-149}$ to $(1-2^{-23})\times 2^{-126}$	$\pm 2^{-126}$ to $(2-2^{-23})\times 2^{127}$	$\pm \sim 10^{-44.85}$ to $\sim 10^{38.53}$
Double Precision	$\pm 2^{-1074}$ to $(1-2^{-52})\times 2^{-1022}$	$\pm 2^{-1022}$ to $(2-2^{-52})\times 2^{1023}$	$\pm \sim 10^{-323.3}$ to $\sim 10^{308.3}$

Since the sign of floating point numbers is given by a special leading bit, the range for negative numbers is given by the negation of the above values.

There are five distinct numerical ranges that single-precision floating-point numbers are **not** able to represent:

1. Negative numbers less than $-(2-2^{-23}) \times 2^{127}$ (*negative overflow*)

2. Negative numbers greater than -2^{-149} (*negative underflow*)
3. Zero
4. Positive numbers less than 2^{-149} (*positive underflow*)
5. Positive numbers greater than $(2-2^{-23}) \times 2^{127}$ (*positive overflow*)

Overflow means that values have grown too large for the representation, much in the same way that you can overflow integers. Underflow is a less serious problem because is just denotes a loss of precision, which is guaranteed to be closely approximated by zero.

Here's a table of the effective range (excluding infinite values) of IEEE floating-point numbers:

	Binary	**Decimal**
Single	$\pm (2-2^{-23}) \times 2^{127}$	$\sim \pm 10^{38.53}$
Double	$\pm (2-2^{-52}) \times 2^{1023}$	$\sim \pm 10^{308.25}$

Note that the extreme values occur (regardless of sign) when the exponent is at the maximum value for finite numbers (2^{127} for single-precision, 2^{1023} for double), and the mantissa is filled with 1s (including the normalizing 1 bit).

Special Values

IEEE reserves exponent field values of all 0s and all 1s to denote special values in the floating-point scheme.

Zero

As mentioned above, zero is not directly representable in the straight format, due to the assumption of a leading 1 (we'd need to specify a true zero mantissa to yield a value of zero). Zero is a special value denoted with an exponent field of zero and a fraction field of zero. Note that -0 and +0 are distinct values, though they both compare as equal.

Denormalized

If the exponent is all 0s, but the fraction is non-zero (else it would be interpreted as zero), then the value is a *denormalized* number, which does *not* have an assumed leading 1 before the binary point. Thus, this represents a number $(-1)^s \times 0.f \times 2^{-126}$, where s is the sign bit and f is the fraction. For double precision, denormalized numbers are of the form $(-1)^s \times 0.f \times 2^{-1022}$. From this you can interpret zero as a special type of denormalized number.

Infinity

The values +infinity and -infinity are denoted with an exponent of all 1s and a fraction of all 0s. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations. *Operations with infinite values are well defined in IEEE floating point.*

Not A Number

The value NaN (*Not a Number*) is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1s and a non-zero fraction. There are two categories of NaN: QNaN (*Quiet NaN*) and SNaN (*Signalling NaN*).

A QNaN is a NaN with the most significant fraction bit set. QNaN's propagate freely through most arithmetic operations. These values pop out of an operation when the result is not mathematically defined.

An SNaN is a NaN with the most significant fraction bit clear. It is used to signal an exception when used in operations. SNaN's can be handy to assign to uninitialized variables to trap premature usage.

Semantically, QNaN's denote *indeterminate* operations, while SNaN's denote *invalid* operations.

Special Operations

Operations on special numbers are well-defined by IEEE. In the simplest case, any operation with a NaN yields a NaN result. Other operations are as follows:

Operation	Result
n ÷ ±Infinity	0
±Infinity × ±Infinity	±Infinity
±nonzero ÷ 0	±Infinity
Infinity + Infinity	Infinity
±0 ÷ ±0	*NaN*
Infinity - Infinity	*NaN*
±Infinity ÷ ±Infinity	*NaN*
±Infinity × 0	*NaN*

Summary

To sum up, the following are the corresponding values for a given representation:

Float Values (*b* = bias)

Sign	Exponent (*e*)	Fraction (*f*)	Value
0	00..00	00..00	+0
0	00..00	00..01 : 11..11	Positive Denormalized Real $0.f \times 2^{(-b+1)}$
0	00..01 : 11..10	XX..XX	Positive Normalized Real $1.f \times 2^{(e-b)}$
0	11..11	00..00	+Infinity
0	11..11	00..01 : 01..11	SNaN
0	11..11	10..00 : 11..11	QNaN
1	00..00	00..00	-0
1	00..00	00..01 : 11..11	Negative Denormalized Real $-0.f \times 2^{(-b+1)}$
1	00..01 : 11..10	XX..XX	Negative Normalized Real $-1.f \times 2^{(e-b)}$
1	11..11	00..00	-Infinity
1	11..11	00..01 : 01..11	SNaN
1	11..11	10..00 : 11.11	QNaN

References

A lot of this stuff was observed from small programs I wrote to go back and forth between hex and floating point (*printf*-style), and to examine the results of various operations. The bulk of this material, however, was lifted from Stallings' book.

1. *Computer Organization and Architecture*, William Stallings, pp. 222-234 Macmillan Publishing Company, ISBN 0-02-415480-6
2. IEEE Computer Society (1985), *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE Std 754-1985.
3. *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture* , (a PDF document downloaded from intel.com.)

See Also

- IEEE Standards Site
- *Comparing floating point numbers*, Bruce Dawson, http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm. This is an excellent article on the traps, pitfalls and solutions for comparing floating point numbers. Hint — epsilon comparison is usually the *wrong* solution.
- *x86 Processors and Infinity*, Bruce Dawson, http://www.cygnus-software.com/papers/x86andinfinity.html. This is another good article covering performance issues with IEEE specials on X86 architecture.
