Understanding the IEEE Floating-Point Standard

By Allen Samuels, Weitek Corp.

This article is the first in an occasional series on floating-point arithmetic. As chip densities have increased, floating-point units have become a standard part of high-performance microprocessors, yet many of the issues regarding floating-point are poorly understood by microprocessor users. This article explores the IEEE standard for binary floating-point arithmetic. Subsequent articles will cover hardware implementation, instruction set, and application issues. The author is an architect at Weitek Corp., where he has been responsible for the design of several high-performance floating-point units.

In the early days of computing, trying to write portable numerical programs was a lot like traveling through a swamp—it was easy to forget where you were going when you spent all of your time looking out for alligators. This situation was caused by every CPU manufacturer having their own proprietary floating-point arithmetic. A numerical program run on two different makes of computers would seldom deliver the same answer. All too often, different models from the *same manufacturer* would disagree on the correct answer.

So you ask, "Why even use the blasted things at all?" Yes, it is true, many programmers successfully avoid using floating-point numbers for their entire careers. However, for some things, nothing else solves the problem.

Most computer software is able to avoid the use of floating-point numbers by scaling to integers. For example, pennies are used in place of dollars internally. This can be surprisingly effective; a 64-bit integer is large enough to hold the entire U.S. gross national product, even when expressed in Argentinean Australs (currently ~10K to the dollar). Scaling is effective whenever the scale factor is fixed or when a single scaling factor can apply to several variables.

Many problems are stubbornly resistant to this type of attack. A good example is solving a system of linear equations. We all learned in linear algebra class the straightforward arithmetic of Gaussian elimination. What you learn when you try to implement this in a computer program is that each variable and coefficient may come from a completely different part of the number line. This requires that each variable, coefficient and intermediate computation have its own, private, scaling factor. Floating-point numbers solve exactly this problem.

The IEEE Floating-Point Standard

In the late '70s, a committee, under the auspices of the IEEE, created the IEEE standard for binary floating-point arithmetic (ANSI/IEEE Std 754-1985), which was formally adopted in 1985. The committee recognized the futility of trying to standardize any of the existing solutions (unlike the ANSI C committee whose *purpose* was to standardize the existing solution) or of trying to change any existing implementation. Instead, a standard for new architectures was created.

This strategy has been effective; virtually all new architectures since the start of the standardization effort have adopted it. Indeed, all desktop computers in common use today (except for DEC's micro-VAX series) are based on the IEEE standard. Non-adoptees seem to fall into two major categories. One category is companies that capitalize on the software of existing architectures and therefore need to copy their floating-point formats, (e.g., Convex and Cray). The other category is architectures that target special-purpose applications, such as digital signal processing (TI, NEC, etc.).

The IEEE standard covers the binary representation of floating-point numbers, operations involving a floating-point number, and the exceptional conditions that these operations can encounter. Specifically not covered by the standard is the relationship between programming languages and the specific features of the standard. This omission, necessary no doubt to allow its acceptance as a standard, has prevented the widespread use of many of the more exotic features.

Numerical Formats

The standard specifies four floating-point formats: two basic formats (single and double) and two extended formats (single-extended and double-extended). The two basic formats correspond to what we normally think of as single and double precision, i.e., 32- and 64-bit numbers, respectively. The extended format numbers are primarily used to hold intermediate results of computations and are not accessible in most standard programming languages. For this reason, many of the newer architectures have chosen not to support either of the extended formats while the remainder support only double-extended.

Single-extended is intended for machines that do not support double or double-extended. Supporting both is redundant. The extra precision of double-extended is very rarely needed. The 53 bits of fraction in the double format is enough to express the distance between the earth and the sun with an accuracy of ±16

microns (millionths of a meter)! All these variants are permitted by the standard, which requires only the single format.

The standard leaves many aspects of extended-format numbers up to the implementor. For example, the number of bits in an extended-format number is not specified; only a lower bound is given. For this reason, we will concentrate on the more rigorously defined single and double formats.

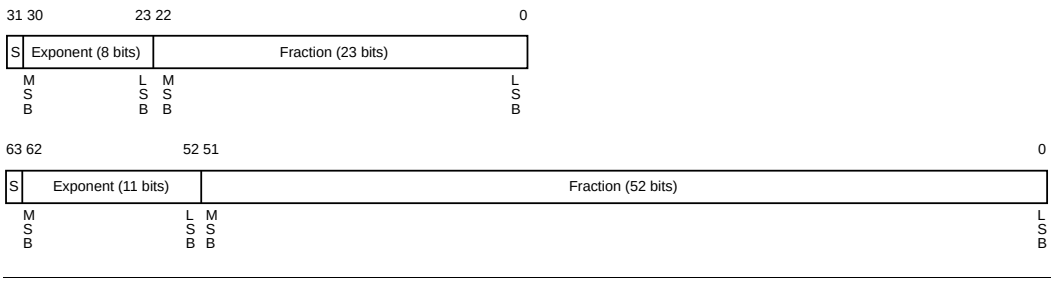

Figure 1. Single-precision (above) and double-precison IEEE formats.

A floating-point number consists of three fields: sign, exponent, and fraction (commonly, though incorrectly, called the mantissa). Figure 1 shows the format and width of single and double precision numbers. The sign field is a single bit (also called the sign bit). The exponent field of 8 bits for single-precision (SP) and 11 bits for double-precision (DP) is considered to be an unsigned number running from 0 to 255 (SP) or 2047 (DP). The fraction is interpreted differently depending on the exponent field.

As determined by the exponent and fraction field there are five classes of values: normalized, zero, infinity, denormalized and not-a-number (NaN).

Normalized Numbers

An exponent value that is not all zeros or ones (i.e., not 0, or 255 for SP or 2047 for DP), regardless of the fraction field, indicates a normalized number. A normalized number consists of a significand (which is the fraction with a leading "1." before it) between 1.0 and 2.0, multiplied by two raised to an integral power (derived from the exponent field). Negative numbers are indicated by setting the sign bit to one and leaving the other fields undisturbed. (This implies that numbers are stored in "sign magnitude" format, not the two's complement format common for binary integers.)

The exponent field is derived from the true exponent by adding a bias value of 127 (1023 for DP). Thus 2^1 is represented by an exponent field value of 128 (1024 for DP). By eliminating all-zero and all-one exponent fields, the true exponent range is restricted to –126 to +127 (–1022 to +1023 for DP), inclusive.

The significand range of 1.0 to 2.0 was carefully chosen. All numbers in this range are represented as 1.xxxxx in binary. The fraction field is created by discarding the leading "1." (also known as the hidden bit). Omitting the hidden bit provides an extra bit of precision than would otherwise be expected. Thus single format is considered to have 24 bits of precision even though only 23 bits are actually present!

Table 1 show some examples in single format. Normalized numbers are in the range 1.175×10^{-38} to 3.40×10^{38} (2.225×10^{-308} to 1.797×10^{308} for DP).

Zero and Infinity

A zero value for both the exponent and fraction fields indicates a zero. The standard supports both positive and negative zero. Negative zero can be produced by a large number of common operations and, except in a few obscure cases, behaves like positive zero.

An all-ones value for the exponent and a zero value for the fraction field indicates infinity. Again, the sign bit distinguishes positive from negative infinity. IEEE infinity behaves just as you learned in school: as the limiting case of arithmetic on operands of arbitrarily large magnitude. Infinity is produced by an overflowed computation.

Denormalized Numbers

A zero exponent field and a non-zero fraction field indicate a denormalized number. A denormalized number is similar to a normalized number except that the hidden bit is a zero and the true exponent is fixed at –126 (–1022 for DP). Denormalized numbers represent numbers smaller than the smallest normalized number, but the extended range comes with a steep price: reduced precision. The smallest denormalized numbers have only a few significant bits of precision—good enough to indicate the presence of a tiny non-zero number. The range of denormalized numbers is 1.75×10^{-38} to 10^{-45} (2.225×10^{-308} to 10^{-324} for DP).

Denormalized numbers are a controversial feature of the standard. Proponents applaud the "gradual underflow" close to zero rather than the abrupt transition to zero that is required in their absence. Opponents feel

IEEE	HEX	Actual	Decimal
1 01111111 00000000000000000000000	BF800000	$-(1.0 \times 2^0)$	-1.0
0 01111111 00000000000000000000000	7F800000	(1.0×2^0)	1.0
0 10000000 00000000000000000000000	40000000	(1.0×2^1)	2.0
0 10000000 10000000000000000000000	40400000	$(1.1_2 \times 2^1)$	3.0
0 11111110 00000000000000000000000	7F000000	(1.0×2^{127})	$\sim 1.7 \times 10^{38}$

Table 1. Examples of normalized, single-precision numbers.

that reduced precision generates misleading results.

Implementors of high-performance floating-point hardware find it expensive to handle denormalized numbers. Even the best implementations are unable to overcome inherent performance limitations. Many have chosen to emulate operations on these infrequently occurring numbers in software with corresponding savings in hardware. Some of these systems provide a method of treating denormalized numbers as zero so that even the expense of software emulation is avoided.

Not a Number (NaN)

An all-ones value for the exponent and a non-zero value of the fraction field indicate a Not-a-Number (NaN). As the name indicates, NaNs do not have an arithmetic value; they are intended to serve as implementation vehicles for a variety of features, including uninitialized variable detection, extended-range numerical values, diagnostic error values, error propagation, lazy evaluation, etc. Any operation with a NaN produces a NaN as a result. Any invalid computation (divide by zero, square root of a negative number, etc.) yields a NaN as its result.

There are two types of NaNs: signaling and quiet. Sadly, the standard is silent on their actual encodings other than requiring at least one of each type. A signaling NaN causes an exception whenever it is used. A quiet NaN propagates through computations silently. No computation generates a signaling NaN as its result; signaling NaNs must be introduced explicitly by the user.

Rounding

All floating-point computations can generate a result that requires infinite precision (e.g., an infinitely repeating binary fraction representing a value such as $^1/_5$). The process of deriving the finite precision result from the infinitely precise result is called rounding. In IEEE rounding the result is selected from the two numbers that bracket the infinitely precise result (i.e., the nearest that is smaller and the nearest that is larger). Four different rounding modes are specified: round to nearest, round to zero, round to negative infinity, and round to positive infinity.

Round-to-nearest specifies that the nearer of the two possible results be used. If the two results are equidistant (i.e., the case of .5 in decimal) then the one with the least-significant bit of zero is used (well, you gotta choose something!). Round-to-nearest mode is the most intuitively appealing and is specified as the default.

The other three modes, collectively known as directed rounding modes, choose the result that is nearer zero, $+\infty$ or $-\infty$, respectively.

Round-to-zero mode is used by C and FORTRAN for converting floating-point numbers to integers.

The round-to-infinity modes are useful for rounding error analysis. If you perform a sequence of computations twice, once in each of the modes, then you put a bound on the rounding error in the result.

Operations

The IEEE standard requires the following operations: add, subtract, multiply, divide, remainder, square root, compare, format conversion, and conversion of a floating-point number to an integral value in the same floating-point format. Surprisingly to many people, transcendental, logarithmic, and other mathematical functions are not part of the standard.

All the operations perform pretty much as expected except for the compare operation, which deserves further discussion. When I was in school, I learned that the compare operation generated only three possible results: less than, greater than, or equal to. Thanks to IEEE and the existence of NaNs, the possibilities now number four: less than, greater than, equal to, and unordered. Any compare operation involving a NaN (or two NaNs) yields unordered as the result. Further complicating matters, a compare operation can optionally request an exception if a NaN is involved.

Exceptions and Traps

Each exception has a status flag and a trap which can be enabled or disabled by the user. If an exception occurs and the trap is disabled, then the status flag is set, a result is delivered for the operation (the result is clearly specified by the standard), and execution continues. If the trap is enabled, then the executing program is suspended and the specified trap handler is invoked. The trap handler executes like a subroutine: when it completes, the executing program is resumed. The trap handler can substitute a result of its own for the result of the operation. There are five possible exceptions: invalid operation, division by zero, overflow, underflow, and inexact.

The invalid operation exception is signaled for any operation on a signaling NaN, any operation that is mathematically ill-defined (0/0 or $\infty - \infty$) or conversions of NaNs and infinities to formats that do not support them.

The inexact exception is signaled when the result differs from the infinitely precise result; i.e., rounding was performed. The overflow, underflow, and divide-by-zero exceptions are self-explanatory.

Programming Language Rules

The standard ignores its impact on high-level programming languages. For example, the extension of the compare operation does not fit into any of the modern programming languages.

The three possible results of the normal compare operation give rise to the six common programming language compare predicates: >, >=, <, <=, ==, and != (In C notation). The four possible results of the IEEE compare operation combined with the optional trapping on NaNs requires 26 different predicates to cover all of the useful combinations!

Exceptions are a problem area. The standard is very explicit about exception handling; a great deal of useful functionality must be supported by every implementation. However, since there are no required or recommended language bindings, each compiler and operating system vendor has a separate incompatible interface. This lack of compatibility causes programmers to avoid using the exception mechanism.

Summary

The IEEE standard has done an admirable job of draining the floating-point swamp. Scientific programmers can now write code and expect that the results will be essentially the same on a wide range of machines.

However, the working programmer tends to use only those features in the standard that are directly accessible in his or her favorite programming language. This has created an unfortunate situation. Microprocessor designers spend a great deal of time, energy, nanoseconds, and transistors to conform to the standard. (Trust me, some of the features are very difficult to implement efficiently.) Working programmers avoid these features, however, because every system they use accesses them differently.

Despite its omissions, the standard provides binary encodings, predictable answers, mathematically useful rounding, consistent error handling, and a sufficiently rich set of primitives to allow numerical programmers to concentrate on the problem being solved and not on tracking down errors introduced by some quirk in the latest version of some box. ♦

OOP

Continued from page 15

through a standardized object "Esperanto." There is, in fact, an effort underway to standardize the way objects communicate. With object communication occurring at a high level, programs can be broken up into user-interface objects that run on desktop computers and scientific, vectorizable objects that run on multiprocessor supercomputers. Heterogeneous computer networks could become advantageous instead of bothersome. And, of course, the network could be inside the computer, enabling desktop multiprocessors with x86, RISC, and vector processors to "transparently" exploit the benefits of each processor.

OOP holds great promise for accelerating the application development process. While it is unlikely that OOP will allow non-programmers to suddenly be able to develop sophisticated applications, it will make it possible for more people to become programmers. If application developers wish it, OOP should allow the creation of applications that are end-user customizable, at least to a certain extent. An example is the NeXT IB's ability to accept third-party user-interface objects.

Will object-oriented systems cause a mass migration to a new operating system? The answer to this question is "perhaps," but it is likely to have a familiar name like "ObjectDOS," "WinObj," or "Macintobject" because it is possible to layer many of the important features of object-orientation on top of existing systems. Operating systems designed from the ground up to incorporate and support object-orientation—such as the one Taligent is building—will no doubt have compelling advantages, but the success of the PC clone over the Macintosh proves that technical advantages are not always enough to sway market demand.

Will the move to object-orientation unseat the x86 family as the dominant force in desktop microprocessors? The answer to this is "almost definitely no." Certainly Microsoft will continue to support the x86, NeXT has already ported NeXTstep to the 486, Sun is porting its software to the 486 and will no doubt continue to do so as more and more object-orientation is incorporated, and Taligent will make its product available on a variety of platforms including x86-based computers.

There is one other way object-orientation could affect microprocessors. As some university and industry experiments have proven, it is possible to use special hardware to accelerate some of the dynamic aspects of object-oriented systems, but since traditional architectures are fully capable of supporting OOP systems, it seems unlikely that microprocessors will shift away from conventional organizations anytime soon. ♦

To Learn More

A wealth of information and tools is available to those interested in learning more about object-orientation. A good book is *Object Orientation: Concepts, Languages, Databases, User Interfaces* by Khoshafian & Abnous, published by J. Wiley. Rambaugh, et. al have a book on the general topic of object-oriented design called *Object-oriented Modeling and Design* (Prentice Hall). Pinson & Wiener's book *Objective-C*, published by Addison-Wesley, describes the original NeXT programming language. Borland and Microsoft have OOP systems for PCs, ThinkC and Prograph are OOP systems available for the Macintosh, and Smalltalk V is available for both. The glossy document *The NeXTstep Advantage* from NeXT serves as a general introduction to the NeXT OOP system. Finally, there is *Object Magazine* (SIGS Publications) for those who need an up-to-date, monthly dose of object-orientation. Actually, there is a wide variety of magazines, including technical journals, that deal with OOP. Even RISC cannot make such a bold claim.