Remarks on the IEEE Standard for Floating-Point Arithmetic

David M. Russinoff

November 29, 2015

Although the publication of the Institute of Electrical and Electronics Engineers labeled IEEE-754-2008 is generally regarded as the ultimate standard of correctness for implementations of floating-point arithmetic, I shall argue that as a prescriptive specification, it is conspicuously deficient. If a camel is a horse designed by a committee, then this document, a collaboration of ninety-two leading experts in the field of computer arithmetic ratified by one hundred knowledgable reviewers, is the ruminant of computing standards.

IEEE-754-2008 is the long-deliberated revision of IEEE-754-1985, which bore the title "IEEE Standard for Binary Floating-Point Arithmetic". With the benefit of twenty-three years of post-mortem analysis, it extends the original in various directions, including a treatment of decimal data formats, resulting in a fourfold increase in length (fifty-eight pages vs. fourteen). But perhaps the most ambitious objective undertaken in the revision is its overall-stated "purpose":

> This standard provides a method for computation with floating-point numbers that will yield the same result whether the processing is done in hardware, software, or a combination of the two. The results of the computation will be identical, independent of implementation, given the same input data. Errors, and error conditions, in the mathematical processing will be reported in a consistent manner regardless of implementation.

The first sentence is essentially inherited from the 1985 version, but the other two suggest a radical detarture from the original, which allowed some freedom of interpretation. Are pre-existing compliant implementations now expected to be redesigned to conform to a new order? One need not read beyond page 2 to find that this is not at all the case:

> This standard does not define all aspects of a conforming programming environment. Such behavior should be defined by a programming language definition supporting this standard, if available, and otherwise by a particular implementation. Some programming language specifications might permit some behaviors to be defined by the implementation.

Indeed, we find in the sequel that many critical aspects of behavior are unspecified, and that compliant implementations may produce widely divergent computational results. But this sets

a precedent that is followed throughout: it is a mistake to assume that the author of one page is in communication with that of the next.

Perhaps the first obligation of a proposed standard for this domain, which centers on the representation of abstract numbers by binary sequences, is a characterization of the objects to be represented and those that represent them, as distinct entities, as this would seem to be a prerequisite for any meaningful discussion of such a representation. This was addressed in the 1985 version by the following definition:

> **binary floating-point number**: a bit-string characterized by three components: a sign, a signed exponent, and a significand. Its numerical value, if any, is the signed product of its significand and two raised to the power of its exponent. In this standard a bit-string is not always distinguished from a number it may represent.

Here is a creative solution to the problem of characterizing the correspondence between real numbers and their encodings with respect to a floating-point format: we simply pretend that there is no distinction between the two entities and ignore the problem altogether. Needless to say, nothing useful came of this approach—it is often difficult to ascertain, in reading the subsequent text, what sort of object is under discussion at any given point. To their credit, the committee of 2008 seems to have taken its responsibility in this regard somewhat more seriously, identifying four distinct classes of relevant entities, listed in Table 3.1 as the "different specification levels for a particular format": extended real numbers, floating-point data, floating-point representations, and bit strings. The curious 1985 disclaimer regarding their distinction, however, is preserved:

> **floating-point datum**: A floating-point number or non-number (NaN) that is representable in a floating-point format. In this standard, a floating-point datum is not always distinguished from its representation or encoding.

This might be defensible if the mapping between numbers and representations were bijective, but we find later (Table 3.1) that this is not the case. The definition provides no clue to the nature of a "non-number" or how many of them exist, but it later transpires that there is only one such datum, denoted simply as "NaN". The remaining data are characterized by the following definition (in which the indefinite article preceding "NaN" serves only to confuse):

> **floating-point number**: A finite or infinite number that is representable in a floating-point format. A floating-point datum that is not a NaN. All floating-point numbers, including zeroes and infinities, are signed.

Thus, the set of floating-point numbers determined by any format contains the four distinguished elements $+0$, -0, $+\infty$, and $-\infty$; the rest are the finite non-zero numbers that are representable with respect to the format, according to a scheme specified as follows:

> **floating-point representation**: An unencoded member of a floating-point format, representing a finite number, a signed infinity, a quiet NaN, or a signaling NaN. A representation of a finite number has three components: a sign, an exponent, and a significand; its numerical value is the signed product of its significand and its radix raised to the power of its exponent.

This provides a clue to the nature of formats, which is confirmed by the next definition:

> **format**: A set of representations of numerical values and symbols, perhaps accompanied by an encoding.

Neither *encoding* nor *bit string* is listed in the glossary, but the following assertion is found in Section 3.2:

> An *encoding* maps a representation of a floating-point datum to a bit string.

Nor is there an entry for *extended real number*, but this is explained in the same section:

> The mathematical structure underpinning the arithmetic in this standard is the extended reals, that is, the set of real numbers together with positive and negative infinity.

A definition is included for one of the three components of a floating-point representation:

> **significand**: A component of a finite floating-point number containing its significant digits. The significand can be thought of as an integer, a fraction, or some other fixed-point form, by choosing an appropriate exponent offset. A decimal or binary subnormal significand can also contain leading zeroes.

The following related definition is also of interest:

> **trailing significand field**: A component of an encoded binary or decimal floating-point format containing all the significand digits except the leading digit. In these formats, the biased exponent or combination field encodes or implies the leading significand digit.

It is unfortunate that these definitions, which are critical to an overall understanding of the standard, are presented as an alphabetized glossary rather than in a more logical sequence befitting a mathematical exposition. Various difficulties encountered in relating the definitions to the ensuing discussion raise a number of questions that suggest considerable confusion within the standard committee:

- *What is the radix of a floating-point number?*

  The glossary defines a *binary* (resp., *decimal*) *floating-point number* to be a "floating-point number with radix two" (resp., "ten"). But as we have seen, a floating-point number is either a signed zero, a signed infinity, or an ordinary non-zero real number, none of which possesses an intrinsic radix. It might make more sense to associate a radix with a format than with a number.

- *What is a significand?*

  In the definition of *floating-point representation*, and again in Section 3.2, we are told that a significand is a component of a floating-point representation, and that it occurs as a factor in the computation of the represented number, implying that it is itself a number. On the other hand, according to the definition of *significand* itself, it is instead a component of a floating-point number, and that it "contain[s] its significant digits" and "can also contain leading zeroes", suggesting that it is something other than a simple number. (What digits does a number "contain"?)

  It is true that we have been warned that the distinction between numbers and representations will not always be respected, but since a given number admits a variety of representations, how can we know which of them determines the significand of the number?

  Furthermore, according to the definition, a significand "can be thought of" as an integer, a fraction, or something else. Is it not the purpose of a definition to specify the nature of the thing being defined? Is this an issue on which there was no meeting of the ninety-two minds?

- *A trailing significand field is a component of what sort of object?*

  According to the definition, it is a component of an "encoded binary or decimal floating-point format", but what could that possibly mean? We have been told that floating-point representations, rather than formats, are the things that are encoded. Whatever an encoded format may be, it apparently includes a "combination field", but since this term appears nowhere else in the document, there is no way to know what it means.

- *Does a format include a specific encoding?*

  According to the definition, a *format* is "perhaps accompanied by an encoding." Well, is it or not?

- *What value is represented by the exponent field of a denormal encoding?*

  In Section 3.4, we find that the exponent field of the encoding of any floating-point number with respect to a binary interchange format is $E = e + bias$, and that in the case of a subnormal number, $e = emin$ and E has the reserved value 0. But we see in Section 3.3 that $emin = 1 - emax$, and according to Table 3.5, $bias = emax$. Thus, for a subnormal number,

  $$E = e + bias = emin + emax = 1 \neq 0.$$

- *How many non-numerical floating-point data are there?*

  In Section 3.3, qNaN and sNaN are identified as distinct floating-point data, but according to Table 3.1, as confirmed by item (a) in the middle of page 9, there is only one NaN datum, which admits qNaN and sNaN as representations.

- *What is the result of rounding 0?*

  In Table 3.1, rounding is characterized as a "many-to-one" mapping from the extended real numbers to the set of floating-point data, implying that every extended real is rounded to a unique datum. It seems likely that the datum to which the number 0 is rounded is either $+0$ or -0, perhaps depending on the rounding mode. Which is it?

- *What information is provided by the sign bit of a zero?*

  According to Section 3.3, "For a floating-point number that has the value zero, the sign bit s provides an extra bit of information." What is the nature of that information?

  In Section 6.1, the significance of infinite operands is explained as follows:

  > The behavior of infinity in floating-point arithmetic is derived from the limiting cases of real arithmetic with operands of arbitrarily large magnitude, when such a limit exists.

  For example, if x is a positive number, then since multiplication of x by a large negative number yields a large negative number, $x \cdot (-\infty) = -\infty$. It might make sense for the standard to include an analogous statement about zeroes, such as this:

  > The behavior of signed zeroes in floating-point arithmetic is derived from the limiting cases of real arithmetic with non-zero operands of arbitrarily small magnitude with the same sign, when such a limit exists.

  In fact, this is consistent with most floating-point operations. For example, if x is a positive number, then since division of x by a small negative number yields a large negative number, $x/(-0) = -\infty$. The glaring exception is the square root operation. As prescribed in 1985 and repeated in 2008, the square root of -0 is -0, a counter-intuitive result that violates the usual defining identity $(\sqrt{x})^2 = x$ as well as the principle proposed above, since the square root of a small negative number is undefined.

Another subject that warrants attention is the handling of exceptions. The reader who takes the droll definition of *non-computational operation*, "an operation that is not computational", as a sign that no term will go undefined will be sorely disappointed by the following entry:

> **exception**: An event that occurs when an operation on some particular operands has no outcome suitable for every reasonable application. That operation might signal one or more exceptions by invoking the default or, if explicitly requested, a language-defined alternate handling. Note that *event*, *exception*, and *signal* are defined in diverse ways in different programming environments.

Is this a definition or an apology? It certainly represents a retreat from the more definitive position taken in 1985:

> There are five types of exceptions that shall be signaled when detected. The signal entails setting a status flag, taking a trap, or possibly doing both. With each exception should be a trap under user control, as specified in Section 8. The default response to an exception shall be to proceed without a trap. This standard specifies results to be delivered in both trapping and nontrapping situations.

The specified results were designed to reflect the architecture of the pre-existing Intel 80x87 coprocessor, the first "IEEE-compliant" floating-point system. The only ambiguities in the specification are in the handling of underflow, with respect to both (a) its detection, which may be based on either the unrounded or rounded result at the discretion of the implementation, and (b) the value returned in the absence of an enabled trap, "which might be zero, denormalized, or $\pm 2^{\mathrm{E}min}$. No explanation is offered, either for this latitude or for its not being similarly extended to the handling of overflow.

The main problem faced by the committee of 2008 is that a variety of architectures and instruction sets appeared during the intervening period that do not conform to the original standard, especially in the "trapping" case. In a convoluted attempt to account for these developments, the new standard distinguishes between "default exception handling" and "alternate exception handling". The former roughly corresponds to the behavior previously associated with the "trapping situation", although a number of variations with respect to the raising of flags and the values returned are allowed. The latter is further decomposed according to various "attributes", which may or may not be provided by a language or an implementation. An exception handling attribute may be "immediate", which means that control is to be transferred to a handler block "as soon as possible", or "delayed", meaning that the exception is handled by default until the associated block terminates normally, at which time control is transferred. After execution of the handler, control may or may not be returned to the point at which the exception was signaled. In contrast to the 1985 standard, nothing is stated about the value returned in the case of alternate exception handling, which is apparently assigned at the full discretion of the implementation.

In short, the new standard is not a specification of behavior at all, but rather a futile attempt to provide a framework to accommodate all conceivable behaviors. What is most puzzling is that it nevertheless fails to address a number of exception handling issues that are relevant to contemporary architectures. One of the six industry-standard floating-point exceptional conditions, the denormal operand, is not mentioned at all. Nor is there any discussion of the interaction between exceptions that arise during the execution of instructions that perform several operations in parallel, such as the Streaming SIMD (single instruction, multiple data) Extensions to the x86 architecture, which have been in existence since 1999.

While the drawbacks of "design by committee" are apparent, one might hope for a silver lining: when a document benefits from the contributions and services of so many participants and reviewers, shouldn't it be expected that any glaring deficiencies be exposed prior to its final publication? Apparently not—although at least two members of the intersection of those two groups are known to be capable of a careful reading of such a document, there is no evidence that this occurred. We might also expect at least one reviewer to hold a sufficient command of English to recognize simple distinctions in the accepted usage of common words, as between *assure* and *ensure*, or *that* and *which*. However, at the top of page 1, we find the

"IMPORTANT NOTICE" that "This standard is not intended to assure [*sic*] safety, security, health, or environmental protection in all circumstances" (as if we might be tempted to believe otherwise), and on page 5, *precision* is defined as "The maximum number p of significant digits that can be represented in a format, or the number of digits to that [*sic*] a result is rounded."

There is no doubt that IEEE-754-2008 could be improved by removing the inconsistencies, anbiguities, and errors that we have brought to light, but there are deeper issues that are more difficult to address. The stated objective of establishing consistent behavior, independent of implementation, is in conflict with the tacit requirement of accommodating a spectrum of established architectures and industry standards. The latter consideration has resulted in a "standard" that is less a prescriptive specification than a descriptive account of prevailing behavior, and therefore offers little in the way of guidance to architects or implementors. Moreover, since the divergent features of existing architectures are not specifically identified, its utility to application programmers is also limited. For this purpose, a comprehensive specification of a particular instruction set would be far more useful. It is remarkable that all existing architectural programming manuals fail so miserably in this regard, but that is a subject for another rant.