Change and Open Issues Log

29,30 May 2003
Decimal format with scale preservation accepted, along with some concomitant changes. More refinements to section 3. Added equivalent predicate to recommended functions.
Open issues:
• preferred exponents are not defined for all decimal operations.
• canonical infinity needs to be agreed upon
• recommended functions need updates for decimal

18 April 2003
Refinements to section 3; reversed decimal signaling/quiet NaN encoding. Format definitions added, max/min accepted, many small changes approved.

20 March 2003
No substantive changes. Some minor rewordings during draft review, especially in the formats section.

20 February 2003
Basic format names: we decided on binary32, binary 64, binary128, decimal32, decimal64, decimal128.
Required basic formats: if 64 bits is implemented, so must be 32; if 128, so must be 64 and 32. One may implement binary or decimal or both.
Extended formats: a discussion should be added to the formats section, perhaps to be moved later to the expression evaluation section.
Canonical representations will be removed.
Signaling NaNs will be restored to their former mandatory state (2002-07-26 draft).
In the glossary, the operations preserving quiet NaNs and noticing signaling NaNs will be listed.

23 January 2003

§DECIMAL – add dense packed decimal formats.
[Corollary changes: removed extended precision format specifications and restrictions on combinations of formats.]

21 November 2002

§MINMAX – add min(x,y) and max(x,y).

17 October 2002

§ “programmer” -> “user” consistently.

19 September 2002

§RAISE – the nomenclature for status flags will be “raised and lowered” instead of “set and reset”.

22 August 2002

§TRAP – traps are moved to appendix 8. Sections 7 and 8 are renamed Default Exception Handling and Alternate Exception Handling.

§PRED-TABLE Proposal for Comparison Predicate Table – a reorganization of Table 4 for comparison predicates to try to indicate what we really meant all along.

21 March 2002

§BIAS-ADJUST We decided to hyphenate bias-adjust.

21 February 2002

§GROSS Rules for gross over/underflow are revised to either deliver the original operand or a scaled result and an explicit scale factor.
§NU nextup function – intended to replace nextafter in contexts where the second argument of nextafter would actually be a constant.  Rationale:  used much more often, more likely to be implemented as a machine instruction.

§CONV – stringtofloating() is 854’s renamed conv() function for run-time string to floating point conversion.

17 January 2002

§NEXTNX nextafter(x,y) generates y if x==y.

§AINTNX The integral-value-in-floating-point-format conversion is never inexact.

§PRED Predicates listed in Appendix A corresponding to entries in Table 4 will be removed.

§FMA Fused multiply-add defined in glossary.

13 December 2001

§CLASSPRED Add individual quiet predicates.
Rationale: I am not sure if I have ever used case (class(..., when I was concerned about performance; in that case I typically use a sequence like isnormal – finite – iszero; isinf, issignaling, in order to get to the common case first.

Simple one-operand logical predicates have a higher chance of hardware implementation, programmer usage in performance-oriented code, and correct compiler optimization.

Classification predicates (Section 5.9) and functions (Appendix A) are much more efficient if hardware implementations of floating-point registers classify their contents as they are loaded or computed, and record the classification into extra tag bits for each register.

§BI Standardize quiet functions.
Rationale: these functions are usually implemented without exceptions.

§FMA Fused multiply-add.
Rationale: Standardize a best practice for fused multiply-add operations, which are now available in several instruction sets, often implemented slightly differently.  Note that 0 ∞ + qnan does not signal invalid, because a NaN is an operand, but 0 ∞ + ∞ does signal invalid, and generates qnan rather than ∞, because no operand is NaN, and even if all three operands have positive sign bits, the product is still undefined.  In general, an operation with a floating-point

Copyright © 2003 by the Institute of Electrical and Electronics Engineers, Inc. This document is an unapproved draft of a proposed IEEE-SA Standard - USE AT YOUR OWN RISK. See statement on page 1.
destination can generate an invalid exception if no operand is a quiet NaN, and generates no invalid exception if one or more operands is a quiet NaN and none is a signaling NaN.

12 November 2001

§Q Quad: Add 128-bit quadruple precision with 112 fraction bits and implicit integer bit. Rationale: match existing hardware and software implementations, and discourage undesirable alternatives such as double-double.

§1 New running footer for IEEE drafts substituted for published 754 copyright notice. Rationale: comply with IEEE rules.

11 April 2001

§4 SCOPE and PURPOSE defined. Rationale: previous intent was not universally understood. OPEN ISSUE: too narrow a purpose? Is uniqueness specified and achievable? OPEN ISSUE: merge the new PURPOSE: with the Foreword; merge the new and existing SCOPE: to remove redundancy.

§3 EXCEPTION and TRAP defined in glossary. Rationale: too many confusing usages of these terms in the whole of computer science.

§2 “denormal” -> “subnormal” almost everywhere. Rationale: conform to 854's better usage.

12 Feb 2001

§6 Rounding precision modes SHOULD modify exponent range to mimic base precision. Rationale: languages like Java specify basic-precision arithmetic that is expensive to simulate on extended-based systems that do not modify exponent range when precision is shortened. The footnote admonition against instructions that combine operands of a higher precision and produce a result of lower precision with only one rounding is there because such implementations are very costly to emulate on conventional architectures.
FOREWORD

(This Foreword is not a part of ANSI/IEEE Std 754–1985, IEEE Standard for Binary Floating-Point Arithmetic.)

This standard is a product of the Floating-Point Working Group of the Microprocessor Standards Subcommittee of the Standards Committee of the IEEE Computer Society. This work was sponsored by the Technical Committee on Microprocessors and Minicomputers. Draft 8.0 of this standard was published to solicit public comments. [FOOTNOTE 1: Computer Magazine vol 14, no 3, March 1981.] Implementation techniques can be found in An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic by Jerome T. Coonen, [FOOTNOTE 2: Computer Magazine vol 13, no 1, January 1980.] which was based on a still earlier draft of the proposal. §4.

PURPOSE: This standard provides a discipline for performing floating-point computation that yields results independent of whether the processing is done in hardware, software, or a combination of the two. For operations specified in this standard, numerical results and exceptions are uniquely determined by the values of the input data, sequence of operations, and destination formats, all under user control. §4.

This standard defines a family of commercially feasible ways for new systems to perform binary and decimal floating-point arithmetic. The issues of retrofitting were not considered. Among the desiderata that guided the formulation of this standard were

1. Facilitate movement of existing programs from diverse computers to those that adhere to this standard.

2. Enhance the capabilities and safety available to users programmers who, though not expert in numerical methods, may well be attempting to produce numerically sophisticated programs. However, we recognize that utility and safety are sometimes antagonists.

3. Encourage experts to develop and distribute robust and efficient numerical programs that are portable, by way of minor editing and recompilation, onto any computer that conforms to this standard and possesses adequate capacity. When restricted to a declared subset of the standard, these programs should produce identical results on all conforming systems.
4. Provide direct support for
   a. Execution-time diagnosis of anomalies
   b. Smoother handling of exceptions
   c. Interval arithmetic at a reasonable cost

5. Provide for development of
   a. Standard elementary functions such as exp and cos
   b. Very high precision (multiword) arithmetic
   c. Coupling of numerical and symbolic algebraic computation

6. Enable rather than preclude further refinements and extensions.
An American National Standard

IEEE Standard for Binary Floating-Point Arithmetic

1. Scope

1.1. Implementation Objectives

SCOPE: This standard specifies formats and methods for binary and decimal floating-point arithmetic in computer programming environments: standard and extended functions in 32-, 64-, and 128-bit basic formats single, double, quad §Q, and extended precision formats, and recommends formats for data interchange. Exception conditions are defined and default handling of these conditions is specified. §4

It is intended that an implementation of a floating-point system conforming to this standard can be realized entirely in software, entirely in hardware, or in any combination of software and hardware. It is the environment the programmer or user of the system sees that conforms or fails to conform to this standard. Hardware components that require software support to conform shall not be said to conform apart from such software.

1.2. Inclusions

This standard specifies

1. Basic and extended floating-point number formats

2. Add, subtract, multiply, divide, fused multiply-add, square root, remainder, and compare operations

3. Conversions between integer and floating-point formats

4. Conversions between different floating-point formats

5. Conversions between basic format floating-point numbers and external decimal formats decimal strings

6. Floating-point exceptions and their handling, including nonnumbers (NaNs)

1.3. Exclusions
This standard does not specify

1. Formats of decimal strings and integers
2. Interpretation of the sign and significand fields of NaNs
3. Conversions between extended formats and external decimal formats
   Binary <-> decimal conversions to and from extended formats
2. Definitions

**basic format.** A format whose set of representable values is fixed by this standard. This standard also specifies the encoding of basic formats.

**biased exponent.** The sum of the exponent and a constant (bias) chosen to make the biased exponent's range nonnegative.

**floating-point number.** A bit-string encoding 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 its radix $two$ raised to the power of its exponent. In this standard a bit-string is not always distinguished from a number it may represent.

**binary floating-point number.** A floating-point number with radix two.

**cohort.** In a decimal format, the set of different representations of the same numerical value.

[This term’s name is subject to change. Guy Steele offers “clan,” “clique,” “clump,” or “flock” as alternatives. I favor cohort since as term used to describe a set of warriors, it implies a shared purpose. Also, cohort has a vague decimal connection since the term was used to describe one of the ten divisions of a Roman legion. -JDD]

**decimal floating-point number.** A floating-point number with radix ten.

**declet.** An encoding of three decimal digits into ten bits using the densely packed decimal encoding scheme.

**denormalized number.** See subnormal number.

**destination.** The location for the result of an operation upon one or more operands a binary or unary operation. A destination may be either explicitly designated by the user or implicitly supplied by the system (for example, intermediate results in subexpressions or arguments for procedures). Some languages place the results of intermediate calculations in destinations beyond the user's control. Nonetheless, this standard defines the result of an operation in terms of that destination's format and the operands' values.

**exception.** (REJECTED) An event that occurs when an operation has no outcome suitable for every reasonable application. §3 That operation might signal one or more exceptions (listed in section 7) by invoking the
default (section 7) or user-specified alternate (section 8) handling. Note that “event,” “exception,” and “signal” are defined in diverse ways in different programming environments.

exponent. The component of a binary floating-point number that normally signifies the integer power to which the radix two is raised in determining the value of the represented number. Occasionally the exponent is called the signed or unbiased exponent.

extended format. A format whose set of representable values is constrained to contain a particular set of values as a subset; values outside the specified set may be representable too. This standard does not specify the encoding for extended formats.

external decimal format. A decimal format intended to be more readily interpreted by humans rather than computers; for example, the strings of decimal floating-point literals in a particular programming language.

fraction. The field of the significand that lies to the right of its implied binary point.

format. A set of binary or decimal floating-point values together with an encoding for that set of values.

fused multiply-add. The operation fma(a, b, c) computes (a×b)+c as if with unbounded range and precision, rounding only once, to the destination format. Thus no underflow, overflow, or inexact exception (section 7) can arise due to the multiply, but only due to the add; and so fused multiply-add differs from a multiply operation followed by an add operation. §FMA

mode. A variable that a user may write, read, set, sense, save, and restore to control the execution of subsequent arithmetic operations. The default mode is the mode that a program can assume to be in effect unless an explicitly contrary statement is included in either the program or its specification. The following mode shall be implemented: rounding, to control the direction of rounding errors. In certain implementations, rounding precision may be required, to shorten the precision of results.

The user may enable alternate exception handling modes. The implementor may, at his option, implement the following modes: traps disabled/enabled, to handle exceptions. §TRAP
[A program that does not inherit modes from another source, begins execution with all modes default.]

_NaN_. Not a Number, a symbolic entity encoded in floating-point format. There are two types of NaNs (6.2), _quiet and signaling_. _Except as specified in section 6.2_, quiet NaNs propagate through almost every arithmetic operations without signaling exceptions, while signaling NaNs signal the invalid operation exception (7.1) whenever they appear as operands.

_If we're going to retain signaling NaNs in a useful way, they should probably signal on operations that would change numerical values, such as negate and abs. Like symbolic links in Unix, you shouldn't be able to detect a signaling NaN pointing to a number except by asking if it's a signaling NaN by an issignaling predicate or a class function._

**normal number.** For a particular format, a nonzero finite floating-point number with magnitude greater than or equal to \( b^{emin} \) (see tables 1.a and 1.f). Normal numbers can use the full precision available in a format. This standard treats zero as neither normal nor subnormal.

_(previous normal definition)_ In a particular basic format, a finite nonzero floating-point number that can be represented in that format with an implicit (binary) or explicit (decimal) nonzero leading significand digit. This standard treats zero as neither normal nor subnormal.

**radix.** The base for the representation of binary or decimal floating-point numbers, two or ten.

**result.** The bit string (usually representing a number) that is delivered to the destination.

**shall.** The use of the word _shall_ signifies that which is obligatory in any conforming implementation.

**should.** The use of the word _should_ signifies that which is strongly recommended as being in keeping with the intent of the standard, although architectural or other constraints beyond the scope of this standard may on occasion render the recommendations impractical.
**signal. (REJECTED)** When an operation has no outcome suitable for every reasonable application, that operation might signal one or more exceptions (listed in section 7) by invoking the default (section 7) or user-specified alternate (section 8) handling. Note that “exception” and “signal” are defined in diverse ways in different programming environments.

**significand.** A component of an unencoded binary or decimal floating-point number containing its significant digits. The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right. In this standard, the significand is represented as an integer. By redefining the bias, the significand may be thought of as an integer, a fraction, or some other fixed-point form.

**status flag.** A variable that may take two states, raised or lowered set and clear. A user may raise a status flag, lower it, test it, clear a flag, copy it, or restore it to a previous state. When raised set, a status flag may convey additional system-dependent information, possibly inaccessible to some users. The operations of this standard, when exceptional, may as a side effect raise set some of the following status flags: inexact result, underflow, overflow, divide by zero, and invalid operation. [A program that does not inherit status flags from another source, begins execution with all status flags lowered.]

**subnormal denormalized number.** In a particular format, a nonzero floating-point number with magnitude less than the magnitude of that format's smallest normal number. A subnormal number cannot be represented in that format with an implicit (binary) or explicit (decimal) nonzero leading significand digit. exponent has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero. Supersedes 754–1985's denormalized number. §2

**user.** Any person, hardware, or program not itself specified by this standard, having access to and controlling those operations of the programming environment specified in this standard.
3. Formats

This standard defines three basic floating-point formats in three lengths: 32, 64, and 128 bits, and in two radices: binary formats based on radix two and decimal formats based on radix ten. A programming environment conforms to this standard, in either or both radices, by providing one of these basic formats and all shorter basic formats: thus an implementation providing a 64-bit format shall provide the 32-bit format in the same radix, and an implementation providing a 128-bit format shall provide the 32- and 64-bit formats in that radix.

[An implementation providing only decimal32 would not be useful for much, but neither would it harm the standard to allow it. An implementation providing single and quad conforms if it calls quad “extended” which is why I don't think specifying allowable combinations of formats is a meaningful exercise.]

[Note that 854 requires an implementation to have a single base; from 854 section 3.1 “[The base] b shall be either 2 or 10 and shall be the same for all supported precisions.” -JDD]
3.1. Sets of Representable Numerical Values

This section primarily specifies the sets of numerical values representable within floating-point formats; the encodings for those values in basic formats are discussed in sections 3.2 and 3.3. The set of finite numerical values representable within a particular format is determined by the settings of the following integer parameters:

- $b =$ the radix, 2 or 10
- $p =$ the number of significant digits (precision)
- $emax =$ the maximum exponent
- $emin =$ the minimum exponent.

The values of these parameters for each basic format are given in Table 1a; constraints on these parameters for extended formats are given in Table 1f. Basic formats are named according to the number of bits in their encoding. Within each basic or extended format, the following entities shall be provided:

- Positive and negative zero
- Nonzero numbers of the form $(-1)^s b^e b^{1-p} m$, where
  - $s$ is 0 or 1 [$s$ is any integer $0 <= s <= 1$]
  - $e$ is any integer $emin <= e <= emax$
  - $m$ is any integer $0 < m <= b^p - 1$
- Two infinities, $+\infty$ and $-\infty$
- Quiet and Signaling NaNs

In a basic format, these are the only entities provided. In this standard, the significand $m$ is regarded as an integer. Besides being thought of as an integer, by associating some portion of the scaling factor $b^{1-p}$ with the significand, the significand may be thought of as a rational value greater than or equal to 0 and less than $b$ ($0 <= m_{scaled} = b^{1-p} m < b$), or some other fixed-point form.
The smallest normal magnitude number is $b^{emin}$ and the largest is $b^{emax}(b - b^{1-p})$. The nonzero values with magnitude less than $b^{emin}$ are called subnormal because their magnitudes lie between zero and the smallest normal magnitude. One reason to distinguish between normal and subnormal binary numbers is that formats may encode these two kinds of numbers differently. Every finite representable number is an integral multiple of the smallest subnormal magnitude $b^{emin}b^{1-p}$.

For any variable that has the value zero, the sign bit $s$ provides an extra bit of information. Although all formats have distinct representations for +0 and –0, the signs are significant in some circumstances, such as division by zero, but not in others (see section 6.3). In this standard, 0 and $\infty$ are written without a sign when the sign is not important.

[In this section, it may be appropriate to note and explain the differences between the real numbers (or real numbers extended with signed infinities) and a format’s approximation to real numbers with respect to zero. In other words, while the real number system only has a single zero, floating-point numbers have both positive and negative zero for pragmatic reasons to make certain computations easier to formulate (branch cuts, etc.) and to allow more regular rules (e.g. defining sign of 1/0). -JDD]

<table>
<thead>
<tr>
<th>Summary of Basic Format Parameters: Representable Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
</tr>
<tr>
<td>$p$ digits</td>
</tr>
<tr>
<td>$emax$</td>
</tr>
</tbody>
</table>

Table 1a: Representable Value Parameters

[Note: These choices for $emin$, $emax$, and $p$ meet the constraints of section 3.1 of IEEE 854:

- $(emax-emin)/p$ shall exceed 4 and should exceed 10
- $b^{p-1} >= 10^5$

However, the decimal exponents to not meet the recommendation that “$b^{(emax-emin+1)}$ is the smallest integral power of $b$ that is $>= 4$.” -JDD]
3.2. Basic Binary Format Encodings

Each representable nonzero numerical value has just one encoding in a basic binary format. To make the encoding unique, in terms of the parameters in section 3.1, the value of the significand \( m \) is maximized by decreasing the exponent value until either the exponent is \( e_{\text{min}} \) or \( m \geq 2^{p-1} \). After this normalization process is done, if the exponent is \( e_{\text{min}} \) and \( m < 2^{p-1} \), the value is subnormal. Subnormal numbers (and zero) are encoded with a reserved exponent value.

Numbers in the binary formats are encoded in the following three fields as shown in Fig. 1-2:

1. 1-bit sign \( S \)
2. \( w \)-bit biased exponent \( E = e + \text{bias} \)
3. \( t = p-1 \) -bit trailing significand \( T = d_1 d_2 \ldots d_{p-1} \); the leading bit of the logical significand, \( d_0 \), is implicitly encoded in the exponent \( E \).

![Figure 1-2 Basic Binary Floating-Point Format](image)

The values of \( w \), \( \text{bias} \), and \( t \) for the basic binary formats are listed in Table 1b.

The range of the representation's biased exponent \( E \) shall include:

- Every integer between the 1 and \( 2^w - 2 \), inclusive, to encode normal values
- The reserved value 0 to encode \( \pm 0 \) and subnormal numbers

---

Copyright © 2003 by the Institute of Electrical and Electronics Engineers, Inc. This document is an unapproved draft of a proposed IEEE-SA Standard - USE AT YOUR OWN RISK. See statement on page 1.
The reserved value $2^w - 1$ to encode $\pm\infty$ and NaNs.

[Should comment on, or reference to, a discussion of NaN encoding issues such as]

- What does the sign bit of a NaN mean?
- What does the significand of a NaN mean?

These issues are largely elided from sections 6.2 and 6.3. -JDD]

The value $v$ is inferred from the constituent fields thus:

1. If $E = 2^w - 1$ and $T \neq 0$, then $v$ is NaN regardless of $S$. If $d_1$, the most significant bit of $T$, is 0, then the NaN is signaling; otherwise the NaN is quiet.

2. If $E = 2^w - 1$ and $T = 0$, then $v = (-1)^S \infty$

3. If $1 \leq E \leq 2^w - 2$, then $v = (-1)^S 2^{E-bias} (1 + 2^{1-p} T)$; thus normal numbers have an implicit leading significand bit of 1.

4. If $E = 0$ and $T \neq 0$, then $v = (-1)^S 2^{emin} (0 + 2^{1-p} T)$; thus subnormal numbers have an implicit leading significand bit of 0.

5. If $E = 0$ and $T = 0$, then $v = (-1)^S 0$ (signed zero, see section 6.3)

<table>
<thead>
<tr>
<th>Summary of Basic Binary Format Parameters: Encodings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format Name</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>Storage width</td>
</tr>
<tr>
<td>Trailing significand field width $t$</td>
</tr>
<tr>
<td>Exponent field width $w$</td>
</tr>
<tr>
<td>Exponent bias</td>
</tr>
</tbody>
</table>

Table 1b: Binary Encodings
3.3. Basic Decimal Format Encodings

Unlike basic binary floating-point formats, a representable nonzero numerical value may have multiple representations in a basic decimal format; the set of all the distinct representations of the name numerical value form that value’s cohort. For example, in terms of the parameters in section 3.1, the values

\{ e = 0, 10^{1-p} m = 100 \},
\{ e = 1, 10^{1-p} m = 10 \}, and
\{ e = 2, 10^{1-p} m = 1 \}

are three distinct and distinguishable representations from the cohort of one hundred. Consequently, unlike basic binary formats, basic decimal formats do not require values to have a normalized representation. While numerically equal, different members of a cohort can be distinguished by the decimal-specific operations discussed in section 5.11. The cohorts of different values have different numbers of elements. If a nonzero value has \( v \) decimal digits from its most significand nonzero digit to its least significand nonzero digit, the value’s cohort will have at most \((p-v)+1\) elements where \( p \) is the number of digits of precision in the format. For example, a 1-digit value can have up to \( p \) different representations while a \( p \)-digit value only has one representation. (A \( v \) digit value may have fewer than \((p-v)+1\) elements in its cohort if the value is near the extremes of the format’s exponent range.) For decimal arithmetic, besides specifying a numerical result, the arithmetic operands also specify which member of the cohort to return; see section 5.12 for a discussion of this determination. Traditional decimal applications make use of the additional exponent encoding information cohorts allow.

[The cohort of zero is a special case that may need further consideration. Arguably, NaN and infinity have cohorts too. -JDD]

[Note: Unlike 854, we are explicitly requiring different encodings of the same numerical value to be observably different; from 854 section 3.1 “The standard allows an implementation to encode some values redundantly provided that it does not distinguish redundant encodings of nonzero values.” -JDD]

Numbers in the decimal formats are encoded in the following four fields as shown in Figure 1-10:

1. 1-bit sign \( S \)
2. 5-bit combination field $G$ encoding classification, two leading exponent bits whose value together is 0, 1, or 2, and one leading significand digit

3. $w$-bit following exponent field which, when combined with the two leading exponent bits from the combination field, provides a $w+2$-bit biased exponent $E = e + \text{bias}$

4. $t$-bit trailing significand field $T = j_1 \ldots j_J$. There are $J = t/10$ groups $j_i$; each these groups of ten bits is a declet encoding three decimal digits. When the declets are combined with the leading significand digit from the combination field, the format has a total of $p = 1 + 3 \times J$ decimal digits.

The format's values $v$ are encoded thus according to the value of the five-bit $G$ field:

1. If $G$ is 11111, then $v$ is a NaN regardless of $S$. The values of $E$ and $T$ distinguish various NaNs. If $E_0$, the most significant bit of $E$, is 1, then the NaN is signaling; otherwise the NaN is quiet. [The sense of signaling NaN distinction is reversed from binary; but this allows the all-1 bit pattern to be a decimal]
signaling NaN. However, the all-1 bit pattern will not be propagated; see section 6.2]

2. If $G$ is 11110, then $v = (-1)^S \infty$. The values of $E$ and $T$ are ignored. However, the canonical infinity representation has $E = 0$, $T = 0$; this is the infinity encoding returned as a result by arithmetic operations. [Last sentence based on suggestion of MFC in email correspondence after May meeting -JDD]

3. For finite numerical values, $v = (-1)^S 10^{E-bias} 10^{1-p} m$; the decimal digits $d_0 d_1 ... d_{p-1}$ of the significand $m$ are encoded in the combination and trailing significand fields, while the biased exponent $e$ is encoded in the combination and following exponent fields:

- When the combination field $G$ is 110xx or 1110x, the leading significand digit $d_0$ is $8+G_4$, a value 8 or 9, and the leading exponent bits are $2G_2+G_3$, a value 0, 1, or 2.

- When the combination field $G$ is 0xxxx or 10xxx, the leading significand digit $d_0$ is $4G_2+2G_3+G_4$, a value in the range 0..7, and the leading exponent bits are $2G_0+G_1$, a value 0, 1, or 2. And consequently if $T$ is 0 and $G$ is 00000, 01000, or 10000, then $v = (-1)^S 0$.

[So there are emax + -emin representations of zero? -JDD]

Decoding Densely Packed Decimal

Table 1c decodes a declet, with 10 bits \( b(0) \) to \( b(9) \), into 3 decimal digits \( d(1) \), \( d(2) \), \( d(3) \). The first column is in binary and an “x” denotes “don’t care”. Thus all 1024 possible 10-bit patterns shall be accepted and mapped into 1000 possible 3-digit combinations with some redundancy.

<table>
<thead>
<tr>
<th>( b(6) ), ( b(7) ), ( b(8) ), ( b(3) ), ( b(4) )</th>
<th>( d(1) )</th>
<th>( d(2) )</th>
<th>( d(3) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 x x x x</td>
<td>( 4b(0) + 2b(1) + b(2) )</td>
<td>( 4b(3) + 2b(4) + b(5) )</td>
<td>( 4b(7) + 2b(8) + b(9) )</td>
</tr>
<tr>
<td>1 0 0 x x</td>
<td>( 4b(0) + 2b(1) + b(2) )</td>
<td>( 4b(3) + 2b(4) + b(5) )</td>
<td>( 8 + b(9) )</td>
</tr>
<tr>
<td>1 0 1 x x</td>
<td>( 4b(0) + 2b(1) + b(2) )</td>
<td>( 8 + b(5) )</td>
<td>( 4b(3) + 2b(4) + b(9) )</td>
</tr>
<tr>
<td>1 1 0 x x</td>
<td>( 8 + b(2) )</td>
<td>( 4b(3) + 2b(4) + b(5) )</td>
<td>( 4b(0) + 2b(1) + b(9) )</td>
</tr>
<tr>
<td>1 1 0 0 0</td>
<td>( 8 + b(2) )</td>
<td>( 8 + b(5) )</td>
<td>( 4b(0) + 2b(1) + b(9) )</td>
</tr>
<tr>
<td>1 1 0 0 1</td>
<td>( 8 + b(2) )</td>
<td>( 4b(0) + 2b(1) + b(5) )</td>
<td>( 8 + b(9) )</td>
</tr>
<tr>
<td>1 1 1 1 0</td>
<td>( 4b(0) + 2b(1) + b(2) )</td>
<td>( 8 + b(5) )</td>
<td>( 8 + b(9) )</td>
</tr>
<tr>
<td>1 1 1 1 1</td>
<td>( 8 + b(2) )</td>
<td>( 8 + b(5) )</td>
<td>( 8 + b(9) )</td>
</tr>
</tbody>
</table>

Table 1c: Decoding 10-bit Densely Packed Decimal to 3 Decimal Digits
Encoding Densely Packed Decimal

Table 1d encodes 3 decimal digits \(d(1), d(2),\) and \(d(3),\) each having 4 bits which can be expressed by a second subscript \(d(1,0:3), d(2,0:3),\) and \(d(3,0:3),\) where bit 0 is the most significant and bit 3 the least significant, into a declet, with 10 bits \(b(0)\) to \(b(9).\) Arithmetic operations generate only the 1000 10-bit patterns defined by table 1d.

<table>
<thead>
<tr>
<th>(d(1,0), d(2,0), d(3,0))</th>
<th>(b(0),b(1),b(2))</th>
<th>(b(3),b(4),b(5))</th>
<th>(b(6))</th>
<th>(b(7),b(8),b(9))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>(d(1,1:3))</td>
<td>(d(2,1:3))</td>
<td>0</td>
<td>(d(3,1:3))</td>
</tr>
<tr>
<td>0 0 1</td>
<td>(d(1,1:3))</td>
<td>(d(2,1:3))</td>
<td>1</td>
<td>0, 0, (d(3,3))</td>
</tr>
<tr>
<td>0 1 0</td>
<td>(d(1,1:3))</td>
<td>(d(3,1:2),d(2,3))</td>
<td>1</td>
<td>0, 1, (d(3,3))</td>
</tr>
<tr>
<td>0 1 1</td>
<td>(d(1,1:3))</td>
<td>1, 0, (d(2,3))</td>
<td>1</td>
<td>1, 1, (d(3,3))</td>
</tr>
<tr>
<td>1 0 0</td>
<td>(d(3,1:2),d(1,3))</td>
<td>(d(2,1:3))</td>
<td>1</td>
<td>1, 0, (d(3,3))</td>
</tr>
<tr>
<td>1 0 1</td>
<td>(d(2,1:2),d(1,3))</td>
<td>0, 1, (d(2,3))</td>
<td>1</td>
<td>1, 1, (d(3,3))</td>
</tr>
<tr>
<td>1 1 0</td>
<td>(d(3,1:2),d(1,3))</td>
<td>0, 0, (d(2,3))</td>
<td>1</td>
<td>1, 1, (d(3,3))</td>
</tr>
<tr>
<td>1 1 1</td>
<td>0, 0, (d(1,3))</td>
<td>1, 1, (d(2,3))</td>
<td>1</td>
<td>1, 1, (d(3,3))</td>
</tr>
</tbody>
</table>

Table 1d: Encoding 3 Decimal Digits to 10-bit Densely Packed Decimal
The 24 patterns of the form $01x11x111x$, $10x11x111x$, or $11x11x111x$ (where an "x" denotes "don't care") are not generated in the result of an arithmetic operation. However, as listed in table 1c, these 24 bit patterns do map to valid numerical values.

The bit pattern in a NaN significand can affect how the NaN is propagated; see section 6.2 for details.

<table>
<thead>
<tr>
<th>Format Name</th>
<th>decimal32</th>
<th>decimal64</th>
<th>decimal128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage width</td>
<td>32</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>Trailing significand field</td>
<td>20</td>
<td>50</td>
<td>110</td>
</tr>
<tr>
<td>width $t$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exponent field width $w$</td>
<td>6</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>Combination field width</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Exponent bias</td>
<td>95</td>
<td>383</td>
<td>6143</td>
</tr>
</tbody>
</table>

Table 1e: Decimal Encodings
3.4. Extended Formats

An implementation supporting binary basic formats but not binary128 should also support the binaryx extended format. An implementation supporting decimal basic formats but not decimal128 should also support the decimalx extended format. Table 1f specifies the minimum precision and exponent range of extended formats, which depend on the size of the largest supported basic format. The minimum exponent range is that of the next larger basic format, while the minimum precision is intermediate between the largest supported basic format and the next larger basic format.

The encoding of extended formats is implementation-defined, and thus their size in storage. Extended formats must represent all the values defined in that implementation’s basic formats — signed zeros, signed infinities, quiet and signaling NaNs, and numbers — as well as numbers of wider precision and exponent range according to Table 1f.

[I think if extended formats are specified, they should be specified more fully. For example, the original 754 specification as well as the current text does not mandate \( E_{\text{min}} = -E_{\text{max}} + 1 \); therefore, a perverse but legal extended format could have a greatly unbalanced exponent range. Section 3.1 of 854 recommends that “\( b^{(e_{\text{max}} + e_{\text{min}} + 1)} \) is the smallest integral power of \( b \) that is \( \geq 4 \),” which implies \( e_{\text{max}} = -e_{\text{min}} + 1 \) for binary and \( e_{\text{max}} = -e_{\text{min}} \) for decimal. –JDD]

<table>
<thead>
<tr>
<th>Summary of Extended Format Parameters: Representable Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Largest basic binary format</td>
</tr>
<tr>
<td>Largest basic decimal format</td>
</tr>
<tr>
<td>binary32, binary64, decimal32, decimal64</td>
</tr>
<tr>
<td>binaryx parameter</td>
</tr>
<tr>
<td>( p ) digits ( \geq )</td>
</tr>
<tr>
<td>( E_{\text{max}} ) ( \geq )</td>
</tr>
<tr>
<td>( E_{\text{min}} ) ( \leq )</td>
</tr>
</tbody>
</table>

Table 1f: Extended Formats
This standard provides declarations for evaluating expressions partly or completely in precision higher than that of the operands. For instance, with appropriate compile-time declarations, an expression involving binary64 format operands may be evaluated in binary64 format, a binaryx format, or a binary128 format.

[I think specifying extended formats is unnecessary. It suffices to provide an expression evaluation mode that allows operations with smaller relative or absolute (in the case of underflow) or inverse absolute (in the case of overflow) error bound than that of operations in the precision of the operands. That also encompasses fused multiply-add.]

[Leaving above draft paragraph red pending expression evaluation discussion. Prof. Kahan notes that rounding policy information is necessary for algorithmic reuse. -JDD]
3. (Previous draft) Formats

This standard defines four floating-point formats in two groups, basic and extended, each having two widths, single and double. The standard levels of implementation are distinguished by the combinations of formats supported.

3.1. Sets of Values

This section concerns only the numerical values representable within a format, not the encodings. The only values representable in a chosen format are those specified by way of the following three integer parameters:

- \( p \) = the number of significant bits (precision)
- \( E_{\text{max}} \) = the maximum exponent
- \( E_{\text{min}} \) = the minimum exponent.

Each format’s parameters are given in Table 1. Within each format only the following entities shall be provided:

- Numbers of the form \((-1)^{s} 2^{E} (b_0, b_1, b_2, ..., b_{p-1})\), where
  - \( s \) = 0 or 1
  - \( E \) = any integer between \( E_{\text{min}} \) and \( E_{\text{max}} \), inclusive
  - \( b_i \) = 0 or 1

- Two infinities, +\( \infty \) and –\( \infty \)

- At least one signaling NaN

- At least one quiet NaN

The foregoing description enumerates some values redundantly, for example, \( 2^0 \cdot (1.0) = 2^1 \cdot (0.1) = 2^2 \cdot (0.01) = ... \). However, the encodings of such nonzero values may be redundant only in extended formats (3.3). The nonzero values of the form \( \pm 2^E (0, b_1, b_2, ..., b_{p-1}) \) are called denormalized. Reserved exponents may be used to encode NaNs, +\( \infty \), –\( \infty \), ±0, and denormalized numbers. For any variable that has the value zero, the sign bit \( s \) provides an extra bit of information. Although all formats have distinct
representations for +0 and −0, the signs are significant in some circumstances, such as division by zero, and not in others. In this standard, 0 and −∞ are written without a sign when the sign is not important.

### 3.2. Basic Formats

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Single</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>24</td>
</tr>
<tr>
<td>(E_{\text{max}})</td>
<td>+127</td>
</tr>
<tr>
<td>(E_{\text{min}})</td>
<td>-126</td>
</tr>
<tr>
<td>Exponent bias</td>
<td>+127</td>
</tr>
<tr>
<td>Exponent width in bits</td>
<td>8</td>
</tr>
<tr>
<td>Format width in bits</td>
<td>32</td>
</tr>
</tbody>
</table>

Numbers in the single and double formats are composed of the following three fields:

4. 1-bit sign \(s\)

5. Biased exponent \(e = E + \text{bias}\)

6. Fraction \(f = .b_1b_2...b_p\)

The range of the unbiased exponent \(E\) shall include every integer between two values \(E_{\text{min}}\) and \(E_{\text{max}}\), inclusive, and also two other reserved values \(E_{\text{min}}-1\) to encode ±0 and denormalized numbers, and \(E_{\text{max}}+1\) to encode ±\(\infty\) and NaNs. The foregoing parameters are given in Table 1. Each nonzero numerical value has just one encoding. The fields are interpreted as follows:

### 3.2.1. Single
A 32-bit single format number \( X \) is divided as shown in Fig 1. The value \( v \) of \( X \) is inferred from its constituent fields thus

6. If \( e = 255 \) and \( f \neq 0 \), then \( v \) is NaN regardless of \( s \)

7. If \( e = 255 \) and \( f = 0 \), then \( v = (-1)^{s} \cdot \infty \)

8. If \( 0 < e < 255 \), then \( v = (-1)^{s} \cdot 2^{e-127} (1.f) \)

9. If \( e = 0 \) and \( f \neq 0 \), then \( v = (-1)^{s} \cdot 2^{-126} (0.f) \) (denormalized numbers)

10. If \( e = 0 \) and \( f = 0 \), then \( v = (-1)^{s} \cdot 0 \) (zero)

**Figure 1. Single Format**

<table>
<thead>
<tr>
<th>s</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>msb</td>
<td>lsb</td>
<td>msb</td>
</tr>
</tbody>
</table>

3.2.2. Double

A 64-bit double format number \( X \) is divided as shown in Fig 2. The value \( v \) of \( X \) is inferred from its constituent fields thus

1. If \( e = 2047 \) and \( f \neq 0 \), then \( v \) is NaN regardless of \( s \)

2. If \( e = 2047 \) and \( f = 0 \), then \( v = (-1)^{s} \cdot \infty \)

3. If \( 0 < e < 2047 \), then \( v = (-1)^{s} \cdot 2^{e-1023} (1.f) \)

4. If \( e = 0 \) and \( f \neq 0 \), then \( v = (-1)^{s} \cdot 2^{-1022} (0.f) \) (denormalized numbers)

5. If \( e = 0 \) and \( f = 0 \), then \( v = (-1)^{s} \cdot 0 \) (zero)

**Figure 2. Double Format**

<table>
<thead>
<tr>
<th>s</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>msb</td>
<td>lsb</td>
<td>msb</td>
</tr>
</tbody>
</table>

Copyright © 2003 by the Institute of Electrical and Electronics Engineers, Inc. This document is an unapproved draft of a proposed IEEE-SA Standard - USE AT YOUR OWN RISK. See statement on page 1.
3.3. Extended Formats

The single extended and double extended formats encode in an implementation-dependent way the sets of values in 3.1 subject to the constraints of Table 1. This standard allows an implementation to encode some values redundantly, provided that redundancy be transparent to the user in the following sense: an implementation either shall encode every nonzero value uniquely or it shall not distinguish redundant encodings of nonzero values. An implementation may also reserve some bit strings for purposes beyond the scope of this standard. When such a reserved bit string occurs as an operand the result is not specified by this standard.

An implementation of this standard is not required to provide (and the user should not assume) that single extended have greater range than double.

3.4. Combinations of Formats

All implementations conforming to this standard shall support the single format. Implementations should support the extended format corresponding to the widest basic format supported, and need not support any other extended format. [FOOTNOTE 3: Only if upward compatibility and speed are important issues should a system supporting the double extended format also support single extended.]
4. Rounding

Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit in the destination's format while signaling the inexact exception (7.5). Except for conversion between internal floating-point formats and external decimal formats, every operation specified in Section 5 shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result according to one of the modes in this section.

The rounding modes affect all arithmetic operations except comparison and remainder. The rounding modes may affect the signs of zero sums (6.3), and do affect the thresholds beyond which overflow (7.3) and underflow (7.4) may be signaled.

4.1. Round to Nearest

An implementation of this standard shall provide round to nearest as the default rounding mode. In this mode the representable value nearest to the infinitely precise result shall be delivered; if the two nearest representable values are equally near, the one with its least significant bit zero shall be delivered. However, an infinitely precise result with magnitude at least
\[ b^{e_{\text{max}}} \left( b - 0.5 b^{1-p} \right) 2^{e_{\text{max}}/(2-2^p)} \]
shall round to \( \infty \) with no change in sign; here \( e_{\text{max}} \) and \( p \) are determined by the destination format (see Section 3) unless overridden by a rounding precision mode (4.3).

[Additional decimal-only mode: round to nearest with ties rounding away from zero? -JDD]

4.2. Directed Roundings

An implementation shall also provide three other user-selectable directed rounding modes: the directed rounding modes round toward \( +\infty \), round toward \( -\infty \), and round toward 0.

When rounding toward \( +\infty \) the result shall be the format's value (possibly \( +\infty \)) closest to and no less than the infinitely precise result. When rounding toward \( -\infty \) the result shall be the format's value (possibly \( -\infty \)) closest to and no greater than the infinitely precise result. When rounding toward 0 the
result shall be the format's value closest to and no greater in magnitude than the infinitely precise result.

[In addition to the IEEE 754 rounding modes, the Java BigDecimal classes provides a number of other rounding modes:

• round to nearest with ties rounding away from zero
• round to nearest with ties rounding toward zero
• round away from zero
• round unnecessary (throw exception for inexact)

-JDD]

4.3. Rounding Precision

Normally, a result is rounded to the precision of its destination. However, some systems deliver arithmetic results only to destinations wider than their operands. On such a system the user, which may be a high-level language compiler, shall be able to specify that a result be rounded instead to any supported narrower precision with only one rounding, though it may be stored in a wider format with its wider exponent range. $Q$

Normally, a result is rounded to the precision of its destination. However, some systems deliver results only to double or extended destinations. On such a system the user, which may be a high-level language compiler, shall be able to specify that a result be rounded instead to single precision, though it may be stored in the double or extended format with its wider exponent range. $Q$

[FOOTNOTE 4: Control of rounding precision is intended to allow systems whose destinations are always 64-bit or wider to mimic, in the absence of over/underflow, the precisions of systems with only narrower destinations. But an implementation should not provide operations that combine wider operands to produce a narrower result, with only one rounding.] $Q$

[FOOTNOTE 4: Control of rounding precision is intended to allow systems whose destinations are always double or extended to mimic, in the absence of over/underflow, the precisions of systems with single and double destinations. An implementation should not provide operations that combine double or extended operands to produce a single result, nor operations that combine double extended operands to produce a double result, with only one rounding.] $Q$
However, while it is permissible to provide only rounding to narrower precision with wider exponent range, on such a system it should also be possible to round to the precision and exponent range of a narrower format, even if that rounded value is stored in a wider format. §6

[FOOTNOTE: If it is only possible to round to narrower precision with a wider exponent range, it can be unnecessarily complicated to exactly emulate the behavior of systems that only support narrower precisions single or double. In particular, emulating the overflow and underflow behavior can be costly.] §6

Note that to meet the specifications in 4.1, the result cannot suffer more than one rounding error. §6

[Should this rounding precision section be outlawed for decimal? -JDD]
5. Operations

All conforming implementations of this standard shall provide operations to add, subtract, multiply, divide, compute fused multiply-add §FMA, extract the square root, find the remainder, round to integer in floating-point format, convert between different floating-point formats, convert between floating-point and integer formats, convert between internal floating-point formats and external decimal formats. Whether copying without change of format is considered an operation is an implementation option §BI. Except for conversion between internal floating-point formats and external decimal formats, each of the operations shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then coerced this intermediate result to fit in the destination's format (see Sections 4 and 7). Section 6 augments the following specifications to cover ±0, ±∞, and NaN; Section 7 enumerates exceptions caused by exceptional operands and exceptional results.

In this standard, some operators such as fused multiply-add fma(a,b,c) and predicates such as isnormal(x) are written as named generic functions due to lack of widely-accepted operator notations; in a specific programming environment they might be represented by operators, or by families of format-specific functions, or by generic functions whose names may differ from those in this standard. §FMA

[To mesh better with the claim in Section 4 that “rounding modes affect all arithmetic operations,” arithmetic operations should be more clearly defined as a set. From reading section 4, one could conclude that comparison is an arithmetic operation. Section 4 of 854 states “The rounding modes affect all arithmetic operations except comparison and remainder.” -JDD]

5.1. Arithmetic

For each supported format §FMA, an implementation shall provide the add, subtract, multiply, divide, and remainder operations for any two operands of the same format, and a fused multiply-add operation for any three operands of the same format §FMA; for each supported format, it should also provide these operations for operands of differing formats. The destination format (regardless of the rounding precision control of 4.3) shall be at least as wide
as a widest operand's format. All results shall be rounded as specified in Section 4.

When \( y \neq 0 \), the remainder \( r := x \ REM \ y \) is defined regardless of the rounding mode by the mathematical relation \( r := x - y \times n \), where \( n \) is the integer nearest the exact value \( x/y \); whenever \( |n - x/y| = \frac{1}{2} \), then \( n \) is even. Thus, the remainder is always exact. If \( r = 0 \), its sign shall be that of \( x \). Precision control (4.3) shall not apply to the remainder operation.

### 5.2. Square Root

The square root operation shall be provided in all supported formats. The result is defined and has a positive sign for all operands \( \geq 0 \), except that \( \sqrt{-0} \) shall be \(-0\). The destination format shall be at least as wide as the operand's. The result shall be rounded as specified in Section 4.

### 5.3. Floating-Point Format Conversions

It shall be possible to convert floating-point numbers between all supported formats. If the conversion is to a narrower precision, the result shall be rounded as specified in Section 4. Conversion to a format with the same radix but wider precision and range is always exact.

[The use of “wider” and “narrower” in the above paragraph should include some mention of exponent range. For example, you can have have an overflow converting a single extended value to the double format since double might have a smaller exponent range. An example of this sort of problem which can occur in practice is rounding an x87 register value rounded to single precision (but extended exponent range) to double. -JDD]

### 5.4. Conversion Between Floating-Point and Integer Formats

It shall be possible to convert between all supported floating-point formats and all supported integer formats. Conversion to integer shall be effected by rounding as specified in Section 4. Conversions between floating-point integers and integer formats shall be exact unless an exception arises as specified in 7.1.

### 5.5. Round Floating-Point Number to Integer Value
It shall be possible to round a floating-point number to an integral valued floating-point number in the same format. The rounding shall be as specified in Section 4, with the understanding that when rounding to nearest, if the difference between the unrounded operand and the rounded result is exactly one half, the rounded result is even. This operation never generates an inexact exception (Section 7).

§AINTNX

5.6. Conversion between internal floating-point formats and external decimal formats

Binary <-> Decimal Conversion

[I am not going to update this section for decimal until we consider correctly-rounded base conversion.] Conversion between decimal strings in at least one format and binary floating-point numbers in all supported basic formats shall be provided for numbers throughout the ranges specified in Table 2. The integers $M$ and $N$ in Tables 2 and 3 are such that the decimal strings have values $\pm M \times 10^N$. On input, trailing zeros shall be appended to or stripped from $M$ (up to the limits specified in Table 2) so as to minimize $N$. When the destination is a decimal string, its least significant digit should be located by format specifications for purposes of rounding.

When the integer $M$ lies outside the range specified in Tables 2 and 3, that is, when $M \geq 10^9$ for single or $10^{17}$ for double, the implementor may, at his option, alter all significant digits after the ninth for single and seventeenth for double to other decimal digits, typically 0.

<table>
<thead>
<tr>
<th>Format</th>
<th>Decimal to Binary</th>
<th>Binary to Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max $M$</td>
<td>Max $N$</td>
</tr>
<tr>
<td>Single</td>
<td>$10^9 – 1$</td>
<td>99</td>
</tr>
<tr>
<td>Double</td>
<td>$10^{17} – 1$</td>
<td>999</td>
</tr>
</tbody>
</table>

Conversions shall be correctly rounded as specified in Section 4 for operands lying within the ranges specified in Table 3. Otherwise, for rounding to nearest, the error in the converted result shall not exceed by more than 0.47 units in the destination's least significant digit the error that is incurred by the rounding specifications of Section 4, provided that exponent over/underflow does not occur. In the directed rounding modes the
error shall have the correct sign and shall not exceed 1.47 units in the last place.

Conversions shall be monotonic, that is, increasing the value of a binary floating-point number shall not decrease its value when converted to a decimal string; and increasing the value of a decimal string shall not decrease its value when converted to a binary floating-point number.

When rounding to nearest, conversion from binary to decimal and back to binary shall be the identity as long as the decimal string is carried to the maximum precision specified in Table 2, namely, 9 digits for single and 17 digits for double. [FOOTNOTE 5: The properties specified for conversions are implied by error bounds that depend on the format (single or double) and the number of decimal digits involved; the 0.47 mentioned is a worst-case bound only. For a detailed discussion of these error bounds and economical conversion algorithms that exploit the extended format, see COONEN, JEROME T. Contributions to a Proposed Standard for Binary Floating-Point Arithmetic. Ph.D. Thesis, University of California, Berkeley, CA, 1984.]

If decimal to binary conversion over/underflows, the response is as specified in Section 7. Over/underflow, NaNs, and infinities encountered during binary to decimal conversion should be indicated to the user by appropriate strings. NaNs encoded in decimal strings are not specified in this standard.

To avoid inconsistencies, the procedures used for binary <-> decimal conversion should give the same results regardless of whether the conversion is performed during language translation (interpretation, compilation, or assembly) or during program execution (run-time and interactive input/output).

<table>
<thead>
<tr>
<th>Format</th>
<th>Decimal to Binary</th>
<th>Binary to Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max M</td>
<td>Max N</td>
</tr>
<tr>
<td>Single</td>
<td>10^9 – 1</td>
<td>13</td>
</tr>
<tr>
<td>Double</td>
<td>10^{17} – 1</td>
<td>27</td>
</tr>
</tbody>
</table>

[This section needs to be updated to take decimal arithmetic into account; i.e. defining an external representation that preserves both the numerical value and the representation exponent. -JDD]
5.7. Comparison

For every supported format, it shall be possible to compare two floating-point numbers in that format all supported formats, even if the operands' formats differ. Additionally, floating-point numbers represented in different formats shall be comparable as long as the operands' formats have the same radix. Comparisons are exact and never overflow nor underflow. Four mutually exclusive relations are possible: less than, equal, greater than, and unordered. The last case arises when at least one operand is NaN. Every NaN shall compare unordered with everything, including itself. Comparisons shall ignore the sign of zero (so +0 = –0).

The result of a comparison shall be delivered in one of two ways at the implementor's option: either as a condition code identifying one of the four relations listed above, or as a true-false response to a predicate that names the specific comparison desired. In addition to the true-false response, an invalid operation exception (7.1) shall be signaled when, as indicated in Table 4, last column, unordered operands are compared using one of the predicates involving < or > but not ? (Here the symbol ? signifies unordered).

Tables 4abc exhibit twenty the twenty-six functionally distinct useful predicates and negations with various ad-hoc and traditional names and symbols named, in the first column, using three notations: ad hoc, FORTRAN-like, and mathematical. Each predicate is true if any of the its indicated condition codes is true. It shows how they are obtained from the four condition codes and tells which predicates Table 4b lists the predicates that cause an invalid operation exception when the relation is unordered. The entries T and F indicate whether the predicate is true or false when the respective relation holds. That invalid exception defends against unexpected quiet NaNs arising in programs written using the six standard predicates = ≠ < ≤ ≥ > without considering the possibility of a quiet NaN argument. Newer programs that explicitly take account of the possibility of quiet NaN arguments may use the quiet predicates in Table 4c that do not signal such an invalid exception. §PRED-TABLE

Note that predicates come in pairs, each a logical negation of the other; applying a prefix such as NOT to negate a predicate in Tables 4abc reverses the true/false sense of its associated entries, but does not change whether unordered relations cause an invalid operation exception. §PRED-TABLE
Implementations that provide predicates shall provide the first six predicates in Table 4 and should provide the seventh, and a means of logically negating predicates. §PRED-TABLE

These quiet predicates do not signal an exception on quiet NaN operands and shall be available to all programs: §PRED-TABLE

<table>
<thead>
<tr>
<th>Quiet predicate</th>
<th>Quiet negation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>True relations</strong></td>
<td><strong>Names</strong></td>
</tr>
<tr>
<td>eq</td>
<td>equal</td>
</tr>
<tr>
<td></td>
<td>=</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
These signaling predicates signal an invalid exception on quiet NaN operands and shall be available to all programs not written to take into account the possibility of NaN operands [e.g. programs written in C or Fortran prior to IEEE 754].

Table 4b: Required signaling predicates and negations

<table>
<thead>
<tr>
<th>Signaling predicate</th>
<th>Signaling negation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>True relations</strong></td>
<td><strong>Names</strong></td>
</tr>
<tr>
<td>gt</td>
<td>greater</td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
</tr>
<tr>
<td>gt eq</td>
<td>greater-equal</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
</tr>
<tr>
<td>lt</td>
<td>less</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
</tr>
<tr>
<td>lt eq</td>
<td>less-equal</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
These quiet predicates do not signal an exception on quiet NaN operands and shall be available to all programs written to take into account the possibility of NaN operands [e.g., programs written in languages invented after IEEE 754].

<table>
<thead>
<tr>
<th>Table 4c: Required quiet predicates and negations §PRED-TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Quiet predicate</strong></td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td>gt</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>gt eq</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>lt</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>lt eq</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>un</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
FOOTNOTE 6: There are may appear to be two ways to write the logical negation of a predicate, one using NOT explicitly and the other reversing the relational operator. For example, Thus in programs written without considering the possibility of a NaN argument, the logical negation of the signaling predicate (X < Y) is just the signaling predicate NOT(X < Y); the quiet reversed predicate (X ?>= Y) is different in that it does not signal an invalid operation exception when X and Y are unordered. In contrast, For example, the logical negation of (X = Y) may be written either NOT(X = Y) or (X ?<> Y); in this case both expressions are functionally equivalent to (X != Y). However, this coincidence does not occur for the other predicates.

§PRED-TABLE

[For those keeping track, the three tables list 20 predicates. The remaining 12 are:
quiet false
quiet true
quiet <>
quiet ?=
signaling false
signaling true
signaling <>
signaling ?=
signaling =
signaling !=
signaling ?
signaling <=>
]
<table>
<thead>
<tr>
<th>Predicates</th>
<th>Relations</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ad-hoc</td>
<td>Mat</td>
<td>GreaterThan</td>
</tr>
<tr>
<td>=</td>
<td>=</td>
<td>F</td>
</tr>
<tr>
<td>?&lt;&gt;</td>
<td>&lt;&gt;</td>
<td>T</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td>T</td>
</tr>
<tr>
<td>&gt;=</td>
<td>&gt;=</td>
<td>T</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td>F</td>
</tr>
<tr>
<td>&lt;=</td>
<td>&lt;=</td>
<td>F</td>
</tr>
<tr>
<td>?</td>
<td>unordered</td>
<td>-</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>&lt;&gt;</td>
<td>T</td>
</tr>
<tr>
<td>&lt;=&lt;&gt;</td>
<td>&lt;=&lt;&gt;</td>
<td>T</td>
</tr>
<tr>
<td>??</td>
<td>??</td>
<td>T</td>
</tr>
<tr>
<td>??</td>
<td>??</td>
<td>T</td>
</tr>
<tr>
<td>??</td>
<td>??</td>
<td>T</td>
</tr>
<tr>
<td>??</td>
<td>??</td>
<td>T</td>
</tr>
<tr>
<td>NOT(&gt;)</td>
<td>-</td>
<td>F</td>
</tr>
<tr>
<td>NOT(&gt;=)</td>
<td>-</td>
<td>F</td>
</tr>
<tr>
<td>NOT(&lt;)</td>
<td>-</td>
<td>T</td>
</tr>
<tr>
<td>NOT(&lt;=)</td>
<td>-</td>
<td>T</td>
</tr>
<tr>
<td>NOT(?)</td>
<td>-</td>
<td>T</td>
</tr>
<tr>
<td>NOT(&lt;&gt;?)</td>
<td>-</td>
<td>F</td>
</tr>
<tr>
<td>NOT(?&gt;)</td>
<td>-</td>
<td>F</td>
</tr>
</tbody>
</table>

Copyright © 2003 by the Institute of Electrical and Electronics Engineers, Inc. This document is an unapproved draft of a proposed IEEE-SA Standard - USE AT YOUR OWN RISK. See statement on page 1.

Page 43
5.8. Quiet Operations §BI

Implementations shall provide the following operations for all supported formats. They are performed as if on strings of bits, treating numbers and NaNs alike, and hence signal no exception.

- **copy (y := x)** copies a floating-point operand \( x \) to a destination \( y \) in the same format, with no change.
- **negate (y := –x)** copies a floating-point operand \( x \) to a destination \( y \) in the same format, reversing the sign. (This is not the same as \( 0 – x \)).
- **abs (y := abs(x) or y := |x|)** copies a floating-point operand \( x \) to a destination \( y \) in the same format, changing the sign to positive.
- The function \( z := \text{copysign}(x, y) \) copies a floating-point operand \( x \) to a destination \( z \) in the same format as \( x \), but with the sign of \( y \).
- **getexponent(\( x \))** returns the exponent the format uses to represent \( x \). [For binary basic formats, getexponent(infinity) is emax+1, getexponent(0) and getexponent(subnormal) are emin-1. For decimal formats, getexponent could be used to differentiate between different members of a cohort. However, return values need to be defined for decimal infinities and NaNs since the combination fields is not part of the exponent. -JDD]

5.9. Quiet Predicates §CLASSPRED

Implementations shall provide classification predicates which deliver a true-false response and signal no exception.

- **issigned(\( x \))** iff \( x \) has negative sign, and applies equally to zeros and NaNs.
- **isnormal(\( x \))** iff \( x \) is normal (not zero, subnormal, infinity, or NaN).
isfinite(x) iff x is zero, subnormal or normal (not infinity or NaN).

iszero(x) iff x = ±0.

issubnormal(x) iff x is subnormal.

isinf(x) iff x is infinity.

isnan(x) iff x is a NaN.

issignaling(x) iff x is an optional signaling NaN.
5.10. Min and Max §MINMAX

An implementation shall provide operations to find the minimum and maximum of two floating-point values. These operations signal no exception for quiet NaN operands: if both operands are quiet NaNs, a NaN is returned as specified in section 6.2; if only one operand is quiet NaN, these operations return the numerical operand. When both operands are numbers:

- \( \min(x, y) \) returns the minimum of its operands. When its operands are \(-0\) and \(+0\), \( \min(x, y) \) returns \(-0\).

- \( \max(x, y) \) returns the maximum of its operands. When its operands are \(-0\) and \(+0\), \( \max(x, y) \) returns \(+0\).

- \( \minmag(x, y) \) returns the operand of minimum magnitude. When its operands are equal in magnitude but different in sign, \( \minmag(x, y) \) returns the operand with negative sign.

- \( \maxmag(x, y) \) returns the operand of maximum magnitude. When its operands are equal in magnitude but different in sign, \( \maxmag(x, y) \) returns the operand with positive sign.

[RATIONALE for min and max:

\( \max(x, y) \) is defined as that value for which the following is true for every \( z \):

\[ z \leq \max(x, y) \iff z \leq x \text{ OR } z \leq y. \]

As a consequence of this definition, \( \max(5, \text{NaN}) \) is 5.

Another possible definition considered and rejected:

\[ z \geq \max(x, y) \iff z \geq x \text{ AND } z \geq y. \]

This definition implies \( \max(5, \text{NaN}) \) is NaN and \( \max(\text{inf}, \text{NaN}) \) is NaN.

A third possibility:

\[ z \max(x, y) \text{ iff } z > x \text{ AND } z > y. \]

This definition implies \( \max(5, \text{NaN}) \) is NaN and \( \max(\text{inf}, \text{NaN}) \) is inf.
Note that C99 does not define the sign of \( \max(+0, -0) \). Note that Java min/max have defined NaN semantics different from the above. Why prefer numbers over NaNs?

- Many algorithms pre-process matrices by scaling rows or columns by their largest element. Reducing with a max operator that prefers NaNs will deliver a scale factor of NaN and subsequently wipe out an entire row or column. Large matrix computations are typically done in place, so this will destroy potentially useful information.

- In a statistical application, NaN entries may denote missing data. The fact that some entries are missing may not matter, but scanning through a large data set and removing them up front imposes a significant performance penalty. It is much faster to apply a max reduction and check the invalid flag if it matters.

Predicates such as ismax(x,y), duplicating the logic of max(x, y), returning true when max(x,y) would select x and false otherwise, were rejected as adding insufficient additional capability over normal comparisons for sorting.

### 5.11 Decimal Specific Operations

Decimal implementations of the standard shall provide operations to query and set the exponent of a decimal value.

- `checkQuantum(x, y)` for decimal arguments x and y, returns true if the representation exponent of x and y is the same and false otherwise. `checkQuantum(NaN, NaN)` and `checkQuantum(\infty, \infty)` return true.
- `requantize(x, i)` for decimal argument x and integer argument i, return a representation of the numerical value of x with exponent i; if there is no such representation, signal invalid and return a NaN.

[This is analagous to the rescale operation in Java’s BigDecimal. The name for this functionality may be

Copyright © 2003 by the Institute of Electrical and Electronics Engineers, Inc. This document is an unapproved draft of a proposed IEEE-SA Standard - USE AT YOUR OWN RISK. See statement on page 1.
changed/improved. There was discussion of having requantize’s second argument be a decimal number instead of an integer; in that case, requantize would try to return a value numerically equal to \( x \) but having the second argument’s exponent.

For decimal, we may want to define a getExponent method non-trivially distinct from logb. In other words, getExponent (for binary and decimal) would return the representation exponent while logb would return the “normalized” exponent. -JDD]

### 5.12 Decimal Exponent Calculation

As discussed in section 3.3, decimal arithmetic involves not only computing the proper numerical result but also selecting the proper member of that value’s cohort. If the result is a finite value, the required element of the cohort is selected based on the preferred exponent for a result of operation. The preferred exponent of an operation is a function of the exponents of the inputs, as listed in table 5.

<table>
<thead>
<tr>
<th>Operation with decimal result</th>
<th>Preferred Exponent of Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>( \min(\text{getexponent}(\text{addend}), \text{getexponent}(\text{augend})) )</td>
</tr>
<tr>
<td>subtraction</td>
<td>( \min(\text{getexponent}(\text{minuend}), \text{getexponent}(\text{subtrahend})) )</td>
</tr>
<tr>
<td>multiplication</td>
<td>( \text{getexponent}(\text{multiplier}) + \text{getexponent}(\text{multiplicand}) )</td>
</tr>
<tr>
<td>division</td>
<td>( \text{getexponent}(\text{dividend}) – \text{getexponent}(\text{divisor}) )</td>
</tr>
<tr>
<td>fused multiply add, ((a \times b) + c )</td>
<td>( \min(\text{getexponent}(a) + \text{getexponent}(b), \text{getexponent}(c)) )</td>
</tr>
<tr>
<td>square root</td>
<td>( \text{floor}(\text{getexponent}(\text{radicand}) / 2) )</td>
</tr>
<tr>
<td>conversion between decimal formats</td>
<td>( \text{getexponent}(\text{operand}) (? )</td>
</tr>
<tr>
<td>conversion from a binary format to a decimal format</td>
<td>(? )</td>
</tr>
<tr>
<td>round to integer value in same format</td>
<td>( \text{getexponent}(\text{operand}) (? )</td>
</tr>
<tr>
<td>copy, negate, abs</td>
<td>( \text{getexponent}(\text{operand}) (? )</td>
</tr>
</tbody>
</table>
Table 5: Preferred Exponents of Decimal Arithmetic Operations

<table>
<thead>
<tr>
<th>Operation with decimal result</th>
<th>Preferred Exponent of Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>copysign(x, y)</td>
<td>x.exponent (?)</td>
</tr>
<tr>
<td>min, max, minmag, maxmag</td>
<td>Exponent of lesser or greater operand: if operands are numerically equal, return either value (?) [From MFC, using subtraction as a guide, return smaller exponent if the results are numerically equal? -JDD]</td>
</tr>
</tbody>
</table>

[Table entries marked with a question mark do not, at the time of this writing, have clear known precedents from prior decimal work. -JDD]

If a member of the result’s cohort does not include an element with the preferred exponent, the element with the least exponent larger than the preferred exponent is used.

[This description should be double-checked for accuracy. -JDD]

[Some examples would be very helpful in this section; e.g., divide with lots more result digits (1/32), multiply or add with a carry out, etc. -JDD]

[The effect of different representations of zero needs to be clarified; each exponent value has a zero. Therefore, adding a zero (with the right exponent) to a nonzero value could cause another member of that value’s cohort to be returned. Is this a bug or feature? The cohorts of nonfinite values need further discussion and definition if they are to be integrated into table 5. -JDD]
6. Infinity, NaNs, and Signed Zero

6.1. Infinity Arithmetic

Infinity arithmetic shall be construed as the limiting case of real arithmetic with operands of arbitrarily large magnitude, when such a limit exists. Infinities shall be interpreted in the affine sense, that is, \(-\infty < (\text{every finite number}) < +\infty\).

Arithmetic on \(\infty\) is always exact and therefore shall signal no exceptions, except for the invalid operations specified for \(\infty\) in 7.1. [And the case of remainder(subnormal, infinity) with underflow alternate exception handling enabled.] The exceptions that do pertain to \(\infty\) are signaled only when

1. \(\infty\) is created from finite operands by overflow (7.3) or division by zero (7.2), with corresponding trap disable $\text{TRAP}$

2. \(\infty\) is an invalid operand (7.1).

[MFC recommends decimal arithmetic operations always return the canonical infinity encoding for an infinite result. Therefore, unlike NaNs, there is no support for (or intended support for) trying to propagate diagnostic information in decimal infinities. - JDD]

6.2. Operations with NaNs

Two different kinds of NaN, signaling and quiet, shall be supported in all operations. Signaling NaNs afford values for uninitialized variables and arithmetic-like enhancements (such as complex-affine infinities or extremely wide range) that are not the subject of the standard. Quiet NaNs should, by means left to the implementor's discretion, afford retrospective diagnostic information inherited from invalid or unavailable data and results. Propagation of the diagnostic information requires that information contained in the NaNs be preserved through arithmetic operations and floating-point format conversions.

[How is this information preserved?]
How do NaN sign bits propagate?

What are the bit patterns of newly created NaNs? -JDD

For every operation listed in sections 5.1-5.7 and 5.10, signaling NaNs shall be reserved operands that signal the invalid operation exception (7.1). Signaling NaNs shall be reserved operands that signal the invalid operation exception (7.1) for every operation listed in Section 5. Whether copying a signaling NaN without a change of format signals the invalid operation exception is the implementor's option. Under default exception handling, any operation signaling an invalid exception for which a floating-point result is to be delivered, shall deliver a quiet NaN.

Every operation involving one or more §FMA two input NaNs, none of them signaling, shall signal no exception but, if a floating-point result is to be delivered, shall deliver as its result a quiet NaN, which should be one of the input NaNs. If the bits of the significand field of a decimal input NaN are regarded as declets and all those declets have bit patterns corresponding to the 1000 bit patterns generated by arithmetic operations (Table 1d), then the bit pattern of the NaN significand shall be preserved if that NaN is chosen as the result NaN. [Propagating only “canonical” NaN significands decided May 30, 2003. Note that the “all 1’s” significand is not one of the canonical bit patterns guaranteed to be preserved. -JDD]

Note that format conversions might be unable to deliver the same NaN. Quiet NaNs do have effects similar to signaling NaNs §SNAN on operations that do not deliver a floating-point result; these operations, namely comparison and conversion to a format that has no NaNs, are discussed in 5.4, 5.6, 5.7, and 7.1.

6.3. The Sign Bit

The sign of a NaN is not determined by this standard and this standard does not interpret the sign of a NaN. [Augment 754 wording with 854 wording. -JDD] Therefore, the following rules only apply when neither the inputs nor result are NaN. Otherwise, The sign of a product or quotient is the exclusive or of the operands' signs; the sign of a sum, or of a difference x−y regarded as a sum x+(−y), differs from at most one of the addends' signs, and the sign of the result of the round floating-point number to integral value operation is the sign of the operand. These rules shall apply even when operands or results are zero or infinite.
[This section points to the more general issue of specifying what bit pattern newly created NaNs should have. 854 also discusses conversion to external decimal formats and other conversions in its corresponding section. -JDD]

When the sum of two operands with opposite signs (or the difference of two operands with like signs) is exactly zero, the sign of that sum (or difference) shall be + in all rounding modes except round toward $-\infty$; in which mode, the sign of an exact zero sum (or difference) shall be -. However, $x+x = x-(-x)$ retains the same sign as $x$ even when $x$ is zero.

When $(a\times b)+c$ would vanish in exact arithmetic, the sign of $\text{fma}(a,b,c)$ shall be determined by the rules above for a sum of operands. §FMA

Except that $\sqrt{-0}$ shall be $-0$, every valid square root shall have a positive sign.
7. Default Exception Handling Exceptions

There are five types of exceptions that shall be signaled when detected. This section specifies default nonstop exception handling, which entails raising a status flag, delivering a default result, and continuing execution. Section 8 specifies alternate exception handling methods that a user may select. The signal entails setting a status flag, taking a trap, or possibly doing both. With each exception should be associated 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. In some cases, the result is different if a trap is enabled. §TRAP [Appendix 8 specifies minimum system capabilities to implement alternate exception handling efficiently via asynchronous traps.]

For each type of exception the implementation shall provide a status flag that shall be raised set when on any occurrence of the corresponding exception is signaled. When no corresponding trap occurs, §TRAP It shall be lowered reset only at the user's request. The user shall be able to test and to alter the status flags individually, and should further be able to save and restore all five at one time.

The only exceptions that can coincide on the same operation are inexact with overflow and inexact with underflow.

7.1. Invalid Operation

The invalid operation exception is signaled if an operand is invalid for the operation to be performed. The default result, when the exception occurs without a trap, §TRAP shall be a quiet NaN (6.2) provided the destination has a floating-point format. The invalid operations are:

1. Any operation on a signaling NaN (6.2);
2. Multiplication; 0 × ∞ or ∞ × 0 §FMA;
3. Fused multiply-add: fma(0, ∞, c) or fma(∞, 0, c) unless c is a quiet NaN §FMA;
4. Addition or subtraction or fused multiply-add §FMA; magnitude subtraction of infinities, such as (+∞) + (−∞);
5. Division: $0/0$ or $\infty/\infty$;

6. Remainder: $x$ REM $y$, where $y$ is zero or $x$ is infinite and neither is NaN;

7. Square root if the operand is less than zero;

8. Conversion of a binary floating-point number to an integer or external decimal format when overflow, infinity, or NaN precludes a faithful representation in that format and this cannot otherwise be signaled;

9. Comparison by way of predicates involving $<$ or $>$, without $?$, when the operands are unordered (5.7, Table 4).

### 7.2. Division by Zero

If the divisor is zero and the dividend is a finite nonzero number, then the division by zero exception shall be signaled. The default result, when no trap occurs, §TRAP shall be a correctly signed $\infty$ (6.3).

### 7.3. Overflow

The overflow exception shall be signaled whenever the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result (Section 4) were the exponent range unbounded. [Providing formulae for the overflow/underflow thresholds based on rounding mode might be helpful here; the underflow threshold is complicated by trapping/nontrapping status and the selection of tininess and loss of accuracy policies. - JDD] The default result, when no trap occurs, §TRAP shall be determined by the rounding mode and the sign of the intermediate result as follows:

1. Round to nearest carries all overflows to $\infty$ with the sign of the intermediate result

2. Round toward 0 carries all overflows to the format's largest finite number with the sign of the intermediate result

3. Round toward $-\infty$ carries positive overflows to the format's largest finite number, and carries negative overflows to $-\infty$

4. Round toward $+\infty$ carries negative overflows to the format's most negative finite number, and carries positive overflows to $+\infty$
7.4. Underflow

Two correlated events contribute to underflow. One is the creation of a tiny nonzero result between $\pm b^{emin}$ which, because it is so tiny, may cause some other exception later such as overflow upon division. The other is extraordinary loss of accuracy during the approximation of such tiny numbers by subnormal denormalized $\$2$ numbers. The implementor may choose how these events are detected, but shall detect these events in the same way for all operations. Tininess may be detected either

1. After rounding - when a nonzero result computed as though the exponent range were unbounded would lie strictly between $\pm b^{emin}$

2. Before rounding - when a nonzero result computed as though both the exponent range and the precision were unbounded would lie strictly between $\pm b^{emin}$.

Loss of accuracy may be detected as either

1. A denormalization [subnormal??] loss - when the delivered result differs from what would have been computed were exponent range unbounded

2. An inexact result - when the delivered result differs from what would have been computed were both exponent range and precision unbounded. (This is the condition called inexact in 7.5).

By default, when an underflow trap is not implemented, or is not enabled (the default case), $\$TRAP$ underflow shall be signaled (by way of the underflow flag) only when both tininess and loss of accuracy have been detected. The method for detecting tininess and loss of accuracy does not affect the delivered result which might be zero, subnormal denormalized $\$2$, or $\pm b^{emin}$.

7.5. Inexact

If the rounded result of an operation is not exact or if it overflows with default handling without an overflow trap $\$TRAP$ then the inexact exception shall be signaled. The rounded or overflowed result shall be delivered to the destination or, if an inexact trap occurs, to the trap handler $\$TRAP$. 
8. Alternate Exception Handling

To be supplied!

8.4. Precedence

If an operation signals both overflow and inexact exceptions, or both underflow and inexact exceptions, and alternate exception handling is specified for the overflow or for the underflow, then that alternate exception handling takes precedence over any alternate exception handling specified for inexact.
Appendix 1

Recommended Functions and Predicates

(This Appendix is not a part of ANSI/IEEE Std 754–1985, IEEE Standard for Binary Floating-Point Arithmetic.)

The following functions and predicates are recommended as aids to program portability across different systems, perhaps performing arithmetic very differently. They are described generically, that is, the types of the operands and results are inherent in the operands. Languages that require explicit typing will have corresponding families of functions and predicates.

Some functions, such as the copy operation $y := x$ without change of format, may at the implementor’s option be treated as nonarithmetic operations which do not signal the invalid operation exception for signaling NaNs; the functions in question are (1), (2), (6), and (7).§BI

1. Copysign($x$, $y$) returns $x$ with the sign of $y$. Hence, $\text{abs}(x) = \text{copysign}(x, 1.0)$, even if $x$ is NaN.§BI

2. $-x$ is $x$ copied with its sign reversed, not $0 - x$; the distinction is germane when $x$ is ±0 or NaN. Consequently, it is a mistake to use the sign bit to distinguish signaling NaNs from quiet NaNs. §BI

3. Scalb($y$, $N$) returns $y \times b^N$ for integral values $N$ without computing $b^N$. If $y$ is a decimal value, scalb acts as if the $10^N$ were stored with an exponent of $N$. [In other words, in the absence of rounding you can just add or subtract $N$ to the exponent for decimal too. -JDD]

4. Logb($x$) returns the unbiased exponent of $x$, a signed integer in the format of $x$, except that logb (NaN) is a NaN, logb (∞) is $+\infty$, and logb(0) is $-\infty$ and signals the division by zero exception. When $x$ is positive and finite the expression scalb($x$, $-\logb(x)$) lies strictly between 0 and $b$; it is less than 1 only when $x$ is subnormal denormalized. §2. [For decimal, I recommend logb return the logical “normalized” exponent. -JDD]

5. Radix($x$) returns the radix of the format of $x$, 2 or 10.

6. Nextafter($x$, $y$) returns the next representable neighbor of $x$ in the direction toward $y$, in the format of $x$. The following special cases...
arise: if \( x = y \), then the result is \( \text{copysign}(x, y) \) – which differs from \( x \) only if \( x \) and \( y \) are zeros of different sign; without any exception being signaled; if \( x = 0 \) and \( y \neq 0 \), nextafter returns the subnormal of least magnitude with the sign of \( y \).\(^\text{NextNX}\) Otherwise, if either \( x \) or \( y \) is \( \text{NaN} \), then the result is according to Section 6.2. a quiet \( \text{NaN} \), then the result is one or the other of the input \( \text{NaNs} \).\(^\text{NextNX}\)

7. \( \text{nextup}(x) \) returns the next more positive value in \( x \)'s format, with no exception for numeric results. \( \text{nextup} \) (negative subnormal of least magnitude) is \( -0 \). \( \text{nextup}(\pm 0) \) is the smallest positive nonzero subnormal. \( \text{nextup}(+\text{INFINITY}) \) is +\( \text{INFINITY} \), and \( \text{nextup}(–\text{INFINITY}) \) is the finite negative number largest in magnitude. When \( x \) is \( \text{NaN} \), \( \text{nextup}(x) \) is a quiet \( \text{NaN} \), with an invalid exception if \( x \) is signaling. \( \text{nextdown}(x) \) returns \( –\text{nextup}(–x) \).\(^\text{NU}\)

8. \( \text{Finite}(x) \) returns the value TRUE if \( -\infty < x < +\infty \), and returns FALSE otherwise.\(^\text{ClassPred}\)

9. \( \text{Isnans}(x) \), or equivalently \( x \neq x \), returns the value TRUE if \( x \) is a \( \text{NaN} \), and returns FALSE otherwise.\(^\text{ClassPred}\)

10. \( x \preceq y \) is TRUE only when \( x \leq y \) or \( x = y \), and is distinct from \( x \neq y \), which means \( \text{NOT}(x = y) \) (Table 4).\(^\text{Pred}\)

11. \( \text{Unordered}(x, y) \), or \( x ? y \), returns the value TRUE if \( x \) is unordered with \( y \), and returns FALSE otherwise (Table 4).\(^\text{Pred}\)

12. \( \text{stringtofloating}(x) \) converts an external decimal string \( x \) as though at run time (rather than compile time) to a floating-point value in the widest format supported by the implementation, paying due heed to exceptions and the current rounding direction and precision as specified in 5.6.\(^\text{Conv}\)

13. \( \text{Class}(x) \) tells which of the following ten classes \( x \) falls into: signaling \( \text{NaN} \), quiet \( \text{NaN} \), \( -\infty \), negative normal nonzero, negative subnormal \( \text{denormalized} \)\(^\text{\#2}\), \( -0 \), \(+0 \), positive subnormal \( \text{denormalized} \)\(^\text{\#2}\), positive normal nonzero, \( +\infty \). This function is never exceptional, not even for signaling \( \text{NaNs} \).

\( Copyright \) \( \odot \) \( 2003 \) by the Institute of Electrical and Electronics Engineers, Inc. This document is an unapproved draft of a proposed IEEE-SA Standard - USE AT YOUR OWN RISK. See statement on page 1.
14. equivalent(x,y) Two floating-point values x and y are equivalent if they are both NaNs, both infinities with the same sign, or both finite values with the same settings for all of the format's representation fields. Therefore, -0.0 and +0.0 are not equivalent since the sign field differs. Numerically equal decimal values with different exponents in their representations are not equivalent either.
Appendix 8 Traps §TRAP

A trap is a change in control flow that occurs as a result of an exception (not all exceptions trap). Such traps can be used to efficiently implement alternate exception handling (section 8). This appendix enumerates the minimum requirements for such traps. §TRAP

A user should be able to request a trap on any of the five exceptions by specifying a handler for it. The user should be able to request that an existing handler be disabled, saved, or restored. The user should also be able to determine whether a specific trap handler for a designated exception has been enabled. When an exception whose trap is disabled is signaled, it shall be handled in the manner specified in Section 7. When an exception whose trap is enabled is signaled the execution of the program in which the exception occurred shall be suspended, the trap handler previously specified by the user shall be activated, and a result, if specified in Section 7, shall be delivered to it.

8.1. Trap Handler

A trap handler should have the capabilities of a subroutine that can return a value to be used in lieu of the exceptional operation's result; this result is undefined unless delivered by the trap handler. Similarly, the status flag(s) corresponding to the exceptions being signaled with their associated traps enabled may be undefined unless raised or lowered set or reset by the trap handler.

When a system traps, the trap handler should be able to determine

1. Which exception(s) occurred on this operation

2. The kind of operation that was being performed

3. The destination's format

4. In overflow, underflow, and inexact exceptions, the correctly rounded result, including information that might not fit in the destination's format
5. In invalid operation and divide by zero exceptions, the operand values.

8.2. Trapped Overflow

Trapped overflows on all operations except conversions shall deliver to the trap handler the result obtained by dividing the infinitely precise result by \( b^\alpha \) and then rounding. The bias-adjust \( \alpha \) is determined by the format's exponent bias from Tables 1b and 1e: \((3/2)(\text{bias}+1)\) - thus 192 in the binary32 format. [Adding table 1e for decimal values. The decimal biases satisfy the recommendation in 854 section 7.3 that the exponent adjust is divisible by 12. -JDD] 1536 in the double, 24576 in the quad \( \$Q \), and \( 3 \times 2^n \) in the extended format, when \( n \) is the number of bits in the exponent field. [FOOTNOTE 7: This bias-adjust is chosen to translate over/underflowed values as nearly as possible to the middle of the exponent range so that, if desired, they can be used in subsequent scaled operations with less risk of causing further exceptions.] Trapped overflow on conversion from a binary floating-point format shall deliver to the trap handler a result in that or a wider format, possibly with the exponent bias adjusted, but rounded to the destination's precision. Trapped overflow on decimal to binary conversion shall deliver to the trap handler a result in the widest supported format, possibly with the exponent bias adjusted, but rounded to the destination's precision; when the result lies too far outside the range for the bias to be adjusted, a quiet \( \text{NaN} \) shall be delivered instead.

A trapped overflowed result \( Z \) of a conversion or scaling operation, intended for a binary floating-point destination, might not round to a normal number in the destination format, even after dividing by the bias-adjust \( b^\alpha \). But for some integral value \( n > \alpha \), \( Z / b^\alpha \) would round to a normal number \( z \) in the destination format, \( 1 \leq |z| < b \). In that case the result delivered to the trap handler shall be either the original operands or the pair \( z \) and \( n \), with \( z \) delivered in the destination format and \( n \) delivered in an integer or floating-point format. \( \$GROSS \)

[What about trapped overflow converting from binary to decimal? Should this even generate a floating-point overflow exception? Or is “appropriate strings” of section 5.6 enough?]
8.3. Trapped Underflow

When an underflow trap has been implemented and is enabled, underflow shall be signaled when tininess is detected regardless of loss of accuracy. Trapped underflows on all operations except conversion shall deliver to the trap handler the result obtained by multiplying the infinitely precise result by $b^\alpha$ and then rounding. The bias-adjust $\alpha$ is determined by the format's exponent bias from Table 1b: $(3/2)(\text{bias}+1)$ - thus 192 in the binary32 format. The bias-adjust $\alpha$ is 192 in the single, 1536 in the double, 24576 in the quad\footnote{FOOTNOTE 8: Note that a system whose underlying hardware always traps on underflow, producing a rounded, bias-adjusted result, shall indicate whether such a result is rounded up in magnitude in order that the correctly subnormal denormalized\footnote{GROSS} result may be produced in system software when the user underflow trap is disabled.} and $3 \times b^{n-2}$ in the extended format, where $n$ is the number of bits in the exponent field.

A trapped underflowed result $Z$ of a conversion or scaling operation, intended for a binary floating-point destination, might not round to a normal number in the destination format, even after multiplying by the bias-adjust $b^\alpha$. But for some integral value $n > \alpha$, $Z \times b^n$ would round to a normal number $z$ in the destination format, $1 \leq |z| < b$. In that case the result delivered to the trap handler shall be either the original operands or the pair $z$ and $n$, with $z$ delivered in the destination format and $n$ delivered in an integer or floating-point format.\footnote{GROSS}

[What about trapped underflow converting from binary to decimal? Should this even generate a floating-point underflow exception? Or is “appropriate strings” of section 5.6 enough?]

8.4. Precedence

If enabled, the overflow and underflow traps take precedence over a separate inexact trap.
Contents

SECTION
1. Scope
   1.1. Implementation Objectives
   1.2. Inclusions
   1.3. Exclusions

2. Definitions

3. Formats
   3.1. Sets of Representable Numerical Values
   3.2. Basic Binary Format Encodings
   3.3. Basic Decimal Format Encodings
   3.4. Extended Formats
   3.1. Sets of Values
   3.2. Basic Formats
   3.3. Extended Formats
   3.4. Combinations of Formats

4. Rounding
   4.1. Round to Nearest
   4.2. Directed Roundings
   4.3. Rounding Precision

5. Operations
   5.1. Arithmetic
   5.2. Square Root
   5.3. Floating-Point Format Conversions
   5.4. Conversion Between Floating-Point and Integer Formats
   5.5. Round Floating-Point Number to Integer Value
   5.6. Conversion between binary or decimal internal formats and external decimal formats
   5.7. Comparison
5.8 Quiet Operations §BI
5.9 Quiet Predicates §CLASSPRED
5.10. Min and Max §MINMAX
5.11. Decimal Specific Operations
5.12 Decimal Exponent Calculation

6. Infinity, NaNs, and Signed Zero
   6.1. Infinity Arithmetic
   6.2. Operations with NaNs
   6.3. The Sign Bit

7. Default Exception Handling
   7.1. Invalid Operation
   7.2. Division by Zero
   7.3. Overflow
   7.4. Underflow
   7.5. Inexact

8. Alternate Exception Handling

FIGURES
   Fig 1.

TABLES
   Table 1. Summary of Format Parameters
   Table 2. Decimal Conversion Ranges
   Table 3. Correctly Rounded Decimal Conversion Range
   Table 4. Predicates and Relations

APPENDIX
   1. Recommended Functions and Predicates

8. Traps §TRAP

CHANGE LOG