Editorial note: This draft document is intended to encompass all the technical content of the existing standard ANSI/IEEE Std 754–1985, along with *accepted* additions§C and ~~deletions~~§C and *proposed* additions§C and ~~deletions~~§C in distinctive fonts. The footnote §C refers to the change log/rationale/open issues list below. Open issues and rationale appear in magenta.

DRAFT IEEE Standard for ~~Binary~~ Floating-Point Arithmetic

IEEE Standards Activities Department
Standards Licensing and Contracts
445 Hoes Lane, P.O. Box 1331
Piscataway, NJ 08855-1331, USA

Note change approved 23 Jan 2003 but not yet incorporated in this draft: revisions to section 5.6 base conversion to accommodate decimal formats.

Change and Open Issues Log

23 January 2003

§DECIMAL – add dense packed decimal formats.
[Corollary changes: removed extended precision format specifications and restrictions on combinations of formats.]
§SNAN – remove signaling NaNs by making all NaNs quiet.
[We had previously decided to move the signaling NaNs into an optional appendix, but this was the worst possible choice, combining all the disadvantages of signaling NaNs with all the disadvantages of undefined bit patterns.
We considered three alternative resolutions:

1. Define the former signaling NaN bit patterns as "undefined" and thus remove signaling NaNs from the standard completely - implementors would be free to do anything at all with operands that are "undefined" - the undefined bit patterns would be WELL-defined, it's the operations upon them that are UN-defined.

2. Restore signaling NaNs to their previous place in the mandatory part of the standard (with however the slight incompatibilities previously agreed to, specifying that half the NaNs are quiet and half signaling and stating conclusively which are which).

3. Define the former signaling NaN bit patterns as quiet NaNs and thus remove signaling NaNs from the standard completely.

Option #3 was chosen by the committee. The primary argument is that the signaling NaNs are a burden for all implementors which however have hardly ever been effectively exploited for arithmetic extensions or debugging; those few cases were
necessarily machine-dependent and relied on the trapping facility which we had already voted out of 754R in favor of a higher-level specification of desirable alternative exception handling methods. The failure of signaling NaNs can be traced to 754's mistaken inclusion of them in the same exception

as the invalid arithmetic operations, so their traps were entwined;
this was aggravated by Intel's 8087 decision to add non-754 operating system exceptions to the invalid class; and later further aggravated by Microsoft's taking control of the invalid trap in some Windows implementations and not allowing user-level trap handling.

So then we had to face squarely the meta-issue:
whether 754R must be strictly upward-performance-compatible with existing hardware implementations. Because any or all of 754 or 754R can be implemented entirely in software or entirely in hardware or in any combination, strictly speaking upward incompatibility is impossible.
What I mean is that existing important hardware-based 754 implementations should be able to implement 754R with no performance loss - in most cases it suffices to consider whether existing IA32 systems could be so upgraded. (Thus adding new features like fmac and minmax to the standard has no performance effect, since previous implementations of similar functions were not standardized, but changing the way signaling NaNs work does have a performance effect). The probable performance effect of #3 on existing implementations is that at a hardware/operating system level, the invalid trap would have to be always enabled in order to catch signaling NaNs to prevent them from changing the user-visible invalid flag or causing a user-visible invalid trap - thus slowing down all untrapped invalid operations such as 0 * inf .

Despite some heated opposition, the majority view remained that old systems would claim conformance to 754 and new systems to 754R in any event, and therefore well-justified changes to 754R would be fair game for consideration even if they were not upward-performance-compatible. And furthermore, many of the most popular "IEEE 754" environments have failed to comply with one or more aspects of the 754 standard, so it seemed likely that marketing science would continue to act in the same vein. One would still like to encourage general-purpose arithmetic implementations to provide the possibility of extensions. To that end an appendix provides that general-purpose 754R environments "should" provide
a method of extending the representation to include values not representable in a particular format; one way to do that is by providing a non-standard

non-default mode bit
that causes one class of NaNs to become 754-signaling; another is to provide
a non-standard exception (and corresponding non-default trap) that causes
a trap on any NaN operand. Thus extensions are not available in the
standard default portable environment - which is effectively the current situation -
but are available for system-dependent
extensions and debugging by system-dependent means.]

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 $0 \times \infty +$ qnan does not signal invalid, because a NaN is an operand, but $0 \times \infty + \infty$ 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 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 floating-point arithmetic in computer programming environments: standard and extended functions with single, double, quad **§Q**, and extended precision, 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, 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 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. Binary <-> decimal conversions to and from extended formats

2. Definitions

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

binary floating-point number. A bit-string containing ~~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.

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

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

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

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.~~ Required ~~quiet~~ NaNs (6.2) propagate through almost every algebraic arithmetic operation without signaling exceptions. ~~Optional signaling NaNs (Appendix 6) signal the invalid operation exception (see 7.1) whenever they appear as operands of algebraic arithmetic operations.~~

[how about: Without signaling exceptions, NaNs (6.2) propagate through arithmetic operations that deliver floating-point results.]

radix. The base for the representation of binary or decimal floating-point numbers.

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. An integer component of an unencoded binary or decimal floating-point number containing its significant digits. ~~that consists of an explicit or implicit leading bit to the left of its implied~~ radix ~~binary point and a fraction field to the right.~~

status flag. A variable that may take two states, raised or lowered ~~set and clear.~~ A user ~~may~~ can 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 ~~contain~~ additional system-dependent information, possibly inaccessible to some users. The operations of this standard, when exceptional, can ~~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.]

canonical representation. In a decimal format, for numbers which have redundant representations, one representation is specified as canonical, to be used when the non-canonical representations would convey no additional information. [What about sign of zero and sign of NaN – applies to binary too. Is +0 canonical – all 0 bits? What about a canonical NaN – all 1 bits? Aside from that:] The binary formats contain only canonical representations.

normal number. In a particular format, a nonzero floating-point number that can be represented in that format with an implicit (binary) or explicit (decimal) nonzero leading significand digit.

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 can not 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 floating-point formats: single, double, and quad, in each of two radices: binary formats based on radix 2 and decimal formats based on radix 10. A programming environment conforms to this standard by supporting one or more of these six defined formats.[Note: Decimal subcommittee recommends that minimal implementations should support either single binary or double decimal, and further argue against the single/double/quad terminology for decimal. I think that argues in favor of eliminating 754's combination of formats discussion and referring to both the binary and decimal formats according to bit length or byte length rather than single/double/quad. An implementation providing only single decimal would not be useful for much, but neither would it harm the standard to allow it.]

[Given positive future action on proposals for static declarations in general and expression evaluation modes in particular, the following is all I think we need to say about extended format – and it also encompasses using fma without explicit invocation. If we require that implementations provide (static is OK) means to evaluate expressions in lower precision than register precision, we can get rid of the rounding precision modes implementation method as well.] 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 double format operands may be evaluated in double format, quad format, or other formats not specified by this standard, that represent a superset of the values representable in double format, by providing greater precision and exponent range. [If we require that implementations provide means (static declarations are good enough) to evaluate expressions in lower precision than register precision, we can get rid of the rounding precision modes implementation method requirement as well. Static expression evaluation declarations should be available for “strict” according

to operand type, “accurate” using the widest available precision, and “fast” using the fastest available precision >= the operand types; additionally precision could be determined on a per-operation basis or globally over an entire expression.]

[As an example why one might prefer to standardize less rather than more in this respect, observe that if 754 had better foreseen that double-extended would mostly be provided as an anonymous type for intermediate expressions, it might better have specified that the extended precision rounding always be to zero, with the sticky bit OR'ed into the lsb, so that a subsequent store to lower precision would always be correctly rounded according to that precision and the dynamic rounding mode. At the cost of inconsequentially higher error bounds in complicated expressions, all the confusion about double roundings to storage formats could have been avoided.]

3.1. Sets of Representable Values

This section specifies the numerical values representable within numerical formats, not their encodings. The only numerical values representable within a particular format are those specified by means of the following integer parameters:

- b = the radix 2 or 10
- p = the number of significant digits(precision)
- E_{max} = the maximum exponent
- E_{min} = the minimum exponent .

Each format's parameters are given in Table la (representable values) and Table 1b(encodings). Within each format only the following entities shall be provided:

- Numbers of the form $(-1)^s\ b^E\ b^{1-p}(\ d_0\ d_1\ d_2\ \ldots\ d_{p-1})$, where
  - s = 0 or 1
  - E = any integer between E_{min} and E_{max}, inclusive
  - d_i = a digit in the range 0...b-1
- Two infinities, +∞ and –∞
- NaNs

The foregoing *description* enumerates *values* redundantly, for example, $10^0\ (1) = 10^{-1}\ (10) = 10^{-2}\ (100) = \ldots$. The binary formats have no redundant (non-canonical) (noncanonical?) *representations* for nonzero numbers, while the decimal formats have redundant (non-canonical) *representations* for numbers and infinities.

The smallest normal magnitude is b^{Emin} and the largest is $b^{Emax}(b - b^{1-p})$. The nonzero values of the form $\pm\ b^{Emin}\ b^{1-p}\ (d_1\ d_2\ \ldots\ d_{p-1}\ )$ are called *subnormal* because their magnitudes lie between zero and the smallest normal magnitude.

Every representable number is an integral multiple of the smallest subnormal magnitude $b^{E\min} b^{1-p}$.

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, and not in others. In this standard, 0 and ∞ are written without a sign when the sign is not important.

Table 1a
Summary of Format Parameters: Representable Values

	Binary format b = 2			Decimal format b = 10		
Parameter	Single	Double	Quad	Single	Double	Quad
Storage size in bits	32	64	128	32	64	128
p digits	24	53	113	7	16	34
E_{max}	+127	1023	+16383	96	384	6144
E_{min}	–126	–1022	–16382	–95	–383	–6143

Table 1b
Summary of Format Parameters: Encodings

	Binary format b = 2			Decimal format b = 10		
Parameter widths in bits	Single	Double	Quad	Single	Double	Quad
Storage width	32	64	128	32	64	128
Following significand field width fs	23	52	112	20	50	110
Exponent field width w	8	11	15	6	8	12
Combination field width				5	5	5
Exponent bias	127	1023	16383	95	383	6143

3.2. Binary Format Encodings

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 $e = E + bias$
3. Following significand bits $f = d_1\, d_1 \ldots d_{p-1}$

Figure 1-2 Binary Floating-Point Format §UF

width	1 bit	*w* bits	*p*–1 bits
field	**sign *s***	**exponent *e***	**following significand *f***
most/least significant bit		most.....least	most.....least

The range of the unbiased exponent E shall include every integer between two values E_{min} and E_{max}, inclusive, and also two other reserved values $E_{min} - 1$ to encode ± 0 and subnormal numbers, and $E_{max} + 1$ to encode ±∞ and NaNs. The foregoing parameters are given in Tables 1a and 1b. Each nonzero numerical value has just one encoding. The value *v* is inferred from the constituent fields thus:

1. If $e = 2^w - 1$ and $f \neq 0$, then *v* is NaN regardless of *s*
2. If $e = 2^w - 1$ and $f = 0$, then $v = (-1)^s \infty$
3. If $0 < e < 2^w - 1$, then $v = (-1)^s 2^{e-bias} (1 + 2^{1-p} f)$; thus there is an implicit leading significand 1-bit
4. If $e = 0$ and $f \neq 0$, then $v = (-1)^s 2^{Emin} (0 + 2^{1-p} f)$ (subnormal numbers); thus there is an implicit leading significand 0-bit
5. If $e = 0$ and $f = 0$, then $v = (-1)^s 0$ (zero)

3.3. Decimal Format Encodings

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 of the combination field, provides a w+2-bit biased exponent $e = E + bias$
4. fs-bit following significand field $f = j_0\, j_1 \ldots j_{J-1}$. There are J = fs÷10 groups j_i, each of ten bits encoding three decimal digits, which, when combined with the leading significand digit from the combination field, provide a total of p = 1 + 3× J decimal digits.

Figure 1-10 Decimal Floating-Point Format

width	1 bit	5 bits	*w* bits	fs bits
field	**sign** ***s***	**combination** ***g***	**exponent** ***e***	**following significand** ***f***
most/least significant bit		most.....least	most...least	most.....least

The special values *v* are encoded thus according to the value of the five-bit g field:

1. If g is 11111, then *v* is NaN regardless of *s*. The values of e and f distinguish various NaNs.

2. If g is 11110, then $v = (-1)^s \infty$. The values of e and f are ignored. The canonical representation of infinity has zero e and f fields.

3. If g is 00000, 01000, or 10000, and f is 0, then $v = (-1)^s 0$. The canonical representation of zero has a zero g field. [Note that in decimal, the zero representations could have been omitted as a special value case as they fall out automatically from the nonzero definition following.]

For nonzero numerical values, $v = (-1)^s \ 10^{e-bias} \ 10^{1-p} \ (d_0 \ d_1 ... d_{p-1})$; the significand is encoded in the combination and following significand fields, while the exponent is encoded in the combination and following exponent fields:

1. 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 g(2:3), a value 0, 1, or 2.

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 g(0:1), a value 0, 1, or 2.

If a decimal number has redundant representations in a particular format, the canonical representation is the one with the smallest exponent.

[The possible uses of non-canonical or unnormalized representations are an aspect of the unnormalized decimal arithmetic proposal to be discussed further. Should unnormalized operands produce unnormalized results if the values are the same as if operands were the first normalize? Should subtractions of normalized operands produce unnormalized results? Should there be a standardized "integer" mode to allow unnormalization to be preserved, as well as a standardize "normalizing" mode? Which should be the default?].

The following significand fields f contain groups of ten bits, each encoding three decimal digits.

Decoding Densely Packed Decimal

Table 1c decodes a densely packed decimal group, 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 a dash "-" denotes "don't care". Thus all 1024 possible 10-bit patterns are mapped into 1000 possible 3-digit combinations with some redundancy.

b(6) ,b(7) , b(8), b(3),b(4)	*d(1)*	*d(2)*	*d(3)*
0 - - - -	4b(0) + 2b(1) + b(2)	4b(3) + 2b(4) +b(5)	4b(7) + 2b(8) +b(9)
1 0 0 - -	4b(0) + 2b(1) + b(2)	4b(3) + 2b(4) +b(5)	8 + b(9)
1 0 1 - -	4b(0) + 2b(1) + b(2)	8 + b(5)	4b(3) + 2b(4) +b(9)
1 1 0 - -	8 + b(2)	4b(3) + 2b(4) +b(5)	4b(0) + 2b(1) +b(9)
1 1 1 0 0	8 + b(2)	8 + b(5)	4b(0) + 2b(1) +b(9)
1 1 1 0 1	8 + b(2)	4b(0) + 2b(1) +b(5)	8 + b(9)
1 1 1 1 0	4b(0) + 2b(1) + b(2)	8 + b(5)	8 + b(9)
1 1 1 1 1	8 + b(2)	8 + b(5)	8 + b(9)

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 densely packed decimal group, with 10 bits **b(0)** to **b(9)**. Arithmetic operations generate only the 1000 10-bit patterns defined by this table.

d(1,0), d(2,0), d(3,0)	*b(0),b(1),b(2)*	*b(3),b(4),b(5)*	*b(6)*	*b(7),b(8),b(9)*
0 0 0	d(1,1:3)	d(2,1:3)	0	d(3,1:3)
0 0 1	d(1,1:3)	d(2,1:3)	1	0, 0, d(3,3)
0 1 0	d(1,1:3)	d(3,1:2),d(2,3)	1	0, 1, d(3,3)
0 1 1	d(1,1:3)	1, 0, d(2,3)	1	1, 1, d(3,3)
1 0 0	d(3,1:2),d(1,3)	d(2,1:3)	1	1, 0, d(3,3)
1 0 1	d(2,1:2),d(1,3)	0, 1, d(2,3)	1	1, 1, d(3,3)
1 1 0	d(3,1:2),d(1,3)	0, 0, d(2,3)	1	1, 1, d(3,3)
1 1 1	0, 0, d(1,3)	1, 1, d(2,3)	1	1, 1, d(3,3)

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

3. (Previous draft) Formats

This standard defines three basic floating-point formats: single, double, and quad, and two families of extended formats: single-extended and double-extended. ~~in two groups, basic and extended, each having two widths, single and double.~~ **§Q** Double is a member of the single-extended family and quad is a member of the double-extended family.**§Q** 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_{max} = the maximum exponent
- E_{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 $(-1)^s\ 2^E\ (b_0 . b_1\ b_2 \ldots\ b_{p-1})$, where
  - s = 0 or 1
  - E = any integer between E_{min} and E_{max}, inclusive
  - b_i = 0 or 1
- Two infinities, +∞ and –∞
- ~~At least one signaling NaN~~§SNAN
- At least one quiet NaN

The foregoing description enumerates some values redundantly, for example, $2^0\ (1.0) = 2^1\ (0.1) = 2^2\ (0.0\ 1) = \ldots$. However, the encodings of such nonzero values may be redundant only in extended formats (3.3). The nonzero values of the form $\pm\ 2^{E_{min}}\ (0 . b_1\ b_2 \ldots\ b_{p-1})$ are called subnormal ~~denormalized~~ **§2**. Reserved exponents may be used to encode NaNs, ±∞ , ±0, and subnormal

~~denormalized~~ **§2** 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 1
Summary of Format Parameters

	Format				
Parameter	**Single**	**Single Extended**	**Double**	**Double Extended**	**Quad §Q**
p	24	>= 32	53	>= 64	113
E_{max}	+127	>= +1023	1023	>= +16383	+16383
E_{min}	–126	<= –1022	–1022	<= –16382	–16382
Exponent bias	+127	unspecified	+1023	unspecified	+16383
Exponent width in bits	8	>= 11	11	>= 15	15
Format width in bits	32	>= 43	64	>= 79	128

Numbers in the single, double, and quad ~~and double~~ **§Q** formats are composed of the following three fields:

4. 1-bit sign *s*
5. Biased exponent $e = E + bias$
6. Fraction $f = . b_1 b_1 \ldots b_{p-1}$

The range of the unbiased exponent E shall include every integer between two values E_{min} and E_{max}, inclusive, and also two other reserved values $E_{min} - 1$ to encode ± 0 and subnormal ~~denormalized~~ **§2** numbers, and $E_{max} + 1$ to encode ±∞ 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 $f \mathrel{!}= 0$, then v is NaN regardless of s
7. If $e = 255$ and $f = 0$, then $v = (-1)^s \infty$
8. If $0 < e < 255$, then $v = (-1)^s 2^{e-127} (1 . f)$
9. If $e = 0$ and $f \mathrel{!}= 0$, then $v = (-1)^s 2^{-126} (0 . f)$ (subnormal ~~denormalized~~ §2 numbers)
10. If $e = 0$ and $f = 0$, then $v = (-1)^s 0$ (zero)

Figure 1. Single Format

```
msb means most significant bit
lsb means least significant bit

 1    8              23             ... widths
+-+-------+-----------------------+
|s|   e   |           f           |
+-+-------+-----------------------+
   msb lsb msb                 lsb  ... order
```

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 $f \mathrel{!}= 0$, then v is NaN regardless of s
2. If $e = 2047$ and $f = 0$, then $v = (-1)^s \infty$
3. If $0 < e < 2047$, then $v = (-1)^s 2^{e-1023} (1 . f)$
4. If $e = 0$ and $f \mathrel{!}= 0$, then $v = (-1)^s 2^{-1022} (0 . f)$ (subnormal ~~denormalized~~ §2 numbers)
5. If $e = 0$ and $f = 0$, then $v = (-1)^s 0$ (zero)

Figure 2. Double Format

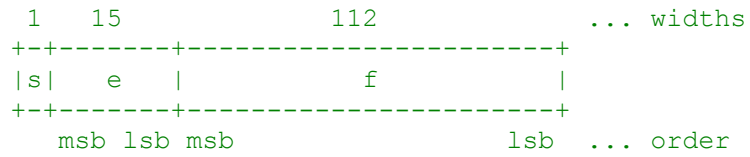
```
 1   11              52             ... widths
+-+-------+-----------------------+
```

3.2.3. Quad §Q

A 128-bit quad format number *X* is divided as shown in Fig. 3. The value *v* of *X* is inferred from its constituent fields thus

1. If *e* = 32767 and *f* != 0 , then *v* is NaN regardless of *s*
2. If *e* = 32767 and *f* = 0 , then $v = (-1)^s \infty$
3. If 0 < *e* < 32767 , then $v = (-1)^s 2^{e-16383} (1 . f)$
4. If *e* = 0 and *f* != 0 , then $v = (-1)^s 2^{-16382} (0 . f)$ (subnormal numbers)
5. If *e* = 0 and *f* = 0 , then *v* = (–1) ^s^ 0 (zero)

Figure 3. Quad Format §Q



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 supporting single as the widest basic format should also support single-extended; implementations supporting double as the widest basic format should also support double-extended. ~~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.]~~**§Q**

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 ~~binary <-> decimal~~ conversion between binary or decimal internal formats and external decimal formats, ~~(~~whose weaker conditions are specified in 5.6~~)~~, 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 $b^{E_{\max}}(b - b^{-p})$ shall round to ∞ with no change in sign; here $E_{\max}$ and p are determined by the destination format (see Section 3) unless overridden by a rounding precision mode (4.3).

4.2. Directed Roundings

An implementation shall also provide three user-selectable 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.

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 double or extended or quad 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 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**

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 binary <-> decimal, and~~ compare, and conversion between binary or decimal internal formats and external decimal formats . ~~Whether copying without change of format is considered an operation is an implementation option.§BI~~ Except for conversion between binary or decimal internal formats and external decimal formats ~~binary <-> decimal conversion~~, 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

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 ~~the wider~~ operand's format. All results shall be rounded as specified in Section 4.

When y != 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 $|n - x/y| = ½$, 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 >= 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 wider precision is exact.

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 binary or decimal internal 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 $\pm M \times 10^{\pm N}$. 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 $M >= 10^9$ 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 2
Decimal Conversion Ranges

Format	Decimal to Binary		Binary to Decimal	
	Max *M*	**Max *N***	**Max *M***	**Max *N***
Single	$10^9 - 1$	99	$10^9 - 1$	53
Double	$10^{17} - 1$	999	$10^{17} - 1$	340

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 3
Correctly Rounded Decimal Conversion Range

Format	Decimal to Binary		Binary to Decimal	
	Max M	**Max N**	**Max M**	**Max N**
Single	$10^9 - 1$	13	$10^9 - 1$	13
Double	$10^{17} - 1$	27	$10^{17} - 1$	27

5.7. Comparison

It shall be possible to compare floating-point numbers in all supported formats, even if the operands' formats differ. 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 inTable 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. ~~leaves the last column's entry unchanged.~~ §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 4a: Required quiet predicates and negations §PRED-TABLE

Quiet predicate		**Quiet negation**	
True relations	**Names**	**True relations**	**Names**
eq	equal =	lt gt un	not-equal ?<> NOT(=) ≠

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]:§PRED-TABLE

Table 4b: Required signaling predicates and negations §PRED-TABLE

Signaling predicate		**Signaling negation**	
True relations	**Names**	**True relations**	**Names**
gt	greater >	eq lt un	signaling-not-greater NOT(>)
gt eq	greater-equal >= ≥	lt un	signaling-less-unordered NOT(>=)
lt	less <	eq gt un	signaling-not-less NOT(<)
lt eq	less-equal <= ≤	gt un	signaling-greater-unordered NOT(<=)

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]:§PRED-TABLE

Table 4c: Required quiet predicates and negations §PRED-TABLE			
Quiet predicate		**Quiet negation**	
True relations	**Names**	**True relations**	**Names**
gt	quiet-greater !<= isgreater	eq lt un	quiet-not-greater ?<= NOT(!<=)
gt eq	quiet-greater-equal !< isgreaterequal	lt un	quiet-less-unordered ?< NOT(!<)
lt	quiet-less !>= isless	eq gt un	quiet-not-less ?>= NOT(!>=)
lt eq	quiet-less-equal !> islessequal	gt un	quiet-greater-unordered ?> NOT(!>)
un	unordered ? isunordered	lt eq gt	ordered <=> NOT(?)

~~[**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 4
Predicates and Relations

Predicates			Relations				Exception
Ad hoc	FORTRAN	Math	GreaterThan	LessThan	Equal	Unordered	InvalidIfUnordered
=	.EQ.	=	F	F	T	F	No
?<>	.NE.	!=	T	T	F	T	No
>	.GT.	>	T	F	F	F	Yes
>=	.GE.	>=	T	F	T	F	Yes
<	.LT.	<	F	T	F	F	Yes
<=	.LE.	<=	F	T	T	F	Yes
?	unordered	-	F	F	F	T	No
<>	.LG.	-	T	T	F	F	Yes
<=>	.LEG.	-	T	T	T	F	Yes
?>	.UG.	-	T	F	F	T	No
?>=	.UGE.	-	T	F	T	T	No
?<	.UL.	-	F	T	F	T	No
?<=	.ULE.	-	F	T	T	T	No
?=	.UE.	-	F	F	T	T	No
NOT(>)	-	-	F	T	T	T	Yes
NOT(>=)	-	-	F	T	F	T	Yes
NOT(<)	-	-	T	F	T	T	Yes
NOT(<=)	-	-	T	F	F	T	Yes
NOT(?)	-	-	T	T	T	F	No
NOT(<>)	-	-	F	F	T	T	Yes
NOT(<=>)	-	-	F	F	F	T	Yes
NOT(?>)	-	-	F	T	T	F	No

~~Table 4~~ ~~Predicates and Relations~~							
~~NOT(?>=)~~	-	-	~~F~~	~~T~~	~~F~~	~~F~~	~~No~~
~~NOT(?<)~~	-	-	~~T~~	~~F~~	~~T~~	~~F~~	~~No~~
~~NOT(?<=)~~	-	-	~~T~~	~~F~~	~~F~~	~~F~~	~~No~~
~~NOT(?=)~~	-	-	~~T~~	~~T~~	~~F~~	~~F~~	~~No~~

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 $z := copysign(x, y)$ copies a floating-point operand x to a destination z in the same format as x, but with the sign of y.
- The function $y := canonicalize(x)$ returns a canonical representation of x's value in y, in the same format as x. [Once again, what about zeros and NaNs?]

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 $x = \pm 0$.

- issubnormal(*x*) iff *x* is subnormal.
- isinf(*x*) iff *x* is infinity.
- isnan(*x*) iff *x* is a NaN.
- iscanonical(x) iff x is a canonical representation. [Once again, what about zeros and NaNs?]
- ~~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 are unexceptional. If both operands are NaN, see section 6.2. If only one operand is 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 <= max(x,y) iff z <=x OR z <=y.
As a consequence of this definition, max(5,NaN) is 5.

Another possible definition considered and rejected:
z >= max(x,y) iff z>=x AND z>=y.
This definition implies max(5,NaN) is NaN and max(inf,NaN) is NaN.

A third possibility:
z>max(x,y) iff z>x AND z>y.
This definition implies max(5,NaN) is NaN and max(inf,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.]

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, $-\infty$ < (every finite number) < $+\infty$.

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

1. ∞ is created from finite operands by overflow (7.3) or division by zero (7.2)~~, with corresponding trap disabled §TRAP~~

2. ∞ is an invalid operand (7.1).

6.2. Operations with NaNs

~~Quiet~~ 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.~~ **§SNAN** ~~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.

~~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.~~**§SNAN**

Every operation signaling ~~involving~~ an ~~signaling NaN or~~ **§SNAN** invalid operation (7.1), for which ~~shall, if no trap occurs and if~~ a floating-point result is to be delivered, shall by default deliver a ~~quiet~~ NaN as its result.

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. Note that format conversions might be unable to deliver the same NaN. ~~Quiet~~ NaNs signal exceptions ~~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

This standard does not interpret the sign of a 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.

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 that sign shall be –. However, $x+x = x–(–x)$ retains the same sign as x even when x is zero.

When (a×b)+c would vanish in exact arithmetic, the sign of 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 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 an optional signaling NaN (6.2);~~

2. Multiplication: $0 \times \infty$ or $\infty \times 0$ §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 $(+\infty) + (-\infty)$;

5. Division: 0/0 or ∞ / ∞;

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 or decimal floating-point number to an integer or 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 ∞ (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. 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 ∞ 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 –∞ carries positive overflows to the format's largest finite number, and carries negative overflows to –∞

4. Round toward +∞ carries negative overflows to the format's most negative finite number, and carries positive overflows to +∞

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, abs(x) = 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.

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 +∞ , and logb(0) is –∞ 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**.

5. 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 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!=0, nextafter returns the subnormal of least magnitude with the sign of y.**§NEXTNX** Otherwise, if either x or y is NaN, then the result is according to Section 6.2. ~~a quiet NaN, then the result is one or the other of the input NaNs.~~ **§NEXTNX**

Overflow is signaled when x is finite but nextafter(x, y) is infinite; underflow is signaled when nextafter(x, y) lies strictly between ± b^{Emin}; in both cases, inexact is signaled.

6. nextup(x) returns the next more positive value in x's format, with no exception for numeric results. nextup(negative subnormal of least magnitude) is –0. nextup(±0) is the smallest positive nonzero subnormal. nextup(+INFINITY) is +INFINITY, and nextup(–INFINITY) is the finite negative number largest in magnitude. When x is NaN, nextup(x) is a ~~quiet~~ NaN~~, with an invalid exception if x is signaling~~. nextdown(x) returns –nextup(–x). **§NU**

~~7. Finite(x) returns the value TRUE if $-\infty < x < +\infty$, and returns FALSE otherwise. §CLASSPRED~~

~~8. Isnan(x), or equivalently x != x, returns the value TRUE if x is a NaN, and returns FALSE otherwise.§CLASSPRED~~

~~9. $x <> y$ is TRUE only when $x < y$ or $x > y$, and is distinct from x != y, which means NOT($x = y$) (Table 4).§PRED~~

10. ~~Unordered(x, y), or x ? y, returns the value TRUE if x is *unordered* with y, and returns FALSE otherwise (Table 4).~~§PRED

11. stringtofloating(x) converts a 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. **§CONV**

12. Class(x) tells which of the following ten classes x falls into: ~~optional signaling NaN, quiet~~ NaN, $-\infty$, negative normal nonzero, negative subnormal ~~denormalized~~ **§2**, –0, +0, positive subnormal ~~denormalized~~ **§2**, positive normal nonzero, $+\infty$. This function is never exceptional~~, not even for signaling NaNs~~. [Should class(x) distinguish between canonical and non-canonical representations?]

Appendix 6

Extensions §SNAN

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

~~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 this the standard. Thus an implementation supporting such enhancements should distinguish two different kinds of NaN, signaling and quiet, shall be supported in all operations.~~

~~For every operation listed in sections 5.1-5.7, signaling NaNs shall be reserved operands that signal the invalid operation exception (7.1) and, if no trap occurs and if a floating-point result is to be delivered, shall by default deliver a quiet NaN.~~

~~Signaling NaNs shall be distinguished from quiet NaNs by the value of the most significant bit of the fraction field: 0 for signaling, 1 for quiet. §5~~

~~Note that to be consistent with those operations listed as quiet in 5.8 and 5.9, an implementation supporting the use of signaling NaNs as links to numeric values, perhaps with higher precision or exponent range, shall link only the magnitude of the numeric value, storing its sign bit in the sign bit of the signaling NaN. Whether copying a signaling NaN without a change of format signals the invalid operation exception is the implementor's option.~~ **§SNAN**

[Programming environments that intend to support general-purpose computation should provide a means to extend the standard to support arithmetic upon values that may not be representable in the supported formats: uninitialized data, extra precision, extra exponent range... Because numbers may only represent themselves, such extensions must use NaNs. The hardware and software means to provide, enable, and exploit such extensions necessarily vary among systems providing them; one approach endorsed by this standard's predecessor, ANSI/IEEE Std 754–1985, is to define some or all NaNs as signaling: these generate an exception and trap whenever attempted to be used as an operand in an arithmetic operation. Because this standard defines all NaNs as quiet, signaling NaN exceptions and traps

must be enabled by implementation-defined means and thus are neither portable nor the default.]

Appendix 8 Traps §TRAP

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

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 ~~he~~ should be able to request that an existing handler be disabled, saved, or restored. The user ~~he~~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^{α} and then rounding. The bias-adjust α is (3/2) of the format's exponent bias from Table 1b: thus 192 in the binary single format. ~~1536 in the double, 24576 in the quad §Q, and $3 \times 2^{n-2}$ in the extended format, when n is the number of bits in the exponent field.~~ [**FOOTNOTE 7:** ~~The~~ This bias-adjust ~~is chosen to translate~~ translates 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 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^{α}. But for some integral value $n > \alpha$, Z / b^{n} would round to a normal number z in the destination format, $1 <= |z| < b$. 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^{α} and then rounding. The bias-adjust α is (3/2) of the format's exponent bias from Table 1b: thus 192 in the binary single format. ~~The bias-adjust α is 192 in the single, 1536 in the double, 24576 in the quad §Q, and $3 \times b^{n-2}$ in the extended format, where n is the number of bits in the exponent field.~~ [**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 correct~~ly~~ subnormal ~~denormalized~~ **§2** result may be produced in system software when the user underflow trap is disabled.] ~~Trapped underflows on conversion shall be handled analogously to the handling of overflows on conversion.~~

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^{α}. But for some integral value $n > \alpha$, $Z \times b^n$ would round to a normal number z in the destination format, $1 <= |z| < b$. 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 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