An American National Standard

IEEE Standard for Radix-Independent Floating-Point Arithmetic

Sponsor
Technical Committee on Microprocessors and Microcomputers of the IEEE Computer Society

Approved March 12, 1987
Reaffirmed March 17, 1994

IEEE Standards Board

Approved September 10, 1987
Reaffirmed August 23, 1994

American National Standards Institute

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE which have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments revived from users of the standard. Every IEEE Standard is subjected to review at least once every five years for revision or reaffirmation. When a document is more than five years old, and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
345 East 47th Street
New York, NY 10017
USA

Foreword

(This Foreword is not a part of ANSI/IEEE Std 854-1987, IEEE Standard for Radix-Independent Floating-Point Arithmetic.)

This standard is a product of the Radix-Independent Floating-Point Arithmetic Working Group of the Microprocessor Standards Subcommittee. This work was sponsored by the Technical Committee on Microprocessors and Minicomputers of the IEEE Computer Society. It generalizes ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic, to remove dependencies on radix and wordlength. The committee believes that, except for a possible conflict with the requirements in 5.6 and 7.2 that unrecognizable decimal input strings signal an exception, and in 6.3 that the sign of zero be preserved in certain conversion operations, any implementation conforming to ANSI/IEEE Std 754-1985 will also conform to this standard. In addition, the definition of *logb* has been enhanced in the Appendix, and two new functions, *conv* and *nearbyinteger*, have been added. Draft 1.0 of this standard was published to solicit public comments.[1]

This standard defines a family of commercially feasible ways for new systems to perform floating-point arithmetic. Issues of retrofitting were not considered. Among the desiderata that guided the formulation of this standard were the following:

1) Facilitate movement of existing programs from diverse computers to those that adhere to this standard.
2) Enhance the capabilities and safety available to 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, via minor editing and recompilation, onto any computer that conforms to this standard and possesses adequate capacity.
4) Provide direct support for
   a) execution-time diagnosis of anomalies,
   b) smoother handling of exceptions, and
   c) interval arithmetic at a reasonable cost.
5) Provide for development of
   a) standard elementary functions like exp and cos,
   b) very high precision (multiword) arithmetic, and
   c) coupling of numerical and symbolic algebraic computation.
6) Enable rather than preclude further refinements and extensions.

Members of the Radix-Independent Floating-Point Arithmetic Working Group who voted on versions of this proposal or participated by correspondence were as follows:

W. J. Cody, *Chair*

J. Bolstad	R. E. James, III	K. C. Ng
J. Boney	W. Kahan	K. A. Norman
L. Breed	R. Karpinski	J. F. Palmer
J. T. Coonen	V. Klema	R. Pexton
J. Demmel	E. LeBlanc	F. N. Ris
A. A. DuBrulle	C. Lewis	J. Ryshpan
P. J. Faillace	Z. Liu	D. Stevenson
A. Fyfe	R. Martin	R. Stewart
D. M. Gay	R. Mateosian	T. Suyehiro
R. Goodman	W. H. McAllister	H. C. Thacher, Jr
K. Hanson	E. McDonnell	J. W. Thomas
D. Hough	M. Mikalajunas	R. J. Wytmar

[1]IEEE *Micro*, vol 4, no 4, Aug 1984.

The Chairman of the Microprocessor Standards Committee at the time of approval was James Davis.

At the time of approval, the Technical Committee on Microprocessors and Microcomputers had as its Executive Committee:

Martin Freeman, *Chair*

James Flournoy Michael Smolin Robert Stewart

The following persons were on the balloting committee that approved this document for submission to the IEEE Standards Board:

A. Allison
P. Ashendem
G. Baldwin
R. Baker
L. Barcenas
M. Biewer
R. Boberg
P. Borrill
C. Camp
W. J. Cody
S. Cooper
J. Davis
R. Davis
S. Diamond
W. Fischer
J. Flournoy
G. Force
M. Freeman
D. Gustavson
T. Harkaway
R. Hochsprung
D. Hough
D. James
R. James
W. Kahan
L. Kaleda
R. Karpinski
H. Kirrman
D. Kraft
G. Langdon
T. Leonard
G. Lyons
R. McLellan
K. Mondal
J. Mooney
G. Nelson
D. Ogden
T. Pittman
S. Prital
B. Shields
M. Smolin
D. Stevenson
R. Stewart
M. Teener
S. Tetrick
E. Waltz
G. White
F. Whittington

When the IEEE Standards Board approved this standard on March 12, 1987, it had the following membership:

Donald C. Fleckenstein, *Chair*
Marco W. Migliaro, *Vice Chair*
Sava I. Sherr, *Secretary*

James H. Beall
Dennis Bodson
Marshall L. Cain
James M. Daly
Stephen R. Dillon
Eugene P. Fogarty
Jay Forster
Kenneth D. Hendrix
Irvin N. Howell
Leslie R. Kerr
Jack Kinn
Irving Kolodny
Joseph L. Koepfinger*
Edward Lohse
John May
Lawrence V. McCall
L. Bruce McClung
Donald T. Michael*
L. John Rankine
John P. Riganati
Gary S. Robinson
Frank L. Rose
Robert E. Rountree
William R. Tackaberry
William B. Wilkens
Helen M. Wood

*Member emeritus

CLAUSE PAGE

An American National Standard

IEEE Standard for Radix-Independent Floating-Point Arithmetic

1. Scope

1.1 Implementation Objectives

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 the following:

1) Constraints on parameters defining values of basic and extended floating-point numbers
2) Add, subtract, multiply, divide, square root, remainder and compare operations
3) Conversions between integers and floating-point numbers
4) Conversions between different floating-point precisions
5) Conversion between basic precision floating-point numbers and decimal strings
6) Floating-point exceptions and their handling, including nonnumbers (NaNs)

1.3 Exclusions

This standard does not specify the following:

1) Formats for internal storage of floating-point numbers
2) Encodings of integers and formats of strings of characters representing decimal numbers
3) Interpretation of the sign and significand fields of NaNs
4) Conversion between extended precision (3.2) floating-point numbers and decimal strings

2. Definitions

destination: The location for the result of a binary or unary operation. A destination may be either explicitly designated by the user or implicitly supplied by the system (that is, 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 precision as well as the operand's values.

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

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

fraction: The component of the significand that lies to the right of its implied radix point.

mode: A variable that a user may 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:

1) Rounding to control the direction of rounding errors
2) In certain implementations, rounding precision, to shorten the precision of results
3) The implementor may, at his option, implement the following modes: traps disabled or enabled, to handle exceptions

NaN: Not a number; a symbolic entry encoded in a floating-point format. There are two types of NaNs (see 6.2). Signaling NaNs signal the invalid operation exception (see 7.1) whenever they appear as operands. Quiet NaNs propagate through almost every arithmetic operation without signaling exceptions.

normal number: A nonzero number that is finite and not subnormal.

radix: The base for the representation of floating-point numbers.

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

significand: The component of a floating-point number that consists of a leading digit to the left of its implied radix point and a fraction field to the right.

status flag: A variable that may take two states, set and clear. A user may clear a flag, copy it, or restore it to a previous state. When set, a status flag may contain additional system-dependent information, possibly inaccessible to some users. The operations of this standard may as a side effect set some of the following flags: inexact result, underflow, overflow, divide by zero, and invalid operation.

subnormal number: A nonzero floating-point number whose exponent is the precision's minimum and whose leading significant digit is zero.[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. Precisions

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

3.1 Sets of Values

The standard does not specify how to encode numbers for internal storage. Four integer parameters specify each precision:

[2] Subnormal numbers are called denormalized numbers in ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-point Arithmetic.

b = the radix
p = the number of base-b digits in the significand
$E_{\max}$ = the maximum exponent
$E_{\min}$ = the minimum exponent

The parameters are subject to the following constraints:

1) b shall be either 2 or 10 and shall be the same for all supported precisions
2) $(E_{\max}–E_{\min})/p$ shall exceed 5 and should exceed 10
3) $b^{p-1} \geq 10^5$

The balance between the overflow threshold $(b^{E_{max}+1})$ and the underflow threshold $(b^{E_{min}})$ is characterized by their product $(b^{E_{max}+E_{min}+1})$, which should be the smallest integral power of b that is ≥4.

Each precision allows for the representation of just the following entities:

1) **Numbers of the form $(–1)^s$b$^E(d_0.d_1d_2…d_{p–1})$, where**
   s = an algebraic sign
   E = any integer between $E_{\min}$ and $E_{\max}$, inclusive
   d_i = a base-b digit ($0 \leq d_i \leq$ b–1)
2) Two infinities, +∞ and –∞
3) At least one signaling NaN
4) At least one quiet NaN

The algebraic sign provides additional information about any variable that has the value zero. Although all precisions have distinct representations for +0, –0, +∞ and –∞ 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 does not matter. An implementation may find it helpful to provide additional information about a variable that is NaN through an algebraic sign, but this standard does not interpret such extensions.

The foregoing description enumerates some values redundantly, that is,

$b^0(1.0) = b^1(0.1) = b^2(0.01) = \ldots$

but this standard does not distinguish them.

The standard allows an implementation to encode some values redundantly provided that it does not distinguish redundant encodings of nonzero values. An implementation may also reserve some digit strings for purposes beyond the scope of this standard.

3.2 Basic Precisions

3.2.1 Single

The narrowest precision supported shall be called *single precision*. When necessary to distinguish from other parameters, those defining single precision are denoted thus:

$E_{max_s}, E_{min_s}, p_s$

3.2.2 Double

When a second, wider basic precision is supported, it shall be called *double precision.* When necessary to distinguish from other parameters, those defining double precision are denoted thus:

$$E_{max_d}, E_{min_d}, p_d$$

In addition to the requirements specified in 3.1, parameters for double precision shall satisfy

$$b^{p_d} \geq 10b^{2p_s}$$

$$E_{max_d} \geq 8E_{max_s} + 7$$

$$E_{min_d} \leq 8E_{min_s}$$

3.3 Extended Precisions

The two extended precisions, single-extended and double-extended, are implementation dependent. When necessary to distinguish from other parameters, those defining, for example, single-extended are denoted thus:

$$E_{max_{se}}, E_{min_{se}}, p_{se}$$

Parameters for single-extended shall satisfy

$$E_{max_{se}} \geq 8E_{max_s} + 7$$

and

$$E_{min_{se}} \leq 8E_{min_s}$$

If $b \neq 10$, p_{se} must be large enough to support conversion to and from decimal strings (see 5.6). Thus, for $b = 2$, condition $p_{se} \geq p_s + \lceil \log_2(E_{max_s} - E_{min_s}) \rceil$ shall be satisfied. For all b, the condition $p_{se} \geq 1.2p_s$ shall be satisfied. In addition, the following condition should be satisfied to protect against error in the computation of y^x:

$$p_{se} > 1 + p_s + \frac{\ln\{3\ln(b)[E_{\max} + 1]\}}{\ln(b)}$$

Double-extended precision bears the same relation to double precision as single-extended bears to single precision.

Note that double precision satisfies the requirements for single-extended precision.

3.4 Combinations of Precisions

All implementations conforming to this standard shall support single precision. Implementations should support the extended precision corresponding to the widest basic precision supported and need not support any other extended precision.[3]

[3]Only if upward compatibility and speed are important issues should a system supporting the double-extended precision also support single-extended.

4. Rounding

Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit the destination's precision while signaling the inexact exception (see 7.5). Except for conversion between floating-point numbers and decimal strings (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 that result rounded 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 (see 6.3), and do affect the thresholds beyond which overflow (see 7.3) and underflow (see 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 digit even shall be delivered. However, an infinitely precise result with magnitude at least $b^{E_{max}}(b - 1/2b^{1-p})$ shall round to ∞ with no change in sign; here $E_{\max}$ and p are determined by the destination precision (see Section 3.) unless overridden by a rounding precision mode (see 4.3).

4.2 Directed Roundings

An implementation shall also provide three user-selectable directed rounding modes: round toward +∞, round toward –∞, and round toward 0.

When rounding toward +∞, the result shall be the precision's value (possibly +∞) closest to and no less than the infinitely precise result. When rounding toward –∞, the result shall be the precision's value (possibly –∞) closest to and no greater than the infinitely precise result. When rounding toward 0, the result shall be the precision'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 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 double or extended precision with its wider exponent range.[4] Similarly, a system that delivers results only to double-extended destinations shall permit the user to specify rounding to single or double precision. Note that to meet the specifications in 4.1, the result cannot suffer more than one rounding error.

5. Operations

All conforming implementations of this standard shall provide operations to add, subtract, multiply, divide, extract the square root, find the remainder, round to a floating-point integer, convert between different floating-point precisions, convert between floating-point numbers and integers, convert between internal floating-point representations and decimal strings, and compare. Whether copying without change of precision is considered an operation is an

[4]Control of rounding precision is intended to allow systems whose destinations are always double or extended to mimic, in the absence of overflow/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 just one rounding.

implementation option. Except for conversion between internal floating-point representations and decimal strings, each of the operations shall be performed as ff 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 precision (see Sections 4. and 7.) Section 6. augments the following specifications to cover ±∞, ±∞, and NaN. Section 7. enumerates exceptions caused by exceptional operands and exceptional results.

5.1 Arithmetic

An implementation shall provide the add, subtract, multiply, divide, and remainder operations for any two operands of the same precision, for each supported precision; it should also provide the operations for operands of differing precisions. The destination precision (regardless of the rounding precision control of 4.3) shall be at least as wide as the wider operand's precision. 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 $r = x - y \cdot n$, where n is the integer nearest the exact value of x/y; whenever $| n - x/y | = {}^1/_2$, then n is even. Thus, the remainder is always exact. If $r = 0$, its sign shall be that of x. Precision control (see 4.3) shall not apply to the remainder operation.

5.2 Square Root

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

5.3 Floating-Point Precision Conversions

It shall be possible to convert floating-point numbers between all supported precisions. If the conversion is to a narrower precision, the result shall be rounded as specified in Section 4., and consequently may signal inexact. Conversion to a wider precision is exact.

5.4 Conversion Between Floating Point and Integer

It shall be possible to convert between all supported floating-point precisions and all supported integer encodings. Conversion to integer shall be effected by rounding as specified in Section 4. When no other exception arises, this operation signals inexact whenever its result differs in value from its operand. Conversions between floating-point integers and integer encodings shall be exact unless an exception arises as specified in 7.1.

5.5 Round Floating-Point Number to Integral Value

It shall be possible to round a floating-point number to an integral valued floating-point number in the same precision. 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 haft, the rounded result is even. This operation signals inexact whenever its argument differs in value from its result. This operation leaves zeros and infinities unchanged.

5.6 Floating-Point ⟷ Decimal String Conversion

Conversion between decimal strings (strings of characters representing decimal numbers) in at least one format and floating-point numbers in all supported basic precisions shall be provided for numbers throughout the ranges specified

in Table 1. The nonnegative integers D and N in Tables 1 and 2 describe decimal strings having values $\pm M \cdot 10^{\pm N}$ where $0 \le M \le 10^{D}-1$.

When there is more than one choice for M and N with $M \le 10^{D}-1$, then Table 1 and the following discussion apply to the choice having the smallest value of N. (In effect, trailing zeros are stripped from or appended to M, subject to $M \le 10^{D}-1$, to minimize N.) When M lies beyond the bound specified by Max D in Table 1, that is, when $M \ge 10^{\mathrm{Max}D}$, the implementor may, at his option, round off all significant digits after the Max Dth to other decimal digits, typically 0, and should signal inexact (see 7.5) when nonzero digits have been discarded. When the destination is a decimal string, its least significant digit should be located by format specifications for purposes of rounding. Note that the largest possible value of N may be less than the boundary specified in Table 1 when the destination is a decimal string.

Conversions shall be correctly rounded as specified in Section 4. for operands lying within the ranges specified in Table 2. Otherwise, for rounding to nearest and b=2, the error in the converted result shall not exceed by more than e units in the destination's least significant digit the error that would be incurred by the rounding specifications of Section 4., provided that exponent overflow/underflow does not occur. Here $\in$ shall satisfy the condition $\in < 0.5$; $\in \approx 0.47$ has been found to be achievable. In the directed rounding modes for b=2, the error shah have the correct sign and shall not exceed $1 + \in$ units in the last place.

Conversions shall be monotonic and preserve the sign, including the sign of zero. That is, increasing the value of a 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 floating-point number.

When rounding to nearest, conversion from floating-point to decimal string and back to floating-point shall be the identity as long as the decimal string is carried to the maximum precision specified in Table 1, namely, Max D digits.[5]

Table 1— Floating-Point ⟷ Decimal String Conversion Ranges

Max D	Max N
$\lceil p \log_{10}(b) + 1 \rceil$, $b \neq 10$	$10^{\lfloor \log_{10}(E_m) \rfloor + 1} - 1$
p, $b = 10$	

NOTE — Here $E_{\mathrm{m}} = \max\{D+(p-1-E_{\min})\log_{10}(b), (E_{\max}+1)\log_{10}(b)+1-D\}$.

Table 2— Correctly Rounded Conversion Ranges

b	Max D	Max N
2	$\lceil p \log_{10}(2) + 1 \rceil$	$\lfloor p_e \log_5(2) \rfloor$
10	p	$10^{\lfloor \log_{10}(E_m) \rfloor + 1} - 1$

NOTE — Here p_e denotes the smallest precision permissible as extended support for the basic precision p (3.3), and $E_m = \max\{D+(p-1-E_{\min})\log_{10}(b), (E_{\max}+1)\log_{10}(b)+1-D\}$.

[5]The properties specified in this section for conversions are implied by error bounds that depend on the floating-point precision and the number of digits in the decimal string; the 0.47 mentioned is a worst-case bound derived for single precision on 32-bit binary machines. For a detailed discussion of these error bounds and economical conversion algorithms that exploit the extended precision on 32-bit binary machines, see J. T. Coonen, "Contributions to a Proposed Standard for Binary Floating-Point Arithmetic," *Ph.D. Dissertation*, University of California, Berkeley, CA 1984.

If decimal string to floating-point conversion overflows/underflows, the response is as specified in Section 7.. Overflow/underflow and NaNs and infinities encountered during floating-point to decimal string conversion should be indicated to the user by appropriate strings. The letters “NAN,” case insensitive, optionally preceded by an algebraic sign, should be the first characters of a string representing a NAN. The remainder of the string may be used for system-dependent information on output, and may be ignored on input. Unless recognized as a quiet NAN on input, an input NAN should become a signaling NAN. The letters “inf” or “infinity,” case insensitive, optionally preceded by an algebraic sign, should be the characters representing signed infinity. Either floating-point shall be the identity as long as the decimal string is carried to the maximum precision specified in Table 1, namely, Max *D* digits.[5] representation may be produced on output; both should be accepted on input.

The default action for attempting to convert an unrecognizable input decimal string is to signal an invalid operation exception.

To avoid inconsistencies, the procedures used for floating-point ⟷ decimal string conversion should give the same results regardless of whether the conversion is performed during language translation (interpretation, compilation, or assembly) or during program execution (runtime and interactive input/output).

5.7 Comparison

It shall be possible to compare floating-point numbers in all supported precisions, even if the operands' precisions differ. Comparisons are exact and never overflow or underflow. Four mutually exclusive relations are possible: “less than,” “equal,” “greater than,” and “unordered.” The last case arises only when at least one operand is a NaN. Every NaN shall come pare “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 (see 7.1) shall be signaled when, as indicated in the last column of Table 3, “unordered” operands are compared using one of the predicates involving “<” or “>” but not “?.” (Here the symbol “?” signifies “unordered.”)

Table 3 exhibits the twenty-six functionally distinct useful predicates named, in the first column, using three notations: *ad hoc*, Fortran-like, and mathematical. It shows how they are obtained from the four condition codes and tells which predicates 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.

Note that predicates come in pairs, each a logical negation of the other; applying a prefix like “NOT” to negate a predicate in Table 3 reverses the true/false sense of its associated entries, but leaves the last column's entry unchanged.[6]

[6]There 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, 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 $X \neq Y$). However, this coincidence does not occur for the other predicates. For instance, the logical negation of ($X < Y$) is just NOT($X < Y$); the reversed predicate (X ?>= Y) is different in that it does not signal an invalid operation exception when X and Y are “unordered.”

Table 3— Predicates and Relations

Predicates			Relations				Exception
Ad hoc	Fortran-like	Math	Greater than	Less than	Equal	Unordered	Invalid if unordered
=	.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
?	.UN.		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
NOT(?>=)			F	T	F	F	No
NOT(?<)			T	F	T	F	No
NOT(?<=)			T	F	F	F	No
NOT(?=)			T	T	F	F	No

Implementations that provide predicates shall provide the first six predicates in Table 3 and should provide the seventh, as well as a means of logically negating predicates.

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, –∞< (every finite number) <+∞.

Arithmetic on ∞ is always exact and therefore shall signal no exceptions, except for the invalid operations specified for ∞ in 7.1. The exceptions that do pertain to ∞ are signaled only when

1) ∞ is created from finite operands by overflow (see 7.3) or division by zero (see 7.2), with the corresponding trap disabled, or
2) ∞ is an invalid operand (see 7.1)

6.2 Operations with NaNs

Two different kinds of NaNs, 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 this 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 basic precision conversions.

Signaling NuNs 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 precision signals the invalid operation exception is the implementor's option.

Every operation involving a signaling NaN or invalid operation (7.1) shall, if no trap occurs and if a floating-point result is to be delivered, deliver a quiet NaN as its result.

Every operation involving one or two input NuNs, 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 NuNs. Note that precision conversions might be unable to deliver the same NaN. Quiet NuNs have effects similar to signaling NuNs on operations that do not deliver a floating-point result; these operations, namely, comparison and conversion to a precision that has no NaNs, are discussed in 5.4, 5.6, 5.7, and 7.1.

6.3 The Algebraic Sign

The sign of a NaN is not determined by this standard. If the result of an operation is not a NaN, then the following rules apply. The sign of a product or quotient is "-" if and only ff the operands have opposite 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. The sign of the result of the operation that rounds floating-point numbers to integral values (5.5), and of precision (5.3) and decimal string (5.6) conversion shall be the sign of the operand. These rules shall apply even when operands or results are zero or infinite. Conversion of zero between floating-point and integer encodings (5.4) shall preserve the sign unless the sign of -0 cannot be distinguished in the integer encoding, in which case the result shall be +0.

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 –∞, in which mode that sign shall be "-." However, $x + x = x -(-x)$ retains the same sign as x even when x is zero.

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

7. Exceptions

There are five types of exceptions that shall be signaled when detected. The signal entails setting a status flag. taking a trap, or possibly doing both. With each exception should be 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.

For each type of exception, the implementation shall provide a status flag that shall be set on any occurrence of the corresponding exception when no corresponding trap occurs. It shall be 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 floating-point result delivered when the exception occurs without a trap shall be a quiet NaN (see 6.2). The invalid operations are as follows:

1) Any operation on a signaling NaN (6.2)
2) Addition or subtraction: magnitude subtraction of infinities such as (+∞) + (–∞)
3) Multiplication: 0 · ∞
4) Division: 0/0 or ∞/∞
5) Remainder: *x* REM *y*, where *y* is zero or *x* is infinite
6) Square root if the operand is less than zero
7) Conversion of an internal floating-point number to an integer or to a decimal string when overflow, infinity, or NaN precludes a faithful representation in that format and this cannot otherwise be signaled
8) Conversion of an unrecognizable input string
9) Comparison via predicates involving “<” or “>,” without “?,” when the operands are “unordered” (5.7, Table 3)

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 result, when no trap occurs, shall be a correctly signed ∞ (6.3).

7.3 Overflow

The overflow exception shall be signaled whenever the destination precision's largest finite number vis exceeded in magnitude by what would have been the rounded floating-point result (Section 4.) were the exponent range unbounded. The result, when no trap occurs, 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 precision's largest finite number with the sign of the intermediate result
3) Round toward –∞ carries positive overflows to the precision's largest finite number and carries negative overflows to –∞

4) Round toward +∞ carries negative overflows to the precision's most negative finite number and carries positive overflows to +∞

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 exponent adjustment a for a precision shall be chosen to be approximately $3 \cdot (E_{\max}\text{-}E_{\min})/4$ for that precision, and should be divisible by twelve.[7] Trapped overflow on conversion from a floating-point precision shall deliver to the trap handler a result in that precision or a wider precision, possibly with the exponent adjusted, but rounded to the destination's precision. Trapped overflow on decimal string to floating-point conversion shall deliver to the trap handler a result in the widest supported precision, possibly with the exponent adjusted, but rounded to the destination's precision; when the result lies too far outside the range for the exponent to be adjusted, a quiet NaN or an appropriately signed infinity shall be delivered instead.

7.4 Underflow

Two correlated events contribute to underflow. One is the creation of a tiny nonzero result between $\pm b^{E_{min}}$ 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 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 $\pm b^{E_{min}}$ or
2) Before rounding: When a nonzero result computed as though both the exponent range and the precision were unbounded would lie strictly between $\pm b^{E_{min}}$

Loss of accuracy may be detected as either

1) A denormalization loss: When the delivered result differs from what would have been computed were the exponent range unbounded or
2) An inexact result: When the delivered result differs from what would have been computed were both the exponent range and precision unbounded (this is the condition called inexact in 7.5)

When an underflow trap is not implemented or is not enabled (the default case), underflow shall be signaled (via 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, or $\pm b^{E_{min}}$. 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} before rounding, where the exponent adjustment α shall be the same as in 7.3.[8] Trapped underflows on conversion shall be handled analogously to the handling of trapped overflows on conversion, with zero in place of infinity.

7.5 Inexact

If the rounded result of an operation is not exact or if it overflows without an overflow 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.

[7]The exponent adjustment is chosen to translate overflowed/underflowed results as nearly as possible to the middle of that exponent range so that, ff desired, they can be used in subsequent scaled operations with less risk of causing further exceptions.

[8]Note that a system whose underlying hardware always traps on underflow, producing a rounded, exponent-adjusted result, must indicate whether such a result is rounded up in magnitude in order that the correct subnormal result may be produced in system software when the user underflow trap is disabled.

8. Traps

A user should be able to request a trap on any of the five exceptions by specifying a handler for it. He should be able to request that an existing handler be disabled, saved, or restored. 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 flag(s) corresponding to the exceptions being signaled with their associated traps enabled may be undefined unless set or reset by the trap handler.

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

1) Which exception(s) occurred on this operation
2) The kind of operation that was being performed
3) The destination's precision
4) In overflow, underflow, and inexact exceptions, the correctly rounded result, including information that might not fit in the destination's precision
5) In invalid operation and divide by zero exceptions, the operand values

8.2 Precedence

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

Annex Recommended Functions and Predicates

(Informative)

(This Appendix is not a part of ANSI/IEEE Std 854-1987, IEEE Standard for Radix-Independent Floating Point Arithmetic, but is included for information only.)

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 below, like the copy operation $y := x$ without change or precision, may at the implementor's option be treated as nonarithmetic operations that neither signal underflow for subnoraml operands nor signal the invalid operation exception for signaling NaNs; the functions in question are (1), (2), (6), and (7).

1) copysign (x, y) returns x with the sign of y. Hence, abs (x) := copysign (x, 1.0), even if x is NaN.
2) $-x$ is x copied with its sign reversed, not $0-x$; the distinction is germane when x is ± 0 or NaN. Consequently, it would be a mistake to use the algebraic sign to distinguish signaling NaNs from quiet NaNs.
3) scal$b$(x,N) returns $x \cdot b^N$, for integral values N without computing b^N. Overflow and underflow should be handled in the same way as for decimal string to floating-point conversion.
4) log$b$ (x) returns the exponenet of x, as though x were represented with infinite range, as a signed interger in the precision of x, except that log$b$ (NaN) is NaN, log$b$(∞) is $+\infty$, and log$b$ (0) is $-\infty$ and signals the division by zero exception. For x positive and finite, $1 \le$ abs (scal$b$ (x, –log$b$(x))) < b.
5) nextafter (x,y) returns the next representable neighbor of x in the direction toward y. The following special cases arise: if $x=y$, then the result is x without any exception being signaled; otherwise, if either x or$y$ is a quiet NaN, then the result is one or the other of the input NaNs. Overflow is signaled when x is finite but nextafter (x,y) lies strictly between $\pm b^{E_{\min}}$; in both cases, inexact is signaled.
6) finite (x) returns the value TRUE if $-\infty < x < +\infty$, and returns FALSE otherwise.
7) isnan (x), or equivalently $x \neq y$, returns the value TRUE if x is a NaN, and returns FALSE otherwise.
8) $x<>y$ is TRUE only when $x<y$ or $x>y$, and is distinct from $x \neq y$, which menas NOT ($x=y$) (Table 3).
9) unordered (x,y) or $x?y$, returns the value TRUE if x is unordered with y, and returns FALSE otherwise (Table 3).
10) class (x) tells which of the following ten classes x falls into; signaling NaN, quiet NaN, $-\infty$, negative normal, negative subnormal, –0, +0, positive subnormal, positive normal, $+\infty$. This function is never exceptional, not even for signaling NaNs.
11) conv(x) converts the 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 currect rounding direction and precision as specified in 5.6.
12) nearbyinterger (x) is the opertion of 5.5 without an inexact exception.