

*An American National Standard*

**IEEE Standard for  
Binary 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

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

**denormalized number.** A nonzero floating-point number whose exponent has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero.

**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 (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.

**exponent.** The component of a binary floating-point number that normally signifies the integer

power to which two 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 binary 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: rounding, to control the direction of rounding errors. In certain implementations, rounding precision may be required, to shorten the precision of results.

The implementor may, at his option, implement the following modes: traps disabled/enabled, to handle exceptions.

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

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

**significand.** The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right.

**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.

**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.

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

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

$p$  = the number of significand 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  $(-1)^s 2^E (b_0 \cdot b_1 b_2 \cdots b_{p-1})$

where

$s = 0$  or  $1$

$E =$  any integer between  $E_{\min}$  and  $E_{\max}$ , inclusive

$b_i = 0$  or  $1$

Two infinities,  $+\infty$  and  $-\infty$

At least one signaling NaN

At least one quiet NaN

The foregoing description enumerates some values redundantly, for example,  $2^0(1 \cdot 0) = 2^1(0 \cdot 1) = 2^2(0 \cdot 01) = \cdots$ . However, the encodings of such nonzero values may be redundant only in extended formats (3.3). The nonzero values of the form  $\pm 2^E (0 \cdot b_1 b_2 \cdots b_{p-1})$  are called denormalized. Reserved exponents may be used to encode NaNs,  $\pm\infty$ ,  $\pm 0$ , and denormalized num-

Table 1  
Summary of Format Parameters

Parameter	Format			
	Single	Single Extended	Double	Double Extended
$p$	24	$\geq 32$	53	$\geq 64$
$E_{\max}$	+127	$\geq +1023$	+1023	$\geq +16383$
$E_{\min}$	-126	$\leq -1022$	-1022	$\leq -16382$
Exponent bias	+127	unspecified	+1023	unspecified
Exponent width in bits	8	$\geq 11$	11	$\geq 15$
Format width in bits	32	$\geq 43$	64	$\geq 79$

EXONENT BIAS TO RADIX 2 +126

+1022

bers. 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  $\infty$  are written without a sign when the sign is not important.

3.2 Basic Formats. Numbers in the single and double formats are composed of the following three fields:

- (1) 1-bit sign  $s$
- (2) Biased exponent  $e = E + bias$
- (3) Fraction  $f = \cdot b_1 b_2 \dots b_{p-1}$

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 val-

ues  $E_{\min} - 1$  to encode  $\pm 0$  and denormalized numbers, and  $E_{\max} + 1$  to encode  $\pm \infty$  and NaNs. The foregoing parameters are given in Table 1. Each nonzero numerical value has just one encoding. The fields are interpreted as follows:

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

- (1) If  $e = 255$  and  $f \neq 0$ , then  $v$  is NaN regardless of  $s$
- (2) If  $e = 255$  and  $f = 0$ , then  $v = (-1)^s \infty$
- (3) If  $0 < e < 255$ , then  $v = (-1)^s 2^{e-127} (1 \cdot f)$
- (4) If  $e = 0$  and  $f \neq 0$ , then  $v = (-1)^s 2^{-126} (0 \cdot f)$  (denormalized numbers)
- (5) If  $e = 0$  and  $f = 0$ , then  $v = (-1)^s 0$  (zero)

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

Fig 1  
Single Format

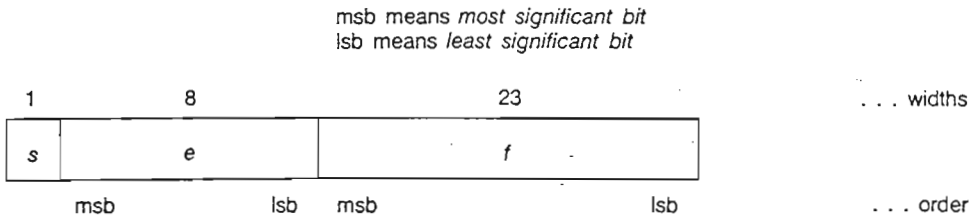
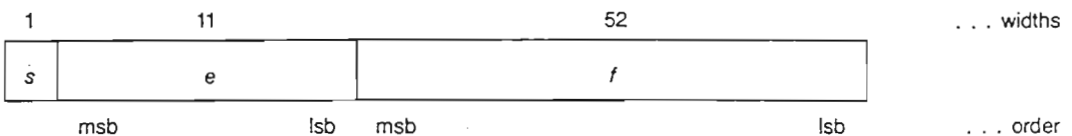


Fig 2  
Double Format



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

**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 non-zero value uniquely or it shall not distinguish redundant encodings of nonzero values. An implementation may also reserve some bit strings for purposes beyond the scope of this standard. When such a reserved bit string occurs as an operand the result is not specified by this standard.

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

**3.4 Combinations of Formats.** All implementations conforming to this standard shall support the single format. Implementations should support the extended format corresponding to the widest basic format supported, and need not support any other extended format.<sup>3</sup>

## 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  $\leftrightarrow$  decimal conversion (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

<sup>3</sup>Only if upward compatibility and speed are important issues should a system supporting the double extended format also support single extended.

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  $2^{E_{\max}}(2-2^{-p})$  shall round to  $\infty$  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.

*APPLIES TO OPERATIONS, "DESTINATION" IS "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 the double or extended format with its wider exponent range.<sup>4</sup> 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.

<sup>4</sup>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.

## 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 integer in floating-point format, convert between different floating-point formats, convert between floating-point and integer formats, convert binary  $\leftrightarrow$  decimal, and compare. Whether copying without change of format is considered an operation is an implementation option. Except for binary  $\leftrightarrow$  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  $\pm 0$ ,  $\pm\infty$ , 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 format, for each supported format; it should also provide the 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 the wider operand's format. All results shall be rounded as specified in Section 4.

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

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

**5.3 Floating-Point Format Conversions.** It shall be possible to convert floating-point numbers between all supported formats. If the conversion is to a narrower precision, the result

shall be rounded as specified in Section 4. Conversion to a 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.

**5.6 Binary  $\leftrightarrow$  Decimal 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  $M \geq 10^9$  for single or  $10^{17}$  for double, the implementor may, at his option, alter all significant digits after the ninth for single and seventeenth for double to other decimal digits, typically 0.

Conversions shall be correctly rounded as specified in Section 4 for operands lying within

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

MATHEMATICALLY  
WRONG  
 $\sqrt{3}$  ALWAYS  
IN RIGHT  
HALF  
PLANE!  
 $\sqrt{-0} \Rightarrow +0$



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.<sup>5</sup>

If decimal to binary conversion over/underflows, the response is as specified in Section 7. Over/underflow and 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  $\leftrightarrow$  decimal conversion should give the same results regardless of whether the conver-

sion is performed during language translation (interpretation, compilation, or assembly) or during program execution (run-time and interactive input/output).

**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 in Table 4, last column, *unordered* operands are compared using one of the predicates involving  $<$  or  $>$  but not  $?$  (Here the symbol  $?$  signifies *unordered*).

Table 4 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 such as NOT to negate a predicate in Table 4 reverses the true/false sense of its associated entries, but leaves the last column's entry unchanged.<sup>6</sup>

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.

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

<sup>5</sup>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.

<sup>6</sup>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  $\text{NOT}(X = Y)$  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 example, the logical negation of  $(X < Y)$  is just  $\text{NOT}(X < Y)$ , the reversed predicate  $(X ? > = Y)$  is different in that it does not signal an invalid operation exception when  $X$  and  $Y$  are *unordered*.

Table 4  
Predicates and Relations

Predicates			Relations						Exception
<i>Ad hoc</i>	FORTTRAN	Math	Greater Than	Less Than	Equal	Unordered		Invalid If Unordered	
=	.EQ.	=	F	F	T	F	F	No	
?<>	.NE.	≠	T	T	F	T	F	No	
>	.GT.	>	T	F	F	F	F	Yes	
>=	.GE.	≥	T	F	T	F	F	Yes	
<	.LT.	<	F	T	F	F	F	Yes	
<=	.LE.	≤	F	T	T	F	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	
NOT(?>=)			F	T	F		F	No	
NOT(?<)			T	F	T		F	No	
NOT(?<=)			T	F	F		F	No	
NOT(?=)			T	T	F		F	No	

## 6. Infinity, NaNs, and Signed Zero

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

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

(1)  $\infty$  is created from finite operands by overflow (7.3) or division by zero (7.2), with corresponding trap disabled

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

**6.2 Operations with NaNs.** Two different kinds of NaN, signaling and quiet, shall be supported in all operations. Signaling NaNs afford values for uninitialized variables and arithmetic-like en-

hancements (such as complex-affine infinities or extremely wide range) that are not the subject of the standard. Quiet NaNs should, by means left to the implementor's discretion, afford retrospective diagnostic information inherited from invalid or unavailable data and results. Propagation of the diagnostic information requires that information contained in the NaNs be preserved through arithmetic operations and floating-point format conversions.

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.

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 NaNs, none of them signaling, shall signal no exception but, if a floating-point result is to be de-

livered, 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 do have effects similar to signaling NaNs 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 an 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.

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

TECHNICAL ERROR - CSQRT( $x+i.0$ ) =  $x+i.0$   
 $x$  IS ALWAYS POSITIVE, THUS EVEN  $\sqrt{-0} = +0$   
 THIS IS A VALID SQUARE ROOT  
**7. Exceptions**  
 ALL RESULTS ARE IN RIGHT HALF PLANE

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 non-trapping 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 result, when the exception occurs without a trap, shall be a quiet NaN (6.2) provided the destination has a floating-point format. The invalid operations are

- (1) Any operation on a signaling NaN (6.2)
- (2) Addition or subtraction—magnitude subtraction of infinities such as,  $(+\infty) + (-\infty)$
- (3) Multiplication— $0 \times \infty$
- (4) Division— $0/0$  or  $\infty/\infty$
- (5) Remainder— $x \text{ REM } y$ , where  $y$  is zero or  $x$  is infinite
- (6) Square root if the operand is less than zero
- (7) Conversion of a binary 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
- (8) 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 result, when no trap occurs, shall be a correctly signed  $\infty$  (6.3).

**7.3 Overflow.** The overflow exception shall be signaled whenever the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result (Section 4) were the exponent range unbounded. 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  $\infty$  with the sign of the intermediate result
- (2) Round toward 0 carries all overflows to the format's largest finite number with the sign of the intermediate result
- (3) Round toward  $-\infty$  carries positive overflows to the format's largest finite number, and carries negative overflows to  $-\infty$
- (4) Round toward  $+\infty$  carries negative overflows to the format's most negative finite number, and carries positive overflows to  $+\infty$

Trapped overflows on all operations except conversions shall deliver to the trap handler the result obtained by dividing the infinitely precise result by  $2^\alpha$  and then rounding. The bias adjust  $\alpha$  is 192 in the single, 1536 in the double, and



$3 \times 2^{n-2}$  in the extended format, when  $n$  is the number of bits in the exponent field.<sup>7</sup> 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.

**7.4 Underflow.** Two correlated events contribute to underflow. One is the creation of a tiny nonzero result between  $\pm 2^{E_{\min}}$  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 denormalized 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  $\pm 2^{E_{\min}}$

(2) *Before rounding*—when a nonzero result computed as though both the exponent range and the precision were unbounded would lie strictly between  $\pm 2^{E_{\min}}$ .

Loss of accuracy may be detected as either

(3) *A denormalization loss*—when the delivered result differs from what would have been computed were exponent range unbounded.

(4) *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).

When an underflow trap is not implemented, or is not enabled (the default case), 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, denormalized, or  $\pm 2^{E_{\min}}$ . When an underflow trap has been imple-

mented 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  $2^\alpha$  and then rounding. The bias adjust  $\alpha$  is 192 in the single, 1536 in the double, and  $3 \times 2^{n-2}$  in the extended format, where  $n$  is the number of bits in the exponent field.<sup>8</sup> Trapped underflows on conversion shall be handled analogously to the handling of overflows on conversion.

**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.

## 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.

<sup>7</sup>The bias adjust is chosen to translate over/underflowed values as nearly as possible to the middle of the exponent range so that, if desired, they can be used in subsequent scaled operations with less risk of causing further exceptions.

<sup>8</sup>Note that a system whose underlying hardware always traps on underflow, producing a rounded, bias-adjusted result, shall indicate whether such a result is rounded up in magnitude in order that the correctly denormalized result may be produced in system software when the user underflow trap is disabled.

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 in-

formation that might not fit in the destination's format

- (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.