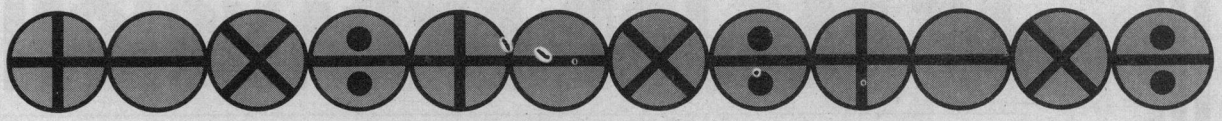
This guide to an IEEE draft standard provides practical algorithms for floating-point arithmetic operations and suggests the hardware/software mix for handling exceptions.



SPECIAL FEATURE

An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic

Jerome T. Coonen
University of California at Berkeley

This is an implementation guide* to a draft standard before an IEEE subcommittee whose goal is to standardize binary floating-point arithmetic for mini- and microcomputers. The purpose of the standard is to assure a uniform floating-point software environment for programmers. It may be implemented entirely in hardware or software or, as is most likely, in a combination of the two. This document provides reasonable algorithms for the arithmetic operations and suggestions for the hardware/software mix in handling exceptions.

Except for its additional discussion of *quad*, this guide is in concordance with Draft 5.11 of the proposal titled, "A Proposed Standard for Floating Point Arithmetic," IEEECS Task P754/D2, by John Palmer, Tom Pittman, William Kahan, David Stevenson, and J. T. Coonen.** W. Kahan made substantial contributions throughout the development of this document, and Harold Stone prepared a first draft in April 1978. J. Palmer discussed several features of this standard in late 1977.*** Comments may be sent to

Jerome T. Coonen
Department of Mathematics
University of California
Berkeley, CA 94720

*This is a much revised version of "Specifications for a Proposed Standard for Floating Point Arithmetic," Memorandum No. UCB/ERL M78/72. This work was partially funded by Office of Naval Research Contract N00014-76-C0013.

**J. Coonen, W. Kahan, J. Palmer, T. Pittman, D. Stevenson, "A Proposed Standard for Floating Point Arithmetic," *SIGNUM Newsletter*, Special Issue, Oct. 1979, pp. 4-12. Available from SIGNUM, c/o ACM, 1133 Avenue of the Americas, New York, NY 10036.

***J. Palmer, "The INTEL Standard for Floating-Point Arithmetic," *Proc. COMPSAC 77*, pp. 107-112.

The standard precisely describes its data formats and the results of arithmetic operations; it must do so to be of use to the producers of microprocessor hardware and software, who cannot afford to provide the support software and personnel to perform conversions between systems conforming to a less rigid standard. It allows for future developments such as interval arithmetic, which provides a certifiable result despite roundoff, Over/Underflow, and other exceptions. And it allows the use of reserved operands to extend the numerical data structure, with complex infinities, say, or with pointers into heaps of numbers with extended range and precision.

Programs which now run in higher-level languages like Fortran should be portable to a system with the new standard arithmetic at the cost of a modest amount of editing and a recompilation, and then should execute with results almost certainly no worse than before, though programs which used to give incorrect results might now give diagnostic messages instead.

1.0 Narrative description of the standard arithmetic

1.1 Sketch of the standard floating-point system.

Combinations of floating-point formats: one of
(A) *single*
(B) *single* and *single-extended*
(C) *single* and *double*
(D) *single, double,* and *double-extended*
(E) *single, double,* and *quad.*

Arithmetic operations:

Add, Subtract, Multiply, Divide, Remainder, Square Root, Compare, Round to Integer, Conversion between various floating-point and in-

0018-9162/80/0100-0068$00.75 © 1980 IEEE

teger formats, Binary-Decimal conversion.

Rounding modes:

(A) Round to Nearest, or optionally
(B) Round—to Nearest, toward 0, toward $+\infty$, toward $-\infty$.

Rounding precision control:

(A) Allow rounding of an *extended* result to the precision of any other implemented format, while retaining the extended exponent.
(B) When all operands have the same precision, allow rounding of the result to that precision.

Infinity arithmetic:

(A) Affine mode: $-\infty<+\infty$.
(B) Projective mode: $-\infty=+\infty$.

Denormalized arithmetic:

(A) Warning mode
(B) Normalizing mode (optional).

Floating-point exceptions, with sticky flags and specified results. The default response is to proceed; a trap to user software is optional.

(A) Invalid-Operation
(B) Overflow
(C) Underflow
(D) Division-by-Zero
(E) Inexact-Result.

1.2 Basic floating-point formats. Any nonzero real number may be expressed in "normalized floating-point" form as $\pm 2^{e}*f$, where e is the signed integer exponent and the significant digit field f satisfies $1 \leqslant f < 2$. The standard describes a machine representation of a finite subset of the real numbers based on this floating-point decomposition, and prescribes rules for arithmetic on them.

There are three basic formats, *single, double* and *quad* (See Table 1), to be implemented in one of the combinations shown in Section 1.1. *Single* is required since it is useful as a debugging precision and is efficient over a wide range of applications where storage economy matters.

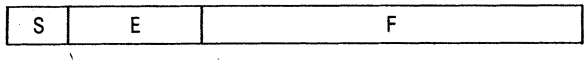
A normalized nonzero number X in the *single* format (see Section 2 for *double* and *quad*) has the form

$$X = (-1)^{S}*2^{E-127}*(1.F) \text{ where}$$

S = sign bit
E = 8-bit exponent biased by 127
F = X's 23-bit fraction which, together with an implicit leading 1, yields the significant digit field "1.—".

The values 0 and 255 of E are reserved to designate special operands discussed in later sections; one of them, signed zero, is represented by E = F = 0. Normalized nonzero *single* numbers can range in magnitude between $2^{-126}*1.000\ldots00$ and $2^{127}*1.111\ldots11$, inclusive.

The number X above is represented in storage by the bit string



This encoding has the special property that the order of floating-point numbers coincides with the lexicographic order of their machine counterparts when interpreted as sign-magnitude binary integers, facilitating comparisons of numbers in the same format.

1.3 Extended formats. To perform the arithmetic operations on numbers stored in the *single* and *double* formats, a system will generally unpack the bit strings into their component fields S, E, and F. Moreover, the leading significant bit will be made explicit, and perhaps the bias will be removed from the exponent.

The standard provides a way to exploit this unpacked format by admitting the optional *single-extended* and *double-extended* formats (See Table 2). If implemented at all, only one *extended* format should be provided, *single-extended* in systems with *single* only, and *double-extended* in systems with *single* and *double* only.

Table 1.
Basic floating-point formats.

	SINGLE	DOUBLE	QUAD
Fields and widths in bits:			
S = Sign	1	1	1
E = Exponent	8	11	15
L = Leading bit	(1)	(1)	1
F = Fraction	23	52	111
Total Width	(1)+32	(1)+64	128
Sign:	+/− represented by 0/1 respectively		
Exponent:	biased integer		
Max E	255	2047	32767
Min E	0	0	0
Bias of E	127	1023	16383
Normalized numbers:	(quad may be unnormalized)		
Range of E	(Min E + 1) to (Max E − 1)		
Represented number	$(-1)^{s} * 2^{E-Bias} *(L.F)$		
Signed zeros:			
E	Min E	Min E	Min E
L	(0)	(0)	0
F	0	0	0
Reserved operands:			
Denormalized numbers:			
E	Min E	Min E	Min E
L	(0)	(0)	0
F	nonzero	nonzero	nonzero
Represented number	$(-1)^{S}*2^{E-Bias}*(L.F)$		
Signed ∞'s:			
E	Max E	Max E	Max E
L	(0)	(0)	0 or 1
F	0	0	0
NaNs:			
E	Max E	Max E	Max E
L	(0)	(0)	0 or 1
F	nonzero	nonzero	nonzero

F = system-dependent, possibly diagnostic, information.

Double-extended format (see Section 2 for *single-extended)* consists of the following fields:

S=sign bit.
E+B=biased exponent; E is a signed integer spanning at least the range −16383 to 16384; the bias B may be zero
L.F=a leading integer bit L followed by a fraction F of at least 63 bits.

A number X is then given by $X=(-1)^{S}*2^{E-B}*(L.F)$. The case E = maximal-value is discussed in later sections. Two possible implementations of E = minimal-value are described below (Section 1.12, Denormalized and unnormalized numbers); signed zero is represented by E = minimal-value and L.F = 0.0. Zero is sometimes referred to as "normal zero" to distinguish it from the "unnormal zeros" with E > minimal-value and L.F = 0.0. The latter behave much as nonzero numbers in the arithmetic operations.

To match the exponent range of *quad* the unbiased *double extended* exponent must range between −16383 and 16384 as indicated above. This suggests that the exponent be represented in 15 bits by its negative in two's complement, biased by 16383 as in the basic formats, or biased by −1. The choice of the exponent representation impacts the use of the nonzero numbers at the bottom of the exponent range.

Table 2.
Extended formats.

	SINGLE-EXTENDED	DOUBLE-EXTENDED
Fields and widths in bits:		
S = Sign	1	1
E = Exponent	11	15
L = Leading bit	1	1
F = Fraction ⩾	31	63
Total width ⩾	44	80
Sign:	+ / − represented by 0/1 respectively	
Unbiased exponent:	(may be stored with a bias)	
Max E ⩾	1024	16384
Min E ⩽	−1023	−16383
Numbers:		
Range of E	(Min E + 1) to (Max E − 1)	
Represented number	$(-1)^{S}*2^{E}*(L.F)$	
Bottom of the exponent range:		
E	Min E	Min E
R	0 or 1	0 or 1
Represented number	$(-1)^{S}*2^{E+R}*(L.F)$	
Signed zeros:	use special indicator bits, or else...	
E	Min E	Min E
L.F	0.0	0.0
Reserved operands:		
Signed ∞'s:	use special indicator bits, or else...	
E	Max E	Max E
L	0 or 1	0 or 1
F	0	0
NaNs:	use special indicator bits, or else...	
E	Max E	Max E
L	0 or 1	0 or 1
F	nonzero	nonzero
F = system-dependent, possibly diagnostic, information.		

Extendeds are assumed to be few in number. The first implementations of this standard will probably allow access to *extended* entities only in assembly language. High-level languages will use *extended* (invisibly) to evaluate intermediate subexpressions, and later may provide *extended* as a declarable data type.

The presence of at least as many extra bits of precision in *extended* as in the exponent field of the basic format it supports greatly simplifies the accurate computation of the transcendental functions, inner products, and the power function Y^{X}. In fact, to meet the accuracy specifications for binary-decimal conversions, some extended capability must be simulated by system software if an *extended* format is not implemented; this is discussed in Section 2.

Another way to obtain most of the computational benefits of an *extended* format is to use the next wider basic format. Indeed, *quad* is included in this document as an alternative for those not wishing to implement *double-extended.* In most implementations *extended* will be as fast as the basic format it supports, as compared to a factor 2 or 4 loss in speed suffered by the next wider basic format, if implemented.

1.4 Arithmetic operations. The standard provides a notably complete set of arithmetic operations (see Section 1.1) in an attempt to facilitate program portability by guaranteeing that results obtained using standard arithmetic may be reproduced on different computer systems, down to the last bit if no *extended* format is used. SQUARE ROOT and REMAINDER are included as primitive operations because they appear so often, for example in matrix calculations and range reduction. REMAINDER is preferable to the MODULO function because REMAINDER is computed without rounding error. Consider, for example

0.01 MOD (−95) vs 0.01 REM (−95)

on a 2-digit machine. MODULO yields the result round (−94.99) = −95 for a complete loss of accuracy, while REMAINDER yields the correct result 0.01. The standard's specification of minimal requirements for binary-decimal conversions is an attempt to allow comparison of data from different systems at the decimal output level rather than via hexadecimal dumps.

All operations except conversions between different data formats are presumed to deliver their results to destinations having no less exponent range than their input operands. This constraint avoids unnecessary complexity in the implementation and simplifies the responses to Over/Underflow. The rare operation

*double * double → single*

is required to function exactly as

*double * double → double*
MOVE (round) double → single,

to assure identical results in all sequences of operations performed in the basic formats only.

Rather than prohibit mixed-format operations, the standard is designed to encourage the provision of some such operations. The sequence

(*single* * *single* → *double*) + *double* → *double*

ought to be available without the overhead of padding the *single* operands to *double*.

1.5 Accuracy and rounding. If the infinite precision result of an arithmetic operation is exactly representable within the exponent range and precision specified for the destination, then it must be given exactly. Otherwise the result must be rounded as follows. Let Z be the infinitely precise result of an arithmetic operation, bracketed most closely by Z1 and Z2, numbers representable exactly in the precision of the destination, but whose exponents may be out of range. That is, $Z1 \leqslant Z \leqslant Z2$, barely.

Round to Nearest(Z) = Unbiased Round (Z)
= the nearer of Z1 and Z2 to Z; in case of a tie choose the one of Z1 and Z2 whose least significant bit is 0.
Round toward Zero(Z) = Chop(Z) = smaller of Z1 and Z2 in magnitude.
Round toward $+\infty$(Z) = Z2.
Round toward $-\infty$(Z) = Z1.

The latter two modes, the "directed roundings," are intended to support interval arithmetic. Round toward Zero is useful in controlling conversions to integers in accordance with conventions embedded in programming languages like Fortran.

An implementation of the standard may support either Round to Nearest only, with Round toward Zero available for Round to Integer, or all four rounding modes. Round to Nearest shall be the default mode for all operations. Calculation of Round to Nearest requires the so-called sticky bit, as shown in Section 2. Once the sticky bit is implemented, the directed roundings may be supplied at very little extra cost, the bulk of which lies in the mechanism, say mode bits or extra opcodes for exercising the choice of rounding mode. While the standard leaves this mechanism up to the implementor, the mode bits are usually preferable. For example, an interval arithmetic computation of upper and lower bounds, performed by executing the same instructions rounding up during one pass and down the next, is greatly expedited if flipping a pair of bits changes rounding modes.

In a system which delivers all floating-point results except format conversions in the widest format supported, the user needs control over the precision to which a result is rounded. Such a system would encourage the evaluation of long expressions in the widest available format, with just one serious rounding error at the end when the expression's value is stored in a narrower destination. But the standard's specifications for roundoff control are burdened by the current programming languages which prohibit mixed-precision calculation, and by the need to mimic systems not providing an *extended* format. Rounding precision control is specified at the end of Section 2.14.

1.6 Exceptions. Once the data formats and operations are determined, there remains the specification of responses to exceptional conditions. The standard classifies the exceptions as Invalid-Operation, Underflow, Overflow, Division-by-Zero and Inexact-Result. They are discussed in the following sections.

The default response to any exception is to deliver a specified result and proceed. However, an implementation may provide optional traps to user software on any of the exceptions. If available, the choice to trap should be exercised at execution time via a trap-enable bit.

Associated with each of the exceptions is a "sticky" flag which is guaranteed to be set on each occurrence of the corresponding exception when there is no trap. The flags may be tested by a program and may be cleared only by the user's program. When the end of a job is obviously at hand, a humane operating system may draw the user's attention to flags still set.

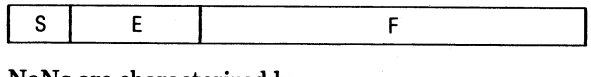
Since the sticky flags need not be set when a trap is to be taken, an implementation may use them to indicate which exceptions have just occurred. A trap handler could determine which exception(s) arose on the aborted operation by checking which have both their sticky and trap-enable flags set, and would then clear those flags at the end of the operation.

To deal effectively with traps, programmers need certain vital information, such as what exceptions occurred, where in the program, and what the operation and operands were. In response, the programmer will normally either depart from the offending block of code, fix up the aberrant result and resume execution, or reinterpret the aberrant operands and recompute the result. The trap handler might be passed information by value, with the option to "return" a result to be inserted to the offending operation's destination. One might dispense with some of the above information, for example when the correct result is available in encoded form as in Over/Underflow.

1.7 Invalid-Operation. The Invalid-Operation exception arises in a variety of arithmetic operations on errors not frequent or important enough to merit their own fault condition. Some samples of Invalid-Operations are:

(A) $\sqrt{-5}$
(B) $(+\infty) - (+\infty)$ (See Section 1.8.)
(C) $0*\infty$.

One class of reserved operands, the Not-a-Number symbols, or NaNs, are specified as the default results of Invalid-Operations. In *single, double,* and *quad* formats, with the format



NaNs are characterized by

S = sign bit (which may be irrelevant)
E = 111...11
F ≠ 0.

In *extended* format NaNs have the most positive exponent. The leading significant bit in *extended* and

quad may be 0 or 1. The sign bit S participates in the obvious way in the execution of statements like X=−Y and Z=X−Y=X+(−Y) without loss of information in the event that Y is a NaN with a numerical connotation.

The nonzero fraction field F of a NaN will contain system-dependent information. For example:

(A) A distinguished class of NaNs may be used by an operating system to initialize storage. The fraction of such a NaN may be a name or a pointer to the region where the NaN is stored.

(B) A NaN generated by an invalid arithmetic operation on numeric data, for example 0 * ∞, may be a pointer to the offending line or block of code.

(C) When complex arithmetic is implemented, it is often useful to think of ∞ as a line rather than a point in the projective plane. A distinguished class of NaNs may be used in pairs to provide the relative sizes and signs of the real and imaginary parts of numbers tending to ∞ along a fixed ray emanating from the origin.

(D) Sometimes an operation could generate a result acceptable but for its inability to pack that result correctly into the intended destination (see the discussion of Over/Underflows). In such a case, a NaN could be supplied, with a fraction pointing to an extended field or a heap where the correct result may be found.

(E) Sometimes a subroutine may encounter data for which only a partial result can be delivered in the time available. The rest of the result can be replaced by NaNs pointing to a piece of the program which will resume execution of that subroutine only if that undelivered portion of the result is really needed.

(F) List-oriented systems like LISP may use *single* format NaNs to point to *double* numerical data.

As the list above shows, there are two distinct types of NaNs. The Nontrapping NaNs, as in (A) and (B), propagate through arithmetic operations without precipitating exceptions. If two such NaNs are picked up as operands, the result is one of the operands, according to a system-dependent precedence rule. On the other hand, the Trapping NaNs would be useful in situations (C) through (F), where an Invalid-Operation trap to user software is required to perform arithmetic on the special operands; when the trap is disabled, a Nontrapping NaN results. The two types of NaNs might be distinguished by the leading bits of their fractions.

1.8 Underflow. Because of the care taken in the treatment of Underflows, the range of normalized numbers in *single, double,* and *quad* formats has been chosen to diminish slightly the risk of Overflow compared with the risk of Underflow. This was done by picking the exponent bias and alignment of the binary point in the significant digit field in such a way that the product of the largest and smallest positive normalized numbers is roughly 4 in each of the basic formats.

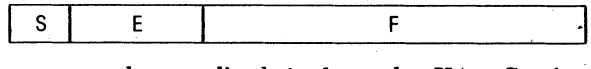
Underflow occurs if the exponent of a result, tested before or after rounding at the implementor's option, lies below the exponent range of the destination field, or if the rounded *extended* or *quad* result of a MULTIPLY or DIVIDE with nonzero, finite operands is normal zero. Note that a product or quotient of grossly unnormalized numbers may have a zero significant digit field; the test above prohibits such a result from masquerading as a normal zero when the operand exponents fortuitously add to the format's minimum.

Because of the restrictions on arithmetic operations presumed in Section 1.4, the exponent can be out of range by at most a factor of 2, except for the MOVE instruction which is discussed in Section 2. If the Underflow trap is enabled, the exponent is wrapped around into the desired range with a bias adjust specified in Section 2, and the resulting value is delivered to the trap handler. The exponent wrap-around is chosen so that the result, while related in a simple way to the Underflowed value, lies somewhere in the middle of the numerical range of representable numbers. This diminishes the risk that a computational response (like scaling) to Underflow will encounter almost immediately a rash of consequent Overflows. The analogous statement holds for Overflows.

If the Underflow trap is disabled, the result is denormalized by right-shifting its significant digit field while the exponent is incremented until it reaches that of the smallest normalized number representable in the destination. Then the result is rounded to fit into the destination.

Note that denormalization is performed before rounding, to avoid double-rounding problems. If the Underflow test is made on a rounded result, that result must be "unrounded" before undergoing denormalization. The difference between testing Underflow before and after rounding is that the Underflow threshold (i.e. the largest infinite precision number that Underflows) is the higher in the latter case by one quarter of a unit in the last place of the smallest normalized number; however, both implementations yield exactly the same numerical values.

In terms of the format



a nonzero denormalized *single* number X (see Section 2 for the other formats) is encoded as

S = sign bit
E = 0
F = X's 23 significant bits (at least one of which must be nonzero) to the right of the binary point.

X is reconstructed via the formula

$$X = (-1)^{S} * 2^{-126} * (0.F),$$

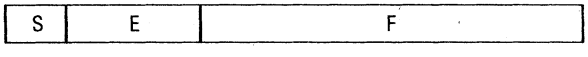
observing that E is not the true biased exponent in *single* format. Comparing this formula with its

analog for normalized numbers, one sees that, when unpacking a denormalized number, the 1-bit that would have gone to the leading bit of the significant digit field for a normalized number is instead added into the unbiased exponent E−127+1.

The denormalized numbers and signed zeros are the reserved operands corresponding to a biased exponent of zero. The values ±0 are obtained just when F=0 above. Zero may result from an Underflow, depending on the rounding mode, when the Underflow is so severe that all nonzero bits are shifted out of the significant digit field.

1.9 Overflow. If the exponent of a rounded result of an arithmetic operation overflows the range of the destination, then the Overflow exception arises, except when Invalid-Operation intervenes because a *single* or *double* result is not normalized. If a trap is to be taken, then the exponent is wrapped around as discussed in Underflow (Section 1.8), except that the bias adjust is subtracted rather than added.

If no trap is to be taken, then the result depends on the rounding mode and the sign of the result, as discussed in Section 2. One possible result is ∞, which in *single, double,* and *quad* formats with the bit pattern



is encoded as

S = sign bit
E = 111...11
F = 0.

In the *extended* formats E = maximal-value and F = 0. The explicit leading bit L in *extended* and *quad* may be 0 or 1.

The ∞'s are given two interpretations. In Affine mode

$$-\infty < \{\text{real numbers}\} < +\infty,$$

which is appropriate for most engineering calculations involving exponentials or disparate time constants or ∞'s generated by Overflows. The sign of ∞ is ignored in Projective mode, which is useful for real and complex rational arithmetic, for continued fractions, and for ∞'s generated by division by zeros not generated by Underflows. Systems shall provide an Affine/Projective mode bit so that the choice can be made under program control. Projective mode is the default because it is less likely to be abused unwittingly.

1.10 Division-by-Zero. The Division-by-Zero exception arises in a division operation when the divisor is normal zero and the dividend is a finite nonzero number. The default result is ∞ with sign according to convention.

1.11 Inexact-Result. The Inexact-Result exception arises when a roundoff error is committed in an arithmetic operation. It is intended for essentially integer calculation as in Cobol and to facilitate multiple-precision calculation. The default result is the correctly rounded number.

1.12 Denormalized and unnormalized numbers. In this document an unnormalized number is one whose leading significant bit, whether implicit or explicit, is zero. Denormalized numbers, nonzero unnormalized numbers in a given format whose exponents are the format's minimum, were introduced as the default results of Underflows. They are designed not so much to extend the exponent range, but rather to allow further computation with some sacrifice of precision in order to defer as long as possible the need to decide whether the Underflow will have significant consequences.

While in *extended* and *quad* formats, with their explicit leading bits, unnormalized numbers may range over the entire exponent range, the only unnormalized numbers that may be represented in *single* and *double* formats are denormalized.

Section 2 specifies the results of arithmetic operations on unnormalized operands; in each case the algorithms are essentially the same as for normalized operands. The only unnormalized result possible with normalized operands is a denormalized number on Underflow.

The usual mode of arithmetic on unnormalized numbers, which may be called Warning mode, recognizes operands' unnormalized character. But the standard allows an optional Normalizing mode in which all results are computed as though all denormalized operands had first been normalized. In a system that offers both, Warning mode shall be the default, and selection of modes shall be exercised via a single-mode bit accessible to programmers.

Normalizing mode precludes both the creation of any unnormalized numbers other than denormalized numbers, and Invalid-Operations due to the inability to store an unnormalized result in a *single* or *double* destination. It might be used by a programmer who has given some thought to Underflow, since, in most cases, the error due to denormalization on Underflow is no worse than that due to roundoff. Normalizing mode sacrifices the diagnostic capability of the unnormalized numbers for the predictability of normalized arithmetic. But if unexpected unnormalized (but not denormalized) operands are somehow picked up in that mode, they are operated on as in Warning mode.

Because it is so often desired, Normalizing mode is recommended for all systems, especially those without an *extended* format to hold unnormalized intermediates. In fact, the Normalizing mode is optional primarily to free the high-performance pipelined array processors from the extra normalizing step at the start of each operation; such systems will probably compute their intermediates in *extended.*

Another way to perform unnormalized arithmetic in *extended* format is according to the rules of significance arithmetic. This would be regarded as an (expensive) enhancement of the standard. If *quad* is implemented, then unnormalized arithmetic should

be performed as significance arithmetic to take advantage of the extravagant word size.

As mentioned in the discussion of the *extended* formats, the standard does not exactly specify the interpretation of the nonzero numbers whose exponents are the format's minimum. One natural implementation simply extends the exponent range one unit, interpreting a number with the format's smallest exponent as it would any other nonzero number. A problem arises since normal 0 can be the unexceptional product or quotient of grossly unnormalized or denormalized numbers. To protect against this anomalous situation, the standard specifies that such a product or quotient be marked as an Underflow. The extra test for normal zero is required after a product or quotient of nonzero numbers.

An alternative encoding of denormalized numbers in *extended* and *quad* formats uses a redundant exponent to permit numbers denormalized by Underflow to be distinguished from unnormalized numbers at the bottom of the exponent range which are the results of operations on unnormalized operands. In a scheme with biased exponent, with the notation introduced earlier,

(A) The nonzero normalized numbers with $E=0$ have exactly the same numeric connotation as their counterparts with $E=1$.

(B) The nonzero nonnormalized numbers with $E=0$ and $F\neq 0$ have the same numeric connotation as the corresponding numbers with $E=1$. Those with $E=0$ are denormalized while those with $E=1$ are unnormalized.

(C) The numbers with $E=L=F=0$ are the signed normal zeros. The numbers with $E\geq 1$ and $L=F=0$ are unnormal zeros.

In this representation normal zero can never be the product or quotient of nonzero operands unless exponent Underflow occurs (i.e., biased exponent less than 1), simplifying the test for Underflow. Also, in systems which implement Normalizing mode, there is a distinction between denormalized numbers and unnormalized numbers at the bottom of the exponent range. Another advantage, for those who implement the standard in hardware that traps to system software in all exceptional circumstances, is that E=maximal-value and E=minimal-value are the conditions for a hardware trap on "exceptional operand."

1.13 Hardware vs user traps. The standard specifies the trap options for exceptions independently of whether the implementation is in hardware, software, or a combination of the two. These are system traps to software that the user has either written or invoked from a system library. They are to be distinguished from hardware traps in the arithmetic unit.

One possible hardware/software implementation would provide a hardware trap to system software on every Over/Underflow. The system software would then test the trap option flag and either deliver the specified result and proceed, or trap to user software. In this case the exceptions' sticky flags and trap-enable bits could be in software. It is important to note that if the hardware trap provided the correctly rounded result with an extended exponent, then the system software would require sufficient information to "unround" the number in case a denormalized result is to be delivered on Underflow; otherwise a second rounding could occur during denormalization, in violation of the standard.

The Invalid-Operation and Division-by-Zero exceptions could be handled by similar hardware/software combinations.

Inexact-Result requires more care. Because this exception will arise (and be ignored) so frequently in floating-point computations, it is impractical to have a hardware trap executed on every occurrence. If the Inexact-Result exception is to be handled by a hardware trap and system software, then that trap should be maskable. In one possible implementation:

(1) The trap would be masked off until . . .

(2) enabled by the library routine invoked by the user to clear the Inexact-Result sticky flag or to enable the user trap, and . . .

(3) on the first occurrence of a rounding error, the hardware trap would set the sticky flag. The user trap would be invoked if enabled; otherwise the system software would disable the hardware trap and resume execution, leaving the sticky flag as an indication of a rounding error.

A possible hardware trap on denormalized operand was mentioned at the end of the last section. A system implementing the Normalizing mode of computation would have software test the Warning/Normalizing mode bit and normalize the denormalized operand if necessary, handling the details of extended exponent range required to represent the operand as normalized.

2.0 Specifications for a conforming implementation of standard arithmetic

2.1 Floating-point formats. *Single, double,* and *quad* are the basic floating-point formats. A standard system shall provide *single* only, both *single* and *double,* or all three basic formats. In addition, either of the first two systems above may provide the *extended* format corresponding to the wider basic format supported. The formats are described in Tables 1 and 2.

2.2 Data types. This standard defines the following floating-point data types: normalized numbers, denormalized numbers, unnormalized numbers (available only in *extended* and *quad*), the normal zeros (± 0), $\pm\infty$, and the NaNs. They are described in detail in Tables 1 and 2.

A standard system must produce denormalized numbers as the default response to Underflow; un-

normalized numbers are their descendants in *extended* or *quad*. A system may optionally allow users to normalize all denormalized numbers when they appear as input operands in arithmetic operations. This shall be called Normalizing mode in contrast to the default, Warning mode. The choice of Normalizing/Warning modes shall be made via a single bit accessible to users.

Signed ∞'s are produced as the default response to Division-by-Zero and certain Overflows. Systems shall provide ∞ arithmetic as specified. Users must be able to choose, via a single-mode bit, whether ±∞ will be interpreted in the Affine or Projective closures of the real numbers. The sign of ∞ is respected in Affine mode and ignored in Projective, the default.

NaNs are symbols which may or may not have a numeric connotation. Nontrapping NaNs are intended to propagate diagnostic information through subsequent arithmetic operations without triggering further exceptions. Trapping NaNs, on the other hand, shall precipitate the Invalid-Operation exception when picked up as operands for an arithmetic operation. Systems shall support both types of NaNs. In the event that two Nontrapping NaNs occur as operands in an arithmetic operation, the result is one of the operands, determined by a system-dependent precedence rule.

2.3 Arithmetic operations. An implementation of this standard must at least provide:

(A) ADD, SUBTRACT, MULTIPLY, DIVIDE, and REMAINDER for any two operands of the same format, for each supported format, with the destination having no less exponent range than the operands.

(B) COMPARE and MOVE for operands of any, perhaps different, supported formats.

(C) ROUND-TO-INTEGER and SQUARE ROOT for operands of all supported formats, with the result having no less exponent range than the input operands. In the former operation, rounding shall be to the nearest integer or by truncation toward zero, at the user's option.

(D) Conversions between floating-point integers in all supported formats and binary integers in the host processor.

(E) Binary-decimal conversions to and from all supported basic formats. Section 2.21 describes one possible implementation.

2.4 Exceptions. One or more of five exceptional conditions may arise during an arithmetic operation: Overflow, Underflow, Division-by-Zero, Invalid-Operation, and Inexact-Result.

The default response to an exception is to deliver a specified result and proceed, though a system may offer traps to user software for any of the exceptions. These traps shall be enabled via bits accessible to programmers.

A system providing a trap on an exceptional condition should give sufficient information to allow correction of the fault and allow processing to continue at the point of the error or elsewhere, at the option of the trap handler. The correct result may be encoded in the destination's format (or even in the destination) or in a heap pointed to by a NaN. On the other hand, if no numeric result can be given, the opcode and aberrant operands must be provided; the trap handler should be able to return a result to be delivered to the destination.

Associated with each of the exceptions is a sticky flag which shall be set on the occurrence of the corresponding exception when no trap is to be taken. The flags may be sensed and changed by user programs, and remain set until cleared by the user.

2.5 Specifications for the arithmetic operations. For definiteness the algorithms below specify one conforming implementation. *Single, double,* and *double-extended* formats are implemented; the exception flags are set on every occurrence of the corresponding exception; the *extended* exponent is biased by 16383. There are many alternative conforming implementations. Those arithmetic operations, except Decimal to Binary conversion, which deliver floating-point results rather than strings or binary integers are broken into three steps:

(0) If either operand is a Trapping NaN, then signal Invalid-Operation and proceed to Step 2. Otherwise, if the Normalize bit is set, then normalize any denormalized operands.

(1) Compute preliminary result Z and, if numeric, round it to the required precision and check for Invalid/Over/Underflow violations. This step is peculiar to the specific operation.

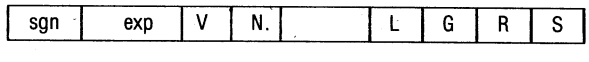
(2) Set exception flags, invoke the trap handler if required, and deliver the result Z to its destination. The second step is the same for all operations except REMAINDER and MOVE; the minor differences are noted.

The following table is used in the specification of Step 1 of the operations with two input operands. It singles out the cases involving special operands.

		Y			
	X op Y	±0	W	±∞	NaN
X	±0	a	b	c	Y
	W	d	e	f	Y
	±∞	g	h	i	Y
	NaN	X	X	X	M

W is any finite number, possibly unnormalized but not normal zero. While X and Y refer to the input operands, the entry M indicates that the system's precedence rule is to be applied to the two Nontrapping NaNs.

Preliminary numeric results may be viewed as:



where V is the overflow bit for the significant digit field, N and L are the most and least significant bits,

G and R are the two bits beyond L, and S, the sticky bit, is the logical OR of all bits thereafter.

2.6 ADD/SUBTRACT. For subtraction, X−Y is defined as X+(−Y).

a: Z is +0 in rounding modes RN, RZ, RP, or if both operands are +0; Z is −0 in mode RM or if both operands are −0.

c,f: Z=Y.

g,h: Z=X.

b,d,e: (Note that in cases *b* and *d*, a narrow rounding precision may cause the result to differ from the nonzero input operand.) Compute:

(1) Align the binary points of X and Y by unnormalizing the operand with the smaller exponent until the exponents are equal. Note whether either of the resulting significands is normalized for (3) below. Add the operands.

(2) Addition of magnitudes: If V=1, then right-shift one bit and increment exponent. During the shift R is ORed into S.

(3) Subtraction of magnitudes:

(a) If all bits of the unrounded significant digit field are zero: Set the sign to "+" in rounding modes RN, RZ, RP, and set the sign to "−" in mode RM. Then, if either operand was normalized after binary point alignment in (1), the exponent is set to its minimum value, i.e., the result is true zero.

(b) Otherwise: If, after binary point alignment in (1), neither operand was normalized, then skip to (4). Otherwise, normalize the result, i.e., left-shift the significand while decrementing the exponent until N=1. S need not participate in the left shifts; zero or S may be shifted into R from the right.

(4) Check Underflow, round, and check Invalid and Overflow.

i: In Affine mode $(+\infty)+(+\infty)\rightarrow(+\infty)$ and $(-\infty)+(-\infty)\rightarrow(-\infty)$. In Affine mode on $(+\infty)+(-\infty)$ and $(-\infty)+(+\infty)$, and in all cases in the Projective mode, signal Invalid-Operation, and if a result must be delivered, set Z to NaN.

2.7 MULTIPLY.

a,b,d: Z=0 with sign.

c,g: Signal Invalid-Operation. If a result must be delivered, set Z to NaN.

e: If either operand is an unnormal zero, proceed as in *c*; otherwise, compute:

(1) Generate sign and exponent according to convention. Multiply the significands.

(2) If V=1 then right-shift the significand one bit and increment the exponent.

(3) Check Underflow, round, and check Invalid and Overflow.

f,h,i: Z=∞ with sign equal to the Exclusive-Or of the operands' signs.

2.8 DIVIDE.

a,i: Signal Invalid-Operation and if a result must be delivered, then set Z to NaN.

b,c,f: Z=0 with sign. Exception: if X is an unnormal zero, proceed as in *a*.

d: Z=∞ with sign. Signal Division-by-Zero. Exception: if X is an unnormal zero, proceed as in *a*.

e: If Y is unnormalized, proceed as in *a*; otherwise, compute:

(1) Generate sign and exponent according to convention. Divide the significands.

(2) If N=0, then left-shift significand one bit and decrement exponent. S need not participate in the left shift; a zero or S may be shifted into R from the right.

(3) Check Underflow, round, and check Invalid and Overflow.

g,h: Z=∞ with sign.

2.9 REMAINDER. Form the preliminary result Z = remainder when X is divided by Y, with integer quotient Q. Q does not participate in Step 2 of the operation unless an exception is raised there, in which case if Z is set to NaN, then Q is assigned the same value. The sign of Q is the Exclusive-Or of the input operands' signs. The standard does not require the quotient Q.

a,d,g,h,i: Signal Invalid-Operation. If results must be delivered, then set Z and Q to NaN.

b,c: If Y is unnormal zero, proceed as in *a*; otherwise Z=X and Q = 0.

e: If Y is unnormalized, proceed as in *a*. Otherwise, normalize X and compute:

(1) Set Q to the integer nearest X/Y computed to as many bits as necessary to round correctly; if X/Y lies halfway between two integers, set Q to the even one. If Q contains more significant bits than its intended destination (the number may be great if X>>Y), then discard the excessive high-order bits.

(2) Set Z to the remainder, X−(Q*Y). Normalize Z, check Underflow, round, and check Invalid and Overflow. There is no rounding error if the destination precision is no narrower than X's and Y's.

f: Q=0 and Z=X.

2.10 ROUND-TO-INTEGER. Set Z to X if X is ±0, ±∞, or NaN; otherwise, compute Z: If X's exponent is so large that it has no (zero or nonzero) significant

fraction bits, then set Z to X; else:

(1) Right-shift X's significand while incrementing the exponent until no bits of the fractional part of X lie within the rounding precision in effect.

(2) Round Z. The user must have the option of rounding by truncation as well as to the nearest integer.

(3) If all of the significant bits of Z are 0, then set Z to normal zero with the sign of Z; otherwise, normalize Z. S, which was set to zero after rounding in (2), need not participate in the left shifts of normalization; zero or S is shifted into R from the right.

2.11 SQUARE ROOT. $Z=\sqrt{X}$. If X is ± 0 or NaN, then set Z to X. If X is unnormalized or $-\infty$, then signal Invalid-Operation and if a result must be delivered, set Z to NaN. If X is $+\infty$, then in Affine mode set Z to X and in Projective mode proceed as for $-\infty$.

If X is positive, finite, and normalized, compute $Z=\sqrt{X}$ to the number of bits required to get a correctly rounded result, and round Z. Only two bits of Z beyond its rounding precision are required, if that precision is no narrower than the precision of X.

If X is negative, finite, and normalized, signal Invalid-Operation. If a result must be delivered, set Z to NaN.

2.12 MOVE. MOVE X → Z (convert between *different* floating-point formats) is an operation whose destination may have shorter range and precision than its source operand, in which case it performs an arithmetic operation. If X is ± 0, $\pm\infty$, or NaN, set Z to X. Otherwise, check X for Underflow, round to the precision of the destination, and check for Invalid and Overflow.

On Over/Underflow with the corresponding trap enabled, the exponent may be more than a factor of 2 (i.e., one bit) beyond the range of the destination, so the exponent wrap-around scheme will not work. One way to cope is to deliver to the trap handler the result in the format of the source, or in the widest format supported, but rounded to the precision of the destination. Another way involves a heap onto which is put the rounded value whose exponent lies beyond the range of the intended destination; into the destination would go a NaN pointing to that value in the heap.

2.13 Detection of Underflow. If the exponent of the nonzero preliminary result underflows the intended destination, then signal Underflow and, if the Underflow trap is disabled, denormalize it as follows. Shift the significant digit field right while incrementing the exponent until it reaches its most negative allowable value. During each right-shift the R bit is ORed in to the S bit, itself not shifted. If the trap is enabled then, except for the MOVE operation, the exponent is wrapped around as described under Bias Adjust (Section 2.16).

Another instance of Underflow, tested after rounding, is a normal zero *extended* or *quad* product or quotient of operands neither of which is normal zero. This special case is precluded by the redundant exponent scheme discussed in Section 1.12.

2.14 Rounding. Four rounding modes are described by the standard:

RN — Round to Nearest
RZ — Round toward Zero
RM — Round toward $-\infty$
RP — Round toward $+\infty$.

An implementation of the standard may support either RN only, with RZ for Round to Integer, or all four rounding modes. RN shall be the default mode for all arithmetic operations. The rounding mode may be specified by, say, preset mode bits, rounding mode options in each instruction, or rounding instructions which can follow the operation whose result is re-rounded, but not double-rounded.

The preliminary result Z, to be rounded, may be viewed as in Section 2.5. S, the sticky bit, assures a result rounded as though first computed to infinite precision. From Z determine Z1 and Z2, the numbers representable in the desired rounding precision that

most closely bracket Z. Since Overflow is not checked until after rounding, the exponent of Z1 or Z2 or both may be overflowed.

If Z1=Z=Z2, there is no rounding error and RN(Z)=RZ(Z)=RP(Z)=RM(Z)=Z. Otherwise, signal Inexact-Result, and

RN(Z)=the nearer of Z1 and Z2 to Z; in case of a tie choose the one of Z1 and Z2 whose least significant bit is 0.
RZ(Z) = the smaller of Z1 and Z2 in magnitude.
RM(Z)= Z1.
RP(Z) = Z2.

When a system supports an *extended* format, it must provide users with the option of rounding to a shorter basic precision a result intended for a wider *extended* destination. Also, when all operands in an operation are of the same format, it shall be possible to round the result to the precision of that format. The specification of that option will require at most two bits of information: one enables precision control; one specifies whether rounding to *single* or *double* precision, effective only when precision control is enabled.

2.15 Detection of Invalid and Overflow. If an unnormalized, but not denormalized, number is destined for a *single* or *double* destination, the Invalid-Operation exception arises. Otherwise. . . .

If Z's exponent overflows the intended destination, then signal Overflow and, if the corresponding trap is enabled, adjust the exponent bias as specified under Bias Adjust (Section 2.16).

On Overflow with the trap disabled, signal Inexact-Result. Then set Z to ∞ with the sign of Z if the rounding mode is RN, RZ, RP and Z is positive, or RM and Z is negative. Otherwise, if Z is normalized, set Z to the largest normalized number representable in the destination field, with the sign of Z; and if Z is not normalized, simply set Z's exponent to that of the format's largest normalized number.

2.16 Bias Adjust. On Over/Underflow, with the corresponding trap enabled, the exponent of a rounded result Z is wrapped around into the required range of the destination. Compute A = 192 in *single,* 1536 in *double,* 24576 in *quad,* and $3*2^{n-2}$ in *extended,* where n is the number of bits in the exponent. On Overflow subtract A from Z's exponent; on Underflow add A to Z's exponent.

This scheme works only when the Over/Underflowed exponent exceeds its destination's range by a factor no larger than 2. The only exception in this implementation is discussed under MOVE (Section 2.12).

2.17 Step 2 of arithmetic operations. Preliminary result Z was developed in Step 1.

(1) In modes RP and RM, "undo" any Over/Underflow signals whose traps were enabled.
(2) If the Invalid-Operation exception was signaled, produce a diagnostic Nontrapping NaN as the preliminary result Z.
(3) Set the sticky exception flags corresponding to the exceptions signaled. Trap if any exception has been signaled whose corresponding trap is enabled, allowing Z to be modified before delivery to the destination.
(4) Deliver Z to its destination.

2.18 FLOATING-TO-INTEGER. This instruction converts a floating-point number X into a binary integer of the host processor. If X is a NaN or ∞, then leave the destination unchanged and set the Invalid-Operation bit, trapping if the corresponding trap is enabled.

For finite X, replace X by ROUND-TO-INTEGER(X). Convert X to an integer in the desired format and write the result into the destination. If X overflows the destination field, then truncate excessive high-order bits and signal Integer-Overflow in the host processor, if it recognizes such an exception; otherwise, set the Invalid-Operation sticky flag and trap if enabled.

2.19 INTEGER-TO-FLOATING. Map the binary integer X in the host processor into a floating-point integer. If X cannot be represented exactly, then round as described in Rounding and set the Inexact-Result bit, trapping if the corresponding trap is enabled.

2.20 COMPARE. A floating-point comparison can have precisely one of four possible results (condition codes): <, =, >, and unordered. When the result is reported as the affirmation or negation of a predicate, the following implications determine that response:

= affirms ⩽, =, and ⩾, and denies <, >, and unordered.
< affirms < and ⩽ and denies =, ⩾, >, and unordered.
> affirms ⩾ and > and denies <, ⩽, =, and unordered.
unordered affirms unordered and denies <, ⩽, =, ⩾, and >.

When two values that are unordered are compared via the predicates <, ⩽, ⩾, >, or their negations, then, in addition to the response specified, the Invalid-Operation flag is set and the trap invoked if enabled.

The following table specifies the compare operation. Unnormalized (and denormalized) operands are treated as though first normalized.

X vs Y	−∞ Affine	Finite	+∞ Affine	∞ Projective	NaN
−∞ Affine	=	<	<	N/A	a
Finite	>	b	<	a	a
+∞ Affine	>	>	=	N/A	a
∞ Projective	N/A	a	N/A	=	a
NaN	a	a	a	a	a

a: unordered.
b: The result is based on the result of X−Y. The subtraction may not have to be carried out completely, and the possible Underflow and Inexact-Result exceptions are suppressed.

2.21 Radix conversion. A system must provide standard conversion to and from its basic formats. The specifications are a compromise designed to ensure that conversions are uniform and in error by less than one unit in the last place delivered, at a nearly minimal cost. The scheme below meets the requirements for *single* and *double*.

The particular decimal character code and format are unspecified. The decimal field widths are:

single: up to 2-digit exponent and up to 9 significant digits.

double: up to 3-digit exponent and up to 17 significant digits, with the option of using up to 19 digits in decimal-to-binary conversion.

Two functions perform conversions between binary floating-point integers and character strings consisting of a sign followed by one or more decimal digits. BINSTR converts a binary floating-point integer X, rounded to the nearest integer, to a signed decimal string. STRBIN converts a signed decimal string with at most 9 digits in *single*, and 19 in *double*, to a binary floating-point number X whose value is that of the decimal integer the string represents.

The function $\log_{10}$ is required and may be computed from the formula

$$\log_{10}(X) = \log_2(X) * \log_{10}(2).$$

It need be computed only to the nearest integer for this calculation. $\text{Log}_2(X)$ may be approximated by X's unbiased exponent. Within the conversion process, arithmetic must be done with at least 32 significant bits for *single* and 64 bits for *double*.

Powers of 10 not exactly calculable in the stated precision shall be procured from tables. The following tables require minimal storage:

(A) Systems with *single* precision only: 10^{13} can be represented exactly with 32 significant bits. To cover the range up to 10^{38}, a table with the single entry 10^{26} suffices.
(B) Systems with both *single* and *double* precisions only: 10^{27} can be represented exactly with 64 significant bits. To cover the range up to 10^{308}, a table of 10^{54}, 10^{108}, and 10^{216} suffices.

Binary-floating-to-Decimal-floating. Given binary floating-point number X and integer k with $1 \leqslant k \leqslant 9$ for *single* precision and $1 \leqslant k \leqslant 17$ for *double* precision, compute signed decimal strings I and E such that I has k significant digits and, interpreting I and E as the integers they represent,

$$X = I * 10^{E+1-k} = \text{sd.ddddddd} * 10^{E}$$

where s is the sign of X and the d's are the k decimal digits of I.

(1) Special cases: If X is $+\infty$, $-\infty$, or NaN, deliver a nondecimal string, for example, ++, −−, .., respectively. If X is zero, then return +0 or −0 as appropriate. Otherwise . . .
(2) Set X to its absolute value, saving its sign.
(3) If X is normalized, compute $U = \log_{10}(X)$; otherwise let $U = \log_{10}(\text{smallest normalized number})$.
(4) Compute $V = U + 1 - k$, rounded to an integer in mode RZ.
(5) Compute $W = X/10^{V}$, rounded to an integer in mode RN.
(6) Adjust W:

If $W \geqslant 10^{k}+1$, then increment V and go to (5).
If $W = 10^{k}$, then increment V, divide W by 10 (exactly), and go to (7).
If $W \leqslant 10^{k-1}-1$ and X was normalized in (3), then decrement V and go to (5).

(7) Return I=BINSTR(W with sign of X) and E=BINSTR(V).

Decimal-floating-to-Binary-floating: The decimal floating-point number X has the form X=sddddd.DDDDDDD * 10^{E}, where leading zeros are not counted as significant digits. The following are given:

(A) signed decimal string E
(B) signed decimal string I= sdddddDDDDDDD
(C) integer P indicating how many digits of I are to the right of the decimal point so that X can be written

$$X = I * 10^{-P} * 10^{E}.$$

(1) Compute U=STRBIN(I).
(2) Compute W=STRBIN(E).
(3) Compute result $X = U * 10^{W-P}$. ■

Jerome T. Coonen is a graduate student of mathematics at the University of California at Berkeley. Since spring 1978 he has worked actively with the IEEE subcommittee on a floating-point standard for minicomputers and microprocessors. His research interests include computer arithmetic, numerical analysis, and differential equations. Coonen is a candidate for the PhD degree; he received the BS and MS degrees from the University of Illinois at Urbana.