reprinted from
IEEE MICRO

A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic

W. J. Cody*
Argonne National Laboratory

J. T. Coonen†
Apple Computer

D. M. Gay
AT&T Bell Laboratories

K. Hanson
Apple Computer

D. Hough
Apple Computer

W. Kahan**
University of California, Berkeley

R. Karpinski
University of California, San Francisco

J. Palmer
Intel

F. N. Ris
IBM

D. Stevenson
Zilog, Inc.

Besides making the proposed IEEE 854 standard available for comment, this article explains how to overcome some of its implementation problems.

A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic

W. J. Cody*
Argonne National Laboratory

W. Kahan**
University of California, Berkeley

J. T. Coonen†
Apple Computer

R. Karpinski
University of California, San Francisco

D. M. Gay
AT&T Bell Laboratories

J. Palmer
Intel

K. Hanson
Apple Computer

F. N. Ris
IBM

D. Hough
Apple Computer

D. Stevenson
Zilog, Inc.

The Microprocessor Standards Committee of the IEEE Computer Society sponsors two groups drafting proposed standards for floating-point arithmetic. The first, Task P754, reported Draft 10.0 of a Proposed Standard for Binary Floating-point Arithmetic out of committee in December, 1982.[1] That document is now a de facto standard[2] and is progressing slowly through the approval process within the IEEE Computer Society.

In August 1983, the second group, Task P854, completed Draft 1.0 of a Proposed Radix- and Word-length-

*Work supported by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the US Department of Energy under Contract W-31-109-Eng-38.

†Work supported in part at the University of California, Berkeley, by an IBM predoctoral fellowship.

**Work supported in part by research grants from the Office of Naval Research and the Department of Energy.

Note: In addition to the above, the following authors' employers supported this effort: Apple Computer, AT&T Bell Laboratories, IBM, Intel, UC San Francisco, and Zilog.

0272-1732/84/0800-0086$01.00 © 1984 IEEE

independent Standard for Floating-point Arithmetic that generalizes and is upward compatible with the IEEE Proposed Standard for Binary Floating-point Arithmetic. This article places their contents before the public for the first time.

Text drawn from the P854 draft is set off from surrounding expository material by indentation from both margins. The article also includes material that describes how decisions were reached in preparing the P854 draft and explains how to overcome some of the implementation problems.

We are publishing this material to invite comment on the work of P854 prior to its submission to the IEEE Standards Board for adoption as an IEEE standard. We intend that such submission follow this publication by six months. We ask that readers of this article direct any comments or criticisms, in writing, to either of the following individuals:

W. J. Cody
MCS-221/C223
Argonne National Laboratory
Argonne, IL 60439

R. Karpinski
U-76
University of California
San Francisco, CA 94143

In what follows, we refer to the P754 draft as the "draft binary standard," and the P854 draft as either the "draft generalized standard" or simply the "draft."

Many individuals helped prepare these drafts. Each contributed as an individual; no endorsement by an employer is implied. The authors of this article were the voting members of P854 when Draft 1.0 was adopted.

1. Scope

This draft has the same scope as the draft binary standard.

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

1.3. Exclusions. This standard does not specify

(1) Formats for internal storage of floating-point numbers,
(2) Formats of decimal strings and integers,
(3) Interpretation of the sign and significand fields of NaNs, or
(4) Conversion between extended precision (§3.2) floating-point numbers and decimal strings.

2. Definitions

The following terms are defined for purposes of the draft generalized standard:

Destination. Every unary or binary operation delivers its result to a destination, either explicitly designated by the user or implicitly supplied by the system (e.g., intermediate results in subexpressions or arguments for procedures). Some languages place the results of intermediate calculations in destinations beyond the user's control. Nonetheless, this standard defines the result of an operation in terms of that destination's precision as well as the operand's values.

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

This definition implies that the radix used for the representation of floating-point numbers is the same as the radix used for scaling. For example, a decimal significand must be scaled by a power of 10. Note, however, that the exponent is an integer, and it need not be implemented as a string of base-b digits.

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

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

Mode. A variable that a user may set, sense, save, and restore to control the execution of subsequent arithmetic operations. The default mode is the mode that a program can assume to be in effect unless an explicitly contrary statement is included in either the program or its specification.

The following mode shall be implemented:

(1) Rounding, to control the direction of rounding errors; and, in certain implementations,
(2) Rounding precision, to shorten the precision of results.

The implementor may, at his option, implement the following modes:

(3) 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.

A NaN is similar in some respects to the "indefinite" on CDC 7600 and Cyber systems, and to the "reserved operand" in DEC PDP-11 and Vax.

Normal number. A nonzero number that is finite and not subnormal.

Radix. The base for the representation of floating-point numbers.

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

Shall and should. In this standard the use of the word "shall" signifies that which is obligatory in any conforming implementation; 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.

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

In the familiar "scientific notation," numbers are expressed in a form like -1.2345×10^{-67}. Here are the first "$-$" is the algebraic sign, "1.2345" is the significand, ".2345" is the fraction, "10" is the radix, and "-67" is the exponent.

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

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

For example, calculators whose lowest exponent is -99 would admit subnormal numbers if they permitted numbers like 0.0123×10^{-99}. Subnormal numbers used as arithmetic operands do not behave exceptionally, but subnormal results, sometimes accompanied by a signal, serve to make underflow gradual. Subnormal numbers correspond to the "denormalized numbers" in the draft binary standard.

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.

The draft is deliberately vague about the meaning of the word "user" because it could refer to a human seeking results, an applications program exploiting the arithmetic, or a compiler generating code for the arithmetic system.

3. Precisions

The main characteristic of the draft binary standard is its attention to detail. Because it is specific to 32-bit words, it specifies even the bit-patterns representing floating-point quantities in the basic formats. Similar detail is found in other discussions, such as binary↔decimal string conversion and the adjustment of overflowed exponents.

While the draft generalized standard also deals with detail, it is best characterized by its parameterization of the arithmetic. Because the draft generalized standard applies to machines of arbitrary word length and to radices other than binary, the *representation* of floating-point quantities is not easily specified. Instead, the draft derives the set of *values* that may be taken by floating-point quantities in each of the supported *precisions* from a set of four integer parameters. These parameters turn out to be fundamental to the whole arithmetic system. They permit a level of detail comparable to that in the draft binary standard while preserving essential abstractness in discussing such things as floating-point ↔ decimal string conversion and the adjustment of overflowed exponents.

This draft is a prescription for arithmetic which, given a choice of the four integer parameters, defines the representable values and the results of all operations precisely. In this respect, the draft differs from descriptive models of arithmetic, like Brown's,[3] which derive some information from similar parameters but cannot say exactly what arithmetic results will be generated. We will see later that proper specification of the four integer parameters produces essentially the arithmetic in the draft binary standard, an important verification of the desired compatibility of the two proposals.

Section 3 of the draft generalized standard discusses the precisions and their parameterization:

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

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

b–the radix,

p–the number of base-b digits in the significand,

E_{max}–the maximum exponent, and

E_{min}–the minimum exponent.

The parameters are subject to the following constraints:

b shall be either 2 or 10 and shall be the same for all supported precisions,

$(E_{max} - E_{min})/p$ shall exceed 5 and should exceed 10, and $b^{p-1} \geq 10^5$.

The balance between the overflow threshold ($b^{E_{max}+1}$) and

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

Because overflow is so much more serious a disaster than gradual underflow, this constraint moves the overflow threshold slightly further away from 1 (at the cost of bringing the underflow threshold slightly closer). The intent is to ensure that normal values can be reciprocated without awkward exception; e.g., the inverse of the smallest positive normal value (the underflow threshold) should not overflow, and the inverses of the largest finite values (almost the overflow threshold) should suffer minimal loss of significance.

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

Numbers of the form

$(-1)^s b^E (d_0.d_1 d_2 \ldots d_{p-1})$, where

s is an algebraic sign,

E is any integer between E_{min} and E_{max}, inclusive, and

each d_i is a base-b digit ($0 \leq d_i \leq b-1$);

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

At least one signaling NaN; and

At least one quiet NaN.

The algebraic sign provides additional information about any variable that has the value zero. Although all precisions have distinct representations for $+0$, -0, $+\infty$, and $-\infty$, the signs are significant in some circumstances, such as division by zero, and not in others. In this standard, 0 and ∞ are written without a sign when the sign does not matter. An implementation may find it helpful to provide additional information about a variable which is NaN through an algebraic sign, but this standard does not interpret such extensions.

Note that §3.1 does not mention the representation of numbers, only their values:

The foregoing description enumerates some values redundantly, e.g.,

$$b^0(1 \cdot 0) = b^1(0 \cdot 1) = b^2(0 \cdot 01) = \ldots$$

but the standard does not distinguish them.

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

Although the draft generalized standard is to be radix-independent, the committee could find no valid technical argument for allowing radices other than 2 or 10. Radices that are higher powers of 2 or 10 have disagreeable properties. Nevertheless, the standard is drafted so that this particular limitation is expressed only in §3.1 and §5.6; we believe that were this limitation removed, only §5.6 would have to be revised.

Constraints on the parameters were not imposed capriciously. The lower limit of $5p$ for $E_{max} - E_{min}$ accommodates instrumentation equipment where large exponent range is not important, but some form of standardized floating-point arithmetic is. The inequality $b^{p-1} \geq 10^5$ permits implementation of 6-significant-decimal arithmetic in 32-bit words without resorting to digit encoding more complicated than BCD.

The subnormal numbers in this scheme are those nonzero numbers with magnitudes less than $b^{E_{min}}$. All bigger finite numbers are normal. The natural way in which zero and subnormal numbers occur is a consequence of focusing on the values of the numbers rather than on their representation. Representation of the other special operands, infinities, and NaNs, requires that something special be done.

Examples:

(1) In the draft binary standard, the parameters for single precision are $b=2$, $p=24$, $E_{max}=127$, and $E_{min}=-126$. All of the special operands are accommodated by reserving extreme values of the exponent beyond E_{max} and E_{min}.

(2) The parameters for many hand calculators are $b=10$, $p=10$, $E_{max}=99$, and $E_{min}=-99$. However, none of the special operands are accommodated.

With the establishment of the parameterization, it is possible to describe the various supported precisions and the relations between them. Note that the terms "single precision" and "double precision" may be known by other names in existing programming languages. For example, "single precision" is called "real" in Pascal and Fortran.

3.2 Basic Precisions

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

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

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

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

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

$$b^{p_d} \geq 10 b^{2p_s},$$
$$E_{max_d} \geq 8 E_{max_s} + 7,$$
$$E_{min_d} \leq 8 E_{min_s}.$$

The condition $b^{p_d} \geq 10b^{2p_s}$ provides that double precision be at least one decimal-digit wider than twice single precision to protect the formation of inner products (vector dot products) of single-precision data using double-precision arithmetic. The consequence when $b=2$ is that at least four

additional bits more than twice single precision must be provided in double precision. The draft binary standard provides five such bits. The constraints on E_{max_d} and E_{min_d} ensure that products of up to eight factors (or powers up to the eighth power) of normal single-precision values will neither overflow nor underflow.

In addition to inner products, certain other commonly occurring computations can be carried out more accurately and naturally when the arithmetic used provides extra exponent range and precision over that found in the data. We will see in a moment that exponentiation and conversion between floating-point representations and decimal strings are computations of this type (there are many others).

If implemented, double-precision arithmetic provides the desired support for single precision. But what supports single precision when double precision is not implemented, and what supports double precision? The draft standard provides extended precisions for these purposes. Although extended precisions afford almost full protection in many important computations, they provide only partial protection for inner products. In a first implementation of a system conforming to this draft standard, supporting the few additional digits in an extended precision may be easier, substantially more economical, and almost as beneficial as supporting the next higher basic precision with its more than doubling of the digits. The issues to be considered when selecting between implementation of an extended precision and implementation of the next higher basic precision are discussed further by Coonen[4] and Kahan.[5]

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

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

Parameters for single-extended shall satisfy

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

and

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

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

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

For $b = 2$, the condition

$$p_{se} \geq p_s + \lceil \log_2(E_{max_s} - E_{min_s}) \rceil$$

in §3.3 states that at least as many bits as are required for representing a single-precision number (significand plus exponent) shall be used for representing the single-extended significand. Wider would be better, but might be uneconomical because of architectural constraints such as bus widths.

Ideally p_{se} should be large enough to protect against error in the computation of y^x, even when the result is λ, the largest single precision floating-point quantity, or σ, the smallest normal positive floating-point quantity. Assuming that

$$\lambda = b^{E_{max}}(b - b^{-p}) \geq 1/\sigma = b^{-E_{min}},$$

we want to be able to compute $ln(\lambda)$ with absolute error less than b^{-p_s} in single-extended precision. (If $\lambda < 1/\sigma$, replace $E_{max} + 1$ with $-E_{min}$ and λ with $1/\sigma$ in the analysis.) Let

$$y^x = exp[x\,ln(y)] \approx \lambda = b^{E_{max}}(b - b^{-p_s}).$$

Then the computed argument $x\,ln(y)$ delivered to the exponential routine is effectively

$$x\,ln(y)(1 \pm \epsilon)^3,$$

where $\epsilon = b^{1-p_{se}}$. Here, the first two rounding errors come from the computation of $ln(y)$ and the product $x\,ln(y)$, and the third accounts for the error to be made in the argument reduction in the exp routine. Computationally, to first order terms in the errors,

$$\begin{aligned} exp[x\,ln(y)] &\rightarrow exp[x\,ln(y)(1 \pm 3\epsilon)](1 + \eta), \\ &\approx (1 + \eta)\lambda\, exp[\pm ln(\lambda)3\epsilon], \\ &\approx (1 + \eta)\lambda\,[1 \pm ln(\lambda)3\epsilon], \end{aligned}$$

where η is a single-precision rounding error. Now the desire is that

$$\pm 3\epsilon ln(\lambda) = 3b^{1-p_{se}} ln[b^{E_{max}+1}] < b^{-p_s}.$$

This simplifies to

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

Because this ideal may not be practical, the condition on p_{se} is recommended but not required. For example, it may not be easy to allocate more space for the p_{se} digits than that space normally allocated to the representation of a single-precision floating-point number, i.e., the space normally used to represent two signs, the p_s digits of the significand and the largest exponent, E_{max_s}, say. In traditional floating-point architectures this is exactly one or two word lengths.

Example:

(1) In the draft binary standard, the parameters are $E_{max_s} = 127$, $E_{max_{se}} \geq 1023 = 8E_{max_s} + 7$, $E_{min_s} = -126$, $E_{min_{se}} \leq -1022 < 8E_{min_s}$, and $p_{se} \geq 32 > p_s + 7$. This extended format falls 1 bit short of what it ideally should be to support y^x, owing to practical considerations of wordlength.

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

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

One of our principal objectives is to facilitate movement of software among machines that conform to the draft standard. So that single-precision software supported by either single-extended or double-precision arithmetic can be insensitive to whichever of the two is actually used, both must treat underflow in the same way. That is why single extended includes subnormals.

3.4 Combinations of precisions. All implementations conforming to this standard shall support single precision. Implementations should support the extended precision corresponding to the widest basic precision supported, and need not support any other extended precision.

Section 3.4 recognizes that double precision satisfies all of the requirements for single extended. It would be wasteful to provide both of these precisions unless complete upward compatibility and speed were important issues.

Why did the authors of the draft standards not follow Coonen[4] in specifying, besides single and double, a quadruple-precision format? Presumably it is obvious that a quadruple format should bear the same relation to double as double bears to single. Less obvious is whether a family of intermediate precisions positioned between single and double, and between double and quadruple, would be worth implementing and supporting with appropriate language and library facilities. Such a question has been considered by T. E. Hull,[6] but the authors have chosen to say nothing more about that question at this time.

4. Rounding

Rounding is one of the least understood and often one of the most badly designed features in traditional arithmetic systems. The draft standard specifies four different rounding modes in §4. These include what we intuitively think of as rounding, but done carefully; what is frequently called truncation; and two directed rounding modes of use when implementing interval arithmetic. The rounding modes are user-selectable and apply to all pertinent operations, described in §5, once selected.

Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit the destination's precision while signaling the inexact exception (§7.5). Except for conversion between floating-point numbers and decimal strings (whose weaker conditions are specified in §5.6), every operation specified in §5 shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result according to one of the modes in this section.

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

4.1. Round to nearest. An implementation of this standard shall provide round to nearest as the default rounding mode. In this mode the representable value nearest to the infinitely precise result shall be delivered; if the two nearest representable values are equally near, the one with its least significant digit even shall be delivered. However, an infinitely precise result with magnitude at least $b^{E_{max}}(b - \frac{1}{2}b^{1-p})$ shall round to ∞ with no change in sign; here E_{max} and p are determined by the destination precision (§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 precision's value (possibly $+\infty$) closest to and no less than the infinitely precise result. When rounding toward $-\infty$, the result shall be the precision's value (possibly $-\infty$) closest to and no greater than the infinitely precise result. When rounding toward 0, the result shall be the precision's value closest to and no greater in magnitude than the infinitely precise result.

Rounding toward zero provides a mode of arithmetic capable of nearly mimicking features of certain widely used machines and languages. First, rounding toward zero prevents infinity from being created by overflow; this approximates the situation on machines that, lacking infinity, replace overflows by the biggest available finite magnitude with the appropriate sign. Second, rounding toward zero is obligatory in Fortran's three conversions from floating point to integer:

```
I = X                          ,
  ...INT(X)... or ...IFIX(X)...,
  ...AINT(X)...                .
```

The draft standard's default rounding mode in conversion to integer nearly matches a construction common in Fortran, namely,

```
... INT(X+0.5) ... or ... AINT(X+0.5) ...,
```

except when X is already a nonnegative even integer. Intel's Fortran 86[7] assigns the name RINT(X) to the draft binary standard's default rounding to nearest integer.

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

Control of rounding precision is intended to allow systems whose destinations are always double or extended to mimic, in the absence of overflow/underflow, the precisions of systems with single and double destinations. Proper mimicking requires that such machines provide operations to combine single operands, returning a single result with only one rounding. An implementation should not provide operations that combine double or extended operands to produce a single result, nor operations that combine double extended operands to produce a double result, with just one rounding, because using such operations would violate §5.1, below.

The precision to which anonymous variables (constants and subexpressions) shall be rounded is determined or left undetermined in a confusing way in most programming languages. For example, Fortran assigns a "type" called "Single Precision" or "Double Precision" to every anonymous real variable, but does not require that a subexpression be rounded to the accuracy that matches its syntactic "type";

better accuracy is acceptable and is provided on many machines that antedate the draft standards, as well as on the Intel 8087 and the forthcoming Motorola 68881 and Zilog Z8070. The language C evaluates all real expressions to double precision regardless of whether they contain only single-precision variables. This issue is discussed further by Kahan and Coonen,[8] and a recommendation aimed specifically at Fortran is provided by Corbett.[9]

5. Operations

Traditional arithmetic systems almost always distinguish between arithmetic operations such as addition and multiplication, and ancillary operations such as conversions, and sometimes even comparisons. The former are considered part of the arithmetic system proper, while the latter are usually left to the whims of a compiler or subroutine writer. Consequently, the benefits of using even the best arithmetic designs may be negated by the inability to properly convert data to and from machine representation, or by spurious branching based on faulty comparisons. Even data conversion in such cases may vary, depending on whether it is done by a compiler or at runtime.

Both the draft generalized standard and its companion proposed binary standard include conversions and comparisons in the arithmetic system proper. (They also include the square-root operation because known algorithms produce speed and accuracy comparable to the other operations specified when implemented in hardware.) The difficulty of specifying accurate conversion between floating-point and decimal-string representations of data is indicated by the complexity of §5.6, reproduced below.

> All conforming implementations of this standard shall provide operations to add, subtract, multiply, divide, extract the square root, find the remainder, round to a floating-point integer, convert between different floating-point precisions, convert between floating-point numbers and integers, convert between internal floating-point representations and decimal strings, and compare. Whether copying without change of precision is considered an operation is an implementation option. Except for conversion between internal floating-point representations and decimal strings, 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 precision (§4 and §7). Section 6 augments the following specifications to cover ± 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 precision, for each supported precision; it should also provide the operations for operands of differing precisions. The destination precision (regardless of the rounding precision control of §4.3) shall be at least as wide as the wider operand's precision. All results shall be rounded as specified in §4.
>
> When $y \neq 0$, the remainder $r = x$ REM y is defined regardless of the rounding mode by the mathematical relation $r = x - y \times n$, where n is the integer nearest the exact value x/y; whenever $|n - x/y| = \frac{1}{2}$, then n is even. Thus, the remainder is always exact. If $r = 0$, its sign shall be that of x. Precision control (§4.3) shall not apply to the remainder operation.

REM is defined as it is, instead of matching the "mod" function found in many programming languages, because the latter can always be computed from the former, but the converse is not always true. This is so because REM's remainder is the smallest possible remainder in magnitude, and is always exact. Note in the definition of remainder that the integer n may not be exactly representable in the precision of x, y, and r because of inadequate precision, exponent range, or both. Nevertheless, r is exactly representable in the precision of x and y. For an extreme example, consider $b = 10$, $p = 7$, $E_{min} = -99$, $E_{max} = 99$, $x = 10^{75}$, and $y = 3 \times 10^{-75}$. Then $n = \lfloor 1/3 \times 10^{150} \rfloor = 333...33$, a 150-digit integer whose value lies above the overflow threshold. Nevertheless, $r = 10^{-75}$, precisely satisfying the defining relation $x = n \times y + r$. If y is near the underflow threshold (i.e., $|y| < b^{E_{min}+p}$), it is possible that $r = x$ *REM* y may be subnormal.

Despite the fact that n may be too huge to represent exactly, many implementations of the draft binary standard return at least the least significant three or four bits of n. In effect, besides delivering $r = x$ *REM* y they deliver n *mod* b^3 or n *mod* b^4. This is convenient for calculating the reduced argument of trigonometric and other periodic functions. The extent of the convenience can be gauged from the following program segment, which uses REM twice to provide the equivalent results:

$temp = x$ *REM* $(b^3 \times y)$ . . . this is exact.

$r = temp$ *REM* y . . . this is also exact.

n *mod* $b^3 = (temp - r)/y$ . . . rounded to nearest integer.

> **5.2. Square root.** The square root operation shall be provided in all supported precisions. The result is defined and has positive sign for all operands ≥ 0, except that $\sqrt{-0}$ shall be -0. The destination precision shall be at least as wide as the operand's. The result shall be rounded as specified in §4.
>
> **5.3. Floating-point precision conversions.** It shall be possible to convert floating-point numbers between all supported precisions. If the conversion is to a narrower precision, the result shall be rounded as specified in §4. Conversion to a wider precision is exact.
>
> **5.4. Conversion between floating point and integer.** It shall be possible to convert between all supported floating-point precisions and all supported integer precisions. Conversion to integer shall be effected by rounding as specified in §4. Conversions between floating-point integers and integer precisions shall be exact unless an exception arises as specified in §7.1.

Floating-point overflow never occurs during conversion to an integer because conversion to a floating-point integer (§5.5) cannot overflow, and conversion to a fixed-point integer (§5.4), if it overflowed, would have to signal either integer overflow, if available, or invalid (§7.1). Conversion

of a nonintegral value to a floating-point integer (§5.5) is always inexact; conversion of a nonintegral value to a fixed-point integer (§5.4) is inexact unless overflow forces invalid to be signaled (§7).

5.5. Round floating-point number to integral value. It shall be possible to round a floating-point number to an integral valued floating-point number in the same precision. The rounding shall be as specified in §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.

The following section on conversions between floating-point and decimal strings was the most technically difficult section to draft. The properties specified in this section are implied by error bounds that depend on the floating-point precision and the number of digits in the decimal string; for example, the 0.47 mentioned is a worst-case bound derived for single precision on machines that conform to the draft binary standard. For a detailed discussion of these error bounds and economical conversion algorithms that exploit the extended precision on such machines, see Coonen.[10]

5.6. Floating point↔decimal string conversion. Conversion between decimal strings in at least one format and floating-point numbers in all supported basic precisions shall be provided for numbers throughout the ranges specified in Table 1. The non-negative integers D and N in Tables 1 and 2 describe decimal strings having values $\pm M \times 10^{\pm N}$, where $0 \leq M \leq 10^D - 1$.

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

For $b \neq 10$, the bounds on Max D in Table 1 correspond to conditions necessary for decimal strings to distinguish floating-point numbers one from another. In some cases slightly tighter bounds are possible. Coonen[10] discusses these matters in some detail for the draft binary standard.

The entry for Max N in Table 1 is determined as follows. Consider the following canonical form, the "scientific notation for floating-point numbers":

$$\pm (P \times b^{1-p}) \times b^E = \pm P_0 . P_1 \ldots P_{p-1} \times b^E,$$

where

$$b^{p-1} \leq P \leq b^p - 1$$

when the number is normal, and

$$0 < P \leq b^p - 1$$

when the number is subnormal. Then (see Appendix)

$$nextafter(0,1) = (1 \times b^{1-p}) \times b^{E_{min}} = b^{E_{min}+1-p}$$

is the smallest positive floating-point number, and

$$nextafter(\infty,1) = (b^p - 1) \times b^{1-p} \times b^{E_{max}} \approx b^{E_{max}+1}$$

is the largest finite floating-point number. Nonzero decimal strings have the canonical form

$$\pm M \times 10^{\pm N} = \pm M_0 M_1 \ldots M_{q-1} \times 10^{\pm N}, \quad q \leq D,$$

where M and N are chosen to minimize $|N|$ subject to the conditions $1 \leq M \leq 10^D - 1$ and $|N| \leq N_{max}$. Let Dec_{min} be the smallest positive number to be converted from decimal to floating point. Then

$$Dec_{min} \leq (10^D - 1) \times 10^{-N_{max}} \leq nextafter(0,1) = b^{E_{min}+1-p}.$$

Thus, $N_{max} \geq D + (p - 1 - E_{min}) \times \log_{10}(b)$. Similarly, let Dec_{max} be the largest finite decimal number to be converted to floating point. Then

$$Dec_{max} = (1 \times 10^{D-1}) \times 10^{N_{max}} \geq nextafter(\infty,1) \approx b^{E_{max}+1}.$$

Thus, $N_{max} \geq (1 - D) + (E_{max} + 1) \times \log_{10}(b)$. The final expression for Max N results from letting E_m denote the larger of these two bounds on N_{max}, and assuming that Max N has the form $10^n - 1$.

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

Conversions shall be monotonic. That is, increasing the value of a floating-point number shall not decrease its value when

Table 1.
Floating point↔decimal string conversion ranges.

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

Note: here $E_m = \max\{D + (p - 1 - E_{min}) \log_{10}(b), (E_{max} + 1) \log_{10}(b) + 1 - D\}$.

Table 2.
Correctly rounded conversion ranges.

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

Note: here p_e denotes the smallest precision permissible as extended support for the basic precision p (§3.3), and

$$E_m = \max\{D + (p - 1 - E_{min}) \log_{10}(b), (E_{max} + 1) \log_{10}(b) + 1 - D\}.$$

converted to a decimal string; and increasing the value of a decimal string shall not decrease its value when converted to a floating-point number.

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

For $b=10$, conversion should be correctly rounded for all values specified in Table 1. For $b=2$, the bounds on Max D and Max N for correctly rounded results are determined as follows. Let p_c be the precision in which the conversion is to be computed. Ordinarily p_c will correspond to one of the available extended precisions. Then the condition $10^D < 2^{p_c}$ defines D such that D digit decimal integers can be represented exactly in the conversion precision. Similarly, the condition $5^N < 2^{p_c}$ defines N such that 10^N can be represented exactly (further scaling of 5^N by powers of 2 to obtain 10^N is exact in a binary system). Thus, numbers of the form $\pm M \times 10^{\pm N}$, where $M \leq 10^D - 1$, can be represented in the conversion precision with, at most, one rounding error.

Example:

(1) In the draft binary standard the parameters for single are $b=2$, $p=24$, and $E_{max}=127$. The corresponding parameters for binary single-precision floating point↔decimal string conversion are Max D = 9, Max $M = 10^9-1$, and Max N = 99 for conversion to floating point and Max N = 53 for conversion to decimal string. The range for correctly rounded conversion is further restricted to Max N = 13.

When rounding to nearest, conversion from floating point to decimal string and back to floating point will be the identity, provided that D is large enough to distinguish floating-point numbers one from another (the conditions in Table 1) and that, for $b=2$, p_c is large enough to represent D digit decimal integers exactly, i.e., provided

$$\lfloor p_c \log_{10}(2) \rfloor \geq D \geq \lceil p_s \log_{10}(2) + 1 \rceil,$$

or

$$\lfloor p_c \log_{10}(2) \rfloor \geq \lfloor p_s \log_{10}(2) + 2 \rfloor.$$

This is implied by the condition

$$p_c \geq p_s + 6.64 \ldots$$

or, because p_c and p_s are integers,

$$p_c \geq p_s + 7.$$

This relation is automatically satisfied when p_c is an extended precision as defined in §3.3. For example, when $p_c = p_{se}$,

$$p_c \geq p_s + \lceil \log_2(E_{max_s} - E_{min_s}) \rceil.$$

Noting that $2^{p_s-1} \geq 10^5$, we have

$$p_s \geq \frac{5}{\log_{10}(2)} + 1 \geq 18.$$

Because $(E_{max_s} - E_{min_s}) > 5p_s \geq 90$, we finally have

$$p_c \geq p_s + \lceil 6.49 \rceil = p_s + 7.$$

If decimal string to floating point conversion overflows/underflows, the response is as specified in §7. Overflow/underflow and NaNs and infinities encountered during floating point to decimal string conversion should be indicated to the user by appropriate strings. The letters "NaN," case insensitive, optionally preceded by an algebraic sign, should be the first characters of a string representing a NaN. The remainder of the string may be used for system-dependent information on output, and may be ignored on input. Unless recognized as a quiet NaN on input, an input NaN should become a signaling NaN. The letters "infinity," case insensitive, optionally preceded by an algebraic sign, or the string "1/0," optionally preceded by an algebraic sign, should be the characters representing signed infinity. Either representation may be produced on output; both should be accepted on input.

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

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

There are two kinds of NaNs in the standard, signaling and quiet, as explained in §6.2. The question now is why should input NaNs be presumed by default to be signaling NaNs? First, the input is not a NaN but a string presumably intended to be converted into a NaN. It seems reasonable further to presume that the intention was to introduce a signaling NaN, since the standard provides no other way to create such a thing. Should that presumption be wrong, little harm is done because in the absence of a trap enabled to handle signaling NaNs, they merely turn into quiet NaNs with no further effect than to set the invalid operation flag.

The choice of strings to designate infinities is limited. Note that "INF" is currently used in some implementations of Basic to designate the largest finite number in the floating-point system.

The standard does not discuss the problem of output-field overflow, a problem that must be solved by language implementors. An attempt to output a string of characters too long for the preallocated field width must somehow notify the user that his implicit assertion about the size of his result has failed. That notification need not disturb the format of subsequent output. For instance, in the case of printing N = −12345.6789 in a Fortran output field defined to be F6.2, the expected line

```
#####±nn.nnxxxxxxxxxx
```

might be replaced by two lines

```
#####-12345.68
          xxxxxxxxxx
```

without disturbing the relationship of subsequent lines on the page.

5.7. Comparison. It shall be possible to compare floating-point numbers in all supported precisions, even if the operands' precisions differ. Comparisons are exact and never

overflow 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: 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 the last column of Table 3, "unordered" operands are compared using one of the predicates involving "<" or ">" but not "?" (Here the symbol "?" signifies "unordered.")

Table 3 exhibits the twenty-six functionally distinct useful predicates named, in the first column, using three notations: *ad hoc*, Fortran-like, and mathematical. It shows how they are obtained from the four condition codes and tells which predicates cause an invalid operation exception when the relation is "unordered." The entries T and F indicate whether the predicate is true or false when the respective relation holds.

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

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

There may appear to be two ways to write the logical negation of a predicate, one using "NOT" explicitly and the other reversing the relational operator. For example, the logical negation of (X = Y) may be written either NOT(X = Y) or (X ?<> Y); in this case both expressions are functionally equivalent to ($X \neq Y$). However, this coincidence does not occur for the other predicates. For instance, the logical negation of ($X < Y$) is just NOT($X < Y$); the reversed predicate (X ?>= Y) is different in that it does not signal an invalid operation exception when X and Y are "unordered."

6. Infinity, NaNs, and signed zero

The special quantities infinity, NaNs, and signed zero included in the draft generalized standard provide closure of sorts to the arithmetic system and permit sensible responses to the exceptional conditions to be discussed in §7. For example, ∞ provides a default result for overflow and division by zero, while the sign on ∞ indicates the direction of the overflow. For mathematical consistency, and to aid in implementing interval arithmetic, zero also carries a sign.

Table 3.
Predicates and relations.

Predicates			Relations				Exception
ad hoc	Fortran	math	greater than	less than	equal	unordered	invalid if unordered
=	.EQ.	=	F	F	T	F	No
?<>	.NE.	≠	T	T	F	T	No
>	.GT.	>	T	F	F	F	Yes
>=	.GE.	≥	T	F	T	F	Yes
<	.LT.	<	F	T	F	F	Yes
<=	.LE.	≤	F	T	T	F	Yes
?	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

This may indicate the direction from which underflow occurred, or it may indicate the sign on an ∞ that has been reciprocated. All of the arithmetic properties of the special quantities have been similarly designed to emulate their mathematical properties as nearly as possible.

The NaNs are special entities introduced to handle otherwise intractable situations, such as providing a default result for 0/0. The fact that they propagate through a computation suggests that they may also be used, with special programming, to provide error traces of various sorts. They might even be used to implement special arithmetics by taking advantage of their triggering of arithmetic exceptions. However, none of these more esoteric applications are mentioned in the draft standard.

> **6.1. Infinity arithmetic.** Infinity arithmetic shall be construed as the limiting case of real arithmetic with operands of arbitrarily large magnitude, when such a limit exists. Infinities shall be interpreted in the affine sense, that is, $-\infty <$ (every finite number) $< +\infty$.
>
> Arithmetic on ∞ is always exact and therefore shall signal no exceptions, except for the invalid operations specified for ∞ in §7.1. The exceptions that do pertain to ∞ are signaled only when
>
> (1) ∞ is created from finite operands by overflow (§7.3) or division by zero (§7.2), with the corresponding trap disabled, or
>
> (2) ∞ 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 enhancements (such as complex-affine infinities or extremely wide range) that are not the subject of the standard. Quiet NaNs should, by means left to the implementor's discretion, afford retrospective diagnostic information inherited from invalid or unavailable data and results. Propagation of the diagnostic information requires that information contained in the NaNs be preserved through arithmetic operations and basic precision conversions.
>
> Signaling NaNs shall be reserved operands that signal the invalid operation exception (§7.1) for every operation listed in §5. Whether copying a signaling NaN without a change of precision signals the invalid operation exception is the implementor's option.

Signaling NaNs are closely analogous to the reserved operands on the DEC PDP/11 and Vax, and to the indefinite operand on the CDC 7600 and Cyber, in that they precipitate some signal when touched but not when created. Quiet NaNs are similar to the reserved operand on the Vax in that they result from an invalid operation, but differ in that they generate no signal when encountered in subsequent operations:

> 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 delivered, shall deliver as its result a quiet NaN, which should be one of the input NaNs. Note that precision conversions might be unable to deliver the same NaN. Quiet NaNs have effects similar to signaling NaNs on operations that do not deliver a floating-point result; these operations, namely, comparison and conversion to a precision that has no NaNs, are discussed in §5.4, §5.6, §5.7, and §7.1.
>
> **6.3. The algebraic sign.** This standard says nothing about the sign of a NaN. Otherwise the sign of a product or quotient is the Exclusive Or of the operand's signs; and the sign of a sum, or of a difference $x-y$ regarded as a sum $x + (-y)$, differs from at most one of the addend's signs. 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 positive sign.

7. Exceptions

One objective of the draft standard is to minimize for users the complications arising from exceptional conditions. The arithmetic system is intended to continue to function on a computation as long as possible, handling unusual situations with reasonable default responses, including setting appropriate flags. Because a user may want to override default results and do something special in response to an exception, the standard also accommodates user-supplied traps:

> 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 §8. The default response to an exception shall be to proceed without a trap. This standard specifies results to be delivered in both trapping and nontrapping situations. In some cases the result is different if a trap is enabled.
>
> For each type of exception the implementation shall provide a status flag that shall be set on any occurrence of the corresponding exception when no corresponding trap occurs. It shall be reset only at the user's request. The user shall be able to test and to alter the status flags individually, and should further be able to save and restore all five at one time.
>
> The only exceptions that can coincide are inexact with overflow and inexact with underflow.

A flag records the occurrence of an exceptional event. Resetting a flag clears it, and it stays clear until the first subsequent event sets it again, or until it is restored by the user to a previous state. Until reset by the user, the set state of a flag perseveres, providing evidence of a past event; the flag may in some implementations point to diagnostic information about the location and nature of the event and/or, perhaps, some subsequent ones.

The crudest implementations will use just one bit for a flag. Next better is to point to the last event since the user altered the flag. Pointing to the first event after the flag was reset is slightly better, and feasible for high-speed machines. Much better, and much harder to implement, is to point to the first and the last, and to keep count of how many occurred in between.

In all cases, a human user should ideally be able to monitor what is going on by means of some kind of annunciator, perhaps a panel light or a blinking dot on the screen to signify that certain (preselected) flags were set and are being or have been dealt with by the program. Interrogating the annunciator should reveal which flags are set, and further interrogation should reveal why.

Because the flag is an object known to the operating system, to alter a flag (to set or reset or restore it) may entail a call to the operating system or to a runtime library procedure. But to sense whether a flag is clear, and perhaps to save its value, may entail little more than a memory reference. Thus, alteration of a flag, including resetting it, may be too slow for inclusion with other operations inside a loop, while a true/false test of a flag should be fast enough for that.

7.1. Invalid operation. The invalid operation exception is signaled if an operand is invalid for the operation to be performed. The floating-point result delivered when the exception occurs without a trap shall be a quiet NaN (§6.2). The invalid operations are

(1) Any operation on a signaling NaN (§6.2);

(2) Addition or subtraction: magnitude subtraction of infinities like $(+\infty) + (-\infty)$;

(3) Multiplication: $0 \times \infty$;

(4) Division: 0/0 or ∞/∞;

(5) Remainder: x REM y, where y is zero or x is infinite;

(6) Square root if the operand is less than zero;

(7) Conversion of an internal floating-point number to an integer or to a decimal string when overflow, infinity, or NaN precludes a faithful representation in that format and this cannot otherwise be signaled;

(8) Conversion of an unrecognizable input string; and

(9) Comparison via predicates involving "<" or ">", without "?", when the operands are "unordered" (§5.7, Table 3).

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

The exception called "Division by Zero" is misnamed for historical reasons. The division by zero flag, when set, indicates precisely that an infinite result was delivered by some previous floating-point operation on finite operands, that the exact result is *not* finite, and that no trap was taken (whether not implemented, or implemented but not enabled by the user). For example, $3.0/0.0 = +\infty$; similarly, $ln(0) = -\infty$, $tan(90°) = -tan(-90°) = \infty$, $atanh(1) = +\infty$, . . . though these are not specified by the draft standards.

A different exception, "Overflow," is signaled when an infinite result is produced inexactly from finite operands. No exception is signaled when an infinite result is produced from infinite operands, as in $\infty \times \infty$, $\infty/0$, $\sqrt{+\infty}$, . . . and, presumably, $exp(+\infty)$, $ln(+\infty)$, $sinh(\infty)$, ...

7.3. Overflow. The overflow exception shall be signaled whenever the destination precision's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result (§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:

(a) Round to nearest carries all overflows to ∞ with the sign of the intermediate result.

(b) Round toward 0 carries all overflows to the precision's largest finite number with the sign of the intermediate result.

(c) Round toward $-\infty$ carries positive overflows to the precision's largest finite number and carries negative overflows to $-\infty$.

(d) Round toward $+\infty$ carries negative overflows to the precision'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 b^{α} and then rounding. The exponent adjustment α for a precision shall be chosen to be approximately $3 \times (E_{max} - E_{min})/4$ for that precision, and should be divisible by twelve. Trapped overflow on conversion from a floating-point precision shall deliver to the trap handler a result in that or a wider precision, possibly with the exponent adjusted, but rounded to the destination's precision. Trapped overflow on decimal string to floating point conversion shall deliver to the trap handler a result in the widest supported precision, possibly with the exponent adjusted, but rounded to the destination's precision; when the result lies too far outside the range for the exponent to be adjusted, a quiet NaN shall be delivered instead.

The exponent adjustment of approximately $3 \times (E_{max} - E_{min})/4$ is chosen to translate overflowed/underflowed results 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. The requirement for divisibility by 12 simplifies the extraction of low order integer (square, cubic, and quartic) roots.

Traditionally overflow has been associated with the production of results whose magnitudes lie above a threshold $b^{E_{max}+1}$ beyond which values cannot be represented. The draft continues this tradition.

If the program will respond to overflow by aborting, then a simple trap suffices. If instead it is desired to continue the computation under circumstances that will not lead to misleading results, some value must be delivered. Some systems simply deliver the number of largest magnitude and correct sign. Others deliver special values which behave approximately (but not exactly) like the ∞ of this draft.

The draft standard prescribes a default infinite result and, in addition, a solid indication that this substitution has

occurred—the overflow flag. The overflow flag, when set, indicates precisely that an infinite result was delivered by some previous floating-point operation on finite operands, that the exact result *is* finite, and that no trap was taken (whether not implemented, or implemented but not enabled by the user). It is therefore also appropriately raised by such things as attempts to compute $exp(10^{30})$.

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

(1) "after rounding": when a nonzero result computed as though the exponent range were unbounded would lie strictly between $\pm b^{E_{min}}$;

or

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

Loss of accuracy may be detected as either

(3) a denormalization loss: when the delivered result differs from what would have been computed were exponent range unbounded;

or

(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 (via the underflow flag) only when both tininess and loss of accuracy have been detected. The method for detecting tininess and loss of accuracy does not affect the delivered result which might be zero, subnormal, or $\pm b^{E_{min}}$. When an underflow trap has been implemented and is enabled, underflow shall be signaled when tininess is detected regardless of loss of accuracy. Trapped underflows on all operations except conversion shall deliver to the trap handler the result obtained by multiplying the infinitely precise result by b^{α} before rounding, where the exponent adjustment α shall be the same as in §7.3. Trapped underflows on conversion shall be handled analogously to the handling of overflows on conversion.

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

Floating-point formats which do not include subnormal values naturally associate underflow with the corresponding production of results whose magnitudes lie beneath the threshold $b^{E_{min}}$. In these cases, the value zero is clearly the best approximation to an underflowed result when it is necessary to deliver some numerical value. This situation has two potentially adverse consequences. First, the zero value thus produced may precipitate a division by zero or other untoward event which would not have occurred absent the underflow. Second, the rounding error which accompanies the replacement of a nonzero result is substantially greater than would have arisen had the exponent not been too small.

The presence of subnormal numbers in the draft allows the effects of underflow to appear gradually. In particular, the subtraction of nearly equal numbers each near $b^{E_{min}}$ always produces a subnormal value which can be represented exactly. These tiny numbers may be more prone to precipitating overflows if reciprocated, but will not lead to division by zero unless exactly equal numbers were subtracted. In the case of multiplication and division, a zero approximation to an underflowed result can still arise, but in any event the maximum error is reduced from $b^{E_{min}}$ to $b^{E_{min}-P}$.

The significance of underflow varies according to application and will often be concerned either with taking evasive action where small (subnormal) values occur or with obtaining a diagnostic indication that approximation error in excess of that attributable to rounding (signaled by inexact) has occurred.

Thus, the definition of underflow differs, depending on whether traps are to be taken. When an underflow trap is implemented and enabled, the threshold test is appropriate for all operations, independent of whether a rounding error would arise in a subnormal approximation. This allows action such as scaling to be taken upon the first production of a value beneath the underflow threshold. When only the flag is to be set, the indication is appropriate only in the presence of a rounding error that may invalidate an a priori error analysis. Thus, addition, subtraction, and remainder, whose results are always exact when they lie beneath the underflow threshold, cannot raise the underflow flag.

The denormalization loss test is preferable to the inexact result test, which can occasionally deliver a slightly pessimistic indication of lost accuracy. However, the denormalization loss test may be substantially harder to implement, in which case the inexact result test is an acceptable compromise.

Implementations that always hold operands and deliver results in double or extended format registers, shortening to single or basic precision only upon storing (§4.3), may signal underflow during a copy operation if treated as an arithmetic operation (which the draft standard permits) provided the trap is enabled; however, when a trap is not present or not enabled, the copy operation cannot raise the underflow flag because it is an exact operation.

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.

Inexact applies only to valid operations. Inexact and underflow flags are destined to be raised frequently, and then

usually ignored. To describe a way to avoid unnecessarily updating these flags, we must distinguish between hardware flags (single bits in a processor's state) and software flags (pointers stored in memory), between hardware trap handlers (part of the operating system) and software trap handlers (subroutines preselected by a user), and between a hardware trap-disable bit and a software-trap enable pointer (to the software trap handler preselected by the user). The scheme is fast because it identifies the hardware flag bit with the hardware trap-disable bit.

In this scheme, the operating system will arrange that each hardware trap-disable bit (hardware flag bit) be on if and only if its corresponding software flag is nonnull, so the occurrence of underflows or inexact results will not be concealed from the user. Assume that the hardware must produce an underflowed or inexact result.

In the usual case, the appropriate trap-disable bit is already on, and the appropriate default result is delivered directly with no further exceptional activity required of the processor. When the appropriate trap-disable bit is off, then that bit should be turned on, and a rounded result, possibly denormalized or with adjusted exponent, should be delivered to the hardware trap handler together with a bit of information sufficient to compute a denormalized result from one with adjusted exponent if necessary.

The hardware trap handler can tell what has just happened by comparing the present state of the hardware trap-disable bits with the previous state. If an appropriate nondefault software trap handler has been enabled by the user, the hardware trap handler resets the hardware trap-disable bit, then invokes the software trap handler. Otherwise, the hardware trap handler places a suitable nonnull pointer into the proper software flag and delivers the appropriate default result. Thus, if an event like inexact occurs many times before its software flag is reset, only the first occurrence delays computation to set the flag.

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 §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 §7, shall be delivered to it.

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

When a system traps, the trap handler 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 precision;

(4) in overflow, underflow, and inexact exceptions, the correctly rounded result, including information that might not fit in the destination's precision; and

(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.

Appendix: Recommended functions and predicates*

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

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

(1) copysign(x,y) returns x with the sign of y. Hence, abs(x) := copysign(x,1.0), even if x is NaN.

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

(3) scalb(x,N) returns $x \times b^N$, for integral values N without computing b^N.

(4) logb(x) returns the exponent of x, a signed integer in the precision of x, except that logb(NaN) is a NaN, logb(∞) is $+\infty$, and logb(0) is $-\infty$ and signals the division by zero exception. When x is positive and finite, the expression scalb (x, $-$logb(x)) lies strictly between 0 and b; it is less than 1 only when x is subnormal.

Logb of a subnormal x is $-E_{\min}$.

(5) nextafter(x,y) returns the next representable neighbor of x in the direction toward y. The following special cases arise: if $x=y$, then the result is x without any exception being signaled; otherwise, if either x or y

*This appendix is not part of the proposed IEEE Standard 854 for Radix- and Word-length-independent Floating-point Arithmetic, but is included for information only.

is a quiet NaN, then the result is one or the other of the input NaNs. Overflow is signaled when x is finite but nextafter(x,y) is infinite; underflow is signaled when nextafter(x,y) lies strictly between $\pm b^{E_{min}}$; in both cases, inexact is signaled.

(6) finite(x) returns the value TRUE if $-\infty < x < +\infty$, and returns FALSE otherwise.

(7) isnan(x), or equivalently $x \neq x$, returns the value TRUE if x is a NaN, and returns FALSE otherwise.

(8) $x<>y$ is TRUE only when $x<y$ or $x>y$, and is distinct from $x \neq y$, which means NOT($x=y$) (see again Table 3).

(9) unordered(x,y), or $x?y$, returns the value TRUE if x is unordered with y, and returns FALSE otherwise (see again Table 3).

(10) class(x) tells which of the following ten classes x falls into: signaling NaN, quiet NaN, $-\infty$, negative normal, negative subnormal, -0, $+0$, positive subnormal, positive normal, and $+\infty$. This function is never exceptional, not even for signaling NaNs. ■

References

1. D. Stevenson, chairman, "A Proposed Standard for Binary Floating-point Arithmetic, Draft 10.0," IEEE Floating-point Subcommittee Working Document P754/82-8.6, 1982. (A copy of this draft is available from R. Karpinski, U-76, University of California, San Francisco, CA 94143. Draft 10.0 supercedes Draft 8.0, which was published as "A Proposed Standard for Binary Floating-point Arithmetic," in *Computer*, Vol. 14, No. 3, Mar. 1981, pp. 51–62.)
2. "Floating-point Standards Move Toward Adoption," to appear in *Computer*.
3. W. S. Brown, "A Simple But Realistic Model of Floating-point Computation," *ACM Trans. Math. Software*, Vol. 7, No. 4, Dec. 1981, pp. 445–480.
4. J. T. Coonen, "An Implementation Guide to a Proposed Standard for Floating-point Arithmetic," *Computer*, Vol. 13, No. 1, Jan. 1980, pp. 68–79. (Errata in *Computer*, Vol. 14, No. 3, Mar. 1981, p. 62.)
5. W. Kahan, "The Proposed IEEE Standard P754 for Floating-point Arithmetic: What Good Is It?" Session 16 of Mini/Micro West, San Francisco, CA, Nov. 10, 1983.
6. T. E. Hull, "The Use of Controlled Precision," in *The Relationship Between Numerical Computation and Programming Languages*, J. K. Reid, ed., North-Holland Pub. Co., Amsterdam, 1982, pp. 71–82.
7. Intel Corp., *Fortran-86 User's Guide*, Santa Clara, CA, 1982.
8. W. Kahan and J. T. Coonen, "The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments," in *The Relationship Between Numerical Computation and Programming Languages*, J. K. Reid, ed., North-Holland Pub. Co., Amsterdam, 1982, pp. 103–113.
9. R. P. Corbett, "Enhanced Arithmetic for Fortran," *ACM Signum Newsletter*, Vol. 18, No. 1, Jan. 1983, pp. 24–28; *ACM Sigplan Newsletter*, Vol. 17, No. 12, Dec. 1982, pp. 41–48.
10. J. T. Coonen, "Accurate, Economical Binary↔Decimal Conversions," submitted for publication.