Contributions to a Proposed Standard for Binary Floating-Point Arithmetic

By

Jerome Toby Coonen

B.S. (University of Illinois) 1975
M.S. (University of Illinois) 1975

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Mathematics

in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved: W Kahan June 1, 1984
Chairman Date

Richard J. Fateman 1 June 1984

Ole Lauren Hald 1 June 1984

.....

Contributions to a Proposed Standard for Binary Floating-Point Arithmetic

Jerome T. Coonen

ABSTRACT

In the fall of 1977 the Institute of Electrical and Electronics Engineers commissioned working group 754 to draft a standard for binary floating-point arithmetic. It was intended to prevent the proliferation of disparate arithmetics in the new microprocessor industry. At that time there were so many different flavors of arithmetic available on mainframes and minicomputers that the cost of reconciling their differences in numerical software had become, and remains, staggering. Now, more than five years later, draft 10.0 of the proposed standard has been voted out of the working group for IEEE approval.

This thesis consists of a set of "footnotes" to the proposed standard. The first of them, an implementation guide published in January 1980, served as a working draft of the standard for over a year. The remaining chapters unfolded as the proposed standard did. They include an analysis of gradual underflow, the most controversial feature of the standard; an exhaustive discussion of radix conversion, which has been specified in the proposed standard only up to a worst-case error bound; and a revised version of the arithmetic test suite which has been available in machine-readable form from the working group.

Approved:

W. Kahan

June 1, 1984

to my parents

Table of Contents

CHAPTER 1

Introduction

"Most numerical analysts have no interest in arithmetic."
B. N. Parlett (1979)

The lack of interest abounds. Professor Parlett's claim applies to computer designers as well as users. And it is usually the speed of arithmetic that incites what interest there is. Yet a proposed IEEE standard for binary floating point arithmetic is in the last stage of approval before that body's Standards Board, and, despite that the proposal is hard to implement, it has become already a *de facto* standard among several of the largest microprocessor manufacturers. Why?

Calculator and computer users are familiar with the fact that the quotient 1/3 must be rounded in order to be representable on a binary or decimal machine. But rounding is not to blame when 1/3 differs from 9/27. Such a capricious discrepancy can cause a perfectly reasonable program to fail mysteriously, arousing dismay, not interest. Also daunting is the prospect of developing software to run across the dozens of diverse arithmetics in use today, a number that will increase with the rise of the microprocessor industry.

This thesis is about the proposed IEEE standard 754 for binary floating point arithmetic. The thesis developed alongside the standard itself, as a set of clarifications and elaborations of the terse 754 document; it is an aid to implementors, and a demonstration that the implementation is feasible. Because of the care taken in the specification of proposed standard 754, and

because of its rising support within the industry, there is hope for an end to the dismay caused by bad arithmetic. In a sense, it is the best arithmetic that arouses the least interest among users.

1. A Brief History of IEEE Working Group 754

In the fall of 1977, working group 754 of the IEEE Computer Society Microprocessor Standards Committee was convened to draft an industry standard for floating point arithmetic on microprocessors. It was known that Intel Corporation was pursuing high-quality arithmetic for its family of products. The orginal intent of the working group was simply to fix a set of common data formats so that binary data could be transferred between different microprocessors. The first meetings of the working group were attended by microprocessor enthusiasts, including Bob Stewart and Tom Pittman, as well as John Palmer of Intel and W. Kahan of the University of California at Berkeley, then consulting to Intel. Richard Delp chaired the meetings.

Due chiefly to the leadership of Kahan, the scope of the working group quickly expanded from data formats to a thorough specification of arithmetic. In early 1978 Kahan enlisted the support of Harold S. Stone, then visiting Berkeley, and the author to draft a proposal whose key ideas were drawn from Kahan's years of experience on machines ranging from mainframes to pocket calculators. Kahan estimated that the project would require "one hard man-month of effort". He underestimated. Over the next three months, drafts of the so-called Kahan-Coonen-Stone proposal were presented to the monthly meetings of the working group. Throughout this period of refinement, Palmer and others at Intel were developing a major VLSI implementation of the proposal.

By late 1978 the working group included members from National Semiconductor, Motorola, Zilog, Monolithic Memories, Apple Computer, Tektronix, and Digital Equipment Corporation. There was a certain irony about the standardization process – on the one hand the working group was chartered to develop an industry standard, while on the other hand its work was supposed to be uninhibited by the kind of partisan politicking that arises naturally among competing manufacturers. At that time, the proposal was embodied in an implementation guide prepared by the author; this paper, finally published in January 1980, appears as Chapter 2.

Over the subsequent year several competing proposals were presented to the working group. Mary H. Payne and William Strecker of DEC proposed what could be thought of as enhanced VAX-11 arithmetic. Steve Walther and Robert Fraley of Hewlett-Packard Laboratories proposed what they thought of as a "safer" scheme, with special symbols for underflowed and overflowed values. Robert Reid, working independently, developed an idea that arises occasionally in the literature, varying the width of a number's exponent field dynamically, widening it (while narrowing the significand) in order to accommodate extremely large or tiny magnitudes. A subcommittee of Pittman, Palmer, Kahan, and the author was commissioned to cast the prevailing proposal in a form suitable for an IEEE standard. David K. Stevenson later joined the group; and subsequently he was voted chairman of the entire working group.

Draft 5.11 of the proposed standard stood without change for over a year. It was revised up to draft 8.0 in preparation for the March 1981 issue of IEEE *Computer* magazine, of which an entire section was devoted to floating point standardization. Discussions in the working group continually bogged

down on the issue of underflow – by far the most controversial aspect of the proposed standard. In an attempt to present the issues on paper, for surely resolution seemed beyond hope, the author prepared the paper which, as published in that issue of *Computer*, appears as Chapter 5.

Shortly after publication of draft 8.0, the working group voted to develop that proposal, to the exclusion of the others. One last round of changes was due. Over mid-1981 two features were removed from the proposal, the *projective* mode interpretation of infinity and the *warning* mode interpretation of the denormalized numbers. In lively debate within the working group it was decided that the modicum of safety bought by these modes was not worth the known complexity of implementing them and explaining them to users. Today, almost seven years since the working group first met, draft 10.0 of proposed standard 754 has reached the last level of approval, the IEEE Standards Board. A slightly abbreviated version of the draft appears as Appendix A.

2. Design Goals – User Friendly Floating Point Arithmetic?

Although common data formats were the goal when the 754 working group was chartered, three simple design principles evolved: ensure that most existing programs would run at least as well on standard systems as they had on earlier machines with comparable range and precision; provide the most robust arithmetic possible with 1980's technology; and include features to enhance software development by experts.

In order to preserve the substantial investment in existing software, the proposal has to be as least as good as any other arithmetic available. This turns out not to be a significant constraint, and is really subsumed by the desire to build the best possible arithmetic. But old software could be

undermined by excellent arithmetic with features unknown to the original programmer. Since most of the innovations in 754 apply when exceptions arise, they affect old programs only when some exception, for example overflow or division by zero, occurs. In such cases an earlier machine would probably stop execution anyway. The situation with the comparison operator is different; here a mechanism was included specifically to defend old programs and programmers. This is the subject of Chapter 6.

Who could determine just how much arithmetic could be implemented on a chip in the current technology? In order to bound its efforts, the working group required some measure of feasibility. This came from two arenas. As mentioned before, Intel was well into the design of the i8087 coprocessor to the 8086/8088 CPUs. They stretched the limits in die size and yield. At the same time, George Taylor, a Berkeley graduate student, was designing a set of circuit boards implementing 754 which could replace the VAX-11/780 floating point accelerator boards. Taylor [9] showed that, with care, the cost and complexity of 754 could be reduced to that of the more ordinary VAX, whose arithmetic is in fact very good already.

In the next section we will survey what the standard *does* include. It is appropriate to discuss here what was deliberately excluded. From the start, 754 was a binary standard. Although decimal arithmetic has obvious advantages for most end users (in contrast to computational advantages of binary), it was deferred to a later standard [2]. The elementary functions, although implemented on chip by Intel and others, were deemed beyond the scope of a standard intended for simple control devices as well as general purpose computers. Also, just the standardization of transcendental functions is complicated by the discussion of allowable errors. (Chapter 7, on

binary-decimal conversions, typifies the kind of analysis involved.) Finally, interval arithmetic was omitted despite its potential for computing and reporting error bounds. However, the standard requires the implementation of modes of rounding that support the economical implementation of interval arithmetic in software.

Adding features to a system is always easy. In the case of 754, to its credit, the experts' features arose naturally from the base design, which is surveyed below. The availability of special rounding modes, such as just mentioned, error flags to check for the occurrence of an exception that would otherwise be dispatched in a specified fashion, or special functions, such as recommended in the appendix to 754, all support the development of high-quality codes.

The point of the 754 design is to provide the most robust arithmetic possible while limiting "error messages" to those times when the bounds of its capability have been surpassed. This is a delicate line to walk. Cry "Wolf!" too often, such as on every occurrence of underflow, and the message will be ignored. Let a computation run amok with no indication, all the while substituting, say, 0 for overflowed values, and inevitably some user of another's software will be misled. In the parlance of human engineering, 754 is user friendly since anyone doing ordinary calculations benefits without knowledge of the sometimes arcane underpinnings. Only when necessary, must a user be faced with the more elaborate aspects of the system.

3. An Overview of Proposed Standard 754

The brew is surprisingly straightforward. Start with single and double data formats of 32 and 64 bits, respectively. Suggest somewhat wider single-extended and double-extended formats for use in expression evaluation to

alleviate intermediate overflow and underflow. Specify a complement of rational arithmetic operations, and include square root, remainder, and binary-decimal conversion. Finally, specify the machine arithmetic to be *closed* under all operations on all operands. These ideas are expanded in the rest of this section. Chapter 8 gives a top-down specification of the arithmetic from the implementor's point of view.

The data formats are quite ordinary. Single has the range and precision of the PDP-11 float format; double has the range of CDC 6000 class single format (a 60-bit word), which is widely used for scientific computing. The extended formats have roots in the IBM 709x and Univac 1108 extended accumulators; their widths in range and precision have been chosen to aid in binary-decimal conversion and the computation of the exponential X^Y.

Square root is required by the standard because of its utility in certain calculations, such as least squares, and because it is known to be just a minor variation of division. Remainder is harder to implement, because so many steps of division may be required before the dividend is reduced to half the magnitude of the divisor. But remainder is vital to the argument reduction required for the elementary functions. Binary-decimal conversion, historically in the province of the systems programmers or language implementors, is included so that tight error bounds can be specified, in lieu of correct rounding which may be infeasible due to cost. Chapter 7 is an extensive analysis of the bounds stated in 754. Appendix D shows a correctly-rounded conversion implemented in Pascal. Other operations required by 754 are means to access and modify the *state* of the arithmetic engine, for example, the rounding modes and error flags.

It is arithmetic closure that gives 754 its true flavor. To cope with overflow and computations like 1/0, signed ∞ symbols were added to the number system. And the sign of ∞ was made to interact with the sign of zero in the ordinary way, so that $1/-\infty = -0$. The cost of this is a sign on zero (unlike the real number system) which is sometimes misinformation when it must be assigned arbitrarily, as with the result of $3.14 - 3.14$. To cope with underflow, the controversial denormalized numbers were added at the bottom of the number range. Simply put, these values ensure that a difference $x-y$ is nonzero just when $x \neq y$; on most current machines, the difference of two tiny values will be flushed to zero if it falls below a certain threshold. Chapter 5 discusses this issue in detail. Contention notwithstanding, arithmetics with infinities and denormalized numbers had been implemented before, for example on the CDC 6000 class machines and the Dutch Electrologica X8, respectively.

Closure of invalid operations like $0/0$ and $\sqrt{-5}$ required a new kind of symbol, for Not-a-Number. The so-called NaNs are a true innovation within the standard. Although they are numerically trivial, since they propagate unchanged through arithmetic, the NaNs have a considerable impact on the overall architecture of a system, as mentioned with language issues below and in Chapter 6. NaNs have already found use not only as diagnostic aids but as placeholders for missing or unavailable data in spreadsheets and statistical applications. The key to the NaNs' utility is their propagation through arithmetic operations; the "indefinite" operands in the CDC 6000 class computers and the "reserved" operand in the DEC PDP-11 and VAX-11 computers trigger a (typically fatal) exception each time they are encountered, rendering them useless for carrying information.

4. Yet Another Standard – 854

When the 754 standard effort was nearing completion, a second standard was launched under the chairmanship of William J. Cody [2]. What started as a radix- and wordlength-independent standard developed into a binary-and-decimal standard, with suggestions about the balance between the range and precision to be provided in a given wordlength. The 854 standard was constrained to be upward compatible from 754. In fact, the drafts were developed by simply modifying 754 in a text editor. The principal difference is in the area of binary-decimal conversion, which is even more obscure when the binary range and precision are not given specifically. Tables of inequalities specify bounds for the allowable errors.

5. Axiomatic Attempts

> "Of course, if [the axiomatization of rounded floating-point arithmetic] is to be useful, the axioms should be simple enough for each comprehension (sic). I am afraid this goal has not yet been achieved."
>
> R. Mansfield (1984)

While standards 754 and 854 maintain essential backward compatibility with arithmetics of the past, their main thrust is toward a future of greater commonality among machines. A coincident development has attempted to make numerical sense of the machines we must program for *today*. W. Stan Brown characterizes a machine's arithmetic according to a set of parameters [1]. The parameters describe the range and precision of the machine's values that satisfy the criteria for Brown *model numbers*. On many machines only a subset of the representable values, such as those not too huge or tiny, or those with one or more trailing zero digits, are model numbers satisfying constraints like commutativity of multiplication. Brown can confirm a machine's parameters by running a crafty test program in portable FORTRAN

developed by Norm Schryer [8].

Brown's attempt to unify current arithmetics sheds further light on the current state of affairs, but falls short of real utility for numerical programmers. First, since Brown stated as a design goal the development of axioms pertinent to every major computer in use in the Free World, his axioms in a sense inherited the worst properties of all the machines. They are subtle indeed. It has been shown, for example, that because of a certain class of division algorithms, one cannot infer from the model that the inequality $0 < x \leq y$ implies that $x/y \leq 1$. Problems like this will be nightmares for programmers who would guarantee robustness [4]. Chapter 4 suggests FORTRAN procedures for interrogating a system about parameters relative to both Brown's model and the proposed standards.

By itself Brown's model is no more than further research into the behavior of computer arithmetics, but when taken as the *standard* characterization of arithmetic from which programmers must work, it can actually hinder advances like the 754 and 854 proposals from taking effect by stripping their advanced features which, of course, don't fit into the "least common denominator" model. A step in this direction has been taken by the Ada standards group, which has incoporated the ideas of the Brown model in the Ada specification of arithmetic. Fortunately, the use of Ada packages permits the incorporation of other arithmetics such as 754 and 854, albeit inconveniently [5].

Brown's is just the most computationally oriented of several attempts at axiomatization. In 1966 A. van Wijngaarden uttered 32 rules for arithmetic, introducing a *tolerance* operator to describe the deviation of machine arithmetic from real arithemtic [10]. More recently, R. Mansfield has listed 45

axioms for computer arithmetic in order to prove that a qualifying arithmetic is in fact rounded from an ordered field [7]. As he testifies in the quote that opens this section, such a blizzard of axioms is incomprehensible.

6. An Algebraic Approach

Another recent development in arithmetic is worth brief mention in contrast with the 754 and 854 efforts. The latter have been dauntlessly pragmatic. Most of what has been written, and this thesis is a prime example, has centered on implementation details and the use of the arithmetic to solve well-known problems. A much more formal approach has been taken by Ulrich Kulisch and Willard Miranker as described in their book *Computer Arithmetic in Theory and Practice* [6]. Their ultimate goal is a machine analog to the algebra of vectors and matrices over the complex domain. The key is the ordinary inner product calculation $\sum a_i b_i$, which they specify to be *correctly rounded* for all machine a_i and b_i except when overflow or underflow intrude. That is, they implement the inner product as an atomic operation through special hardware or software.

What detracts from the Kulisch-Miranker scheme for general use is the cost of implementing the inner product algorithm. It requires what amounts to a fixed-point buffer to hold the intermediate results of an inner product lest there be massive cancellation, promoting tiny addends to the final result. This buffer is as wide in radix digits as the extent of the exponent range; applied to a format like the 754 double, it would be over 2000 bits wide, virtually infeasible for VLSI implementation today. Moreover, their scheme is *sufficient* to perform reliable computation, aided by devious algorithms; there is no evidence that their scheme is *necessary*, nor that the deviousness of their algorithms is unavoidable.

7. The Less Mathematical Alternative

Despite their appearance of mathematical rigor, the schemes described in the last two sections miss the true goal of computer arithmetic – robust calculations at a price users can afford. The important mathematical idea is *closure* of the arithmetic system, for it is closure that leads to predictability when the inevitable exceptional cases arise. Alas, it is here that the mathematical purity fades and engineering appears, for deciding feasible responses to exceptions involves design tradeoffs. This thesis demonstrates that robust computer arithmetic is feasible in the current technology. The underlying mathematical principle, closure, is clear from the start. The difficulty lies in the careful analysis of all the boundary cases encountered enroute.

8. Arithmetic and Languages – Future Directions

The substance of this thesis, implementation aspects of proposed standard 754, is just part of the story. What has really been specified in 754 is a *programming environment*. Even after all these years, incorporation of the full standard into programming languages has barely started. Chapter 3 touches on some of the issues, but there are many more.

The extended formats are strongly suggested by the standard, and are known to be quite useful, but should they be made available in all languages? Pascal, for example, specifies only one type, real, though enthusiasts would extend the language by adding further ones. Arithmetic in C is based on the PDP-11 float and double types. In C, it is natural to have the 754 extended format play the role double did for the PDP-11, yet one wants both single and double 754 types for data storage and exchange. The prospects for FORTRAN have been discussed by R. J. Fateman [3].

Sometimes language extension to incorporate 754 features causes conflict between two standards; for example, the BASIC standard specifies that underflows should be flushed to zero, prohibiting the more useful gradual underflow of 754. Cases like this led to the plea in Chapter 3 that numerical issues be lifted from language standards and left to the domain of numerical enthusiasts. However, some cases are not so clear. The details of comparisons involving NaNs lie totally in neither camp, so some cooperation will be required.

There is work in progress now to bring the full features of 754 and 854 to people not only in high-speed numerical engines but in commodity calculators and computers as well. Attempts to expand the scope of the working groups to include those responsible for languages have not been too successful, partly because the number of people involved is much greater than the few interested in arithmetic itself. When the 754 effort was begun, the standard was to have stood for twenty years. Now, seven years later, through the cooperation of design, language, and systems people, the ideas spawned in the working group are finally on the verge of dissemination among millions of users.

9. References

[1] W. S. Brown, "A Simple But Realistic Model of Floating-Point Computation," Computer Science Technical Report no. 83, May 1980, revised Nov. 1980; Bell Labs, Murray Hill, N.J., 07974.

[2] W. J. Cody, J. T. Coonen, D. M. Gay, K. Hanson, D. Hough, W. Kahan, R. Karpinski, J. Palmer, F. N. Ris, and D. Stevenson, "A Proposed Radix- and Wordlength-Independent Standard for Floating Point Arithmetic," to appear in *MICRO*, August 1984.

[3] R. J. Fateman, "High-Level Language Implications of the Proposed IEEE Floating-Point Standard," *ACM Transactions on Programming Languages and Systems*, **4**, No. 2, April 1982, pp. 239-257.

[4] W. Kahan, "Why do we need a floating point arithmetic standard?" in preparation.

[5] H. Katzan, Jr., *Invitation to ADA & ADA Reference Manual*, Petrocelli, New York, 1982.

[6] U. Kulisch and W. Miranker, *Computer Arithmetic in Theory and Practice*, Academic Press, New York, 1981.

[7] R. Mansfield, "A Complete Axiomatization of Computer Arithmetic," *Mathematics of Computation*, **42**, April 1984, pp. 623-635.

[8] N. L. Schryer, "A test of a computer's floating-point arithmetic unit," *Computer Science Technical Report No. 89*, Bell Laboratories, Murray Hill, N.J., February 1981.

[9] G. S. Taylor, "Compatible Hardware for Division and Square Root," *Proceedings of the 5th IEEE Symposium on Computer Arithmetic*, Ann Arbor, Michigan, May 1981, pp. 127-134.

[10] A. van Wijngaarden, "Numerical Analysis as an Independent Science," *BIT*, **6**, pp. 66-81, 1966.

CHAPTER 2

The Original P754 Implementation Guide

The following paper, reprinted from *Computer* magazine with the publisher's permission, served as a P754 subcommittee working document until its publication in January 1980. Although nominally a monograph, this implementation guide reflected the many hours of debate about the form of the ultimate proposed IEEE binary floating point arithmetic standard. As published, the implementation guide was compatible with draft 5.11 of the subcommittee's formal proposal; an *errata* sheet at the end brings the guide up to date with draft 8.0, as published in *Computer* in March 1981.

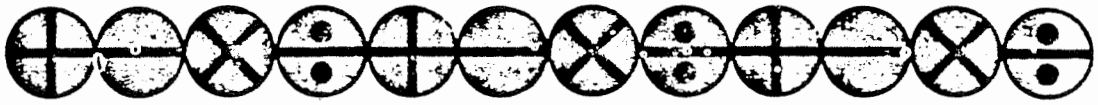
This implementation guide grew out of an earlier document prepared in collaboration with Harold S. Stone and W. Kahan. This author was primarily responsible for an appendix consisting of tables specifying the details of the operations. When it became clear that one inch square table entries would not suffice to describe the arithmetic, the current paper was launched.

Although every attempt was made to represent subcommittee decisions in this implementation guide, it was inadequate for the subcommittee's purposes. Most important, it did not satisfy the stylistic requirements for proposed standards, set forth in the IEEE "blue book". So work was begun on an official version of the proposed standard. W. Kahan, John F. Palmer, Tom Pittman, this author and, later, David K. Stevenson worked on this draft. This implementation guide was published after the proposal had stabilized at draft 5.11.

Draft 10.0 of proposal P754, as voted out of the floating point subcommittee, is fundamentally simpler than draft 8.0 as published in *Computer* magazine and described here. The two principal changes to draft 8.0 were the removal of the projective mode interpretation of ∞ and the warning mode interpretation of denormalized numbers. Draft 10.0 specifies only what were known as the affine and normalizing modes for interpreting ∞ and denormalized numbers, respectively. Among the smaller changes to draft 8.0 were a minor modification to the definition of underflow, a decoupling of the overflow and underflow error flags from their respective traps, and a response to overflow when rounding toward 0 that parallels the response when rounding toward $+\infty$ or $-\infty$, according to the sign of the overflowed result.

The specifications of draft 10.0 are reflected in the pseudo-code description of the the standard in chapter 8. This chapter presents the specifications of draft 8.0; it is one of the few articles describing the proposed standard as it stood for nearly two years (drafts 5.11 to 8.0 were essentially identical), and as it was built in early implementations.

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-

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 $\pm 2^{**}f$, 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

S	E	F

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

Page 2.5 unintentionally left blank.

(That is, the thesis page numbers are incorrect.)

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, $Z1 \leq Z \leq Z2$, 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.

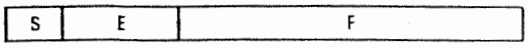
Since the sticky flags need not be set when a trap is to be taken, an implementation may use them to indicate which exceptions have just occurred. A trap handler could determine which exception(s) arose on the aborted operation by checking which have both their sticky and trap-enable flags set, and would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

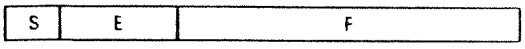
Underflow occurs if the exponent of a result, tested before or after rounding at the implementor's option, lies below the exponent range of the destination field, or if the rounded *extended* or *quad* result of a MULTIPLY or DIVIDE with nonzero, finite operands is normal zero. Note that a product or 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 wraparound 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

S	E	F

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 $\pm\infty$ 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:

sgn	exp	V	N		L	G	R	S

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 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 $1 \leq k \leq 9$ for *single* precision and $1 \leq k \leq 17$ 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 $W \geq 10^{k}+1$, then increment V and go to (5).
If $W = 10^{k}$, then increment V, divide W by 10 (exactly), and go to (7).
If $W \leq 10^{k-1}-1$ 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 $X = U * 10^{W-P}$. ■

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

The changes to Jerome T. Coonen's article in the January 1980 issue of *Computer* (pp. 68-79) are of two types. Those marked (E) correct errors, while the others, marked (U), bring the guide up to date with the most recent draft of the proposal.

(U) Introduction, para. 2, line 2: Replace *Draft 5.11* with *Draft 8.0*. Also update the footnote ** to refer to the March 1981 issue of *Computer*.

(U) §1.1, under Rounding Modes: Delete line (A) and the label "(B)" since all rounding modes are required now.

(E) Table 1: In the formula for represented denormalized numbers the exponent of 2 is incorrect. The correct formula is

$$(-1)^S \times 2^{E-Bias+1} \times (L.F) .$$

(U) §1.5, paragraph beginning *An implementation of . . .*: That first sentence should be shortened to *An implementation of the standard shall support all four rounding modes.*

(U) §1.12: Readers should note that the implementation guide uses *unnormalized* in its traditional sense, that is, describing any number whose leading significant digit is 0; thus denormalized numbers are simply those unnormalized numbers whose exponent is the format's minimum. On the other hand, Draft 8.0 restricts the word *unnormalized* to apply only to numbers whose leading significant bit is zero but which are not denormalized.

(E) §2.7: The special case test

If either operand is an unnormal zero then proceed as in c; otherwise.

should be removed from §e and inserted at the beginning of §f,h,i. Thus §e begins simply *Compute:*.

(E) §2.8: The *Exception* clause of §b,c,f should be changed to *Exception: If in* b, Y *is unnormal zero, proceed as in* a.

(E) §2.9: In §b,c replace *unnormal zero* by *unnormalized*. To §f append *Normalize Z and check for underflow.*

(U) §2.14, para. 1: The sentence beginning *An implementation of. . .*should be shortened to *An implementation of the standard shall support all four rounding modes.*

(E) §2.17: The last word of clause (1) should be changed from *enabled* to *disabled*.

CHAPTER 3

Numerical Programming Environments

The body of this chapter is an article by W. Kahan and this author as published in the book "The Relationship between Numerical Computation and Programming Languages", edited by J. K. Reid. It is reprinted here with the permission of the publisher, North-Holland Publishing Company.

Although the proposed arithmetic standards are intended to specify the total numerical programming environment, they address only indirectly many of the language issues that arise in actual implementations. This chapter is an attempt to defuse some of the conflict between numerical requirements and existing language standards with an argument for the "near" independence of numerical (semantic) and language (syntactic) domains. It is believed that proper partitioning of responsibility for the design of a programming system will lead to the best implementations.

The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments

W. Kahan and Jerome T. Coonen

Mathematics Department
University of California
Berkeley, California 94720
U.S.A.

We can improve numerical programming by recognizing that three aspects of the computing environment belong to intellectually separate compartments. One is the syntax of the language, be it Ada, C, Fortran or Pascal, which gives legitimacy to various expressions without completely specifying their meaning. Another might be called "arithmetic semantics". It concerns the diverse values produced by different computers for the same expression in a given language, including the values delivered after exceptions like over/underflow. The third compartment includes diagnostic aids, like error flags and messages; these too can be specified in language-independent ways. However imperfect, this decoupling should spell out for all concerned the nature of arithmetic responsibilities to be borne by hardware designers, by compiler writers and by operating system programmers.

> "Another of the great advantages of using the axiomatic approach is that axioms offer a simple and flexible technique for leaving certain aspects of a language *undefined*, for example...accuracy of floating point... This is absolutely essential for standardization purposes..."
>
> — C. A. R. Hoare (1969)

Professor Hoare's attitude toward floating point semantics reflects the anarchy that befell commercial floating point hardware early in the 1960's [1], and worsened in the 70's. That anarchy confounded attempts to characterize all floating point arithmetics in one intellectually manageable way. Now there is hope for the 1980's. A new standard for binary floating point arithmetic has been proposed before the IEEE Computer Society, and a radix-independent sequel is in the works. Since the binary standard has been adopted by a broad range of computer manufacturers, including much of the microprocessor industry, we expect numerical programs to behave more nearly uniformly across different computers, and perhaps across different languages as well. A draft of the binary standard, along with several supporting papers, may be found in the March 1981 issue of *Computer* [2-5].

Starting in the 1960's programming language designers came to be the arbiters of most aspects of the programming environment. With control of the programmers' vocabulary, language designers could control fundamental features such as the number of numeric data types available and the extent of run time exception handling. The language even limited the numeric *values* available by constraining the literals in the source text. This is not to say that language designers acted capriciously. They were disinclined to mention any capability not available on all computers. In this respect computer architects have laid a heavy hand on the computing environment. Languages must reflect the least common denominator of available features, and so they tend to vague oversimplifications where floating point is concerned. An extreme case is the new language Ada which, by incorporating W. Stan Brown's very general model for floating point computation [6], pretends that the difference between one computer's arithmetic and another's is merely a matter of a few environmental parameters. But sometimes the

programmer must know his machine's arithmetic to the last detail, especially when trying to circumvent limitations in range or precision. These details, dangling between language designers and computer architects, too often receive short shrift from both. Tying up these loose ends would improve the computing environment.

Of course the computing environment invites numerous improvements, to graphics, file handling, database management and others, as well as floating point and languages. But enhancements to which high-level languages deny access are enhancements destined to die. Those of us working on the proposed IEEE floating point standards have had to face this problem. We believe the solution is a proper division of labor, rather than grand attempts to improve too many aspects of the computing environment simultaneously; the latter way would require impractical coordination. For example, to encourage independent development of programming languages and floating point hardware, we propose that language (syntactic) issues be decoupled from arithmetic (semantic) issues to the extent possible. We present our view of the interplay between syntax, semantics, and diagnostics as parts of the computing environment, and discuss how they interface with each other. Given an adequate interface discipline, we hope that responsibility for these parts can be divided among language designers, numerical analysts, systems programmers, and others. In the past this division has been unclear. Unfortunately, when everybody is responsible, or when nobody is responsible, then everybody can be irresponsible.

Portability

We regard the programming language as just one layer of the computing environment, dissenting from a more traditional view that the language is the environment. What does this mean for program portability? Until very recently, portability of numerical programs was considered to be a quality of source code that could be compiled and run successfully without change on a variety of computers. The issues appeared largely syntactic. For example, programs like the PFORT verifier [7] were developed to check Fortran codes for adherence to a standard for "portable Fortran", their principal task being to weed out various quirks of dialect. Nowadays, we acknowledge that the portability issues go deeper than differences among Fortran dialects. They entail the (semantic) subtleties of over/underflow and rounding that, if ignored, can cause ostensibly portable programs that function beautifully on one machine to fail on another. Programming languages that lack the vocabulary required to address these issues aren't very helpful here. If we cannot "mention" these issues how can we resolve them?

Ideally, the variation of floating point arithmetic from one machine to another should be describable with a few parameters [8] which portable programs could determine through system-dependent environmental inquiries [9]. This scheme works satisfactorily for many programs that do not depend critically upon the finer points of the arithmetic. However, any such parameterization must be based upon an abstract model encompassing simultaneously all current arithmetic engines, some of them disconcertingly anomalous [1, 10]. To insist that this model underlie portable programming is to dump upon programmers the onus to discover and defend against all mishaps the model permits, some of them mere artifacts of generality. This in turn would burden programs with copious tests against subtle (and certainly machine-dependent) thresholds to avoid problems with idiosyncratic rounding and over/underflow phenomena. A programmer who shirks his responsibility to produce robust code obliges the user of his program, possibly another programmer, to unravel a more tangled web. Ultimately, the buck may be passed to users who find either their programs or their computers to be inexplicably unreliable. We doubt that any semantic analog of the PFORT verifier will ever be able to test for robust independence of the underlying arithmetic. Computer arithmetics are too diverse to allow every potentially useful numerical algorithm to be programmed straightforwardly in a fashion formally independent of the underlying

machine.

Portability at the source code level is nice when inexpensive. When not, we are content with "transportability", whereby algorithms can be moved from one environment to another by routine text conversion, possibly with some aid from automation. An algorithm may depend critically upon the underlying arithmetic semantics and upon a system's ability to communicate error reports between subprograms. It is transportable to the extent that the dependencies can be communicated in natural language using mathematical terms, if not in Fortran. We are not advocating yet another programming language. We prefer that programmers accompany their codes with some documentation that explains, and can even be used to verify, how the program handles its interactions with the underlying system. Because computing environments are so diverse, we expect some algorithms to be transportable to only a few systems, not all; this does not undermine the notion of transportability. Essential to transportability is a manageable corpus of information about

- syntax – the programming language to be used,
- semantics – the arithmetic of the underlying computer, including the run-time libraries of functions like cos(), and
- diagnostics – the system's facilities for error reporting and handling,

preferably no more than can fit on a short bookshelf, and yet enough to cover a wide range of manufacturers' equipments.

Syntax

In this paper, *syntax* refers to the expressions in a language – which ones are legitimate and how they are parsed. Issues relevant to numerical calculations include the number of data formats available, how they combine to form arrays and structures, and the order of evaluation in unparenthesized expressions. Languages vary greatly in their provision of numeric data formats, usually called "types". Both Basic and APL have just one numeric type, which is to be used for both integer and floating point calculations; Pascal and Algol 60 have just one real type. Fortran and C have single and double types, although in C all floating expressions are of type double. PL/I programmers may specify the precision of their floating point variables, though they typically map into the single and double types supported by the underlying system. The new language Ada provides syntactic "packages" in which floating types may be defined to correspond to the host system's facilities, but its strong typing prohibits mixing of different user-defined types in expressions without explicit coercions, even if the underlying hardware types are the same.

Expression evaluation is just as varied. For example, in

$$1.0 + 3/2$$

most compilers would recognize the 3 and 2 as integers. Their ratio would be evaluated as the real 1.5 or truncated integer 1 depending upon the strength of the 1.0 to coerce their types. Different Fortran compilers have disagreed in this situation. In Ada such an expression would be illegal unless the 3 and 2 were written with decimal points to indicate that they were real literals. What about the unparenthesized expression

$$A * B + C \quad ?$$

Most languages, like Fortran, evaluate it as if it were written $(A*B) + C$, but APL evaluates $A \times B + C$ as if it were written $A \times (B+C)$. The situation gets more complicated when relational and boolean operators are involved. In Pascal, the attempt to simplify the language by keeping the number of levels of operator precedence small led to some surprises for programmers. For example, because the conjunction $\cap$ has greater precedence than $<$, the expression

$$x < y \cap y < z \ ,$$

used for checking bounds on the variable y, has the bizarre interpretation

$$(x < (y \cap y)) < z$$

which is illegal because of the appearance of the real y as an operand to $\cap$.

Perhaps the widest syntactic liberties are taken by standard C compilers. Expressions of the form

$$a + b + c \ ,$$

where a, b, and c may be subexpressions, are evaluated in an order determined at compile time according to the complexity of a, b, and c. This is so *regardless of parentheses* such as

$$(a + b) + c \ .$$

Such a convention is disastrous in floating point where, say, $(a+b)$ cancels to a small residual to be added into the accumulation c. In such cases all accuracy may be lost if $(b+c)$ is evaluated first at the compiler's whim. The cautious programmer who writes

$$(x - 0.5) - 0.5$$

to defend against a machine's lack of a guard digit during subtraction will always be vulnerable, if not to a C compiler then to an optimizer that collapses the expression into the algebraically, though not numerically, equivalent form $(x - 1.0)$.

To jump the gun a bit, it is clear from the examples above that *syntax constrains semantics*. Syntax also constrains programmers who, C compilers notwithstanding, are well advised to preclude any ambiguity in expression evaluation by inserting parentheses liberally.

Semantics

We concentrate here on arithmetic semantics. That is, after an expression has been parsed — so the computer knows which operations to perform — what does its evaluation yield? Floating point semantics depends vitally on the underlying arithmetic engine. The initiated reader realizes that this is where the real headaches set in. For example, on machines such as programmable calculators where the fundamental constants π and e are available in a few strokes, we might expect

$$(\pi \times e) - (e \times \pi)$$

to evaluate to 0.0 since, *semantically*, we expect multiplication to be commutative despite roundoff. Unfortunately, even this simple statement is not universally true. Different Texas Instruments calculators yield different tiny values for the expression above; and it's not just a matter of machine size and economy, for early editions of the Cray-1 supercomputer exhibited similar noncommutativity.

Another well-known example of murky semantics is the expression

$$X - (1.0 \times X)$$

which is exactly X rather than 0.0 for sufficiently tiny nonzero values X on Cray and CDC computers. On these machines $(1.0 \times X)$ flushes to 0.0 for those tiny X. On some other machines that lacked a guard digit for multiplication, the expression above was nonzero whenever X's last significant digit was odd!

Hardware-related anomalies like these seem to predominate in any serious treatment of arithmetic semantics. Such distractions are what led Professor Hoare to despair about floating point in high-level languages. We will not dig further into the lore of arithmetic anomalies. Interested readers can find an introduction in [1]. The technical report [10] studies the overall impact of anomalies and compares two approaches to improvement.

Arithmetic semantics is not restricted to simple operations. In languages like Basic that include matrix operations, assignments like

$$MAT\ X\ =\ INV(A)\ *\ B$$

are allowed. As users might expect, most implementations evaluate $(A^{-1})*B$ (approximately), following the strict mathematical interpretation of the formula. However, more robust systems by Tektronix and Hewlett-Packard use Gaussian elimination to solve the linear system $AX = B$ for X, thereby obtaining a usually more accurate X that is guaranteed to have a residual $B - AX$ small compared with $|B| + |A|\cdot|X|$. If A is close enough to singular, the subexpression $INV(A)$ may be valid or not depending upon good or bad luck with rounding errors – on all machines except the Hewlett-Packard HP 85. All machines solve $(A + \Delta A)X = B$ with ΔA comparable to roundoff in A though possibly differing from column to column of X. The HP 85 further constrains ΔA to guarantee that $(A + \Delta A)^{-1}$ exists. Thus it has no "SINGULAR MATRIX" diagnostic. Consequently, a program using inverse iteration to compute eigenvectors always succeeds on the HP 85 but on other machines is certain to fail for some innocuous data. Is such a program, using a standard technique, portable or not? Who is to blame if it is not?

Arithmetic exceptions such as over/underflow and division by zero fit into our informal notion of semantics when they are given "values". We take this view in spite of a current trend among authors to consider exceptions under a separate heading *pragmatics*. This trend is understandable, given the variety of exception handling schemes across different hardware. Consider for example the expression 0.0/0.0 . When they are to continue calculation (i.e. without a trap) CDC, DEC PDP/VAX-11, and proposed IEEE standard machines stuff a non-numeric error symbol in the destination field. This symbol is then propagated through further operations. Most other machines just stop, forcing program termination. At least one will store the "answer" 1.0.

Dividing zero by itself is usually bad news within a program, so the diversity of disasters that arise on various machines is not too surprising. A quite different situation arises with the exponentiation operator in Y^X. Since this is part of the syntax of several languages, for example Fortran, Basic, and Ada, responsibility for its semantics has been taken by language implementors. Of the many problems that arise we will consider just one: what is the domain of Y^X when both X and Y are real variables? Consider the simple case $(-3.0)^{3.0}$, which is:

−27.0	...on very good machines,
−26.999...9	...on good machines,
TERMINATION	...on bad machines,
undefined	...on cop-outs,
+27.0	...on *very* bad machines.

Why this bizarre diversity of semantics? Although for arbitrary X the expression Y^X may have no real value when Y is negative, the particular case above is benign because X has an integer value 3.0. Thus restricting the domain of Y to nonnegative numbers is unnecessarily punitive. We recommend that, should X be a floating point Fortran variable with a nonzero integer value,

$$Y\ **\ X\ =\ Y\ **\ \mathrm{INT}(X)\ .$$

This cannot hurt Fortran users, but will help the Basic programmer (and the conversion of programs from Basic) because most implementations of Basic, with just one numeric data type, cannot distinguish the real 3.0 from the integer 3 in the exponent. This recommendation costs extra only when Y is negative. On the other hand, if Y is 0.0 we distinguish $Y^{0.0}$, which is an error, from $Y^0 = 1.0$ which mathematics makes obligatory. Note that none of these issues are language issues, though until now they have been settled by language implementors. Ideally, these responsibilities should be lifted from language designers and implementors, and

borne by people like the members of IFIP Working Group 2.5.

The point of this digression into the murk of pragmatics was to indicate that the current situation in exception handling is the result of a host of design flaws rather than inherent difficulties. We object to the connotation "pragmatics" carries with it of acquiescence to inevitable hazards. We prefer to capture all semantics, including the anomalies, under one heading even if this entails a different semantics for each different implementation of arithmetic. This exposes rather than compounds a bad situation.

A notably clean and complete arithmetic semantics is provided by the proposed binary floating point standard. The IEEE subcommittee responsible for the proposal set out to specify the result of *every* operation, balancing safety against utility when execution must continue after an exception. Even a cursory glance at the proposal indicates the extent to which exception handling motivated the design:

- Signed ∞ for overflow and division by 0.0.
- Signed 0.0 to interact with $\pm\infty$, e.g. $+1.0/-0.0 = -\infty$.
- NaN – not a number – symbols for invalid results like 0.0/0.0 and $\sqrt{-3}$.
- Denormalized numbers – unnormalized and with the format's minimum exponent – to better approximate underflowed values.
- Sticky flags for all exceptions.
- Optional user traps for alternative exception handling.

These features promote comprehensible semantics for "standard" programming systems.

Diagnostics

After syntax and semantics, the third aspect of the numerical programming environment is the set of execution time diagnostic aids. They may be roughly divided into anticipatory and retrospective aids, and according to whether they find use during debugging or during (robust) production use.

The principal anticipatory debugging aid is the breakpoint for control flow and, when the hardware permits, for data too. Some systems can monitor control or data flow according to compiler directives inserted in a program. Retrospective debugging aids include the familiar warnings and termination eulogies, as well as the more voluminous memory dumps and control tracebacks. Systems with sticky error flags can list those still standing when execution stops – in a sense they signal unrequited events.

For the production program that would be robust, and perhaps even portable, the situation is not so clear. Because most current systems provide neither exception flags (such side effects are anathema to some language designers) nor error recovery, a program – if it is not to stop ignominiously on unusual data – must include precautionary tests to avoid zero denominators and negative radicands, and tests against tiny, but carefully chosen, thresholds to ward off the effects of underflow to zero. The lack of flags can force the use of explicit error indicators in subprogram argument lists to communicate exception conditions. The languages Basic, PL/I, and Ada allow for anticipatory exception handlers (e.g. ON <condition> ... in PL/I) but do not allow the exception handler to discover anything about the exception beyond a rough category into which it has been lumped, thereby making an automatic response by the program very cumbersome.

Another variety of anticipatory diagnostic aid is available through an option in the proposed floating point standard. It is essentially an extension of the PL/I "on-condition" except that it is outside any current language syntax. This feature, which might be called trap-with-menu, allows the programmer to preselect from a small list of responses an alternative to the default response. By devising the menu

carefully, we should be able to give the user sufficient flexibility without having to cope with a voluminous floating point "state" at the time of the exception.

The Syntactic–Semantic Interface

From the point of view of the numerical analyst, the semantic content of programming languages is given by the following list.

- What are the numeric types, and what is their range and precision?
- Which numeric types are assigned to anonymous variables like intermediate expressions, converted literals, arguments passed by value,...?
- Which numeric literals are allowed, and are they interpreted differently in the source code than the IO stream?
- Which basic arithmetic operations are available, and what is in the library of scientific functions?
- Is there a well-understood vocabulary reserved for the concepts and functions we need, and defended against collision with user-defined names?
- What happens when exceptions arise? How can error reports be communicated between subprograms?
- Is there a way to alter the default options (for, say, rounding or handling of underflow) by means of global flags?

These are among the knottiest issues in numerical computation. But, to a large extent, they can be freed from the more conventional language issues and thus resolved within the numerical community. Only questions about data types and the change of control flow on exceptions are necessarily tied to language syntax.

Consider a hypothetical language with only skeletal numerical features. Assume that integer types and arithmetic and character strings are "fully" supported. The language supports single and double real variables, pointers to them, and allows real variables to be embedded in arrays and structures. There is also provision for functions returning real values, and for real parameters passed either by value or reference. But the *only* operation on real types is assignment of a single value to a single variable, and of a double value to a double variable.

To be useful numerically, this hypothetical language would require a support library providing the basic arithmetic operations as well as the usual complement of elementary functions. But because each operation more complicated than a straight copying of bits would result only from an explicit function call, the programmer would in principle have complete control of the arithmetic semantics (by choosing a suitable library). As an example, consider the evaluation of the inner product of the single arrays $x[]$ and $y[]$ using a double variable for the intermediate accumulation to minimize roundoff:

```
double_precision temp_sum;
temp_sum := DOUBLE_LITERAL( "0.0" );
for i in 1..n do
      temp_sum := DOUBLE_SUM( temp_sum,
            SINGLE_TO_DOUBLE_PRODUCT( x[i], y[i])); od
inner_product := DOUBLE_TO_SINGLE( temp_sum );
```

Even this simple example exposes many of the questions that arise in numerical programs. Would the constant 0.0 require a special notation (such as 0.0D0) to be assigned to a double variable? In a more conventional rendition of the program the inner loop would involve a statement of the form

```
temp_sum := temp_sum + x[i]*y[i];
```

Would the product be rounded to single precision before the accumulation into *temp_sum*, destroying the advantage of double precision?

Semantic Packages

The skeleton language above may be unambiguous, but it is clearly much too cumbersome for calculations involving complicated expressions. What we must do is bridge the gap between the handy syntactic expression $x[i]*y[i]$ and the semantically well-defined

SINGLE_TO_DOUBLE_PRODUCT($x[i]$, $y[i]$) .

We propose to do this through so-called semantic packages.

It may be a sign of progress that the new language Ada comes very close to suiting our needs. Although Ada incorporates the Brown model for arithmetic by providing a set of predefined attributes for each real type available to the programmer, this is in general insufficient for programs that would be robust. More important for us, Ada allows the overloading and redefinition of the infix operators +, −, etc. and in so doing provides the *explicit* connection between the operators and the real hardware functions they represent. The semantic packages, corresponding directly to the (syntactic) **packages** construct in Ada, could contain exact specifications of the arithmetic functions (which are actually implemented in hardware). Thus there would be a semantic package for each basic architecture, for example IBM 370, DEC PDP/VAX-11, and the proposed IEEE binary standard. Some semantic packages could be more general, encompassing several machines whose arithmetic is similar enough that a few environmental inquiries supply all the distinction that is necessary for a wide range of applications. For example, one such package might include IBM 370, Amdahl, Data General MV/8000, HP 3000, DEC PDP/VAX-11 and PDP-10, relegating TI, CDC 6000, Cray 1 to another.

Our attempt to force the gritty details of arithmetic semantics upon programmers may dismay readers who embrace the modern trend to elevate the programming environment above machine details. Such an attempt is made within Ada, by means of a small set of predefined attributes associated with each real type. We have already explained that this is not enough; sometimes the program that would be robust must respond to machine peculiarities that defy simple parameterization. The report [10] on why we need a standard contains several examples.

An effort to "package" arithmetic semantics within various programming languages may seem impossible. For example, the details of floating point, especially in the proposed IEEE standards, involve global flags to indicate errors, and modes to determine how arithmetic be done. In Fortran, such state variables may be defined as local data within the standard library functions whose job is to test and alter the flags, although the actual implementation involves collusion with the hardware flags. This is not a complete formalization, since Fortran provides no way to describe the connection between the flags and the arithmetic operations. Current trends in language design eschew error flags as side effects of the arithmetic operations (functions). Modes and flags seem to violate the principle that all causes and effects of expression evaluation should be visible within that expression. Perhaps surprisingly, Ada again provides us with the desired facility – but without excessive or expensive generality. In accordance with the Steelman requirements of the United States Department of Defense, Ada permits side effects " limited to own variables of encapsulations". This is exactly our intention in using semantic packages to describe arithmetic.

Optimization

Any treatment of floating point semantics must deal with that favorite whipping boy, the code optimizer. We considered a most extreme example above, in which C compilers would calculate floating sums like

$$(a + b) + c \quad ,$$

without regard to the parentheses, in whatever order makes best use of the register file. This is simply a mistake in the language design.

Not all anomalies are so clear-cut. Some questions arise when, as in architectures suggested by the proposed IEEE standard, extended registers with extra precision and range beyond both single and double types are used as intermediate accumulators. Consider the typical code sequence

```
x := a * b;
y := x / c;
```

in which all variables are assumed to be of type single. If $(a*b)$ were computed in an extended register, should that value or the single value x be used in the evaluation of y? Efficiency dictates the former, saving one register load and lessening the risk of spurious over/underflow. But common sense dictates the latter, so that what the programmer sees is what the programmer gets.

A similar situation arises in inner product calculations of the type discussed above. Consider the loop

```
double_precision temp_sum;
temp_sum := 0.0;
for i in 1..n do
        temp_sum := temp_sum + x[i]*y[i]; od
inner_product := temp_sum;
```

in which, like the earlier example, all variables are single except for the double *temp_sum*. The fully "optimized" compiler might run this loop with just two extended registers, one to compute the products $x[i]*y[i]$ and one to accumulate *temp_sum*, thereby avoiding $(n-1)$ register loads and stores by simply keeping *temp_sum* in a register. Alas, the programmer asked for a double precision intermediate, not extended, so such optimization is precluded.

The moral of these examples is that declared types must be honored. Also, the type assigned by the compiler to anonymous variables must be deducible syntactically, or, better, it should be under the programmer's control. The alleged optimizations above were disparaged because named variables were replaced surreptitiously by extended counterparts that happened to be in registers. This is not to say that extended evaluation is unhealthy; on the contrary, extended temporaries can reduce the risk of spurious over/underflow or serious rounding errors, and therefore should be used for anonymous variables. But the advantage of extended is lost if languages prevent programmers from requesting it for declared temporaries. The expression

$$temp\_sum + x[i]*y[i]$$

in the loop above would best be computed entirely in extended before the store into *temp_sum*. These facilities for extended expression evaluation are not unique to the proposed IEEE standard; the benefits of wide accumulation were realized in the earliest days of computing. The Fortran 77 standard includes some intentionally vague language about expression evaluation in order not to prohibit extended intermediates, and the Ada standard, which seems to avoid some problems by strict typing and requirements for explicit type conversions in programs, uses a so-called *universal_real* type (at least as wide as all supported real types) for the evaluation of literal expressions at compile time.

The use of an extended type for anonymous variables is prone to one class of problems. When real values or expressions may be passed by value to subprograms there may be a conflict between the implicit type of the expression and the declared type of the target formal parameter. This problem arises in current implementations of the language C, which supports both single and double types but specifies that all real expressions are of type double. Suppose that a C program contains the statement

```
y := f(a*b/c);
```

where all variables are of type float (single) and the function $f()$ is defined by

```
float f(x)
  float x;
{ ..... }
```

How can the type of the expression (a*b/c) be double while the type of the formal parameter x is float? C resolves the discrepancy by silently countermanding the declaration of x and replacing **float** by **double**. Once again, what you see is not what you get. This use of wider intermediates, exploiting the PDP-11 floating point architecture, is exactly analogous to one use of extended registers. Though it is efficient and straightforward to implement, it is not acceptable.

Conclusion

We have cited examples to show that progress in numerical computing has been slowed by questionable decisions in the design of computing languages and systems. We have suggested a rough division into three categories, syntax, semantics and diagnostics, so that the difficult issues could be resolved by those most qualified – and most profoundly impacted. IFIP Working Group 2.5 might well take responsibility for the interfaces with semantics. Ideally their efforts will lead to fully specified environments for which reliable numerical software can be derived, possibly automatically, from algorithms expressed in a mathematical form if not already in a programming language. Programming then becomes a three phase translation involving the language (syntax) to be used, the underlying arithmetic engine (semantics), and the host system (diagnostics). We acknowledge that these categories are not completely independent, and that the boundaries between them cannot be drawn precisely, at least not yet. Nonetheless, we remain convinced that those boundaries must be drawn if we are to bring the required expertise to bear on the current morass.

Acknowledgement

This report was developed and originally typeset on a computer system funded by the U. S. Department of Energy, Contract DE-AM03-76SF00034, Project Agreement DE-AS03-79ER10358. The authors also acknowledge the financial support of the Office of Naval Research, Contract N00014-76-C-0013.

References

[1] Kahan, W., "A Survey of Error Analysis," in: *Information Processing 71*, (North-Holland, Amsterdam, 1972) 1214-1239.

[2] "A Proposed Standard for Binary Floating-Point Arithmetic," Draft 8.0 of IEEE Task P754, with an introduction by D. Stevenson, *Computer*, **14**, no. 3, March (1981) 51-62.

[3] Cody, W. J., "Analysis of Proposals for the Floating-Point Standard," *Computer*, **14**, no. 3, March (1981) 63-68.

[4] Hough, David, "Applications of the Proposed IEEE 754 Standard to Floating-Point Arithmetic," *Computer*, **14**, no. 3, March (1981) 70-74.

[5] Coonen, Jerome T., "Underflow and the Denormalized Numbers," *Computer*, **14**, no. 3, March (1981) 75-87.

[6] Brown, W. S., "A Simple But Realistic Model of Floating-Point Computation," to appear in *ACM Transactions on Mathematical Software*, 1981.

[7] Ryder, B. G., "The PFORT Verifier", *Software – Practice and Experience*, **4** (1974) 359-377.

[8] Sterbenz, P. H., *Floating-Point Computation* (Prentice-Hall, Englewood Cliffs, N.J., 1974).

[9] Brown, W. S. and S. I. Feldman, "Environment Parameters and Basic Functions for Floating-Point Computation," *ACM Transactions on Mathematical Software*, **6** (1980) 510-523.

[10] Kahan, W., "Why do we need a standard for floating point arithmetic?", Technical Report, University of California, Berkeley, CA, 94720, February (1981).

CHAPTER 4

Proposed Floating Point Environmental Inquiries in FORTRAN

This is a proposal for floating point environmental inquiries in Fortran. It was drafted by W. Kahan, J. Demmel, and J. T. Coonen. In February 1982, the authors presented it to the ANSI X3J3 Fortran Standards Committee on behalf of IEEE Working Groups 754 and 854, which are developing binary and decimal standards for floating point arithmetic. Although it is intended for inclusion in the next Fortran standard, known for the moment as Fortran 8X, the scheme is designed to be compatible with Fortran 77 implementations.

1. Portability

Fortran is usually associated with high speed computation on mainframes and minicomputers. And numerical Fortran codes are considered portable when they behave reasonably across this class of machines. Portability has been achieved by defining parameters that demarcate the boundaries of the various machines' arithmetics. The Bell Labs PORT Library [4] is just one significant effort. More recently, W. S. Brown has devised a model of arithmetic [2] encompassing nearly all existing arithmetic engines. He captures their diversity in an abstract, parameterized machine which is in some sense the least common denominator of all existing machines. J. L. Blue's program [1] to compute the Euclidean norm of a vector exemplifies the programming style that goes with Brown's model -- and the difficulty of writing such universally portable codes.

But the software situation is changing somewhat. Proposed IEEE standard P754 for binary floating point arithmetic [5] is gaining acceptance in the computing industry. For example, significant hardware support for the standard is already available from one microprocessor manufacturer (Intel) and is expected soon from several others. What is important is that these new processors will not be restricted to a few in-house systems. Rather, they will be embedded in computer systems marketed by diverse companies, and they will perform at the levels of today's minicomputers. The P754 proposal, and its decimal sequel P854, provide features lacking in most previous machines, features such as sticky exception flags for errors, a choice of responses to exceptions like over/underflow, and a choice of direction of rounding. To exploit these features programmers need access to them in high-level languages. And the means of access must be standardized for each language so that codes can, *with minimal extra effort*, be made portable across the entire family of "standard" systems.

2. Design Constraints

This proposal serves two rather different needs. Following the lead of others who have worked in this area, notably W. S. Brown, W. J. Cody, S. I. Feldman, B. Ford, and B. T. Smith, it provides access to machine parameters which permit programming in a style that defends against the peculiar ways machines handle roundoff and exceptions like over/underflow. This facilitates the first kind of portability above. On the other hand, the 754/854 proposals are recognized as important enough to warrant functions to access their features, even though those features are not universal.

The capabilities in this proposal are needed in Fortran 77 now. Therefore the proposal has been devised, particularly in its syntax, to be compatible with existing Fortran 77 systems. And, in order that the proposal be implementable at low cost on a broad range of Fortran engines, it has been designed to have negligible impact on compilers. For example, no new reserved words like .HUGE. are used. Instead, all inquiries are made through intrinsic functions in the same domain as mathematical functions like COS and TAN. This concentrates both the effort and the responsibility where they belong.

Ideally, an inquiry mechanism should be invisible to programmers not interested in it, and readily available to those who are. Since there is no simple "include" mechanism in Fortran 77, no convenient way exists to reserve a named COMMON area with numerous PARAMETERs and variables related to the environment. The prospect that programmers might enter the relevant definitions without error (or complaint) is clearly hopeless. So the inquiries cannot depend on predefined variables or values.

With function names restricted to six characters, and no protection for the programmer whose names may collide with system routines, parsimony is an issue. This proposal consists of a minimal yet useful set of functions from which programmers may easily deduce all the commonly used parameters.

Except for scaling by a power of the machine radix, which is deliberately specified to be fast, environmental inquiries tend to appear not in critical loops but at milestones before and after units of computation. Thus their speed is not important, although in many cases "smart" compilers could replace calls to environmental intrinsics with simple in-line code. Coupled

with the speed issue is exception handling, since there is a price for checking special cases. The inquiries specified here are intended to follow a system's overall conventions for exception handling. This is consistent with the 754/854 philosophy, though it is more restrictive than, say, the proposal of Brown and Feldman [3] which leaves some boundary cases undefined.

What makes this proposal more complicated than previous schemes is its conscientious attempt to deal with boundary cases that jeopardize the robustness and portability of programs. Three classes of funny numbers lie beyond the frontier of Brown's model:

Many computers support a variety of *tiny numbers* that correspond roughly to underflowed values. These might be denormalized numbers as in the 754/854 arithmetics, signed UN symbols that stand for the positive and negative intervals of numbers too small to represent, or even a whole range of "partial underflows" that behave like 0 in some operations but not in others, as on the Cray-1 and the CDC 7600. Some systems can signal underflow, some cannot. Underflow is discussed at length in [5 pp. 75-87].

Some computers support *huge numbers* that correspond roughly to overflowed values. The numbers might be $\pm\infty$ symbols or, as on the Cray-1, a family of numbers that behave very much like $\pm\infty$ in some but not all operations. Systems differ as to when and how overflow will be signaled.

Many computers reserve a set of *non-numbers* to accommodate various invalid operations and, sometimes, overflows and divisions by zero. Depending on the system, the non-numbers (or "NaNs" as they are called in 754/854) may either propagate through or trigger an exception in subsequent operations.

3. Outline of the Approach

Nine intrinsic functions are put forward in the following sections. Those that return floating point values are listed as generics: that is, their return type is determined by their operands in the same way as for intrinsics like COS and TAN. This intrudes very little into the compiler.

Several of the functions accept an argument that selects from among a list of options. The ideal mechanism for this selection would be a compiler-supported enumerated type. However, there is no such thing in Fortran 77, and an artificial version using INTEGER variables is either too cumbersome (for lack of predefinition) or too cryptic. So the functions use six-character strings to specify choices in a reasonably mnemonic fashion.

Only two of the functions are specific to the 754/854 proposals. They concern modes (like the direction of rounding) and flags (to signal errors like over/underflow), features of the 754/854 proposals that, while available in some form or other on various older machines, have never been considered part of the environment available to portable programs. The mode and flag functions are designed to be extensible to other systems, which are accommodated by augmenting the list of arguments recognized by the intrinsics, rather than by adding new names to the system library. On any given system, meaningless intrinsics would be omitted from the library, so that an attempt to use them would cause a fatal error during compilation. However, meaningless arguments to legitmate intrinsics must be caught at execution time. In any case, programmers will not be fooled about what the environment really is.

4. Huge and Tiny Numbers

Functions HUGE and TINY return floating point values near the limits of a machine's range, according to a string parameter FLAVOR.

```
FUNCTION HUGE( X, FLAVOR)
  real type X
  CHARACTER*6 FLAVOR
```

X is a dummy parameter whose value is ignored but whose format determines the format of the return value.

FLAVOR	return value
'MACH'	biggest ordered value, possibly +OV symbol or +∞ (even though the machine may not permit the value to be used in subsequent comparisons)
'THRESH'	biggest finite value that can be used in or result from some arithmetic operations without triggering overflow, though it may behave anomalously in some other operations
'MODEL'	biggest number that can be used safely in Brown's model

Typically, the 'MACH' and 'THRESH' values would differ only on systems that support symbols for values outside the range of finite representable numbers. Some machines support signed ∞, or something very like it. Another possibility is an overflow symbol OV that stands for the interval strictly between ∞ and the largest finite representable number. 'THRESH' and 'MODEL' values would differ only when the Brown model penalizes the system some units of exponent range due to unseemly behavior. Three kinds of HUGE may seem extravagant at first sight, but the fact is that the corresponding return values from HUGE really do vary on some machines. The following table gives the parameter values for the double formats of three sample architectures.

	return value from HUGE(X, FLAVOR)		
FLAVOR	P754 double	VAX-11 D-format	Cray-1 double
'MACH'	$+\infty$	1.7×10^{38}	$+\infty$
'THRESH'	1.8×10^{308}	1.7×10^{38}	$2^{8191}\times(1-2^{-96}) \approx 5.4\times10^{2465}$
'MODEL'	1.8×10^{308}	1.7×10^{38}	$2^{8190}\times(1-2^{-94}) \approx 2.7\times10^{2465}$

```
FUNCTION TINY( X, FLAVOR)
  real type X
  CHARACTER*6 FLAVOR
```

As above, X is a dummy parameter whose value is ignored but whose format determines the format of the return value.

FLAVOR	return value
'MACH'	smallest positive value, possibly a +UN symbol or a denormalized number
'THRESH'	smallest positive value that can be used in or result from some arithmetic operations without triggering underflow, though it may behave anomalously in some other operations
'MODEL'	smallest positive number that can be used safely in Brown's model

This function is similar to HUGE. TINY(X, 'MACH') is the smallest representable positive value in the format of X. It could be a symbol, UN, that behaves arithmetically like the interval between 0 and the tiniest representable magnitude. On some systems, notably 754/854, TINY(X, 'MACH') is the smallest of a family of tiny numbers, beneath the stated underflow threshold, designed to make underflow gradual rather than abrupt. As above, the difference between the 'THRESH' and 'MODEL' values depends on the quality of arithmetic near the bottom of the exponent range. The following table gives the parameter values for the double formats of three sample architectures.

	return value from TINY(X, FLAVOR)		
FLAVOR	P754 double	VAX-11 D-format	Cray-1 double
'MACH'	4.9×10^{-324}	2.9×10^{-39}	$2^{-8193} \approx 3.0\times10^{-2465}$
'THRESH'	2.2×10^{-308}	2.9×10^{-39}	$2^{-8193} \approx 3.0\times10^{-2465}$
'MODEL'	2.2×10^{-308}	2.9×10^{-39}	$2^{-8100} \approx 4.7\times10^{-2439}$

5. Successor Functions

The NEXT and NEXTM functions accept two floating point arguments and return, respectively, the next machine or Brown model number after the first argument toward the second.

```
FUNCTION NEXT( SOURCE, TARGET)  ...next machine number
FUNCTION NEXTM( SOURCE, TARGET)  ...next model number
  real type SOURCE, TARGET
```

The semantics of NEXT were introduced in the appendix to Draft 8.0 of proposal P754. The result is well defined so long as SOURCE and TARGET are ordered as <, =, or > (they aren't numerically ordered if either is a NaN). When they are equal, NEXT returns that value, and NEXTM returns the nearest model number, rounded according to machine convention. When the values SOURCE and TARGET are unordered, the operations NEXT and NEXTM are invalid, and a NaN is returned. Interestingly, NEXTM is the only function strictly related to Brown's model that had to be introduced into this system in order to support his model fully.

6. Radix Logarithm

The function LOGB, when passed an argument of the form $\pm b^e d.ddd...d$ where b is the machine radix and the d's are radix-b digits, returns the integer value of the exponent e in the floating point format of the argument.

```
FUNCTION LOGB( X)
  real type X
```

There are several special cases:

X	LOGB(X)	
±0	−HUGE(X, 'MACH')	
±∞	+∞	...on machines with an ∞ symbol
NaN	X	...on machines with nonnumber symbols

When X is not normalized, LOGB returns the exponent of X if it would be treated as unnormalized in subsequent arithmetic, or the exponent of X as though prenormalized if X would be prenormalized in subsequent arithmetic. Because of the extreme and exceptional cases, and for convenience in some approximations, the return value, although typically an integer value, is in the floating format of X. In many contexts a programmer will use

```
INT( LOGB( X ) )
```

but this is not expected to appear in critical looping code, so the extra call to INT is a negligible added cost. This also has the advantage of keeping the messy exception handling of INT (at least, a conscientious rendition thereof) from being duplicated in LOGB.

7. Scaling

The function SCALB is the companion to LOGB.

```
FUNCTION SCALB( X, FACTOR)
  real type X
  INTEGER FACTOR
```

It returns the value $X \times b^{FACTOR}$, where b is the machine radix. The parameter FACTOR is specified as an integer so that SCALB can be fast, since it is often used in inner loops. Even so, SCALB is expected to conform to system conventions for dealing with exponent over/underflow, which must not occur

unless the final value lies out of range. Underflows are denormalized [5] on systems that underflow gradually; some systems underflow to zero or TINY(X, 'MACH'); some systems also set an error flag. Overflows may be set to HUGE(X, 'MACH') or, as is more usual, may stop computation altogether; there may also be an error flag. Note that because the INTEGER type may be much wider than the exponent field of X, severe over/underflow is possible.

8. Classification

The function CLASS returns an integer indicating the "character" of the floating point argument. This is helpful in filtering special operands.

```
INTEGER FUNCTION CLASS( X)
  real type X
```

The sign of the returned integer indicates the sign of X, even if the sign has no relevance (such as the sign of 0, usually taken to be +, on systems with no −0, or the sign of NaNs). The magnitude of the returned value is defined from the table:

magnitude	X
1	zero
2	finite, nonzero, normalized number
3	∞
4	denormalized number, a la 754/854 proposals
5	unnormalized number, possibly with zero significand
6	quiet NaN -- propagates without exceptions
7	signaling NaN -- triggers exception on attempted use
8	UN symbol, or numbers between TINY(X, 'MODEL') and TINY(X,'MACH')
9	OV symbol, or numbers between HUGE(X, 'MODEL') and HUGE(X,'MACH')
...	...

The arbitrary breakdown above is intended to facilitate branching with a case statement (computed GOTO in Fortran), or the IF-THEN alternative. The

commonest cases appear at the top of the list. Although specified for a wide class of numeric entities, a particular implementation of CLASS will return only the values pertinent for the given machine.

9. Exception Flags

Some arithmetics, in particular 754/854, provide flags which are set when the corresponding floating point exception arises, and which are cleared only at the program's request. The function FLAG gives a programmer access to such flags. It returns the current setting of the flag, and allows the programmer the option of altering the flag. Thus, the exception flags appear to the programmer as implicitly defined global variables, although they can be accessed only through the function FLAG.

```
INTEGER FUNCTION FLAG( TYPE, VALUE)
  CHARACTER*6 TYPE
  INTEGER VALUE
```

where TYPE is one of

TYPE	exception flag affected
'UNFLOW'	underflow
'OVFLOW'	overflow
'INVALD'	invalid operation
'DIVZER'	(nonzero) / zero
'INEXCT'	inexact result
...	... ...

The return value of 0 indicates that the flag is *off*; and any nonzero value indicates that the flag is *on*. A nonzero flag will typically contain some system-dependent reference to what happened and where. Thus there are only two portable uses of a value returned from FLAG: test whether or not it is zero, and save the value for subsequent restoration. FLAG sets the selected flag to VALUE unless the VALUE argument is omitted from the func-

tion call, in which case the flag is not altered.

A program that deals with an exception such as underflow will use FLAG with a VALUE of 0 to clear the 'UNFLOW' flag so as not to distract any following code. It may use FLAG without the VALUE argument to simply test the flag during its calculation. A subprogram that deals with its own exceptions may use FLAG to save the setting of pertinent flags on entry and restore them on exit.

10. Modes

Modes are provided by some systems as a way for a program to control details, for example exception handling, in subsequent operations. The character function MODES allows the programmer to test and possibly alter arithmetic modes in the host machine, in much the same way that FLAG handles flags. All settings are given as six-character strings.

```
CHARACTER*6 FUNCTION MODES( TYPE, VALUE)
  CHARACTER*6 TYPE, VALUE
```

where TYPE and VALUE are given in the following table, which has been tailored for 754/854 systems.

TYPE	VALUE				
'ROUND'	'NEARST'	'ZERO'	'PINF'	'MINF'	'KEEP'
...	...	...	...	...	...

The VALUE 'KEEP' allows the programmer to test a mode without altering it. The modes listed here pertain to the 754/854 standards. The four options for rounding are to nearest, toward zero (chopping), toward $+\infty$, and toward $-\infty$. Many existing systems offer both chopping and rounding to nearest (after a fashion) but usually they are controlled not by processor modes but by extra "rounding" instructions; the use of MODES in such

systems would amount to a compiler directive, if the use were allowed at all.

If user traps are to be provided they might be implemented as mode settings, though the handler address and its input/output parameters require further discussion. Since general traps are not expected to be portable constructs, even across 754/854 systems, this is not discussed further here.

11. Relation to Brown's Model

This section relates the environmental inquiries presented here with those Brown and Feldman proposed [3] in connection with Brown's model. On a machine of radix b, Brown considers a system of *model numbers* consisting of zero and all numbers of the form

$$x = fb^e$$

where

$$f = \pm(f_1 b^{-1} + \cdots + f_p b^{-p}), \quad f_1 = 1, \cdots, b-1,$$
$$f_2, \cdots, f_p = 0, \cdots, b-1,$$

and

$$e_{\min} \le e \le e_{\max}.$$

The following table gives the model parameters and the computational procedures of Brown and Feldman in terms of the present proposal.

```
          RADIX = INT ( SCALB ( 1.0, 1 ) )  ... b
   MODELEPSILON = NEXTM ( 1.0, 2.0 ) − 1.0 ...maximum relative spacing
                  in model
      PRECISION = 1 - INT ( LOGB ( NEXT ( 1.0, 2.0 ) − 1.0 ) )  ...minimum
                  number of radix-b digits
MODELPRECISION = 1 - INT ( LOGB ( MODELEPSILON ) )  ...minimum
                  number of radix-b digits, including a possible penalty
                  if rounding is strange
     MODELHUGE = HUGE ( X, 'MODEL' )  ...biggest number in Brown's
                  model, including a possible penalty if overflow is
                  strange
          EMAX = INT ( LOGB ( MODELHUGE ) ) + 1  ...biggest exponent
     MODELTINY = TINY ( X, 'MODEL' )  ...smallest number in Brown's
                  model, including a possible penalty if underflow is
                  strange
          EMIN = INT ( LOGB ( MODELTINY ) ) + 1  ...smallest exponent
   exponent(X) = INT ( LOGB ( X ) ) + 1
     scale(X, E) = SCALB ( X, E )
   fraction(X) = SCALB ( X, −exponent( X ) )
synthesize(X, E) = SCALB ( fraction( X ), E )
          α(X) = synthesize( 1.0, max ( EMIN, 1 - MODELPRECISION +
                  exponent( X ) ) )  ...if X is nonzero
               = MODELTINY  ...if X is 0
          β(X) = synthesize( ABS ( X ), MODELPRECISION )
```

12. References

[1] Blue, J.L., "A Portable Fortran Program to Find the Euclidean Norm of a Vector," *ACM Trans. Math. Soft.*, **4**, 1, March 1978, 15-23.

[2] Brown, W.S., "A Simple but Realistic Model of Floating-Point Computation," *ACM Trans. Math. Soft.*, **7**, 4, December 1981, 445-480.

[3] Brown, W.S. and S.I. Feldman, "Environment Parameters and Basic Functions for Floating-Point Computation," *ACM Trans. Math. Soft.*, **6**, 4, December 1980, 510-523.

[4] Fox, P.A., A.D. Hall, and N.L. Schryer, "The PORT Mathematical Subroutine Library," *ACM Trans. Math. Soft.*, **4**, 2, June 1978, 104-126.

[5] "A Proposed Standard for Binary Floating Point Arithmetic," (Draft 8.0) and supporting articles, *Computer*, **13**, 3, March 1981, 51-87.

CHAPTER 5

A Guide to Underflow and the Denormalized Numbers

> "Good intelligence work, Control had always preached, was gradual and rested on a kind of gentleness."
>
> John Le Carre, *Tinker, Tailor, Soldier, Spy*

Perhaps it is appropriate to open this chapter with a quote from a spy. Over the five years of the IEEE subcommittee meetings the gradual (sometimes called gentle) treatment of floating point underflow has been the center of controversial arguments and its own share of intrigue. In fact, over the first years of its activity, the subcommittee was not mentioned in the computing press without some reference to the heated controversy. The paper presented in this chapter was an attempt to explain and defuse the arguments. It is reprinted here, with permission of the publisher, from the March 1981 issue of *Computer* in which draft 8.0 of the proposed standard appeared. This paper remains an accurate microscopic view of the issues surrounding floating point underflow, despite that the proposed standard changed significantly from drafts 8.0 to 10.0. On a somewhat higher level, James Demmel's treatment of the implications of gradual underflow for solving linear systems, reference [8] in the attached paper, has been substantially updated and accepted for publication as of this writing.

This paper explains the now-defunct warning mode for handling denormalized numbers. However, the fact that there was a plausible *mathematical* explanation for warning mode, along with a belief among some early implementors that the mode was at least feasible, did not stop the IEEE subcommittee from voting the warning mode out of the proposed standard. Even though warning mode could be made to fit into an arithmetic system with

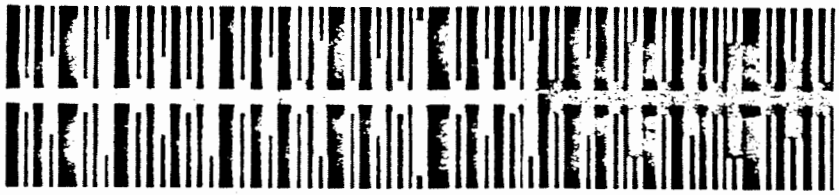
denormalized numbers, there was no simple, non-algorithmic explanation of how warning mode worked. Expositions like the original implementation guide in chapter 2 had the flavor of "do as I do, not as I say." Attempts to specify warning mode without algorithms in draft 8.0 of the proposed standard led to almost incomprehensible subtleties. This defect ultimately killed warning mode, by a nearly unanimous vote of the subcommittee. The purpose of warning mode, as discussed in this chapter, was to provide some defense for old programs written with the presumption that underflowed values would be set to zero; however, the value of this warning was very hard to quantify, unlike the complexity of exposition, which was apparent to anyone who read or attempted to improve upon the prose of draft 8.0. (This same discussion applies to the disappearance of the projective mode for interpretation of ∞. Although the projective mode was easy to describe and only a minor nuisance to implement, its value as a protection for programmers trained on machines like the CDC 6000/7000 class was small relative to its impact on a proposed standard expected to be in use for many years to come.)

Another change to the proposed standard since the publication of this paper is in the definition of underflow. This paper describes underflow as arising when a result, computed as though with unbounded exponent range and checked either before or after rounding to the target precision, falls below a specified threshold. This is a very conventional specification, in view of the computers built up to the 1980's. However, the so-called threshold test is pessimistic in an arithmetic with denormalized numbers. For example, whenever a difference $x-y$ falls below the underflow threshold, the result is given *exactly* by some denormalized number. So why signal underflow? And in some systems the assignment $z \leftarrow w$ between variables of

the same format is performed arithmetically, as opposed to a simple bit copy. In this case the "result" stored into z will fall below the underflow threshold whenever the source value w is a denormalized number. Should underflow be signaled? The answer, according to section 7.4 of draft 10.0 of the proposed standard, is *NO*. That section, using notation explained in detail in this paper, ties the underflow signal to both threshold and rounding phenomena.

Although there have been misconceptions about it, gradual underflow fits naturally into the proposed standard and leads to simple, general statements about the arithmetic.

Underflow and the Denormalized Numbers



Jerome T. Coonen
University of California, Berkeley
Zilog, Inc.

In the spring of 1980, after meeting regularly for over two years, a subcommittee of the IEEE Computer Society Microprocessor Standards Committee voted to endorse a proposed standard for binary floating-point arithmetic (see the proposed standard in this issue). The ballot ended just the first phase of a continuing controversy. Although diverse objections were raised within the subcommittee, discussions usually drifted back to *gradual underflow*, the proposed response to exponent underflow. The arguments even found their way into the computing press, where most articles about the subcommittee's work focused on the "underflow issue"—as if that were all that divided the subcommittee.

This article explains gradual underflow, ranging from its interaction with floating-point number systems to its advantages for numerical software. The discussion is not deep, but it is very detailed and would normally interest only specialists in computer arithmetic. However, the controversy surrounding the proposed standard has become so entangled with misconceptions about underflow that a study of underflow is now of interest to a broader community.

In fact, underflow should not be an important issue. The fundamental issues are the choice of numbers and symbols to be included in the arithmetic, their encoding in storage, and the specification of operations upon them. To this foundation may be added features that cope with exceptions such as over/underflow. The proposed standard was developed this way, designed to be a complete scheme for arithmetic, balanced between utility and implementation cost. Ironically, gradual underflow was expected to go unnoticed by most users, coming into view only when potentially dangerous underflow errors were flagged.

The interconnectedness of the proposed standard's basic features complicated attempts to oppose it. Early challenges within the subcommittee were not easily focused on single aspects of the proposed number system and its encoding, since so many of the design choices were interconnected. These challenges ultimately addressed the proposal as a whole and, quite naturally, tended to drift to its points of least resistance. Thus was it possible for gradual underflow—one of the system's less compelling features—to become its most contentious.

I hope to show that gradual underflow fits naturally into the proposal, leading to simple, general statements about the arithmetic. What remain disproportionately complicated are the arguments about why these statements are more valuable than some others. The proposed standard does not solve all underflow problems, but it does provide many benefits for a small added cost to new implementations. Unfortunately, the prospect of retrofitting existing systems with features such as gradual underflow can be daunting, so manufacturers with prior commitments are faced with a tough choice. For them, gradual underflow is compelling only for systems all of whose formats—like the proposed *single* format—suffer from a narrow exponent range. These problems of retrofitting added to the controversy regarding underflow within the subcommittee.

Floating-point number systems

Conventional implementations of normalized binary floating-point arithmetic use a fixed number of bits to represent numbers in each data format, with a predetermined "boundary" between the exponent and signifi-

cant digit fields. For example, single format numbers in the proposed standard are 32-bit strings of the form shown in Figure 1. The fields S, E, and F are 1, 8, and 23 bits long, respectively. Interpreting E as an unsigned integer in the range 0 to 255, bit strings with $1 \leq E \leq 254$ represent what are called *normalized* numbers whose values are decoded

$$(-1)^S \times 2^{E-127} \times 1.F$$

Since the leading significant bit is known to be 1, it is not explicitly stored.

The representable numbers group naturally into intervals of the form $[2^n, 2^{n+1}]$. We call these intervals *binades*, the binary analog of *decade*. Within the binades, numbers are spaced uniformly at a distance equal to one unit in their last place. As the numbers approach zero, this absolute spacing decreases by a factor of two across each binade. For example, consider an analog of the proposed single format, restricted to six bits of precision. The representable numbers would appear as ticks on a line, as shown in Figure 2. Approaching zero from the right, each successive binade is half the width of its right-hand neighbor, reaching half the remaining distance to zero. This is a property of all binary floating-point systems.

Normalized arithmetic

Given the system of normalized numbers established above, the nicest model for arithmetic is:

> Compute a result as if with infinite precision and range and then, if necessary, round it to the nearest representable number in the destination format.

The proposed standard, as we will see later, conforms to this model whenever the infinitely precise result is mathematically defined and does not overflow. But, for the moment, consider arithmetic suffering at worst only rounding errors, in which case most current implementations correspond roughly to the model. (Some chop numbers to the next smaller representable value; others round correctly to the nearest value "almost always"; some are indescribable.)

In arithmetic conforming to the model above, the roundoff error incurred by results is expressed by the formula

$$(\textit{computed results}) = (\textit{true result}) \pm (\textit{roundoff})$$

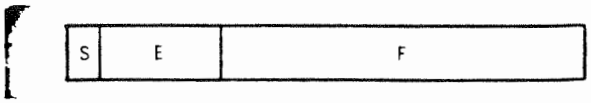


Figure 1. Form of 32-bit string for single format number.

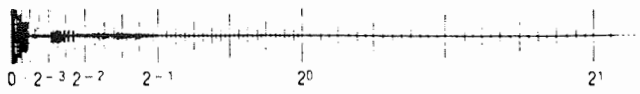


Figure 2. Analog of proposed single format, restricted to six bits of precision.

where *roundoff* $\leq$ ½ ulp (*u*nit in the *l*ast *p*lace) of the *computed result*. This error model parallels the earlier discussion of the binades since the absolute uncertainty of computed results decreases by a factor of two across each binade, as did the absolute spacing between adjacent normalized numbers.

The roundoff error can also be expressed by the formula

$$(\textit{computed result}) = (\textit{true result}) \times (1 \pm \varepsilon)$$

Here ε indicates the relative uncertainty of the calculation. In rounded binary arithmetic carrying t bits of precision $\varepsilon = 2^{-t}$. An analogous formula with 2^- replaced by 5×10^{-t} applies to rounded t-digit decimal arithmetic.

For a numerical example of the relative error formula, consider the product

$$1.23456 \cdot 10^{-12} \times 6.54321 \cdot 10^{+3} \rightarrow 8.07799 \cdot 10^{-9}$$

in a six-digit decimal system. Written in the form of the second formula above,

$$8.07799 \cdot 10^{-9} = (8.0779853376 \cdot 10^{-9}) \times (1 + 0.000000057717\ldots)$$

which is well within the range

$$(8.0779853376 \cdot 10^{-9}) \times (1 \pm 0.000005)$$

since the relative error is

$$0.000000057717\ldots < 0.000005 = \tfrac{1}{2} \text{ ulp of } 1.00000$$

The relative error formula implies what Figure 2 shows clearly—that the gaps between neighboring representable numbers never widen toward the origin. This has an important consequence: in any calculation suffering only one roundoff, the gap between a computed result and the exact result need never exceed any of the gaps between the computed result's several representable neighbors. For an illustration, consider the highly magnified picture of our sample product shown in Figure 3.

This seemingly obvious statement about gaps underlies many important properties of a robust floating-point system. Consider the following three properties, valid for calculations suffering nothing worse than roundoff:

(1) $x \neq y$ implies $x - y \neq 0$

(2) $(x - y) + y \approx x$ to within a rounding error in the larger of x and y

(3) $1/x \neq 0$ when x is a normalized number, and then $1/(1/x) \approx x$

Failure to satisfy statements like (1)-(3) can lead to interesting and elusive anomalies in numerical programs. Because it is our object to investigate the proposed standard, rather than review the past abuses that led to it, we will not pursue here the consequences of violating (1)-(3). Interested readers will find W. Kahan's survey[1] entertaining; refer to D. Hough[2] and Kahan[3] for more details. Suffice it to say that the desirability of an arithmetic system depends greatly upon its users' ability to form a simple yet accurate mental model of its capabilities. Statements (1)-(3) are typical of the high-level properties that permit a reasonable analysis of program behavior, thus expediting the production of robust numerical code.

What is exponent underflow?

Until now, the presentation has been covered by a disclaimer excluding all but normalized arithmetic suffering only rounding errors. The discussion applied to most current implementations of arithmetic. However, there are other sources of error. Because a fixed number of bits are allotted to each number's exponent, the number system's range is bounded. Some normalized binade must be the "smallest," beyond which there are no more normalized numbers. In the hypothetical six-bit normalized number system illustrated in Figure 2, the bottom of the exponent range would look like the representation in Figure 4. We will call the smallest normalized number λ and say that a result whose magnitude is less than λ has *underflowed*. The question is how to represent underflowed results when computation is to continue without a "trap" to a user's exception handler.

The proposed standard spans the gap from 0 to λ with a family of numbers whose absolute spacing is that of the numbers in the last normalized binade, as shown in Figure 5. These are the so-called *denormalized* numbers. They may be thought of as elements of an extra binade beyond λ, but spread apart by a factor of two over their expected spacing in order to reach 0 uniformly. The response to underflow which uses the denormalized numbers to represent underflowed values is called *gradual underflow*.

Gradual underflow has several historical precedents. Most often mentioned in the floating-point subcommittee's meetings has been the Electrologica X8, a Dutch machine. Using gradual underflow without even an underflow error flag, it was, according to T. J. Dekker, "never confusing to naive (and other) users." This is not too surprising, however, since the X8 had a 12-bit exponent providing a range of about $10^{\pm 600}$; it's unlikely that too many naive ("and other") users ever even encountered underflow. The Burroughs B5500, DEC-10, IBM 7094,[4] and IBM 370 also support gradual underflow, although the user must provide brief software routines to denormalize numbers since the hardware delivers underflowed values normalized with their exponents "wrapped around" to within range.

The only other underflow handling scheme that received broad support within the subcommittee is the one provided in most current implementations of arithmetic. Simplest of all the proposals, *Store 0* would set all underflowed values to zero, so that there be no representable numbers in the gap between λ and 0.

Another scheme dates back to work by K. Zuse in Germany during the 1930's[5] and work done independently by Kahan in 1966.[4] It would replace underflowed values by a symbol "UN," representing not any particular number but rather the *interval* between 0 and λ, which in our example would be $(0, 2^{-126})$. R. Fraley and J. S. Walther proposed such an *UN Symbol* scheme to the floating-point subcommittee,[6] though none has ever been implemented.

Yet another possibility would essentially change the "boundary" between the exponent and significant digit fields of a number which has underflowed (or overflowed) in order to obtain some exponent expansion at the expense of bits of precision. The dynamic position of the boundary would be built into the encoding of the numbers. R. Reid proposed this to the subcommittee,[7] though the idea has been attributed by D. Knuth[8] to J. Cocke. Although these schemes benefit from the expanded exponent, their fluctuating precision incurs a noticeable implementation cost and complicates error analysis. I will not discuss them further; however, the careful reader can adapt the arguments of this article to determine that the expanded numbers do not enjoy the simple properties to be attributed to gradual underflow.

Denormalized numbers and gradual underflow

The way denormalized numbers fit into a normalized number system can also be seen by listing the numbers from the smallest binades, with their implicit binary points aligned. Figure 6, from a six-bit analog of the proposed single format, shows representative "numbers" beside their unbiased exponents. X may be 0 or 1. This figure suggests a very natural representation of the de-

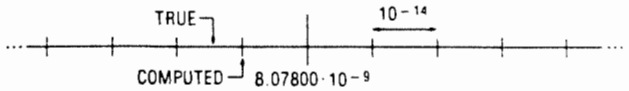


Figure 3. Highly magnified picture of sample product.

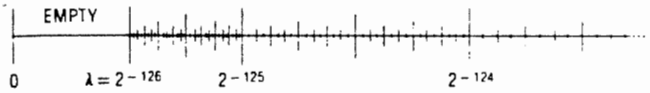


Figure 4. The bottom of the exponent range in the hypothetical system illustrated in Figure 2.

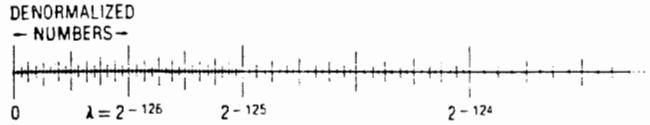


Figure 5. The denormalized numbers augment the number system shown in Figure 4.

EXPONENT	SIGNIFICANT BITS	
− 120	1XXXXX	
− 121	1XXXXX	
− 122	1XXXXX	
− 123	1XXXXX	← NORMALIZED NUMBERS
− 124	1XXXXX	
− 125	1XXXXX	
− 126	1XXXXX	
− 126	01XXXX	← UNDERFLOW THRESHOLD = $\lambda = 2^{-126}$
− 126	001XXX	
− 126	0001XX	← DENORMALIZED NUMBERS
− 126	00001X	
− 126	000001	
(− 126)	000000	← ZERO

Figure 6. A six-bit analog of the proposed single format, showing representative "numbers" beside their unbiased exponents.

normalized numbers in a floating-point system, since the denormalized numbers are precisely the values taken by all unnormalized numbers, of the given precision, whose exponent is that of λ. The single and double formats of the proposed standard exploit this fact by means apparently unknown before 1976. In the single format, for example, numbers in the interval $(\lambda, 2\lambda)$ are encoded with the next-to-lowest biased exponent, 1. The lowest exponent, 0, is reserved for the denormalized numbers and, when all significant bits are 0, for floating-point zero. Thus, the biased exponent 0 encodes two bits of information about the denormalized numbers:

- They have the same effective exponent as the normalized numbers, such as λ, with the next higher encoded exponent, 1.
- Their implicit leading bit is 0 instead of 1.

This encoding fits the denormalized numbers into the bottom of the exponent range inexpensively, using bit patterns that on many current implementations are simply redundant representations of zero. The name *de*normalized distinguishes the underflowed values from the usual *un*normalized numbers that run across a number system's entire exponent range. The single and double formats of the proposed standard have no *un*normalized numbers in this sense. Instead, they obtain an extra bit of precision over the normalized number range by assuming an implicit leading 1 bit for all numbers greater than or equal to λ.

Gradual underflow satisfies the arithmetic model presented earlier since an infinitely precise result, whether or not it is smaller than λ, is simply rounded to the nearest representable number. Although analogous statements can be made about the other underflow handling schemes, the striking difference is the extent to which the schemes admit high-level statements like (1)-(3) presented above.

Examples of denormalization

Figure 7 shows three examples of gradual underflow in a six-digit decimal system in which λ, the smallest normalized number, is 10^{-99}. In (i), the otherwise exact product underflows and must be denormalized by four digits. The number then requires rounding which, in this halfway case, is to the nearest representable number whose least significant digit is even. Intermediate results far below the underflow threshold will be denormalized all the way to 0, as in (ii). This occurs quite naturally in the proposed single and double formats, since signed 0 is represented as the "denormalized number" all of whose significant bits are zero.

Example (iii) illustrates how underflowed sums and differences of numbers in the same format are always free from rounding error. This is simpler than the situation for underflowed products and quotients which must be denormalized before rounding to ensure that their error bound is one-half unit in the result's last place.

These examples suggest the following straightforward implementation of gradual underflow. When a computed result would have an unbiased exponent too small—that is, too negative—to be represented, the number is accommodated by right-shifting (denormalizing) the significant digit field while incrementing the exponent until the exponent is that of the smallest normalized number. The number can then be rounded and stored.

Example (iii) suggests a possible economy in addition and subtraction, when the time required to denormalize is most likely to be noticed. After a magnitude subtract, the result need only be normalized until its exponent is that of λ, since further shifting would only be undone by subsequent denormalization. Such a simple trick is possible only because the denormalized numbers fit so naturally into the number system as a whole. It is typical of the ways in which a careful implementor of gradual underflow can achieve speeds comparable to the "simpler" arithmetic systems, with little additional hardware or microcode.

$$2.50000 \cdot 10^{-60} \times 3.50000 \cdot 10^{-43} = 8.75000 \cdot 10^{-103} \rightarrow 0.00088 \cdot 10^{-99} \quad \text{(i)}$$
$$2.50000 \cdot 10^{-60} \times 3.50000 \cdot 10^{-60} = 8.75000 \cdot 10^{-120} \rightarrow 0.0 \quad \text{(ii)}$$
$$5.67834 \cdot 10^{-97} - 5.67812 \cdot 10^{-97} = 2.20000 \cdot 10^{-101} \rightarrow 0.02200 \cdot 10^{-99} \quad \text{(iii)}$$

Figure 7. Three examples of gradual underflow in a six-digit decimal system.

Error properties of gradual underflow

The error formula describing model normalized arithmetic expressed only the relative uncertainty ε due to roundoff in a result free of other errors such as over/underflow. When underflow occurs, the formula becomes

$$(\textit{computed result}) = (\textit{true result}) \pm \xi$$

The uncertainty ξ of the result depends on the underflow handling scheme.

For purposes of discussion, we consider a hypothetical floating-point system with underflow threshold λ, augmented in turn by three underflow handling schemes. The bit patterns used for denormalized numbers could provide an extra normalized binade in the Store 0 and UN Symbol systems, thereby reducing λ by a factor of two. However, we will see that the analysis depends not upon the size of λ, but upon whether ξ is negligible when compared with λ.

- Gradual Underflow: When underflow is gradual, the error can be no bigger than half an ulp of λ, so $\xi = \varepsilon\lambda$.
- Store 0: When all underflows are set to 0, the error can be almost as large as the smallest normalized number, so $\xi = \lambda$.
- UN Symbol: When underflows are replaced by UN, the error is the same as for Store 0, so $\xi = \lambda$. The difference is that UN is less prone to subsequent misinterpretation.

Comparison of ξ for the various schemes indicates that only the denormalized numbers permit underflowed values to be represented with no more *absolute* error than is tolerable among numbers in the smallest normalized

binade. In other words,

> only with gradual underflow do the gaps between representable numbers not widen near zero; instead the gaps between computed and exact results are no wider than the gaps between any pairs of neighboring representable numbers.

For an example in six-digit decimal arithmetic with $\lambda = 10^{-99}$, consider the underflowed product shown in Figure 8. The Store 0 and UN Symbol schemes suffer an error equal to the product itself, about 8/10 of λ, while gradual underflow cuts the error to less than 2/10 *ulp* of λ, a reduction by several orders of magnitude. Figure 9, a highly magnified graph of the bottom of the exponent range, shows the gaps between true and computed results.

Mindful of the way that gaps around λ and 0 depend on the scheme for handling underflow, let us review the three properties we considered earlier:

(1) $x \neq y$ implies $x - y \neq 0$
(2) $(x-y)+y \approx x$ to within a rounding error in the larger of x or y
(3) $1/x \neq 0$ when x is a normalized number, and then $1/(1/x) \approx x$

This time we will permit the calculations to suffer underflow as well as roundoff errors. Aided by gradual underflow, the proposed standard satisfies (1)-(3) without a hitch for the same reason as applied to rounded normalized arithmetic—that is, the gaps between representable numbers never widen toward zero. This is the sense in which

> gradual underflow tends to make the errors due to underflow commensurate with roundoff errors.

However, (1)-(3) may not apply to the other systems since the gap between 0 and λ is huge when compared to the gaps between λ's neighbors, the tiny normalized numbers. For example, a Store 0 system violates (1) and (2) whenever $x - y$ underflows, and violates (3) whenever $1/x$ underflows. Whether the reciprocal of any number x can underflow depends on the balance between the largest and least exponents.

The UN Symbol scheme is more robust despite the fact that its error bounds are the same as those of Store 0. However, it entails several special cases. (1) is guaranteed because UN retains the sign of underflowed $x - y$ and has nonzero magnitude. In the same way, when $1/x$ underflows in (3) the quotient is nonzero, but then $1/(1/x)$ is OV, the overflow symbol, which is not $\approx x$. As in Store 0, (2) fails once $x - y$ underflows, in which case $(x-y)+y \rightarrow \text{UN} + y \rightarrow y$. To avoid this type of problem, a system bent on safety might deliver an invalid operation warning when UN (known only to lie somewhere between 0 and λ) is added to a tiny y; but, fooled or not, the user still gets the wrong answer.

The statements made here about the gaps are fundamental to floating-point error analysis. However, obsession with tiny errors is not the point. Rather, we would like our system to give reasonable results whenever possible, and a warning otherwise. In this way, we could worry about errors only when necessary and could have confidence in our results.

The trade-off between safety and utility is reflected in the specification of gradual underflow. We observed that if $x - y$ underflowed in (2), gradual underflow would always be accurate, Store 0 could give a wrong answer, and UN Symbol would give either a wrong answer or a warning. All schemes would raise an underflow flag upon computing $x - y$. However, experience with floating-point computation shows that the underflow flag by itself is not a reliable indication of serious error since most underflows can be safely ignored. To be used effectively, the flag must be interrogated after the delicate phases of a calculation. As we will see below, figuring out what should be tested represents a significant cost which is often avoidable when underflow is gradual. This undermines the perceived simplicity of the Store 0 scheme.

Proponents of the UN Symbol scheme emphasize its unwillingness to deliver wrong answers *due to underflow* when implemented conservatively. Alas, often when it signals an error associated with its symbols, an accurate answer could have been obtained using gradual underflow. And, as conservative as it may be, the UN Symbol scheme only catches errors due to underflow; since rounding errors are the source of most difficulties in sensitive calculations, this conservatism is only nominal.

Normalizing mode

The simplest implementation of denormalized numbers and gradual underflow, which has been assumed so far, specifies that each operation be performed without distinguishing denormalized numbers from other numbers—that is, as though all denormalized operands were first normalized. The proposed standard calls this the "normalizing" mode of computation. Such a uniform interpretation of nonzero numbers, regardless of possible loss of relative precision due to underflow, is appropriate when analysis shows that errors no bigger in absolute value than a half ulp of the smallest normalized

$1.23456 \cdot 10^{-60} \times 6.54321 \cdot 10^{-40} = 8.0779853376 \cdot 10^{-100}$ (EXACT)
$\rightarrow 0.80780 \cdot 10^{-99}$ (GRADUAL UNDERFLOW)
$\rightarrow 0.0$ (STORE 0)
$\rightarrow$ UN (UN SYMBOL)

Figure 8. Comparison of the various schemes in six-digit decimal arithmetic with $\lambda = 10^{-99}$.

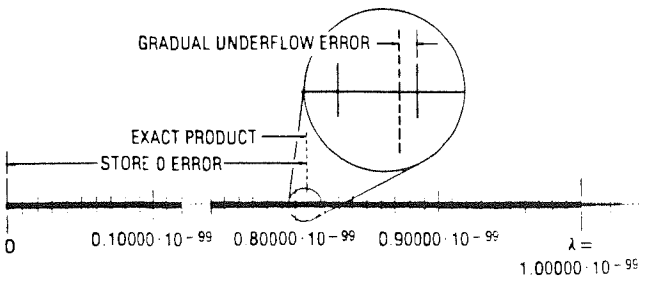


Figure 9. Highly magnified graph of the bottom of the exponent range.

number are no more significant than other comparable or larger errors due to roundoff. Most computations are this way.

Since the normalizing mode deals in principle only with normalized numbers, it follows essentially the same rules for denormalized numbers as for normalized. The only significant implementation cost is the prenormalization step required when denormalized operands participate in multiplication, division, and mixed-format calculations. In addition and subtraction of numbers of the same format, the prenormalization need not be carried out; since denormalized numbers already have the smallest exponent, they will be shifted right, if at all, for binary point alignment. As in the implementation discussion above, accompanying Figure 7, we see that the careful implementor of gradual underflow can trim the execution time cost of the denormalized numbers in addition and subtraction.

To see how gradual underflow works in a program, let us consider an inner product expression common in matrix calculation,

$$(b + \overline{a} \cdot \overline{y}) / c = (b + \sum_{i=1}^{n} a_i y_i) / c$$

and the program loop to evaluate it:

sum := *b*
FOR *i* := 1 TO *n* DO *sum* := *sum* + $a_i \times y_i$
result := *sum* / *c*

Suppose nothing worse happens than roundoff and underflow. If underflow is gradual, then as long as *b* is a normalized nonzero number, *sum* must be accurate to within the uncertainty of an unexceptional vector inner product with normalized numbers, namely a few ulps of $||\overline{a}|| \times ||\overline{y}|| + |b|$, where $||\overline{x}||$ denotes the norm of the vector $\overline{x}$. Consequently, *result* will be about as accurate as roundoff allows.

However, in a Store 0 system, a small but nonzero *sum* could be plausible but wrong in nearly every digit because of underflow. Figure 10 indicates how the two schemes affect small *sums* in one step of the computational loop

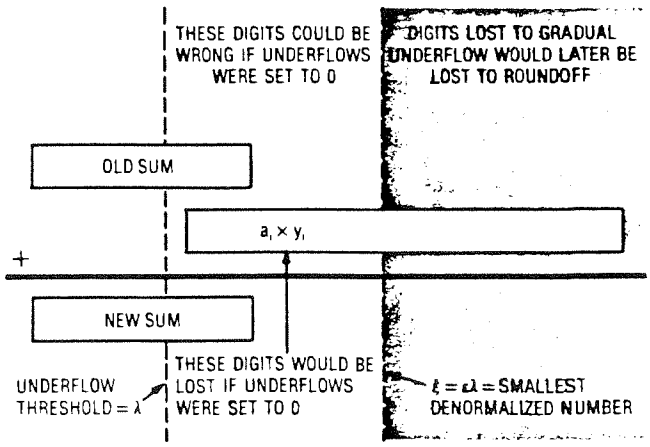


Figure 10. The effect of gradual underflow and Store 0 on small sums in one step of the cited program loop.

above. The accumulation *sum* and product $a_i y_i$ are represented as bit strings with their binary points aligned. When all underflows are set to zero, the information to the right of the vertical broken line marked "λ" (the underflow threshold) is lost—small *sums* can be seriously contaminated. On the other hand, gradual underflow retains enough information beyond λ to ensure that any *sum* greater than λ will be about as accurate as if all operands and products had been normalized to full precision.

Although this example is typical of those in which a simple statement describes the behavior of gradual underflow, it is not by itself a compelling argument for gradual underflow, since a robust program would require scaling to guard against a variety of potential nuisances—ranging from the special case $b = 0$, to overflow in *sum* when *c* is so large that the exact *result* would be well within range.

A more interesting, yet complicated, example is the calculation of the complex quotient

$$a + \mathbf{i} \times b := \frac{p + \mathbf{i} \times q}{r + \mathbf{i} \times s}$$

Assuming $|s| \leq |r|$, the procedure attributed to R. Smith by Knuth[8] is to calculate

$$a + \mathbf{i} \times b := \frac{p + q \times (\frac{s}{r})}{r + s \times (\frac{s}{r})} + \mathbf{i} \times \frac{q - p \times (\frac{s}{r})}{r + s \times (\frac{s}{r})}$$

An analysis can be found in a subcommittee working paper by Hough.[9] The claim is that, despite roundoff, the computed complex result differs from the correct result by no more than if $p + \mathbf{i} \times q$ and $r + \mathbf{i} \times s$ had each been perturbed by a few ulps of its modulus. This conclusion is unchanged by underflow, if it is gradual, except when both *a* and *b* underflow, in which case the error is bounded by a few ulps of $|a + \mathbf{i} \times b|$. No comparably simple statement holds when all underflows are set to 0.

The complex quotient is fundamentally different from the inner product above since Smith's algorithm produces a correct quotient unless intermediate overflow occurs. Furthermore, the formula avoids intermediate overflows when $a + \mathbf{i} \times b$ is in range, unless $|p| + |q|$ or $|r| + |s|$ would overflow. Since gradual underflow copes with all problems at the bottom of the exponent range, Smith's algorithm is so robust that there is little temptation to introduce scaling and its associated complexity.

Much ado about nothing?

Some opponents of the proposed standard have argued that programs which encounter gradual underflow in the normalizing mode would perform "about as well" if all underflows were set to zero instead. We can formalize the claim and a response as follows.

Figure 11 summarizes the notation developed throughout the discussion of the single format. We observed that λ is the absolute uncertainty of an underflowed result in the Store 0 and UN Symbol schemes, and that a nor-

malized computed result x' is related to an exact result x by

$$x' = x \times (1 \pm \varepsilon) = x \pm \varepsilon x$$

so that εx is a bound on the absolute error due to roundoff.

We consider programs which do not use special contingency code to handle underflow. Of particular interest is the class Δ of programs that succeed when underflow is gradual but fail when underflows are set abruptly to zero. These programs tolerate underflow errors bounded by $\xi = \varepsilon\lambda$ because they are no more significant than roundoff errors "εx" of comparable or larger magnitude, but cannot tolerate underflow errors as large as λ. How many programs are in class Δ?

The size of class Δ is a measure of how many programs benefit significantly from gradual underflow. If Store 0 were good enough for most calculations, as might be expected, the class Δ would be small, and then the extra capability afforded by gradual underflow would be inconsequential. However, the surprising fact is that many of the standard techniques of numerical analysis are known to fall into class Δ. This has been shown for linear equation solving by J. Demmel[10]; for polynomial equation solving by S. Linnainmaa[11]; for numerical integration and convergence acceleration by Kahan[12]; and for complex division, as indicated in the previous section.

Once the extent of Δ is established, one may argue that, with only slight amendments, programs in Δ can be made sufficiently robust that they tolerate abrupt underflow to 0. The reasoning is analogous to the motivation for gradual underflow in the first place: since the absolute error due to underflow can be as large as λ when all underflows are set to zero, underflow error can seriously contaminate numbers of which λ represents more than half an ulp. This was illustrated by Figure 6. If in that six-bit system all numbers below the indicated underflow threshold were set to zero, the bound on the incurred error would exceed half an ulp of all the normalized numbers less than 2^{-120}. Thus, numbers in the interval $[2^{-126}, 2^{-120})$ would be suspect in a calculation incurring underflow. In general, the number of contaminated binades equals the number of bits of precision carried. Thus, the threshold of suspicion for the proposed single format would be

$$\vartheta = \lambda/\varepsilon = 2^{-102} \approx 2.0 \cdot 10^{-31}$$

if underflow were not gradual.

For a concrete application of ϑ, consider the calculation

$$sum := b + \sum_{i=1}^{n} a_i y_i$$

in the inner product example presented earlier. We noted that setting all underflows to zero can ruin small *sum*s. More precisely, if underflow occurs in the summation above and $|b| < \vartheta$ then *sum* is not trustworthy.

Testing critical intermediate results against ϑ is really just a poor man's substitute for gradual underflow. In the latter, the threshold of suspicion is the more natural boundary, the underflow threshold, since the denormalized numbers tend to preserve the *granularity* of the number system down to the least significant bit of λ. When this threshold is crossed, the system raises the underflow flag. The difference between thresholds ϑ and λ illustrates the *completeness* that gradual underflow affords. In contrast, programs run with Store 0—even if they are augmented with tests to guard against contamination by underflow—won't achieve good results over so wide a range as simpler programs run with gradual underflow. Rather, as in the inner product example, their authors will be obliged either to explain the thresholds like ϑ to their users, or to insert contingency code, such as scaling, in order to eliminate artificial boundaries.

Another argument against gradual underflow focuses on numbers rather than programs. The claim is that the class Δ is irrelevant since computations rarely encounter underflows, and that when they do, the errors are nearly always inconsequential. This reasoning forces a dilemma upon purveyors of robust software for Store 0 systems, since the cost of the code to handle the rare cases when underflow *does* matter is out of all proportion to the benefit in the typical case. On the other hand, gradual underflow repays its slightly increased implementation cost with accurate results over a wider range of problems and data. And, as we will soon see, gradual underflow has a built-in warning system to lessen the chance that consequential underflows overlooked by programmers will be overlooked by users.

Old programs and the normalizing mode

Unfortunately, it is not reasonable for the proposed standard to specify the normalizing mode of computation as the default way to compute with denormalized numbers. Although the error ξ due to underflow is often negligible, the cases where it is not must be handled with great care—especially in would-be robust portable programs. Currently, most machines set all underflows to zero and most high-level languages lack a flag or name for the underflow condition. Consequently, whenever existing robust programs test for underflow in sensitive calculations, they have no choice but to check for zero results. These programs might be fooled by nonzero values (and hence presumed not underflowed) which have lost significance due to denormalization—especially if these values are later scaled up away from the underflow threshold. To protect the robustness of such programs, the proposed standard must be specified on the side of safety.

To see how a robust program could go wrong, consider the following code fragment intended to avoid errors due

λ = SMALLEST NORMALIZED NUMBER = $2^{-126} \approx 1.2 \cdot 10^{-38}$
ε = RELATIVE UNCERTAINTY OF A NORMALIZED RESULT = $2^{-24} \approx 6.0 \cdot 10^{-8}$
$\xi = \varepsilon\lambda$ = ABSOLUTE UNCERTAINTY OF A DENORMALIZED RESULT = $2^{-150} \approx 7.0 \cdot 10^{-46}$

Figure 11. Notation for discussion of proposed single format.

to underflow:

$q := (x \times y) \times z$
IF $q = 0.0$ THEN $q := x \times (y \times z)$

In a system setting all underflows to zero, the test guarantees a reasonable value for q unless overflow occurs. However, if $(x \times y)$ underflows instead to a denormalized nonzero number of only a few significant digits, and if $|z| >> 1.0$, then q may be well within range, though very inaccurate. For a numerical example in a six-digit decimal system with $\lambda = 10^{-99}$ let

$$x = 4.78295 \cdot 10^{-43}$$
$$y = 1.22805 \cdot 10^{-60}$$
$$z = 5.76623 \cdot 10^{-90}$$

Since $(x \times y)$ underflows gradually, the program produces

$$q := (x \times y) \times z \rightarrow (0.00059 \cdot 10^{-99}) \times z \rightarrow 3.40208 \cdot 10^{-12}$$

whereas the intended result, correct to fully six significant digits, is

$$q := x \times (y \times z) \rightarrow x \times (7.08122 \cdot 10^{30}) \rightarrow 3.38691 \cdot 10^{-12}$$

The fragment above should ideally be translated to the following more robust code in a standard environment, in which over/underflow can be tested explicitly:

underflow-flag := *overflow-flag* := FALSE
$q := (x \times y) \times z$
IF (*underflow-flag* OR *overflow-flag*) THEN
  BEGIN
    underflow-flag := *overflow-flag* := FALSE
    $q := x \times (y \times z)$
  END

This fragment is typical of those designed to cope automatically with what would otherwise be serious errors caused by over/underflow. Although the actual error ξ, suffered when underflow is gradual, is several orders of magnitude smaller than the possible error λ when all underflows are set to zero, the tiny error can nonetheless be catastrophic. Running such programs unchanged in the normalizing mode without further analysis is reckless.

Warning mode

Reckless or not, users will run programs like the first code fragment above, believing—perhaps wrongly—that they will compensate for underflow errors as well in a new environment as they did in the old. Thus, the proposed standard has an obligation to defend such programs against misinterpretation of denormalized numbers. It prescribes the so-called "warning mode" as its *default* mode of arithmetic on denormalized operands, to be in effect unless a program contains an explicit request for the normalizing mode. For example, the calculation above of q failed to produce an accurate result when the underflowed product $(x \times y)$ was normalized wholesale during its multiplication by z; in the warning mode, the second multiplication would be declared invalid and a Not-a-Number symbol, NaN, would be delivered in lieu of the dubious product. By inhibiting indiscriminate normalization of results—thus limiting the growth of relative error in results whose antecedents underflowed—the warning mode protects programs written with another scheme in mind as well as some programs written without any thought at all about underflow.

The warning mode differs from the normalizing mode in that it incorporates a boundary between valid and invalid operations on denormalized numbers. Although the boundary is arbitrary (a paranoid scheme might prohibit *any* further arithmetic on underflowed results), the boundary arises naturally in the proposed system, as we will see below.

A calculation run in the warning mode can be expected to achieve results at least as good as those gotten in the past; but sometimes NaNs will appear, signaling a potential underflow problem. If indeed the invalid results would have been junk, the user is better off with NaNs until the program is repaired. However, analysis often shows that underflow errors, when gradual, will not contaminate final results, as indicated earlier in the discussion of class Δ. In this case, accurate results can be obtained by a recalculation in the normalizing mode. The point is that a user can run programs initially without doing anything special about underflow. The warning mode is intended to defer as long as possible the judgment of whether an error ξ figures significantly in a computation.

For an example of the safety provided by the warning mode, consider the construction of a unit vector, $\bar{u} := \bar{x} / ||\bar{x}||$, by normalization of a given vector $\bar{x}$. This is a very common calculation. If $\bar{x}$ is of modest dimension, n, and its elements are in no special order, then $\bar{u}$ may be calculated in the obvious way with two loops:

sum := 0.0
FOR i := 1 TO n DO *sum* := *sum* + x_i^2
norm := $\sqrt{sum}$
FOR i := 1 TO n DO u_i := x_i / *norm*

If underflow is gradual, then as long as *sum* is a normalized nonzero number, *norm* is accurate to within about $n/2$ ulps, regardless of underflows in the x_i^2; hence $\bar{u}$ is about as accurate as roundoff allows.

However, if all the x_i^2 underflow, the computed *sum* might be denormalized. Then in the normalizing mode, *norm* would be a normalized number well above λ, but with relative uncertainty much larger than attributable to roundoff alone. This could seriously degrade the computed $\bar{u}$. The warning mode prevents this kind of error growth by declaring the square root of a denormalized number, like *sum*, to be invalid. In the extreme case that all the x_i^2 underflow to zero, *norm* and *sum* would be zero in both modes, and the second loop would be marked by division-by-zero errors.

The simple code above has the property that, when run in the default warning mode, it produces a result about as accurate as roundoff allows, so long as no exception besides underflow arises. Only in the rare case that overflow, division-by-zero, or invalid-operation is

flagged will $\bar{u}$ contain only zeros, ∞s and NaNs, and then the programmer will reject $\bar{u}$ and revise the program. This case is typical of the relative safety afforded naive programs by the warning mode. Of course, a truly robust program to compute $\bar{u}$ given any valid $\bar{x}$, however unlikely, would require scaling and some provision to suppress roundoff when n is huge.

This example neatly illustrates how the warning and normalizing modes are distinguished by their different interpretations of the absolute uncertainty ξ of denormalized numbers. The normalizing mode's presumption—that the error ξ is negligible regardless of the associated relative uncertainty—is replaced in the warning mode by rules intended to restrict the relative uncertainty of normalized numbers to what is expected because of roundoff.

The warning mode accounts for ξ by preserving the unnormalized character of denormalized operands. Instead of assuming an implicit prenormalization step at the start of each operation, the warning mode is specified in the proposed standard to be, as much as possible, a byproduct of the implementation of the normalized arithmetic, but allowing for a leading significant bit 0. In fact, the sum or difference of operands of the same format has the same numerical value in both warning and normalizing modes. This follows from the observation made earlier that prenormalization could be avoided during addition and subtraction in the normalizing mode. It is a reflection of how naturally the denormalized numbers augment normalized sums.

However, products and quotients involving denormalized numbers differ in the two modes. The distinction is a matter of acceptable error bounds, and may be characterized as follows. In the warning mode, a denormalized number is considered marked with an uncertainty of at least half a unit in its last place. Thus, it is thought of as an interval—like UN, though much narrower. The following fact, stated for products $a \times b$, applies to quotients a/b as well. It will be discussed in detail in the sections that follow. We use the subscripts W and N to indicate the warning and normalizing modes, respectively.

> Of a product $a \times b$, suppose that b is known to be normalized and presumed exact, and that a is finite, perhaps denormalized, and uncertain by 1/2 ulp. Then either:
> (1) $(a \times b)_W$ is not invalid, in which case it equals $(a \times b)_N$ and its error bound, 3/2 ulps, is the same regardless of whether a was denormalized; or
> (2) $(a \times b)_W$ is invalid, in which case $(a \times b)_N$ is uncertain by at least 5/2 ulps, and possibly much more.

That is, in the warning mode, the only tolerated errors due to underflow are those attributable to the rounding phenomena of arithmetic on normalized numbers.

Consider these statements applied to a recalculation of $q := (x \times y) \times z$ above, this time in the warning mode. The second multiplication

$$(0.00059 \cdot 10^{-99}) \times 5.76623 \cdot 10^{+90} = 0.0034020757 \cdot 10^{-9}$$

would not be normalized and rounded to $3.40208 \cdot 10^{-12}$, but would instead be flagged as an invalid result and replaced by a NaN. This would prevent the gross uncertainty inherited from

$$((0.00059 \pm 0.000005) \cdot 10^{-99} \times 5.76623 \cdot 10^{+90} = (3.40208 \pm 0.03) \cdot 10^{-12}$$

from being overlooked as though the same result $3.40208 \cdot 10^{-12}$ had been produced from relatively accurate normalized operands:

$$((5.90000 \pm 0.000005) \cdot 10^{-103}) \times 5.76623 \cdot 10^{+90} = (3.40208 \pm 0.000008) \cdot 10^{-12}$$

In this example, normalization of the result would have magnified the inherited uncertainty of half a unit in the sixth digit of the denormalized operand to a third of a unit in the *second* digit of the normalized result—a ten-thousand-fold increase. The warning mode permits no magnification bigger than by a factor of two. It is in this sense that the valid/invalid boundary is arbitrary, since in some computations a growth as large as what occurred above might be perfectly acceptable. The warning mode's magnification limit two was chosen because that is as much as roundoff errors can suffer in one operation, regardless of whether denormalized numbers were involved. Furthermore, that limit is straightforward to implement.

Valid results and the storage formats

A very important aspect of the error statements above is that they correspond to a straightforward implementation of the warning mode. One consequence of cases (1) and (2) is:

> In the warning mode, valid products and quotients are precisely those that can be stored in the destination format.

This connection between the floating-point formats and the inherited uncertainty of computed results is tied into the implicit leading bit of numbers above the underflow threshold λ, the subject of the next sections.

A binary floating-point product is computed internally as

$$\begin{array}{rl} A.aaa\ldots aaa & \times 2^{M} \\ \times B.bbb\ldots bbb & \times 2^{N} \\ \hline CC.ccc\ldots cccccc\ldots ccc & \times 2^{P} \end{array}$$

If either of the Cs is a 1, then the result can be rounded and stored, and will be normalized unless over/underflow intervenes. However, when both Cs are 0, then the result can be stored only if $P = N + M$ is no greater than the destination format's minimum exponent; otherwise, it is invalid because it violates the error statement in the last section.

Every product of a denormalized number and a factor bigger than two will have an exponent above the format's minimum. But not every such product is invalid. In some cases, the product of a number barely denormalized, say $0.1aa\ldots aaa$, and a normalized factor $1.bbb\ldots bbb$ will carry out to a product of the form $01.ccc\ldots cccccc\ldots ccc$. Despite the appearance that the absolute error of a

relatively inaccurate factor is being magnified, such a normalized result satisfies the error statement in the previous section. This particular phenomenon of products involving denormalized numbers will be considered in further detail later, in a different context.

From this discussion we see that the warning mode's principal impact upon implementations is the test to detect the unnormalized character of the results produced from denormalized operands. The valid/invalid boundary is maintained by a simple test to catch denormalized numbers that have been promoted to unnormalized numbers bigger than λ.

Analysis of a product

This section and the next explore the fine details that underlie the earlier statements about the error bounds of products and quotients in the warning mode. (The trusting reader may skip to the section entitled "Further Impact.")

First we consider a product of operands in the same floating-point format. Consider the calculation of

$$(C \times 2^P) := (A \times 2^M) \times (B \times 2^N)$$

where A, B, and C have the form $X.xxx\ldots xxx$ with $X = 0$ or 1. Allow $(A \times 2^M)$ to be normalized or denormalized, so that $0 \leqslant A < 2$; but assume that $(B \times 2^N)$ is normalized, so that $1 \leqslant B < 2$.

First, the exceptional cases: If the product underflows, then the denormalized result is the same in both warning and normalizing modes. This result does not satisfy the relative error bound given below, but instead suffers an absolute error bounded by ξ, as described earlier. The warning and normalizing mode results also agree when the product overflows, in which case both operands must have been normalized.

The interesting cases are those whose results are within range and whose only errors are rounding phenomena. Since A is uncertain by half an ulp, the normalized product takes the form

$$C \pm \gamma = 2^I \times (A \pm \varepsilon) \times B \pm \varepsilon$$

with exponent $P = M + N - I$. Sorting this formula out from left to right,

- γ is the error bound of the product, to be expressed in ulps of C;
- I is the number of left shifts required in the normalizing mode and, when $I = -1$, the one right shift required when the product of the significant digit fields is greater than or equal to 2;
- ε is half an ulp of 1.0—the leftmost ε expresses the inherited uncertainty in A, and the trailing ε bounds the rounding error in the product.

We examine three cases to interpret the error bound $\gamma = (2^I B + 1) \times \varepsilon$.

$I = -1$: The product of the significant digit fields is at least 2, so one right shift is required, producing a normalized result. This is possible only if both operands were normalized. Consequently, the warning and normalizing results agree and

$$\gamma = (B/2 + 1) \times \varepsilon < 1 \text{ ulp of C}$$

$I = 0$: The product of the significant digit fields is between 1 and 2, so the result is normalized and requires no shifting. Hence the warning and normalizing mode results agree. Whether A was normalized or not,

$$\gamma = (B + 1) \times \varepsilon < 3/2 \text{ ulps of C}$$

$I > 0$: The product of the significant digit fields is less than 1, so A must have been denormalized. The warning mode result is invalid and is replaced by a NaN. The normalizing mode result requires at least one left shift to produce a normalized result which satisfies

$$\gamma = (2^I B + 1) \times \varepsilon < (2^I + \tfrac{1}{2}) \text{ ulps of C}$$

As noted, the first two cases, $I = 0$ and -1, cover all valid warning mode arithmetic, regardless of the operands. Even if the first operand were denormalized ($0 \leqslant A < 1$), since the product *carried out* to a normalized result falling into case $I = 0$, the error bound of the result would be no worse than for normalized operands. However, the case $I > 0$ points out an important fact alluded to earlier:

> The gap between valid and invalid results in the warning mode is noticeably bigger than a rounding error, since the error bound of an invalid result exceeds by at least an ulp what it would have been for normalized operands.

In the previous section, we saw the close link between the valid/invalid boundary and the single and double storage formats. Now, it is clear that the boundary is not simply an accident of the implementation nor an arbitrary threshold drawn from a continuum. Instead, it is dictated by a *jump* in the error bound.

The case analysis above can be viewed in a different way. Although B was introduced as a normalized number, presumed exact, the computed error bounds were based on the worst case $B \approx 2$. When the analysis is retraced for any particular value $1 \leqslant B < 2$ the conclusion is the same—namely that γ jumps from case $I = 0$ to case $I > 0$, even though the particular values of γ are different, depending on B.

The case $I = 0$ when A is denormalized was mentioned in the last section, and will turn up again later. It received considerable attention within the floating-point subcommittee because of the apparent breach in the warning mode's defense, permitting the absolute uncertainty of denormalized numbers to be magnified. However, we saw in the analysis above that the associated error bound 3/2 ulps of C applies to some normalized products as well. In fact, this word "some" can be strengthened, since there are normalized numbers with $A' \approx A$ and $B' \approx B$ such that

$$C \pm \gamma = (A' \pm \varepsilon) \times B' \pm \varepsilon$$

Thus, the perceived growth of the uncertainty of denormalized A is unexceptional, since nearby normalized operands suffer the same error bound.

Does a quotient really differ?

The last three sections have discussed floating-point products in considerable detail. All the statements about error bounds apply as well to quotients. Given all the assumptions about A, B, and C above, consider the calculation of

$$(C \times 2^P) := (A \times 2^M) / (B \times 2^N)$$

The normalized quotient takes the form

$$C \pm \gamma = 2^I \times (A \pm \varepsilon) / B \pm \varepsilon$$

with exponent $P = M - N - I$. As with the product, three cases determine the error bound $\gamma = (1 + 2^I / B) \times \varepsilon$, namely $I = 0, 1, > 1$. These correspond directly to the cases $I = -1, 0, > 0$ for products.

The offset of 1 in the cases of I reflects an important difference between the two operations. Because B, which is normalized between 1 and 2, is in the denominator, one left-shift of the quotient might be required to normalize C, even if A is normalized. (Divide 3 into 1 in binary, for example.) So this one left shift is permitted of any quotient. Though it may appear to be an extra shift, in the sense that no such shift is allowed a product in the warning mode, quotients in cases $I = 0, 1$ do satisfy the same 3/2 ulps error bound deduced for products. Quotients satisfy the analogous bound $(2^{I-1} + \frac{1}{2})$ ulps of C when $I > 1$ and shifts beyond the first "free" one are required in the normalizing mode.

A more complicated analysis is required for the calculation of

$$(C \times 2^P) := (B \times 2^N) / (A \times 2^M)$$

Although the computed error bounds are similar, division by a denormalized number is invalid in the warning mode. This is another instance of the somewhat arbitrary boundary between valid and invalid results—here, the expense of building divide units capable of handling unnormalized divisors was not considered worth the dubious utility of dividing by tiny numbers in the warning mode.

The extra shift that quotients are permitted gives rise to a curious difference between the product and quotient:

$$(0.1aa\ldots aaa \times 2^M) \times (1.000\ldots00 \times 2^N)$$

and

$$(0.1aa\ldots aaa \times 2^M) / (1.000\ldots00 \times 2^{-N})$$

in the warning mode. Suppose that $N > 0$ and that M is the exponent of λ, the smallest normalized number. Thus, the left operand is a number denormalized by just one bit, and the right operand is a power of two. Since the product would be unnormalized, albeit exact, the result is invalid. However, the quotient $(C \times 2^P)$ is the normalized number $1.aaa\ldots aa0 \times 2^{M+N-1}$.

This distinction between certain products and quotients is an artifact of the measurement of error in ulps, a phenomenon that will be discussed below. Were the product above allowed one left shift then, as noted in the last section, it would be possible to perturb the operands just slightly to get a result suffering an error of up to 5/2 ulps—an ulp more than could be gotten from normalized operands.

Further impact

Now that the rationale behind gradual underflow has been presented, it is appropriate to tie the scheme into the proposed standard as a whole. This will provide some insight into the nature of the arguments that occupied the floating-point subcommittee for so long.

Until now, we have dealt with operations whose operands and results were all single or all double. However, the proposed standard recommends wider *extended* formats for intermediate calculations, thus encouraging mixed-format operations. As in any scheme of arithmetic, these mixed-format operations somewhat complicate the analysis. Also, since the optional extended formats have an explicit leading bit, they permit unnormalized numbers over their entire exponent range. Thus, the rules for normalized arithmetic with gradual underflow must be expanded to accommodate extended formats. This also complicates the analysis, but it is beyond the scope of this article.

The specification of the single and double storage formats is based on several good ideas. It is desirable that the numbers retain their natural ordering when interpreted as signed integers. This implies that when a floating-point number is viewed as a bit string, its most significant bit is its sign, followed by its exponent, and then by the significant digit field. The leading bit of the latter field is stored implicitly for the sake of added precision. This ordering property implies that the exponent be *biased* so that the value 0 of the biased exponent pertains to the most negative true exponent. As suggested when the denormalized numbers were introduced, the exponent 0 is used in the representation of floating-point zero and the denormalized numbers.

Unlike underflow, which is gradual, overflowed results are set abruptly to signed ∞. No effective and economical analog of gradual underflow is known for handling overflow. However, abrupt overflow is reasonable since calculations can be scaled or otherwise transformed so that quantities that must transgress a system's limits will underflow gradually. For example, most iterative procedures are designed to drive a residual value to negligibility. When a residual underflows to zero gradually, it is known to be negligible compared with every normalized number.

The largest value of the biased exponent is reserved for $\pm\infty$ (when all significant digits are 0) and the NaNs (otherwise). In this way, the finite numbers lie between and the NaNs lie beyond $\pm\infty$. The specified signed ∞ allows an affine closure of the number set, although a projective mode which effectively ignores ∞'s sign is specified, too. So that $+\infty$ and $-\infty$ have distinct reciprocals, floating-point zero is signed, though the sign cannot be discovered except by taking zero's reciprocal. The specification of signed zero led to the important decision to use the sign-magnitude ordering of floating-point numbers as integers.

The choice of exponent bias exploits the gradual treatment of underflows. To diminish slightly the risk of overflow, which is abrupt—though possibly at the cost of greater risk of underflow, which is gradual—it favors large numbers in the sense that

$$\lambda \times \Lambda \approx 4$$

where λ and Λ are the smallest and biggest normalized numbers. This means that if x is normalized, then commonplace expressions like $1/x$, $2/x$, $3/x$, and π/x cannot overflow to ∞; and if any underflows, it will lose two bits of precision at worst.

The jaggies

Another argument against gradual underflow arises from a graph of the so-called "jaggies." As represented in Figure 12, the graph is essentially a bit-by-bit account of the case $I = 0$ as discussed earlier under "Analysis of a product" and alluded to in "Valid results and the storage formats." Using the notation from the former section, the normalized factor $(B \times 2^{\wedge})$ ranges across the logarithmic horizontal scale, while the value of A, assuming $(A \times 2^{M})$ has the exponent of λ, ranges across the vertical scale.

The purpose of the graph is to show the jagged edge between valid and invalid products in the warning mode. The edge is the set of pairs A and B such that $A \times B = 1$, with $0 \leq A < 1$ and $1 \leq B < 2$. The product of $(A \times 2^{M})$ and $(B \times 2^{\wedge})$ is valid unless $M + N$ exceeds the exponent of λ and $A \times B < 1$, in which case the result cannot be encoded in a format whose leading significant bit is implicitly 1. The claim is that, despite this simple description, users will not tolerate such "jagged" behavior in their arithmetic—that changing an operand slightly should not make the difference between valid and invalid results.

This argument falls short for several reasons. First, the gist of the detailed analysis presented earlier is that, despite the result of one or another particular product, there is a powerful general statement describing the inherited error in products and quotients in warning mode. And the jagged edge is not peculiar to gradual underflow; indeed, products and quotients were shown to inherit uncertainty with an equally jagged graph. Jagged edges abound whenever calculations depend strongly upon small differences that amount to rounding errors.

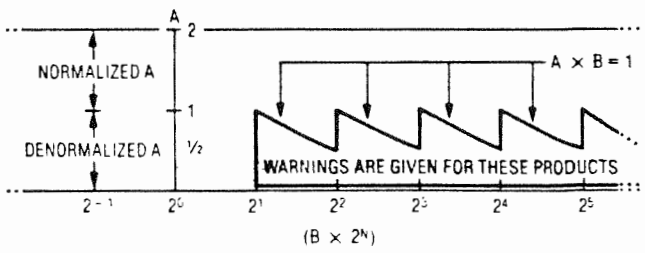


Figure 12. The jaggies.

Independent of the discussion of gradual underflow or particular operations, the graph of the jaggies should be nothing new to users of floating-point arithmetic. If the horizontal scale is simply the real number line and the vertical scale measures the relative uncertainty of real numbers rounded to the form $(B \times 2^{\wedge})$, a graph with *exactly the same* shape results. And not a single arithmetic operation is involved! Thus, the jaggies are simply another rounding phenomenon—which is what gradual underflow is intended to be.

Conclusion

Floating-point computation is intrinsically complicated. Traditionally, implementors have simplified their task at the expense of more complicated—or less reliable—software. However, the proposed standard takes the opposite tack. Consequently, the details of implementation of the proposal are many, as shown in an earlier article.[13] But, as proven here for underflow, a close look reveals an underlying coherence that leads to simple statements about the arithmetic. The implementation complexity will be justified if high-quality software developed for standard environments proves to be simpler, more portable, and thus cheaper than it has been in the past.

New software will tend to employ the normalizing mode, by request in the prologue of the program, since so many computations lend themselves to an analysis proving that denormalized numbers can be normalized with impunity. Nonetheless, the constraints of current and past software practice dictate that the warning mode be specified as the default mode of operation in the proposed standard. Periodically, the simplicity of both explaining and implementing the normalizing mode with its effective lack of unnormalized operands will be rediscovered, and it will be suggested as the default (and perhaps only) mode of operation. This may be fine for the future, but for now, existing programs have to perform at least as well as they have in the past—or stimulate a warning. For this, the warning mode is vital.

Acknowledgments

I wish to acknowledge W. Kahan's invaluable advice throughout the development of this article. This article was developed on a computer system funded by the US Department of Energy, Contract DE-AM03-76SF00034, Project Agreement DE-AS03-79ER10358. An earlier draft was published as report PAM-21 of the Center for Pure and Applied Mathematics at UC, Berkeley.

References

1. W. Kahan, "A Survey of Error Analysis," in *Information Processing 71*, North-Holland, Amsterdam, 1977.

2. D. Hough, ed., "Implementation of Algorithms: Parts 1 and 2," Document DDC AD-769 124, National Technical Information Service, 1973.

3. W. Kahan, "Why Do We Need a Floating-Point Arithmetic Standard?" (to appear).

4. W. Kahan, "7094-II System Support for Numerical Analysis," SHARE Secretarial Distribution SSD-159, Item C4537, 1966.

5. H. Ruthishauser, A. Speiser, and E. Stiefel, "Programmgesteuerte digitale Rechengeräte (elektronische Rechenmaschinen)," *Zeitschrift für Angewandte Mathematik und Physik*, Vol. 1, 1950, p. 348.

6. R. A. Fraley and J. S. Walther, "A Proposed Standard for Binary Floating-Point Arithmetic: Alternate 3," IEEE Floating-Point Subcommittee Working Document P754/80-1.24, 1980.*

7. R. Reid, "The Reid Format," IEEE Floating-Point Subcommittee Working Document #22 in August 7, 1979 mailing.*

8. D. E. Knuth, *The Art of Computer Programming, Vol. 2: Semi-Numerical Algorithms*, Addison-Wesley, Reading, Mass., 1969, p. 195.

9. D. Hough, "Errors and Error Bounds," IEEE Floating-Point Subcommittee Working Document P754/80-3.2, 1980.*

10. J. Demmel, "Solving Linear Systems Using Gradual Underflow," IEEE Floating-Point Subcommittee Working Document P754/80-4.21, 1980.*

11. S. Linnainmaa, "Combatting the Effects of Underflow and Overflow in Determining Real Roots of Polynomials," IEEE Floating-Point Subcommittee Working Document P754/80-2.23, 1980.*

12. W. Kahan, "Aitken's Extrapolation and Gaussian Quadrature," IEEE Floating-Point Subcommittee Working Document P754/80-1.19, 1980.*

13. J. T. Coonen, "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic," *Computer*, Vol. 13, No. 1, Jan. 1980, pp. 68-79; see also "Errata," this issue.

*Subcommittee working documents still in print may be obtained from David Hough, PO Box 561, Cupertino, CA 95015.

CHAPTER 6

Comparisons and Branching

1. Introduction

A basic fact of real arithmetic is that two numbers x and y compare as exactly one of *less*, *equal*, or *greater*. However, this so-called trichotomy property does not hold when the real number system is expanded to include not-a-number symbols (NANs) because these symbols have no natural ordering with the real numbers. This chapter deals with the issues raised by NANs in the number system.

Loss of the trichotomy property complicates comparisons. Consider the simple code sequence:

if $x > 3.1416$ **then** ...

else ...

If x is a NAN then the inequality is surely false, so the **else** clause must be executed. But might the **else** clause have been written with the presumption "$x \leq \pi$" in mind? If so, a NAN value of x may be disastrous. The problem is more historical than technical. Since most computer systems to date have simply stopped when a non-numeric reserved operand appeared, this problem has been avoided, though at considerable cost in the utility of the reserved operands. Nowadays, when arithmetic operators are overloaded to apply to complex numbers, arrays, or intervals, which though "numeric" may have no linear ordering, the very same issues arise.

Here are the subjects to be dealt with in the coming sections:

(1) In a system supporting partially ordered entities, what rules for comparisons hold in lieu of the trichotomy property of the real number system?

(2) What do the expanded rules for comparisons have to say about the relational operators of current languages? For example, Pascal's relational operators

= <> < <= > >=

themselves reflect the presumption that if two values are not *equal*, <>, then they are related as *less* or *greater*.

(3) What protection is there for existing programs and programmers who labor under the assumption that floating point entities enjoy the trichotomy relation?

(4) How can the relational operators of current languages be expanded in a reasonable way? What expansion, if any, is required by the proposed binary floating point standard P754?

(5) What underlying implementations of floating point comparisons best serve the needs of language systems and programmers?

(6) How can the expanded set of relational operators be made compatible with existing computers?

2. Relations

In the P754 number system with its NANs, the trichotomy is expanded to the four-way relation *less*, *equal*, *greater*, or *unordered*. Determining the relation between two floating point values x and y is actually quite easy. Working backward from the special cases:

if x is NAN **or** y is NAN **then** x and y are *unordered* ...

else x and y are *less*, *equal*, or *greater* according to the ordering of real numbers with the understanding that

$$+0 = -0 = \text{real } 0$$

and

$$-\infty < \{ \textit{all real numbers} \} < +\infty \ .$$

Some computers, notably the CDC 6000 class, have been built without a floating point comparison instruction, requiring compilers and assembly language programmers to issue code sequences like

temp ← $x-y$

test *temp* for positive, negative, or zero

to effect comparisons. However, the proposed standards make this type of implementation inconvenient, if not infeasible, by explicitly prohibiting the possible side effects of the subtraction – overflow, underflow, inexact result, invalid operation (see §5.7 of draft 10.0). Even with all due care in suppressing the extraneous exception flags in the subtraction, the scheme above will require special tests for cases like $+\infty = +\infty$, since $(+\infty)-(+\infty)$ is invalid, not zero.

Of course, if a signaling NAN appears as an operand in a comparison it stimulates the invalid operation exception, just as it would in any other arithmetic operation. Like a quiet NAN, it would compare *unordered* with the other operand, though an invalid operation trap handler might modify the relation based on an interpretation of the NAN outside the scope of P754.

3. Current Language Predicates

In a P754 system, current language predicates like =, <, and >= keep their literal interpretation despite the new relation *unordered*. For example, consider the Pascal code fragment:

if $x < y$ **then begin** ... **end**

else begin ... **end**;

If and only if x is *less* than y is the **then** clause executed. If x is *equal* to, *greater* than, or *unordered* with y then the **else** clause is executed. Thus the meaning of the relational < has not changed, only the inference drawn from its negation; execution of the **else** clause no longer implies that $x \geq y$.

Similar rules apply to the relationals =, <=, >, and >=. Their literal interpretation is honored in deciding the fate of an **if-then-else** clause. However, the situation is more interesting for the relational "not equal" because of the way it is written. In Pascal, the literal interpretation of "<>" is "*less* or *greater*". On the other hand, the literal interpretation of the FORTRAN ".NE." is more reasonably "*less*, *greater*, or *unordered*". Current users of both languages probably refer to both relationals as "not equal" and might be surprised at any semantic difference. Is it better to follow the literal interpretation of the syntactic form or to be consistent with the probable intent across different languages? One could argue the former case on taste and the latter on the basis of portability of algorithms between different language systems. Since the computer cannot read the programmer's mind, it has to take what is said literally just in case what is said is what is meant literally.

4. Old Habits

The fourth relation, *unordered*, can undermine old programs, old programmers, and even old programming languages. Proposal P754 provides a measure of security against mistaken inferences in **else** clauses such as

if $x < y$ **then begin** ... **end**

else begin ... **end**;

by stipulating that in such instances, if x and y are indeed *unordered*, the invalid operation exception should be stimulated. This is the best that can be done since there is no floating point "result" from the comparison, with which to propagate the NAN operand's diagnostic information.

According to §5.7 of P754, the invalid operation exception is to be signaled when unordered operands are compared with a predicate "involving" the relations *less* or *greater* but not *unordered*. Thus, two families of relationals are deliberately exempted from the protection mechanism for unordered operands. First, the FORTRAN ".EQ." and ".NE." are always unexceptional since the are used in floating point calculations primarily to weed out special, anomalous, values. This is quite different from using ".LT." to distinguish the condition *less* from "*greater* or *equal*"; this comparison involves a presumption that may not be valid. The second exemption from the invalid exception is for any predicate that explicitly mentions (i.e., "involves") the *unordered* relation. As of this writing, there are few implementations of languages with such relationals. But one could imagine an expanded FORTRAN with ".ULE." for "*unordered*, *less*, or *equal*". P754 exempts a statement like

IF (*X* .ULE. *Y*) **GOTO** 2050

from the invalid operation exception when X and Y are *unordered* since, by writing ".ULE.", the programmer has shown a modicum of regard for the *unordered* contingency; no protection is required.

These special relationals exempt from exceptions on *unordered* raise some additional issues. Consider the two FORTRAN tests

IF (.NOT. X .GT. Y) **GOTO** 2001

IF (X .ULE. Y) **GOTO** 2001

Although the logical negation of "*greater*" is indeed "*unordered, less*, or *equal*", the two tests differ in the invalid operation side effect. The latter test is exempt from the exception because of its mention of *unordered* in the relational; the former test is not. On the other hand all of the tests

IF (X .NE. Y) **GOTO** 1984

IF (.NOT. X .EQ. Y) **GOTO** 1984

IF (X .ULG. Y) **GOTO** 1984

cause a branch precisely when x and y are related as "*unordered, less*, or *greater*", and all are exempt from the invalid operation on *unordered*.

5. P754 Predicates

The following table, adapted from proposal P754, describes the complete set of 26 relational predicates. Since there are four possible relations, *less*, *equal*, *greater*, or *unordered*, each of which may be tested for **true** or **false**, there are in principle 2^4 or 16 possible combinations. The unconditional **true** and **false** are omitted, leaving 14. Including the logical negations, that is $(x<y)$ and NOT$(x<y)$, yields 28. But two pairs of these

$$(x = y) \quad \text{and} \quad \text{NOT}(x \ ?<> y)$$

and

$$\text{NOT}(x = y) \quad \text{and} \quad (x \ ?<> y)$$

are functionally identical; deleting one of each pair leaves 26 functionally distinct relational predicates. (Note that the 12 other such pairs are functionally distinct because one member triggers the invalid operation exception if the operands are *unordered*, and the other is unexceptional.)

6. Extending Existing Languages

P754 specifies what to do with each of the possible relational predicates that can be formed given the four relations *equal*, *less*, *greater*, and

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

unordered. However, the proposed standards do not force a language implementor to provide any given set of relationals. Virtually every programming language provides the set shown here for Pascal, BASIC, C, and FORTRAN.

Pascal and Basic	C	FORTRAN
=	==	.EQ.
<>	!=	.NE.
<	<	.LT.
<=	<=	.LE.
>	>	.GT.
>=	>=	.GE.

How should this set be expanded, if at all?

First consider an easy case. Suppose that the C programming environment is expanded to include the predicate function

```
integer unordered(x, y)
  float x, y;
{ ... }
```

which returns the value one if and only if x and y are *unordered*, without raising the invalid operation exception, and returns zero otherwise. Then the whole gamut of predicates is avaliable through constructions like

```
if (unordered(x, y) || (x < y) ) { ... }
```

The logical OR operator "||" is such that if the left expression is true (i.e., nonzero), then the comparison on the right is bypassed. This short-circuit evaluation allows the programmer to bypass the invalid operation exception the standards would mandate in case *unordered* values of x and y were compared with "<". C's logical operators were designed with just such uses in mind.

Now consider a Pascal system augmented by

function unordered(x, y: **real**): **boolean**;

which returns true if and only if x and y are *unordered*. The Pascal version of the C test above is

if unordered(x, y) **or** ($x < y$) **then begin** ... **end**;

Unlike C, Pascal does not specify the order of evaluation of the two tests. And Pascal says nothing about short-circuit evaluation, in case the first of the two expressions is true. So, although the flow of control is unambiguous, the invalid operation exception side-effect is left to the whims of the Pascal system. The programmer who would avoid unwanted side-effects caused by unpredicatable order of evaluation must force the order by nesting the tests:

```
if unordered(x, y)
   then begin ... end
   else   /* vacuous case */
else
   if x < y
      then begin ... end
      else;   /* vacuous case */
```

Unhappily for the Pascal programmer, it may be necessary to use **goto**'s to avoid duplication of code within the nested cases.

The Pascal programmer would be aided by an expanded set of relationals. Consider the set above augmented by the following set (written for FORTRAN and C as well):

Math	Pascal and BASIC	C	FORTRAN
unordered	?	?	.UO.
unordered or *equal*	?=	?=	.UEQ.
unordered or *less*	?<	?<	.ULT.
unordered or *greater*	?>	?>	.UGT.
unordered, *less* or *equal*	?<=	?<=	.ULE.
unordered, *greater* or *equal*	?>=	?>=	.UGE.
unordered, *greater* or *less* (not *equal*)	?<>	!=	.NE.

The "not *equal*" operator is now written precisely for all of the languages. The "*less* or *greater*" operator "<>" of Pascal is not shared by C and FORTRAN, but it is not so useful anyway. The symbol "?" in the Pascal and C relationals and the letter "U" in the FORTRAN relationals is deliberately placed at the head of the relational to suggest its short-circuit effect, that is, that no invalid operation exception will arise if the operands are *unordered*.

These relationals have two unfortunate properties. The FORTRAN versions are coincidental with the typical assembly-language names for the unsigned integer comparisons, which could cause confusion. Also, the question mark may be inscrutable when used in a context like

if x ? y **then begin** ... **end**;

An alternative is to use either the function unordered(), or the complementary relationals with logical negation, like

if not (x <=> y) **then begin** ... **end**;

In the latter case, P754 calls for the invalid operation side effect when x and y are *unordered* since there is no explicit reference to the *unordered* relation.

7. Hardware Support for Language Constructs

Now that we have explored the language issues in comparisons we can look at the required hardware support. A conditional branch construct like the Pascal

if $x < y$ **then begin** <block A> **end**

else begin <block B> **end**;

might be compiled into assembly code of the form:

```
            COMPARE     x,y
            BRANCH      UGE, LABEL-B    ; skip to block B if ?, >, or =
            <block A>
            BRANCH      FINI            ; unconditionally skip block B
LABEL-B:
            <block B>
FINI:
```

What is important is that the compiler has "flipped" the sense of the predicate being tested, in order to branch around the **then** clause. In this case the relational "<", which triggers invalid if x and y are *unordered*, is implicitly replace by "?>=", which is never invalid. And an optimizer may attempt later to move code blocks A and B by flipping the relational once more. This is bad news if the arithmetic associates the invalid exception with the assembly language branch condition.

The compiler has three fundatmental responsibilities:

(1) Ensure that *unordered* operands trigger the invalid operation exception just when appropriate.

(2) Ensure that flipping the sense of the relational takes into account the four possible relations.

(3) Ensure that subsequent optimizations are safe.

Perhaps the simplest way to a robust implementation is to have two comparison instructions: one just a straight arithmetic comparison, and one that will also trigger the invalid operation exception on *unordered*. Then the compiler can issue the required flavor of comparison on the basis of the relational that appears in the source program, and the conditional branches can be flipped with impunity later.

8. Implementation Examples

The following sections illustrate ways of implementing the P754 predicates using the conditional branch schemes on existing CPUs. These processors were designed with the trichotomy in mind so some special care has been required.

8.1. 16-bit Microporcessors

The families of 16-bit microprocessors available today from Intel (8086), Motorola (68000), National (16000), and Zilog (Z8000) are two's-complement integer-only machines. These CPUs implement trichotomy comparisons using a set of condition code bits like:

C – carry-out of result

Z – zero result

S – sign of result

V – integer overflow

S is sometimes called N, for "negative bit". These bits are typically set according to the result of each integer arithmetic operation. They are tested using the conditional branch instructions. All the CPUs above either already have or are intended to have hardware support for floating point in the form of co-processor or slave chips. Will their existing branching

schemes suffice, even though the trichotomy property does not apply to P754 comparisons?

The conditional branch instructions come in two flavors depending on whether they interpret integer results as unsigned or two's-complement signed. The unsigned branches use the C and Z bits, and the signed branches use the Z, S, and V bits. By an appropriate mapping of floating point comparisons into the condition code bits, the two flavors of branches can be reinterpreted so as to incorporate the *unordered* relation.

For definiteness the following discussion is based specifically on the Zilog Z8000 microprocessor. Execpt for notational differences, the situation is the same for the other three microprocessors. One possible mapping of the condition code bits for floating point comparisons is:

C – set iff *less*

Z – set iff *equal*

S – set iff *less*

V – set iff *unordered*

A useful interpretation of the Z8000 branches is given for the expanded list of Pascal relationals. A question mark signifies *unordered* in the ad hoc relational predicates that mention that relation. Note that of the fourteen possible combinations of the four relations (ignoring the trivial **true** and **false**) only one complementary pair cannot be tested with a single Z8000 conditional branch.

Pascal Predicate	Integer Predicate	Z8000 Condition Code Setting
=	=	Z = 1
<	unsigned <	C = 1
<=	unsigned <=	C or Z = 1
>	>	Z or (S xor V) = 0

>=	>=	S xor V = 0
?	<	S xor V = 1
?<=	<=	Z or (S xor V) = 1
?>	unsigned >	C or Z = 0
?>=	unsigned >=	C = 0
?	overflow	V = 1
<=>	no overflow	V = 0
?<>	not equal	Z = 0
?=	NONE	Z or V = 1
<>	NONE	Z or V = 0

With this mapping of the condition codes, full support is given the assembly language programmer (and the compiler) if the assembler merely recognizes the set of "floating relationals" and maps them into the appropriate condition code test. For example, the assembly instruction

```
JR   FLE,LABEL3
```

requesting a Jump (Relative to the current program counter) to LABEL3 if the floating relation <= is true, would be interpreted as the actual Z8000 instruction

```
JR   ULE,LABEL3
```

using the integer relation unsigned <=.

Although this mapping between integer and floating relationals may seem nonintuitive at first, it is an exercise to show that this is the best that can be done. The only nontrivial flexibility is in choosing which two "double" relationals will require two branch intructions. In this case, the relationals ?= and <> were chosen as the least likely to arise in practice.

8.2. 8-bit Microprocessors

The Intel 8080, Rockwell 6502, and the Zilog Z80 are three common 8-bit integer-only microprocessors. Each has 4 condition code bits

C – carry-out of result

Z – zero result

S – sign of result

V – integer overflow

like the 16-bit processors above. But the 8-bit processors lack the full complement of signed and unsigned branches. Instead, each of the condition code bits must be tested individually with instructions like "branch on carry set", "branch on carry clear", etc.

So there is no clever mapping between the floating point relational predicates and the signed and unsigned integer predicates. The best that can be done is simply to map each of the the four floating relations onto one of the condition code bits:

C – set iff *less*

Z – set iff *equal*

S – set iff *greater*

V – set iff *unordered*

A useful interpretation of the Z80 branches is given for the expanded list of Pascal relationals. A question mark signifies *unordered* in the ad hoc relational predicates that mention that relation. Note that of the fourteen possible combinations of the four relations (ignoring the trivial **true** and **false**) only the combinations involving one or three relations can be tested with just one conditional branch.

Pascal Predicate	Condition Code Setting
=	Z = 1
<	C = 1
<=	C = 1 or Z = 1
>	S = 1

>=	S = 1 or Z = 1
?	V = 1
?<=	C = 1 or V = 1
?>	S = 1 or V = 1
?>=	C = 0
?	V = 1
<=>	V = 0
?<>	Z = 0
?=	Z = 1 or V = 1
<>	C = 1 or S = 1

Beyond this, this situation differs on the three microprocessors. The 8080 a set of three-byte branch instructions (one-byte opcode followed by one-word absolute address) to test each of the condition code bits; the Z80 has these instructions plus two-byte branch instructions (one-byte opcode followed by a byte offset from the current program counter) to test the C and Z bits. On the other hand, the 6502 has only two-byte instructions to test the condition code bits; branches beyond the range of the one-byte offset must be handled with an unconditional three-byte jump.

CHAPTER 7

Accurate Yet Economical Binary – Decimal Conversions

> "The ultimate aim is to persuade all of the civilized world to abandon the decimal numeration and to use octonal in its place; to discontinue counting in tens and to count in eights instead. However, it seems unlikely that the whole civilized world will be persuaded to complete this change during the next twelve months, having previously declined similar invitations."
>
> E. William Phillips (1936)

Introduction

Because of our "uncivilized" insistence on decimal arithmetic for everyday calculations, today's high-speed computers, most of which perform arithmetic in radix two or a power of two, must be supplied with conversion routines to expedite input and output of data in decimal form. These utilities typically run without the benefit of extra range or precision, in which case they are provably inaccurate, and often they use many more floating-point operations than do more robust algorithms. Now, proposed IEEE standard P754 for binary floating-point arithmetic [1] attempts to impose accuracy specifications for binary-decimal conversions. It turns out that the required accuracy can be achieved with very economical algorithms.

This chapter is an extended footnote to proposal P754. It describes algorithms that guarantee correctly rounded results for all input values. However, these schemes can be costly in time and space. The principal contribution of this chapter is an economical alternative, a set of fast algorithms that provide results that are just accurate enough. These algorithms have been adapted from an earlier implementation guide [3]. Implementors interested only in the algorithms may turn immediately to §2 of this chapter.

For the more leisurely reader, §1 introduces P754 and discusses the important issues in radix conversion. Unfortunately, discovering what is accurate enough in lieu of correct rounding, and correlating this with an efficient implementation, entail a surprisingly tedious error analysis. This analysis constitutes §3.

1. Radix Conversion Issues

1.1. Proposed Standard P754

A brief survey of proposed IEEE standard P754 for binary floating-point arithmetic will explain some of the terminology in the rest of the paper. The basic goal of the standard is to provide users with a computing environment conducive to the production and portability of numerical software. P754 specifies 32-bit single and 64-bit double formats, as well as optional system-dependent extended formats. The extended formats may be thought of as a computer's internal types; when available to programmers, they offer some valuable extra range and precision at little added cost in execution time and implementation complexity. P754 requires results computed as though with unbounded range and precision, and then coerced (by rounding and checks for exponent over/underflow) to fit in the destination format.

Four modes of rounding are specified in P754: the default mode *to nearest* and the three directed modes *toward* $-\infty$, *toward 0*, and *toward* $+\infty$. To express them in terms of radix conversion, let x and X represent binary and decimal floating-point numbers, respectively, with preassigned precision. Then the conversion $x \rightarrow X$ is correctly rounded if when rounding

to nearest: X is the nearest decimal to x, in case of a tie X has an even least significant digit

toward 0: X is the nearest decimal to x satisfying $|X| \le |x|$

toward $+\infty$: X is the nearest decimal to x satisfying $x \le X$

toward $-\infty$: X is the nearest decimal to x satisfying $X \le x$.

Analogous rules apply for decimal to binary conversion $X \to x$. However, for huge and tiny values of x and X these rules are so expensive that P754 permits them to be relaxed by, roughly speaking, replacing "nearest" with "nearest or next to nearest".

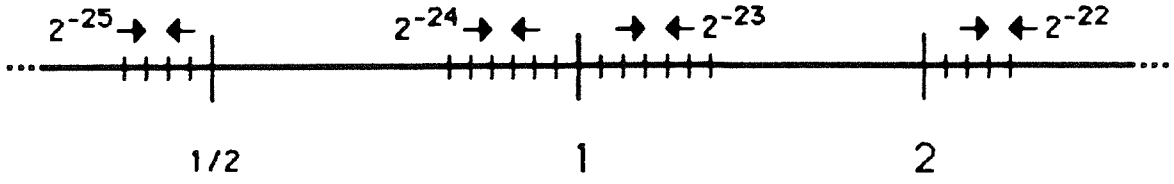
Radix conversions are vulnerable to rounding errors, exponent overflow, and exponent underflow. In addition to these exceptions, P754 distinguishes two others, division by zero, and invalid operation (like 0/0), but these do not matter for our purposes. Associated with each of the exceptions is a status flag accessible to programs. A flag must be set whenever its corresponding exception arises; it may be cleared only by user software. An implementation may also support traps for each of the exceptions, but these are optional. Traps present problems more system-related than numerical, but they are mentioned later in the few instances where they affect the algorithms. Finally, P754 specifies the symbolic entities $\pm\infty$ to cope with overflow and division by zero, and NAN (not-a-number) to deal with invalid operations. Conversion to and from these symbols is left as a special case to be handled by the implementor.

1.2. Floating-Point Number Systems

A conventional floating-point number system is characterized by its radix, precision, and range. For example, the values of the finite numbers in the P754 single format are precisely the values

$$\pm b_0 . b_1 b_2 b_3 \cdots b_{23} \times 2^e \quad ,$$

where each b_k is either 0 or 1 and $-126 \le e \le +127$. A simple way to view this number system is to divide the real number line into intervals of the form $[2^{n-1}, 2^n]$. We call these *binades*, the binary analog of decimal *decades*. Within each such binade the P754 single numbers have the absolute spacing 2^{n-24}, so they divide the binade into 2^{23} equal pieces. The size of the pieces doubles from binade to binade to the right. The following picture illustrates the number system near 1 on a logarithmic scale.

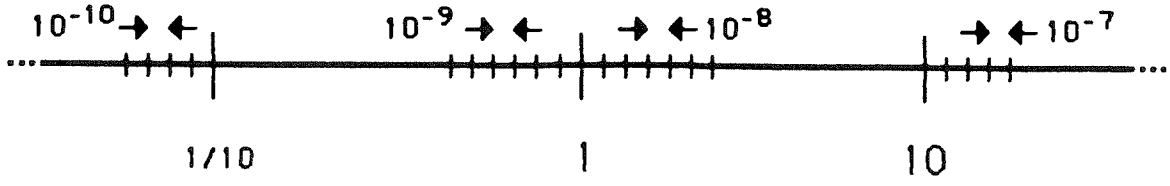


Of course this picture does not apply across the entire number line because of the constraints on the exponent e. What happens at the limits of the representable number range poses no serious problem in radix conversions. In particular, the tiny but notorious *denormalized* numbers of the P754 formats [4] require no special treatment.

Decimal number systems are analogous, using instead of bits b_k decimal digits d_k. In a decimal format with values

$$\pm d_0 . d_1 d_2 d_3 \cdots d_{P-1} \times 10^E$$

the intervals of interest are the decades $[10^{N-1}, 10^N]$ wherein the absolute spacing is 10^{N-P}. The spacing jumps by a factor of 10 from decade to decade to the right. The case $P=9$ is shown in the following diagram.



Our goal in this paper is to devise mappings between binary and decimal number systems that satisfy as nearly as practical the rules for correct rounding. What complicates the problem is that the two systems do not mesh compatibly; at some places the binary spacing doubles while at others the decimal spacing jumps tenfold.

We can be more precise about the relation between binary and decimal spacings. Suppose we have p-bit binary and P-digit decimal floating-point approximations to a real number Z:

$$b_0.b_1b_2\cdots b_{p-1}\times 2^e \;\approx\; Z \;\approx\; d_0.d_1d_2\cdots d_{P-1}\times 10^E\ ,$$

with $b_0=1$ and $d_0>0$. Then the binary and decimal spacings near Z are simply the units in the last place (ulps) of the respective approximations. They are

$$\mathrm{ulp}_2 = 2^{e-p+1} \quad \text{and} \quad \mathrm{ulp}_{10} = 10^{E-P+1}$$

from which we get the relation

$$\frac{\mathrm{ulp}_{10}}{\mathrm{ulp}_2} = \left(\frac{10^{-P}}{2^{-p}}\right)\times\left(\frac{10^{E+1}}{2^{e+1}}\right)$$

between ulps_{10} and ulps_2. The fixed ratio $10^{-P}/2^{-p}$ depends on the precisions of the binary and decimal formats. However, the ratio $10^{E+1}/2^{e+1}$ depends on Z. It varies between a maximum of almost 10, when Z lies in intervals of the form $[10^N, 2^n]$ where $10^N \approx 2^n$, and a minimum just above 1/2, in the corresponding intervals $[2^m, 10^M]$. So we deduce the formula

$$\frac{1}{2}\times\left(\frac{10^{-P}}{2^{-p}}\right) < \frac{\mathrm{ulp}_{10}}{\mathrm{ulp}_2} < 10\times\left(\frac{10^{-P}}{2^{-p}}\right) \qquad \textbf{(C)}$$

which is useful in bounding ulps_{10} and ulps_2 in terms of each other.

From formula **C** we can find roughly equivalent binary and decimal precisions. If we choose precisions p and P such that the ratio $10^{-P}/2^{-p}$ is about 1, then ulp_2 and ulp_{10} would be about the same size, up to the factor

$10^{E+1}/2^{e+1}$. For example, the P754 single format has precision $p = 24$; since 2^{-24} is about 6.0×10^{-8}, the corresponding decimal precision is somewhere between $P = 7$ and $P = 8$. The P754 double format has precision $p = 53$, with 2^{-53} about 1.1×10^{-16}; so the corresponding decimal precision is about 16.

1.3. A Distinguished Decimal Precision

Some applications demand that any representable binary floating-point value be obtainable by rounding an aptly chosen decimal number. That is, the decimals should be so dense as to *distinguish* the binary numbers. How many decimals are required? That is the question we turn to now.

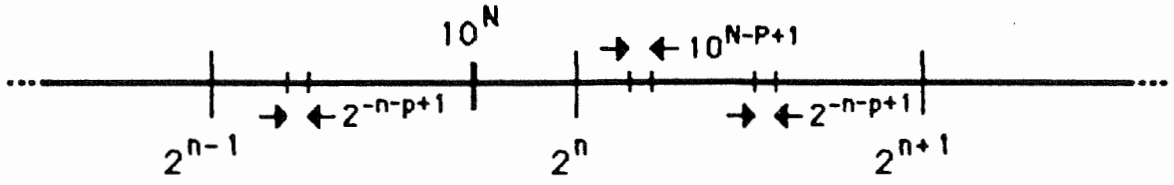
This separation property has been discussed in the literature before, for example in I. B. Goldberg's astute note [5] on the binary precision required to distinguish eight-digit decimal numbers. He worked in the opposite direction, distinguishing decimals with internal binary values, but the issues are the same. What we need for this paper will be redeveloped here.

The problem is, given binary precision p, to find the decimal precision P required to distinguish the binary numbers. A condition sufficient for distinction is given by the following:

Separation Requirement. For every binary number x, either x is exact in the decimal format, or x's nearest decimal neighbors $X^- < x < X^+$ are such that $X^+ - X^-$ is less than the distance from x to its nearest binary neighbor.

This requirement implies for every x that there is a decimal number nearer to x than to any other number in x's format. Thus it guarantees that some decimal number would round to x in a correctly rounded conversion; that is, it guarantees distinction.

To see how to satisfy the separation requirement, consider the number line below on which a power of ten is bracketed thus, $2^{n-1} \le 10^N < 2^n$, by adjacent powers of two. The spacings of p-bit and P-digit numbers in the respective binades and decades are shown, although the representable ticks are omitted for clarity.



If the separation requirement is satisfied in the interval $[10^N, 2^n]$, then it is surely satisfied throughout the entire decade $[10^N, 10^{N+1}]$ in which the decimal mesh is uniform while the binary spacing doubles across successive binades.

So it is enough to study the critical intervals $[10^N, 2^n]$. If P is the number of decimals carried and p is the number of bits, the separation requirement is equivalent to requiring that

$$\mathrm{ulp}_{10} = 10^{N-P+1} < 2^{n-p} = \mathrm{ulp}_2$$

hold over all pairs of corresponding N and n. Rewriting the inequality in the form

$$\frac{10^{-P+1}}{2^{-p}} < \frac{2^n}{10^N}$$

shows that $2^{-p} > 10^{-P+1}$ is a sufficient condition for separation, because $2^n > 10^N$. In the P754 single and double formats, with $p=24$ and $p=53$, respectively,

$$2^{-24} \approx 6.0\times10^{-8} > 10^{-8} \quad \text{and} \quad 2^{-53} \approx 1.1\times10^{-16} > 10^{-16} ,$$

so P=9 and P=17 satisfy the separation requirement.

We have derived the chain of inferences

$$10^{-P+1} < 2^{-p} \rightarrow \text{Separation Requirement} \rightarrow \text{Distinction} .$$

Now, can we complete the chain and show that all three conditions are logically equivalent? The answer in general is NO, but the explanation is deferred to the Nit-Picking at the end of the paper. The answer for P754 single and double is YES. To see that P=9 and P=17 are actually necessary for distinction, we need only consider the critical interval $[10^3, 2^{10}]$. There, the binary spacing 6.1×10^{-5} for p=24 is coarser than the decimal spacing 10^{-5} for P=9, but is almost twice as fine as the spacing 10^{-4} for P=8. So by the pigeonhole principle P = 8 could not achieve distinction. The situation for p=53 and P=17 is similar.

In the last section we looked at roughly equivalent binary and decimal precisions on the basis that $\text{ulp}_{10} \approx \text{ulp}_2$. Although the P754 single format gives about 7 or 8 significant digits of precision, P=9 is required to ensure that $\text{ulp}_{10} \leq \text{ulp}_2$ even in the most critical intervals $[10^N, 2^n]$. In general, the decimal precision P necessary and sufficient for separating binary numbers of precision p, is the smallest P satisfying $10^{-P+1} < 2^{-p}$. This may be thought of as a requirement that the widest relative spacing in the decimal format be just narrower than the narrowest relative spacing in the binary format.

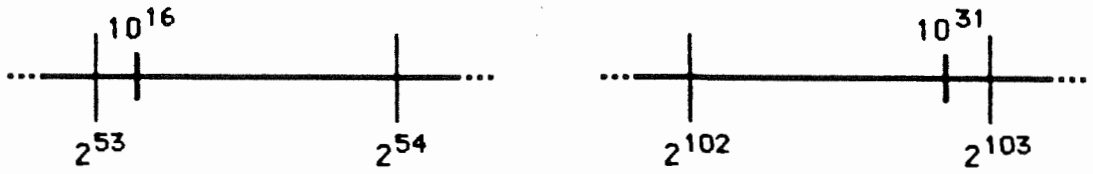
Now that we have fixed the relation between p and P, we can flip the ratios in formula **C** to bound ulp_2 in terms of ulp_{10}. The ratio $10^P / 2^p$ is about 59.6 for P754 single and 11.1 for double. Thus the spacings of 9-digit decimal numbers and P754 single format numbers satisfy

$$5.96\ \mathrm{ulp}_{10} \ < \ \mathrm{ulp}_2 \ < \ 119\ \mathrm{ulp}_{10}\ ,$$

and the spacings of 17-digit decimal numbers and P754 double satisfy

$$1.11\ \mathrm{ulp}_{10} \ < \ \mathrm{ulp}_2 \ < \ 22.2\ \mathrm{ulp}_{10}\ .$$

These bounds are nearly achieved in practice. Consider the two border cases $2^{53} \approx 10^{16}$ and $2^{103} \approx 10^{31}$ illustrated in the figures



for which the following table applies.

Approximate spacing ulp_2 as a multiple of ulp_{10}.		
	$[2^{53}, 10^{16}]$	$[10^{31}, 2^{103}]$
P754 single	107 ulp_{10}	6.04 ulp_{10}
P754 double	20 ulp_{10}	1.13 ulp_{10}

From these examples and the discussion above we see that the 9-digit decimal numbers are always at least six times as dense as P754 single format numbers, while in some intervals the 17-digit numbers just barely distinguish double format numbers. It is a remarkable coincidence that the P754 single and double formats reflect the near extremes of tightness in decimal encodings! We will return to the separation property later when we analyze imperfectly rounded conversions in §3.

1.4. Less than Perfect Rounding

Conversions using a computer's built-in floating-point arithmetic typically commit somewhat more than the expected rounding error. Just how imperfect may such conversions be, and still be accurate enough? We might attempt to preserve as many as possible of the important properties of ideal

conversions. Consider the following list, in which binary values are given in lower case (x, y), and decimal values in upper case (X, Y).

[Sign symmetry.] When rounding *to nearest* or *toward 0*, if $x \rightarrow X$, then $-x \rightarrow -X$; and if $X \rightarrow x$, then $-X \rightarrow -x$. When rounding *toward* $+\infty$, if $x \rightarrow X$, then when rounding *toward* $-\infty$, $-x \rightarrow -X$; similar relations hold for the conversion $X \rightarrow x$ and with the rounding directions swapped.

[Monotonicity.] If $x < y$, $x \rightarrow X$, and $y \rightarrow Y$, then $X \leq Y$. If $X < Y$, $X \rightarrow x$, and $Y \rightarrow y$, then $x \leq y$.

[Direction.] When rounding *toward* $+\infty$, if $x \rightarrow X$ then $x \leq X$, and if $X \rightarrow x$ then $X \leq x$. Similar inequalities hold when rounding *toward 0* or *toward* $-\infty$.

[Recovery.] If X is carried to at least 9 (17) decimals then $x \rightarrow X \rightarrow x$ when rounding *to nearest* in single (double). And if X is carried to no more than 6 (16) decimals then $X \rightarrow x \rightarrow X$.

[Sensibility.] Applied to numbers of reasonable size, conversions should be correctly rounded. For example, results like 3.0 → 2.99999...9 and 0.5 → 0.5000...01 from binary to decimal conversion are unacceptable.

[Consistency.] X should map to the same internal value x regardless of whether X appears in the source text of a program or is put in as data at execution time. Similarly, a value x should be displayed as the same decimal X (for a given format) regardless of the programming language or output medium used.

The consistency property often falls victim to system or language idiosyncracies. Perhaps the most bothersome situation can arise when a language compiler uses a different (imperfect) conversion scheme than the run-time I/O facility. In that case, a user might be unpleasantly surprised to

discover that the debugging statement

$$x := 3.14159265$$

has a different effect than does typing that decimal string in response to the prompt "Test value x = ?" at an interactive terminal.

Recovery of a binary number x from the chained conversion $x \to X \to x$ is guaranteed if the conversions are correctly rounded and if X is kept to decimal precision P sufficient to distinguish binary numbers with the precision p of x. We discussed the relation between P and p in the last section. Now we would like to carry the recovery property over to imperfectly rounded conversions. We must ensure that the total error in the two conversions is less than one ulp_2. Formula **C** bounds the binary to decimal error, measured in $ulps_{10}$, as a fraction of an ulp_2. The condition

$$\left(\frac{10^{-P+1}}{2^{-p}}\right) \times b \to d \text{ error in } ulp_{10} \quad + \quad d \to b \text{ error in } ulp_2 \quad < \quad 1\ ulp_2$$

is sufficient for recovery $x \to X \to x$. Measured in their respective ulps, the individual bounds are at least $\frac{1}{2}$ulp due to rounding. But the factor $(10^{-P+1}/2^{-p})$, which is about 1/6 for single and 9/10 for double, provides a cushion in binary to decimal conversions, so it is possible to keep the total error less than 1 ulp_2.

The factor $(10^{-P+1}/2^{-p})$ is the maximum relative spacing of full precision decimal numbers to representable binary numbers. The value 1/6 for the single format suggests that the 9-digit decimal numbers are so dense that perhaps a few full $ulps_{10}$ error could be tolerated in binary to decimal conversions without losing the recovery property. On the other hand, the factor 9/10 leaves little margin for extra error in binary to decimal conversion from the double format.

The properties listed at the beginning of this section are reasonable requirements for binary-decimal conversions but they are incomplete as a set of specifications. It is a simple exercise to invent bizarre conversions that satisfy these rules but almost always yield ridiculous results. What is needed is a bound on the extra rounding error incurred. The cryptic figure 0.47 ulp was put in proposed standard P754 as a worst-case bound, not to guarantee the properties listed above. In fact, it is too high for all conversions but binary to decimal from the single format in a directed rounding mode, and for that case it is lower than absolutely necessary to preserve the other properties. But we suspend further discussion of the error bounds until we have analyzed the algorithms below.

2. Algorithms

2.1. Correctly Rounded Conversions

We will look first at algorithms for correctly rounded binary-decimal conversions. The error properties of such conversions are already well known, thanks especially to an exhaustive series of papers by D. W. Matula[7]. But the algorithms themselves have not been discussed, due perhaps to their impracticality.

Consider conversion from the P754 single format to decimal. The input values will have the form

$$\pm b_0 . b_1 b_2 \cdots b_{23} \times 2^e \quad \text{where } -126 \le e \le +127 .$$

These values are representable exactly in the binary fixed point format

$$i_{127} i_{126} i_{125} \cdots i_2 i_1 i_0 . f_{-1} f_{-2} \cdots f_{-148} f_{-149}$$

and can be converted *exactly* to the decimal format

$$I_{38}I_{37}I_{36}\cdots I_2I_1I_0{\bullet}F_{-1}F_{-2}\cdots F_{-44}F_{-45}\cdots F_{-148}F_{-149}$$

with equally many fraction digits. Of course the decimal value will usually be rounded down to some more manageable length, depending on the output precision desired. The important point about such conversions is that they require arithmetic on a wide bit buffer for the binary input and a wide digit buffer for the decimal output.

There are several ways to perform the integer conversion. One is to repeatedly divide the binary integer buffer by a power of ten; then the successive remainders give the decimal digits from right to left. Another way is to scan the integer bits from left to right, accumulating a decimal value that must be doubled at each step. In yet another scheme the binary integer would be divided by a huge power of ten, *perturbed upward* a little bit, and then converted as a fraction.

A binary fraction may be converted to decimal by repeated multiplication by a power of ten; the successive integer parts give the decimal digits from left to right. For example, since 10 = 8+2, multiplication of a bit buffer by 10 can be accomplished by shifts of three and one bits, followed by an add of the shifted values. The case 1000 = 1024−16−8 is similar and provides three digits at each step.

Once the integer and fraction parts are converted as necessary, the decimal fixed point value, if not exact, must be rounded to the precision of the target format. In the worst case this entails propagating a carry across a string of nines, possibly causing a carry out of the left end. Correct rounding is possible – even in the half-way cases when the least significant digit output must be even – because the integer and fraction schemes above produce successive digits correctly. For example there is never a question whether a

string of digits "4999..." should actually be "5000...", as is the case with elementary transcendental functions. Only in the integer conversion requiring the small perturbation must care be taken not to confuse the perturbation with rounding error.

Further discussion of integer and fraction conversion algorithms may be found in [6, pp. 302-312] and [9, pp. 436-459]. Appendix D contains a sample implementation of correctly rounded conversions. The procedures are presented as a Pascal *unit* (in the notation of Apple III Pascal [2]) suitable for inclusion in a system library. They may be parameterized to support P754 single, double, or even double-extended format conversions.

Although the correctly rounded conversions are conceptually simple, all of the schemes discussed above suffer time penalties on machines without significant support for the wide binary and decimal quantities involved. For example, the first two integer schemes require that *all* integer digits be converted. Fraction conversion is somewhat simpler, and it has the advantage of producing digits from left to right, so it may be stopped when enough digits have been obtained to round to the target precision. The time and space penalties incurred are severe for operands of wide range and precision. The Pascal routines in the appendix require one 1400-bit packed BCD buffer and one 1000-bit binary buffer in order to perform P754 double format conversions. Such conversions are unsuitable for implementation, say, on a chip supporting the rest of a floating-point engine and presumably subject to time and memory tradeoffs. But they are ideal for low-end implementations either done entirely in software or lacking extended support for the algorithms of the next section.

2.2. Imperfect Conversions

In this section we look at algorithms for converting between decimal strings and the P754 binary floating-point formats. All arithmetic is performed in a P754 extended format, whose exact requirements are discussed at the end of the section. The only decimal operations required are exact conversions between decimal integers of modest length and integer values in the extended binary floating-point format.

The basic strategy in binary to decimal conversion is to scale the input value by a suitable power of ten so that, when rounded to an integer, the scaled value has the desired number of digits in its exact decimal representation. Together, this integer and the scale factor determine the decimal significant digits and exponent. Rounding errors can occur during binary to decimal conversion; floating-point overflow and underflow in the sense of P754 do not arise because because the decimal format has no range restriction. However, a kind of overflow arises if the decimal destination field has insufficient width to accommodate the desired number of significant digits and the computed exponent. What happens in this situation is highly system-dependent; further discussion is deferred to the Nit-Picking section at the end of the paper. What makes the following algorithm interesting is its near-minimal rounding error.

Algorithm B *(Binary to decimal conversion.)* Given a binary floating-point number xin, a positive integer N, and implicitly the current direction of rounding, this algorithm finds the significant digit and exponent components of the floating decimal string $\sigma d_1 \bullet d_2 d_3 \cdots d_N E dexp$ approximating xin. The named temporary variables are integers *LOGX* and *SCALE*, and extendeds x and y.

B0. [Special cases.] Dispatch zero, infinite and NAN values of xin.

B1. [Extend xin.] Set $x \leftarrow xin$. (x will be normalized.) Save $y \leftarrow x$. (x will be normalized.)

B2. [Log base 10.] Set $LOGX \leftarrow \lfloor \log_{10}(|x|) \rfloor$, perhaps underestimating by 1. (See algorithm L below.)

B3. [Scale factor exponent.] Set $SCALE \leftarrow N\text{-}LOGX\text{-}1$. (Rounding $x \times 10^{SCALE}$ to an integer should yield the N-digit significand.)

B4. [Scale x.] Scale x by 10^{SCALE} as in algorithm S below.

B5. [Round to integer.] Round x to an integer, according to *RMODE*.

B6. [Check for N digits.] If $|x| \geq 10^N$ then increase $LOGX$ by 1, restore $x \leftarrow y$, and go back to step B3. Otherwise, if $|x| < 10^{N-1}$ then replace x by 10^{N-1} with the sign of x. (The latter test is not necessary for all implementations. See the analysis of algorithm B for details.)

B7. [Significant digits.] Convert x to the signed decimal string $\sigma d_1 d_2 \cdots d_N$.

B8. [Exponent.] Convert $LOGX$ to the signed decimal string $dexp$. ▪

Algorithm B is designed for FORTRAN E-format conversions, where the number of significant digits is specified in advance. With a small modification, the algorithm can be applied as well to F-format conversions, where only the number of fraction digits is specified. Let a separate flag indicate whether E- or F-format output is desired; for the latter N specifies the number of fraction digits to be displayed; then $SCALE$ in step B3 is simply N (even if N itself is negative), and steps B2 and B6 are unnecessary. F-format conversion may suffer "format overflow" in step B7 if $|x|$ is too big to fit into the destination to receive it. In this case a helpful system might print the number in E-format with a modest number of digits.

Input conversion from decimal to binary is computationally simpler, but is open to several hazards associated with free-format character strings. For instance, if polynomial coefficients are read from a file built by an algebraic manipulation system with very high precision and range, what is to be done with 35-significant-digit numbers, or numbers with (outrageous) 13-digit exponents? Some problems lie outside the domain of the conversion routine. Literals in program text may be decomposed into significant digit and exponent strings during a compiler's lexical scan, and subjected to the arbitrary size constraints of the scanner. Will the compiler even recognize special values like $\pm\infty$ or NAN? Ideally, recognition of floating-point numbers should be the responsibility of a system routine. Figure 1 at the end of the paper shows how floating strings might be discovered. In any case, decimal strings might be constrained to have fewer than, say, 80 characters.

Algorithm D uses the conversion strategy of algorithm B above, in reverse. The significant digits are converted as a wide integer to be scaled by a suitable power of ten, whose exponent depends on the exponent field as well as the placement of the decimal point in the input string. Figure 2 at the end of the paper shows one way to parse floating strings into significant digit and exponent fields. Of course algorithm D is vulnerable to rounding errors; unlike algorithm B, it may also suffer overflow or underflow.

Algorithm D *(Decimal to binary conversion.)* Given the signed decimal strings $\sigma d_1 d_2 \cdots d_N$ (with $d_1 \neq 0$), and $dexp$, corresponding to the value $\sigma d_1 d_2 \cdots d_N.0 \times 10^{dexp}$, and implicitly the current rounding direction, this algorithm computes a corresponding binary floating-point number $xout$. The constant *NMAX* is the maximum number of significant digits that may be input. The named temporaries are integers *SCALE* and *LOST* and extended

value x.

D0. [Special cases.] Dispatch zero, infinite, and NAN strings.

D1. [Convert exponent.] Set integer $SCALE \leftarrow dexp$. (This will be exact except when $|dexp|$ is outrageously large, in which case $SCALE$ should be set to some huge value like 4000. This will produce a scaled value x that, while not outside the extended range, will provoke the suitable overflow or underflow in step D6.)

D2. [Excess digits.] Set $LOST \leftarrow 0$. If $N \leq NMAX$, skip to step D3. Otherwise, truncate the excess $N-NMAX$ digits $d_{NMAX+1}d_{NMAX+2}\cdots d_N$, setting $LOST \leftarrow 1$ if any of them are nonzero. Add $N-NMAX$ to $SCALE$. Go to step D4.

D3. [Canonical form.] Minimize $|SCALE|$ as follows. If $SCALE > 0$, pad the digit string on the right with up to $NMAX-N$ zeros, subtracting from $SCALE$ the number of zeros appended. Otherwise, if $SCALE < 0$ truncate up to $-SCALE$ trailing zeros, adding to $SCALE$ the number of zeros dropped.

D4. [Significant digits.] Convert the digit string: $x \leftarrow \sigma d_1 d_2 \cdots d_M$. (Steps D2 and D3 assure that $1 \leq M \leq NMAX$, so the conversion is exact.)

D5. [Scale x.] Set Scale x by 10^{SCALE} as in algorithm S below.

D6. [Round.] Logically OR $LOST$ into the least significant bit of x. Convert to storage format: $xout \leftarrow x$. (This final step may overflow or underflow. If there is no trap, the result is as in P754. If there is a trap two cases arise. If the overflow or underflow was "reasonable" then a correctly wrapped-around result is sent to the trap hanler in lieu of $xout$. If the exponent of x cannot be wrapped-around to within the range of $xout$, then the value of x, though it may be available to the trap handler, is meaningless since the decimal exponent may have been set arbitrarily in step D1; in this case the most useful information is the original decimal

string, but it may not be available.) •

How much extended arithmetic is actually needed? We have seen that discriminating binary-decimal conversions require rather more decimal precision than binary. For example, nine decimals are required for conversion from the P754 single format. Since

$$10^9 > 2^{24} \approx 1.7 \times 10^7 ,$$

and both algorithms B and D require that a nine-digit integer be stored exactly, it is clear that conversions cannot be carried out entirely in the single format with its 24 significant bits. Proposal P754 includes optional extended formats for just such calculations. These formats follow the P754 conventions for, say, rounding and the handling of over/underflow but their particular encoding is system-dependent. P754 requires that there be at least 8 extra bits of precision and 3 extra exponent bits in single-extended, and 11 extra bits of precision and 5 extra exponent bits in double-extended. Since

$$10^9 < 2^{24+8} \approx 4.3 \times 10^9 ,$$

any nine decimal significant digit string can be held, as an integer, in the single-extended format, so the scalings of algorithms B and D can be performed with a few extra bits to suppress rounding errors. We will see later that the numbers 8 and 11 of extra bits are very tight – there is hardly a bit to spare in providing accurate binary-decimal conversions.

If an extended format is not implemented in hardware, algorithms B and D may be less attractive than the correctly rounded conversions of the previous section. But if time, space, or even compatibility constrain one to the methods of this section, some provision must be made in software. The only arithmetic operations required for the conversions are multiplication, divi-

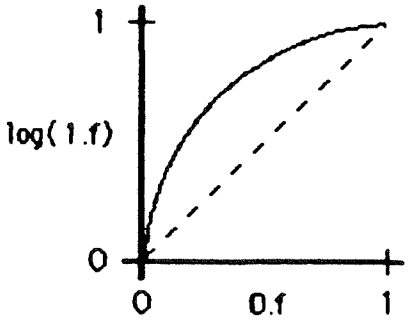
sion, comparison, round to integer, and conversion to and from the single or double formats being supported. The operations are simplified by the absence of special cases involving infinite and NAN operands and by the restriction to results which usually suffer only rounding errors (the conversion in step D8 of algorithm D may over/underflow). So it is feasible to build these functions from a reasonable complement of intrinsic integer operations.

2.3. A Poor Man's Logarithm

Step B2 of algorithm B calls for the calculation of $\lfloor \log_{10}(z) \rfloor$, where z is a positive normalized number. It turns out that a suitable approximation *LOGX*, perhaps too low by 1, may be found with just a few integer operations. If we express z in the form $2^e \times 1.f$, we can see that

$$\log_{10}(z) = \log_{10}(2) \times \log_2(z) = \log_{10}(2) \times (e + \log_2(1.f)).$$

A look at the graph of $\log_2(1.f)$ versus $0.f$



and a little calculus indicate that $0.f \le \log_2(1.f)$ with a maximum deviation of about 0.086. So $\log_2(2^e \times 1.f)$ is approximated from below by $e + 0.f$, that is "$e.f$" as a fixed-point number! This suggests the following simple procedure for computing *LOGX*.

Algorithm L *(Log base 10.)* Given a positive binary floating-point number z, this algorithm computes *LOGX* as $\lfloor \log_{10}(z) \rfloor$ or the next integer toward $-\infty$.

The temporary variables $LOG2$ and $L2X$ hold fixed point values.

L0. [$\log_{10}(2)$.] Set $LOG2 \leftarrow 0.4D104D427\cdots_{16}$, $\log_{10}(2)$ in hexadecimal, truncated to a convenient length like 8 or 16 bits.

L1. [$\log_2(z)$.] Set $L2X \leftarrow e + 0.f$, where $z = 2^e \times 1.f$. The fraction $0.f$ may be truncated to as few as 6 bits.

L2. [Ensure a lower bound.] If $L2X < 0$, increase $LOG2$ by one unit in its last place.

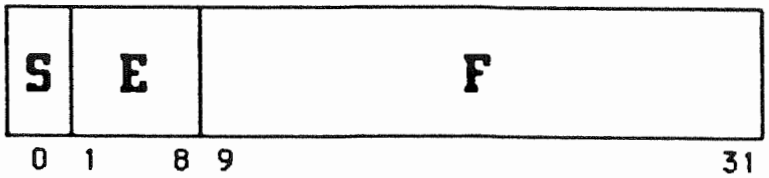
L3. [$\log_{10}(z)$.] The result is $LOGX \leftarrow \lfloor LOG2 \times L2X \rfloor$. •

The maximum possible error in $LOG2 \times L2X$ is approximately $\log_{10}(2) \times 0.086 \approx 0.026$, caused by the linear approximation to $\log_2(z)$. By comparison, the errors due to truncating low-order bits of $e.f$ and rounding $\log_{10}(2)$ are small. In any case, all errors are toward $-\infty$. Only rarely will the computed $LOGX$ be wrong, and then it will be off by 1. If we assume that $\log_2(1.f)$ is uniformly distributed between 0 and 1 [6 pp. 238-247], then the average induced error in $LOGX$ is about

$$\log_{10}(2) \times \int_0^1 (\log_2(1+t) - t)dt \approx 0.017 .$$

Assuming that ($\log_{10}(z)$ mod 1), too, is uniformly distributed between 0 and 1, this means that $LOGX$ will fall short less than 2% of the time and then only for values z barely greater than powers of ten.

As usual, the analysis is more complicated than the implementation. To illustrate the ideas, we can compute $\lfloor \log_{10}(Y) \rfloor$ where Y is a positive, normalized number in the P754 single format. Y is encoded as a 32-bit string



representing the value

$$Y = (-1)^S \times 2^{E-127} \times 1.F \quad .$$

The sign bit S is zero for positive Y. So to approximate $\log_2(Y)$ we need only subtract the bias 127 from E and imagine a binary point between E and F. Then the product with an approximate $\log_{10}(2)$ is essentially an integer operation. The following assembly language sequence will compute $\lfloor \log_{10}(Y) \rfloor$ on a Zilog Z8000 microprocessor [10].

Program L *(Log of a single format number.)* Given the value Y in register RR2, compute $\lfloor \log_{10}(Y) \rfloor$. (On the Z8000, RR2 refers to the pair of 16-bit registers R2 and R3; RH2 and RL2 refer to the most and least significant bytes of R2.)

```
        LD      R3, #%4D10      ! Overwrite the low-order half of Y in R3 with log10(2),
                                  chopped, whose implicit binary point is to the left of
                                  R3. The '%' flags the constant as hexadecimal. !
        SLA     R2, #1          ! Shift the high-order half of Y left 1 bit, leaving the
                                  exponent in RH2 and the seven leading fraction bits,
                                  followed by a 0 bit, in RL2. !
        SUBB    RH2, #%7F       ! Unbias the exponent to get a two's complement ap-
                                  proximation to log2(Y), with an implicit binary point
                                  between RH2 and RL2. !
        JR      PL, PLUS        ! Chopped log10(2) is fine if unbiased E ≥ 0. !
        INC     R3, #1          ! Round log10(2) up. !
PLUS:   MULT    RR2, R2         ! RR2 gets R2×R3 ≈ log10(Y) in two's complement
                                  with the binary point between RH2 and RL2. The ap-
                                  proximate ⌊log10(Y)⌋ is in RH2 since in two's comple-
                                  ment arithmetic the floor function is achieved by
                                  truncation. ! •
```

2.4. Scaling in Algorithms B and D

This section contains a scaling algorithm that lies at the heart of both algorithms B and D.

Algorithm S *(Scaling in binary-decimal conversions.)* Given an extended floating-point number x, an integer *SCALE*, and implicitly the current direction of rounding, this algorithm computes $x \times 10^{SCALE}$, rounded toward zero, and sets the least significant bit of x to 1 if any nonzero bits have been rounded off. Extended variable z holds the value 10^{SCALE}, possibly rounded. The pseudo-variable *RMODE* contains the current rounding direction. The integer pseudo-variable *IXFLAG* corresponds to the P754 inexact flag; it signals rounding errors in floating-point operations. The values *RMODE* and *IXFLAG* are saved in and restored from the variables *RSAVE* and *IXSAVE*.

S0. [Rounding direction for scale factor.] Set *RSAVE* ← *RMODE*. If *RSAVE* = ***to nearest***, skip to step S1. (These next tests handle the other three, directed, roundings.) If *RSAVE* = ***toward*** $-\infty$ and $x < 0$, or *RSAVE* = ***toward*** $+\infty$ and $x > 0$, set *RMODE* ← ***toward*** $+\infty$; otherwise set *RMODE* ← ***toward*** $-\infty$. Finally, if *SCALE* < 0, reverse the sense of *RMODE*.

S1. [Scale factor.] Set $z \leftarrow 10^{|SCALE|}$. (See algorithms P and Q below. Both algorithms B and D are designed so that z will not overflow the extended range.)

S2. [Perform scaling.] Save *IXSAVE* ← *IXFLAG* and set *IXFLAG* ← 0. Set *RMODE* ← ***toward*** 0. If *SCALE* > 0, set $x \leftarrow x \times z$, otherwise set $x \leftarrow x / z$. (*IXFLAG*, assumed to take the values 0 — clear and 1 — set, records any rounding error in the multiplication or division of x by z.)

S3. [Collect roundoff.] Logically OR *IXFLAG* into x's least significant bit. Restore *RMODE* ← *RSAVE*. Logically OR *IXSAVE* into *IXFLAG*. ▪

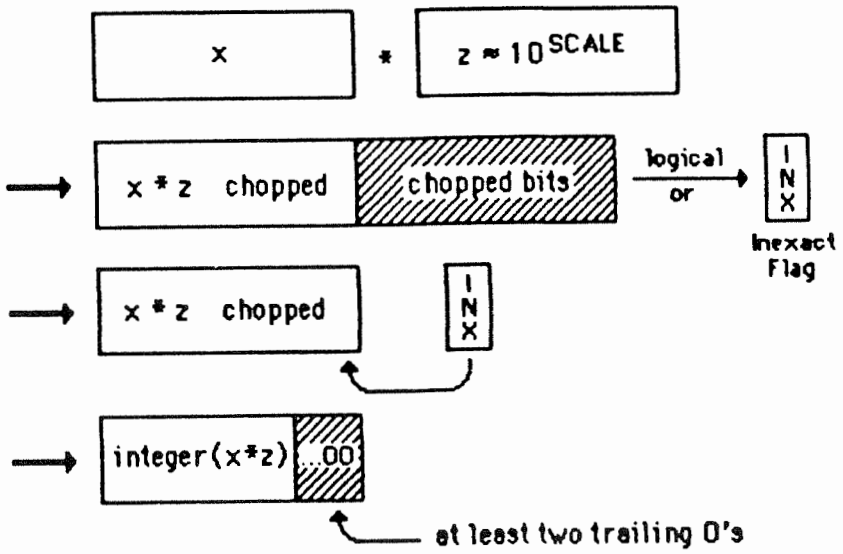
If the scale factor $10^{|SCALE|}$ cannot be represented exactly in the extended variable z, then it is rounded in a direction that guarantees that the ultimate result in algorithm B or D will honor the intended rounding

direction.

Algorithm S is vulnerable to errors in step S1 when $10^{|SCALE|}$ is computed and in S2 when the input x is scaled. However, the latter error may be avoided. Since both algorithms B and D will round the scaled value x to a precision narrower than extended, any low-order bits chopped off in step S2 will participate correctly as "guard bits" for the rounding in step B5 or step D6, if they are logically OR'ed into the least significant bit of x. And when rounding *toward 0*, the P754 inexact exception flag, *IXFLAG*, contains precisely the logical OR of all chopped bits. The figure below tells the story.

2.5. Evaluating Positive Powers of Ten

Step S1 of algorithm S involves the calculation of a nonnegative power of ten in an extended variable z. Since it is this calculation that contributes to any error algorithms B and D commit in excess of the expected rounding error, it is worthwhile to compute z as accurately as possible.



Avoiding an unnecessary error.

Expressing nonnegative powers of ten in the form $10^k = 2^k \times 5^k$, we see that 10^k is exactly representable in a binary floating-point format with p significant bits and reasonable exponent range only if $5^k < 2^p$. The P754 single-extended and double-extended formats, with 32 and 64 significant bits, can accommodate powers of ten up to 13 and 27, respectively, since

$$5^{13} < 2^{32} < 5^{14} \quad \text{and} \quad 5^{27} < 2^{64} < 5^{28} .$$

Unfortunately, these exact powers of ten are not sufficient for scaling in steps B6 and D7. For example, in conversion from the single format to decimal, the input values *xin* to algorithm B satisfy

$$-45 \le \left\lfloor \log_{10}(\mid xin \mid) \right\rfloor \le 38 ,$$

with an asymmetric range because *xin* may be a tiny denormalized number [4]. Then, since the digit count N can range from 1 to 9, the value *SCALE* computed in step B3 can range from −38 to 53. Somehow the powers of ten up to 10^{53} must be computed for scaling in single format conversions. Happily, there is a strategy blessed by a stroke of good luck.

Suppose that the exact values 10^0, 10^1, ..., 10^{13} are available, either from a table or to be computed on the fly. And suppose there is available the table of values:

$$\begin{aligned} P_{13} &= 0.9184\text{E}72\text{A}_{16} \times 2^{44} &&= 10^{13} , \\ P_{27} &= 0.\text{CECB8F28}_{16} \times 2^{90} &&\approx 10^{27} \times (1 + 2^{-36}) \text{ and} \\ P_{40} &= 0.\text{EB194F8E}_{16} \times 2^{133} &&\approx 10^{40} \times (1 - 2^{-35}) . \end{aligned}$$

Given the table values above algorithm P below will compute any nonnegative power of ten up to 10^{53} with *just one* rounding error, regardless of the rounding mode. This is possible because of the extraordinary accuracy of the rounded values P_{27} and P_{40} and because of extra care in a few special

cases. And it is fortunate since, as we will see in the analysis of the next sections, accuracy to the last bit is required to guarantee monotonicity in algorithms B and D for single format conversions.

Algorithm P *(Nonnegative power of 10, single format.)* Given $N \geq 0$ and implicitly the current rounding direction, compute extended $z \approx 10^N$ with the property that $z \geq 10^N$ if rounding *toward* $+\infty$ and $z \leq 10^N$ if rounding *toward* 0 or $-\infty$. The integer pseudo-variable *IXFLAG* corresponds to the P754 inexact result flag.

P0. [Exact case 0-13.] If $N > 13$ then set $IXFLAG \leftarrow 1$ and go to step P1. Otherwise set $z \leftarrow 10^N$, exactly and exit.

P1. [Case 14-26.] If $N > 26$ then go to step P2. Otherwise set $z \leftarrow P_{13} \times 10^{N-13}$ and exit.

P2. [Case 27-40.] If $N > 40$ then go to step P3. Otherwise set $z \leftarrow P_{27} \times 10^{N-27}$. If N is either 27 or 28 and the rounding mode is *toward* $-\infty$ or *toward* 0 then subtract 1 in the last place of z. Exit.

P3. [Case 41-53.] Set $z \leftarrow P_{40} \times 10^{N-40}$. If N is either 42 or 48 and the rounding mode is not *toward* 0 then add 1 in the last place of z. •

Conversions to and from the P754 double format are more complicated. With 64 significant bits in the double-extended format, powers of ten up to 10^{27} can be represented exactly. But the wider exponent range of the double format requires powers up to 10^{340}, in order to convert the tiniest denormalized number. The strategy here is similar to algorithm P above except that the table of powers of ten depends on tradeoffs among time, space, and accuracy. Fortunately, it is not necessary to produce perfectly rounded powers of ten as was the case above.

Algorithm Q exploits a carefully chosen table of increasing powers of ten: $pten(1) = 10^{27}$, $pten(2)$, $\cdots$, $pten(IMAX)$. These values are kept in the extended format, and all but the first are rounded. Let $pexp(1) = 27$, $pexp(2)$, ... $pexp(IMAX)$ be the corresponding decimal exponents. Then the following algorithm computes 10^N with a loop that multiplies the necessary table values, followed by a final multiply by an exact power of ten. The directed rounding modes are honored in the sense that all rounding errors have the correct sign.

Algorithm Q *(Nonnegative power of 10, double format.)* Given $N \geq 0$ and implicitly the current rounding direction, compute $z \approx 10^N$ with the property that $z \geq 10^N$ if rounding *toward* $+\infty$ and $z \leq 10^N$ if rounding *toward 0* or $-\infty$. The temporaries used are integer I and extended z. The integer pseudo-variable *IXFLAG* corresponds to the P754 inexact result flag.

Q0. [Initialize.] Set $I \leftarrow IMAX$ and set $z \leftarrow 1.0$.

Q1. [Check threshold.] If $N < pexp(I)$, skip to step Q3.

Q2. [Scale z.] Set $z \leftarrow z \times \text{FIXED}(pten(I))$, and decrease N by $pexp(I)$. (The value $pten(I)$, which is kept rounded to nearest, might require an adjustment of 1 in its last place to comply with a directed rounding mode. It suffices to keep an array $pfix(1)$, $pfix(2)$, ... $pfix(IMAX)$ of integers with value 0, +1, or −1 according to whether the corresponding table entry is exact or is rounded up or down. In any case, for the table values suggested below, the fix never amounts to more than a change in the low order 16 bits of $pten(I)$, that is, a simple integer operation.) If $pfix(I)$ is nonzero then set $IXFLAG \leftarrow 1$.

Q3. [Iterate.] Decrease I by 1. If $I > 0$, go back to step Q1.

Q4. [Last multiply.] Set $z \leftarrow z \times 10^N$. ($N \le pexp(1)$ so 10^N is exact.) •

If space is to be economized, a good choice for the table *pten*() is:

$$\begin{aligned}
0.\text{CECB8F27F4200F3A}_{16} \times 2^{90} &= 10^{27}, \\
0.\text{D0CF4B50CFE20766}_{16} \times 2^{183} &\approx 10^{55} \times (1+2^{-76}), \\
0.\text{DA01EE641A708DEA}_{16} \times 2^{359} &\approx 10^{108} \times (1+2^{-67}) \text{ and} \\
0.\text{9F79A169BD203E41}_{16} \times 2^{885} &\approx 10^{206} \times (1-2^{-67})
\end{aligned}$$

Given 10^0 through 10^{27}, exactly, any power of ten through 10^{340} can be computed with at most three multiplications, using at worst two rounded table values. The rounded table entries are so accurate that, when rounding *to nearest*, the error bound in any computed power of ten will be dominated by the error in the multiplications alone. A conservative error estimate would be

$$\begin{aligned}
10^N_{comp} &= 10^N_{exact} \times (1 \pm 2^{-64})^3 \times (1 \pm 2^{-67})^2 \\
&= 10^N_{exact} \times (1 \pm (7/2) \times 2^{-64})
\end{aligned}$$

for a worst case bound under four rounding errors even when computing the largest required power of ten. In any of the directed rounding modes the error estimate is

$$\begin{aligned}
10^N_{comp} &= 10^N_{exact} \times (1 \pm 2^{-63})^3 \times (1 \pm 2^{-63})^2 \\
&= 10^N_{exact} \times (1 \pm 5 \times 2^{-63})
\end{aligned}$$

where the sign of the error depends upon the direction of rounding.

Two more accurate variants of this scheme are worth considering although the accuracy above is sufficient. A table of 10^0 through 10^{27} along with a table of the powers

$$10^{55},\ 10^{83},\ 10^{111},\ 10^{139},\ \cdots,\ 10^{27+28\times k},\ \cdots$$

permits the evaluation of any power of ten with at most one multiplication, reducing loop Q1-Q3 in to just one pass. This scheme suffers at most two rounding errors, one inherited from the latter table value and one from the multiplication, but it requires about forty extended table entries to reach 10^{340}.

A more extravagant form of the table just mentioned can produce any 10^N with one multiplication, and with a guaranteed error bound of $\frac{1}{2}$ulp$_2$ when rounding to nearest. Rather than using values spaced by the factor 10^{28} as above, it uses a denser table carefully chosen to produce correctly rounded intervening powers. Experiments indicate that about sixty table entries would be required just to achieve results correctly rounded to nearest [11].

When the value z in algorithm P or Q can be computed exactly, algorithms B and D are guaranteed to suffer at most one rounding error – in step B5 or D6. Whether z is exact depends on the value $SCALE$ in the algorithms B and D. If $|SCALE|$ does not exceed 13 in single conversions and 27 in double conversions then $10^{|SCALE|}$ can be computed exactly in the single extended and double extended formats, respectively. This accounts for the ranges in Table 3 of the proposed standard P754 [1].

2.6. Testing Algorithms B and D

Of course the best way to test a program is to compare its results with the right answers. Fortunately that is possible, if only the algorithm for correctly rounded conversions is implemented along with algorithms B and D. Over what ranges should the two programs agree? The analysis in §3 shows that the key to correct conversions in algorithms B and D is a correct scale factor $10^{|SCALE|}$. For single format conversions, algorithm D is correct

for all 9-digit values in the range

$$100000000. \times 10^{-13} \quad \text{to} \quad 999999999. \times 10^{13} \ ,$$

and algorithm B is correct if its output lies in this range. Smaller values, down to 10^{-13} itself, may be converted correctly if the number of significant digits involved diminishes accordingly. For example, the decimal string "1.234e−10" would be cast as the value $000001234. \times 10^{-13}$ by step D3 of algorithm D, lending itself to correct conversion.

When correctly rounded conversion is not guaranteed, how far off can algorithms B and D be? Not more than an ulp in the destination format, as we will see. So let us consider binary to decimal conversion from an input x. Let X_m, X_n, and X_p be the decimal values resulting from algorithm B with rounding *toward* $-\infty$, *to nearest*, *toward* $+\infty$, respectively. Let C_m, C_n, and C_p be the corresponding correctly rounded values. Finally, suppose x is not exactly representable in the decimal format. Then C_m and C_p differ by one ulp_{10} and C_n is one or the other of those values. Ideally, the corresponding C's and X's should match. But this may not hold for huge or tiny x. Then, X_n is in error by less than an ulp so it too must be one of C_m or C_p, though not necessarily the right one, C_n. And the direction property ensures that

$$X_m \le C_m \le X_n \le C_p \le X_p \ .$$

One of the innermost inequalities is equality, so the other is strict. And one of the outermost inequalities is equality since the 0.47 ulp_{10} bound on extra error guarantees that X_m and X_p differ by at most 2 ulps_{10}. Note that this discussion carries over to algorithm D as well. These facts about the interlacing can be used with the correctly rounded conversions to test algorithms B and D.

A byproduct of the interlacing is the fact that another implementation of algorithms B and D must produce corresponding values Y_m, Y_n, and Y_p satisfying the same relation to the C's, and so differing from the X's by at most an ulp_{10} apiece.

The recovery property leads to a different kind of test for the algorithms. It is particularly convenient since no decimal manipulations are involved. Simply run the conversion $x \to X \to y$ to full decimal precision, rounding *to nearest*, and check that $x=y$. Recall that recovery is most difficult in intervals of the form $[10^E, 2^e]$, where the decimals are sparsest. A set of interesting intervals is given below. The center column is suitable for single format conversions. The outer columns span the range of the double format. Reciprocating the endpoints produces the intervals $[2^{-e}, 10^{-E}]$ wherein the decimals are relatively dense with respect to the binary values. Other pairs E and e can be obtained by noting that the nearer E/e is to $\log_{10}(2)$, the more nearly equal are $10^E \approx 2^e$.

A flavor of recovery is available for the directed roundings, too. Convert $x \to X$, and then from $X \to y$, rounding first *toward* $+\infty$, then *toward* $-\infty$. How are x and y related? An exhaustive analysis shows that, so long as X is carried to full decimal precision, y is either x or the next representable number left of x. The key observation, based on bounds from §3, is that even when the decimals are sparsest, X may be slightly more than an ulp_{10} greater than

Intervals $[10^E, 2^e]$ where decimals are sparse

E	e	E	e	E	e
−308	−1203	−28	−93	59	196
−292	−970	−16	−53	121	402
−146	−485	−1	−3	298	990
		3	10	304	1010
		31	103		

x but is certainly less than the next binary number to the right of x. A similar bound applies if the sense of the rounding modes is reversed.

3. Analyses

3.1. Analysis of Algorithm B

Algorithm B scales the exact input *xin* by a power of ten and rounds the result to an integer. Together that integer and the scale factor determine the decimal significant digits and exponent. This process is vulnerable to two rounding errors. The scale factor in step S1 of algorithm S will be computed as

$$10^{|SCALE|} \times (1 \pm \delta)$$

where δ is nonzero precisely when $SCALE$ is so large that $5^{|SCALE|}$ cannot be represented exactly in the extended format. Then, an error can be committed in step B5 when the scaled value x is rounded to an integer. The difference between algorithm B and a correctly rounded conversion is the error δ in the scale factor.

What about the multiply or divide in step S2? The discussion following algorithm S shows how to avoid any *extra* error there. The key for single format conversions is that the result x of steps B5 and B6 is an integer less than 10^9. Assuming x is normalized in a format with at least 32 significant bits, at least two of its trailing bits must be 0. The situation is similar for double format conversions in 64 significant bits.

Now let us bound the extra error incurred in step S1. First, consider conversions from the single format with rounding *to nearest*. The value to be rounded in step B5 has the form

$$x \times 10^{SCALE} \times (1 \pm \delta) \ .$$

Since the ultimate result of this scaling is an integer value, the extra error, in ulps_{10}, is just

$$x \times 10^{SCALE} \times (\pm\delta)\,\text{ulp}_{10} \ .$$

We saw in the discussion of algorithm P that the relative error δ may be as high as 2^{-32}. So its absolute contribution the final error is bounded by

$$(10^9 \times 2^{-32})\,\text{ulp}_{10} \approx 0.23\,\text{ulp}_{10} \ .$$

The notorious $0.47\,\text{ulp}_{10}$ error bound that has appeared in many drafts of P754 was based on an analysis of algorithm P in which 31 rather than 32 bits of precision were kept for intermediates. Now it is known that 32 bits are required. The bound 0.47 still applies, however, as the maximum error in single conversions with a directed rounding mode.

We can bound the extra error in ulps_{10} for any binary to decimal conversion by choosing appropriate values for δ and the number of significant digits to be delivered. The following table gives values relevant to P754 when output is delivered to the maximum decimal precision, namely 9 digits for single and 17 for double. If k fewer than the maximum number of digits are delivered, the error bound is smaller by a factor of 10^k.

	to nearest		*directed*	
format	δ	bound in ulp_{10}	δ	bound in ulp_{10}
single	2^{-32}	0.23	2^{-31}	0.47
double	$(7/2) \times 2^{-64}$	0.019	5×2^{-63}	0.054

3.2. Pathologies in Algorithm B

Steps B3-B6 of algorithm B are a loop whose implicit termination condition is $10^{N-1} \le |x| < 10^N$, where N is the number of significant digits to be

output. Does the loop actually terminate, and does it impact the rounding analysis? From the discussion of the last section, we can assume that the scaling operation in S2 is carried out exactly since its error is subsumed in step B5.

First, suppose that the input *xin* is $10^D \times (1 + \gamma)$ for some γ less than, say, 1/2. Then the scaled value in step S2 is

$$x = 10^{N-1} \times (1 + \gamma) \times (1 \pm \delta) ,$$

if *LOGX* was computed correctly in step B2. Can $(1 + \gamma) \times (1 \pm \delta)$ be less than 1 before x is rounded to an integer? If so, the result of step B5 could fall below 10^{N-1}. Positive γ is a relative measure of how much the single or double input value *xin* exceeds 10^D, while δ is the relative error of the scale factor when computed to extended precision. So the scenario is possible only for very small γ. A careful inspection of the powers of ten expressed in binary reveals that the answer to the last question is *NO* for single and *YES* for double. For example, $10^{303} \times (1 + 2^{-62})$ is representable in double; it may be rounded by a scale factor from algorithm Q with δ as large as 5×2^{-63}, depending on the rounding mode and the number of digits desired. Thus, the test against 10^{N-1} may be omitted for single conversions, but it is necessary for double conversions, if only for rare circumstances. Note that the corrective measure, forcing the magnitude up to 10^{N-1}, shrinks an error which already lies within the computed bound.

If *LOGX* was miscalculated as $\lfloor \log_{10}(x) \rfloor - 1$ in step B2, which may happen for γ less than about 0.06, the scaled value above would be

$$x = 10^{N} \times (1 + \gamma) \times (1 \pm \delta) .$$

This case is benign if x rounds down to less than 10^N; if it does not, *LOGX* is corrected and the situation is that of the last paragraph.

Now suppose the input value *xin* is $10^D \times (1 - \gamma)$, for some γ less than 1/2. In this case *LOGX* is always correct in step B2. So the scaled value x in step S2 is

$$x = 10^N \times (1 - \gamma) \times (1 \pm \delta) \ .$$

The scaled value x falls out of range if $(1 - \gamma) \times (1 + \delta)$ is at least 1. As above, this may only happen for some rare double format conversions in which γ is very tiny. If this occurs, *LOGX* is increased to $\lfloor \log_{10}(x) \rfloor + 1$ and the scaling is retried. The scaled value is then

$$10^{N-1} \times (1 - \gamma) \times (1 \pm \delta) \ .$$

If the result of step B5 is less than 10^{N-1} it will be forced up to 10^{N-1}, satisfying the stated error bound.

We can conclude from all this that the branch back to step B3 will be taken at most once, so long as *LOGX* is in error by no more than 1; when the branch is taken, the loop is guaranteed to terminate after the second pass.

3.3. Analysis of Algorithm D

Like algorithm B, algorithm D is vulnerable to two rounding errors, one in the evaluation of the scale factor in step S1 and another when the scaled value is rounded to the destination precision in step D6. And algorithm D exploits the same trick with the inexact exception flag and chopped arithmetic to avoid an extraneous error in step S2.

The conversion of the significant digit string in step D4 is exact, once any excess digits are truncated in step D2. Recording the presence of lost nonzero digits in the flag *LOST* assures that the directed rounding modes will be honored, but in no way takes the place of a very wide decimal buffer for the digit string. For example, a P754 single format number w between one-

half and one has the value

$$w = 2^{-1} + b_2 \times 2^{-2} + b_3 \times 2^{-3} + \cdots + b_{24} \times 2^{-24}$$

where the b_j are either 0 or 1. Since any such number can be closely represented by a decimal fraction of 24 digits, it takes just 25 digits to represent values half way between a pair of them. Truncating all but the first 9 digits in algorithm D dooms any prospect of perfect rounding *to nearest.*

The error analysis parallels that of algorithm B exactly. It is the error in step S1 that contributes to any error beyond what is expected in the rounding in step D6. Let us use the P754 single format for illustration. Ideally, algorithm D computes

$$x \times 10^{SCALE} = 2^n \times (b_0 b_1 \cdots b_{23} \cdot b_{24} b_{25} \cdots) ,$$

where the binary point is aligned so that, as in algorithm B, it is the fraction part that is rounded off to produce the delivered result. When an error is committed by algorithm P, what is computed is

$$x \times 10^{SCALE} \times (1 \pm \delta) = 2^n \times (b_0 b_1 \cdots b_{23} \cdot b_{24} b_{25} \cdots) \times (1 \pm \delta) .$$

So the error, expressed in ulps_2 is

$$(b_0 b_1 \cdots b_{23} \cdot b_{24} b_{25} \cdots) \times (\pm\delta) ,$$

leading to the bound $(2^{24} \times \delta \,\text{ulp}_2)$. If we assume that, when rounding *to nearest*, the scale factor will suffer at most one rounding error in extended precision, then the extra error is bounded by

$$(2^{24} \times 2^{-32})\,\text{ulp}_2 \approx 0.0039\,\text{ulp}_2 .$$

The following table gives the error bounds for P754 conversions.

	to nearest		*directed*	
format	δ	bound in ulp_2	δ	bound in ulp_2
single	2^{-32}	0.0039	2^{-31}	0.0078
double	$(7/2) \times 2^{-64}$	0.0017	5×2^{-63}	0.0049

These bounds hold for all applications of algorithm D, unlike those of algorithm B, which were parameterized according to the number of decimal digits produced.

Algorithm D is subject to over/underflow problems, since the exponent field of the decimal input may contain values far out of the range of the target format. It is only in step D2 that care must be taken to screen out unreasonable exponent values. Since the range of the extended intermediates exceeds that of the target variable, it is possible to replace unreasonable exponents with huge but reasonable ones and still achieve the correct over/underflow response in steps S2 and D6.

3.4. Accuracy Revisited

Now that we have analyzed algorithms B and D we can determine whether they actually satisfy the accuracy requirements set forth earlier in the paper. Was it all worth it?

The sign symmetry and rounding direction properties are built right in to both algorithms B and D, so they are easily seen to hold. The sensibility property holds since, for numbers of reasonable size, algorithms P and Q compute $10^{|SCALE|}$ exactly, so conversions in both directions are correctly rounded. The consistency property is a matter of system convention.

The recovery property is verified using formula C and the absolute error bounds of algorithms B and D. Earlier we derived the inequality

$$\left(\frac{10^{-P+1}}{2^{-p}}\right) \times b \to d \text{ error in } \text{ulp}_{10} \;+\; d \to b \text{ error in } \text{ulp}_2 \;<\; 1\ \text{ulp}_2$$

which gives a sufficient condition for the recovery property. Now we can fill in the blanks. The ratio $10^{-P+1}/2^{-p}$ is just under 1/6 and 9/10 for the P754 single and double formats, respectively. Using the values from the tables of the last few sections, we can write

$$(1/6) \times 0.73\ \text{ulp}_{10} + 0.5039\ \text{ulp}_2 \approx 0.63\ \text{ulp}_2 < 1\ \text{ulp}_2$$

for single, and

$$(9/10) \times 0.519\ \text{ulp}_{10} + 0.5017\ \text{ulp}_2 \approx 0.97\ \text{ulp}_2 < 1\ \text{ulp}_2$$

for double. So binary to decimal to binary conversion is the identity map if the decimal value is kept to full precision. And this of course guarantees the separation of binary numbers by decimals, namely that for each binary x there is some decimal X such that $X \to x$.

The monotonicity property is more subtle. At first sight, monotonicity appears to be built into the algorithms, both of which compute

$$x \times 10^{SCALE}$$

in order to convert an input value x. What happens though is that nearby values x may be scaled quite differently. In algorithm D, trailing zeros may be appended to or stripped from the input significant digit string in order to minimize the magnitude of *SCALE*. Here is an example of single format conversions, using adjacent 9-digit numbers:

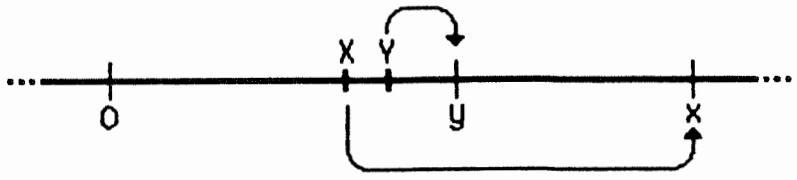
$$1.23499999\text{e}{-10} \to 123499999. \,/\, 10^{18}$$

$$1.23500000\text{e}{-10} \to 1235. \,/\, 10^{13}$$

The latter value is converted with just one rounding error since 10^{13} is exact; but the former suffers an extra error in 10^{18}. If these decimal values happened to be nearly half-way between two single format numbers and round-

ing were *to nearest*, the extra error incurred in the former case might cause it to round up while the latter value (correctly) rounds down – violating the monotonicity rule.

To see that algorithm D is monotonic for directed roundings it suffices to consider the following case. Let X, Y be decimal numbers such that $0 < X < Y$ and suppose $X \to x$, $Y \to y$ in decimal to binary conversion with rounding *toward* $+\infty$. The direction property assures that $X \le x$ and $Y \le y$. Can $y < x$? In a picture:

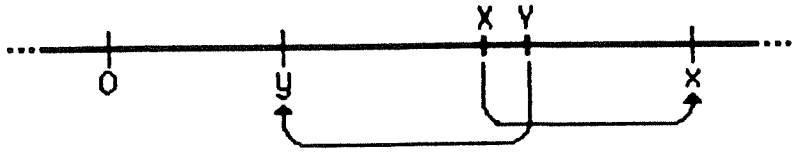


Bad news for monotonicity – directed roundings.

This situation can arise only if the error in the conversion $X \to x$ exceeds one ulp_2 by at least $Y-X$, which of course is at least one ulp_{10}. From formula **C** we see that, with 9 decimal digits, $Y-X$ is at least $0.0084\,ulp_2$ for single numbers; and that with 17 decimal digits, $Y-X$ is at least $0.05\,ulp_2$ for double numbers. However, the table in the discussion of algorithm D limits the extra error to $0.0078\,ulp_2$ for single and $0.0049\,ulp_2$ for double, barely precluding the possibility that $y < x$.

Why is the bound so tight for single conversions? Recall that the 9-digit decimal numbers are up to 120 times as dense as the P754 single numbers in some critical intervals $[2^e, 10^E]$. This means that, in the picture above, X and Y may be *very* close to each other and to x, relative to the gap from x to y. The extra error required to lose monotonicity is just a tiny fraction of the input spacing ulp_2.

Carrying this analysis over to the case of rounding *to nearest* is easy; it is only the picture that changes. As before let X and Y be decimal numbers such that $0 < X < Y$, and let $X \to x$ and $Y \to y$. Again, can $y < x$? No direction property applies here, but the bounds given after algorithm D assure that the conversion error must be less than one ulp_2. First, if $x \geq X$ then $y < x$ implies an error in excess of one ulp_2. Similarly for $y \leq Y$. So monotonicity is jeopardized only if we have the situation:



Bad news for monotonicity – rounding *to nearest*.

In the worst case, X and Y are situated about the midpoint between x and y, which must be adjacent binary numbers if the error is fall below an ulp_2. The only difference is that here we ensure that $Y-X$ is less than *half* of the extra error allowed; this way the two errors can never conspire to cross the midpoint between x and y. But all is well since the value δ limiting the extra error inherited from the scale factor is at least halved when rounding *to nearest*.

Monotonicity makes sense in algorithm B only for a predetermined output precision. For example a binary value just less than 1.5 will print as "1." to one significant digit while any number of binary values just less will print as "1.5" to two significant decimals. With this in mind, monotonicity is indeed built into binary to decimal conversions. The only way for nearby binary values to be scaled by different powers of ten is for them to straddle a power of ten or to both be just greater than a power of ten. Since *LOGX*, the estimated floor of the $\log_{10}$ of the input value, is itself monotonic,

monotonicity is easily verified in the few cases that neighboring binary input values are scaled by different powers of ten.

We have now succeeded in verifying that algorithms B and D satisfy the accuracy properties requested in lieu of correct rounding.

Out of this flurry of bounds and inequalities come a few interesting relationships. The monotonicity and recovery properties seem to oppose each other. When the decimal numbers are dense relative to the binary numbers, as is the case with P754 single, the recovery property is trivially satisfied but monotonicity is barely guaranteed. And when the decimal numbers are relatively sparse, as with P754 double, just the opposite is the case. In some sense, the monotonicity and recovery properties have the last word on the accuracy of algorithms B and D since the other properties are built right in. Are B and D overkill? Look back at the discussion of monotonicity in single format conversions. The required bound was barely met there, saying that not only are 32 significant bits required for intermediate calculations, but that the factor $10^{|SCALE|}$ must be computed with just one rounding error. Algorithm P showed this was possible. The situation for double format conversions is quite different. Algorithm Q is allowed its expected complement of errors in producing $10^{|SCALE|}$, and it can even be shown that only 63 significant bits are required for sufficiently accurate conversions.

3.5. Nit-picking

What follows is a collection of lesser details, included as much for their curiosity as for an air of completeness they may lend. They were omitted from the body of the text so as not to distract the patient reader.

We have seen that the 9-digit decimal numbers are up to 120 times as dense as the P754 single format numbers. A concrete example shows how

the formats' relative spacings can be surprising. Consider the value

$$\frac{1}{3} = 0.555555555\cdots_{16}$$

which rounds to 0.5555558_{16} in P754 single with its 24 significant bits. The absolute rounding error is exactly $\frac{1}{3}\text{ulp}_2$. Now in the neighborhood of 1/3, one ulp_{10} is about $\frac{1}{30}\text{ulp}_2$, so the error in rounding 1/3 to 24 bits corresponds to over 10 ulps_{10}. The nearest 9-digit decimal to the *rounded* value of 1/3 turns out to be 0.333333343_{10}. And the nearest 9-digit decimal to the next smaller single format number happens to be 0.333333313_{10}. Thus there is no way to produce 0.333333333_{10} from a P754 single format value! The apparent discrepancy in the second to the last digit is likely to be mistaken for a bug in the conversion routine, rather than a reflection of the relative density of decimal and binary numbers.

The number of decimal digits required to *distinguish* binary numbers of a given precision was discussed in the context of correctly rounded results. Is the separation requirement, from which the relation was derived, compromised by the extra error δ suffered in computing the scale factor? The answer NO is guaranteed by the recovery property as verified in the last section. This is the sense in which recovery is the computational analog of separation.

Goldberg's paper [5] about the separation property is of historical significance to P754 enthusiasts. Not only is it one of the first technical arguments for an implicit leading bit in a binary floating-point format, but it is the first known discussion of how to encode denormalized numbers [4] and zero by reserving the bottommost exponent.

It is a simple exercise to reverse the arguments about the Separation Requirement and deduce, as Goldberg did, that $2^{-p+1} < 10^{-P}$ is a sufficient condition to guarantee that p-bit binary numbers will distinguish P-digit decimal numbers. The P754 single format numbers, with 24 significant bits, distinguish 6-digit decimal numbers, and the P754 double numbers distinguish 16-digit numbers.

In the discussion of the separation requirement, we deduced the chain of inferences

$$10^{-P+1} < 2^{-p} \quad \rightarrow \quad \text{Separation Requirement} \quad \rightarrow \quad \text{Distinction} \quad .$$

but noted that the three are not generally equivalent. In some cases the inequality is stronger than absolutely necessary. The Separation Requirement is equivalent to the inequality

$$\frac{\text{ulp}_{10}}{\text{ulp}_2} = \frac{10^{-P}}{2^{-p}} \times \frac{10^{E+1}}{2^{e+1}} < 1 \quad .$$

Recall that the latter ratio varies between 1/2 and 10. The inequality derived before simply assured that $10^{-P}/2^{-p}$ was less than 1/10. However, it is a fact of number theory (the existence of $(P-1)/p$ approximating $\log_{10}(2)$ arbitrarily closely from above) that there exist pairs P and p such that $10^{-P}/2^{-p}$ is just slightly above 1/10. Then, if we simply restrict the range of E and e so that $10^{E+1}/2^{e+1}$ stays far enough below 10, then the Separation Requirement will be met by a pair P and p just barely failing the inequality $10^{-P+1}<2^{-p}$. Knuth presents this as an exercise relating the Separation Requirement and the Distinction Property [6, p. 312 exercise 18, with solution].

For conversions between two floating-point number systems, the Separation Requirement and distinction property are equivalent, although this fact is not of great importance for the purposes of this paper.

Care was taken in algorithm L to ensure a lower bound on $\lfloor \log_{10}(x) \rfloor$. Why? The issue is looping in algorithm B. If L were allowed to be too big then corrective step B6 would have to branch back to step B3 whether the scaled value was too big or too small. It is possible that an input value very near to a power of ten could round in such a way as to fail both tests and loop indefinitely. Getting the lower bound on *LOGX* is much easier than defending the loop criteria against further pathologies.

This paper discusses conversions from the P754 single and double formats backed up by an extended format. It should be obvious that single format conversions backed up by the double-extended format easily satisfy the accuracy requirements. But what about extended conversions? Algorithms B and D may be used to convert to and from an extended format, but there may be a significant loss of accuracy due to lack of extra precision beyond double-extended; and without extra exponent range, numbers at the extremes of the double-extended range will be converted incorrectly because of intermediate overflows and underflows. In order to cover the full range of extended numbers, the table in algorithm Q must be extended. The following are reasonable table values:

$$0.\text{C6B0A096A95202BD}_{16} \times 2^{1369} \approx 10^{412} \times (1+2^{-66}) ,$$

$$0.\text{9A35B24641D05953}_{16} \times 2^{2738} \approx 10^{824} \times (1+2^{-65}) ,$$

$$0.\text{B9C94B7FA8D76515}_{16} \times 2^{5475} \approx 10^{1648} \times (1+2^{-65}) \text{ , and}$$

$$0.\text{86D48D6626C27EEC}_{16} \times 2^{10950} \approx 10^{3296} \times (1+2^{-65})$$

Alternative values may be computed with the algorithm providing correctly-rounded conversions, supplied in the Appendix. To find the appropriate bound on the rounding error simply compute each desired 10^k to a modest number of extra significant bits.

In the discussion of pathologies in algorithm B, we dismissed the need for the second test in step B6 for single format conversions. However, if extended values are to be converted using algorithm B the second test in step B6 is essential. There are potentially many more representable values $10^E \times (1+\gamma)$ which will scale to $10^{N-1} \times (1-\varepsilon)$ when N digits are required.

The rounded table values 10^{27} and 10^{40} in algorithm P just barely cover the range of P754 single numbers. If in step B2 of algorithm B, *LOGX* of a tiny number were computed as −46 instead of the correct −45, then *SCALE* in step B3 could be 9−(−46)−1 = 54, beyond the range of algorithm P. Fortunately this does not happen; all of the denormalized numbers whose correct *LOGX* is −45 are sufficiently far above 10^{-45} that algorithm L computes their *LOGX* correctly as −45.

Although the rounded value 10^{40} is available to algorithm P directly from the table, the value is deliberately computed from $P_{27} \times P_{13}$ in order to cause a rounding error. The rounding error suffered in the multiply causes the value 10^{40} to correctly honor the rounding mode in effect.

Step B7 of algorithm B calls for the conversion of an integer value in the extended format to a decimal string. Here is an efficient way to accomplish this for single format conversions. First, express the extended value as a true 32-bit binary integer, in this case

$$00b_{29}b_{28}b_{27} \cdots b_1 b_0 \bullet \ ,$$

since the value is bounded by 10^9. Then divide this by 10^9 producing the chopped binary fraction

$$0 \bullet b_{-1}b_{-2}b_{-3} \cdots b_{-31}b_{-32} \ .$$

Adjust this value upward by one unit in the 32nd bit, producing a value slightly greater than the true quotient. In a 9-step loop, repeatedly multiply

the binary fraction by ten (two shifts and an add) stripping off successive BCD digits as they appear left of the binary point. At the end of the loop, discard the remaining fraction. The loop operations are exact; the *only* error arises from chopping the quotient and adjusting upward, that is,

$$0 < 2^{-32} \times (1 - 0{\bullet}b_{-33}b_{-34}b_{-35} \cdots) < 2^{-32} \quad .$$

Its impact on the final digit string is bounded by $2^{-32}\times 10^{9} < 1$, so the computed digits are correct.

For the purposes of exception handling, binary-decimal conversions are treated as atomic operations in P754. Algorithms B and D are presented as *programs* based upon a few P754 arithmetic operations. Algorithms B and D always signal the inexact exception when their results are inexact; they pessimistically signal inexact in the rare circumstances when multiple rounding errors cancel and the result is in fact exact. Algorithm D may also suffer overflow and underflow. It is set up to encounter any range exception in the format conversion in step D6. If values at the limits of the range of extended are converted there is no way to represent scale factors guaranteed to generate the appropriate error in step D6.

Algorithm B can suffer a format overflow error if the destination string cannot accommodate the converted value. For example, suppose binary x is converted to the 8-digit decimal value $-1.2345678\times 10^{-250}$, but is destined for a string of at most 14 characters. The string

"−1.2345678E-250"

is one character too long. More severe cases are possible. The problem is complicated by the possibility that a massive amount of printed output may be ruined if just one field, and hence one line, is allowed to overflow by a character. There are several remedies. The value may be converted again,

but to fewer significant digits. Or if the value must overflow the field, the printer driver program may allow the offending line to spill over, and then skip to the next page; in this way the output is intact but for the few pairs of partial pages where a line overflowed. Historically, a reproof such as "???.??" has been printed when all else failed.

It was shown under Accuracy Revisited that conversion from 17-digit decimal values to the P754 double format using algorithm D would be monotonic. The same argument guarantees monotonicity for conversion from 18-digit values, but it fails for 19 digits. Some systems may allow 19-digit values to be input, since the 64-significant-bit double-extended format will accommodate any 19-digit value exactly, but these conversions will not in general be monotonic.

P754 requires that the conversion of input values in a certain range be perfectly rounded; that is, the power of ten used for scaling must be computed ***exactly***. Is this requirement actually met? Step D3 of algorithm D preconditions the input to decimal to binary conversion specifically to meet P754, so the scale factor is always the correct one. However, the situation for binary to decimal conversion is less obvious, since the scale factor depends on *LOGX*, which may be too low by one. For instance, if nine decimal digits were desired, a single format input value just larger than 10^{-5} would ideally be scaled by the exact value 10^{13} and rounded to an integer to determine the significant digit string:

$$1.00000xxx\cdots \times 10^{-5} \approx 100000yyy. \times 10^{-13}$$

However, if *LOGX* were miscalculated as −6 then the scale factor would be 10^{14}, known to be wrong by a full half ulp in the single-extended format. If the error in the scale factor 10^{14} caused the significant digit field to be com-

puted as

$$999999zzz_{\bullet} \times 10^{-14}$$

then algorithm B would produce an imperfectly rounded result – in violation of the standard. So the question is, when can a miscalculated *LOGX* lead to an incorrectly rounded output value, rather than a branch back in step B6 of algorithm B? As we saw in the discussion of pathologies in algorithm B, the answer is NEVER for single format conversions and RARELY for double conversions; indeed, the situation can arise in double only for values far outside the range in which *LOGX* can make the difference between perfect and imperfect conversion. So there is no hazard after all.

What is the point of all this? On the one hand we have the simple but usually uneconomical correctly rounded conversions. On the other we have reasonably accurate, yet economical conversions whose economy is bought with a tedious verification that they are "accurate enough". These conversions are so nearly correctly rounded that, although different implementations may produce results differing in just one ulp, those differences – and the deviation from correct rounding – will be almost imperceptible to users.

Acknowledgments

The author gratefully acknowledges the various sources and means of support throughout the protracted development of this chapter. W. Kahan has made substantial contributions to its content and form. Several colleagues at Apple Computer, Hewlett-Packard, and Motorola patiently worked through earlier drafts. Much of the work on the algorithms was undertaken while the author worked part-time at Zilog, Inc., and later at Apple Computer. Occasional financial support was received from an IBM Fellowship and the U. S. Department of Energy.

References

[1] IEEE Subcommittee 754, "A Proposed Standard for Binary Floating-Point Arithmetic", Draft 10.0, IEEE Subcommittee 754 working document 82-8.6.

[2] Apple Computer, *Apple III Pascal Programmer's Manual*, Product #A3L0003, 1981, chapter 14.

[3] J. T. Coonen, "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic," *Computer*, **13**, 1, January 1980, pp. 68-79.

[4] J. T. Coonen, "Underflow and the Denormalized Numbers," *Computer*, **14**, 3, March 1981, pp. 75-87.

[5] I. B. Goldberg, "27 Bits are Not Enough for 8-Digit Accuracy," *CACM*, **10**, 2, February 1967, pp. 105-106.

[6] D. E. Knuth, *The Art of Computer Programming*, Vol. 2, Addison-Wesley, 1981.

[7] D. W. Matula, "In-and-Out Conversions," *CACM*, **11**, 1, January 1968, pp. 47-50.

[8] E. W. Phillips, "Binary Calculation," excerpted in *The Origins of Digital Computers*, 2nd Edition, B. Randell ed., Springer-Verlag, 1975, p. 293.

[9] R. K. Richards, *Digital Design*, Addison-Wesley, 1981.

[10] Zilog, Inc., *Z8000 CPU Technical Manual*, Document 00-2010-C, 1981.

[11] D. Zuras, Hewlett-Packard Corporation, private communication.

Suggested Number Grammar

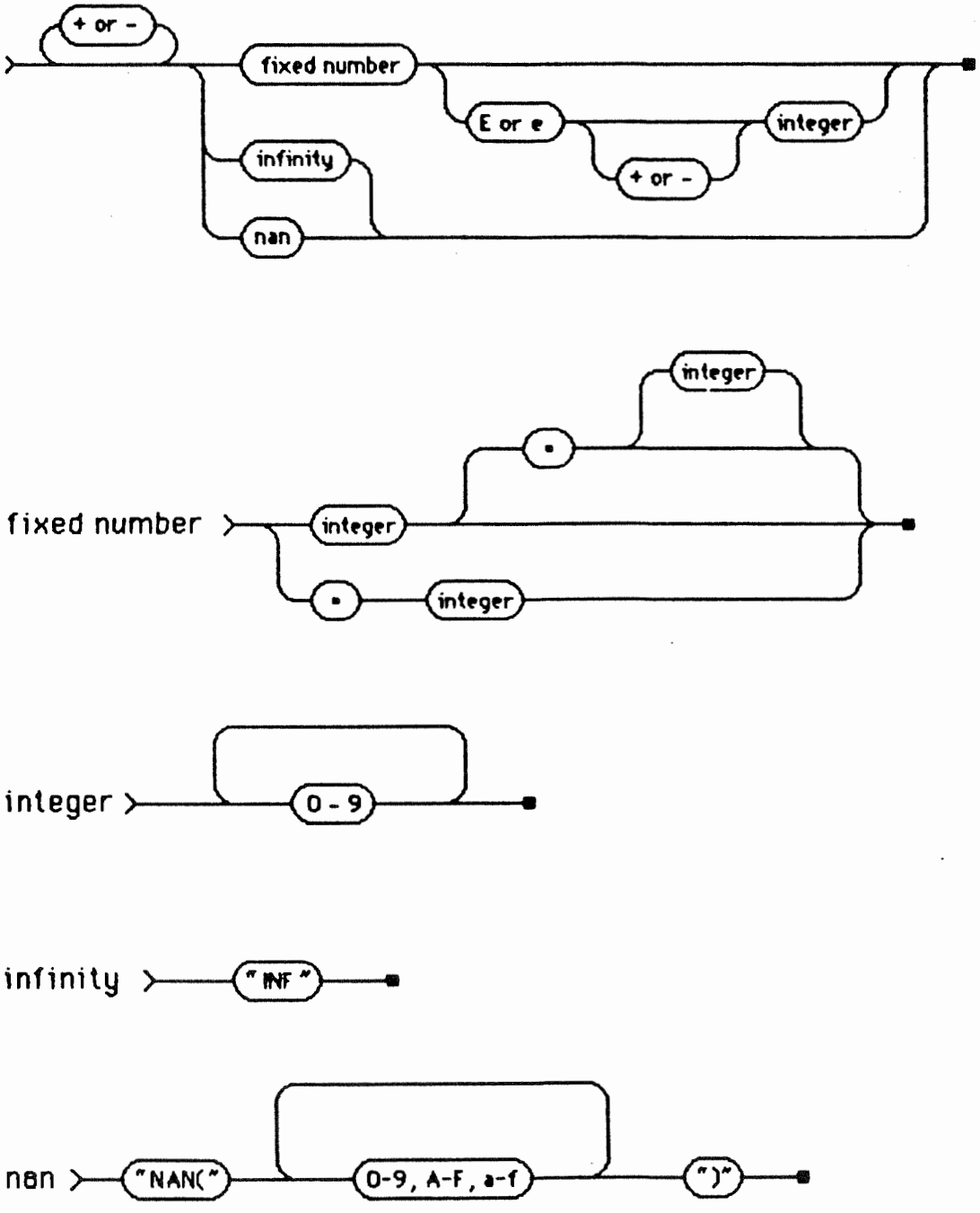
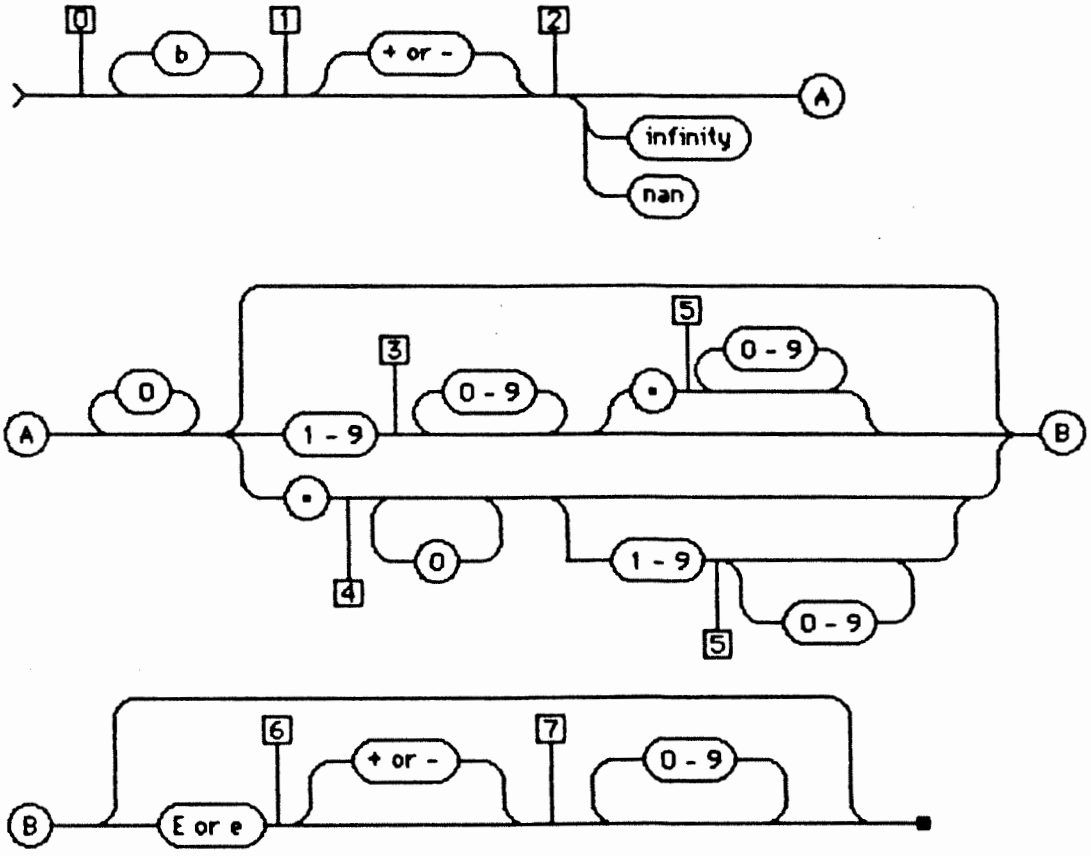


Figure 1.

Eight State Decoder



0: Skip leading white space: blanks, tabs, etc.

1: First non-white character – is it a sign?

2: Sign found – number/NAN/∞ must follow.

3: First significant digit found before decimal point.

4: Decimal point found before first significant digit.

5: Seeking more sigificant fraction digits.

6: Found E or e – start of exponent.

7: Found exponent sign – more exponent digits?

Figure 2.

CHAPTER 8

Radix-Independent Description of the Proposed Standard

1. Introduction

The intent of proposed IEEE standard P754 for binary floating-point arithmetic is to regulate the numerical programming environment. The story really begins with implementations of high-level languages, whose semantics must be carefully defined with regard to the overall structure of programs and the control of side-effects. But this chapter picks up in the middle, at the level of a single arithmetic operation like

$$z := x \times y;$$

Simple as it may appear, this operation involves many subtleties if x, y, and z are allowed to have different number formats, or if an exception like overflow should arise in the computation of the product $x \times y$. Since steps are taken in P754 to handle every exception, such as $x \times y$ overflowing to ∞, further error possibilities are introduced, such as $0 \times \infty$ in a subsequent product.

The system described here conforms to draft 10.0 of IEEE proposal P754 and is intended to be compatible with the forthcoming radix-free proposal P854. The first version of this implementation guide, based on draft 8.0 of P754, was presented at a tutorial on the proposed standard in May 1981.

The paradigm for the operation above is:

Compute the product $x \times y$ as though with unbounded range and precision, and pack the result in z.

But this very natural statement has many ramifications. For example, how accurate is the actual implementation of "unbounded precision" when the ideal result must be packed into a destination with limited range and precision? And what of the error conditions overflow and occurrence of invalid operands? In fact, the computation of the result z is not so much an atomic operation as it is a process that may be viewed as:

(1) Unpack x.

(2) Unpack y.

(3) Compute the ideal result (as though with unbounded...).

(4) Trim the ideal result to within z's format limitations.

(5) Pack the result into z.

This process is expressed precisely in the Control Flow section. The various steps of the arithmetic operation are written as subroutines.

The heart of the operation, (3) above, is discussed in terms of operands in a so-called canonical format. Thus they are radix and format free, while following the rules laid out in proposal P754.

The unpack/pack operations are of course format specific. Three sections of this document describe these operations for the binary formats specified in P754.

Though this is ostensibly an implementation guide, it is not intended to translate directly to an implementation. Efficiency and compactness have been sacrificed throughout to obtain the greatest modularity. For example, each individual arithmetic operation handles

input NANs in the opening **switch** statement; a more effective implementation filters NANs just once, in a preamble to the operations. Also, each step of the trim_result() sequence, checking for underflow, rounding, and checking for overflow, is coded independently, necessitating redundant tests for special short circuit cases. This modularity permits the reader to study individual sections of the code without having to know the state of the system as a whole. The ultimate object is twofold: to convey an idea of the data and control flow through an arithmetic operation, and to prescribe the result of any operation.

This description is written in a type of pseudo-code based on the programming language C. Our pseudo-C has a rich set of data types and a high tolerance for abuses of types. For example, a significant digit field will in some contexts be viewed as an array of digits while in others it will be given its mathematical interpretation as a value whose radix point lies after the leading significant digit. Most of the syntactic short-cuts for which C is notorious (for example, "x++;" means the same as "x = x + 1;") have been carefully avoided. Readers unfamiliar with C should be able to follow the control flow without getting lost in the language constructs, since the language is quite terse and only the simplest control structures are used here.

Aficianodos will note several deviations from conventional C. Usually the meaning will be clear from the context rather than from strict C semantics.

(1) Subsets of arrays are used. For example, if fraction[] is defined as an array of digits (decimal, binary, or otherwise), the expres-

sion "fraction[1 ... 23]" denotes the first 23 digits taken as a group. The expression "fraction" by itself denotes the entire array.

(2) When a set of elements of a structure are to be taken as a unit, notation like "operand.(msd, fraction)" is used.

(3) The passing of parameters is quite cavalier. For example, the expression "normalize(op)" is used instead of "normalize(&op)" when it intended that the caller's operand be modified. Strictly speaking, op's address, "&op", should be passed.

(4) In each use of the C switch/case construct, the cases are mutually exclusive, so the **break** instruction is omitted.

(5) C indexes arrays from 0, that is the N elements of an array x are x[0], x[1], ..., x[N-1]. That notation is clumsy for the present discussion, so the convention x[1], ..., x[N] is used instead. The text is very explicit about this when it matters.

(6) Most of the variables used in the pseudo-code are global, that is they are known to all procedures. For definiteness, the globals used in any routine are declared **extern** as in C.

This chapter makes many detailed references to the P754 document, in an attempt to illuminate what may be stated very tersely there. Each reference is marked by a section number (such as §4, which introduces the notion of rounding).

Once again, this is not a complete "implementation" of P754. Aside from lacking any detailed mention of the programming environment, this discussion omits several operations. Binary-decimal conversions are treated extensively in chapter 7. And floating-integer

conversions are left out because of their highly system-dependent nature; they differ from the floating-point round to integer instruction only in the exceptions that arise from attempting to store huge or nonnumeric values in an integer format with no reserved operands (§5.4 and §7.1.7 of P754 discuss these issues).

The proposed standard entails a small number of implementation options. The reader's attention will be called to those situations where a variety of responses are possible.

2. Control Flow

The following procedure effects the operation $z = x \# y$. The dyadic operations add, subtract, multiply, divide, and remainder produce a floating-point result. Comparison produces a condition code in this presentation. P754 also permits comparisons to be effected by high-level language predicates (§5.7); see chapter 6 for a discussion of this style of comparison. The monadic operations round to integer, square root, and the various format conversions have an obvious analog of the form $z = \# x$.

For simplicity, the storage operands x, y, and z are declared generically, that is without reference to their storage foramts. In fact, the types may differ. The only constraint of proposal P754 is that the z's format be no narrower than the wider of the x and y formats, except for the format conversion operations (§5.1 - 5.3).

Statements of the form $z = x$, $z = -x$, or $z = |x|$ in which z and x have the same format are non-arithmetic since no conversion is required. They may be effected by simple translation of the digits, perhaps with a sign change as in absolute value or negation, or they

may be implemented arithmetically (§5, Appendix). Actual format conversions follow the form of the other monadic operations, except that the compute step is trivial – all the work is in the trimming and packing.

---

arithmetic_operation(z, x, y) *arithmetic_operation*

```
/*
 * The types of z, x, and y may differ, as explained
 * in the text above.
 */

storage_types     z, x, y;

{
    extern canonical          op1, op2; /* unpacked inputs */

    set_globals();            /* collect mode information */

    unpack(op1, x);           /* op1 <-- x, unpacked */
    unpack(op2, y);           /* op2 <-- y, unpacked */

    /*
     * The following routine is generic -- add(), subtract(),
     * multiply(), divide(), ... should be called as appropriate.
     */

    compute_result();         /* set result <-- op1 # op2 */

    /*
     * The subtlest phase of any operation is trimming the
     * result according to the limitations of the destination z.
     * This is distinguished from the actual packing, to
     * preserve as much of the format-free nature of the
     * trimming.  Note that the comparison operation will
     * bypass the next two steps, trim and pack.
     */

    trim_result();

    pack_result(z);           /* just a bit-mapping operation */

    side_effects();           /* collect and handle error flags */
}
```

---

3. Globals

Underlying the basic arithmetic process is a group of global variables, defined as custom C-like structures. They are listed here with a brief summary of their purpose. Also included are the initialization routine set_globals() and the clean-up routine side_effects() that do basic housekeeping operations on the globals.

Canonical format operands op1 and op2 hold the unpacked input operand(s), and result holds the computed "infinite precision" value.

---

canonical op1, op2, result;

The mode structure determines the rounding precision and direction. Since this variable has a life-span of just one operation, it must be fetched from the user's environment at the start of each operation. The definition of "mode" in §2 describes the behavior of the mode as part of the user's environment.

---

mode_str mode;

The error structure logs exceptions for *each* operation separately. At the end of the operation, the error structure is used to update the user's "status flags" (see §2). The trap structure determines whether the user wants a software trap when the corresponding exceptions

arise. Like the mode variable, the trap structure is loaded from the user's environment at the start of each operation.

---

exc_str error, trap;

The dst structure contains data about the destination format, for use by the trim() routines especially.

---

dst_str dst;

The initialization routine collects state information -- from control registers of the arithmetic device, from the user's "process data area", or possibly from the instruction itself; thus the fetch() operation is is highly implementation-dependent.

---

set_globals

```
set_globals()
{
        extern mode_str         mode;
        extern exc_str          error, trap;
        extern dst_str          dst;

        /*
         * Determine rounding precision and direction.
         * If the operation is remainder, ignore any precision
         * control specification -- use the range and precision
         * of the destination format (section 5.1).
         */
        fetch(mode);
```

```
        /* Clear all flags (for this operation) to FALSE. */
        clear(error);

        /* Determine which exceptions the user will trap on. */
        fetch(trap);

        /*
         * Set the range and precision of the destination,
         * subject to the precision control mode.
         */
        set(dst);
}
```

The termination routine stores the error flags back where the user can interrogate them. (Note that the flags are never cleared by arithmetic, but only at the user's specific request.) Also, if a software trap is to occur the mechanism is initiated here.

---

side_effects

```
side_effects()
{
        extern exc_str          error, trap;

        /* Logically OR error into the user's flags. */
        save(error);

        /*
         * Check whether any of the errors that arose are
         * to stimulate a user trap.
         */
        if (error & trap)
        {
                /* System-dependent trap interface. */
        }
}
```

---

4. User state

User-determined state variables are kept in a defined structure called mode. The particular encodings used here are representative, not mandatory.

```
typedef struct mode_str
{
        bit     round[2];
        bit     precision[2];
} mode_str;
```

These are the encoded values of the rounding directions, kept in the round[] element of mode_str. All four rounding modes must be implemented (§4).

```
# define        TO_NEAREST      0       /* default */
# define        TOWARD_0        1
# define        TOWARD_PLUS     2
# define        TOWARD_MINUS    3
```

When available, rounding precision control permits a user to round results to a narrower precision than that of the destination format. This is intended to help users of different systems to overcome architectural differences in producing matching results. For example, suppose that a program is to be run on identical single format data sets on two different systems. The first system does all calculations in single, while the other delivers all intermediate results to the double-

extended format. If the program sets the precision control to single on the second system, then, in the absence of overflow or underflow in the *first* system's calculations, both will obtain identical results.

§4.3 and footnote 4 specify which systems must have precision control. However, it is up to the implementor to decide whether precision control implies *range control* too, that is, whether the exponent is coerced to within the bounds corresponding to the precision. If both precision and range are controlled, then identical results can be obtained regardless of the presence of extended intermediates (because they are coerced as though they are single). This option is a tradeoff in P754. Although it is desirable to achieve identical results (despite overflow and underflow) when the same calculation is performed on different systems, the cost of range coercion may be very high.

Note that precision control is intimately tied to the complicated issues of expression evaluation in high-level languages. But that is beyond the scope of this guide.

```
# define        EXTENDED    0       /* default */
# define        SINGLE      1
# define        DOUBLE      2
```

Corresponding to each of the 5 elements of exc_str is a sticky error flag and a trap-enable flag. Since support of user traps is optional, the trap structure is optional (§8).

```
typedef struct exc_str
{
        boolean         inexact;
        boolean         invalid;
        boolean         div_zero;
        boolean         oflow;
        boolean         uflow;
} exc_str;
```

5. Canonical format

This canonical format is described in radix-free form following the spirit of P754. Only this format is referred to below in the discussion of the operations. This description of operands as data structures of bits, digits, integers, etc. permits a precise specification of the arithmetic in terms of primitive operations such as shift and increment. The canonical numeric data type is defined as:

```
typedef struct
{
        int     tag;
        bit     sign;
        int     exponent;
        digit   c_out;
        digit   msd;
        digit   fraction[ CANON_FRACTION ];
} canonical;
```

The tag is a small integer used to identify special operands not having the usual form

$$(-1)^S \times RADIX^E \times X.XXXXXXX.$$

This greatly simplifies the discussion by distinguishing the special values from numerical representations.

```
# define    ZERO_TAG      0     /* 0 */
# define    INF_TAG       1     /* infinity */
# define    S_NAN_TAG     2     /* signaling NAN */
# define    Q_NAN_TAG     3     /* quiet NAN */
# define    NUM_TAG       4     /* finite nonzero number */
```

The sign is just one bit of information, 0 for + and 1 for −.

The canonical exponent, is presumed to accommodate all result exponents from operations on supported formats. Thus neither overflow nor underflow will arise in canonical numbers until they are trimmed to within the constraints of the destination format. Though the exponent is described as type integer above, care must be taken to provide sufficient range. For example, 17 (two extra bits) of working range are required of a P754 implementation supporting the double-extended format, or else some extra tests are required in the overflow and underflow handlers. Chapter 9 deals with this in detail. No assumption is made about the radix of the exponent as an integer. For example, it may be desirable to implement decimal floating-point arithmetic with a binary exponent.

The discussion of the operations is independent of the radix of the underlying implementation. Although this discussion applies to arithmetic with any positive, integer radix, the interesting cases are expected to be 2 and 10. The parameters are set for binary arith-

metic here.

```
# define        RADIX           2       /* or 10 or 8 or 16 ... */
# define        HALF_RADIX      1       /* for use in rounding */
# define        RADIX_1         1       /* radix minus 1 */

typedef         digit           bit;
```

The canonical format has an extra (second) digit, c_out, to the left of the radix point to catch carries out of the msd (most significant digit). C_out is named explicitly only to simplify the description. Typically, an implementor will provide for a carry-out only in those few places where one can arise.

The canonical format carries three extra low-order fraction digits so that results can be rounded as in §5 of P754. These digits are commonly known as guard, round, and sticky:

Guard is next digit beyond the least significant digit of the widest storage format supported.

Round is the next digit beyond guard. It is crucial to the operations addition, subtraction, and division which may entail a left shift before rounding.

Sticky conveys just one bit of information (though it will normally be an entire digit). It is nonzero precisely when the associated infinite precision number has nonzero digits to the right of the round digit.

The working precision as specified here is suitable for a P754 implementation supporting the double-extended format.

```
# define        CANON_FRACTION        66
```

The "infinite precision" result is trimmed to the destination format according to a set of parameters kept in the special purpose structure:

```
typedef struct
{
        int             othresh;        /* overflow threshold */
        int             uthresh;        /* underflow threshold */
        int             biasadjust;     /* exponent fix for traps */

        /* index of least significant digit in fraction[] */
        int             lsd;
} dst_str;
```

---

6. P754 Formats

The following two structures define data types corresponding to the single and double formats specified in §3.2 and 3.3 of P754. Each format may be thought of as a trio of bit strings, denoted as arrays of bits below. As bit strings:

The sign bit is 0 for +, 1 for -.

The exponent is an unsigned integer, biased by 127 for single, and 1023 for double.

The fraction lies just to the right of the binary point of the unpacked number.

The ordering of the bits, from most to least significant, is suggested by figures 1 and 2 in §3, but P754 does not specify how they are to be ordered in byte or word groupings.

```
typedef struct
{
        bit     sign;
        bit     exponent[8];
        bit     fraction[23];
} single_binary;

typedef struct
{
        bit     sign;
        bit     exponent[11];
        bit     fraction[52];
} double_binary;
```

The extended formats are optional in implementations of P754. A typical system will support (only) the extended format corresponding to the wider basic (single or double) format supported.

Unlike the basic formats, the extended types have range and precision subject only to minimum bounds, rather than specifications down to the bit. The most significant bit may be implicit or explicit at the implementor's option. (This may be inferred from §3.3 and the width parameters in table 1.)

```
# define            S_EXT_RANGE             11
# define            S_EXT_FRACTION          31

typedef struct
{
        bit         sign;
        bit         exponent[S_EXT_RANGE];
        bit         msb;
        bit         fraction[ S_EXT_FRACTION ];
} single_extended_binary;


# define            D_EXT_RANGE             15
# define            D_EXT_FRACTION          63

typedef struct
{
        bit         sign;
        bit         exponent[D_EXT_RANGE];
        bit         msb;
        bit         fraction[ D_EXT_FRACTION ];
} double_extended_binary;
```

7. Unpack Binary Formats

7.1. P754 Single

Unpack a P754 single format number s to the canonical format.

single_unpack(w, s) *single_unpack*

```
        /* w <-- s, unpacked. */
        canonical               w;
        single_binary           s;

{
        extern mode_str         mode;
```

```
        /* Assume s is a normal number; then check special cases. */
        w.tag = NUM_TAG;
        w.sign = s.sign;
        w.exponent = s.exponent - 127;      /* 127 is the exponent bias */
        w.c_out = 0;
        w.msd = 1;                          /* presumed normalized... */

        /* Fraction of s is left-justified in w, and zero padded. */
        w.fraction[1 ... 23] = s.fraction;
        w.fraction[24 ... CANON_FRACTION] = 0;

        if (s.exponent == 0)
        {
                /* Zero or denormalized. */
                if (s.fraction == 0)
                        w.tag = ZERO_TAG;           /* Zero. */
                else
                {
                        w.msd = 0;          /* Denormalized. */
                        w.exponent = w.exponent + 1;
                        normalize(w);
                }
        }

        else if (s.exponent == 255)
        {
                /* Infinity or NAN. */
                if (s.fraction == 0)
                        w.tag = INF_TAG;            /* infinity */
                else
                        /*
                         * Distinction between signaling and
                         * quiet NANs is system-dependent.
                         * Leading FRACTION bit is used here.
                         */
                        if (s.fraction[1] == 1)
                                w.tag = Q_NAN_TAG;
                        else
                                w.tag = S_NAN_TAG;
        }
}
```

7.2. P754 Double

Unpack a P754 double format number to the canonical format. This is precisely analogous to the single unpack routine above.

double_unpack(w, d) *double_unpack*

```
        /* w <-- d, unpacked. */
        canonical               w;
        double_binary           d;

{
        extern mode_str         mode;

        /* Assume d is a normal number; then check special cases. */
        w.tag = NUM_TAG;
        w.sign = d.sign;
        w.exponent = d.exponent - 1023;     /* 1023 is the exponent bias */
        w.c_out  = 0;
        w.msd = 1;                          /* presumed normalized... */

        /* Fraction of d is left-justified w and zero padded. */
        w.fraction[1 ... 52] = d.fraction;
        w.fraction[53 ... CANON_FRACTION] = 0;

        if (d.exponent == 0)
        {
                /* Zero or denormalized. */
                if (d.fraction == 0)
                        w.tag = ZERO_TAG;           /* Zero */
                else
                {
                        w.msd = 0;          /* denormalized */
                        w.exponent = w.exponent + 1;
                        normalize(w);
                }
        }

        else if (d.exponent == 2047)
        {
                /* Infinity or NAN. */
                if (d.fraction == 0)
                        w.tag = INF_TAG;            /* Infinity. */
                else
                        if (d.fraction[1] == 1)
                                w.tag = Q_NAN_TAG;
                        else
                                w.tag = S_NAN_TAG;
        }
}
```

7.3. P754 Single-Extended

There are many plausible implementations of the extended formats that meet the range and precision specifications of P754. For example, rather than having reserved exponent values as in the single and double formats, the extended formats may use a tag field to distinguish operands like zero, infinity, and NAN (the canonical format of this document uses such a field.) Also, there are two possible interpretations of the smallest possible exponent, as explained in chapters 2 and 5.

The extended formats discussed here use a convenient 80-bit format. The exponent is an unsigned, biased integer as in the single and double formats. The exponent value 111...11 is reserved for INF and NAN, in which case the msd is irrelevant. The exponent value 000...00 has only one special case, namely zero, when all significant digits are 0. For simplicity, all finite extended values are normalized when they are unpacked into the canonical format. However, P754 does not require this normalization for unnormalized numbers above bottom of the extended range, so long as the system does not produce such unnormalized results (see §3.3).

---

single_extended_unpack

```
single_extended_unpack(w, se)
        /* w <-- se, unpacked. */
        canonical               w;
        single_extended_binary  se;
{
        extern mode_str         mode;
```

```
        /* Assume se is a normal number. */
        w.tag = se.tag;
        w.sign = se.sign;
        w.exponent = se.exponent - 1023;          /* bias = 1023. */
        w.c_out  = 0;
        w.msd = e.msb;              /* Copy explicit leading digit.

        /* w's fraction is left-justified and zero padded. */
        w.fraction[1 ... 31] = se.fraction;
        w.fraction[32 ... CANON_FRACTION] = 0;

        if (se.(exponent, msd, fraction) == 0)
                w.tag = ZERO_TAG;           /* Zero. */

        else if (se.exponent == 2047)
        {
                /* Infinity or NAN -- msd irrelevant. */
                if (se.fraction == 0)
                        w.tag = INF_TAG;            /* Infinity. */
                else
                        /*
                         * Distinction between signaling and
                         * quiet NANs is system-dependent.
                         * Leading FRACTION bit is used here.
                         */
                        if (se.fraction[1] == 1)
                                w.tag = Q_NAN_TAG;
                        else
                                w.tag = S_NAN_TAG;
        }
        else
                /* All nonzero operands are prenormalized. */
                normalize(w);
}
```

7.4. P754 Double-Extended

This routine is analogous to the single-extended unpack above.

---

double_extended_unpack(w, de) *double_extended_unpack*

```
        /* w <-- se, unpacked. */
        canonical                w;
        double_extended_binary   de;

{
        extern mode_str          mode;

        /* Assume de is a normal number. */
        w.tag = NUM_TAG;
        w.sign = de.sign;
        w.exponent = de.exponent - 16383;          /* bias = 16383 */
        w.c_out  = 0;
        w.msd = de.msb;            /* Copy lead digit. */

        /* w's fraction is left-justified and zero padded. */
        w.fraction[1 ... 63] = de.fraction;
        w.fraction[64 ... CANON_FRACTION] = 0;

        if (de.(exponent, msd, fraction) == 0)
                w.tag = ZERO_TAG;          /* Zero. */

        else if (de.exponent == 32767)
        {
                /* Infinity or NAN. */
                if (de.fraction == 0)
                        w.tag = INF_TAG;          /* Infinity. */
                else
                        /*
                         * Distinction between signaling and
                         * quiet NANs is system-dependent.
                         * Leading FRACTION bit is used here.
                         */
                        if (de.fraction[1] == 1)
                                w.tag = Q_NAN_TAG;
                        else
                                w.tag = S_NAN_TAG;
        }

        else
                /* All nonzero operands are prenormalized. */
                normalize(w);
}
```

---

8. Pack Binary Formats

After the "infinitely precise" intermediate result is trimmed to the precision and range of the destination format (or perhaps somewhat narrower, due to precision control), the result is be packed from the canonical format into the storage format by biasing the exponent and copying the sign and significant bits.

8.1. Pack P754 Single

---

single_pack_result

```
single_pack_result(s)
        single_binary           s;
{
        extern canonical        result;

        s.sign = result.sign;   /* Regardless of special cases. */

        switch (result.tag)
        {
          case NUM_TAG:
                s.exponent = result.exponent + 127;

                /* Denormalized numbers have a bias of 126 */
                if (result.msd == 0)
                        s.exponent = s.exponent - 1;
                s.fraction = result.fraction[1 ... 23];

          case ZERO_TAG:
                s.exponent = 0;
                s.fraction = 0;

          case INF_TAG:
                s.exponent = 255;
                s.fraction = 0;

          case S_NAN_TAG:
          case Q_NAN_TAG:
                s.exponent = 255;
                s.fraction = result.fraction[1 ... 23];
        }
}
```

8.2. Pack P754 Double

double_pack_result

```
double_pack_result(d)
        double_binary           d;
{
        extern canonical        result;

        d.sign = result.sign;

        switch (result.tag)
        {
         case NUM_TAG:
                d.exponent = result.exponent + 1023;

                /* Denormalized numbers have a bias of 1022. */
                if (result.msd == 0)
                        d.exponent = d.exponent - 1;
                d.fraction = result.fraction[1 ... 52];

         case ZERO_TAG:
                d.exponent = 0;
                d.fraction = 0;

         case INF_TAG:
                d.exponent = 2047;
                d.fraction = 0;

         case Q_NAN_TAG:
         case S_NAN_TAG:
                d.exponent = 2047;
                d.fraction = result.fraction[1 ... 52];
        }
}
```

8.3. Pack P754 Single-Extended

```
single_extended_pack_result(se)        single_extended_pack_result
        single_extended_binary          se;
{
        extern canonical                result;

        se.sign = result.sign;

        switch (result.tag)
        {
         case NUM_TAG:
                se.exponent = result.exponent + 1023;
                se.msb = result.msd;
                se.fraction = result.fraction[1 ... 31];

         case ZERO_TAG:
                se.exponent = 0;
                se.msb = 0;
                se.fraction = 0;

         case INF_TAG:
                se.exponent = 2047;
                se.msb = 0;
                se.fraction = 0;

         case Q_NAN_TAG:
         case S_NAN_TAG:
                se.exponent = 2047;
                se.msb = result.msd;
                se.fraction = result.fraction[1 ... 31];
        }
}
```

8.4. Pack P754 Double-Extended

```
double_extended_pack_result(de)  double_extended_pack_result
        double_extended_binary   de;
{
        extern canonical         result;

        de.sign = result.sign;

        switch (result.tag)
        {
         case NUM_TAG:
                de.exponent = result.exponent + 16383;
                de.msb = result.msd;
                de.fraction = result.fraction[1 ... 63];

         case ZERO_TAG:
                de.exponent = 0;
                de.msb = 0;
                de.fraction = 0;

         case INF_TAG:
                de.exponent = 32767;
                de.msb = 0;
                de.fraction = 0;

         case Q_NAN_TAG:
         case S_NAN_TAG:
                de.exponent = 32767;
                de.msb = result.msd;
                de.fraction = result.fraction[1 ... 63];
        }
}
```

9. Trimming the Result

This basic trim sequence applies to all operations that produce floating-point results. For simplicity, it is written as though every result would be trimmed, though in an actual implementation a trim sequence might be set up for each operation, and then applied only to finite, nonzero results.

---

trim_result

```
trim_result()
{
        under_result();
        round_result();
        over_result();
}
```

P754 permits three different underflow criteria (§7.4) when there is to be no trap on underflow:

(1) An intermediate result is less than the smallest normalized number, when tested before rounding, and does indeed suffer a rounding error in round_result().

(2) Like (1) except that tininess is tested after rounding as though the range were unbounded.

(3) The final result differs from what would have been computed were exponent range unbounded.

This implementation uses (1), which is perhaps the most straightforward to implement. In (2), the routine under_result() would follow rather than precede round_result() in sequence; a tiny, rounded result would be flagged as underflowed, "unrounded", and then sent back through round_result(). It can be shown that a result can be unrounded if it is known whether the result was rounded up in magnitude during the first application of round_result(). The most difficult to implement, (3), is similar to (2) in that under_result() would follow round_result(); however, the criterion for underflow is not that the rounded result be tiny and inexact, but that it be tiny and yet incapa-

ble of storage in the destination format without further alteration (i.e., it must be rerounded).

---

under_result

```
under_result()
{
        extern canonical        result;
        extern dst_str          dst;
        extern mode_str         mode;
        extern exc_str          error, trap;

        if (result.tag != NUM_TAG)          /* Bypass special results. */
                return;

        if (result.exponent >= dst.uthresh)          /* Tiny? */
                return;

        /*
         * Set tentative signal based on tininess only.  Flag will
         * be reset later if the result is exact.
         */
        error.uflow = TRUE;

        if (trap.uflow == FALSE)
                /* Denormalize... */
                shift_right(result, dst.uthresh - result.exponent);

        else
                /* System-dependent action, including... */
                result.exponent = result.exponent + dst.biasadjust;

}
```

---

round_result

```
round_result()
{
        extern canoncial        result;
        extern dst_str          dst;
        extern mode_str         mode;
        extern exc_str          error, trap;
        digit                   guard;
        bit                     sticky;
```

```
if (result.tag != NUM_TAG)          /* Bypass special results. */
{
        if (result.tag == Q_NAN_TAG)
        {
                /*
                 * System-dependent action to check that
                 * the quiet NAN has some nonzero digits
                 * in the leading dst.lsd digits.
                 */
        }
        return;
}

/* Guard is the next digit after rounding precision. */
guard  = result.frac[ (dst.lsd + 1) ];

/*
 * Sticky bit is 1 if and only if any digits beyond guard
 * are nonzero.  In includes the so-called round bit, which
 * already served its purpose in +, -, and /.
 */
if (result.frac[ (dst.lsd + 2) ... CANON_FRACTION ] != 0)
        sticky = 1;
else
        sticky = 0;

/*
 * Test for exact result.  If so, and underflow is not
 * trapped, then undo any tentative underflow signal.
 */
if ((guard == 0) && (sticky == 0))
{
        if (trap.uflow == FALSE)
                error.uflow = FALSE;
        return;
}
else
{
        error.inexact = TRUE;

        switch (mode.round)
        {
         /*
          * In the unlikely case of an odd radix, the half-way
          * case will never arise, and the following test
          * could be simplified.
          */
         case TO_NEAREST:
                if (guard > HALF_RADIX)
                        inc_result();
                else if (guard < HALF_RADIX)
                        chop_result();
                else
```

```
                                        /* (guard == HALF_RADIX) */
                                        if ((sticky == 1) ||
                                        (result.frac[dst.lsd] IS ODD))
                                                inc_result();
                                        else
                                                chop_result();

                        case TOWARD_0:
                                chop_result();

                        case TOWARD_MINUS:
                                if (result.sign == 1)
                                        inc_result();
                                else
                                        chop_result();

                        case TOWARD_PLUS:
                                if (result.sign == 0)
                                        inc_result();
                                else
                                        chop_result();
                        }
                }
}
```

---

over_result

```
over_result()
{
        extern canonical        result;
        extern dst_str          dst;
        extern mode_str         mode;
        extern exc_str          error, trap;

        if (result.tag != NUM_TAG)          /* Special operands. */
                return;

        if (result.exponent <= dst.othresh)
                return;

        if (trap.oflow == FALSE)
        {
                error.inexact = TRUE;       /* Inexact if untrapped. */
                error.oflow   = TRUE;

                if ( (mode.round == TO_NEAREST) ||
                ((mode.round == TOWARD_PLUS) && (result.sign == 0)) ||
```

```
                ((mode.round == TOWARD_MINUS) && (result.sign == 1)) )
                        result.tag = INF_TAG;
                else
                        huge_result();
        }
        else
        {
                /* System-dependent action, including... */
                result.exponent = result.exponent - dst.biasadjust;
        }
}
```

---

10. Low-Level Utility Routines

```
/* Short-hand for long mnemonic... */
# define        CF      CANON_FRACTION
```

When shifting right, 0 is shifted into c_out and fraction digits lost off the right are accumulated in the trailing digit.

---

shift_right

```
shift_right(w, cnt)
        canonical       w;
        int             cnt;
{
        while (cnt > 0)
        {
                /* Logically OR the last digit into the second last... */
                w.fraction[ CF ] = w.fraction[ CF ] | w.fraction[ CF - 1 ];

                /* ...before the right shift. */
                w.(c_out, msd, fraction) = w.(c_out, msd, fraction) >> 1;

                w.exponent = w.exponent + 1;        /* Adjust exponent. */
                cnt = cnt - 1;
        }
}
```

The arithmetic is such that left shifts may be made without regard to the special "sticky" nature of the lowest fraction digit. The carry-out digit c_out will always be 0.

---

shift_left

```
shift_left(w, cnt)
        canonical       w;
        int             cnt;
{
        while (cnt > 0)
        {
                /* Just shift left, with 0 into fraction[ CF ]. */
                w.(c_out, msd, fraction) = w.(c_out, msd, fraction) << 1;

                w.exponent = w.exponent - 1;        /* Adjust exponent. */
                cnt = cnt - 1;
        }
}
```

Normalize by shifting left. c_out and fraction[CF] are always 0. If all significant digits are zero, the number is set to Normal 0.

---

normalize

```
normalize(w)
        canonical       w;
{
        if ( w.(msd, fraction) == 0 )        /* Dismiss special case. */
                w.tag = ZERO_TAG;
        else
                while (w.msd == 0)
                {
                        w.(msd, fraction) = w.(msd, fraction) << 1;
                        w.exponent = w.exponent - 1;
                }
}
```

Increment by a unit in the last place of rounding precision. Then clean up trailing digits.

---

inc_result

```
inc_result()
{
        extern canonical        result;
        extern dst_str          dst;
        canonical               tmp;

        /* Set up dummy significant digit field for incrementation. */
        tmp.msd = 0;
        tmp.fraction = 0;
        tmp.fraction[dst.lsd] = 1;

        result.(c_out, msd, fraction) =
                result.(msd, fraction) + tmp.(msd, fraction);

        if (c_out != 0)                         /* Catch carry-out. */
                shift_right(result, 1);

        /* Clean up trailing digits. */
        result.fraction[ (dst.lsd + 1) ... CF ] = 0;
}
```

Chop at the last place of rounding precision.

---

chop_result

```
chop_result()
{
        extern canonical        result;
        extern dst_str          dst;

        result.fraction[ (dst.lsd + 1) ... CF ] = 0;
}
```

Set result to the largest number of the specified range and precision.

---

huge_result

```
huge_result()
{
        extern canonical        result;
        extern dst_str          dst;

        result.exponent = dst.othresh;          /* largest exponent */
        result.msd = RADIX_1;
        result.fraction[1 ... dst.lsd] = (RADIX_1, RADIX_1, ..., RADIX_1);
        result.fraction[ (dst.lsd + 1) ... CF ] = 0;
}
```

---

11. Operations

Each of the operations is broken into a large switch-case statement to handle the cases of zero, infinite, NAN, and normal operands. All operations on NANs are dealt with in the NAN-Hanlders section. Invalid operands are flagged for later processing during the Trim step.

In this implementation, all numeric inputs are normalized when unpacked, so there is no need for special provision for unnormalized operands. However, this is not required by P754. §3.1 and 3.3 explicitly allow an implementation to interpret unnormalized values in the sense of the obsolete Warning mode. This interpretation is discussed in chapter 5.

11.1. Add

Set result to the sum of op1 and op2.

add

```
add()
{
        extern canonical        op1, op2, result;
        extern mode_str         mode;
        extern exc_str          error;

        /*
         * Special              | 0  NUM INF NAN
         * case table:      -----+----------------
         *                    0  | A   B   B   F
         *                  NUM  | C   D   B   F
         *                  INF  | C   C   E   F
         *                  NAN  | F   F   F   F
         */

        switch ( op1.tag versus op2.tag )
        {
         case A: /* 0 + 0 */
                 result = op1;
                 if (op1.sign != op2.sign)
                         if (mode.round == TOWARD_MINUS)
                                 result.sign = 1;
                         else
                                 result.sign = 0;

         case B: /* op1 = 0 or op2 = INF */
                 result = op2;

         case C: /* op2 = 0 or op1 = INF */
                 result = op1;

         case E: /* op1 and op2 = INF */
                 if (op1.sign == op2.sign)
                         result = op1;
                 else
                         make_nan();

         case F: /* NANs! */
                 two_nans();

         case D: /* Typical case of two nonzero numbers. */
                 /* Arrange to have op1 >= op2 in magnitude. */
                 if (op2.exponent > op1.exponent)
                         swap(op1, op2);

                 /* Align op2's radix point with op1's. */
                 shift_right(op2, op1.exponent - op2.exponent);

                 if (op1.sign == op2.sign)
                 {
```

```
                    /* Add magnitude case. */

                    /* Tentative tag, sign, exponent. */
                    result = op1;

                    result.(c_out, msd, fraction) =
                            op1.(msd, fraction) + op2.(msd, fraction);

                    /* Handle possible carry-out. */
                    if (result.c_out != 0)
                            shift_right(result, 1);
            }
            else
            {
                    /* Subtract magnitude case. */

                    /*
                     * The following swap() prevents a borrow,
                     * which this notation is unequipped to describe.
                     */
                    if ( op2.(msd, fraction) > op1.(msd, fraction) )
                            swap(op1, op2);

                    /* Tentative tag, sign, exponent. */
                    result = op1;

                    result.(msd, fraction) =
                            op1.(msd, fraction) - op2.(msd, fraction);

                    /*
                     * Case of total cancellation --
                     * determine sign as in case A.
                     */
                    if (result.(msd, fraction) == 0)
                            if (mode.round == TOWARD_MINUS)
                                    result.sign = 1;
                            else
                                    result.sign = 0;

                    normalize(result);
            }
        }
}
```

11.2. Subtract

Set result to the difference of op1 and op2, using add().

```
subtract()                                                    subtract
{
        extern canonical        op2;

        /* Flip the sign of op2 with exclusive-or. */
        op2.sign = op2.sign ^ 1;
        add();
}
```

11.3. Multiply

Set result to the product of op1 and op2. When the product of two finite numbers is actually computed, the significant digit fields are interpreted as

<digit> . <fraction digits>

so that their product has the form

<carry-out digit> <digit> . <double-length fraction>

Only CANON_FRACTION fraction digits need be computed here, with the last digit reflecting the logical OR of all digits farther to the right of the "infinitely precise" result.

```
multiply()
{
        extern canonical        op1, op2, result;

        /*
         * Special              | 0  NUM INF NAN
         * case table:      -----+----------------
         *                    0  | A  A   C   E
         *                  NUM  | A  B   D   E
         *                  INF  | C  D   D   E
         *                  NAN  | E  E   E   E
         */

        /* Sign is exclusive-or of operand signs. */
        result.sign = op1.sign ^ op2.sign;

        switch ( op1.tag versus op2.tag )
        {
          case A: /* 0 times finite. */
                  result.tag = ZERO_TAG;

          case C: /* 0 times INF. */
                  make_nan();

          case D: /* INF times nonzero. */
                  result.tag = INF_TAG;

          case E: /* NANs! */
                  two_nans();

          case B: /* Two finite, nonzero numbers. */
                  result.exponent = op1.exponent + op2.exponent;

                  result.(c_out, msd, fraction) =
                          op1.(msd, fraction) * op2.(msd, fraction);

                  /*
                   * Watch for carry-out -- product of numbers
                   * between 1 and RADIX may exceed RADIX,
                   * requiring a one-digit shift.
                   */
                  if (result.c_out != 0)
                          shift_right(result, 1);
        }
}
```

11.4. Divide

Set result to the quotient op1 / op2. When the actual quotient of two numbers must be computed, the significant digit fields are interpreted as

<digit> . <fraction digits>

so that the quotient takes the form

<digit> . <fraction string, perhaps nonterminating>

Only CANON_FRACTION correct fraction digits need be computed, with the last of them reflecting the logical OR of all digits farther to the right.

---

divide

```
divide()
{
        extern canonical        op1, op2, result;
        extern exc_str          error;

        /*
         * Special              | 0  NUM  INF  NAN
         * case table:          -----+-----------------
         *                       0   | A    B    B    F
         *                      NUM  | C    D    B    F
         *                      INF  | E    E    A    F
         *                      NAN  | F    F    F    F
         */

        /* Result sign is exclusive-or of operand signs. */
        result.sign = op1.sign ^ op2.sign;

        switch ( op1.tag versus op2.tag )
        {
          case A: /* 0/0 or INF/INF. */
                  make_nan();

          case B: /* 0/NONZERO or finite/INF. */
                  result.tag = ZERO_TAG;
```

```
            case C: /* finite/0. */
                    result.tag = INF_TAG;
                    error.div_zero = TRUE;

            case E: /* INF/finite. */
                    result.tag = INF_TAG;

            case F: /* NANs! */
                    two_nans();

            case D: /* finite/finite */
                    result.exponent = op1.exponent - op2.exponent;
                    result.(msd, fraction) =
                            op1.(msd, fraction) / op2.(msd, fraction);

                    /*
                     * Quotient of two values between 1 and RADIX
                     * may be less than 1, in which case a one-digit
                     * shift is required.
                     */
                    if (result.msd == 0)
                            shift_left(result, 1);
        }
}
```

11.5. Remainder

Find the value result such that

$$op1 = (op2 \times Q) + result$$

where Q is an integer and

$$|result| \leq 0.5 \times |op2| ,$$

with Q an even integer in the case of equality. Q need not be delivered, though its sign and several low-order bits would be useful for trigonometric argument reduction.

In principle, result may be computed by computing all of the integer bits of op1/op2 (discarding the high-order 1's) and fixing up

the remainder to satisfy the above inequality. However, it turns out in practice to be easier to compute Q and the first fraction quotient bit and then fix the remainder. The fraction bit aids in checking the inequality.

According to §5.1, precision control is not to apply to remainder. Thus, the result doesn't require rounding. Even if op2 is tiny and the remainder falls below the underflow threshold, the result will be exact and so will not underflow.

---

remainder

```
remainder()
{
        extern canonical        op1, op2, result;
        extern mode_str         mode;
        extern exc_str          error;
        int                     Q, Qsign;

        /*
         * Special              | 0  NUM INF NAN
         * case table:      -----+----------------
         *                   0  | A  B   B   D
         *                  NUM | A  C   B   D
         *                  INF | A  A   A   D
         *                  NAN | D  D   D   D
         */

        Qsign = op1.sign ^ op2.sign;        /* Quotient sign. */

        switch ( op1.tag versus op2.tag )
        {
          case A: /* op1 rem 0 or INF is invalid. */

                  error.invalid = TRUE;

          case B: /* X rem INF and 0 rem Y are trivial. */

                  result = op1;

          case D: /* NANs! */

                  two_nans();
```

```
case C: /* finite rem finite. */

        /* Set tentative sign and exponent. */
        result.sign = op1.sign;
        result.exponent = op2.exponent;

        /* Generate all integer and one fraction quotient bits. */
        Q = LOW(op1.exponent - op2.exponent + 2)
                BITS OF QUOTIENT;
        result.(msd, fraction) = REMAINDER;

        /* Low bit of Q = 1 when REM is at least half op2. */
        if ((Q & 1) == 1)
        {

                if (result.(msd, fraction) == 0)
                {

                        /*
                         * Half-way case -- result
                         * has half magnitude of op2,
                         * with sign flipped if
                         * integer Q is odd.
                         */
                        result.(msd, fraction) =
                                op2.(msd, fraction);
                        result.exponent = result.exponent - 1;

                        if ((Q & 2) == 2)
                        {
                                /* Test low integer bit of Q. */
                                result.sign = result.sign ^ 1;
                                Q = Q + 2;
                        }
                }

                else
                {

                        /* More than half-way. */
                        result.sign = result.sign ^ 1;
                        result.(msd, fraction) =
                                op2.(msd, fraction)
                                - result.(msd, fraction);
                        Q = Q + 2;
                }
        }

        normalize(result);
```

```
                /*
                 * Now Q and its sign are available...
                 */
        }
}
```

11.6. Compare

Compare op1 and op2 and return the condition EQUAL, LESS THAN, GREATER THAN, or UNORDERED. To implement the language aspects of comparisons, two versions of the comparison instruction are useful, one that triggers Invalid on UNORDERED and one that is silent. See chapter 6 for further details.

---

compare(iftrigger) *compare*

```
        /* Trigger invalid error if operands are UNORDERED? */
        boolean iftrigger;

{
        extern canonical        op1, op2;
        extern mode_str         mode;
        extern exc_str          error;
        int                     cond;

        /*
         * Special              | 0  NUM INF NAN
         * case table:      -----+----------------
         *                   0  | A  B   B   F
         *                  NUM | C  D   B   F
         *                  INF | C  C   E   F
         *                  NAN | F  F   F   F
         */

        switch ( op1.tag versus op2.tag )
        {

          case A: /* 0 vs 0. */

                cond = EQUAL;
```

```
case B: /* Sign of op2 determines. */

        if (op2.sign == 0)
                cond = LESS;
        else
                cond = GREATER;

case C: /* Sign of op1 determines. */

        if (op1.sign == 0)
                cond = GREATER;
        else
                cond = LESS;

case E: /* INF vs INF. */

        if (op1.sign == op2.sign)
                cond = EQUAL;
        else if (op1.sign == 0)
                cond = GREATER;
        else
                cond = LESS;

case F: /* NANs! */

        /*
         * Call NAN-handler to deal with exceptions
         * like signaling NANs, but ignore the setting
         * of the result.() structure.
         */
        two_nans();
        cond = UNORDERED;

case D: /* finite vs finite */

        if (op1.sign != op2.sign)

                /* Trivial if signs differ. */
                if (op1.sign == 0)
                        cond = GREATER;
                else
                        cond = LESS;

        else
        {

                /*
                 * Since operands are prenormalized,
                 * unequal exponents determine order.
                 */
                if (op1.exponent > op2.exponent)
                        if (op1.sign == 0)
                                cond = GREATER;
```

```
                                else
                                        cond = LESS;

                        else if (op1.exponent < op2.exponent)
                                if (op1.sign == 0)
                                        cond = LESS;
                                else
                                        cond = GREATER;

                        else if (op1.(msd, fraction) > op2.(msd, fraction))
                                if (op1.sign == 0)
                                        cond = GREATER;
                                else
                                        cond = LESS;

                        else
                                if (op1.sign == 0)
                                        cond = LESS;
                                else
                                        cond = GREATER;
                }
        }

        /* Raise a flag if necessary. */
        if ((iftrigger == TRUE) && (cond == UNORDERED))
                error.invalid = TRUE;

        return(cond);
}
```

11.7. Round to Integer

Set result to op1, rounded to an integer.

---

rnd_integer

```
rnd_integer()
{
        extern canonical        op1, result;
        extern mode_str         mode;
        extern exc_str          error;

        /*
         * Special              | 0  NUM  INF  NAN
         * case table:          -----+----------------
         *                      0  | A   B    A    C
         */
```

```
        switch ( op1.tag )
        {
         case A: /* int(zero or INF) is itself. */
                 result = op1;
         case C: /* NAN! */
                 one_nan();
         case B: /* typical case of finite number */
                 result = op1;
                 /*
                  * Nothing to be done if exponent is bigger than
                  * the index (since it's already an integer).
                  * Otherwise right-align the significant digits
                  * to round off the fraction part.
                  */
                 if (result.exponent < dst.lsd)
                         shift_right(result, (dst.lsd - result.exponent));
                 round_result();              /* May be unnormalized. */
                 normalize(result);
        }
}
```

11.8. Square Root

Set result to the square root of op1. The core of this operation is the computation of the square root of a number between 1 and RADIX × RADIX, which root is always of the form d.ddd before rounding. After CANON_FRACTION correct fraction digits of the root are found, a 1 should be logically OR-ed into the last digit of result.fraction to signal the nonzero digits further to the right.

```
sqrt()
{
        extern canonical        op1, result;
        extern mode_str         mode;

        /*
         * Special              | 0  NUM  INF  NAN
         * case table:      -----+-----------------
         *                    0  | A   B    C    D
         */

        switch ( op1.tag )
        {
         case A: /* sqrt( +/- 0 ) is +/- 0 (\(sc5.2). */
                 result = op1;

         case C: /* Only sqrt( +INF ) is valid. */
                 if (op1.sign = 0)
                         result = op1;
                 else
                         make_nan();

         case D: /* NAN! */
                 one_nan();

         case B: /* sqrt(finite). */
                 /* Negative values are invalid. */
                 if (op1.sign == 1)
                         make_nan();

                 else
                 {
                         /* Handle odd exponents with care. */
                         if (op1.exponent & 1)
                                 shift_left(op1, 1);

                         result.sign = 0;
                         result.exponent = op1.exponent / 2;

                         result.c_out = 0;
                         result.(msd, fraction) =
                                 root(op1.(c_out, msd, fraction));
                 }
        }
}
```

11.9. NAN-Handler

The treatment of NANs is quite system-dependent. The intention is that quiet NANs should propagate through operations without generating exceptions. When two operands are such NANs, a system-dependent precedence rule should arbitrate, designating one of the input NANs as the result. The choice should be made on the basis of the operands' fraction fields only (see §6.2 of P754, especially the last paragraph, and the discussion of NANs in chapter 2).

Signaling NANs generate an exception whenever they are touched, presumably because the user has some specific interpretation to be effected by special trap handling software. Signaling NANs might also be used by a system to provide a menu of alternatives to the default exception handling schemes provided by the arithmetic.

---

two_nans

```
two_nans()
{
        extern canonical        op1, op2, result;
        canonical               precedent_nan();

        /*
         * Special              | Q_NAN  S_NAN  ELSE
         * case table:  -------+--------------------
         *                Q_NAN |  A      B     C
         *                S_NAN |  B      B     B
         *                ELSE  |  D      B     NA
         */

        switch ( op1.tag versus op2.tag )
        {
          case A: /* Two quiet NANs. */
                  result = precedent_nan(op1, op2);

          case B: /* One or two signaling NANs. */
                  make_nan();
```

```
            case C: /* op1 is quiet NAN, op2 is ELSE. */
                    result = op1;

            case D: /* op2 is quiet, op1 is ELSE. */
                    result = op2;
            }
}
```

one_nan *one_nan*

```
one_nan()
{
            extern canonical            op1, result;

            if (op1.tag == Q_NAN_TAG)
                    result = op1;
            else
                    make_nan();
}
```

make_nan *make_nan*

```
make_nan()
{
            extern canonical            result;
            extern exc_str              error;

            error.invalid = true;

            /*
             * Set result to some quiet NAN, perhaps indicating the
             * nature of the error.
             */
}
```

CHAPTER 9

Intermediate Exponent Calculations

1. Introduction

Proposed IEEE standard P754 for binary floating-point arithmetic specifies that results be computed as though with unbounded range and precision and then coerced to within the constraints of the destination number format. Just how much exponent range is required for the "infinitely precise intermediate result" is the subject of this brief chapter.

Among the unusual features of P754 are the so-called denormalized numbers, which alleviate some common problems due to exponent underflow (see chapter 5). The denormalized numbers effectively extend the exponent range of the host format by a small amount, though this is not their primary purpose. But just this small amount can have a serious impact on exponent calculations. For example, a typical implementation of the P754 double-extended format will use 15 exponent bits, biased by $3FFF_{16}$. Since multiplication and division entail adding and subtracting their operands' exponents, one extra exponent bit — for a total of 16 — would seem to suffice for intermediate results, pending checks for overflow and underflow. However, the extra range afforded by the denormalized numbers is slightly wider than can be covered by 16 bits alone. We will see how an implementor can make do with 16 bits when the cost of an extra exponent bit is very high.

Throughout this chapter, all four-digit integer constants are hexadecimal unless otherwise indicated.

2. An Implementation

In P754, extended formats are specified by lower bounds on the range and precision to be provided. For definiteness, let us assume a double-extended format with a biased 15-bit exponent ranging from 0000 to 7FFF, including an added 3FFF. Suppose that the maximum exponent, 7FFF, is reserved to encode $\pm\infty$ and NANs, so the unbiased exponent ranges from −3FFF to 3FFF for finite numbers. If there are 64 significant bits, all of them explicit, then the set of finite representable numbers is

$$\pm 2^n \times b_0 . b_1 b_2 b_3 \cdots b_{63}$$

where $-3FFF \le n \le 3FFF$. The special value zero is encoded with an exponent −3FFF and all significant bits zero. Three numbers are of particular interest in what follows:

$$\begin{aligned} B &= 2^{3FFF} \times 1.111\cdots 11 && = \text{biggest normalized} \\ S &= 2^{-3FFF} \times 1.000\cdots 00 && = \text{smallest normalized} \\ D &= 2^{-3FFF} \times 0.000\cdots 01 && = \text{smallest denormalized} \\ &= 2^{-403E} \times 1.0 \end{aligned}$$

3. Extreme Overflows and Underflows

The extreme cases for intermediate results are these:

$$\begin{aligned} B \times B &= 2^{7FFF} \times 1.111\cdots bbbb\cdots \\ &= 2^{8000} \times 1.0 \quad \text{rounded to single or double precision} \\ B / S &= 2^{7FFE} \times 1.111\cdots 11 \\ &= 2^{7FFF} \times 1.0 \quad \text{rounded to single or double precision} \\ B / D &= 2^{803D} \times 1.111\cdots 11 \\ &= 2^{803E} \times 1.0 \quad \text{rounded to single or double precision} \\ S \times S &= 2^{-7FFE} \times 1.0 \\ D \times D &= 2^{-807C} \times 1.0 \end{aligned}$$

$$S / B = 2^{-7FFF} \times 1.000\cdots 0bbbb\cdots$$

$$D / B = 2^{-803E} \times 1.000\cdots 0bbbb\cdots$$

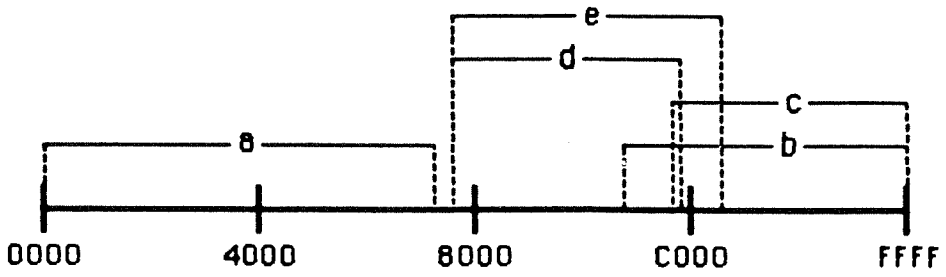
The range covered by results involving only normalized numbers is −7FFF to 8000, a total of 2^{16} values. This may barely be covered with a 16-bit intermediate exponent. However, with denormalized inputs the effective range is −807C to 803E. Of course, a 17-bit exponent covering the range -10000_{16} to $0FFFF_{16}$ would more than suffice for intermediate calculations, but the cost of the seventeenth bit may be high. The rest of this paper discusses a way to get by with just sixteen bits.

4. Overflow and Underflow Ranges

Suppose that floating-point arithmetic is performed with a 16-bit intermediate exponent biased by 3FFF. And suppose that exponent calculations are performed in integer arithmetic, modulo 2^{16}, as in two's-complement signed arithmetic. Then the exponent ranges of interest in unbiased and biased forms are:

Case	Unbiased Range	Biased Range
(a) unexceptional	−3FFF to 3FFF	0000 to 7FFE
(b) × underflow	−807C to −4000	B783 to FFFF
(c) / underflow	−803E to −4000	BFC1 to FFFF
(d) × overflow	4000 to 8000	7FFF to BFFF
(e) / overflow	4000 to 803E	7FFF to C03C

Here they are on a number line:



The amount of range in excess of sixteen bits is shown by the overlapping overflow and underflow ranges of × and /.

5. Facts about Over/Underflow

Only double-extended products and quotients are susceptible to ambiguous overflow and underflow cases when a 16-bit exponent is used for intermediate values. An exponent in the range [BF83, BFC1] is either overflowed or underflowed.

Let's call *big* any extended number with a biased exponent larger than, say, 7F00 and call *small* any extended number with a biased exponent smaller than 0100 (this includes the denormalized numbers). The extreme underflow cases can arise only from

$$small \times small \quad \text{or} \quad small \;/\; big$$

and the extreme overflow cases can arise only from

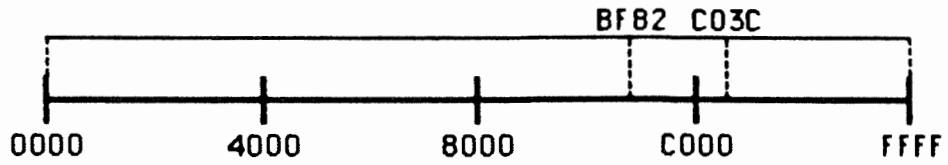
$$big \times big \quad \text{or} \quad big \;/\; small \;.$$

This suggests that the ambiguous cases can be resolved by checking the left operand: if it is *small* the result has underflowed, and if it is *big* the the result has overflowed.

6. Tests of Overflow and Underflow

As explained in §7.4 of P754 and in chapter 8, underflow is signaled when a result is both inexact (that is, rounded) and tiny. Tininess is the conventional criterion that a value underflows when it falls below a certain threshold. However, the denormalized numbers enable unconventionally tiny values to be represented. So underflow is signaled only when a tiny value suffers some *unusual* loss of accuracy due to denormalization. This section discusses only the tininess criterion. Chapter 8 treats both underflow criteria.

When testing a result for tininess, three intervals are of particular interest:



[0000, BF82] – result cannot be tiny (though overflow may be detected later).

[BF83, C03C] – result is tiny if and only if the left operand is *small* (otherwise the left operand must be *big* and overflow will be detected later).

[C03D, FFFF] – result is unambiguously tiny.

To test whether the left operand is *small* it suffices to check whether its biased exponent is at most 4000, unsigned;. that is, simply ensure that the exponent is not *big*.

In P754, the test for tininess always precedes the test for overflow. Thus the ambiguous cases are eliminated by the time overflow is tested. The test for overflow is simply:

if exponent < 7FFE **then** either in range or already underflowed...

else overflow...

where the 16-bit comparison is unsigned.

7. Single and Double Results

Since P754 specifies that products and quotients involving extended operands cannot be delivered directly to single or double destinations, the ambiguous cases cannot arise there. In a so-called "extended based" system which delivers all arithmetic results to extended destinations, single and double destinations only arise in format conversions. On such a system, the test for tininess in extended → double conversion is

if exponent < 3C01 **then** underflow...

else in range or overflowed...

where the comparison is signed two's-complement. The signed comparison is used to catch denormalized inputs which, when prenormalized, have exponents of the form FF*xx* – modest negative numbers in the two's-complement system. There is no problem with overflowed exponents like 80*xx* because the largest finite extended input has exponent 7FFE. The situation for extended → single format conversion is analogous.

8. Summary

The cost of keeping a 16-bit exponent for intermediate results is a slightly more complicated test for tininess, using two thresholds, and the need to inspect the exponent of one of the input operands. The extra nuisance may be small compared to the cost of a seventeenth exponent bit for all exponent calculations when there is a natural 16-bit boundary, as is the case with some bit-slice and software implementations.

CHAPTER 10

A Compact Test Suite for P754 Arithmetic — Version 2.0

The initial version of this test data base for the proposed IEEE 754 binary floating point standard (draft 8.0) was developed for Zilog, Inc. and was donated to the floating point working group for dissemination. Errors in or additions to the distributed data base should be reported to the agency of distribution, with copies to Zilog, Inc., 1315 Dell Avenue, Campbell, CA, 95008.

The above statement, which is to accompany any copy of this test suite, indicates the origin of this effort. The author developed the tests while employed at Zilog. Since then, with help from James W. Thomas of Apple Computer, the tests have been expanded and updated to conform to draft 10.0 of proposed IEEE standard P754 for binary floating point arithmetic.

1. Distribution format

The data base consists of several files of ASCII data: this description, the test vectors [Appendix B], and a sample Pascal program to drive the tests [Appendix C].

Currently, the tests are available on an unlabeled magnetic tape, 1600 BPI, composed of physical blocks of 40 "card images" of 80 ASCII characters. Files are separated by file marks, with a double file mark at the end of the last file. The tape may be obtained by mailing $100 (payable to the Regents of the University of California) to Keith Sklower, Computer Science Division, Evans Hall, University of California, Berkeley, CA, 94720.

2. The design goal

Our object was to exercise the P754 arithmetic, the special case logic in particular, with as terse a test set as possible. By keeping the test fields

brief we could generate new tests by simply typing the vectors ourselves, rather than using a table-driven or random scheme. And it was easy to update the data base as new cases occurred to us and errors were detected. Most important, the tests were designed to be as format-independent as possible, so that the same vectors would apply to all formats – single, double, single-extended, and double-extended – without regard to the implementation-dependent features of the extended formats.

No claim is made about the completeness of these tests. Attempting to maintain format independence led to two important restrictions. First, we could not describe arbitrary bit patterns, so we were limited to a special class of numbers, roughly speaking, "simple" numbers modified in their low-order bits and possibly scaled up or down. Second, the tests were written as though all operations were of the form

$$x \text{ op } y \rightarrow z$$

where x, y, and z all have the same format. However, this is not the architecture of several known microprocessor implementations. Those implementations are fundamentally two-address, with extended format destinations for all operations except conversion from extended to a narrower format. The test suite does not explicitly test such mixed-format operations. But with care such operations can be used to simulate the type of architecture the test vectors apply to – even though this simulation will not be used for ordinary calculations.

P754 is really a specification of a programming environment. This test scheme simply exercises an arithmetic engine that purports to "support" the proposed standard. Thus the tests do not address the more global P754 issues such as which formats are supported, how expression evaluation is

carried out (including possible provision for precision control), how comparisons are handled, how binary-decimal conversion is provided (and how accurate it is), and how exceptions are reported.

3. Test vector format

The test vectors are contained in several files of ASCII text. Each line of a test file is either a comment (beginning with '!' or entirely blank), or a test vector such as:

```
2*  =  1i1  -1i2  x  -1i3  an inexact product
```

The leading '2' is the version number; the first version of the tests, distributed through 1982, had no version number. This particular example is a product (*) with rounding to nearest (=). The factors are 1.0 incremented (i) by a unit in its last place (to the precision of the format under consideration), and the negative of 1.0 incremented in magnitude by two units in its last place. The result, which is inexact (x), is the negative of 1.0 incremented by three units in its last place.

Each test vector consists of seven fields: version number and operator, modes, first operand, second operand, result flags, result, comment. The fields are separated by white space — blanks or tabs; thus, no field but the last may be blank, and only the last field can itself contain white space. In the case of unary operations like square root, the value "0" is used as a placeholder for the second operand.

The operators supported in version 2.0 of the tests are: +, −, *, /, C (compare), V (square root), % (remainder), I (round to integer), N (nextafter), A (absolute value), ~ (negate), @ (copysign), S (scalb), L (logb), and F (fraction part). The last seven operators are taken from the P754 Appendix

(F is a combination of S and L, as shown in the accompanying program). They are recommended but not required by P754; they were not included in version 1.1.

The modes are = (round to nearest), 0 (round toward 0), < (round toward −infinity), > (round toward +infinity), s (single operands), d (double operands), t (single extended operands), e (double extended operands). The modes s, d, t, and e are used when the result explicitly depends on a specific exponent range or precision; thus, modes t and e must be used with great care since those formats are implementation-dependent. Modes for the affine and projective interpretations of infinity and for the normalizing and warning interpretations of denormalized numbers were included in version 1.1, but they are omitted here since the projective and warning modes were removed from P754 in the passage from draft 8.0 to draft 10.0. In the notation of draft 8.0, all operations in the version 2.0 tests are run implicitly in the affine and normalizing modes. If one or more rounding modes appears in a vector, then the test is run in those modes only; otherwise, the test is run for all rounding modes. Similarly, if any format restrictions are listed then they exclude any others. If a test applies to all formats in all rounding modes then the key "ALL" is used as a placeholder, since the mode field must be non-empty.

The error flags are o (overflow), x (inexact), i (invalid operation), z (division by zero), and u/v/w (underflow). There are three flags for underflow since P754 now permits an implementor to use any one of three slightly different definitions of underflow for all operations. In the language of section 7.4 of P754, u indicates underflow due to tininess and "extraordinary" error; v indicates underflow due to tininess and inexactness, where tininess is

tested after rounding; and w indicates underflow due to tininess and inexactness, where tininess is tested before rounding. The three definitions are nested in the sense that u-underflow implies v-underflow which in turn implies w-underflow. The three definitions differ in subtle ways, and a few multiply and divide tests have been devised to distinguish them. Version 1.1 had two other error flags, d and t, concerning denormalized and signaling NAN operands, specific to the original Zilog implementation; these have been omitted from version 2.0. Unexceptional tests have the key "OK" in the result flag field as a placeholder.

A numeric operand field is scanned left to right. It consists of an optional sign, a mandatory root number, and zero or more modifier suffixes. The sign is + or −; as usual, plus is presumed if the sign is omitted. Root numbers are of several types: integers, NANs, and tiny and huge numbers. The single-digit integers 0, 1, ..., 9 speak for themselves. S and Q signify signaling and quiet NANs, respectively (T and N were used in version 1.1 corresponding to the oblsolete names "trapping" and "nontrapping"). Ex, where x is a single digit, is a tiny power of two: E0 is the smallest normalized number, E1 is twice E0, E2 is twice E1, etc. Similarly, Hx is a huge power of two: H0 is infinity (a special case), H1 is the largest power of two, H2 is half of H1, etc. Finally, there is a notation for specifying arbitrary root values, though it is intended for further expansion of the test vectors and is not used in version 2.0. The form is:

$$\$xxx\cdots x\hat{}yyy\cdots y$$

The dollar sign indicates that a literal root value follows. The x-field is a string of hex digits with an implicit binary point after the leading bit of the leading hex digit. The y-field is the decimal exponent (optionally signed) of

two. The value represented is thus

$$0.xxx\cdots x * (2 \hat{} (yyy\cdots y + 1))$$

with the binary point moved over to the left of the x-field for notational convenience.

The five suffixes have the form sK, where s is one of i, d, u, p, or m and K is a digit 0, 1, ..., 9. The increment (i) and decrement (d) suffixes cause the root value to be altered by K units in its last place (ulps). The ulp (u) operator replaces the root value by K units in its last place. The plus (p) and minus (m) operators cause the root value to be scaled up or down by $2\hat{}K$. Since it is easier to see how the operators apply than to enumerate formal rules, further discussion is deferred until several examples have been presented.

4. Sample Numerical Values

The following list of numerical operands illustrates most of the subtleties of the test vector representation. The subsequent text discusses the examples.

Test Operand	Mathematical Value	Single Format Encoding
1	1	3F80 0000
1i1	1 + (2^−23)	3F80 0001
1d1	1 − (2^−24)	3F7F FFFF
1u1	2^−23	3400 0000
1p1	1 * 2	4000 0000
1m1	1 * 2^−1	3F00 0000
2	2	4000 0000
−2i3	−(2 + 3*(2^−22))	C000 0003
2u1	2^−22	3480 0000
2i3u1	2^−22	3480 0000
2d1u1	2^−23	3400 0000
−2p1	−2 * 2^1	C080 0000
2m1	2 * 2^−1	3F80 0000
$800000^1	2	4000 0000

$800001^1	2 + (2^-22)	4000 0001
3i1	3 + (2^-22)	4040 0001
3u1	2^-22	3480 0000
4	4	4080 0000
4m1	2	4000 0000
0	0	0000 0000
-0	-0	8000 0000
0i5	5 * (2^-149)	0000 0005
-0i2	-2 * (2^-149)	8000 0002
E0	2^-126	0080 0000
E0i1	(2^-126) + (2^-149)	0080 0001
E0d1	(2^-126) - (2^-149)	007F FFFF
E0i1u1	2^-149	0000 0001
E0d1u1	2^-149	0000 0001
E0m1	(2^-126) * (2^-1)	0040 0000
H0	infinity	7F80 0000
H0d1	(2^128) - (2^104)	7F7F FFFF
H0m1	(2^128) * (2^-1)	7F00 0000
H1	2^127	7F00 0000
-Q	negative quiet NAN	FF81 0000
S	signaling NAN	7FC1 0000

The increment (i) and decrement (d) operators are defined to yield the next represenatable value to the number to which they are applied. When the root value is a power of two and is greater than E0, the amounts incremented and decremented differ by a factor of two. Compare, for example, 1i1 and 1d1. However, when the root value is a power of two no bigger than E0 (the smallest denormalized number), the magnitude of the increment and decrement are the same, namely the value of the tiniest denormalized number. This follows from the fact that numbers in the range E0 to E1 have the same spacing as the numbers in the range 0 to E0.

There are two special cases of i and d. 0i1 is the tiniest denormalized number (that is, the next representable number to 0), and in general 0i*K* is defined to be *K* times 0i1. When H0, representing infinity, is decremented, as

in H0d1 above, H0 behaves as though it had the value 2^128, that is the smallest power of 2 too large to represent.

The ulp operator (u) gives units in the last place of the number to which it is applied. The operator is motivated by the need to describe the results of magnitude subtractions. The ulp operator may best be thought of as satisfying the following formula: for any value X, $X\mathrm{u}K = X\mathrm{i}K - X$. Thus only the exponent of X, not its significand, determines the magnitude of the ulps. For example, 2u1, 2i3u1, and 3u1 all have the same value since the root values 2, 2i3, and 3 all of the form (2^1)*1.f.

The scaling operators p and m typically affect only the exponent of a number, as in the cases 1p1 and 4m1, both of which equal 2. However, when the root value is no bigger than E0, the scaled value must be denormalized, as in the case of E0m1 above.

The NAN root values Q and S are system-dependent since P754 specifies only that they have the maximum exponent and some nonzero bits in the significand. In the examples shown, the leading *fraction* bit is used to distinguish the two kinds of NAN.

A negative sign applies to the number as a whole, as in −2i3 above. Regardless of any sign, the increment and decrement operators add and subtract in magnitude, respectively.

5. Sample Driver Program

Appendix C contains a Pascal program which has been used to run the test vectors. The program was developed by James W. Thomas and the author and has been run on both an Apple III and an Apple Lisa computer (using prototype floating point software just becoming available as products).

The program is broken into three parts, the main program FPTEST and two "units" (in the notation of UCSD Pascal) FP and FPSOFT.

FPTEST parses the test vectors, builds the numeric operands in a canonical format, invokes FP to run the tests, and checks the results.

The unit FP is composed of subprograms to pack canonical values into the P754 storage types and to perform single, double and extended format tests. This unit is highly implementation-dependent. If an extended format is implemented, then packing from the canonical format to extended will depend on details of the extended format. Even packing into the single and double formats depends on the ordering of the bytes in the 32 and 64 bit words. FP invokes the actual arithmetic operations to be tested; in some cases, such as this sample program, the arithmetic is available only through subroutine calls. The unit FPSOFT describes one interface to such routines. FP simulates single-only, double-only, and extended-only operations. In this sample program the arithmetic is two-address extended-based so extra care is taken to avoid the so-called double-rounding that may arise when a result is computed in an extended intermediate variable and then stored (and possibly rounded again) to a single or double destination. It can be shown that because the extended format has more than twice as many significant bits as does the single format, this hazard only arises in double format tests. (We note again that this restriction to operations on just one format is an arbitrary constraint set by the test scheme, NOT by P754.)

FPSOFT is an hypothetical interface to a floating point package, to supply the operations needed by FP. Of course, this unit would not be required if the host system fully supported floating point arithmetic right in Pascal, in which case the unit FP could be greatly simplified.

APPENDIX A

Excerpts from a Proposed Standard for Binary Floating-Point Arithmetic

Based on Draft 10.0 of IEEE Task P754 December 2, 1982

Foreword

This foreword and the footnotes are not part of IEEE Standard 754 for Binary Floating-Point Arithmetic.

This standard is a product of the Floating-Point Working Group of the Microprocessor Standards Subcommittee of the IEEE Computer Society Computer Standards Committee. Draft 8.0 of this standard was published to solicit public comments.[1] Implementation techniques can be found in "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic" by Jerome T. Coonen,[2] which was based on a still earlier draft of the proposal.

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

(1) Facilitate movement of existing programs from diverse computers to those that adhere to this standard.

(2) Enhance the capabilities and safety available to programmers who, though not expert in numerical methods, may well be attempting to produce numerically sophisticated programs. However we recognize that utility and safety are sometimes antagonists.

(3) Encourage experts to develop and distribute robust and efficient numerical programs portable, via minor editing and recompilation, onto any computer that conforms to this standard and possesses adequate capacity. When restricted to a declared subset of the standard, these programs should produce identical results on all conforming systems.

(4) Provide direct support for

Execution-time diagnosis of anomalies,

Smoother handling of exceptions, and

Interval arithmetic at a reasonable cost.

(5) Provide for development of

Standard elementary functions like exp and cos,

Very high precision (multi-word) arithmetic, and

---

[1] *Computer*, Vol. 14, No. 3, March 1981.

[2] *Computer*, Vol. 13, No. 1, January 1980.

Coupling of numerical and symbolic algebraic computation.

(6) Enable rather than preclude further refinements and extensions.

Contents

FIGURES

Excerpts from a Proposed Standard for Binary Floating-Point Arithmetic
Based on Draft 10.0 of IEEE Task P754 December 2, 1982

1. Scope

1.1. Implementation objectives. It is intended that an implementation of a floating-point system conforming to this standard can be realized entirely in software, entirely in hardware, or in any combination of software and hardware. It is the environment the programmer or user of the system sees that conforms or fails to conform to this standard. Hardware components that require software support to conform shall not be said to conform apart from such software.

1.2. Inclusions. This standard specifies

(1) Basic and extended floating-point number formats;

(2) Add, subtract, multiply, divide, square root, remainder and compare operations;

(3) Conversions between integer and floating-point formats;

(4) Conversions between different floating-point formats;

(5) Conversions between basic format floating-point numbers and decimal strings; and

(6) Floating-point exceptions and their handling, including non-numbers (NaNs).

1.3. Exclusions. This standard does not specify

(1) Formats of decimal strings and integers,

(2) Interpretation of the sign and significand fields of NaNs, or

(3) Binary↔decimal conversions to and from extended formats.

2. Definitions

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

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

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

Destination. 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 format as well as the operands' values.

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

Fraction. The field of the significand that lies to the right of its implied binary point.

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

The following mode shall be implemented:

(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.

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

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

Shall 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.

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

User. Any person, hardware, or program not itself specified by this standard, having access to and controlling those operations of the programming environment specified in this standard.

3. Formats

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

3.1. Sets of values. This section concerns only the numerical values representable within a format, not the encodings which are the subject of the following sections. The only values representable in a chosen format are those specified via the following three integer parameters:

p – the number of significand bits (precision),

E_{max} – the maximum exponent, and

E_{min} – the minimum exponent.

Each format's parameters are displayed in Table 1. Within each format just the following entities shall be provided:

Numbers of the form $(-1)^s 2^E (b_0 \cdot b_1 b_2 \cdots b_{p-1})$ where
s is 0 or 1,
E is any integer between E_{min} and E_{max}, inclusive, and
each b_i is 0 or 1;

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

At least one signaling NaN; and

At least one quiet NaN.

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

$$2^0(1 \cdot 0) = 2^1(0 \cdot 1) = 2^2(0 \cdot 01) = \cdots .$$

However, the encodings of such nonzero values may be redundant only in extended formats (3.3). The nonzero values of the form $\pm 2^{E_{min}}(0 \cdot b_1 b_2 \cdots b_{p-1})$ are called denormalized. Reserved exponents may be used to encode NaNs, $\pm\infty$, ± 0, and denormalized numbers. For any variable that has the value zero, the sign bit s provides an extra bit of information. Although all formats have distinct representations for +0 and −0, the signs are significant in some circumstances, like division by zero, and not in others. In this standard, 0 and ∞ are written without a sign when the sign does

Table 1. Summary of format parameters.

Parameter	Format			
	Single	Single Extended	Double	Double Extended
p	24	≥ 32	53	≥ 64
E_{max}	+127	≥ +1023	+1023	≥ +16383
E_{min}	−126	≤ −1022	−1022	≤ −16382
exponent *bias*	+127	unspecified	+1023	unspecified
exponent width in bits	8	≥ 11	11	≥ 15
format width in bits	32	≥ 43	64	≥ 79

not matter.

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

A 1-bit sign s,

A biased exponent $e = E+bias$, and

A fraction $f = .b_1 b_2 \cdots b_{p-1}$.

The range of the unbiased exponent E shall include every integer between two values E_{min} and E_{max}, inclusive, and also two other reserved values: $E_{min}-1$ to encode ± 0 and denormalized numbers, and $E_{max}+1$ to encode $\pm\infty$ and NaNs. The foregoing parameters appear in Table 1. Each nonzero numerical value has just one encoding. The fields are interpreted as follows.

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

(1) If $e = 255$ and $f \neq 0$, then v is NaN regardless of s.

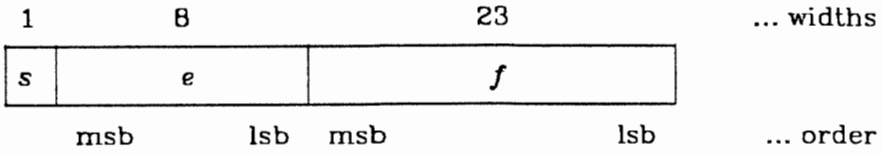
(2) If $e = 255$ and $f = 0$, then $v = (-1)^s \infty$.

(3) If $0 < e < 255$, then $v = (-1)^s 2^{e-127}(1.f)$.

(4) If $e = 0$ and $f \neq 0$, then $v = (-1)^s 2^{-126}(0.f)$ (denormalized numbers).

(5) If $e = 0$ and $f = 0$, then $v = (-1)^s 0$ (zero).

Figure 1. Single format.



"msb" means "most significant bit"
"lsb" means "least significant bit"

3.2.2. *Double.* A 64-bit double format number X is divided as shown in Figure 2. The value v of X is inferred from its constituent fields thus:

(1) If $e = 2047$ and $f \neq 0$, then v is NaN regardless of s.

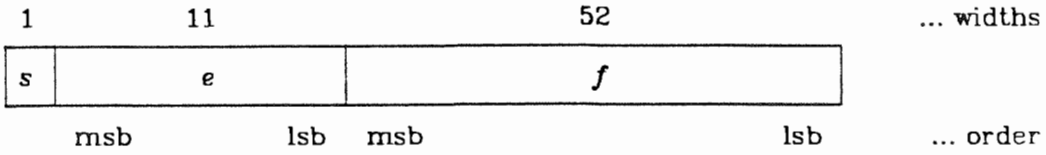
(2) If $e = 2047$ and $f = 0$, then $v = (-1)^s \infty$.

(3) If $0 < e < 2047$, then $v = (-1)^s 2^{e-1023}(1.f)$.

(4) If $e = 0$ and $f \neq 0$, then $v = (-1)^s 2^{-1022}(0.f)$ (denormalized numbers).

(5) If $e = 0$ and $f = 0$, then $v = (-1)^s 0$ (zero).

Figure 2. Double format.



3.3. Extended formats. The single extended and double extended formats

encode in an implementation-dependent way the sets of values in 3.1 subject to the constraints of Table 1. This standard allows an implementation to encode some values redundantly, provided that redundancy be transparent to the user in the following sense: an implementation either shall encode every nonzero value uniquely or it shall not distinguish redundant encodings of nonzero values. An implementation may also reserve some bit strings for purposes beyond the scope of this standard; when such a reserved bit string occurs as an operand the result is not specified by this standard.

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

3.4. Combinations of formats. All implementations conforming to this standard shall support the single format. Implementations should support the extended format corresponding to the widest basic format supported, and need not support any other extended format.[3]

4. Rounding

Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit in the destination's format while signaling the inexact exception (7.5). Except for binary↔decimal conversion (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 bit zero shall be delivered. However, an infinitely precise result with magnitude at least $2^{E_{max}}(2-2^{-p})$ shall round to ∞ with no change in sign; here E_{max} and p are determined by the destination format (§3) unless overridden by a rounding precision mode (4.3).

4.2. Directed roundings. An implementation shall also provide three user-selectable directed rounding modes: round toward $+\infty$, round toward $-\infty$, and round toward 0.

When rounding toward $+\infty$, the result shall be the format's value (possibly $+\infty$) closest to and no less than the infinitely precise result. When rounding toward $-\infty$, the result shall be the format's value (possibly $-\infty$) closest to and no greater than the infinitely precise result. When rounding toward 0, the result shall be the format's value closest to and no greater in magnitude

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

than the infinitely precise result.

4.3. Rounding precision. Normally a result is rounded to the precision of its destination. However, some systems deliver results only to double or extended destinations. On such a system the user, which may be a high-level language compiler, shall be able to specify that a result be rounded instead to single precision, though it may be stored in the double or extended format with its wider exponent range.[4] Similarly, a system that delivers results only to double extended destinations shall permit the user to specify rounding to single or double precision. Note that to meet the specifications in 4.1, the result cannot suffer more than one rounding error.

5. Operations

All conforming implementations of this standard shall provide operations to add, subtract, multiply, divide, extract the square root, find the remainder, round to integer in floating-point format, convert between different floating-point formats, convert between floating-point and integer formats, convert binary↔decimal, and compare. Whether copying without change of format is considered an operation is an implementation option. Except for binary↔decimal conversion, each of the operations shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then coerced this intermediate result to fit in the destination's format (§4 and §7). Section 6 augments the following specifications to cover ±0, ±∞, and NaN; section 7 enumerates exceptions caused by exceptional operands and exceptional results.

5.1. Arithmetic. An implementation shall provide the add, subtract, multiply, divide and remainder operations for any two operands of the same format, for each supported format; it should also provide the operations for operands of differing formats. The destination format (regardless of the rounding precision control of 4.3) shall be at least as wide as the wider operand's format. All results shall be rounded as specified in §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 $|n - x/y| = ½$, then n is even. Thus, the remainder is always exact. If $r = 0$, its sign shall be that of x. Precision control (4.3) shall not apply to the remainder operation.

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

5.3. Floating-point format conversions. It shall be possible to convert

[4]Control of rounding precision is intended to allow systems whose destinations are always double or extended to mimic, in the absence of over/underflow, the precisions of systems with single and double destinations. An implementation should not provide operations that combine double or extended operands to produce a single result, nor operations that combine double extended operands to produce a double result, with just one rounding.

floating-point numbers between all supported formats. 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 formats. It shall be possible to convert between all supported floating-point formats and all supported integer formats. Conversion to integer shall be effected by rounding as specified in §4. Conversions between floating-point integers and integer formats shall be exact unless an exception arises as specified in 7.1.

5.5. Round floating-point number to integral value. It shall be possible to round a floating-point number to an integral valued floating-point number in the same format. The rounding shall be as specified in §4, with the understanding that when rounding to nearest, if the difference between the unrounded operand and the rounded result is exactly one half, the rounded result is even.

5.6. Binary↔decimal conversion. Conversion between decimal strings in at least one format and binary floating-point numbers in all supported basic formats shall be provided for numbers throughout the ranges specified in Table 2. The integers M and N in Tables 2 and 3 are such that the decimal strings have values $\pm M \times 10^{\pm N}$. On input, trailing zeros shall be appended to or stripped from M (up to the limits specified in Table 2) in order to minimize N. When the destination is a decimal string, its least significant digit should be located by format specifications for purposes of rounding.

When the integer M lies outside the range specified in Tables 2 and 3, i.e., when $M \geq 10^9$ for single or 10^{17} for double, the implementor may, at his option, alter all significant digits after the ninth for single and seventeenth for double to other decimal digits, typically 0.

Conversions shall be correctly rounded as specified in §4 for operands lying within the ranges specified in Table 3. Otherwise, for rounding to nearest, the error in the converted result shall not exceed by more that 0.47 units in the destination's least significant digit the error that would be incurred by the rounding specifications of §4, provided that exponent over/underflow does not occur. In the directed rounding modes the error shall have the correct sign and shall not exceed 1.47 units in the last place.

Conversions shall be monotonic. That is, increasing the value of a binary floating-point number shall not decrease its value when converted to a decimal string; and increasing the value of a decimal string shall not decrease its value when converted to a binary floating-point number.

When rounding to nearest, conversion from binary to decimal and back to binary shall be the identity as long as the decimal string is carried to the maximum precision specified in Table 2, namely, 9 digits for single and 17 for double.[5]

---

[5]The properties specified for conversions are implied by error bounds that depend on the format (single or double) and the number of decimal digits involved; the 0.47 mentioned is a worst-case bound only. For a detailed discussion of these error bounds and economical conversion algorithms that exploit the extended format, see "Accurate Yet Economical Binary↔Decimal Conversions" by Jerome T. Coonen (to appear).

If decimal to binary conversion over/underflows, the response is as specified in §7. Over/underflow and NaNs and infinities encountered during binary to decimal conversion should be indicated to the user by appropriate strings. This standard says nothing about dealing with NaNs encoded in decimal strings.

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

Table 2. Decimal conversion ranges.

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

Table 3. Correctly rounded decimal conversion range.

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

5.7. Comparison. It shall be possible to compare floating-point numbers in all supported formats, even if the operands' formats differ. Comparisons are exact and never overflow nor underflow. Four mutually exclusive relations are possible: "less than", "equal", "greater than", and "unordered". The last case arises when at least one operand is NaN. Every NaN shall compare "unordered" with everything, including itself. Comparisons shall ignore the sign of zero (so +0 = −0).

The result of a comparison shall be delivered in one of two ways: 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 4, "unordered" operands are compared using one of the predicates involving "<" or ">" but not "?". (Here the symbol "?" signifies "unordered".)

Table 4 exhibits the twenty-six functionally distinct useful predicates named, in the first column, using three notations: *ad hoc*, FORTRAN-like, and mathematical. It shows how they are obtained from the four condition codes and tells which predicates cause an invalid operation exception when the relation is "unordered". The entries T and F indicate whether the predicate is true or false when the respective relation holds.

Table 4. 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

Note that predicates come in pairs, each a logical negation of the other; applying a prefix like "NOT" to negate a predicate in Table 4 reverses the true/false sense of its associated entries, but leaves the last column's entry unchanged.[6]

Implementations that provide predicates shall provide the first six predicates in Table 4 and should provide the seventh, as well as a means of logically negating predicates.

---

[6]There may appear to be two ways to write the logical negation of a predicate, one using "NOT" explicitly and the other reversing the relational operator. For example, the logical negation of (X = Y) may be written either NOT(X = Y) or (X ?<> Y); in this case both expressions are functionally equivalent to (X ≠ 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

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 floating-point format 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 format signals the invalid operation exception is the implementor's option.

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

Every operation involving one or two input NaNs, none of them signaling, shall signal no exception but, if a floating-point result is to be delivered, shall deliver as its result a quiet NaN, which should be one of the input NaNs. Note that format conversions might be unable to deliver the same NaN. Quiet NaNs do have effects similar to signaling NaNs on operations that do not deliver a floating-point result; these operations, namely comparison and conversion to a format that has no NaNs, are discussed in 5.4, 5.6, 5.7, and 7.1.

6.3. The sign bit. This standard does not interpret the sign of a NaN. Otherwise the sign of a product or quotient is the Exclusive Or of the operands' 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 addends' 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

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.

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

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

(2) Addition or subtraction: magnitude subtraction of infinities 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 a binary floating-point number to an integer or decimal format when overflow, infinity, or NaN precludes a faithful representation in that format and this cannot otherwise be signaled; and

(8) Comparison via predicates involving "<" or ">", without "?", when the operands are "unordered" (5.7, Table 4).

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

7.3. Overflow. The overflow exception shall be signaled whenever the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result (§4) were the exponent range unbounded. The result, when no trap occurs, shall be determined by the rounding mode and the sign of the intermediate result as follows:

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

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

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

(4) Round toward $+\infty$ carries negative overflows to the format's most negative finite number, and carries positive overflows to $+\infty$.

Trapped overflows on all operations except conversions shall deliver to the trap handler the result obtained by dividing the infinitely precise result by 2^a and then rounding. The bias adjust a is 192 in the single, 1536 in the double, and $3\times2^{n-2}$ in the extended format, where n is the number of bits in the exponent field.[7] Trapped overflow on conversion from a binary floating-point format shall deliver to the trap handler a result in that or a wider format, possibly with the exponent bias adjusted, but rounded to the destination's precision. Trapped overflow on decimal to binary conversion shall deliver to the trap handler a result in the widest supported format, possibly with the exponent bias adjusted, but rounded to the destination's precision; when the result lies too far outside the range for the bias to be adjusted, a quiet NaN shall be delivered instead.

7.4. Underflow. Two correlated events contribute to underflow. One is the creation of a tiny nonzero result between $\pm2^{E_{min}}$ which, because it is so tiny, may cause some other exception later such as overflow upon division. The other is extraordinary loss of accuracy during the approximation of such tiny numbers by denormalized numbers. The implementor may choose how these events are detected, but shall detect these events in the same way for all operations. Tininess may be detected either

(1) "After rounding": when a nonzero result computed as though the exponent range were unbounded would lie strictly between $\pm2^{E_{min}}$;

or

(2) "Before rounding": when a nonzero result computed as though both the exponent range and the precision were unbounded would lie strictly between $\pm2^{E_{min}}$.

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, denormalized or $\pm2^{E_{min}}$. 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 2^a and then rounding. The bias adjust a is 192 in the single, 1536 in the double, and $3\times2^{n-2}$ in the extended format, where n is

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

the number of bits in the exponent field.[8] Trapped underflows on conversion shall be handled analogously to the handling of overflows on conversion.

7.5. Inexact. If the rounded result of an operation is not exact or if it overflows without an overflow trap, then the inexact exception shall be signaled. The rounded or overflowed result shall be delivered to the destination or, if an inexact trap occurs, to the trap handler.

8. Traps

A user should be able to request a trap on any of the five exceptions by specifying a handler for it. He should be able to request that an existing handler be disabled, saved or restored. He should also be able to determine whether a specific trap handler for a designated exception has been enabled. When an exception whose trap is disabled is signaled, it shall be handled in the manner specified in §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 format;

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

---

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

Appendix: Recommended functions and predicates

This appendix is not part of IEEE Standard 754
for Binary Floating-Point Arithmetic,
but is included for information only.

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

Some functions below, like the copy operation $y := x$ without change of format, may at the implementor's option be treated as nonarithmetic operations which do not signal the invalid operation exception for signaling NaNs; the functions in question are (1), (2), (6), and (7).

(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 sign bit to distinguish signaling NaNs from quiet NaNs.

(3) scalb(y,N) returns $y \times 2^N$ for integral values N without computing 2^N.

(4) logb(x) returns the unbiased exponent of x, a signed integer in the format of x, except that logb(NaN) is a NaN, logb(∞) is $+\infty$, 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 2; it is less than 1 only when x is denormalized.

(5) nextafter(x,y) returns the next representable neighbor of x in the direction toward y. The following special cases arise: if $x=y$, then the result is x without any exception being signaled; otherwise, if either x or y is a quiet NaN, then the result is one or the other of the input NaNs. Overflow is signaled when x is finite but nextafter(x,y) is infinite; underflow is signaled when nextafter(x,y) lies strictly between $\pm 2^{E_{min}}$; 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($x=y$) (Table 4).

(9) unordered(x,y), or $x?y$, returns the value TRUE if x is unordered with y, and returns FALSE otherwise (Table 4).

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

APPENDIX B

Test Vectors for P754 Arithmetic – Version 2.0

The initial version of this test data base for the proposed IEEE 754 binary floating-point standard (draft 8.0) was developed for Zilog, Inc. and was donated to the floating-point working group for dissemination. Errors in or additions to the distributed data base should be reported to the agency of distribution, with copies to Zilog, Inc., 1315 Dell Avenue, Campbell, CA, 95008.

There are sixteen files of test vectors, for the operations add (+), subtract (–), multiply (*), divide (/), square root (V), compare (C), remainder (%), round to integer (I), nextafter (N), absolute value (A), negate (~), copysign (@), scalb (S), logb (L), and fraction part (F).

```
! First some easy integer cases.
2+ ALL 1 1 OK 2
2+ ALL 1 2 OK 3
2+ ALL 2 1 OK 3
2+ ALL 2 2 OK 4
2+ =0> 2 -2 OK 0
2+ < 2 -2 OK -0
2+ =0> 5 -5 OK 0
2+ < 5 -5 OK -0
2+ ALL 1 7 OK 8
2+ ALL 5 -1 OK 4
2+ ALL 2 -5 OK -3
2+ ALL 5 -0 OK 5
2+ ALL 5 +0 OK 5
! Infinity vs Infinity.
2+ ALL H H OK H ok - affine sum
2+ ALL -H -H OK -H
2+ ALL -H H i Q different signs
2+ ALL H -H i Q
! Infinity vs huge.
2+ ALL H Hm1 OK H
2+ ALL H -Hm1 OK H
2+ ALL -H Hm1 OK -H
2+ ALL -H -Hm1 OK -H
2+ ALL Hm1 H OK H
2+ ALL Hm1 -H OK -H
2+ ALL -Hm1 H OK H
2+ ALL -Hm1 -H OK -H
! Infinity vs 0.
2+ ALL H 0 OK H
2+ ALL H -0 OK H
2+ ALL -H 0 OK -H
2+ ALL -H -0 OK -H
2+ ALL 0 H OK H
2+ ALL -0 H OK H
2+ ALL 0 -H OK -H
2+ ALL -0 -H OK -H
! Infinity vs denormalized.
2+ ALL H Ed1 OK H
2+ ALL -H Ed1 OK -H
2+ ALL H -Ed1 OK H
2+ ALL -H -Ed1 OK -H
2+ ALL 0i3 H OK H
2+ ALL 0i3 -H OK -H
2+ ALL -0i3 H OK H
2+ ALL -0i3 -H OK -H
! Zero vs finite -- watch that sign of 0
! is meaningless.
2+ ALL 0 Hm1 OK Hm1
2+ ALL -0 Hm1 OK Hm1
2+ ALL -Hm1 0 OK -Hm1
2+ ALL -Hm1 -0 OK -Hm1
2+ ALL 1 -0 OK 1
2+ ALL -1 -0 OK -1
2+ ALL 0 1 OK 1
2+ ALL -0 -1 OK -1
! Zero vs denormalized -- underflows.
2+ ALL 0 Ed1 OK Ed1
2+ ALL -0 Ed1 OK Ed1
2+ ALL 0 -Ed1 OK -Ed1
2+ ALL -0 -Ed1 OK -Ed1
2+ ALL 0i3 0 OK 0i3
2+ ALL 0i3 -0 OK 0i3
2+ ALL -0i3 0 OK -0i3
2+ ALL -0i3 -0 OK -0i3
! Zero vs tiny -- just in case.
2+ ALL -0 -E OK -E
2+ ALL E 0 OK E
2+ ALL 0 -E OK -E
2+ ALL -E 0 OK -E
! Zero vs Zero -- watch signs and
! rounding modes.
2+ =0> 0 -0 OK 0
2+ =0> -0 0 OK 0
2+ < 0 -0 OK -0
2+ < -0 0 OK -0
2+ ALL 0 0 OK 0
2+ ALL -0 -0 OK -0
! Double a number -- may overflow so
! watch rounding mode.
2+ => Hm1 Hm1 xo H
2+ 0< Hm1 Hm1 xo Hd1
2+ =< -Hm1 -Hm1 xo -H
2+ 0> -Hm1 -Hm1 xo -Hd1
2+ ALL Hm1d2 Hm1d2 OK Hd2
2+ ALL -Hm1d2 -Hm1d2 OK -Hd2
2+ => Hd2 Hd2 xo H
2+ 0< Hd2 Hd2 xo Hd1
2+ =< -Hd2 -Hd2 xo -H
2+ 0> -Hd2 -Hd2 xo -Hd1
! Double an innocent number.
2+ ALL 1 1 OK 2
2+ ALL 3 3 OK 6
2+ ALL E E OK Ep1
2+ ALL Hm2 Hm2 OK Hm1
! Double a tiny number -- may underflow.
2+ ALL Ed1 Ed1 OK Ep1d2
2+ ALL -Ed1 -Ed1 OK -Ep1d2
2+ ALL 0i4 0i4 OK 0i8
2+ ALL -0i4 -0i4 OK -0i8
2+ ALL 0i1 0i1 OK 0i2
2+ ALL -0i1 -0i1 OK -0i2
! Cancellation to 0 -- to plus 0.
2+ =0> Hm1 -Hm1 OK 0
2+ =0> -Hm1d2 Hm1d2 OK 0
2+ =0> 1 -1 OK 0
2+ =0> -3 3 OK 0
2+ =0> E -E OK 0
2+ =0> -E E OK 0
2+ =0> Ed4 -Ed4 OK 0
2+ =0> -Ed1 Ed1 OK 0 no underflow
2+ =0> 0i1 -0i1 OK 0
2+ =0> -0i1 0i1 OK 0
2+ =0> Hd1 -Hd1 OK 0
! Cancellation to 0 -- to minus 0.
2+ < Hm1 -Hm1 OK -0
2+ < -Hm1d2 Hm1d2 OK -0
2+ < 1 -1 OK -0
2+ < -3 3 OK -0
2+ < E -E OK -0
2+ < -E E OK -0
2+ < Ed4 -Ed4 OK -0
2+ < -Ed1 Ed1 OK -0 no underflow
2+ < 0i1 -0i1 OK -0
2+ < -0i1 0i1 OK -0
2+ < Hd1 -Hd1 OK -0
! Cancel forcing normalization of LSB
! (no rounding errors). Difference is in
```

```
! last place of larger number.
! Medium numbers...
2+ ALL 1i1 -1 OK 1u1
2+ ALL -1i1 1 OK -1u1
2+ ALL 1i1 -1i2 OK -1u1
2+ ALL -1i1 1i2 OK 1u1
2+ ALL 2 -2i1 OK -2u1
2+ ALL -2 2i1 OK 2u1
2+ ALL 2i4 -2i3 OK 2u1
2+ ALL -2i4 2i3 OK -2u1
2+ ALL 4d1 -4d2 OK 3u1
2+ ALL -4d1 4d2 OK -3u1
2+ ALL 2d4 -2d3 OK -1u1
2+ ALL -2d4 2d3 OK 1u1
! Huge numbers...
2+ ALL Hm1i1 -Hm1 OK Hm1u1
2+ ALL -Hm1i1 Hm1 OK -Hm1u1
2+ ALL Hm1i1 -Hm1i2 OK -Hm1u1
2+ ALL -Hm1i1 Hm1i2 OK Hm1u1
2+ ALL Hm2 -Hm2i1 OK -Hm2u1
2+ ALL -Hm2 Hm2i1 OK Hm2u1
2+ ALL Hm2i4 -Hm2i3 OK Hm2u1
2+ ALL -Hm2i4 Hm2i3 OK -Hm2u1
2+ ALL Hm2d1 -Hm2d2 OK Hm3u1
2+ ALL -Hm2d1 Hm2d2 OK -Hm3u1
2+ ALL -Hd2 Hd1 OK Hd1u1
2+ ALL Hd2 -Hd1 OK -Hd1u1
! Tiny numbers...
2+ ALL -Ei1 E OK -Eu1
2+ ALL Ei1 -E OK Eu1
2+ ALL -Ed1 E OK Eu1
2+ ALL Ed1 -E OK -Eu1
2+ ALL Ei1 -Ei2 OK -Eu1
2+ ALL -Ei1 Ei2 OK Eu1
2+ ALL Ed1 -Ed2 OK Eu1
2+ ALL -Ed1 Ed2 OK -Eu1
2+ ALL Ed3 -Ed2 OK -Eu1
2+ ALL -Ed3 Ed2 OK Eu1
2+ ALL 0i2 -0i1 OK Eu1
2+ ALL -0i2 0i1 OK -Eu1
2+ ALL 0i3 -0i2 OK Eu1
2+ ALL -0i3 0i2 OK -Eu1
! Normalize from round bit -- set up
! tests so that operands have
! exponents differing by 1 unit.
! Medium numbers...
2+ ALL 2 -2d1 OK 1u1
2+ ALL -2 2d1 OK -1u1
2+ ALL -2d1 2 OK 1u1
2+ ALL 2d1 -2 OK -1u1
2+ ALL 4i1 -4d1 OK 3u3
2+ ALL -4i1 4d1 OK -3u3
2+ ALL 4d1 -4i2 OK -3u5
2+ ALL -4d1 4i2 OK 3u5
2+ ALL 2i1 -1i1 OK 1i1
2+ ALL -2i1 1i1 OK -1i1
2+ ALL 2i2 -1i1 OK 1i3
2+ ALL -2i2 1i1 OK -1i3
2+ ALL 2i2 -1i3 OK 1i1
2+ ALL -2i2 1i3 OK -1i1
! Huge numbers...
2+ ALL Hm2 -Hm2d1 OK Hm3u1
2+ ALL -Hm2 Hm2d1 OK -Hm3u1
2+ ALL -Hm1d1 Hm1 OK Hm2u1
2+ ALL Hm1d1 -Hm1 OK -Hm2u1
2+ ALL Hm4i1 -Hm4d1 OK Hm5u3
2+ ALL -Hm4i1 Hm4d1 OK -Hm5u3
2+ ALL Hm2d1 -Hm2i2 OK -Hm3u5
2+ ALL -Hm2d1 Hm2i2 OK Hm3u5
2+ ALL Hm2i1 -Hm1i1 OK -Hm2i1
2+ ALL -Hm2i1 Hm1i1 OK Hm2i1
2+ ALL Hm1i2 -Hm2i1 OK Hm2i3
2+ ALL -Hm1i2 Hm2i1 OK -Hm2i3
2+ ALL Hm2i2 -Hm3i3 OK Hm3i1
2+ ALL -Hm2i2 Hm3i3 OK -Hm3i1
! Tiny numbers...
2+ ALL Ep1 -Ep1d1 OK Eu1
2+ ALL -Ep1 Ep1d1 OK -Eu1
2+ ALL -Ep1d1 Ep1 OK Eu1
2+ ALL Ep1d1 -Ep1 OK -Eu1
2+ ALL Ep1i1 -Ep1d1 OK Eu3
2+ ALL -Ep1i1 Ep1d1 OK -Eu3
2+ ALL Ep2 -Ep2d1 OK Eu2
2+ ALL -Ep2 Ep2d1 OK -Eu2
2+ ALL -Ep2d1 Ep2 OK Eu2
2+ ALL Ep2d1 -Ep2 OK -Eu2
2+ ALL Ep2i1 -Ep2d1 OK Eu6
2+ ALL -Ep2i1 Ep2d1 OK -Eu6
2+ ALL Ep1d1 -Ep1i2 OK -Eu5
2+ ALL -Ep1d1 Ep1i2 OK Eu5
2+ ALL Ep1d1 -Ep1i4 OK -Eu9
2+ ALL -Ep1d1 Ep1i4 OK Eu9
2+ ALL Ep1i1 -Ei1 OK Ei1
2+ ALL -Ep1i1 Ei1 OK -Ei1
2+ ALL Ep1i2 -Ei1 OK Ei3
2+ ALL -Ep1i2 Ei1 OK -Ei3
2+ ALL Ep2i2 -Ep1i3 OK Ep1i1
2+ ALL -Ep2i2 Ep1i3 OK -Ep1i1
! Add magnitude:
! cases where one operand is off in sticky --
! rounding perhaps to an overflow.
! Huge vs medium.
2+ =0< Hm1 1 x Hm1
2+ > Hm1 1 x Hm1i1
2+ =0> -Hm1 -1 x -Hm1
2+ < -Hm1 -1 x -Hm1i1
2+ =0< Hm1d1 1 x Hm1d1
2+ > Hm1d1 1 x Hm1
2+ =0> -Hm1d1 -1 x -Hm1d1
2+ < -Hm1d1 -1 x -Hm1
2+ =0< Hd1 1 x Hd1
2+ > Hd1 1 xo H signal overflow
2+ =0> -Hd1 -1 x -Hd1
2+ < -Hd1 -1 xo -H
2+ =0< Hd2 1 x Hd2
2+ > Hd2 1 x Hd1
2+ =0> -Hd2 -1 x -Hd2
2+ < -Hd2 -1 x -Hd1
! Huge vs denormal.
2+ =0< 0i1 Hm1 x Hm1
2+ > 0i1 Hm1 x Hm1i1
2+ =0> -0i1 -Hm1 x -Hm1
2+ < -0i1 -Hm1 x -Hm1i1
2+ =0< 0i1 Hm1d1 x Hm1d1
2+ > 0i1 Hm1d1 x Hm1
2+ =0> -0i1 -Hm1d1 x -Hm1d1
2+ < -0i1 -Hm1d1 x -Hm1
2+ =0< 0i1 Hd1 x Hd1
```

```
2+ > 0i1 Hd1 xo H signal overflow
2+ =0> -0i1 -Hd1 x -Hd1
2+ < -0i1 -Hd1 xo -H
2+ =0< 0i1 Hd2 x Hd2
2+ > 0i1 Hd2 x Hd1
2+ =0> -0i1 -Hd2 x -Hd2
2+ < -0i1 -Hd2 x -Hd1
! Medium vs denormal.
2+ =0< 0i1 1 x 1
2+ > 0i1 1 x 1i1
2+ =0> -0i1 -1 x -1
2+ < -0i1 -1 x -1i1
2+ =0< 0i1 1d1 x 1d1
2+ > 0i1 1d1 x 1
2+ =0> -0i1 -1d1 x -1d1
2+ < -0i1 -1d1 x -1
2+ =0< 0i1 2d1 x 2d1
2+ > 0i1 2d1 x 2
2+ =0> -0i1 -2d1 x -2d1
2+ < -0i1 -2d1 x -2
2+ =0< 0i1 2d2 x 2d2
2+ > 0i1 2d2 x 2d1
2+ =0> -0i1 -2d2 x -2d2
2+ < -0i1 -2d2 x -2d1
!
! Magnitude subtract when an operand is
! in the sticky bit. The interesting cases
! will arise when directed rounding
! forces a nonzero cancellation.
! Huge and medium.
2+ => Hm1 -1 x Hm1
2+ 0< Hm1 -1 x Hm1d1
2+ =< -Hm1 1 x -Hm1
2+ 0> -Hm1 1 x -Hm1d1
2+ => Hm1d1 -1 x Hm1d1
2+ 0< Hm1d1 -1 x Hm1d2
2+ =< -Hm1d1 1 x -Hm1d1
2+ 0> -Hm1d1 1 x -Hm1d2
2+ => Hd1 -1 x Hd1
2+ 0< Hd1 -1 x Hd2
2+ =< -Hd1 1 x -Hd1
2+ 0> -Hd1 1 x -Hd2
2+ => Hd2 -1 x Hd2
2+ 0< Hd2 -1 x Hd3
2+ =< -Hd2 1 x -Hd2
2+ 0> -Hd2 1 x -Hd3
! Huge and tiny.
2+ => Hd1 -0i1 x Hd1
2+ 0< Hd1 -0i1 x Hd2
2+ =< -Hd1 0i1 x -Hd1
2+ 0> -Hd1 0i1 x -Hd2
2+ => -0i3 Hm1 x Hm1
2+ 0< -0i3 Hm1 x Hm1d1
2+ =< 0i3 -Hm1 x -Hm1
2+ 0> 0i3 -Hm1 x -Hm1d1
! Medium and tiny.
2+ => 1d1 -0i1 x 1d1
2+ 0< 1d1 -0i1 x 1d2
2+ =< -2d1 0i1 x -2d1
2+ 0> -2d1 0i1 x -2d2
2+ => -0i3 3 x 3
2+ 0< -0i3 3 x 3d1
2+ =< 0i3 -5 x -5
2+ 0> 0i3 -5 x -5d1
!
! Add magnitude with difference in LSB
! so, except for denorms, round bit
! is crucial. Half-way cases arise.
! Medium cases.
2+ =0< 1i1 1 x 2
2+ > 1i1 1 x 2i1
2+ =0> -1i1 -1 x -2
2+ < -1i1 -1 x -2i1
2+ =0> -2 -2i1 x -4
2+ < -2 -2i1 x -4i1
2+ =0< 2 2i1 x 4
2+ > 2 2i1 x 4i1
2+ => 1 1i3 x 2i2
2+ 0< 1 1i3 x 2i1
2+ =< -1 -1i3 x -2i2
2+ 0> -1 -1i3 x -2i1
2+ =< -2i1 -2i2 x -4i2
2+ 0> -2i1 -2i2 x -4i1
2+ => 2i1 2i2 x 4i2
2+ 0< 2i1 2i2 x 4i1
! Huge cases.
2+ => Hd2 Hd1 xo H
2+ 0< Hd2 Hd1 xo Hd1
2+ =< -Hd2 -Hd1 xo -H
2+ 0> -Hd2 -Hd1 xo -Hd1
2+ => Hm1d1 Hm1 xo H
2+ 0< Hm1d1 Hm1 x Hd1
2+ =< -Hm1d1 -Hm1 xo -H
2+ 0> -Hm1d1 -Hm1 x -Hd1
2+ => Hm1i1 Hm1 xo H
2+ 0< Hm1i1 Hm1 xo Hd1
2+ =< -Hm1i1 -Hm1 xo -H
2+ 0> -Hm1i1 -Hm1 xo -Hd1
2+ =0< Hm2i1 Hm2 x Hm1
2+ > Hm2i1 Hm2 x Hm1i1
2+ =0> -Hm2i1 -Hm2 x -Hm1
2+ < -Hm2i1 -Hm2 x -Hm1i1
2+ =0< Hm1d2 Hm1d1 x Hd2
2+ > Hm1d2 Hm1d1 x Hd1
2+ =0> -Hm1d2 -Hm1d1 x -Hd2
2+ < -Hm1d2 -Hm1d1 x -Hd1
! Check rounding.
2+ > 2 1u1 x 2i1
2+ =0< 2 1u1 x 2
2+ => 2i1 1u1 x 2i2
2+ 0< 2i1 1u1 x 2i1
2+ => 4d1 1u1 x 4
2+ 0< 4d1 1u1 x 4d1
2+ > 4d1 1u1d1 x 4
2+ 0=< 4d1 1u1d1 x 4d1
2+ =< -4d1 -1u1 x -4
2+ 0> -4d1 -1u1 x -4d1
2+ < -4d1 -1u1d1 x -4
2+ 0=> -4d1 -1u1d1 x -4d1
!NAN operands.
2+ ALL Q 0 OK Q
2+ ALL Q -0 OK Q
2+ ALL 0 Q OK Q
2+ ALL -0 Q OK Q
2+ ALL Q 1 OK Q
2+ ALL Q -1 OK Q
2+ ALL 1 Q OK Q
2+ ALL -1 Q OK Q
```

```
2+ ALL Ed1 Q OK Q
2+ ALL -Ed1 Q OK Q
2+ ALL Q Ed1 OK Q
2+ ALL Q -Ed1 OK Q
2+ ALL Q Oi1 OK Q
2+ ALL Q -Oi1 OK Q
2+ ALL Oi1 Q OK Q
2+ ALL -Oi1 Q OK Q
2+ ALL Q Hd1 OK Q
2+ ALL Q -Hd1 OK Q
2+ ALL Hd1 Q OK Q
2+ ALL -Hd1 Q OK Q
2+ ALL Q H OK Q
2+ ALL Q -H OK Q
2+ ALL H Q OK Q
2+ ALL -H Q OK Q
2+ ALL Q Q OK Q
2+ ALL S 0 i Q
2+ ALL S -0 i Q
2+ ALL 0 S i Q
2+ ALL -0 S i Q
2+ ALL S 1 i Q
2+ ALL S -1 i Q
2+ ALL 1 S i Q
2+ ALL -1 S i Q
2+ ALL Ed1 S i Q
2+ ALL -Ed1 S i Q
2+ ALL S Ed1 i Q
2+ ALL S -Ed1 i Q
2+ ALL S Oi1 i Q
2+ ALL S -Oi1 i Q
2+ ALL Oi1 S i Q
2+ ALL -Oi1 S i Q
2+ ALL S Hd1 i Q
2+ ALL S -Hd1 i Q
2+ ALL Hd1 S i Q
2+ ALL -Hd1 S i Q
2+ ALL S H i Q
2+ ALL S -H i Q
2+ ALL H S i Q
2+ ALL -H S i Q
2+ ALL Q S i Q
2+ ALL S Q i Q
2+ ALL S S i Q
```

```
! First some easy integer cases.
2- ALL 1 -1 OK 2
2- ALL 1 -2 OK 3
2- ALL 2 -1 OK 3
2- ALL 2 -2 OK 4
2- =0> 2 2 OK 0
2- < 2 2 OK -0
2- =0> 5 5 OK 0
2- < 5 5 OK -0
2- ALL 1 -7 OK 8
2- ALL 5 1 OK 4
2- ALL 2 5 OK -3
2- ALL 5 0 OK 5
2- ALL 5 -0 OK 5
! Infinity vs Infinity.
2- ALL H -H OK H  ok - affine sum
2- ALL -H H OK -H
2- ALL -H -H i Q  different signs
2- ALL H H i Q
! Infinity vs huge.
2- ALL H -Hm1 OK H
2- ALL H Hm1 OK H
2- ALL -H -Hm1 OK -H
2- ALL -H Hm1 OK -H
2- ALL Hm1 -H OK H
2- ALL Hm1 H OK -H
2- ALL -Hm1 -H OK H
2- ALL -Hm1 H OK -H
! Infinity vs 0.
2- ALL H -0 OK H
2- ALL H 0 OK H
2- ALL -H -0 OK -H
2- ALL -H 0 OK -H
2- ALL 0 -H OK H
2- ALL -0 -H OK H
2- ALL 0 H OK -H
2- ALL -0 H OK -H
! Infinity vs denormalized.
2- ALL H -Ed1 OK H
2- ALL -H -Ed1 OK -H
2- ALL H Ed1 OK H
2- ALL -H Ed1 OK -H
2- ALL 0i3 -H OK H
2- ALL 0i3 H OK -H
2- ALL -0i3 -H OK H
2- ALL -0i3 H OK -H
! Zero vs finite -- watch that sign of
! 0 is meaningless.
2- ALL 0 -Hm1 OK Hm1
2- ALL -0 -Hm1 OK Hm1
2- ALL -Hm1 -0 OK -Hm1
2- ALL -Hm1 0 OK -Hm1
2- ALL 1 0 OK 1
2- ALL -1 0 OK -1
2- ALL 0 -1 OK 1
2- ALL -0 1 OK -1
! Zero vs denormalized -- underflows.
2- ALL 0 -Ed1 OK Ed1
2- ALL -0 -Ed1 OK Ed1
2- ALL 0 Ed1 OK -Ed1
2- ALL -0 Ed1 OK -Ed1
2- ALL 0i3 -0 OK 0i3
2- ALL 0i3 0 OK 0i3
2- ALL -0i3 -0 OK -0i3
2- ALL -0i3 0 OK -0i3
! Zero vs tiny -- just in case.
2- ALL -0 E OK -E
2- ALL E -0 OK E
2- ALL 0 E OK -E
2- ALL -E -0 OK -E
! Zero vs Zero -- watch signs and
! rounding modes.
2- =0> 0 0 OK 0
2- =0> -0 -0 OK 0
2- < 0 0 OK -0
2- < -0 -0 OK -0
2- ALL 0 -0 OK 0
2- ALL -0 0 OK -0
! Double a number -- may overflow so
! watch rounding mode.
2- => Hm1 -Hm1 xo H
2- 0< Hm1 -Hm1 xo Hd1
2- =< -Hm1 Hm1 xo -H
2- 0> -Hm1 Hm1 xo -Hd1
2- ALL Hm1d2 -Hm1d2 OK Hd2
2- ALL -Hm1d2 Hm1d2 OK -Hd2
2- => Hd2 -Hd2 xo H
2- 0< Hd2 -Hd2 xo Hd1
2- =< -Hd2 Hd2 xo -H
2- 0> -Hd2 Hd2 xo -Hd1
! Double an innocent number.
2- ALL 1 -1 OK 2
2- ALL 3 -3 OK 6
2- ALL E -E OK Ep1
2- ALL Hm2 -Hm2 OK Hm1
! Double a tiny number -- may underflow.
2- ALL Ed1 -Ed1 OK Ep1d2
2- ALL -Ed1 Ed1 OK -Ep1d2
2- ALL 0i4 -0i4 OK 0i8
2- ALL -0i4 0i4 OK -0i8
2- ALL 0i1 -0i1 OK 0i2
2- ALL -0i1 0i1 OK -0i2
! Cancellation to 0 -- to plus 0.
2- =0> Hm1 Hm1 OK 0
2- =0> -Hm1d2 -Hm1d2 OK 0
2- =0> 1 1 OK 0
2- =0> -3 -3 OK 0
2- =0> E E OK 0
2- =0> -E -E OK 0
2- =0> Ed4 Ed4 OK 0
2- =0> -Ed1 -Ed1 OK 0 no underflow
2- =0> 0i1 0i1 OK 0
2- =0> -0i1 -0i1 OK 0
2- =0> Hd1 Hd1 OK 0
! Cancellation to 0 -- to minus 0.
2- < Hm1 Hm1 OK -0
2- < -Hm1d2 -Hm1d2 OK -0
2- < 1 1 OK -0
2- < -3 -3 OK -0
2- < E E OK -0
2- < -E -E OK -0
2- < Ed4 Ed4 OK -0
2- < -Ed1 -Ed1 OK -0 no underflow
2- < 0i1 0i1 OK -0
2- < -0i1 -0i1 OK -0
2- < Hd1 Hd1 OK -0
! Cancel forcing normalization of LSB
! (no rounding errors).  Difference is in
```

```
! last place of larger number.
! Medium numbers...
2- ALL 1i1 1 OK 1u1
2- ALL -1i1 -1 OK -1u1
2- ALL 1i1 1i2 OK -1u1
2- ALL -1i1 -1i2 OK 1u1
2- ALL 2 2i1 OK -2u1
2- ALL -2 -2i1 OK 2u1
2- ALL 2i4 2i3 OK 2u1
2- ALL -2i4 -2i3 OK -2u1
2- ALL 4d1 4d2 OK 3u1
2- ALL -4d1 -4d2 OK -3u1
2- ALL 2d4 2d3 OK -1u1
2- ALL -2d4 -2d3 OK 1u1
! Huge numbers...
2- ALL Hm1i1 Hm1 OK Hm1u1
2- ALL -Hm1i1 -Hm1 OK -Hm1u1
2- ALL Hm1i1 Hm1i2 OK -Hm1u1
2- ALL -Hm1i1 -Hm1i2 OK Hm1u1
2- ALL Hm2 Hm2i1 OK -Hm2u1
2- ALL -Hm2 -Hm2i1 OK Hm2u1
2- ALL Hm2i4 Hm2i3 OK Hm2u1
2- ALL -Hm2i4 -Hm2i3 OK -Hm2u1
2- ALL Hm2d1 Hm2d2 OK Hm3u1
2- ALL -Hm2d1 -Hm2d2 OK -Hm3u1
2- ALL -Hd2 -Hd1 OK Hd1u1
2- ALL Hd2 Hd1 OK -Hd1u1
! Tiny numbers...
2- ALL -Ei1 -E OK -Eu1
2- ALL Ei1 E OK Eu1
2- ALL -Ed1 -E OK Eu1
2- ALL Ed1 E OK -Eu1
2- ALL Ei1 Ei2 OK -Eu1
2- ALL -Ei1 -Ei2 OK Eu1
2- ALL Ed1 Ed2 OK Eu1
2- ALL -Ed1 -Ed2 OK -Eu1
2- ALL Ed3 Ed2 OK -Eu1
2- ALL -Ed3 -Ed2 OK Eu1
2- ALL 0i2 0i1 OK Eu1
2- ALL -0i2 -0i1 OK -Eu1
2- ALL 0i3 0i2 OK Eu1
2- ALL -0i3 -0i2 OK -Eu1
! Normalize from round bit -- set up tests
! so that operands have
! exponents differing by 1 unit.
! Medium numbers...
2- ALL 2 2d1 OK 1u1
2- ALL -2 -2d1 OK -1u1
2- ALL -2d1 -2 OK 1u1
2- ALL 2d1 2 OK -1u1
2- ALL 4i1 4d1 OK 3u3
2- ALL -4i1 -4d1 OK -3u3
2- ALL 4d1 4i2 OK -3u5
2- ALL -4d1 -4i2 OK 3u5
2- ALL 2i1 1i1 OK 1i1
2- ALL -2i1 -1i1 OK -1i1
2- ALL 2i2 1i1 OK 1i3
2- ALL -2i2 -1i1 OK -1i3
2- ALL 2i2 1i3 OK 1i1
2- ALL -2i2 -1i3 OK -1i1
! Huge numbers...
2- ALL Hm2 Hm2d1 OK Hm3u1
2- ALL -Hm2 -Hm2d1 OK -Hm3u1
2- ALL -Hm1d1 -Hm1 OK Hm2u1
2- ALL Hm1d1 Hm1 OK -Hm2u1
2- ALL Hm4i1 Hm4d1 OK Hm5u3
2- ALL -Hm4i1 -Hm4d1 OK -Hm5u3
2- ALL Hm2d1 Hm2i2 OK -Hm3u5
2- ALL -Hm2d1 -Hm2i2 OK Hm3u5
2- ALL Hm2i1 Hm1i1 OK -Hm2i1
2- ALL -Hm2i1 -Hm1i1 OK Hm2i1
2- ALL Hm1i2 Hm2i1 OK Hm2i3
2- ALL -Hm1i2 -Hm2i1 OK -Hm2i3
2- ALL Hm2i2 Hm3i3 OK Hm3i1
2- ALL -Hm2i2 -Hm3i3 OK -Hm3i1
! Tiny numbers...
2- ALL Ep1 Ep1d1 OK Eu1
2- ALL -Ep1 -Ep1d1 OK -Eu1
2- ALL -Ep1d1 -Ep1 OK Eu1
2- ALL Ep1d1 Ep1 OK -Eu1
2- ALL Ep1i1 Ep1d1 OK Eu3
2- ALL -Ep1i1 -Ep1d1 OK -Eu3
2- ALL Ep2 Ep2d1 OK Eu2
2- ALL -Ep2 -Ep2d1 OK -Eu2
2- ALL -Ep2d1 -Ep2 OK Eu2
2- ALL Ep2d1 Ep2 OK -Eu2
2- ALL Ep2i1 Ep2d1 OK Eu6
2- ALL -Ep2i1 -Ep2d1 OK -Eu6
2- ALL Ep1d1 Ep1i2 OK -Eu5
2- ALL -Ep1d1 -Ep1i2 OK Eu5
2- ALL Ep1d1 Ep1i4 OK -Eu9
2- ALL -Ep1d1 -Ep1i4 OK Eu9
2- ALL Ep1i1 Ei1 OK Ei1
2- ALL -Ep1i1 -Ei1 OK -Ei1
2- ALL Ep1i2 Ei1 OK Ei3
2- ALL -Ep1i2 -Ei1 OK -Ei3
2- ALL Ep2i2 Ep1i3 OK Ep1i1
2- ALL -Ep2i2 -Ep1i3 OK -Ep1i1
! Add magnitude:
! cases where one operand is off in sticky --
! rounding perhaps to an overflow.
! Huge vs medium.
2- =0< Hm1 -1 x Hm1
2- > Hm1 -1 x Hm1i1
2- =0> -Hm1 1 x -Hm1
2- < -Hm1 1 x -Hm1i1
2- =0< Hm1d1 -1 x Hm1d1
2- > Hm1d1 -1 x Hm1
2- =0> -Hm1d1 1 x -Hm1d1
2- < -Hm1d1 1 x -Hm1
2- =0< Hd1 -1 x Hd1
2- > Hd1 -1 xo H signal overflow
2- =0> -Hd1 1 x -Hd1
2- < -Hd1 1 xo -H
2- =0< Hd2 -1 x Hd2
2- > Hd2 -1 x Hd1
2- =0> -Hd2 1 x -Hd2
2- < -Hd2 1 x -Hd1
! Huge vs denormal.
2- =0< 0i1 -Hm1 x Hm1
2- > 0i1 -Hm1 x Hm1i1
2- =0> -0i1 Hm1 x -Hm1
2- < -0i1 Hm1 x -Hm1i1
2- =0< 0i1 -Hm1d1 x Hm1d1
2- > 0i1 -Hm1d1 x Hm1
2- =0> -0i1 Hm1d1 x -Hm1d1
2- < -0i1 Hm1d1 x -Hm1
2- =0< 0i1 -Hd1 x Hd1
```

```
2- > 0i1 -Hd1 xo H signal overflow
2- =0> -0i1 Hd1 x -Hd1
2- < -0i1 Hd1 xo -H
2- =0< 0i1 -Hd2 x Hd2
2- > 0i1 -Hd2 x Hd1
2- =0> -0i1 Hd2 x -Hd2
2- < -0i1 Hd2 x -Hd1
! Medium vs denormal.
2- =0< 0i1 -1 x 1
2- > 0i1 -1 x 1i1
2- =0> -0i1 1 x -1
2- < -0i1 1 x -1i1
2- =0< 0i1 -1d1 x 1d1
2- > 0i1 -1d1 x 1
2- =0> -0i1 1d1 x -1d1
2- < -0i1 1d1 x -1
2- =0< 0i1 -2d1 x 2d1
2- > 0i1 -2d1 x 2
2- =0> -0i1 2d1 x -2d1
2- < -0i1 2d1 x -2
2- =0< 0i1 -2d2 x 2d2
2- > 0i1 -2d2 x 2d1
2- =0> -0i1 2d2 x -2d2
2- < -0i1 2d2 x -2d1
!
! Magnitude subtract when an operand
! is in the sticky bit. The interesting
! cases will arise when directed rounding
! forces a nonzero cancellation.
! Huge and medium.
2- => Hm1 1 x Hm1
2- 0< Hm1 1 x Hm1d1
2- =< -Hm1 -1 x -Hm1
2- 0> -Hm1 -1 x -Hm1d1
2- => Hm1d1 1 x Hm1d1
2- 0< Hm1d1 1 x Hm1d2
2- =< -Hm1d1 -1 x -Hm1d1
2- 0> -Hm1d1 -1 x -Hm1d2
2- => Hd1 1 x Hd1
2- 0< Hd1 1 x Hd2
2- =< -Hd1 -1 x -Hd1
2- 0> -Hd1 -1 x -Hd2
2- => Hd2 1 x Hd2
2- 0< Hd2 1 x Hd3
2- =< -Hd2 -1 x -Hd2
2- 0> -Hd2 -1 x -Hd3
! Huge and tiny.
2- => Hd1 0i1 x Hd1
2- 0< Hd1 0i1 x Hd2
2- =< -Hd1 -0i1 x -Hd1
2- 0> -Hd1 -0i1 x -Hd2
2- => -0i3 -Hm1 x Hm1
2- 0< -0i3 -Hm1 x Hm1d1
2- =< 0i3 Hm1 x -Hm1
2- 0> 0i3 Hm1 x -Hm1d1
! Medium and tiny.
2- => 1d1 0i1 x 1d1
2- 0< 1d1 0i1 x 1d2
2- =< -2d1 -0i1 x -2d1
2- 0> -2d1 -0i1 x -2d2
2- => -0i3 -3 x 3
2- 0< -0i3 -3 x 3d1
2- =< 0i3 5 x -5
2- 0> 0i3 5 x -5d1
! Add magnitude with difference in LSB so,
! except for denorms, round bit is crucial.
! Half-way cases arise.
! Medium cases.
2- =0< 1i1 -1 x 2
2- > 1i1 -1 x 2i1
2- =0> -1i1 1 x -2
2- < -1i1 1 x -2i1
2- =0> -2 2i1 x -4
2- < -2 2i1 x -4i1
2- =0< 2 -2i1 x 4
2- > 2 -2i1 x 4i1
2- => 1 -1i3 x 2i2
2- 0< 1 -1i3 x 2i1
2- =< -1 1i3 x -2i2
2- 0> -1 1i3 x -2i1
2- =< -2i1 2i2 x -4i2
2- 0> -2i1 2i2 x -4i1
2- => 2i1 -2i2 x 4i2
2- 0< 2i1 -2i2 x 4i1
! Huge cases.
2- => Hd2 -Hd1 xo H
2- 0< Hd2 -Hd1 xo Hd1
2- =< -Hd2 Hd1 xo -H
2- 0> -Hd2 Hd1 xo -Hd1
2- => Hm1d1 -Hm1 xo H
2- 0< Hm1d1 -Hm1 x Hd1
2- =< -Hm1d1 Hm1 xo -H
2- 0> -Hm1d1 Hm1 x -Hd1
2- => Hm1i1 -Hm1 xo H
2- 0< Hm1i1 -Hm1 xo Hd1
2- =< -Hm1i1 Hm1 xo -H
2- 0> -Hm1i1 Hm1 xo -Hd1
2- =0< Hm2i1 -Hm2 x Hm1
2- > Hm2i1 -Hm2 x Hm1i1
2- =0> -Hm2i1 Hm2 x -Hm1
2- < -Hm2i1 Hm2 x -Hm1i1
2- =0< Hm1d2 -Hm1d1 x Hd2
2- > Hm1d2 -Hm1d1 x Hd1
2- =0> -Hm1d2 Hm1d1 x -Hd2
2- < -Hm1d2 Hm1d1 x -Hd1
! Check rounding.
2- > 2 -1u1 x 2i1
2- =0< 2 -1u1 x 2
2- => 2i1 -1u1 x 2i2
2- 0< 2i1 -1u1 x 2i1
2- => 4d1 -1u1 x 4
2- 0< 4d1 -1u1 x 4d1
2- > 4d1 -1u1d1 x 4
2- 0=< 4d1 -1u1d1 x 4d1
2- =< -4d1 1u1 x -4
2- 0> -4d1 1u1 x -4d1
2- < -4d1 1u1d1 x -4
2- 0=> -4d1 1u1d1 x -4d1
! NaN operands.
2- ALL Q 0 OK Q
2- ALL Q -0 OK Q
2- ALL 0 Q OK Q
2- ALL -0 Q OK Q
2- ALL Q 1 OK Q
2- ALL Q -1 OK Q
2- ALL 1 Q OK Q
2- ALL -1 Q OK Q
2- ALL Ed1 Q OK Q
```

```
2- ALL -Ed1 Q OK Q
2- ALL Q Ed1 OK Q
2- ALL Q -Ed1 OK Q
2- ALL Q Oi1 OK Q
2- ALL Q -Oi1 OK Q
2- ALL Oi1 Q OK Q
2- ALL -Oi1 Q OK Q
2- ALL Q Hd1 OK Q
2- ALL Q -Hd1 OK Q
2- ALL Hd1 Q OK Q
2- ALL -Hd1 Q OK Q
2- ALL Q H OK Q
2- ALL Q -H OK Q
2- ALL H Q OK Q
2- ALL -H Q OK Q
2- ALL Q Q OK Q
2- ALL S O i Q
2- ALL S -O i Q
2- ALL O S i Q
2- ALL -O S i Q
2- ALL S 1 i Q
2- ALL S -1 i Q
2- ALL 1 S i Q
2- ALL -1 S i Q
2- ALL Ed1 S i Q
2- ALL -Ed1 S i Q
2- ALL S Ed1 i Q
2- ALL S -Ed1 i Q
2- ALL S Oi1 i Q
2- ALL S -Oi1 i Q
2- ALL Oi1 S i Q
2- ALL -Oi1 S i Q
2- ALL S Hd1 i Q
2- ALL S -Hd1 i Q
2- ALL Hd1 S i Q
2- ALL -Hd1 S i Q
2- ALL S H i Q
2- ALL S -H i Q
2- ALL H S i Q
2- ALL -H S i Q
2- ALL Q S i Q
2- ALL S Q i Q
2- ALL S S i Q
```

```
! First some easy tests for consistency.
2* ALL 1 1 OK 1
2* ALL 1 2 OK 2
2* ALL 2 1 OK 2
2* ALL 2 3 OK 6
2* ALL 3 2 OK 6
2* ALL 3 3 OK 9
! Check out sign manipulation.
2* ALL -1 1 OK -1
2* ALL -1 2 OK -2
2* ALL 2 -1 OK -2
2* ALL -2 3 OK -6
2* ALL 3 -2 OK -6
2* ALL -3 3 OK -9
2* ALL -1 -1 OK 1
2* ALL -1 -2 OK 2
2* ALL -2 -1 OK 2
2* ALL -2 -3 OK 6
2* ALL -3 -2 OK 6
2* ALL -3 -3 OK 9
! Some zero tests, round mode is
! irrelevant.
2* ALL 0 0 OK 0
2* ALL -0 0 OK -0
2* ALL 0 -0 OK -0
2* ALL -0 -0 OK 0
! Infinity tests, round mode
! irrelevant.
2* ALL H H OK H
2* ALL -H H OK -H
2* ALL H -H OK -H
2* ALL -H -H OK H
! Inf * 0 -- always bad news.
2* ALL H 0 i Q
2* ALL -0 H i -Q
2* ALL H -0 i -Q
2* ALL -0 -H i Q
! Inf * small_integer -> Inf.
2* ALL H 1 OK H
2* ALL -2 H OK -H
2* ALL H -3 OK -H
2* ALL -4 -H OK H
2* ALL 5 H OK H
2* ALL -H 6 OK -H
2* ALL 7 -H OK -H
2* ALL -H -8 OK H
! Inf * huge -> Inf.
2* ALL Hm1 H OK H
2* ALL -Hm2 H OK -H
2* ALL H -Hm1 OK -H
2* ALL -H -Hm2 OK H
2* ALL H Hm1d1 OK H
2* ALL -Hm2d1 H OK -H
2* ALL H -Hd1 OK -H
2* ALL -Hd1 -H OK H
! Inf * tiny -> Inf.
2* ALL E H OK H
2* ALL -Ep1 H OK -H
2* ALL H -Ep1 OK -H
2* ALL -H -E OK H
2* ALL H Ep1d1 OK H
2* ALL -Ei1 H OK -H
2* ALL H -Ei1 OK -H
2* ALL -Ep1d1 -H OK H
! Inf * denormalized -> Inf.
2* ALL 0i1 H OK H
2* ALL -0i3 H OK -H
2* ALL H -0i2 OK -H
2* ALL -H -0i4 OK H
2* ALL H Ed1 OK H
2* ALL -Ed1 H OK -H
2* ALL H -Ed1 OK -H
2* ALL -Ed1 -H OK H
! 0 * small_integer -> 0.
2* ALL 0 1 OK 0
2* ALL -2 0 OK -0
2* ALL 0 -3 OK -0
2* ALL -4 -0 OK 0
2* ALL 5 0 OK 0
2* ALL -0 6 OK -0
2* ALL 7 -0 OK -0
2* ALL -0 -8 OK 0
! 0 * huge -> 0.
2* ALL Hm1 0 OK 0
2* ALL -Hm2 0 OK -0
2* ALL 0 -Hm1 OK -0
2* ALL -0 -Hm2 OK 0
2* ALL 0 Hm1d1 OK 0
2* ALL -Hm2d1 0 OK -0
2* ALL 0 -Hm2d1 OK -0
2* ALL -Hm1d1 -0 OK 0
2* ALL Hd1 0 OK 0
2* ALL -Hd1 -0 OK 0
2* ALL 0 -Hd1 OK -0
2* ALL -0 Hd1 OK -0
! 0 * tiny -> 0.
2* ALL E 0 OK 0
2* ALL -Ep1 0 OK -0
2* ALL 0 -Ep1 OK -0
2* ALL -0 -E OK 0
2* ALL 0 Ep1d1 OK 0
2* ALL -Ei1 0 OK -0
2* ALL 0 -Ei1 OK -0
2* ALL -Ep1d1 -0 OK 0
! 0 * denormalized -> 0.
2* ALL 0i1 0 OK 0
2* ALL -0i3 0 OK -0
2* ALL 0 -0i2 OK -0
2* ALL -0 -0i4 OK 0
2* ALL 0 Ed1 OK 0
2* ALL -Ed1 0 OK -0
2* ALL 0 -Ed1 OK -0
2* ALL -Ed1 -0 OK 0
! Exact cases huge and 2.
2* ALL 2 Hm2 OK Hm1
2* ALL Hm2 -2 OK -Hm1
2* ALL -2 Hm2d1 OK -Hm1d1
2* ALL 2 -Hm2d3 OK -Hm1d3
2* ALL 2 Hm2 OK Hm1
2* ALL Hm2 -2 OK -Hm1
2* ALL -2 Hm2d1 OK -Hm1d1
2* ALL 2 -Hm2d3 OK -Hm1d3
2* ALL 2 Hm1d1 OK Hd1
2* ALL Hm1d1 -2 OK -Hd1
2* ALL -2 Hm2i1 OK -Hm1i1
2* ALL 2 -Hm2i3 OK -Hm1i3
2* ALL 2 Hm1d1 OK Hd1
2* ALL Hm1d1 -2 OK -Hd1
```

```
2* ALL -2 Hm2i1 OK -Hm1i1
2* ALL 2 -Hm2i3 OK -Hm1i3
! Exact cases huge and 4.
2* ALL 4 Hm2d1 OK Hd1
2* ALL -4 Hm2d1 OK -Hd1
2* ALL 4 -Hm2d1 OK -Hd1
2* ALL -4 -Hm2d1 OK Hd1
2* ALL 4 Hm2d1 OK Hd1
2* ALL -4 Hm2d1 OK -Hd1
2* ALL 4 -Hm2d1 OK -Hd1
2* ALL -4 -Hm2d1 OK Hd1
2* ALL Hm2d3 4 OK Hd3
2* ALL Hm2d3 -4 OK -Hd3
2* ALL -Hm2d3 4 OK -Hd3
2* ALL -Hm2d3 -4 OK Hd3
2* ALL Hm2d3 4 OK Hd3
2* ALL Hm2d3 -4 OK -Hd3
2* ALL -Hm2d3 4 OK -Hd3
2* ALL -Hm2d3 -4 OK Hd3
! Exact cases tiny and 2.
2* ALL 2 E OK Ep1
2* ALL E -2 OK -Ep1
2* ALL -2 Ei1 OK -Ep1i1
2* ALL 2 -Ei3 OK -Ep1i3
2* ALL 2 E OK Ep1
2* ALL E -2 OK -Ep1
2* ALL -2 Ei9 OK -Ep1i9
2* ALL 2 -Ei5 OK -Ep1i5
2* ALL 2 Ei1 OK Ep1i1
2* ALL Ei1 -2 OK -Ep1i1
2* ALL -2 Ei5 OK -Ep1i5
2* ALL 2 -Ei3 OK -Ep1i3
2* ALL 2 Ei1 OK Ep1i1
2* ALL Ei1 -2 OK -Ep1i1
2* ALL -2 Ei5 OK -Ep1i5
2* ALL 2 -Ei3 OK -Ep1i3
! Just below denormalization threshold.
2* ALL Ed1 2 OK Ep1d2
2* ALL -2 Ed3 OK -Ep1d6
2* ALL -Ed3 -2 OK Ep1d6
2* ALL -2 Ed3 OK -Ep1d6
2* ALL Ed4 2 OK Ep1d8
2* ALL 2 -Ed3 OK -Ep1d6
! Normalizing tinies.
2* ALL 0i1 2 OK 0i2
2* ALL 3 0i2 OK 0i6
2* ALL -0i1 5 OK -0i5
2* ALL 1 -0i9 OK -0i9
2* ALL -0i4 -1 OK 0i4
2* ALL 4 0i2 OK 0i8
2* ALL 0i1 2 OK 0i2
2* ALL 3 0i2 OK 0i6
2* ALL -0i1 5 OK -0i5
2* ALL 1 -0i9 OK -0i9
2* ALL -0i4 -1 OK 0i4
2* ALL 4 0i2 OK 0i8
2* ALL 0i1 2 OK 0i2
2* ALL 3 0i2 OK 0i6
2* ALL -0i1 5 OK -0i5
2* ALL 1 -0i9 OK -0i9
2* ALL -0i4 -1 OK 0i4
2* ALL 4 0i2 OK 0i8
! 1.0 * various.
2* ALL 1 Ep1i3 OK Ep1i3
2* ALL -Ep1d2 1 OK -Ep1d2
2* ALL -1 Ei9 OK -Ei9
2* ALL -Ei1 -1 OK Ei1
2* ALL 1 Ep1i3 OK Ep1i3
2* ALL -Ep1d2 1 OK -Ep1d2
2* ALL -1 Ei9 OK -Ei9
2* ALL -Ei1 -1 OK Ei1
2* ALL 1 Ed3 OK Ed3
2* ALL -0i2 1 OK -0i2
2* ALL -1 0i9 OK -0i9
2* ALL -Ed1 -1 OK Ed1
2* ALL 1 Ed3 OK Ed3
2* ALL -0i2 1 OK -0i2
2* ALL -1 0i9 OK -0i9
2* ALL -Ed1 -1 OK Ed1
! Now some tricky rounding cases
! involving 1.0 with some ulps.
! result = 1.00000...010|000...0001
2* =0< 1i1 1i1 x 1i2
2* > 1i1 1i1 x 1i3
! Try signs...
2* =0> -1i1 1i1 x -1i2
2* < -1i1 1i1 x -1i3
2* =0> 1i1 -1i1 x -1i2
2* < 1i1 -1i1 x -1i3
2* =0< -1i1 -1i1 x 1i2
2* > -1i1 -1i1 x 1i3
! result = 1.0000..011|0000100
2* =0< 1i2 1i1 x 1i3
2* > 1i2 1i1 x 1i4
! Try signs...
2* =0> -1i2 1i1 x -1i3
2* < -1i2 1i1 x -1i4
2* =0> 1i1 -1i2 x -1i3
2* < 1i1 -1i2 x -1i4
2* =0< -1i2 -1i1 x 1i3
2* > -1i1 -1i2 x 1i4
2* > -1i2 -1i1 x 1i4
2* =0< -1i1 -1i2 x 1i3
! (m + k ulps of m) * (1 + j ulps of 1)
! = m + (k + m*j/2^floor(log m)) ulps
! of m + tiny.
2* => 3i1 1i1 x 3i3
2* 0< 3i1 1i1 x 3i2
2* >= 3i1 1i3 x 3i6
2* 0< 3i1 1i3 x 3i5
2* =< -3i1 1i1 x -3i3
2* 0> -3i1 1i1 x -3i2
2* <= 3i1 -1i3 x -3i6
2* 0> 3i1 -1i3 x -3i5
2* > 5i1 1i1 x 5i3
2* =0< 5i1 1i1 x 5i2
2* > -5i1 -1i1 x 5i3
2* =0< -5i1 -1i1 x 5i2
2* >= 7i1 1i1 x 7i3
2* <0 7i1 1i1 x 7i2
2* 0<= 3d1 1d1 x 3d2
2* > 3d1 1d1 x 3d1
```

```
2* 0< 3d1 1d3 x 3d4
2* => 3d1 1d3 x 3d3
2* 0>= -3d1 1d1 x -3d2
2* < -3d1 1d1 x -3d1
2* 0> 3d1 -1d3 x -3d4
2* =< 3d1 -1d3 x -3d3
2* => 3d1 1d2 x 3d2
2* 0< 3d1 1d2 x 3d3
2* 0<= 5d1 1d1 x 5d2
2* > 5d1 1d1 x 5d1
2* 0<= -5d1 -1d1 x 5d2
2* > -5d1 -1d1 x 5d1
2* <=0 7d1 1d1 x 7d2
2* > 7d1 1d1 x 7d1
2* => 7d1 1d4 x 7d4
2* 0< 7d1 1d4 x 7d5
! Some overflow conditions, watching
! round mode.
2* => Hm1 2 ox H
2* 0< Hm1 2i1 ox Hd1
2* =< -3d2 Hm1 ox -H
2* 0> Hm1 -4i5 ox -Hd1
2* => -5d2 -Hm1 ox H
2* 0< Hm1 6i1 ox Hd1
2* =< -7d7 Hm1 ox -H
2* 0> Hm1 -8i3 ox -Hd1
2* => -9i1 -Hm1 ox H
2* 0< Hm1 6 ox Hd1
2* =< -9 Hm1 ox -H
2* 0> Hm1 -2 ox -Hd1
2* 0< -7 -Hm1 ox Hd1
2* => Hm1 2 ox H
2* 0> -5 Hm1 ox -Hd1
2* 0> Hm1 -2 ox -Hd1
2* => -3 -Hm1 ox H
! Heavy overflow conditions,
! watching round mode.
2* => Hm1 Hm1 xo H
2* =< -Hd3 Hm1 xo -H
2* =< Hm1 -Hm2i4 xo -H
2* => -Hm1i5 -Hm1i1 ox H
2* => Hm1i9 Hd6 xo H
2* =< -Hm2d7 Hm1 xo -H
2* =< Hm1 -Hm2 xo -H
2* => -Hd1 -Hd1 xo H
2* 0< Hm1 Hm2i6 xo Hd1
2* =< -Hm1d9 Hm2i1 xo -H
2* =< Hm1 -Hm1 xo -H
2* 0< -Hm2d7 -Hd1 xo Hd1
2* => Hm1 Hd2 xo H
2* 0> -Hm2 Hm1 xo -Hd1
2* 0> Hm1i9 -Hm2i2 xo -Hd1
2* => -Hd3 -Hm1i1 xo H
! Mixed bag overflow conditions,
! watching round mode. Tricky cases
! require careful look at power series
! expansion. Example -- -Hmd1 * 1i1:
! In single...
! -(2^127 (1 - 2^-24)) * (1 + 2^-23) -->
! -( 2^127 (1 + 2^-24 - 2^-47) ) -->
! -2^127 except when rounding <, in which
! case -( 2^127 (1 + 2^-23) ); that is,
! -Hm or -Hmi1, respectively!
2* =0> -Hm1d1 1i1 x -Hm1
2* < -Hm1d1 1i1 x -Hm1i1
2* =0< -1d1 -Hd1 x Hd2
2* > -1d1 -Hd1 x Hd1
2* < -Hm2d1 2i1 x -Hm1i1
2* =0> -Hm2d1 2i1 x -Hm1
2* <= Hm1d3 -2i8 xo -H
2* >0 Hm1d3 -2i8 xo -Hd1
2* =0< -Hm2d7 -4d1 x Hd8
2* > -Hm2d7 -4d1 x Hd7
2* => 1i2 Hd2 xo H
2* 0< 1i2 Hd2 xo Hd1
2* =< Hm1i9 -6i2 xo -H
2* 0> Hm1i9 -6i2 xo -Hd1
2* => -Hd3 -3i1 xo H
2* 0< -Hd3 -3i1 xo Hd1
! Exact and below denomalization
! threshold -- no underflow.
2* ALL E 1d2 OK Ed1
2* ALL 0i1 1 OK 0i1
2* ALL 1 -0i1 OK -0i1
2* ALL Ep1d2 1m1 OK Ed1
2* ALL -Ep1d4 -1m1 OK Ed2
2* ALL Ep1d2 -1m1 OK -Ed1
2* ALL -Ep1d4 1m1 OK -Ed2
2* ALL Ep1d8 1m1 OK Ed4
2* ALL 0i8 1m3 OK 0i1
2* ALL 0i6 1m1 OK 0i3
2* ALL -0i8 1m3 OK -0i1
2* ALL 0i6 -1m1 OK -0i3
! Inexact, extreme underflows.
2* =0< E E xu 0
2* =0< -E -E xu 0
2* > E Ep1 xu 0i1
2* > -Ep1 -Ep1 xu 0i1
2* =0> -E E xu -0
2* =0> E -E xu -0
2* < -E Ep1 xu -0i1
2* < E -E xu -0i1
2* =0< Ed1 Ed2 xu 0
2* =0< -Ed1 -Ed2 xu 0
2* > Ed1 Ed2 xu 0i1
2* > -Ed1 -Ed2 xu 0i1
2* =0> -Ed9 Ep1i3 xu -0
2* =0> Ed9 -Ep1i3 xu -0
2* < -Ed9 Ep1i3 xu -0i1
2* < Ed9 -Ep1i3 xu -0i1
2* > 0i1 1m1 xu 0i1
2* =0< 0i1 1m1 xu 0
2* < 1m1 -0i1 xu -0i1
2* =0> 1m1 -0i1 xu -0
2* <0 0i1 1d1 xu 0
2* => 0i1 1d1 xu 0i1
2* > 0i1 0i1 xu 0i1
2* =0< 0i1 0i1 xu 0
2* >0 -0i1 1d1 xu -0
2* =< 0i1 -1d1 xu -0i1
2* < 0i1 -0i1 xu -0i1
2* =0> -0i1 0i1 xu -0
! Underflow, barely.
2* 0< Ep1d1 1m1 xu Ed1
2* 0< -Ep1d1 -1m1 xu Ed1
2* 0> -Ep1d1 1m1 xu -Ed1
2* >= Ep1d1 1m1 xu E
2* <= Ep1d1 -1m1 xu -E
```

```
2* 0< Ed1 1i1 xu Ed1
2* 0> Ed1 -1i1 xu -Ed1
2* 0< Ei1 1d6 xu Ed3
2* > Ed2 1d4 xu Ed3
2* 0< Ed4 1i1 xu Ed4
2* 0< Ei1 1d2 xu Ed1
! Underflow, unless detected as accuracy
! loss due to denormalization.
2* >= Ed2 1i1 xv Ed1
2* <= Ed2 -1i1 xv -Ed1
2* >= Ed8 1i1 xv Ed7
2* <= -Ed9 1i1 xv -Ed8
2* <= Ed8 -1i1 xv -Ed7
2* => Ei1 1d6 xv Ed2
2* <=0 Ed2 1d4 xv Ed4
! Underflow, only if tininess is detected
! before rounding.
2* >= Ed1 1i1 xw E
2* <= -Ed1 1i1 xw -E
2* >= Ed8 1i8 xw E
2* <= Ed8 -1i8 xw -E
2* >= Ei1 1d2 xw E
2* >= Ei2 1d4 xw E
! NaN operands.
2* ALL Q 0 OK Q
2* ALL Q -0 OK Q
2* ALL 0 Q OK Q
2* ALL -0 Q OK Q
2* ALL Q 1 OK Q
2* ALL Q -1 OK Q
2* ALL 1 Q OK Q
2* ALL -1 Q OK Q
2* ALL Ed1 Q OK Q
2* ALL -Ed1 Q OK Q
2* ALL Q Ed1 OK Q
2* ALL Q -Ed1 OK Q
2* ALL Q 0i1 OK Q
2* ALL Q -0i1 OK Q
2* ALL 0i1 Q OK Q
2* ALL -0i1 Q OK Q
2* ALL Q Hd1 OK Q
2* ALL Q -Hd1 OK Q
2* ALL Hd1 Q OK Q
2* ALL -Hd1 Q OK Q
2* ALL Q H OK Q
2* ALL Q -H OK Q
2* ALL H Q OK Q
2* ALL -H Q OK Q
2* ALL Q Q OK Q
2* ALL S 0 i Q
2* ALL S -0 i Q
2* ALL 0 S i Q
2* ALL -0 S i Q
2* ALL S 1 i Q
2* ALL S -1 i Q
2* ALL 1 S i Q
2* ALL -1 S i Q
2* ALL Ed1 S i Q
2* ALL -Ed1 S i Q
2* ALL S Ed1 i Q
2* ALL S -Ed1 i Q
2* ALL S 0i1 i Q
2* ALL S -0i1 i Q
2* ALL 0i1 S i Q
2* ALL -0i1 S i Q
2* ALL S Hd1 i Q
2* ALL S -Hd1 i Q
2* ALL Hd1 S i Q
2* ALL -Hd1 S i Q
2* ALL S H i Q
2* ALL S -H i Q
2* ALL H S i Q
2* ALL -H S i Q
2* ALL Q S i Q
2* ALL S Q i Q
2* ALL S S i Q
```

```
! First the consistency checks.
2/ ALL 1 1 OK 1
2/ ALL 2 1 OK 2
2/ ALL 9 3 OK 3
2/ ALL 5 5 OK 1
2/ ALL 8 2 OK 4
! Check out sign manipulation.
2/ ALL -1 1 OK -1
2/ ALL -2 1 OK -2
2/ ALL 2 -1 OK -2
2/ ALL -8 2 OK -4
2/ ALL 3 -3 OK -1
2/ ALL -7 7 OK -1
2/ ALL -1 -1 OK 1
2/ ALL -2 -1 OK 2
2/ ALL -6 -3 OK 2
2/ ALL -9 -3 OK 3
! Some zero tests, round mode
! is irrelevant.
2/ ALL 0 0 i Q
2/ ALL -0 0 i -Q
2/ ALL 0 -0 i -Q
2/ ALL -0 -0 i Q
! Infinity tests, round mode
! irrelevant.
2/ ALL H H i Q
2/ ALL -H H i -Q
2/ ALL H -H i -Q
2/ ALL -H -H i Q
! Inf / 0 -> Inf with no problem.
2/ ALL H 0 OK H
2/ ALL -H 0 OK -H
2/ ALL H -0 OK -H
2/ ALL -H -0 OK H
! 0 / Inf -> 0 with no problem.
2/ ALL 0 H OK 0
2/ ALL -0 H OK -0
2/ ALL 0 -H OK -0
2/ ALL -0 -H OK 0
! Inf / small_integer -> Inf.
2/ ALL H 1 OK H
2/ ALL -H 2 OK -H
2/ ALL H -3 OK -H
2/ ALL -H -4 OK H
2/ ALL H 5 OK H
2/ ALL -H 6 OK -H
2/ ALL H -7 OK -H
2/ ALL -H -8 OK H
! Small_int / Inf -> 0.
2/ ALL 1 H OK 0
2/ ALL -2 H OK -0
2/ ALL 3 -H OK -0
2/ ALL -4 -H OK 0
2/ ALL 5 H OK 0
2/ ALL -6 H OK -0
2/ ALL 7 -H OK -0
2/ ALL -8 -H OK 0
! Huge / Inf -> 0.
2/ ALL Hm1 H OK 0
2/ ALL -Hm2 H OK -0
2/ ALL Hm1 -H OK -0
2/ ALL -Hm2 -H OK 0
2/ ALL Hm1d1 H OK 0
2/ ALL -Hm2d1 H OK -0
2/ ALL Hd1 -H OK -0
2/ ALL -Hd1 -H OK 0
! Inf / huge -> Inf.
2/ ALL H Hm1 OK H
2/ ALL -H Hm2 OK -H
2/ ALL H -Hm1 OK -H
2/ ALL -H -Hm2 OK H
2/ ALL H Hm1d1 OK H
2/ ALL H -Hm2d1 OK -H
2/ ALL H -Hd1 OK -H
2/ ALL -H -Hd1 OK H
! Inf / tiny -> Inf.
2/ ALL H E OK H
2/ ALL -H Ep1 OK -H
2/ ALL H -Ep1 OK -H
2/ ALL -H -E OK H
2/ ALL H Ep1d1 OK H
2/ ALL -H Ei1 OK -H
2/ ALL H -Ei1 OK -H
2/ ALL -H -Ep1d1 OK H
! Tiny / Inf -> 0.
2/ ALL E H OK 0
2/ ALL -Ep1 H OK -0
2/ ALL Ep1 -H OK -0
2/ ALL -E -H OK 0
2/ ALL Ep1d1 H OK 0
2/ ALL -Ei1 H OK -0
2/ ALL Ei1 -H OK -0
2/ ALL -Ep1d1 -H OK 0
! Inf / denormalized -> Inf.
2/ ALL H 0i1 OK H
2/ ALL -H 0i3 OK -H
2/ ALL H -0i2 OK -H
2/ ALL -H -0i4 OK H
2/ ALL H Ed1 OK H
2/ ALL -H Ed1 OK -H
2/ ALL H -Ed1 OK -H
2/ ALL -H -Ed1 OK H
! Denorm / Inf -> 0.
2/ ALL 0i1 H OK 0
2/ ALL -0i3 H OK -0
2/ ALL 0i2 -H OK -0
2/ ALL -0i4 -H OK 0
2/ ALL Ed1 H OK 0
2/ ALL -Ed1 H OK -0
2/ ALL Ed1 -H OK -0
2/ ALL -Ed1 -H OK 0
! 0 / small_integer -> 0.
2/ ALL 0 1 OK 0
2/ ALL -0 2 OK -0
2/ ALL 0 -3 OK -0
2/ ALL -0 -4 OK 0
2/ ALL 0 5 OK 0
2/ ALL -0 6 OK -0
2/ ALL 0 -7 OK -0
2/ ALL -0 -8 OK 0
! Small_int / 0 -> Inf with DivBy0.
2/ ALL 1 0 z H
2/ ALL -2 0 z -H
2/ ALL 3 -0 z -H
2/ ALL -4 -0 z H
2/ ALL 5 0 z H
2/ ALL -6 0 z -H
2/ ALL 7 -0 z -H
```

```
2/ ALL -8 -0 z H
! 0 / huge -> 0.
2/ ALL 0 Hm1 OK 0
2/ ALL -0 Hm2 OK -0
2/ ALL 0 -Hm1 OK -0
2/ ALL -0 -Hm2 OK 0
2/ ALL 0 Hm1d1 OK 0
2/ ALL -0 Hm2d1 OK -0
2/ ALL 0 -Hm2d1 OK -0
2/ ALL -0 -Hm1d1 OK 0
! Huge / 0 -> Inf with DivBy0.
2/ ALL Hm1 0 z H
2/ ALL -Hm2 0 z -H
2/ ALL Hm1 -0 z -H
2/ ALL -Hm2 -0 z H
2/ ALL Hm1d1 0 z H
2/ ALL -Hm2d1 0 z -H
2/ ALL Hm2d1 -0 z -H
2/ ALL -Hm1d1 -0 z H
! 0 / tiny -> 0.
2/ ALL 0 E OK 0
2/ ALL -0 Ep1 OK -0
2/ ALL 0 -Ep1 OK -0
2/ ALL -0 -E OK 0
2/ ALL 0 Ep1d1 OK 0
2/ ALL -0 Ei1 OK -0
2/ ALL 0 -Ei1 OK -0
2/ ALL -0 -Ep1d1 OK 0
! Tiny / 0 -> Inf with DivBy0.
2/ ALL E 0 z H
2/ ALL -Ep1 0 z -H
2/ ALL Ep1 -0 z -H
2/ ALL -E -0 z H
2/ ALL Ep1d1 0 z H
2/ ALL -Ei1 0 z -H
2/ ALL Ei1 -0 z -H
2/ ALL -Ep1d1 -0 z H
! 0 / denormalized -> 0.
2/ ALL 0 0i1 OK 0
2/ ALL -0 0i3 OK -0
2/ ALL 0 -0i2 OK -0
2/ ALL -0 -0i4 OK 0
2/ ALL 0 Ed1 OK 0
2/ ALL -0 Ed1 OK -0
2/ ALL 0 -Ed1 OK -0
2/ ALL -0 -Ed1 OK 0
! Denormalized * 0 -> Inf, DivBy0.
2/ ALL 0i1 0 z H
2/ ALL -0i3 0 z -H
2/ ALL 0i2 -0 z -H
2/ ALL -0i4 -0 z H
2/ ALL Ed1 0 z H
2/ ALL -Ed1 0 z -H
2/ ALL Ed1 -0 z -H
2/ ALL -Ed1 -0 z H
! Exact cases huge and 2.
2/ ALL Hm1 2 OK Hm2
2/ ALL Hm1 -2 OK -Hm2
2/ ALL -Hm1d1 2 OK -Hm2d1
2/ ALL Hm1d3 -2 OK -Hm2d3
2/ ALL Hm1 2 OK Hm2
2/ ALL Hm1 -2 OK -Hm2
2/ ALL -Hm1d1 2 OK -Hm2d1
2/ ALL Hm1d3 -2 OK -Hm2d3
2/ ALL Hd1 Hm1d1 OK 2
2/ ALL Hd1 -2 OK -Hm1d1
2/ ALL -Hm1i1 Hm2i1 OK -2
2/ ALL Hm1i3 -Hm2i3 OK -2
2/ ALL Hd1 Hm1d1 OK 2
2/ ALL Hd1 -2 OK -Hm1d1
2/ ALL -Hm1i1 Hm2i1 OK -2
2/ ALL Hm1i3 -Hm2i3 OK -2
! Exact cases huge and 4.
2/ ALL Hd1 Hm2d1 OK 4
2/ ALL -Hd1 Hm2d1 OK -4
2/ ALL Hd1 -Hm2d1 OK -4
2/ ALL -Hd1 -Hm2d1 OK 4
2/ ALL Hd1 Hm2d1 OK 4
2/ ALL -Hd1 Hm2d1 OK -4
2/ ALL Hd1 -Hm2d1 OK -4
2/ ALL -Hd1 -Hm2d1 OK 4
2/ ALL Hd3 4 OK Hm2d3
2/ ALL Hd3 -4 OK -Hm2d3
2/ ALL -Hd3 4 OK -Hm2d3
2/ ALL -Hd3 -4 OK Hm2d3
2/ ALL Hd3 4 OK Hm2d3
2/ ALL Hd3 -4 OK -Hm2d3
2/ ALL -Hd3 4 OK -Hm2d3
2/ ALL -Hd3 -4 OK Hm2d3
! Exact cases tiny and 2.
2/ ALL Ep1 E OK 2
2/ ALL Ep1 -2 OK -E
2/ ALL -Ep1i1 Ei1 OK -2
2/ ALL Ep1i3 -2 OK -Ei3
2/ ALL Ep1 E OK 2
2/ ALL Ep1 -2 OK -E
2/ ALL -Ep1i1 Ei1 OK -2
2/ ALL Ep1i3 -2 OK -Ei3
2/ ALL Ep1i1 Ei1 OK 2
2/ ALL Ep1i1 -2 OK -Ei1
2/ ALL -Ep1i5 Ei5 OK -2
2/ ALL Ep1i3 -Ei3 OK -2
2/ ALL Ep1i1 Ei1 OK 2
2/ ALL Ep1i1 -2 OK -Ei1
2/ ALL -Ep1i5 Ei5 OK -2
2/ ALL Ep1i3 -Ei3 OK -2
2/ ALL Ed1 1m1 OK Ep1d2
2/ ALL Ed1 1m9 OK Ep9d2
! Huge / tiny -> overflow.
2/ => Hm1 1m1 ox H
2/ 0< Hm1 1m1 ox Hd1
2/ => -Hm1 -1m1 ox H
2/ 0< -Hm1 -1m1 ox Hd1
2/ =< Hm1 -1m1 ox -H
2/ =< -Hm1 1m1 ox -H
2/ 0> Hm1 -1m1 ox -Hd1
2/ 0> -Hm1 1m1 ox -Hd1
2/ => Hm9 Ep9 ox H
2/ 0< Hm9 Ep9 ox Hd1
2/ => Hd1 0i1 ox H
2/ 0< Hd1 0i1 ox Hd1
2/ => Hm1 Ed1 ox H
2/ 0< Hm1 Ed1 ox Hd1
2/ => Hd1 1d1 ox H
2/ 0< Hd1 1d1 ox Hd1
! Will underflow unless loss of accuracy
! is detected as a denormalization loss.
2/ =0< E 1i1 xv Ed1
```

```
2/ =0> -E 1i1 xv -Ed1
2/ >= Ed2 1d2 xv Ed1
2/ >= Ed9 1d2 xv Ed8
2/ <= -Ed8 1d2 xv -Ed7
2/ <=0 Ei1 1i2 xv Ed1
2/ <=0 Ed1 1i2 xv Ed3
2/ <=0 Ei2 1i6 xv Ed4
2/ 0< Ed1 1i1 xv Ed2
! Tiny / huge -> underflow.
2/ =<0 0i1 Hd1 xu 0
2/ > 0i1 Hd1 xu 0i1
2/ =<0 -0i1 -Hd1 xu 0
2/ > -0i1 -Hd1 xu 0i1
2/ =0> 0i1 -Hd1 xu -0
2/ < 0i1 -Hd1 xu -0i1
2/ =0> -0i1 Hd1 xu -0
2/ < -0i1 Hd1 xu -0i1
! Tiny / 2.
2/ > 0i1 2 xu 0i1
2/ =0< 0i1 2 xu 0
2/ > -0i1 -2 xu 0i1
2/ =0< -0i1 -2 xu 0
2/ < 0i1 -2 xu -0i1
2/ =0> 0i1 -2 xu -0
2/ < -0i1 2 xu -0i1
2/ =0> -0i1 2 xu -0
! Barely underflow.
2/ 0< Ep1d1 2 xu Ed1
2/ 0> Ep1d1 -2 xu -Ed1
2/ >= Ep1d1 2 xu E
2/ > E 1i1 xu E
2/ < -E 1i1 xu -E
2/ > Ei1 1i2 xu E
2/ > Ed1 1i2 xu Ed2
! Denorm result but will not underflow.
2/ ALL Ep1d2 2 OK Ed1
2/ ALL Ed1 1 OK Ed1
2/ ALL 0i1 1m1 OK 0i2
2/ ALL 0i1 1m3 OK 0i8
2/ ALL 0i9 9 OK 0i1
2/ ALL 0i9 -9 OK -0i1
2/ ALL Ed1 -1 OK -Ed1
2/ ALL -0i1 1m1 OK -0i2
! Tricky divides based on power
! series expansions
! 1 / (1 + Nulp+) ->
! 1 - (2Nulp-) + tiny.
2/ = 1 1i1 x 1d2
2/ 0 1 1i1 x 1d2
2/ < 1 1i1 x 1d2
2/ > 1 1i1 x 1d1
2/ = 1 1i2 x 1d4
2/ 0 1 1i2 x 1d4
2/ < 1 1i2 x 1d4
2/ > 1 1i2 x 1d3
2/ = 1 1i3 x 1d6
2/ 0 1 1i3 x 1d6
2/ < 1 1i3 x 1d6
2/ > 1 1i3 x 1d5
2/ = 1 1i4 x 1d8
2/ 0 1 1i4 x 1d8
2/ < 1 1i4 x 1d8
2/ > 1 1i4 x 1d7
! 1 / (1 - Nu-) -->
! 1 + (Q/2 u+) + tiny.
2/ = 1 1d1 x 1i1
2/ 0 1 1d1 x 1
2/ < 1 1d1 x 1
2/ > 1 1d1 x 1i1
2/ = 1 1d2 x 1i1
2/ 0 1 1d2 x 1i1
2/ < 1 1d2 x 1i1
2/ > 1 1d2 x 1i2
2/ = 1 1d3 x 1i2
2/ 0 1 1d3 x 1i1
2/ < 1 1d3 x 1i1
2/ > 1 1d3 x 1i2
2/ = 1 1d4 x 1i2
2/ 0 1 1d4 x 1i2
2/ < 1 1d4 x 1i2
2/ > 1 1d4 x 1i3
2/ = 1 1d5 x 1i3
2/ 0 1 1d5 x 1i2
2/ < 1 1d5 x 1i2
2/ > 1 1d5 x 1i3
2/ = 1 1d8 x 1i4
2/ 0 1 1d8 x 1i4
2/ < 1 1d8 x 1i4
2/ > 1 1d8 x 1i5
2/ = 1 1d9 x 1i5
2/ 0 1 1d9 x 1i4
2/ < 1 1d9 x 1i4
2/ > 1 1d9 x 1i5
! (1 + Mu+) / (1 + Nu+) ->
! Case M > Q: (1 + Mu+) *
! (1 - Nu+ + (Nu+)^2 - tiny) -->
! 1 + (M-Q)u+ - (MN-NN)(u+)^2 + tiny ->
! 1 + (M-Q)u+ - tiny.
! M + Q = 3.
2/ = 1i2 1i1 x 1i1
2/ 0 1i2 1i1 x 1
2/ < 1i2 1i1 x 1
2/ > 1i2 1i1 x 1i1
! M + Q = 4.
2/ = 1i3 1i1 x 1i2
2/ 0 1i3 1i1 x 1i1
2/ < 1i3 1i1 x 1i1
2/ > 1i3 1i1 x 1i2
! M + Q = 5.
2/ = 1i4 1i1 x 1i3
2/ 0 1i4 1i1 x 1i2
2/ < 1i4 1i1 x 1i2
2/ > 1i4 1i1 x 1i3
! M + Q = 9.
2/ = 1i7 1i2 x 1i5
2/ 0 1i7 1i2 x 1i4
2/ < 1i7 1i2 x 1i4
2/ > 1i7 1i2 x 1i5
! Q = 17.
2/ = 1i9 1i8 x 1i1
2/ 0 1i9 1i8 x 1
2/ < 1i9 1i8 x 1
2/ > 1i9 1i8 x 1i1
! (1 + Mulp+) / (1 + Nulp+) -->
! Case M < Q: (1 + 2Mulp-) *
! (1 - 2Nulp- + (2Nulp-)^2 - tiny) ->
! 1 - 2(Q-M)ulp- +
!  4(NN-MN)(ulp-)^2 + tiny ->
```

```
! 1 - 2(Q-M)ulp- + tiny.
! M + Q = 3.
2/ = 1i1 1i2 x 1d2
2/ 0 1i1 1i2 x 1d2
2/ < 1i1 1i2 x 1d2
2/ > 1i1 1i2 x 1d1
! M + Q = 4.
2/ = 1i1 1i3 x 1d4
2/ 0 1i1 1i3 x 1d4
2/ < 1i1 1i3 x 1d4
2/ > 1i1 1i3 x 1d3
! M + Q = 5.
2/ = 1i2 1i3 x 1d2
2/ 0 1i2 1i3 x 1d2
2/ < 1i2 1i3 x 1d2
2/ > 1i2 1i3 x 1d1
! M + Q = 11.
2/ = 1i4 1i7 x 1d6
2/ 0 1i4 1i7 x 1d6
2/ < 1i4 1i7 x 1d6
2/ > 1i4 1i7 x 1d5
! M + Q = 14.
2/ = 1i6 1i8 x 1d4
2/ 0 1i6 1i8 x 1d4
2/ < 1i6 1i8 x 1d4
2/ > 1i6 1i8 x 1d3
! (1 - Mulp-) / (1 - Nulp-) -->
! Case M > Q: (1 - Mulp-) *
! (1 + Nulp- + (Nulp-)~2 + tiny) -->
! 1 - (M-Q)ulp- -
!  (MN-NN)(ulp-)~2 + tiny -->
! 1 - (M-Q)ulp- - tiny.
! M + Q = 3.
2/ = 1d2 1d1 x 1d1
2/ 0 1d2 1d1 x 1d2
2/ < 1d2 1d1 x 1d2
2/ > 1d2 1d1 x 1d1
! M + Q = 4.
2/ = 1d3 1d1 x 1d2
2/ 0 1d3 1d1 x 1d3
2/ < 1d3 1d1 x 1d3
2/ > 1d3 1d1 x 1d2
! M + Q = 5.
2/ = 1d3 1d2 x 1d1
2/ 0 1d3 1d2 x 1d2
2/ < 1d3 1d2 x 1d2
2/ > 1d3 1d2 x 1d1
2/ = 1d4 1d1 x 1d3
2/ 0 1d4 1d1 x 1d4
2/ < 1d4 1d1 x 1d4
2/ > 1d4 1d1 x 1d3
! M + Q = 6.
2/ = 1d4 1d2 x 1d2
2/ 0 1d4 1d2 x 1d3
2/ < 1d4 1d2 x 1d3
2/ > 1d4 1d2 x 1d2
! M + Q = 7.
2/ = 1d4 1d3 x 1d1
2/ 0 1d4 1d3 x 1d2
2/ < 1d4 1d3 x 1d2
2/ > 1d4 1d3 x 1d1
! M + Q = 11.
2/ = 1d8 1d3 x 1d5
2/ 0 1d8 1d3 x 1d6
2/ < 1d8 1d3 x 1d6
2/ > 1d8 1d3 x 1d5
2/ = 1d9 1d2 x 1d7
2/ 0 1d9 1d2 x 1d8
2/ < 1d9 1d2 x 1d8
2/ > 1d9 1d2 x 1d7
! M + Q = 12.
2/ = 1d8 1d4 x 1d4
2/ 0 1d8 1d4 x 1d5
2/ < 1d8 1d4 x 1d5
2/ > 1d8 1d4 x 1d4
! M + Q = 14.
2/ = 1d9 1d5 x 1d4
2/ 0 1d9 1d5 x 1d5
2/ < 1d9 1d5 x 1d5
2/ > 1d9 1d5 x 1d4
! (1 - Mulp-) / (1 - Nulp-) -->
! Case M < Q: (1 - (M/2)ulp+) *
! (1 + (Q/2)ulp+ +
!  ((Q/2)ulp+)~2 + tiny) -->
! 1 + ((Q-M)/2)ulp+ +
!  (NN-MN)/4(ulp+)~2 + tiny -->
! 1 + (Q-M)/2ulp+ + tiny.
! M + Q = 3.
2/ = 1d1 1d2 x 1i1
2/ 0 1d1 1d2 x 1
2/ < 1d1 1d2 x 1
2/ > 1d1 1d2 x 1i1
! M + Q = 4.
2/ = 1d1 1d3 x 1i1
2/ 0 1d1 1d3 x 1i1
2/ < 1d1 1d3 x 1i1
2/ > 1d1 1d3 x 1i2
! M + Q = 5.
2/ = 1d2 1d3 x 1i1
2/ 0 1d2 1d3 x 1
2/ < 1d2 1d3 x 1
2/ > 1d2 1d3 x 1i1
2/ = 1d1 1d4 x 1i2
2/ 0 1d1 1d4 x 1i1
2/ < 1d1 1d4 x 1i1
2/ > 1d1 1d4 x 1i2
! M + Q = 6.
2/ = 1d2 1d4 x 1i1
2/ 0 1d2 1d4 x 1i1
2/ < 1d2 1d4 x 1i1
2/ > 1d2 1d4 x 1i2
! M + Q = 7.
2/ = 1d3 1d4 x 1i1
2/ 0 1d3 1d4 x 1
2/ < 1d3 1d4 x 1
2/ > 1d3 1d4 x 1i1
! M + Q = 8.
2/ = 1d1 1d7 x 1i3
2/ 0 1d1 1d7 x 1i3
2/ < 1d1 1d7 x 1i3
2/ > 1d1 1d7 x 1i4
! M + Q = 9.
2/ = 1d2 1d7 x 1i3
2/ 0 1d2 1d7 x 1i2
2/ < 1d2 1d7 x 1i2
2/ > 1d2 1d7 x 1i3
! M + Q = 10.
2/ = 1d3 1d7 x 1i2
```

```
2/ 0 1d3 1d7 x 1i2
2/ < 1d3 1d7 x 1i2
2/ > 1d3 1d7 x 1i3
! M + Q = 11.
2/ = 1d4 1d7 x 1i2
2/ 0 1d4 1d7 x 1i1
2/ < 1d4 1d7 x 1i1
2/ > 1d4 1d7 x 1i2
! M + Q = 12.
2/ = 1d5 1d7 x 1i1
2/ 0 1d5 1d7 x 1i1
2/ < 1d5 1d7 x 1i1
2/ > 1d5 1d7 x 1i2
! M + Q = 13.
2/ = 1d6 1d7 x 1i1
2/ 0 1d6 1d7 x 1
2/ < 1d6 1d7 x 1
2/ > 1d6 1d7 x 1i1
! (1 + Mulp+) / (1 - Nulp-) -->
! (1 + Mulp+) * (1 + (Q/2)ulp+ +
! ((Q/2)ulp+)^2 + tiny) -->
! 1 + (M + Q/2)ulp+ + tiny.
! M + Q = 2.
2/ = 1i1 1d1 x 1i2
2/ 0 1i1 1d1 x 1i1
2/ < 1i1 1d1 x 1i1
2/ > 1i1 1d1 x 1i2
! M + Q = 3.
2/ = 1i1 1d2 x 1i2
2/ 0 1i1 1d2 x 1i2
2/ < 1i1 1d2 x 1i2
2/ > 1i1 1d2 x 1i3
2/ = 1i2 1d1 x 1i3
2/ 0 1i2 1d1 x 1i2
2/ < 1i2 1d1 x 1i2
2/ > 1i2 1d1 x 1i3
! M + Q = 4.
2/ = 1i1 1d3 x 1i3
2/ 0 1i1 1d3 x 1i2
2/ < 1i1 1d3 x 1i2
2/ > 1i1 1d3 x 1i3
2/ = 1i3 1d1 x 1i4
2/ 0 1i3 1d1 x 1i3
2/ < 1i3 1d1 x 1i3
2/ > 1i3 1d1 x 1i4
2/ = 1i2 1d2 x 1i3
2/ 0 1i2 1d2 x 1i3
2/ < 1i2 1d2 x 1i3
2/ > 1i2 1d2 x 1i4
! M + Q = 5.
2/ = 1i3 1d2 x 1i4
2/ 0 1i3 1d2 x 1i4
2/ < 1i3 1d2 x 1i4
2/ > 1i3 1d2 x 1i5
2/ = 1i2 1d3 x 1i4
2/ 0 1i2 1d3 x 1i3
2/ < 1i2 1d3 x 1i3
2/ > 1i2 1d3 x 1i4
! M + Q = 6.
2/ = 1i3 1d3 x 1i5
2/ 0 1i3 1d3 x 1i4
2/ < 1i3 1d3 x 1i4
2/ > 1i3 1d3 x 1i5
2/ = 1i1 1d5 x 1i4
2/ 0 1i1 1d5 x 1i3
2/ < 1i1 1d5 x 1i3
2/ > 1i1 1d5 x 1i4
2/ = 1i5 1d1 x 1i6
2/ 0 1i5 1d1 x 1i5
2/ < 1i5 1d1 x 1i5
2/ > 1i5 1d1 x 1i6
2/ = 1i2 1d4 x 1i4
2/ 0 1i2 1d4 x 1i4
2/ < 1i2 1d4 x 1i4
2/ > 1i2 1d4 x 1i5
2/ = 1i4 1d2 x 1i5
2/ 0 1i4 1d2 x 1i5
2/ < 1i4 1d2 x 1i5
2/ > 1i4 1d2 x 1i6
! (1 - Mulp-) / (1 + Nulp+) -->
! (1 - Mulp-) * (1 - 2Nulp- +
! (2Nulp-)^2 - tiny) -->
! 1 - (M + 2N)ulp- + tiny.
! M + Q = 2.
2/ = 1d1 1i1 x 1d3
2/ 0 1d1 1i1 x 1d3
2/ < 1d1 1i1 x 1d3
2/ > 1d1 1i1 x 1d2
! M + Q = 3.
2/ = 1d2 1i1 x 1d4
2/ 0 1d2 1i1 x 1d4
2/ < 1d2 1i1 x 1d4
2/ > 1d2 1i1 x 1d3
2/ = 1d1 1i2 x 1d5
2/ 0 1d1 1i2 x 1d5
2/ < 1d1 1i2 x 1d5
2/ > 1d1 1i2 x 1d4
! M + Q = 4.
2/ = 1d3 1i1 x 1d5
2/ 0 1d3 1i1 x 1d5
2/ < 1d3 1i1 x 1d5
2/ > 1d3 1i1 x 1d4
2/ = 1d1 1i3 x 1d7
2/ 0 1d1 1i3 x 1d7
2/ < 1d1 1i3 x 1d7
2/ > 1d1 1i3 x 1d6
2/ = 1d2 1i2 x 1d6
2/ 0 1d2 1i2 x 1d6
2/ < 1d2 1i2 x 1d6
2/ > 1d2 1i2 x 1d5
! M + Q = 5.
2/ = 1d4 1i1 x 1d6
2/ 0 1d4 1i1 x 1d6
2/ < 1d4 1i1 x 1d6
2/ > 1d4 1i1 x 1d5
2/ = 1d1 1i4 x 1d9
2/ 0 1d1 1i4 x 1d9
2/ < 1d1 1i4 x 1d9
2/ > 1d1 1i4 x 1d8
2/ = 1d3 1i2 x 1d7
2/ 0 1d3 1i2 x 1d7
2/ < 1d3 1i2 x 1d7
2/ > 1d3 1i2 x 1d6
2/ = 1d2 1i3 x 1d8
2/ 0 1d2 1i3 x 1d8
2/ < 1d2 1i3 x 1d8
2/ > 1d2 1i3 x 1d7
! Nan operands.
```

```
2/ ALL Q 0 OK Q
2/ ALL Q -0 OK Q
2/ ALL 0 Q OK Q
2/ ALL -0 Q OK Q
2/ ALL Q 1 OK Q
2/ ALL Q -1 OK Q
2/ ALL 1 Q OK Q
2/ ALL -1 Q OK Q
2/ ALL Ed1 Q OK Q
2/ ALL -Ed1 Q OK Q
2/ ALL Q Ed1 OK Q
2/ ALL Q -Ed1 OK Q
2/ ALL Q Oi1 OK Q
2/ ALL Q -Oi1 OK Q
2/ ALL Oi1 Q OK Q
2/ ALL -Oi1 Q OK Q
2/ ALL Q Hd1 OK Q
2/ ALL Q -Hd1 OK Q
2/ ALL Hd1 Q OK Q
2/ ALL -Hd1 Q OK Q
2/ ALL Q H OK Q
2/ ALL Q -H OK Q
2/ ALL H Q OK Q
2/ ALL -H Q OK Q
2/ ALL Q Q OK Q
2/ ALL S 0 i Q
2/ ALL S -0 i Q
2/ ALL 0 S i Q
2/ ALL -0 S i Q
2/ ALL S 1 i Q
2/ ALL S -1 i Q
2/ ALL 1 S i Q
2/ ALL -1 S i Q
2/ ALL Ed1 S i Q
2/ ALL -Ed1 S i Q
2/ ALL S Ed1 i Q
2/ ALL S -Ed1 i Q
2/ ALL S Oi1 i Q
2/ ALL S -Oi1 i Q
2/ ALL Oi1 S i Q
2/ ALL -Oi1 S i Q
2/ ALL S Hd1 i Q
2/ ALL S -Hd1 i Q
2/ ALL Hd1 S i Q
2/ ALL -Hd1 S i Q
2/ ALL S H i Q
2/ ALL S -H i Q
2/ ALL H S i Q
2/ ALL -H S i Q
2/ ALL Q S i Q
2/ ALL S Q i Q
2/ ALL S S i Q
```

```
! Middle-range numbers.
2% ALL 1 2 OK 1
2% ALL 1 -2 OK 1
2% ALL -1 2 OK -1
2% ALL -1 -2 OK -1
2% ALL 3 2 OK -1
2% ALL 3 -2 OK -1
2% ALL -3 2 OK 1
2% ALL -3 -2 OK 1
2% ALL 2 2 OK 0
2% ALL 2 -2 OK 0
2% ALL -2 2 OK -0
2% ALL -2 -2 OK -0
2% ALL 1i1 2 OK -1d2
2% ALL 3d1 2 OK 1d4
2% ALL 1 4 OK 1
2% ALL 2 4 OK 2
2% ALL 3 4 OK -1
2% ALL 4 4 OK 0
2% ALL 5 4 OK 1
2% ALL 6 4 OK -2
2% ALL 7 4 OK -1
2% ALL 8 4 OK 0
2% ALL 0 1m1 OK 0
2% ALL 1m3 1m1 OK 1m3
2% ALL 3m3 1m1 OK -1m3
2% ALL 5m3 1m1 OK 1m3
! Step across jump.
2% ALL 2i1 4 OK -2d2
2% ALL 2i1 -4 OK -2d2
2% ALL -2i1 4 OK 2d2
2% ALL -2i1 -4 OK 2d2
2% ALL 2i8 4 OK -2d8d8
2% ALL 6d1 4 OK 2d4
2% ALL 6d1 -4 OK 2d4
2% ALL -6d1 4 OK -2d4
2% ALL -6d1 -4 OK -2d4
2% ALL 6d8 4 OK 2d8d8d8d8
2% ALL 1m2 1m1 OK 1m2
2% ALL 1i1m2 1m1 OK -1d2m2
! (1+x)/(1+y), x,y<<1.
2% ALL 1i1 1i5 OK -1u4
2% ALL 1i1 -1i5 OK -1u4
2% ALL -1i1 1i5 OK 1u4
2% ALL -1i1 -1i5 OK 1u4
2% ALL 1i2 1i5 OK -1u3
2% ALL 1i3 1i5 OK -1u2
2% ALL 1i4 1i5 OK -1u1
2% ALL 1i6 1i5 OK 1u1
2% ALL 3d1 3 OK -3u1
2% ALL 3d1 -3 OK -3u1
2% ALL -3d1 3 OK 3u1
2% ALL -3d1 -3 OK 3u1
2% ALL 2d1 2 OK -1u1
2% ALL 1i1 1d2 OK 1u2
2% ALL 1 1d2 OK 1u1
2% ALL 1d4 1d2 OK -1u1
2% ALL 1d1 2d1 OK 1d1
2% ALL 1 2d1 OK -1d2
! Large numbers.
2% ALL Hm1i1 Hm1d2 OK Hm1u2
2% ALL Hm1 Hm1d2 OK Hm1u1
2% ALL Hm1d4 Hm1d2 OK -Hm1u1
2% ALL Hm1d1 Hd1 OK Hm1d1
2% ALL Hm1 Hd1 OK -Hm1d2
2% ALL Hm2 Hm1 OK Hm2
2% ALL Hd1 Hd2 OK Hd1u1
2% ALL Hd1 -Hd2 OK Hd1u1
2% ALL -Hd1 Hd2 OK -Hd1u1
2% ALL -Hd1 -Hd2 OK -Hd1u1
2% ALL Hm1u1 Hm1u4 OK Hm1u1
2% ALL Hd1 Hm1 OK -Hm1u1
2% ALL Hm1i3 Hm1i5 OK -Hm1u2
2% ALL Hm1i4 Hm1i5 OK -Hm1u1
2% ALL Hm1i6 Hm1i5 OK Hm1u1
! Large and small numbers.
2% ALL Hd1 0i1 OK 0
2% ALL Hd1 -0i1 OK 0
2% ALL -Hd1 0i1 OK -0
2% ALL -Hd1 -0i1 OK -0
2% ALL Hd1 Eu1 OK 0
2% ALL Hd1 Ep1d1 OK 0
2% ALL Hd1 E OK 0
2% ALL Hm1d1 Hm1 OK -Hm2u1
2% ALL Hm1d1 -Hm1 OK -Hm2u1
2% ALL -Hm1d1 Hm1 OK Hm2u1
2% ALL -Hm1d1 -Hm1 OK Hm2u1
! Small numbers.
2% ALL 0i1 0i4 OK 0i1
2% ALL 0i1 -0i4 OK 0i1
2% ALL -0i1 0i4 OK -0i1
2% ALL -0i1 -0i4 OK -0i1
2% ALL 0i2 0i4 OK 0i2
2% ALL 0i3 0i4 OK -0i1
2% ALL 0i3 -0i4 OK -0i1
2% ALL -0i3 0i4 OK 0i1
2% ALL -0i3 -0i4 OK 0i1
2% ALL 0i4 0i4 OK 0
2% ALL 0i4 -0i4 OK 0
2% ALL -0i4 -0i4 OK -0
2% ALL -0i4 0i4 OK -0
2% ALL Ep8d1 Ep8 OK -Ep8u1
2% ALL Ei1 Ed2 OK Eu3
2% ALL E Ed2 OK Eu2
2% ALL Ed4 Ed2 OK -Eu2
2% ALL Ed4 -Ed2 OK -Eu2
2% ALL -Ed4 Ed2 OK Eu2
2% ALL -Ed4 -Ed2 OK Eu2
2% ALL Ed1 Ep1d1 OK Ed1
2% ALL E Ep1d1 OK -Ed1
2% ALL Ei3 Ei5 OK -Eu2
2% ALL Ei4 Ei5 OK -Eu1
2% ALL Ei6 Ei5 OK Eu1
2% ALL Ep1d1 Ep1 OK -Eu1
! Special case: invalid operations
! delivering NaNs.
2% ALL 0 0 i Q
2% ALL 0 -0 i Q
2% ALL -0 0 i Q
2% ALL -0 -0 i Q
2% ALL 1 0 i Q
2% ALL 1d1 0 i Q
2% ALL Hd1 0 i Q
2% ALL Hd1 -0 i Q
2% ALL -Hd1 0 i Q
2% ALL -Hd1 -0 i Q
2% ALL Ed1 0 i Q
2% ALL Ed1 -0 i Q
```

```
2% ALL -Ed1 0 i Q
2% ALL -Ed1 -0 i Q
2% ALL 0i1 0 i Q
2% ALL H 0 i Q
2% ALL H -0 i Q
2% ALL -H 0 i Q
2% ALL -H -0 i Q
2% ALL H 1 i Q
2% ALL H Hd1 i Q
2% ALL H -Hd1 i Q
2% ALL -H Hd1 i Q
2% ALL -H -Hd1 i Q
2% ALL H Ed1 i Q
2% ALL H 0i1 i Q
2% ALL H H i Q
! 0 rem y = 0,  y a number <> 0.
2% ALL 0 1 OK 0
2% ALL 0 -1 OK 0
2% ALL -0 1 OK -0
2% ALL -0 -1 OK -0
2% ALL 0 1d1 OK 0
2% ALL 0 Hd1 OK 0
2% ALL 0 Ed1 OK 0
2% ALL 0 0i1 OK 0
2% ALL 0 -0i1 OK 0
2% ALL -0 0i1 OK -0
2% ALL -0 -0i1 OK -0
2% ALL 0 H OK 0
2% ALL 0 -H OK 0
! x rem INF = x,  x a number <> 0.
2% ALL 1 H OK 1
2% ALL 1 -H OK 1
2% ALL -1 H OK -1
2% ALL -1 -H OK -1
2% ALL 1d1 H OK 1d1
2% ALL Hd1 H OK Hd1
2% ALL Hd1 -H OK Hd1
2% ALL -Hd1 H OK -Hd1
2% ALL -Hd1 -H OK -Hd1
2% ALL Ed1 H OK Ed1
2% ALL 0i1 H OK 0i1
2% ALL 0i1 -H OK 0i1
2% ALL -0i1 H OK -0i1
2% ALL -0i1 -H OK -0i1
! Vectors based on
! (x + 1) | (x^n + 1) for n odd -
! for significands with even
! numbers of bits.
2% s Hm1i1 Hm1u3 OK 0
2% s Hm1i2 Hm1u3 OK Hm1u1
2% s Hm1i3 Hm1u3 OK -Hm1u1
2% s Hm1i1 3 OK 0
2% s Hm1i1 0i3 OK 0
2% s Hm1 Hm1u3 OK -Hm1u1
2% s Hm1d2 Hm1u3 OK Hm1u1
2% s Ei1 Eu3 OK 0
2% s E Eu3 OK -0i1
2% s Ed1 Eu3 OK 0i1
2% s Ei1 0i3 OK 0
2% s Ei2 Eu3 OK Eu1
2% s Ei3 Eu3 OK -Eu1
2% s Hm1i1 -Hm1u3 OK 0
2% s Hm1i2 -Hm1u3 OK Hm1u1
2% s Hm1i3 -Hm1u3 OK -Hm1u1
2% s Hm1i1 -3 OK 0
2% s Hm1i1 -0i3 OK 0
2% s Hm1 -Hm1u3 OK -Hm1u1
2% s Hm1d2 -Hm1u3 OK Hm1u1
2% s Ei1 -0i3 OK 0
2% s E -Eu3 OK -Eu1
2% s Ed1 -Eu3 OK Eu1
2% s Ei1 -Eu3 OK 0
2% s Ei2 -Eu3 OK Eu1
2% s Ei3 -Eu3 OK -Eu1
2% s -Hm1i1 Hm1u3 OK -0
2% s -Hm1i2 Hm1u3 OK -Hm1u1
2% s -Hm1i3 Hm1u3 OK Hm1u1
2% s -Hm1i1 3 OK -0
2% s -Hm1i1 0i3 OK -0
2% s -Hm1 Hm1u3 OK Hm1u1
2% s -Hm1d2 Hm1u3 OK -Hm1u1
2% s -Ei1 0i3 OK -0
2% s -E Eu3 OK Eu1
2% s -Ed1 Eu3 OK -Eu1
2% s -Ei1 Eu3 OK -0
2% s -Ei2 Eu3 OK -Eu1
2% s -Ei3 Eu3 OK Eu1
2% s -Hm1i1 -Hm1u3 OK -0
2% s -Hm1i2 -Hm1u3 OK -Hm1u1
2% s -Hm1i3 -Hm1u3 OK Hm1u1
2% s -Hm1i1 -3 OK -0
2% s -Hm1i1 -0i3 OK -0
2% s -Hm1 -Hm1u3 OK Hm1u1
2% s -Hm1d2 -Hm1u3 OK -Hm1u1
2% s -Ei1 -0i3 OK -0
2% s -E -Eu3 OK Eu1
2% s -Ed1 -Eu3 OK -Eu1
2% s -Ei1 -Eu3 OK -0
2% s -Ei2 -Eu3 OK -Eu1
2% s -Ei3 -Eu3 OK Eu1
! Vectors based on
! (x + 1) | (x^n + 1) for n odd;
! for significands with
! odd numbers of bits.
2% d Hm1d2 Hm1u3 OK 0
2% d Hm1i3 Hm1u3 OK Hm1u1
2% d Hm1i4 Hm1u3 OK -Hm1u1
2% d Hm1i2 3 OK 0
2% d Hm1i2 0i3 OK 0
2% d Hm1d4 Hm1u3 OK -Hm1u1
2% d Hm1 Hm1u3 OK Hm1u1
2% d Ed1 Eu3 OK 0
2% d Ei1 Eu3 OK -0i1
2% d E Eu3 OK 0i1
2% d Ei2 0i3 OK 0
2% d Ei3 Eu3 OK Eu1
2% d Ei4 Eu3 OK -Eu1
2% d Hm1d2 -Hm1u3 OK 0
2% d Hm1i3 -Hm1u3 OK Hm1u1
2% d Hm1i4 -Hm1u3 OK -Hm1u1
2% d Hm1i2 -3 OK 0
2% d Hm1i2 -0i3 OK 0
2% d Hm1d4 -Hm1u3 OK -Hm1u1
2% d Hm1 -Hm1u3 OK Hm1u1
2% d Ed1 -0i3 OK 0
2% d Ei1 -Eu3 OK -Eu1
2% d E -Eu3 OK Eu1
2% d Ei2 -Eu3 OK 0
```

```
2% d Ei3 -Eu3 OK Eu1
2% d Ei4 -Eu3 OK -Eu1
2% d -Hm1d2 Hm1u3 OK -0
2% d -Hm1i3 Hm1u3 OK -Hm1u1
2% d -Hm1i4 Hm1u3 OK Hm1u1
2% d -Hm1i2 3 OK -0
2% d -Hm1i2 0i3 OK -0
2% d -Hm1d4 Hm1u3 OK Hm1u1
2% d -Hm1 Hm1u3 OK -Hm1u1
2% d -Ed1 0i3 OK -0
2% d -Ei1 Eu3 OK Eu1
2% d -E Eu3 OK -Eu1
2% d -Ei2 Eu3 OK -0
2% d -Ei3 Eu3 OK -Eu1
2% d -Ei4 Eu3 OK Eu1
2% d -Hm1d2 -Hm1u3 OK -0
2% d -Hm1i3 -Hm1u3 OK -Hm1u1
2% d -Hm1i4 -Hm1u3 OK Hm1u1
2% d -Hm1i2 -3 OK -0
2% d -Hm1i2 -0i3 OK -0
2% d -Hm1d4 -Hm1u3 OK Hm1u1
2% d -Hm1 -Hm1u3 OK -Hm1u1
2% d -Ei2 -0i3 OK -0
2% d -Ei1 -Eu3 OK Eu1
2% d -E -Eu3 OK -Eu1
2% d -Ei2 -Eu3 OK -0
2% d -Ei3 -Eu3 OK -Eu1
2% d -Ei4 -Eu3 OK Eu1
! NaN operands.
2% ALL Q 0 OK Q
2% ALL Q -0 OK Q
2% ALL 0 Q OK Q
2% ALL -0 Q OK Q
2% ALL Q 1 OK Q
2% ALL Q -1 OK Q
2% ALL 1 Q OK Q
2% ALL -1 Q OK Q
2% ALL Ed1 Q OK Q
2% ALL -Ed1 Q OK Q
2% ALL Q Ed1 OK Q
2% ALL Q -Ed1 OK Q
2% ALL Q 0i1 OK Q
2% ALL Q -0i1 OK Q
2% ALL 0i1 Q OK Q
2% ALL -0i1 Q OK Q
2% ALL Q Hd1 OK Q
2% ALL Q -Hd1 OK Q
2% ALL Hd1 Q OK Q
2% ALL -Hd1 Q OK Q
2% ALL Q H OK Q
2% ALL Q -H OK Q
2% ALL H Q OK Q
2% ALL -H Q OK Q
2% ALL Q Q OK Q
2% ALL S 0 i Q
2% ALL S -0 i Q
2% ALL 0 S i Q
2% ALL -0 S i Q
2% ALL S 1 i Q
2% ALL S -1 i Q
2% ALL 1 S i Q
2% ALL -1 S i Q
2% ALL Ed1 S i Q
2% ALL -Ed1 S i Q
2% ALL S Ed1 i Q
2% ALL S -Ed1 i Q
2% ALL S 0i1 i Q
2% ALL S -0i1 i Q
2% ALL 0i1 S i Q
2% ALL -0i1 S i Q
2% ALL S Hd1 i Q
2% ALL S -Hd1 i Q
2% ALL Hd1 S i Q
2% ALL -Hd1 S i Q
2% ALL S H i Q
2% ALL S -H i Q
2% ALL H S i Q
2% ALL -H S i Q
2% ALL Q S i Q
2% ALL S Q i Q
2% ALL S S i Q
```

```
! First some easy integer cases.
2C ALL 1 1 OK =
2C ALL 1 2 OK <
2C ALL 2 1 OK >
2C ALL 2 2 OK =
2C ALL 2 -2 OK >
2C ALL 5 -5 OK >
2C ALL 1 7 OK <
2C ALL 5 -1 OK >
2C ALL 2 -5 OK >
2C ALL 5 -0 OK >
2C ALL 5 +0 OK >
! Infinity vs Infinity.
2C ALL H H OK = always equal
2C ALL -H -H OK = always equal
2C ALL H -H OK >
2C ALL -H H OK <
! Infinity vs huge.
2C ALL H Hm1 OK >
2C ALL H -Hm1 OK >
2C ALL -H Hm1 OK <
2C ALL -H -Hm1 OK <
2C ALL H Hd1 OK >
2C ALL H -Hd1 OK >
2C ALL -H Hd1 OK <
2C ALL -H -Hd1 OK <
2C ALL Hm1 H OK <
2C ALL Hm1 -H OK >
2C ALL -Hm1 H OK <
2C ALL -Hm1 -H OK >
! Infinity vs 0.
2C ALL H 0 OK >
2C ALL H -0 OK >
2C ALL -H 0 OK <
2C ALL -H -0 OK <
2C ALL 0 H OK <
2C ALL -0 H OK <
2C ALL 0 -H OK >
2C ALL -0 -H OK >
! Infinity vs denormalized.
2C ALL H Ed1 OK >
2C ALL -H Ed1 OK <
2C ALL H -Ed1 OK >
2C ALL -H -Ed1 OK <
2C ALL H 0i1 OK >
2C ALL -H 0i1 OK <
2C ALL H -0i1 OK >
2C ALL -H -0i1 OK <
2C ALL Ed1 H OK <
2C ALL Ed1 -H OK >
2C ALL -Ed1 H OK <
2C ALL -Ed1 -H OK >
! Zero vs finite -- watch that sign
! of 0 is meaningless.
2C ALL 0 Hm1 OK <
2C ALL -0 Hm1 OK <
2C ALL -Hm1 0 OK <
2C ALL -Hm1 -0 OK <
2C ALL 1 -0 OK >
2C ALL -1 -0 OK <
2C ALL 0 1 OK <
2C ALL -0 -1 OK >
! Zero vs denormalized.
2C ALL 0 Ed1 OK <
2C ALL -0 Ed1 OK <
2C ALL 0 -Ed1 OK >
2C ALL -0 -Ed1 OK >
2C ALL 0 0i1 OK <
2C ALL -0 0i1 OK <
2C ALL 0 -0i1 OK >
2C ALL -0 -0i1 OK >
2C ALL Ed1 0 OK >
2C ALL Ed1 -0 OK >
2C ALL -Ed1 0 OK <
2C ALL -Ed1 -0 OK <
! Zero vs tiny -- just in case.
2C ALL -0 -E OK >
2C ALL E 0 OK >
2C ALL 0 -E OK >
2C ALL -E 0 OK <
! Zero vs Zero -- watch signs
! and rounding modes.
2C ALL 0 -0 OK =
2C ALL -0 0 OK =
2C ALL 0 -0 OK =
2C ALL -0 0 OK =
! Big cancellations.
2C ALL Hm1 Hm1 OK =
2C ALL Hm1 Hm1 OK =
2C ALL -Hm1 -Hm1 OK =
2C ALL -Hm1 -Hm1 OK =
2C ALL Hm1d2 Hm1d2 OK =
2C ALL -Hm1d2 -Hm1d2 OK =
2C ALL Hd1 Hd1 OK =
2C ALL Hd1 Hd1 OK =
2C ALL -Hd1 -Hd1 OK =
2C ALL -Hd1 -Hd1 OK =
! Medium cancellations.
2C ALL 1 1 OK =
2C ALL 1m1 1m1 OK =
2C ALL 3 3 OK =
2C ALL E E OK =
2C ALL Hm2 Hm2 OK =
! Tiny cancellations -- might
! have underflowed.
2C ALL Ed1 Ed1 OK =
2C ALL -Ed1 -Ed1 OK =
2C ALL 0i4 0i4 OK =
2C ALL -0i4 -0i4 OK =
2C ALL 0i1 0i1 OK =
2C ALL -0i1 -0i1 OK =
! Doublings.
2C ALL Hm1 -Hm1 OK >
2C ALL -Hm1d2 Hm1d2 OK <
2C ALL 1 -1 OK >
2C ALL -3 3 OK <
2C ALL E -E OK >
2C ALL -E E OK <
2C ALL Ed4 -Ed4 OK >
2C ALL -Ed1 Ed1 OK <
2C ALL 0i1 -0i1 OK >
2C ALL -0i1 0i1 OK <
! Cancellation with diff in LSB
! Difference is in last place of
! larger number.
! Medium numbers...
2C ALL 1i1 1 OK >
2C ALL -1i1 -1 OK <
```

```
2C ALL 1i1 1i2 OK <
2C ALL -1i1 -1i2 OK >
2C ALL 2 2i1 OK <
2C ALL -2 -2i1 OK >
2C ALL 2i4 2i3 OK >
2C ALL -2i4 -2i3 OK <
2C ALL 4d1 4d2 OK >
2C ALL -4d1 -4d2 OK <
2C ALL 2d4 2d3 OK <
2C ALL -2d4 -2d3 OK >
! Huge numbers...
2C ALL Hm1i1 Hm1 OK >
2C ALL -Hm1i1 -Hm1 OK <
2C ALL Hm1i1 Hm1i2 OK <
2C ALL -Hm1i1 -Hm1i2 OK >
2C ALL Hm2 Hm2i1 OK <
2C ALL -Hm2 -Hm2i1 OK >
2C ALL Hm2i4 Hm2i3 OK >
2C ALL -Hm2i4 -Hm2i3 OK <
2C ALL Hm2d1 Hm2d2 OK >
2C ALL -Hm2d1 -Hm2d2 OK <
2C ALL -Hd2 -Hd1 OK >
2C ALL Hd2 Hd1 OK <
! Tiny numbers...
2C ALL -Ei1 -E OK <
2C ALL Ei1 E OK >
2C ALL -Ed1 -E OK >
2C ALL Ed1 E OK <
2C ALL Ei1 Ei2 OK <
2C ALL -Ei1 -Ei2 OK >
2C ALL Ed1 Ed2 OK >
2C ALL -Ed1 -Ed2 OK <
2C ALL Ed3 Ed2 OK <
2C ALL -Ed3 -Ed2 OK >
2C ALL 0i2 0i1 OK >
2C ALL -0i2 -0i1 OK <
2C ALL 0i3 0i2 OK >
2C ALL -0i3 -0i2 OK <
! Normalize from round bit -- set up
! tests so that operands have
! exponents differing by 1 unit.
! Medium numbers...
2C ALL 2 2d1 OK >
2C ALL -2 -2d1 OK <
2C ALL -2d1 -2 OK >
2C ALL 2d1 2 OK <
2C ALL 4i1 4d1 OK >
2C ALL -4i1 -4d1 OK <
2C ALL 4d1 4i2 OK <
2C ALL -4d1 -4i2 OK >
2C ALL 2i1 1i1 OK >
2C ALL -2i1 -1i1 OK <
2C ALL 2i2 1i1 OK >
2C ALL -2i2 -1i1 OK <
2C ALL 2i2 1i3 OK >
2C ALL -2i2 -1i3 OK <
! Huge numbers...
2C ALL Hm2 Hm2d1 OK >
2C ALL -Hm2 -Hm2d1 OK <
2C ALL -Hm1d1 -Hm1 OK >
2C ALL Hm1d1 Hm1 OK <
2C ALL Hm4i1 Hm4d1 OK >
2C ALL -Hm4i1 -Hm4d1 OK <
2C ALL Hm2d1 Hm2i2 OK <
2C ALL -Hm2d1 -Hm2i2 OK >
2C ALL Hm2i1 Hm1i1 OK <
2C ALL -Hm2i1 -Hm1i1 OK >
2C ALL Hm1i2 Hm2i1 OK >
2C ALL -Hm1i2 -Hm2i1 OK <
2C ALL Hm2i2 Hm3i3 OK >
2C ALL -Hm2i2 -Hm3i3 OK <
! Tiny numbers...
2C ALL Ep1 Ep1d1 OK >
2C ALL -Ep1 -Ep1d1 OK <
2C ALL -Ep1d1 -Ep1 OK >
2C ALL Ep1d1 Ep1 OK <
2C ALL Ep1i1 Ep1d1 OK >
2C ALL -Ep1i1 -Ep1d1 OK <
2C ALL Ep2 Ep2d1 OK >
2C ALL -Ep2 -Ep2d1 OK <
2C ALL -Ep2d1 -Ep2 OK >
2C ALL Ep2d1 Ep2 OK <
2C ALL Ep2i1 Ep2d1 OK >
2C ALL -Ep2i1 -Ep2d1 OK <
2C ALL Ep1d1 Ep1i2 OK <
2C ALL -Ep1d1 -Ep1i2 OK >
2C ALL Ep1d1 Ep1i4 OK <
2C ALL -Ep1d1 -Ep1i4 OK >
2C ALL Ep1i1 Ei1 OK >
2C ALL -Ep1i1 -Ei1 OK <
2C ALL Ep1i2 Ei1 OK >
2C ALL -Ep1i2 -Ei1 OK <
2C ALL Ep2i2 Ep1i3 OK >
2C ALL -Ep2i2 -Ep1i3 OK <
!
! Add magnitude cases where one operand
! is off in sticky -- rounding
! perhaps to an overflow.
! Huge vs medium
2C ALL Hm1 1 OK >
2C ALL -Hm1 -1 OK <
2C ALL Hm1d1 -1 OK >
2C ALL Hm1d1 1 OK >
2C ALL -Hm1d1 1 OK <
2C ALL -Hm1d1 -1 OK <
2C ALL Hd1 1 OK >
2C ALL Hd1 -1 OK >
2C ALL -Hd1 1 OK <
2C ALL -Hd1 -1 OK <
2C ALL Hd2 -1 OK >
2C ALL Hd2 1 OK >
2C ALL -Hd2 1 OK <
2C ALL -Hd2 -1 OK <
! Huge vs tiny.
2C ALL 0i1 Hm1 OK <
2C ALL 0i1 -Hm1 OK >
2C ALL -0i1 Hm1 OK <
2C ALL -0i1 -Hm1 OK >
2C ALL 0i1 Hm1d1 OK <
2C ALL 0i1 -Hm1d1 OK >
2C ALL -0i1 Hm1d1 OK <
2C ALL -0i1 -Hm1d1 OK >
2C ALL 0i1 Hd1 OK <
2C ALL 0i1 -Hd1 OK >
2C ALL -0i1 Hd1 OK <
2C ALL -0i1 -Hd1 OK >
2C ALL 0i1 Hd2 OK <
2C ALL 0i1 -Hd2 OK >
```

```
2C ALL -0i1 Hd2 OK <
2C ALL -0i1 -Hd2 OK >
! Medium vs tiny.
2C ALL 0i1 1 OK <
2C ALL 0i1 -1 OK >
2C ALL -0i1 1 OK <
2C ALL -0i1 -1 OK >
2C ALL 0i1 1d1 OK <
2C ALL 0i1 -1d1 OK >
2C ALL -0i1 1d1 OK <
2C ALL -0i1 -1d1 OK >
2C ALL 0i1 2d1 OK <
2C ALL 0i1 -2d1 OK >
2C ALL -0i1 2d1 OK <
2C ALL -0i1 -2d1 OK >
2C ALL 0i1 2d2 OK <
2C ALL 0i1 -2d2 OK >
2C ALL -0i1 2d2 OK <
2C ALL -0i1 -2d2 OK >
!
! Magnitude subtract when an operand
! is in the sticky bit.
! The interesting cases will arise
! when directed rounding
! forces a nonzero cancellation.
! Huge and medium.
2C ALL Hm1 1 OK >
2C ALL Hm1 -1 OK >
2C ALL -Hm1 1 OK <
2C ALL -Hm1 -1 OK <
2C ALL Hm1d1 1 OK >
2C ALL Hm1d1 -1 OK >
2C ALL -Hm1d1 1 OK <
2C ALL -Hm1d1 -1 OK <
2C ALL Hd1 1 OK >
2C ALL Hd1 -1 OK >
2C ALL -Hd1 1 OK <
2C ALL -Hd1 -1 OK <
2C ALL Hd2 1 OK >
2C ALL Hd2 -1 OK >
2C ALL -Hd2 1 OK <
2C ALL -Hd2 -1 OK <
! Huge and tiny.
2C ALL Hd1 0i1 OK >
2C ALL Hd1 -0i1 OK >
2C ALL -Hd1 0i1 OK <
2C ALL -Hd1 -0i1 OK <
2C ALL 0i3 Hm1 OK <
2C ALL -0i3 Hm1 OK <
2C ALL 0i3 -Hm1 OK >
2C ALL -0i3 -Hm1 OK >
! Medium and tiny.
2C ALL 1d1 0i1 OK >
2C ALL 1d1 -0i1 OK >
2C ALL 2d1 0i1 OK >
2C ALL -2d1 0i1 OK <
2C ALL 0i3 3 OK <
2C ALL -0i3 3 OK <
2C ALL 0i3 5 OK <
2C ALL 0i3 -5 OK >
!
! Add magnitude with difference in
! LSB so, except for denorms,
! round bit is crucial.
! Half-way cases arise.
! Medium cases.
2C ALL 1i1 1 OK >
2C ALL 1i1 -1 OK >
2C ALL -1i1 1 OK <
2C ALL -1i1 -1 OK <
2C ALL -2 2i1 OK <
2C ALL -2 -2i1 OK >
2C ALL 2 -2i1 OK >
2C ALL 2 2i1 OK <
2C ALL 1 1i3 OK <
2C ALL 1 -1i3 OK >
2C ALL -1 1i3 OK <
2C ALL -1 -1i3 OK >
2C ALL -2i1 -2i2 OK >
2C ALL -2i1 2i2 OK <
2C ALL 2i1 -2i2 OK >
2C ALL 2i1 2i2 OK <
! Huge cases.
2C ALL Hd2 Hd1 OK <
2C ALL Hd2 -Hd1 OK >
2C ALL -Hd2 Hd1 OK <
2C ALL -Hd2 -Hd1 OK >
2C ALL Hm1d1 Hm1 OK <
2C ALL Hm1d1 -Hm1 OK >
2C ALL -Hm1d1 Hm1 OK <
2C ALL -Hm1d1 -Hm1 OK >
2C ALL Hm1i1 Hm1 OK >
2C ALL Hm1i1 -Hm1 OK >
2C ALL -Hm1i1 Hm1 OK <
2C ALL -Hm1i1 -Hm1 OK <
2C ALL Hm2i1 Hm2 OK >
2C ALL Hm2i1 -Hm2 OK >
2C ALL -Hm2i1 Hm2 OK <
2C ALL -Hm2i1 -Hm2 OK <
2C ALL Hm1d2 Hm1d1 OK <
2C ALL Hm1d2 -Hm1d1 OK >
2C ALL -Hm1d2 Hm1d1 OK <
2C ALL -Hm1d2 -Hm1d1 OK >
! NaN operands.
2C ALL Q 0 OK ?
2C ALL Q -0 OK ?
2C ALL 0 Q OK ?
2C ALL -0 Q OK ?
2C ALL Q 1 OK ?
2C ALL Q -1 OK ?
2C ALL 1 Q OK ?
2C ALL -1 Q OK ?
2C ALL Ed1 Q OK ?
2C ALL -Ed1 Q OK ?
2C ALL Q Ed1 OK ?
2C ALL Q -Ed1 OK ?
2C ALL Q 0i1 OK ?
2C ALL Q -0i1 OK ?
2C ALL 0i1 Q OK ?
2C ALL -0i1 Q OK ?
2C ALL Q Hd1 OK ?
2C ALL Q -Hd1 OK ?
2C ALL Hd1 Q OK ?
2C ALL -Hd1 Q OK ?
2C ALL Q H OK ?
2C ALL Q -H OK ?
2C ALL H Q OK ?
2C ALL -H Q OK ?
```

```
2C ALL Q Q OK ?
2C ALL S 0 i ?
2C ALL S -0 i ?
2C ALL 0 S i ?
2C ALL -0 S i ?
2C ALL S 1 i ?
2C ALL S -1 i ?
2C ALL 1 S i ?
2C ALL -1 S i ?
2C ALL Ed1 S i ?
2C ALL -Ed1 S i ?
2C ALL S Ed1 i ?
2C ALL S -Ed1 i ?
2C ALL S Oi1 i ?
2C ALL S -Oi1 i ?
2C ALL Oi1 S i ?
2C ALL -Oi1 S i ?
2C ALL S Hd1 i ?
2C ALL S -Hd1 i ?
2C ALL Hd1 S i ?
2C ALL -Hd1 S i ?
2C ALL S H i ?
2C ALL S -H i ?
2C ALL H S i ?
2C ALL -H S i ?
2C ALL Q S i ?
2C ALL S Q i ?
2C ALL S S i ?
```

```
! First a few trivial cases...
2V ALL 1 0 OK 1
2V ALL 4 0 OK 2
2V ALL 9 0 OK 3
2V ALL 1p8 0 OK 1p4
2V ALL 1m8 0 OK 1m4
2V ALL 4p6 0 OK 2p3
2V ALL 4m6 0 OK 2m3
2V ALL 9p8 0 OK 3p4
2V ALL 9m8 0 OK 3m4
2V ALL 9p9p9 0 OK 3p9
2V ALL 9m9m9 0 OK 3m9
! And the usual zero business.
2V ALL +0 0 OK +0
2V ALL -0 0 OK -0
! And tests for infinity.
2V ALL +H 0 OK +H
2V ALL -H 0 i Q
! Case: 2^EVEN * (1 + Nulp+) -->
! 2^(EVEN/2) *
!  (1 + (1/2)Nulp+ -
!  (1/8)(Nulp+)^2 + tiny)
! 1 + 1ulp --> 1 + 0.5ulp - tiny.
2V =0< 1i1 0 x 1
2V > 1i1 0 x 1i1
! 1 + 2ulp --> 1 + 1ulp - tiny.
2V => 1i2 0 x 1i1
2V 0< 1i2 0 x 1
! 1 + 3ulp --> 1 + 1.5ulp - tiny.
2V =0< 1i3 0 x 1i1
2V > 1i3 0 x 1i2
! 1 + 4ulp --> 1 + 2ulp - tiny.
2V => 1i4 0 x 1i2
2V 0< 1i4 0 x 1i1
! (1 + 5ulp) --> 1 + 2.5ulp - ...
2V =0< 1i5 0 x 1i2
2V > 1i5 0 x 1i3
! (1 + 6ulp) --> 1 + 3ulp - ...
2V => 1i6 0 x 1i3
2V 0< 1i6 0 x 1i2
! (1+7ulp) --> 1 + 3.5ulp - ...
2V =0< 1i7 0 x 1i3
2V > 1i7 0 x 1i4
! sqrt(1 - Nulp-) -->
! 1 - (1/2)Nulp- -
!  (1/8)(Nulp-)^2 - tiny
! 1 - 1ulp- -->
! 1 - 0.5ulp- - tiny.
2V =0< 1d1 0 x 1d1
2V > 1d1 0 x 1
! 1 - 2ulp- -->
! 1 - 1ulp- - tiny.
2V => 1d2 0 x 1d1
2V 0< 1d2 0 x 1d2
! 1 - 3ulp- -->
! 1 - 1.5ulp- - tiny.
2V =0< 1d3 0 x 1d2
2V > 1d3 0 x 1d1
! 1 - 4ulp- -->
! 1 - 2ulp- - tiny.
2V => 1d4 0 x 1d2
2V 0< 1d4 0 x 1d3
! 1 - 5ulp- -->
! 1 - 2.5ulp- - tiny.
2V =0< 1d5 0 x 1d3
2V > 1d5 0 x 1d2
! 1 - 6ulp- -->
! 1 - 3ulp- - tiny.
2V => 1d6 0 x 1d3
2V 0< 1d6 0 x 1d4
! 1 - 7ulp- -->
! 1 - 3.5ulp- - tiny.
2V =0< 1d7 0 x 1d4
2V > 1d7 0 x 1d3
! 1 - 8ulp- -->
! 1 - 4ulp- - tiny.
2V => 1d8 0 x 1d4
2V 0< 1d8 0 x 1d5
! 1 - 9ulp- -->
! 1 - 4.5ulp- - tiny.
2V =0< 1d9 0 x 1d5
2V > 1d9 0 x 1d4
! Invalid negative cases.
2V ALL -1 0 i Q
2V ALL -2i2 0 i Q
2V ALL -3i4 0 i Q
2V ALL -4d5 0 i Q
2V ALL -1u1 0 i Q
2V ALL -1u2 0 i Q
2V ALL -1u3 0 i Q
2V ALL -Hm1i2 0 i Q
2V ALL -Hm2i2 0 i Q
2V ALL -Hm1d1 0 i Q
2V ALL -Hm2d4 0 i Q
2V ALL -Ep1i1 0 i Q
2V ALL -Ep1d3 0 i Q
2V ALL -Ep1 0 i Q
2V ALL -Ep1 0 i Q
2V ALL -Ed4 0 i Q
2V ALL -Ed3 0 i Q
2V ALL -Ed2 0 i Q
2V ALL -Ed1 0 i Q
2V ALL -Ed4 0 i Q
2V ALL -Ed3 0 i Q
2V ALL -Ed7 0 i Q
2V ALL -Ed9 0 i Q
2V ALL -0i1 0 i Q
2V ALL -0i1 0 i Q
2V ALL -0i9 0 i Q
2V ALL -0i7 0 i Q
2V ALL -0i5 0 i Q
2V ALL -0i2 0 i Q
! NaN operand.
2V ALL Q 0 OK Q
2V ALL S 0 i Q
```

```
! Exact cases.
2I ALL 1 0 OK 1
2I ALL Hd1 0 OK Hd1
2I ALL -1 0 OK -1
2I ALL -Hd1 0 OK -Hd1
2I ALL 9p9 0 OK 9p9
2I ALL -9p9 0 OK -9p9
2I ALL 0 0 OK 0
2I ALL -0 0 OK -0
2I ALL Hm9 0 OK Hm9
2I ALL Hm9d1 0 OK Hm9d1
2I ALL Hm9d9 0 OK Hm9d9
2I ALL Hm9d9d9 0 OK Hm9d9d9
2I ALL -Hm9 0 OK -Hm9
2I ALL -Hm9d1 0 OK -Hm9d1
2I ALL -Hm9d9 0 OK -Hm9d9
2I ALL -Hm9d9d9 0 OK -Hm9d9d9
! Infinities.
2I ALL H 0 OK H
2I ALL -H 0 OK -H
! Inexact cases.
2I =0< 1i1 0 x 1
2I > 1i1 0 x 2
2I => 1d1 0 x 1
2I 0< 1d1 0 x 0
2I =< -1d1 0 x -1
2I 0> -1d1 0 x -0
2I =0> -1i1 0 x -1
2I < -1i1 0 x -2
2I > E 0 x 1
2I =0< E 0 x 0
2I < -E 0 x -1
2I =0> -E 0 x -0
2I > Ed1 0 x 1
2I =0< Ed1 0 x 0
2I < -Ed1 0 x -1
2I =0> -Ed1 0 x -0
2I =0< 0i1 0 x 0
2I > 0i1 0 x 1
2I =0> -0i1 0 x -0
2I < -0i1 0 x -1
2I > 8i1 0 x 9
2I 0=< 8i1 0 x 8
2I < -8i1 0 x -9
2I 0=> -8i1 0 x -8
2I => 8d1 0 x 8
2I 0< 8d1 0 x 7
2I =< -8d1 0 x -8
2I 0> -8d1 0 x -7
2I => 1p9d8 0 x 1p9
2I =< -1p9d8 0 x -1p9
2I => 1p9p9d1 0 x 1p9p9
2I =< -1p9p9d1 0 x -1p9p9
2I =<0 1p9i8 0 x 1p9
2I =>0 -1p9i8 0 x -1p9
2I =<0 1p9p9i1 0 x 1p9p9
2I =>0 -1p9p9i1 0 x -1p9p9
! Half-way cases.
2I > 1m1 0 x 1
2I =0< 1m1 0 x 0
2I < -1m1 0 x -1
2I =0> -1m1 0 x -0
2I >= 3m1 0 x 2
2I <0 3m1 0 x 1
2I >0 -3m1 0 x -1
2I =< -3m1 0 x -2
2I > 9m1 0 x 5
2I =0< 9m1 0 x 4
2I < -9m1 0 x -5
2I =0> -9m1 0 x -4
2I = 1m1i1 0 x 1
2I = -1m1i1 0 x -1
2I = 3m1d1 0 x 1
2I = -3m1d1 0 x -1
2I = 9m1i1 0 x 5
2I = -9m1i1 0 x -5
! NAN operand.
2I ALL Q 0 OK Q
2I ALL S 0 i Q
```

```
!
! CopySign test vectors:
!
2@ ALL 1 1 OK 1
2@ ALL 1 -1 OK -1
2@ ALL -1 1 OK 1
2@ ALL -1 -1 OK -1
2@ ALL 1 0i1 OK 1
2@ ALL 1 -0i1 OK -1
2@ ALL -1 0i1 OK 1
2@ ALL -1 -0i1 OK -1
2@ ALL 1 Hd1 OK 1
2@ ALL 1 -Hd1 OK -1
2@ ALL -1 Hd1 OK 1
2@ ALL -1 -Hd1 OK -1
2@ ALL 1 H OK 1
2@ ALL 1 -H OK -1
2@ ALL -1 H OK 1
2@ ALL -1 -H OK -1
2@ ALL 1 0 OK 1
2@ ALL 1 -0 OK -1
2@ ALL -1 0 OK 1
2@ ALL -1 -0 OK -1
2@ ALL 0i1 1 OK 0i1
2@ ALL 0i1 -1 OK -0i1
2@ ALL -0i1 1 OK 0i1
2@ ALL -0i1 -1 OK -0i1
2@ ALL 0i1 H OK 0i1
2@ ALL 0i1 -H OK -0i1
2@ ALL -0i1 H OK 0i1
2@ ALL -0i1 -H OK -0i1
2@ ALL 0i1 0 OK 0i1
2@ ALL 0i1 -0 OK -0i1
2@ ALL -0i1 0 OK 0i1
2@ ALL -0i1 -0 OK -0i1
2@ ALL Hd1 E OK Hd1
2@ ALL Hd1 -E OK -Hd1
2@ ALL -Hd1 E OK Hd1
2@ ALL -Hd1 -E OK -Hd1
2@ ALL Hd1 H OK Hd1
2@ ALL Hd1 -H OK -Hd1
2@ ALL -Hd1 H OK Hd1
2@ ALL -Hd1 -H OK -Hd1
2@ ALL Hd1 0 OK Hd1
2@ ALL Hd1 -0 OK -Hd1
2@ ALL -Hd1 0 OK Hd1
2@ ALL -Hd1 -0 OK -Hd1
2@ ALL H 1 OK H
2@ ALL H -1 OK -H
2@ ALL -H 1 OK H
2@ ALL -H -1 OK -H
2@ ALL H Ed1 OK H
2@ ALL H -Ed1 OK -H
2@ ALL -H Ed1 OK H
2@ ALL -H -Ed1 OK -H
2@ ALL H 0 OK H
2@ ALL H -0 OK -H
2@ ALL -H 0 OK H
2@ ALL -H -0 OK -H
2@ ALL H H OK H
2@ ALL H -H OK -H
2@ ALL -H H OK H
2@ ALL -H -H OK -H
! NaNs -- FPTEST checks that NaNs
! are returned and with no exceptions.
2@ ALL Q 1 OK Q
2@ ALL Q -1 OK -Q
2@ ALL -Q 1 OK Q
2@ ALL -Q -1 OK -Q
2@ ALL Q 0i1 OK Q
2@ ALL Q -0i1 OK -Q
2@ ALL -Q 0i1 OK Q
2@ ALL -Q -0i1 OK -Q
2@ ALL Q H OK Q
2@ ALL Q -H OK -Q
2@ ALL -Q H OK Q
2@ ALL -Q -H OK -Q
2@ ALL Q 0 OK Q
2@ ALL Q -0 OK -Q
2@ ALL -Q 0 OK Q
2@ ALL -Q -0 OK -Q
2@ ALL S 1 OK S
2@ ALL S -1 OK -S
2@ ALL -S 1 OK S
2@ ALL -S -1 OK -S
2@ ALL S 0i1 OK S
2@ ALL S -0i1 OK -S
2@ ALL -S 0i1 OK S
2@ ALL -S -0i1 OK -S
2@ ALL S H OK S
2@ ALL S -H OK -S
2@ ALL -S H OK S
2@ ALL -S -H OK -S
2@ ALL S 0 OK S
2@ ALL S -0 OK -S
2@ ALL -S 0 OK S
2@ ALL -S -0 OK -S
2@ ALL 1 Q OK 1
2@ ALL 1 -Q OK -1
2@ ALL 1 S OK 1
2@ ALL 1 -S OK -1
2@ ALL -1 Q OK 1
2@ ALL -1 -Q OK -1
2@ ALL -1 S OK 1
2@ ALL -1 -S OK -1
2@ ALL H Q OK H
2@ ALL H -Q OK -H
2@ ALL H S OK H
2@ ALL H -S OK -H
2@ ALL -H Q OK H
2@ ALL -H -Q OK -H
2@ ALL -H S OK H
2@ ALL -H -S OK -H
2@ ALL S Q OK S
2@ ALL S -Q OK -S
2@ ALL S S OK S
2@ ALL S -S OK -S
2@ ALL -S Q OK S
2@ ALL -S -Q OK -S
2@ ALL -S S OK S
2@ ALL -S -S OK -S
2@ ALL Q Q OK Q
2@ ALL Q -Q OK -Q
2@ ALL Q S OK Q
2@ ALL Q -S OK -Q
2@ ALL -Q Q OK Q
2@ ALL -Q -Q OK -Q
2@ ALL -Q S OK Q
```

```
20 ALL -Q -S OK -Q
```

```
!
! Negate test vectors:
!
2~ ALL 1 0 OK -1
2~ ALL -1 0 OK 1
2~ ALL Ed1 0 OK -Ed1
2~ ALL -Ed1 0 OK Ed1
2~ ALL 0i1 0 OK -0i1
2~ ALL -0i1 0 OK 0i1
2~ ALL Hm1 0 OK -Hm1
2~ ALL -Hm1 0 OK Hm1
2~ ALL Hd1 0 OK -Hd1
2~ ALL -Hd1 0 OK Hd1
2~ ALL H 0 OK -H
2~ ALL -H 0 OK H
2~ ALL 0 0 OK -0
2~ ALL -0 0 OK 0
! NaNs -- FPTEST checks only that
! NaNs are produced and with no exceptions.
2~ ALL -Q 0 OK Q
2~ ALL Q 0 OK -Q
2~ ALL -S 0 OK S
2~ ALL S 0 OK -S
```

```
!
! Absolute value test vectors:
!
2A ALL 1 0 OK 1
2A ALL -1 0 OK 1
2A ALL Ed1 0 OK Ed1
2A ALL -Ed1 0 OK Ed1
2A ALL 0i1 0 OK 0i1
2A ALL -0i1 0 OK 0i1
2A ALL Hm1 0 OK Hm1
2A ALL -Hm1 0 OK Hm1
2A ALL Hd1 0 OK Hd1
2A ALL -Hd1 0 OK Hd1
2A ALL H 0 OK H
2A ALL -H 0 OK H
2A ALL 0 0 OK 0
2A ALL -0 0 OK 0
! NaNs -- FPTEST checks that results
! are NaNs with no exceptions.
2A ALL Q 0 OK Q
2A ALL -Q 0 OK Q
2A ALL S 0 OK S
2A ALL -S 0 OK S
```

```
!
! Nextafter Test Vectors:
!
! From 1.
2N ALL 1 2 OK 1i1
2N ALL 1 0 OK 1d1
2N ALL 1 -0 OK 1d1
2N ALL 1 1i1 OK 1i1
2N ALL 1 1d1 OK 1d1
2N ALL 1 Hm1 OK 1i1
2N ALL 1 Hd1 OK 1i1
2N ALL 1 1 OK 1
2N ALL 1 -Hd1 OK 1d1
2N ALL 1 E OK 1d1
2N ALL 1 Ed1 OK 1d1
2N ALL 1 0i1 OK 1d1
2N ALL 1 -1 OK 1d1
2N ALL 1 -H OK 1d1
2N ALL 1 H OK 1i1
! From -1.
2N ALL -1 -2 OK -1i1
2N ALL -1 0 OK -1d1
2N ALL -1 -0 OK -1d1
2N ALL -1 -1i1 OK -1i1
2N ALL -1 -1d1 OK -1d1
2N ALL -1 Hm1 OK -1d1
2N ALL -1 Hd1 OK -1d1
2N ALL -1 1 OK -1d1
2N ALL -1 -Hd1 OK -1i1
2N ALL -1 E OK -1d1
2N ALL -1 Ed1 OK -1d1
2N ALL -1 0i1 OK -1d1
2N ALL -1 -1 OK -1
2N ALL -1 H OK -1d1
2N ALL -1 -H OK -1i1
! From 1 + 1ulp of 1.
2N ALL 1i1 2 OK 1i2
2N ALL 1i1 0 OK 1
2N ALL 1i1 1i2 OK 1i2
2N ALL 1i1 1 OK 1
2N ALL 1i1 Hm1 OK 1i2
2N ALL 1i1 Hd1 OK 1i2
2N ALL 1i1 -1i1 OK 1
2N ALL 1i1 -Hd1 OK 1
2N ALL 1i1 E OK 1
2N ALL 1i1 Ed1 OK 1
2N ALL 1i1 0i1 OK 1
2N ALL 1i1 1i1 OK 1i1
2N ALL 1i1 H OK 1i2
2N ALL 1i1 -H OK 1
! From 1 - 1ulp of 1.
2N ALL 1d1 2 OK 1
2N ALL 1d1 0 OK 1d2
2N ALL 1d1 1 OK 1
2N ALL 1d1 1d2 OK 1d2
2N ALL 1d1 Hm1 OK 1
2N ALL 1d1 Hd1 OK 1
2N ALL 1d1 -1d1 OK 1d2
2N ALL 1d1 -Hd1 OK 1d2
2N ALL 1d1 E OK 1d2
2N ALL 1d1 Ed1 OK 1d2
2N ALL 1d1 0i1 OK 1d2
2N ALL 1d1 1d1 OK 1d1
2N ALL 1d1 H OK 1
2N ALL 1d1 -H OK 1d2
! From largest power of 2.
2N ALL Hm1 Hm2 OK Hm1d1
2N ALL Hm1 0 OK Hm1d1
2N ALL Hm1 Hm1d1 OK Hm1d1
2N ALL Hm1 Hm1 OK Hm1
2N ALL Hm1 Hd1 OK Hm1i1
2N ALL Hm1 -Hm1 OK Hm1d1
2N ALL Hm1 -Hd1 OK Hm1d1
2N ALL Hm1 E OK Hm1d1
2N ALL Hm1 Ed1 OK Hm1d1
2N ALL Hm1 0i1 OK Hm1d1
2N ALL Hm1 H OK Hm1i1
2N ALL Hm1 -H OK Hm1d1
! From largest number.
2N ALL Hd1 Hm1 OK Hd2
2N ALL Hd1 0 OK Hd2
2N ALL Hd1 -0 OK Hd2
2N ALL Hd1 Hd2 OK Hd2
2N ALL Hd1 Hd1 OK Hd1
2N ALL Hd1 -Hd1 OK Hd2
2N ALL Hd1 E OK Hd2
2N ALL Hd1 Ed1 OK Hd2
2N ALL Hd1 0i1 OK Hd2
2N ALL Hd1 H ox H
2N ALL Hd1 -H OK Hd2
2N ALL -Hd1 -H ox -H
2N ALL -Hd1 H OK -Hd2
! From smallest normalized number.
2N ALL E 2 OK Ei1
2N ALL E 0 xu Ed1
2N ALL E -0 xu Ed1
2N ALL E Ei1 OK Ei1
2N ALL E Ed1 xu Ed1
2N ALL E Hm1 OK Ei1
2N ALL E Hd1 OK Ei1
2N ALL E -E xu Ed1
2N ALL E -Hd1 xu Ed1
2N ALL E E OK E
2N ALL E 0i1 xu Ed1
2N ALL E H OK Ei1
2N ALL E -H xu Ed1
! From largest denormalized number.
2N ALL Ed1 2 OK E
2N ALL Ed1 0 xu Ed2
2N ALL Ed1 E OK E
2N ALL Ed1 Ed2 xu Ed2
2N ALL Ed1 Hm1 OK E
2N ALL Ed1 Hd1 OK E
2N ALL Ed1 -Ed1 xu Ed2
2N ALL Ed1 -Hd1 xu Ed2
2N ALL Ed1 Ed1 OK Ed1
2N ALL Ed1 0i1 xu Ed2
2N ALL Ed1 H OK E
2N ALL Ed1 -H xu Ed2
2N ALL -Ed1 -2 OK -E
2N ALL -Ed1 -0 xu -Ed2
2N ALL -Ed1 -E OK -E
2N ALL -Ed1 -Ed2 xu -Ed2
2N ALL -Ed1 -Hm1 OK -E
2N ALL -Ed1 -Hd1 OK -E
2N ALL -Ed1 Ed1 xu -Ed2
2N ALL -Ed1 Hd1 xu -Ed2
2N ALL -Ed1 -Ed1 OK -Ed1
```

```
2N ALL -Ed1 -0i1 xu -Ed2
2N ALL -Ed1 -H OK -E
2N ALL -Ed1 H xu -Ed2
! From smallest denormalized number.
2N ALL 0i1 2 xu 0i2
2N ALL 0i1 0 xu 0
2N ALL 0i1 0i2 xu 0i2
2N ALL 0i1 Hm1 xu 0i2
2N ALL -0i1 -0 xu -0
2N ALL -0i1 -0i2 xu -0i2
2N ALL -0i1 -Hm1 xu -0i2
2N ALL 0i1 Hd1 xu 0i2
2N ALL 0i1 0 xu 0
2N ALL 0i1 -0 xu 0
2N ALL -0i1 -0 xu -0
2N ALL -0i1 0 xu -0
2N ALL 0i1 -Hd1 xu 0
2N ALL 0i1 E xu 0i2
2N ALL 0i1 Ed1 xu 0i2
2N ALL 0i1 0i1 OK 0i1
2N ALL 0i1 H xu 0i2
2N ALL 0i1 -H xu 0
! From 0.
2N ALL 0 2 xu 0i1
2N ALL 0 0 OK 0
2N ALL 0 -0 OK 0
2N ALL 0 0i1 xu 0i1
2N ALL 0 -0i1 xu -0i1
2N ALL 0 Hm1 xu 0i1
2N ALL 0 Hd1 xu 0i1
2N ALL 0 -Hd1 xu -0i1
2N ALL 0 E xu 0i1
2N ALL 0 Ed1 xu 0i1
2N ALL 0 H xu 0i1
2N ALL 0 -H xu -0i1
! From -0.
2N ALL -0 2 xu 0i1
2N ALL -0 -0 OK -0
2N ALL -0 0 OK -0
2N ALL -0 0i1 xu 0i1
2N ALL -0 -0i1 xu -0i1
2N ALL -0 Hm1 xu 0i1
2N ALL -0 Hd1 xu 0i1
2N ALL -0 -Hd1 xu -0i1
2N ALL -0 E xu 0i1
2N ALL -0 Ed1 xu 0i1
2N ALL -0 H xu 0i1
2N ALL -0 -H xu -0i1
! From infinity.
2N ALL H 2 OK Hd1
2N ALL H 0 OK Hd1
2N ALL H -0 OK Hd1
2N ALL H Hm1 OK Hd1
2N ALL H Hd1 OK Hd1
2N ALL H -Hd1 OK Hd1
2N ALL H E OK Hd1
2N ALL H Ed1 OK Hd1
2N ALL H 0i1 OK Hd1
2N ALL H H OK H
2N ALL H -H OK Hd1
2N ALL -H 2 OK -Hd1
2N ALL -H 0 OK -Hd1
2N ALL -H -0 OK -Hd1
2N ALL -H -Hm1 OK -Hd1
2N ALL -H -Hd1 OK -Hd1
2N ALL -H Hd1 OK -Hd1
2N ALL -H -E OK -Hd1
2N ALL -H -Ed1 OK -Hd1
2N ALL -H -0i1 OK -Hd1
2N ALL -H H OK -Hd1
2N ALL -H -H OK -H
! Next-afters
2N ALL Q 0 OK Q
2N ALL Q -0 OK Q
2N ALL 0 Q OK Q
2N ALL -0 Q OK Q
2N ALL Q 1 OK Q
2N ALL Q -1 OK Q
2N ALL 1 Q OK Q
2N ALL -1 Q OK Q
2N ALL Ed1 Q OK Q
2N ALL -Ed1 Q OK Q
2N ALL Q Ed1 OK Q
2N ALL Q -Ed1 OK Q
2N ALL Q 0i1 OK Q
2N ALL Q -0i1 OK Q
2N ALL 0i1 Q OK Q
2N ALL -0i1 Q OK Q
2N ALL Q Hd1 OK Q
2N ALL Q -Hd1 OK Q
2N ALL Hd1 Q OK Q
2N ALL -Hd1 Q OK Q
2N ALL Q H OK Q
2N ALL Q -H OK Q
2N ALL H Q OK Q
2N ALL -H Q OK Q
2N ALL Q Q OK Q
2N ALL S 0 i Q
2N ALL S -0 i Q
2N ALL 0 S i Q
2N ALL -0 S i Q
2N ALL S 1 i Q
2N ALL S -1 i Q
2N ALL 1 S i Q
2N ALL -1 S i Q
2N ALL Ed1 S i Q
2N ALL -Ed1 S i Q
2N ALL S Ed1 i Q
2N ALL S -Ed1 i Q
2N ALL S 0i1 i Q
2N ALL S -0i1 i Q
2N ALL 0i1 S i Q
2N ALL -0i1 S i Q
2N ALL S Hd1 i Q
2N ALL S -Hd1 i Q
2N ALL Hd1 S i Q
2N ALL -Hd1 S i Q
2N ALL S H i Q
2N ALL S -H i Q
2N ALL H S i Q
2N ALL -H S i Q
2N ALL Q S i Q
2N ALL S Q i Q
2N ALL S S i Q
```

```
! Scalb test vectors.  Those with
! 2nd arguments that overflow
! the integer format are commented
! out, since the response to
! floating->integer conversion on
! overflow is system-dependent
!
! Warm ups.
2S ALL 1 1 OK 2
2S ALL -1 1 OK -2
2S ALL 1 -1 OK 1m1
2S ALL -1 -1 OK -1m1
2S ALL 1 3 OK 8
2S ALL 1 -3 OK 1m3
2S ALL 9 9 OK 9p9
2S ALL 9 -9 OK 9m9
2S ALL 7 8 OK 7p8
2S ALL -7 -8 OK -7m8
2S ALL 5 0 OK 5
2S ALL 5 -0 OK 5
2S ALL -5 -0 OK -5
! Big numbers.
2S ALL Hm1 -8 OK Hm9
2S ALL Hm9 8 OK Hm1
2S ALL Hd1 -9 OK Hd1m9
2S ALL Hd1m9 9 OK Hd1
2S ALL -Hd1 -9 OK -Hd1m9
2S ALL -Hd1m9 9 OK -Hd1
2S ALL Hd1 0 OK Hd1
2S ALL Hd1 -0 OK Hd1
! Overflows.
2S >= Hm1 1 xo H
2S <= -Hm1 1 xo -H
2S s>= 1 1p7 xo H
2S s<= -1 1p7 xo -H
! 2S >= 1 Hm9 xo H
2S ds>= 1 1p9p5 xo H
! 2S >= 1 Hd1 xo H
2S ds<= -1 1p9p5 xo -H
! 2S <= -1 Hd1 xo -H
! 2S >= 1m9 Hm9 xo H
2S >= Hd1 1 xo H
2S >= Hm9 9 xo H
2S ds>= E 1p9p5 xo H
2S ds>= Ed1 1p9p5 xo H
2S ds>= 0i1 1p9p5 xo H
2S ds<= -0i1 1p9p5 xo -H
! 2S >= E Hm1 xo H
! 2S >= Ed1 Hm1 xo H
! 2S >= 0i1 Hm1 xo H
! 2S <= -0i1 Hm1 xo -H
2S <0 Hm1 1 xo Hd1
2S >0 -Hm1 1 xo -Hd1
2S s<0 1 1p7 xo Hd1
2S s>0 -1 1p7 xo -Hd1
! 2S <0 1 Hm9 xo Hd1
2S ds<0 1 1p9p5 xo Hd1
! 2S <0 1 Hd1 xo Hd1
2S ds>0 -1 1p9p5 xo -Hd1
! 2S >0 -1 Hd1 xo -Hd1
! 2S <0 1m9 Hm9 xo Hd1
2S <0 Hd1 1 xo Hd1
2S <0 Hm9 9 xo Hd1
2S ds<0 E 1p9p5 xo Hd1
2S ds<0 Ed1 1p9p5 xo Hd1
2S ds<0 0i1 1p9p5 xo Hd1
2S ds>0 -0i1 1p9p5 xo -Hd1
! 2S <0 E Hm1 xo Hd1
! 2S <0 Ed1 Hm1 xo Hd1
! 2S <0 0i1 Hm1 xo Hd1
! 2S >0 -0i1 Hm1 xo -Hd1
! Tiny operand.
2S s E 1p7 OK 4
2S s Ed1 1p7 OK 1d2p2
2S s -Ed1 1p7 OK -1d2p2
2S d E 1p7p3 OK 4
2S d Ed1 1p7p3 OK 1d2p2
2S d -Ed1 1p7p3 OK -1d2p2
2S ALL 0i1 1 OK 0i2
2S ALL -0i1 1 OK -0i2
2S ALL 0i2 -1 OK 0i1
2S ALL 0i1 3 OK 0i8
2S ALL 0i8 -3 OK 0i1
2S ALL Ed1 1 OK Ep1d2
2S ALL Ep1d2 -1 OK Ed1
2S ALL Ed1 0 OK Ed1
2S ALL Ed1 -0 OK Ed1
! Underflows.
2S <=0 0i1 -1 xu 0
2S > 0i1 -1 xu 0i1
2S >=0 -0i1 -1 xu -0
2S < -0i1 -1 xu -0i1
2S <0 0i3 -2 xu 0
2S => 0i3 -2 xu 0i1
2S <=0 0i9 -3 xu 0i1
2S > 0i9 -3 xu 0i2
2S => 0i3 -1 xu 0i2
2S 0< 0i3 -1 xu 0i1
2S >= Ep1d1 -1 xu E
2S 0< Ep1d1 -1 xu Ed1
2S >= Ep9d1 -9 xu E
2S 0< Ep9d1 -9 xu Ed1
2S <=0ds 1 -1p9p5 xu 0
2S >ds 1 -1p9p5 xu 0i1
! 2S <=0 1 -Hm1 xu 0
! 2S > 1 -Hm1 xu 0i1
2S <= -Ep9d1 -9 xu -E
2S 0> -Ep9d1 -9 xu -Ed1
2S >=0ds -1 -1p9p5 xu -0
2S <ds -1 -1p9p5 xu -0i1
2S <=0ds E -1p9p5 xu 0
2S >ds E -1p9p5 xu 0i1
2S <=0ds 0i1 -1p9p5 xu 0
2S >ds 0i1 -1p9p5 xu 0i1
2S >=0ds -0i1 -1p9p5 xu -0
2S <ds -0i1 -1p9p5 xu -0i1
! 2S >=0 -1 -Hm1 xu -0
! 2S < -1 -Hm1 xu -0i1
! 2S <=0 E -Hm1 xu 0
! 2S > E -Hm1 xu 0i1
! 2S <=0 0i1 -Hd1 xu 0
! 2S > 0i1 -Hd1 xu 0i1
! 2S >=0 -0i1 -Hd1 xu -0
! 2S < -0i1 -Hd1 xu -0i1
! Infinity operands.
2S ALL H 0 OK H
2S ALL H -0 OK H
2S ALL -H 0 OK -H
```

```
2S ALL -H -0 OK -H
2S ALL H 1 OK H
2S ALL H 1p9p5 OK H
2S ALL H -1p9p5 OK H
2S ALL -H 1p9p5 OK -H
2S ALL -H -1p9p5 OK -H
!2S ALL H Hd1 OK H
!2S ALL H -Hd1 OK H
!2S ALL -H Hd1 OK -H
!2S ALL -H -Hd1 OK -H
!2S ALL 1 H OK H
!2S ALL Hd1 H OK H
!2S ALL 0i1 H OK H
!2S ALL H H OK H
!2S ALL -H H OK -H
!2S ALL H -H i Q
!2S ALL 0 H i Q
!Zeros.
2S ALL 0 1 OK 0
2S ALL 0 1p9p5 OK 0
2S ALL 0 -1p9p5 OK 0
!2S ALL 0 Hd1 OK 0
!2S ALL 0 -Hd1 OK 0
2S ALL 0 0 OK 0
2S ALL 0 -0 OK 0
2S ALL -0 1 OK -0
!2S ALL -0 Hd1 OK -0
!2S ALL -0 -Hd1 OK -0
2S ALL -0 -0 OK -0
2S ALL -0 0 OK -0
!NaNs.
2S ALL Q 1 OK Q
2S ALL Q 1p9p5 OK Q
2S ALL Q -1p9p5 OK Q
!2S ALL Q H OK Q
!2S ALL Q -Hd1 OK Q
!2S ALL Q H OK Q
!2S ALL Q -Hd1 OK Q
!2S ALL Q -H OK Q
!2S ALL Q Hd1 OK Q
2S ALL Q 0 OK Q
!2S ALL Q Q OK Q
!2S ALL 1 Q OK Q
!2S ALL H Q OK Q
!2S ALL -H Q OK Q
!2S ALL 0 Q OK Q
2S ALL S 1 i Q
2S ALL S 1p9p5 i Q
2S ALL S -1p9p5 i Q
2S ALL S 1p9p5 i Q
2S ALL S -1p9p5 i Q
!2S ALL S H i Q
!2S ALL S -Hd1 i Q
!2S ALL S -H i Q
!2S ALL S Hd1 i Q
2S ALL S 0 i Q
!2S ALL S S i Q
!2S ALL Q S i Q
!2S ALL S Q i Q
!2S ALL 1 S i Q
!2S ALL H S i Q
!2S ALL -H S i Q
!2S ALL 0 S i Q
```

```
!
! Test vectors for the fraction part
! of number as if with infinite range.
!
! Mid-range.
2F ALL 1 0 OK 1
2F ALL -1 0 OK -1
2F ALL 2 0 OK 1
2F ALL 3 0 OK 3m1
2F ALL 4 0 OK 1
2F ALL 5 0 OK 5m2
2F ALL 6 0 OK 6m2
2F ALL 7 0 OK 7m2
2F ALL -7 0 OK -7m2
2F ALL 8 0 OK 1
2F ALL 9 0 OK 9m3
2F ALL 1i1 0 OK 1i1
2F ALL 2i1 0 OK 1i1
2F ALL 3i1 0 OK 3m1i1
2F ALL 4i1 0 OK 1i1
2F ALL 5i1 0 OK 5m2i1
2F ALL 6i1 0 OK 6m2i1
2F ALL 7i1 0 OK 7m2i1
2F ALL 8i1 0 OK 1i1
2F ALL 9i1 0 OK 9m3i1
2F ALL -9i1 0 OK -9m3i1
2F ALL 1d1 0 OK 2d1
2F ALL 2d1 0 OK 2d1
2F ALL 3d1 0 OK 3d1m1
2F ALL 4d1 0 OK 2d1
2F ALL 5d1 0 OK 5d1m2
2F ALL 6d1 0 OK 6d1m2
2F ALL 7d1 0 OK 7d1m2
2F ALL 8d1 0 OK 2d1
2F ALL -8d1 0 OK -2d1
2F ALL 9d1 0 OK 9d1m3
! Small.
2F ALL E 0 OK 1
2F ALL -E 0 OK -1
2F ALL Ei1 0 OK 1i1
2F ALL Ed1 0 OK 2d2
2F ALL Ei8 0 OK 1i8
2F ALL Ed4 0 OK 2d8
2F ALL 0i1 0 OK 1
2F ALL -0i1 0 OK -1
2F ALL 0i8 0 OK 1
2F ALL 0i9 0 OK 9m3
2F ALL Ep1d1 0 OK 2d1
2F ALL Ep1d9 0 OK 2d9
2F ALL Ep1i1 0 OK 1i1
! Large.
2F ALL Hm1 0 OK 1
2F ALL Hd1 0 OK 2d1
2F ALL -Hm1 0 OK -1
2F ALL -Hd1 0 OK -2d1
2F ALL Hd9 0 OK 2d9
2F ALL Hm1i1 0 OK 1i1
2F ALL Hm1i8 0 OK 1i8
2F ALL Hm1d1 0 OK 2d1
```

```
! LogB test vectors
2L ALL 1 0 OK 0
2L ALL 2 0 OK 1
2L ALL -2 0 OK 1
2L ALL 3 0 OK 1
2L ALL 4 0 OK 2
2L ALL 5 0 OK 2
2L ALL 6 0 OK 2
2L ALL 7 0 OK 2
2L ALL 8 0 OK 3
2L ALL 9 0 OK 3
2L ALL 1p9 0 OK 9
2L ALL 2p8 0 OK 9
2L ALL 3p8 0 OK 9
2L ALL -3p8 0 OK 9
2L ALL 4p7 0 OK 9
2L ALL 5p7 0 OK 9
2L ALL 6p7 0 OK 9
2L ALL 7p7 0 OK 9
2L ALL 8p6 0 OK 9
2L ALL 9p6 0 OK 9
2L ALL 1p9d1 0 OK 8
2L ALL 2p8d1 0 OK 8
2L ALL 3p8d1 0 OK 9
2L ALL 4p7d1 0 OK 8
2L ALL -4p7d1 0 OK 8
2L ALL 5p7d1 0 OK 9
2L ALL 6p7d1 0 OK 9
2L ALL 7p7d1 0 OK 9
2L ALL 8p6d1 0 OK 8
2L ALL 9p6d1 0 OK 9
2L ALL 1m1 0 OK -1
2L ALL 3m1 0 OK 0
2L ALL 3m2 0 OK -1
2L ALL 9m1 0 OK 2
2L ALL 9m2 0 OK 1
2L ALL 9m3 0 OK 0
2L ALL 9m4 0 OK -1
2L ALL 9m5 0 OK -2
2L ALL 9m6 0 OK -3
2L ALL -9m6 0 OK -3
2L ALL 9m7 0 OK -4
2L ALL 9m8 0 OK -5
2L ALL 1d1 0 OK -1
2L ALL 2d1 0 OK 0
2L ALL 3d1 0 OK 1
2L ALL 4d1 0 OK 1
2L ALL 5d1 0 OK 2
2L ALL 6d1 0 OK 2
2L ALL 7d1 0 OK 2
2L ALL 8d1 0 OK 2
2L ALL -8d1 0 OK 2
2L ALL 9d1 0 OK 3
2L ALL 1m1d1 0 OK -2
2L ALL 3m1d1 0 OK 0
2L ALL 3m2d1 0 OK -1
2L ALL 9m1d1 0 OK 2
2L ALL 9m2d1 0 OK 1
2L ALL 9m3d1 0 OK 0
2L ALL 9m4d1 0 OK -1
2L ALL 9m5d1 0 OK -2
2L ALL 9m6d1 0 OK -3
2L ALL 9m7d1 0 OK -4
2L ALL 9m8d1 0 OK -5
2L ALL -9m7d1 0 OK -4
2L ALL 1p8p8 0 OK 1p4
2L ALL 1p8p8p8p8 0 OK 1p5
2L ALL 1p8p8p8p8p8p8p8p8 0 OK 1p6
2L ALL 1p8p8p8p8p8p8p8p8i9 0 OK 1p6
2L ALL -1p8p8p8p8p8p8p8p8i9 0 OK 1p6
2L ALL 1m8m8 0 OK -1p4
2L ALL 1m8m8m8m8 0 OK -1p5
2L ALL 1m8m8m8m8m8m8m8m8 0 OK -1p6
2L ALL 1m8m8m8m8m8m8m8m8i9 0 OK -1p6
2L ALL -1m8m8m8m8m8m8m8m8i9 0 OK -1p6
! Exceptional cases.
2L ALL Q 0 OK Q
2L ALL S 0 i Q
2L ALL H 0 OK H
2L ALL -H 0 OK H
2L ALL 0 0 z -H
2L ALL -0 0 z -H
```

APPENDIX C

Test Program for P754 Arithmetic – Version 2.0

```
{*
** FPTEST:  Program to test IEEE floating-point units.
**     Written by Jim Thomas and Jerome Coonen, 5 Jan 83.
**
** Overview:  FPTEST is a general Pascal program suitable for testing
**     different floating-point units.  FPTEST calls certain procedures and
**     functions from a unit FP; these are specific to the system being
**     tested.
**
**     One input file contains a list of filenames of test files.
**     These files contain test vectors, one per line.
**     Each test vector specifies environment, operands, arithmetic
**     operation, correct result, and correct exception flags for a
**     given test.  FPTEST sets the environment,
**     performs the operation on the operands, checks the
**     result and flags obtained with those specified in the test vector,
**     and reports discrepancies to a specified output file.
**
** Use:  FPTEST begins with a series of questions for the user:
**     Verbose? -- printing all is slow but aids debugging.
**     Check flags? -- check flags as well as numeric results?
**     Stop on errs? -- or continue, listing all to the output file.
**     Single? Double? Extended? -- which formats are to be tested?
**     File with list of test files?
**     Output file?
**
** Test Vectors:  An example:
**
**     Version
**      & Op     Modes   Oprnd1  Oprnd2  Flags  Result     Comment
**
**       2+       =     4d1     1u3     x      4     check rounding
**
**     Each test vector consists of seven fields: version number
**     and operator; rounding mode and precisions; 1st operand; 2nd operand;
**     flags; result; and comment.  The fields are separated by white
**     space (blanks and/or tabs); thus, no field but the last may be blank,
**     and only the last field can itself contain white space.  Each line
**     in a file of test vectors must be blank, a test vector, or a comment
**     line beginnig with an exclamation point (!).
**
**     In the example,
**       version    = 2
**       operator   = addition (+)
**       rounding   = round to nearest (=)
**       precision  = single (s)
**       1st operand = 4 decremented by 1 in its least significant bit,
**                  to single precision (4d1)
```

```
**      2nd operand = 3 units in the last place of 1, to single
**                precision (1u3)
**      flag      = inexact (x)
**      result    = 4
**      comment   = check rounding.
**
**
** Operators:
**    The operators available with this version of FPTEST are +, -, *,
**    /, V (square root), % (remainder), I (round to integer), N (next-
**    after), ~ (negate), @ (copy sign), A (absolute value), S (scalb),
**    L (logb), F (fraction part), and C (compare).
**
** Modes:
**    The rounding modes are = (to nearest), > (toward +INF),
**    < (toward -INF), and 0 (toward zero).  The precisions are
**    s (single), d (double), and e (extended).  Both operands and the
**    "correct" result will be constructed in the specified precision.
**    The test vector is processed only if its precision is one of those
**    initially requested by the user.  If no rounding mode is specified
**    then all are tested, and similarly for the precisions.  The
**    placekeeper ALL is used when there are no mode or precision
**    restrictions.
**
** Flags:
**    The flags are i (invalid), o (overflow), x (inexact),
**    z (divide by zero), and u, v, and w (underflow).  A 'w' flag
**    indicates that underflow must be signaled only if the
**    floating-point implementation tests for tininess before rounding.
**    A 'v' flag indicates that underflow must be signaled
**    unless the floating-point detects underflow as a loss of
**    accuracy due to denormalization.  A 'u' indicates that
**    all implementations must signal underflow.  OK indicates
**    no exceptions signaled.
**
** Numeric Value Specifiers:
**    These specifiers are scanned left to right.  They consist
**    of an optional sign, a root number, and one or more optional
**    modifiers for the root number.  The sign is specified by + or -
**    as usual, though the + may be omitted.  The root number is
**    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, H (infinity), E (the smallest
**    normalized power of 2, Q (a quiet NaN), and S (a signaling NaN).
**    Each modifier is a letter, i (increment in the last place),
**    d (decrement in the last place), u (units in the last place),
**    p (plus exponent bias), or m (minus exponent bias), followed by
**    a single digit.  "Units in the last place" refers to binary units.
**    The following examples illustrate the notation:
**
**      3i2 = 3 incremented by 2 units in its last place, i.e. the
**          2nd representable number after 3.
**      1u3 = 3 units in the last place of 1, e.g. 3 * 2^-23 in single.
**      Hd1 = the largest finite number.
**      Hm1 = the largest power of 2.
**      Ed1 = the largest denormal number.
**      0i1 = the smallest positive denormal number.
**      3m1 = 3 * 2^-1 = 1 1/2.
**      9p3 = 9 * 2^3 = 72.
**     -1d1 = the 1st number greater than -1 (note that the minus is
```

```
**          applied last).
**
*}

program FPTEST;                                                    FPTEST

uses
        FPSoft,  {* interface to software floating-point arithmetic *}
        FP;      {* interface to test routines *}

var
        {*
        ** Type Str8, string[8], is defined in fp. The XXXStr values
        ** are parsed from LinBuf. The XXXLim values limit tests
        ** to certain rounding modes and precisions. PossErrs is
        ** the list of possible error flags.
        *}
        PossErrs, RndLim, PrcLim, PrcStr, FlgStr, CFlgStr : Str8;

        {*
        ** Type Str90, string[90], is defined in fp. LinBuf is the input
        ** buffer for test vectors, TmpBuf is for I/O utilities, and the
        ** StrXXX variables are the string values in LinBuf representing
        ** numerical arguments.
        *}
        LinBuf, TmpBuf, StrArg1, StrArg2, StrRes : Str90;

        {*
        ** Type UnpForm is defined in fp. The UnpXXX variables contain
        ** values from the corresponding StrXXX variables. The following
        ** integer variables refer to the UnpXXX record, for a given
        ** precision.
        *}
        UnpArg1, UnpArg2, UnpRes : UnpForm;
        MaxExp, MinExp, SigBits, LowBit, LowByte : integer;

        {*
        ** Type PckForm is defined in fp. The PckXXX variables contain
        ** values packed from the corresponding UnpXXX variables.
        *}
        PckArg1, PckArg2, PckRes, PckFndRes : PckForm;

        {*
        ** UflowType is defined in fp. It tells which of the three
        ** P754 definitions of underflow is in effect.
        *}
        UflowOpt : UflowType;

        WhiteSpace : set of char;  {* contains <space> and <tab> *}

        {*
        ** FlgErr and NumErr record errors; ChkFlgErr determines whether
        ** flags are to be checked; StopOnErr determines whether to stop
        ** on further errors. Verbose requests same. The LinOut flag
        ** records whether input line and unpacked values mask has been
        ** printed yet.
        *}
        FlgErr, NumErr, ChkFlgErr, StopOnErr,
```

```
        Verbose, LinBufOut : boolean;

        {*
        ** The Xptr are indexes into argument and line buffer strings.
        ** Dots, errors and successful tests are counted by XXXCnt.
        *}
        aptr, lptr, DotCnt, FlgErrCnt, NumErrCnt, OKCnt : integer;

        {*
        ** pc and rc are the current precision and rounding characters.
        ** rev and operator are the revision number and arithmetic
        ** operator parsed from LinBuf.  The XXXRes are the results
        ** of comparisons.
        *}
        pc, rc, rev, operator, CmpRes, CCmpRes : char;

        {*
        ** ListFile contains a list of potential InFile's containing
        ** test vectors.  Error reports are written to OutFile.
        *}
        ListFile, InFile, OutFile : text;


{*
** Called by AddUlps and AddExp to normalize an UnpForm.
*}
procedure Normalize(var r : UnpForm);                    Normalize

var
        i, c, t : integer;

begin
        while (r.man[1] < 128) and (r.exp > MinExp) do
        begin
                c := 0;
                for i := MANLEN downto 1 do
                begin
                        t := r.man[i] * 2 + c;
                        if t > 255 then
                        begin
                                r.man[i] := t - 256;
                                c := 1
                        end
                        else
                        begin
                                r.man[i] := t;
                                c := 0
                        end
                end;
                r.exp:=r.exp-1
        end
end;
```

AddUlps

```
{*
** Called by BuildNum.
** Add n ulps to the number in UnpForm r and normalize the result
** as much as possible.  This routine is complicated by the need
** to do bit operations using Pascal types.
*}
procedure AddUlps(var r : UnpForm; n : integer);

var
        c, i, j, t : integer;

begin
        if n >= 0 then

                {*
                ** Add one ulp at a time up to n.  This is much easier
                ** than trying to add all at once.  Integer c propagates
                ** the carry-out from byte to byte.
                *}
                for i := 1 to n do
                begin
                        c := LowBit;
                        for j := LowByte downto 1 do
                        begin
                                t := r.man[j] + c;
                                if t > 255 then
                                begin
                                        r.man[j] := t - 256;
                                        c := 1
                                end
                                else
                                begin
                                        r.man[j] := t;
                                        c := 0
                                end
                        end;

                        if c = 1 then      {* Carry out of left end? *}
                        begin
                                r.man[1] := 128;
                                r.exp    := r.exp + 1
                        end
                end

        else  {* n < 0 *}

                for i := 1 to -n do
                begin
                        c := LowBit;
                        for j := LowByte downto 1 do
                        begin
                                t := r.man[j] - c;
                                if t < 0 then
                                begin
                                        r.man[j] := t + 256;
                                        c := 1
                                end
                                else
```

```
                                begin
                                        r.man[j] := t;
                                        c := 0
                                end
                        end;

                        if (r.man[1] < 128) and (r.exp > MinExp) then
                        begin
                                r.man[1] := r.man[1] + 128;
                                r.exp    := r.exp - 1
                        end
                end;

                Normalize(r)
end;
```

```
{*
** Called by BuildNum.
** Add n to the exponent of UnpForm r, taking account of
** the bottom of the exponent range.  If the number must
** be denormalized, shift right by a given number of bytes and
** then normalize to the extent possible.
*}
procedure AddExp(var r : UnpForm; n : integer);
```

AddExp

```
var
        i, j : integer;

begin
        r.exp := r.exp + n;

        if r.exp < MinExp then
        begin
                i := ((MinExp - r.exp) div 8) + 1;
                for j := MANLEN downto (i + 1) do
                        r.man[j] := r.man[j - i];
                for j := 1 to i do
                        r.man[j] := 0;
                r.exp := r.exp + (i * 8)
        end;

        Normalize(r)
end;
```

```
{*
** Called by BuildNum.
*}
procedure HexFloating(s: Str90; var r: UnpForm);
```

HexFloating

```
var
        i, val : integer;
        HiNib, more : boolean;

begin
```

```
aptr := aptr + 1;  {* skip over $ *}

HiNib := true;     {* place first nibble in high half of byte *}
i    := 1;         {* index of first man[] *}
more  := true;
while more and (aptr <= length(s)) do
begin
        case s[aptr] of

        '0', '1', '2', '3', '4', '5', '6', '7', '8', '9':
                val := ord(s[aptr]) - ord('0');

        'A', 'B', 'C', 'D', 'E', 'F':
                val := ord(s[aptr]) - ord('A') + 10;

        'a', 'b', 'c', 'd', 'e', 'f':
                val := ord(s[aptr]) - ord('a') + 10;

        otherwise more := false

        end;

        if more then
        begin
                if HiNib then
                        val := val * 16  {* left-align nibble in byte *}
                else
                        i := i - 1; {* recover from last i := i + 1 *}
                r.man[i] := r.man[i] + val;
                i := i + 1;
                HiNib := not HiNib;
                aptr := aptr + 1
        end
end;

r.exp := 0;
i := 1;  {* exponent sign carrier *}
if aptr <= length(s) then
begin
        if s[aptr] = '^' then
        begin
                aptr := aptr + 1;

                if aptr <= length(s) then
                        if s[aptr] = '+' then
                                aptr := aptr + 1
                        else if s[aptr] = '-' then
                        begin
                                aptr := aptr + 1;
                                i := -1
                        end;

                more := true;
                while more and (aptr <= length(s)) do
                        if (ord('0') <= ord(s[aptr]))
                                and (ord('9') >= ord(s[aptr])) then
                        begin
                                r.exp := (r.exp * 10)
```

```
                                                    + (ord(s[aptr]) - ord('0'));
                                        aptr := aptr + 1
                                end
                                else
                                        more := false
                        end
                end;

                r.exp := r.exp * i;
                aptr := aptr - 1  {* because will increment upon return *}
end;


{*
** Called by BuildUnpOps.
*}
procedure BuildNum(s: Str90; var r: UnpForm);                    BuildNum

var
        i : integer;

begin
        aptr := 1;  {* index into argument string *}

        r.sgn := 0;
        if s[aptr] = '+' then
                aptr := aptr + 1
        else if s[aptr] = '-' then
        begin
                r.sgn := 1;
                aptr := aptr + 1
        end;

        for i := 1 to MANLEN do
                r.man[i] := 0;

        case s[aptr] of
        '0': r.exp := MinExp;
        '1': begin r.exp := 0;  r.man[1] := 128 end;
        '2': begin r.exp := 1;  r.man[1] := 128 end;
        '3': begin r.exp := 1;  r.man[1] := 192 end;
        '4': begin r.exp := 2;  r.man[1] := 128 end;
        '5': begin r.exp := 2;  r.man[1] := 160 end;
        '6': begin r.exp := 2;  r.man[1] := 192 end;
        '7': begin r.exp := 2;  r.man[1] := 224 end;
        '8': begin r.exp := 3;  r.man[1] := 128 end;
        '9': begin r.exp := 3;  r.man[1] := 144 end;
        'e', 'E': begin r.exp := MinExp;  r.man[1] := 128 end;
        'h', 'H': begin r.exp := MaxExp;  r.man[1] := 128 end;
        'q', 'Q': begin r.exp := MaxExp;  r.man[1] := 1   end;
        's', 'S': begin r.exp := MaxExp;  r.man[1] := 65  end;
        '$': HexFloating(s, r)
        end;

        aptr := aptr + 1;

        while aptr < length(s) do
```

```
        begin
                case s[aptr] of
                'i': AddUlps(r, ord(s[aptr+1]) - ord('0'));
                'd': AddUlps(r, ord('0') - ord(s[aptr+1]));
                'u':
                        begin
                                for i := 1 to MANLEN do
                                        r.man[i] := 0;
                                AddUlps(r, ord(s[aptr+1]) - ord('0'))
                        end;
                'p': AddExp(r, ord(s[aptr+1]) - ord('0'));
                'm': AddExp(r, ord('0') - ord(s[aptr+1]))
                end;

                aptr := aptr + 2
        end
end;


{*
** Called by ErrReport and BuildUnpOps.
*}
procedure DispMask;                                              DispMask
begin
        writeln(OutFile);
        writeln(OutFile, 'rev: ', rev, '  op: ', operator);
        writeln(OutFile, 'Modes: ', RndLim, '  Precs: ', PrcLim);
        writeln(OutFile, 'FlgStr : ', FlgStr, '  ');
        FpShow(PckArg1, TmpBuf, pc);
        write(OutFile, 'PckArg1 :', TmpBuf);
        FpShow(PckArg2, TmpBuf, pc);
        writeln(OutFile, '  PckArg2 :', TmpBuf)
end;


{*
** Called by ErrReport.
*}
procedure DispRes;                                               DispRes
begin
        writeln(OutFile);
        writeln(OutFile, 'Rnd:', rc, '  CFlags:', CFlgStr, '  Flags:', FlgStr);

        if operator = 'C' then
                writeln(OutFile,'Computed: ',CCmpRes, '  Should be: ', CmpRes)
        else
        begin
                FpShow(PckFndRes, TmpBuf, pc);
                write(OutFile, 'Computed: ', TmpBuf);
                FpShow(PckRes, TmpBuf, pc);
                writeln(OutFile, '  Should be: ', TmpBuf)
        end
end;
```

BuildUnpOps

```
{*
** Called by TestLoop.
** First, the string operands are built in the generic unpacked
** format UnpForm, then they are packed into the variant record
** PckForm according to the precision pc.
*}
procedure BuildUnpOps;

var
        i : integer;

begin
        case pc of

        's':
                begin
                        MaxExp  := 128;
                        MinExp  := -126;
                        SigBits := 24;
                        LowBit  := 1;
                        LowByte := 3
                end;

        'd':
                begin
                        MaxExp  := 1024;
                        MinExp  := -1022;
                        SigBits := 53;
                        LowBit  := 8;
                        LowByte := 7
                end;

        'e':
                begin
                        MaxExp  := EXTMAXEXP;
                        MinExp  := EXTMINEXP;
                        SigBits := EXTSIGBITS;

                        LowBit  := 1;
                        i := (EXTSIGBITS mod 8);
                        while (i mod 8) <> 0 do
                        begin
                                LowBit := LowBit + LowBit;
                                i := i + 1
                        end;

                        LowByte := (EXTSIGBITS + 7) div 8
                end

        end;

        BuildNum(StrArg1, UnpArg1);
        FpPack(UnpArg1, PckArg1, pc);

        BuildNum(StrArg2, UnpArg2);
        FpPack(UnpArg2, PckArg2, pc);

        if operator <> 'C' then
```

```
        begin
                BuildNum(StrRes, UnpRes);
                FpPack(UnpRes, PckRes, pc)
        end;

        if Verbose then
                DispMask
end;


{*
** Called by TestLoop to
** set rounding mode, clear error flags, and compute.
*}
procedure ComputeResult;                                    ComputeResult
begin
        FpSetRound(rc);
        FpClearFlags;

        case operator of
                '+': FpAdd  (PckArg1, PckArg2,  PckFndRes, pc);
                '-': FpSub  (PckArg1, PckArg2,  PckFndRes, pc);
                '*': FpMul  (PckArg1, PckArg2,  PckFndRes, pc);
                '/': FpDiv  (PckArg1, PckArg2,  PckFndRes, pc);
                'V': FpSqrt (PckArg1, PckFndRes, pc);
                '%': FpRem  (PckArg1, PckArg2,  PckFndRes, pc);
                'C': FpCmp  (PckArg1, PckArg2,  CCmpRes,   pc);
                'I': FpInt  (PckArg1, PckFndRes, pc);
                'N': FpNxt  (PckArg1, PckArg2,  PckFndRes, pc);
                '~': FpNeg  (PckArg1, PckFndRes, pc);
                '@': FpCpySgn(PckArg1, PckArg2, PckFndRes, pc);
                'S': FpScl  (PckArg1, PckArg2,  PckFndRes, pc);
                'L': FpLog  (PckArg1, PckFndRes, pc);
                'A': FpAbs  (PckArg1, PckFndRes, pc);
                'F': FpFrc  (PckArg1, PckFndRes, pc)
        end
end;


{*
** Called by TestLoop to check the error flags.
*}
procedure FlgChk;                                           FlgChk

var
        i : integer;
        ChrStr : string[1];

begin
        CFlgStr := '';
        FlgErr  := false;

        for i := 1 to length(PossErrs) do
                if FpIfX(PossErrs[i]) then
                begin
                        ChrStr := copy(PossErrs, i, 1);
```

```
                        CFlgStr := concat(CFlgStr, ChrStr);
                        FlgErr  := FlgErr or (pos(ChrStr, FlgStr) = 0)
                end;

        FlgErr := ChkFlgErr and (FlgErr or (length(FlgStr) <> length(CFlgStr)))
end;


{*
** Called by TestLoop to check the numerical result.
** If both operands are NANs, they needn't be equal.
** Comparisons have a one-character result.
*}
procedure NumChk;                                                    NumChk
begin
        if operator <> 'C' then
        begin
                NumErr := not FpEqual(PckFndRes, PckRes, pc);
                if FpIsNAN(PckFndRes, pc) and FpIsNAN(PckRes, pc) then
                        NumErr := false
        end
        else
                NumErr := CCmpRes <> CmpRes
end;


{*
** Called by ErrReport and main program.
** Asks user Yes/No question, defaulting to yes.
*}
function InYesNo(Query : Str90) : boolean;                           InYesNo
begin
        writeln;
        write(Query, '  [default Y]? ');
        readln(TmpBuf);
        InYesNo := true;
        if (length(TmpBuf) > 0) then
                InYesNo := not (TmpBuf[1] in ['n', 'N'])
end;


{*
** Called by TestLoop.
** If OK, print a dot (no more than 50 per line).
** Otherwise display bad news and stop if requested.
*}
procedure ErrReport;                                                 ErrReport
begin
        if not (FlgErr or NumErr) then
        begin
                OKCnt := OKCnt + 1;
                DotCnt := DotCnt + 1;
                if DotCnt > 50 then
                begin
                        DotCnt := 0;
```

```
                writeln(OutFile)
            end;
            write(OutFile, '.')
        end;

        if (FlgErr or NumErr) and (not LinBufOut) then
        begin
            LinBufOut := true;
            writeln(OutFile);
            writeln(OutFile, LinBuf);
            DispMask
        end;

        if Verbose or FlgErr or NumErr then
            DispRes;

        if NumErr then
        begin
            NumErrCnt := NumErrCnt + 1;
            writeln(OutFile, 'NUM ERROR')
        end;

        if FlgErr then
        begin
            FlgErrCnt := FlgErrCnt + 1;
            writeln(OutFile, 'FLAG ERROR')
        end;

        if (FlgErr or NumErr) and StopOnErr then
            StopOnErr := InYesNo('Keep stopping on errors')
end;


{*
** Called by ReadLoop.
** For a given parsed input line, coordinate the tests for
** each desired precision and rounding mode, and check results.
*}
procedure TestLoop;                                          TestLoop

var
        i, ir, ip: integer;

begin
        {*
        ** For each precision, run the tests for this line.
        *}
        for ip := 1 to length(PrcLim) do
        begin
            pc := PrcLim[ip];

            BuildUnpOps;

            for ir := 1 to length(RndLim) do
            begin
                rc := RndLim[ir];
                ComputeResult;
```

```
                        FlgChk;
                        NumChk;
                        ErrReport
                end
        end
end;


{*
** Called by ParseLine to get revsion number and operator.
** If revision number is invalid, then force an illegal
** operator code.
*}
procedure GetOperator;                                          GetOperator
begin
        rev := LinBuf[lptr];
        lptr := lptr + 1;
        operator := LinBuf[lptr];
        lptr := lptr + 1;

        if rev <> '2' then
                operator := '!'
end;


{*
** Called by ParseLine to set rounding mode and precisions.
** If no rounding modes are specified, test all four.
** If no precisions are specified, test all of PrcStr;
** otherwise test only those specified that are in PrcStr.
** WARNING: if none of the specified precisions are in
** PrcStr, then test no precisions at all.
*}
procedure GetModes;                                             GetModes

var
        PrcLost : boolean;
        ChrStr  : string[1];

begin
        while LinBuf[lptr] in WhiteSpace do
                lptr := lptr + 1;

        RndLim  := '';
        PrcLim  := '';
        PrcLost := false;

        while not (LinBuf[lptr] in WhiteSpace) do
        begin
                ChrStr := copy(LinBuf, lptr, 1);

                case ChrStr[1] of

                '=', '0', '<', '>':
                        RndLim := concat(RndLim, ChrStr);
```

```
                's', 'd', 'e':
                        if pos(ChrStr, PrcStr) <> 0 then
                                PrcLim := concat(PrcLim, ChrStr)
                        else
                                PrcLost := true
                end;

                lptr := lptr + 1
        end;

        if length(RndLim) = 0 then
                RndLim := '=<>0';
        if (not PrcLost) and (length(PrcLim) = 0) then
                PrcLim := PrcStr
end;
```

```
{*
** Called by ParseLine to get operand strings verbatim.
** This routine simple retrieves the next non-white subtring
** of LinBuf.
*}
procedure GetVerbatim(var s: Str90);
```

GetVerbatim

```
var
        oldptr : integer;

begin
        while LinBuf[lptr] in WhiteSpace do
                lptr := lptr + 1;

        oldptr := lptr;   {* Start of numeric string. *}

        while not (LinBuf[lptr] in WhiteSpace) do
                lptr  := lptr  + 1;

        s := copy(LinBuf, oldptr, (lptr - oldptr))
end;
```

```
{*
** Called by ParseLine to place flags in a string.
*}
procedure GetFlags;
```

GetFlags

```
var
        c: char;
        ChrStr : string[1];

begin
        while LinBuf[lptr] in WhiteSpace do
                lptr := lptr + 1;

        FlgStr := '';
```

```
        ChrStr := '1';   {* Dummy one-character string. *}

        while not (LinBuf[lptr] in WhiteSpace) do
        begin
                c := LinBuf[lptr];

                {*
                ** Of the 3 underflow flags, u --> v --> w.
                ** Set the character in FlgStr to 'u' if
                ** underflow should occur for the system tested.
                *}
                if (c = 'w') and (UflowOpt = UFLBEFORE) then
                        c := 'u'
                else if (c = 'v') and (UflowOpt <> UFLIDEAL) then
                        c := 'u';

                ChrStr[1] := c;
                if c in ['x', 'i', 'o', 'u', 'z'] then
                        FlgStr := concat(FlgStr, ChrStr);

                lptr := lptr + 1   {* Skip over flag character. *}
        end
end;


{*
** Called by ReadLoop to parse the line of input.
*}
procedure ParseLine;                                          ParseLine
begin
        lptr := 1;        {* Index into LinBuf. *}

        GetOperator;
        GetModes;
        GetVerbatim(StrArg1);
        GetVerbatim(StrArg2);
        GetFlags;
        GetVerbatim(StrRes);

        if operator = 'C' then     {* Compare has character result. *}
                CmpRes := StrRes[1];

        if Verbose then    {* End line started by parse routines. *}
                writeln(OutFile)
end;


{*
** Called by main program to process test vectors.
*}
procedure ReadLoop;                                           ReadLoop

begin
        repeat
                readln(ListFile, TmpBuf);
                writeln(OutFile);
```

```
                writeln(OutFile, 'Input file: ', TmpBuf);
                reset(InFile, TmpBuf);

                repeat
                        readln(InFile, LinBuf);
                        LinBuf := concat(LinBuf, ' ');
                                        {* end with white space *}

                        if Verbose then
                        begin
                                writeln(OutFile);
                                writeln(OutFile, LinBuf);
                        end;
                        LinBufOut := Verbose;

                        {* Skip lines too short (blank) or starting with '!' *}
                        if (length(LinBuf) > 8) and (LinBuf[1] <> '!') then
                        begin
                                ParseLine;
                                TestLoop
                        end
                until eof(InFile);

                close(InFile)
        until eof(ListFile);
end;


begin {* main program *}

        {*
        ** Initialize constants and counters.
        *}
        WhiteSpace := [ chr(32), chr(9) ];  {* space and tab chars *}
        PossErrs  := 'iouxz';
        UflowOpt  := UFLBEFORE;
        DotCnt    := 0;
        FlgErrCnt := 0;
        NumErrCnt := 0;
        OKCnt     := 0;

        Verbose := InYesNo('Verbose');
        ChkFlgErr := InYesNo('Check flags');
        StopOnErr := InYesNo('Stop on errors');

        PrcStr := '';
        if InYesNo('Test Single') then
                PrcStr := 's';
        if InYesNo('Test Double') then
                PrcStr := concat(PrcStr, 'd');
        if InYesNo('Test Extended') then
                PrcStr := concat(PrcStr, 'e');

        writeln;
        write('File with list of test files [default TLIST.TEXT]: ');
        readln(TmpBuf);
        if length(TmpBuf) = 0 then
```

```
            TmpBuf := 'TLIST.TEXT';
        reset(ListFile, TmpBuf);

        writeln;
        write('Output file [default CONSOLE:]: ');
        readln(TmpBuf);
        if length(TmpBuf) = 0 then
            TmpBuf := 'CONSOLE:';
        rewrite(OutFile, TmpBuf);

        ReadLoop;

        writeln(OutFile);
        writeln(OutFile);
        writeln(OutFile, 'Successful tests: ', OKCnt);
        writeln(OutFile, 'Numerical Errors: ', NumErrCnt);
        writeln(OutFile, 'Flag Errors:      ', FlgErrCnt);

        close(OutFile);
        close(ListFile)
end.
```

```
{*
** FP:  Unit to be used by the program FPTEST for testing the
**      SANE floating-point unit for Apple computers.
**      Written by Jim Thomas and Jerome Coonen, 5 Jan 83.
**
**      FP uses the SANE Interface and should not require modification
**      unless the SANE Interface or the parameters (in INTERFACE below)
**      change.
**
**      The ordering of the bytes in a floating-point number differs
**      for different computers.  On the III, the bytes, from low address
**      to high, run from least to most significant.  The order is just
**      the opposite for Lisa.  This matters in FpPack, which
**      converts from type UnpForm to PckForm and in FpShow, which displays
**      a number as a string of hex digits (most to least significant).
**      For the arithmetic routines that logically OR a 1 into
**      a double number's least significant bit, the constant LSW
**      indicates which is the least significant word
**      of a double format number.
*}
unit FP;


INTERFACE

uses FPSoft;

const
        {*
        ** SYSTEM-DEPENDENT: index of least significant word of a double
        ** format number.  0 for III, 3 for Lisa.
        *}
        LSW       = 0;

        EXTMAXEXP  = 16384;
        EXTMINEXP  = -16383;
        EXTSIGBITS = 64;
        MANLEN     = 9;    {* MANLEN = (EXTSIGBITS + 7) div 8  +  1 *}

type
        UflowType = (UFLIDEAL, UFLAFTER, UFLBEFORE);
        Str90     = string[90];
        Str8      = string[8];

        UnpForm =
                record
                        sgn: 0..1;     {* 0 for + and 1 for - *}
                        exp: integer;  {* unbiased *}
                        man: packed array [1..MANLEN] of 0..255
                                {* explicit 1-bit to left of binary point *}
                end;

        PckForm =
                record
                        case char of
                                's': (s : Single);
                                'd': (d : Double);
                                'e': (e : Extended);
```

```
                                'b': (b : packed array [0..9] of 0..255)
        end;

procedure FpPack   (var x : UnpForm; var a : PckForm; pc : char);          FpPack
procedure FpShow   (var a : PckForm; var v : Str90; pc : char);            FpShow
procedure FpClearFlags;                                                    FpClearFlags
function FpIfX     (err : char) : boolean;                                 FpIfX
procedure FpSetRound (rndc : char);                                        FpSetRound
function FpEqual   (var a,b : PckForm; pc : char) : boolean;               FpEqual
function FpIsNAN   (var a : PckForm; pc : char) : boolean;                 FpIsNAN
procedure FpAdd    (var a,b,c : PckForm; pc : char);                       FpAdd
procedure FpSub    (var a,b,c : PckForm; pc : char);                       FpSub
procedure FpMul    (var a,b,c : PckForm; pc : char);                       FpMul
procedure FpDiv    (var a,b,c : PckForm; pc : char);                       FpDiv
procedure FpRem    (var a,b,c : PckForm; pc : char);                       FpRem
procedure FpNxt    (var a,b,c : PckForm; pc : char);                       FpNxt
procedure FpScl    (var a,b,c : PckForm; pc : char);                       FpScl
procedure FpLog    (var a,c   : PckForm; pc : char);                       FpLog
procedure FpSqrt   (var a,c   : PckForm; pc : char);                       FpSqrt
procedure FpInt    (var a,c   : PckForm; pc : char);                       FpInt
procedure FpCpySgn (var a,b,c : PckForm; pc : char);                       FpCpySgn
procedure FpNeg    (var a,c   : PckForm; pc : char);                       FpNeg
procedure FpAbs    (var a,c   : PckForm; pc : char);                       FpAbs
procedure FpFrc    (var a,c   : PckForm; pc : char);                       FpFrc
procedure FpCmp    (var a,b   : PckForm; var c : char; pc : char);         FpCmp

IMPLEMENTATION

{*
** The following variables are used as local temporaries in the
** routines that follow.  They are declared globally for convenience.
*}
```

```
var
        t, t0 : Extended;
        EnvSav : Environ;
        RndSav : RoundDir;


{*
** Pack number in UnpForm x into PckForm a with precision pc.
** SYSTEM DEPENDENCY : The ordering of bytes in a floating-point
** "word" is the vital issue here.
*}
procedure FpPack {* (var x : UnpForm; var a : PckForm; pc : char) *};      FpPack

var
        i, bexp : integer;

begin
        case pc of
        's' :
                begin
                        bexp := x.exp+127;
                        a.b[3] := bexp div 2 + 128*x.sgn;
                        a.b[2] := (bexp mod 2)*128 + x.man[1] mod 128;
                        a.b[1] := x.man[2];
                        a.b[0] := x.man[3];
                        if (x.man[1]<128) and (bexp=1) then a.b[2] := a.b[2]-128
                end;

        'd' :
                begin
                        bexp := x.exp+1023;
                        a.b[7] := bexp div 16 + 128*x.sgn;
                        a.b[6] := (bexp mod 16)*16 + (x.man[1] div 8) mod 16;
                        for i := 5 downto 0 do
                                a.b[i] := (x.man[6-i] mod 8)*32
                                                + x.man[7-i] div 8;
                        if (x.man[1]<128) and (bexp=1) then a.b[6] := a.b[6]-16
                end;

        'e' :
                begin
                        bexp := x.exp+16383;
                        a.b[9] := bexp div 256 + 128*x.sgn;
                        a.b[8] := bexp mod 256;
                        for i := 7 downto 0 do a.b[i] := x.man[8-i];
                        if (x.exp = EXTMAXEXP) and (x.man[1] > 127) then
                                a.b[7] := a.b[7]-128
                end
        end
end;


{*
** Called by FpShow; returns the hex digit for the nibble n.
*}
function Nib2Hex(n : integer) : char;                                      Nib2Hex
```

```
begin
        if n < 10 then
                Nib2Hex := chr(ord('0') + n)
        else
                Nib2Hex := chr(ord('A') + n - 10);
end;


{*
** Return with v equal the hexadecimal representation of a.
** SYSTEM DEPENDENCY : order of bytes presumed here.
*}
procedure FpShow {* (var a : PckForm; var v : Str90; pc : char) *};        FpShow

var
        i, last : integer;
        s : string[3];

begin
        case pc of
                's': last := 3;
                'd': last := 7;
                'e': last := 9
        end;

        v := '';
        for i := last downto 0 do
        begin
                s := '  ';
                s[2] := Nib2Hex(a.b[i] div 16);
                s[3] := Nib2Hex(a.b[i] mod 16);
                v := concat(v, s)
        end
end;


{*
** Clear flags.
*}
procedure FpClearFlags;                                                    FpClearFlags

var
        xcp : Exception;

begin
        for xcp := INVALID to INEXACT do
                SetXcp(xcp,false)
end;


{*
** Return true iff err flag is set.
*}
```

FpIfX

```
function FpIfX {* (err : char) : boolean *};

begin
        case err of
        'u': FpIfX := TestXcp(UNDERFLOW);
        'o': FpIfX := TestXcp(OVERFLOW);
        'x': FpIfX := TestXcp(INEXACT);
        'i': FpIfX := TestXcp(INVALID);
        'z': FpIfX := TestXcp(DIVBYZERO)
        end
end;
```

FpSetRound

```
{*
** Set rounding modes.
*}
procedure FpSetRound {* (rndc : char) *};

begin
        case rndc of
        '=': SetRnd(TONEAREST);
        '>': SetRnd(UPWARD);
        '<': SetRnd(DOWNWARD);
        '0': SetRnd(TOWARDZERO)
        end
end;
```

FpEqual

```
{*
** Return true iff a and b are bit-for-bit equal.
*}
function FpEqual {* (a,b : PckForm; pc : char) : boolean *};

var
        i, last : integer;

begin
        case pc of
                's' : last := 1;
                'd' : last := 3;
                'e' : last := 4
        end;
        FpEqual := true;
        for i := 0 to last do
                if a.e[i] <> b.e[i] then
                        FpEqual := false
end;
```

```
{*
** Return true iff a is a NaN.
*}
```

FpIsNAN

```
function FpIsNAN {* (var a : PckForm; pc : char) : boolean *};

var
        sign : integer;

begin
        case pc of
        's': FpIsNAN := (ClassS(a.s,sign)=QNAN) or (ClassS(a.s,sign)=SNAN);
        'd': FpIsNAN := (ClassD(a.d,sign)=QNAN) or (ClassD(a.d,sign)=SNAN);
        'e': FpIsNAN := (ClassX(a.e,sign)=QNAN) or (ClassX(a.e,sign)=SNAN)
        end
end;



{*
** FPoperations :
**
**  Perform  c <--- a operation b  where a, b, and c have precision pc.
**  The actual procedure is move b to extended, operate on the extended
**  value with a, and move the result to c.  Care is taken to avoid double
**  roundings in double precision by simulating atomic operations.
*}

{*
** c  :=  a + b
*}
```

FpAdd

```
procedure FpAdd {* (var a,b,c : PckForm; pc : char) *};
begin
        case pc of
        's':
                begin
                        S2X(a.s,t);
                        AddS(b.s,t);
                        X2S(t,c.s)
                end;

        'd':
                begin
                        D2X(a.d,t);
                        AddD(b.d,t);
                        if TestXcp(INEXACT) then
                        begin
                                RndSav := GetRnd;
                                SetRnd(TOWARDZERO);
                                D2X(a.d,t);
                                AddD(b.d,t);
                                if not odd(t[LSW]) then
                                        t[LSW] := t[LSW] + 1;
                                SetRnd(RndSav)
                        end;
                        X2D(t,c.d)
                end;

        'e':
                begin
                        c.e := a.e;
                        AddX(b.e,c.e)
```

```
                end
        end
end;
```

```
{*
** c  :=  a - b
*}
procedure FpSub {* (var a,b,c : PckForm; pc : char) *};
begin
        case pc of
        's':
                begin
                        S2X(a.s,t);
                        SubS(b.s,t);
                        X2S(t,c.s)
                end;

        'd':
                begin
                        D2X(a.d,t);
                        SubD(b.d,t);
                        if TestXcp(INEXACT) then
                        begin
                                RndSav := GetRnd;
                                SetRnd(TOWARDZERO);
                                D2X(a.d,t);
                                SubD(b.d,t);
                                if not odd(t[LSW]) then
                                        t[LSW] := t[LSW] + 1;
                                SetRnd(RndSav)
                        end;
                        X2D(t,c.d)
                end;

        'e':
                begin
                        c.e := a.e;
                        SubX(b.e,c.e)
                end
        end
end;
```

FpSub

```
{*
** c  :=  a * b
*}
procedure FpMul {* (var a,b,c : PckForm; pc : char) *};
begin
        case pc of
        's':
                begin
                        S2X(a.s,t);
                        MulS(b.s,t);
                        X2S(t,c.s)
                end;
```

FpMul

```
        'd':
                begin
                        RndSav := GetRnd;
                        SetRnd(TOWARDZERO);
                        D2X(a.d,t);
                        MulD(b.d,t);
                        if TestXcp(INEXACT) and (not odd(t[LSW]))then
                                t[LSW] := t[LSW] + 1;
                        SetRnd(RndSav);
                        X2D(t,c.d)
                end;

        'e':
                begin
                        c.e := a.e;
                        MulX(b.e,c.e)
                end
        end
end;


{*
** c := a / b
*}
procedure FpDiv {* (var a,b,c : PckForm; pc : char) *};                    FpDiv
begin
        case pc of
        's':
                begin
                        S2X(a.s,t);
                        DivS(b.s,t);
                        X2S(t,c.s)
                end;

        'd':
                begin
                        RndSav := GetRnd;
                        SetRnd(TOWARDZERO);
                        D2X(a.d,t);
                        DivD(b.d,t);
                        if TestXcp(INEXACT) and (not odd(t[LSW])) then
                                t[LSW] := t[LSW] + 1;
                        SetRnd(RndSav);
                        X2D(t,c.d)
                end;

        'e':
                begin
                        c.e := a.e;
                        DivX(b.e,c.e)
                end
        end
end;
```

```
{*
** c := a rem b
*}
procedure FpRem {* (var a,b,c : PckForm; pc : char) *};
```

FpRem

```
var
        quo : integer;

begin
        case pc of
        's':
                begin
                        S2X(a.s,t);
                        S2X(b.s,t0);
                        RemX(t0,t,quo);
                        X2S(t,c.s)
                end;

        'd':
                begin       {* double rounding ignored *}
                        D2X(a.d,t);
                        D2X(b.d,t0);
                        RemX(t0,t,quo);
                        X2D(t,c.d)
                end;

        'e':
                begin
                        c.e := a.e;
                        RemX(b.e,c.e,quo)
                end
        end
end;


{*
** c := sqrt(a)
*}
procedure FpSqrt {* (var a,c : PckForm; pc : char) *};
```

FpSqrt

```
begin
        case pc of
        's':
                begin
                        S2X(a.s,t);
                        SqrtX(t);
                        X2S(t,c.s)
                end;

        'd':
                begin
                        RndSav := GetRnd;
                        SetRnd(TOWARDZERO);
                        D2X(a.d,t);
                        SqrtX(t);
                        if TestXcp(INEXACT) and (not odd(t[LSW])) then
                                t[LSW] := t[LSW] + 1;
```

```
                        SetRnd(RndSav);
                        X2D(t,c.d)
                end;

        'e':
                begin
                        c.e := a.e;
                        SqrtX(c.e)
                end
        end
end;


{*
** c  :=  a rounded to an integer
*}
procedure FpInt {* (var a,c : PckForm; pc : char) *};                    FpInt

begin
        case pc of
        's':
                begin
                        S2X(a.s,t);
                        RintX(t);
                        X2S(t,c.s)
                end;

        'd':
                begin
                        D2X(a.d,t);
                        RintX(t);
                        X2D(t,c.d)
                end;

        'e':
                begin
                        c.e := a.e;
                        RintX(c.e)
                end
        end
end;


{*
** c  :=  next representable value from a to b.
*}
procedure FpNxt {* (var a,b,c : PckForm; pc : char) *};                  FpNxt

begin
        c := a;
        case pc of
                's': NextS(c.s,b.s);
                'd': NextD(c.d,b.d);
                'e': NextX(c.e,b.e)
        end
end;
```

```
{*
** c := a * 2^c
*}
procedure FpScl {* (var a,b,c : PckForm; pc : char) *};

var
        n : integer;

begin
        case pc of
        's':
                begin
                        S2X(b.s,t);
                        X2I(t,n);
                        S2X(a.s,t);
                        ScalbX(n,t);
                        X2S(t,c.s)
                end;

        'd':
                begin
                        D2X(b.d,t);
                        X2I(t,n);
                        D2X(a.d,t);
                        ScalbX(n,t);
                        X2D(t,c.d)
                end;

        'e':
                begin
                        X2I(b.e,n);
                        c.e := a.e;
                        ScalbX(n,c.e)
                end
        end
end;
```

FpScl

```
{*
** c := binary exponent of a
*}
procedure FpLog {* (var a,c  : PckForm; pc : char) *};

begin
        case pc of
        's':
                begin
                        S2X(a.s,t);
                        LogbX(t);
                        X2S(t,c.s)
                end;

        'd':
                begin
                        D2X(a.d,t);
                        LogbX(t);
                        X2D(t,c.d)
```

FpLog

```
                end;

        'e':
                begin
                        c.e := a.e;
                        LogbX(c.e)
                end
        end
end;


{*
** n  := logb(a)
** c  := scalb(-n, a)
*}
procedure FpFrc {* (var a,c   : PckForm; pc : char) *};                    FpFrc

var
        n : integer;

begin
        case pc of
        's':
                begin
                        S2X(a.s,t);
                        t0 := t;
                        LogbX(t0);
                        X2I(t0,n);
                        ScalbX(-n,t);
                        X2S(t,c.s)
                end;

        'd':
                begin
                        D2X(a.d,t);
                        t0 := t;
                        LogbX(t0);
                        X2I(t0,n);
                        ScalbX(-n,t);
                        X2D(t,c.d)
                end;

        'e':
                begin
                        t := a.e;
                        t0 := t;
                        LogbX(t0);
                        X2I(t0,n);
                        ScalbX(-n,t);
                        c.e := t
                end
        end
end;
```

```
{*
** The next three procedures, FpCpySgn, FpNeg, and FpAbs are
** set up to unexceptional, even for signaling NANs.  The
** arithmetic environment is save and restored across the calls.
** If the source operand is a signaling NAN, a quiet NAN is
** returned, but its sign is appropriately tweaked.
*}

{*
** c  :=  a with the sign of b
*}
procedure FpCpySgn {* (var a,b,c : PckForm; pc : char) *};

var
        sgn :  integer;

begin
        case pc of
        's':
                begin
                        GetEnv(EnvSav);
                        S2X(a.s,t0);
                        S2X(b.s,t);
                        SetEnv(EnvSav);
                        CpySgnX(t0,t);
                        X2S(t0,c.s)
                end;

        'd':
                begin
                        GetEnv(EnvSav);
                        D2X(a.d,t0);
                        D2X(b.d,t);
                        SetEnv(EnvSav);
                        CpySgnX(t0,t);
                        X2D(t0,c.d)
                end;

        'e':
                begin
                        c.e := a.e;
                        CpySgnX(c.e,b.e)
                end
        end
end;
```

FpCpySgn

```
{*
** c  :=  a, but with opposite sign
*}
procedure FpNeg {* (var a,c   : PckForm; pc : char) *};

var
        sgn :  integer;

begin
        case pc of
```

FpNeg

```
        's':
                begin
                        GetEnv(EnvSav);
                        S2X(a.s,t);
                        SetEnv(EnvSav);
                        NegX(t);
                        X2S(t,c.s)
                end;

        'd':
                begin
                        GetEnv(EnvSav);
                        D2X(a.d,t);
                        SetEnv(EnvSav);
                        NegX(t);
                        X2D(t,c.d)
                end;

        'e':
                begin
                        c.e := a.e;
                        NegX(c.e)
                end
        end
end;


{*
** c  :=  absolute value of a
*}
procedure FpAbs {* (var a,c   : PckForm; pc : char) *};                    FpAbs

var
        sgn : integer;

begin
        case pc of
        's':
                begin
                        GetEnv(EnvSav);
                        S2X(a.s,t);
                        SetEnv(EnvSav);
                        AbsX(t);
                        X2S(t,c.s)
                end;

        'd':
                begin
                        GetEnv(EnvSav);
                        {* to avoid invalid on signaling NaNs -- a quiet *}
                        D2X(a.d,t);
                        {* NaN is returned but FPTEST does not notice *}
                        SetEnv(EnvSav);
                        AbsX(t);
                        X2D(t,c.d)
                end;
```

```
        'e':
                begin
                        c.e := a.e;
                        AbsX(c.e)
                end
        end
end;


{*
** Comparisons :
** This rather elaborate set of procedures tests two kinds of comparison:
** (1) Condition code -- as in the test vectors.
** (2) Predicates.
** P754 specifies which of the predicates should signal invalid on
** unordered (one operand is NAN).  The predicates available through
** the type RelOp are:
**    GT -- >, LT -- <, GL -- <>, EQ -- =, GE -- >=, LE -- <=,
**    GEL -- <=>, UNORD -- unordered.
** If all tests are satisfied, the appropriate condition =, <, >, ?
** is returned to CCmpRes (via parameter c); otherwise ! is returned.
*}
procedure FpCmp {* (var a,b : PckForm;var c : char; pc : char) *};          FpCmp

const
        UNORDFLAGS = 'iii0iii0';
        INVFLAGS   = 'iiiiiiii';
        OKFLAGS    = '00000000';

var
        rslts,t         : integer;
        rel             : RelOp;
        flgs0,flgs1,flgs  : Str90;
        ae,be           : Extended;



        {*
        ** Save flags as a string of i, o, u, x, and z.
        *}
        procedure SavFlgs(var flgs : Str90);                                SavFlgs

        var
                xcp : Str8;
                i   : integer;

        begin
                xcp := 'iouxz';
                flgs := '';
                for i := 1 to 5 do
                        if FpIfX(xcp[i]) then
                                flgs := concat(flgs, copy(xcp, i, 1));
                if flgs='' then
                        flgs := '0'
        end;
```

```
{*
** Restore flags according to the string : flgs.
*}
procedure RstFlgs(flgs : Str90);                                    RstFlgs

var i : integer;

begin
        fpclearflags;
        if flgs<>'0' then
                for i := 1 to length(flgs) do
                        case flgs[i] of
                                'i': SetXcp(INVALID,true);
                                'o': SetXcp(OVERFLOW,true);
                                'u': SetXcp(UNDERFLOW,true);
                                'x': SetXcp(INEXACT,true);
                                'z': SetXcp(DIVBYZERO,true)
                        end
end;


{*
** Clear all flags and signal inexact.
*}
procedure MarkInx;                                                  MarkInx
begin
        FpClearFlags;
        SetXcp(INEXACT, true)
end;


begin  {* FpCmp *}
        case pc of
                's':  begin S2X(a.s,ae); S2X(b.s,be) end;
                'd':  begin D2X(a.d,ae); D2X(b.d,be) end;
                'e':  begin ae := a.e; be := b.e end
        end;

        rslts := 0;
        flgs := '';
        SavFlgs(flgs0);

        {* SYSTEM DEPENDENCY : linear ordering of relationals *}
        t := 1;
        for rel := GT to UNORD do
        begin
                RstFlgs(flgs0);
                if CmpX(ae,rel,be) then
                        rslts := rslts + t;
                t := t*2;
                SavFlgs(flgs1);
                flgs := concat(flgs, flgs1)
        end;

        c := 'X';
        case rslts of
```

```
                128: if RelX(ae,be) = UNORD then c := '?';
                85:  if RelX(ae,be) = GT then c := '>';
                102: if RelX(ae,be) = LT then c := '<';
                120: if RelX(ae,be) = EQ then c := '='
        end;

        case rslts of
        128:
                if FpIsNAN(a,pc) or FpIsNAN(b,pc) then
                        if (flgs <> INVFLAGS) and (flgs <> UNORDFLAGS) then
                                MarkInx
                        else
                else if flgs <> UNORDFLAGS then
                        MarkInx;

        85,102,120:
                if (flgs <> OKFLAGS) and (flgs <> INVFLAGS) then
                        MarkInx;
        end
end;


end {* of unit fp *}.
```

```
UNIT FPSoft;
        {* Interface to floating-point software library. *}

INTERFACE

CONST

        SIGDIGLEN = 20;    { Maximum length of SigDig. }
        DECSTRLEN = 80;    { Maximum length of DecStr. }

TYPE

        {-----------------------------------------------
        ** Numeric types.
        -----------------------------------------------}

        Single   = array [0..1] of integer;
        Double   = array [0..3] of integer;
        Extended = array [0..4] of integer;

        {-----------------------------------------------
        ** Decimal string type and intermediate decimal
        ** type, representing the value:
        **     (-1)^sgn * 10^exp * dig
        -----------------------------------------------}

        SigDig   = string [SIGDIGLEN];
        DecStr   = string [DECSTRLEN];
        Decimal  =
                        record
                                sgn : 0..1;   { Sign (0 for pos, 1 for neg). }
                                exp : integer; { Exponent. }
                                sig : SigDig   { String of significant digits. }
                        end;

        {-----------------------------------------------
        ** Modes, flags, and selections.
        -----------------------------------------------}

        Environ   = integer;
        RoundDir  = (TONEAREST, UPWARD, DOWNWARD, TOWARDZERO);
        RelOp     = (GT, LT, GL, EQ, GE, LE, GEL, UNORD);
                                { >  <  <>  =  >=  <=  <=> }
        Exception = (INVALID, UNDERFLOW, OVERFLOW, DIVBYZERO, INEXACT);
        NumClass  = (SNAN, QNAN, INFINITE, ZERO, NORMAL, DENORMAL);
        DecForm   =
                        record
                                style  : (FLOAT, FIXED);
                                digits : integer
                        end;

{-----------------------------------------------
** Two address, extended-based arithmetic.
-----------------------------------------------}

procedure AddS (x : Single;  var y : Extended);                    AddS

procedure AddD (x : Double;  var y : Extended);                    AddD
```

```
procedure AddX (x : Extended; var y : Extended);                    AddX
        { y := y + x }

procedure SubS (x : Single;   var y : Extended);                    SubS

procedure SubD (x : Double;   var y : Extended);                    SubD

procedure SubX (x : Extended; var y : Extended);                    SubX
        { y := y - x }

procedure MulS (x : Single;   var y : Extended);                    MulS

procedure MulD (x : Double;   var y : Extended);                    MulD

procedure MulX (x : Extended; var y : Extended);                    MulX
        { y := y * x }

procedure DivS (x : Single;   var y : Extended);                    DivS

procedure DivD (x : Double;   var y : Extended);                    DivD

procedure DivX (x : Extended; var y : Extended);                    DivX
        { y := y / x }

function  CmpX (x : Extended; r : RelOp; y : Extended) : boolean;   CmpX
        { x r y }

function  RelX (x, y : Extended) : RelOp;                           RelX
        { x RelX y, where RelX in [GT, LT, EQ, UNORD] }

{-------------------------------------------------
** Conversions between Extended and
** the other numeric types.
-------------------------------------------------}

procedure S2X (x : Single;   var y : Extended);                     S2X

procedure D2X (x : Double;   var y : Extended);                     D2X

procedure X2X (x : Extended; var y : Extended);                     X2X
        { y := x (arithmetic assignment) }

procedure X2S (x : Extended; var y : Single);                       X2S

procedure X2D (x : Extended; var y : Double);                       X2D
        { y := x (arithmetic assignment) }

{-------------------------------------------------
** Numerical 'library' procedures and functions.
-------------------------------------------------}

procedure RemX   (x : Extended; var y : Extended; var quo : integer);   RemX
        { new y := remainder of ((old y)/ x), such that |new y| <= |x|/ 2;
                   quo := low order seven bits of integer quotient y/ x,
                   so that -127 <= quo <= 127.  }

procedure SqrtX  (var x : Extended);                                SqrtX
```

```
        { x := sqrt (x) }
procedure RintX  (var x : Extended);                                    RintX
        { x := rounded value of x }
procedure NegX   (var x : Extended);                                    NegX
        { x := -x }
procedure AbsX   (var x : Extended);                                    AbsX
        { x := |x| }
procedure CpySgnX (var x : Extended; y : Extended);                     CpySgnX
        { x := x with the sign of y }

procedure NextS  (var x : Single;   y : Single);                        NextS

procedure NextD  (var x : Double;   y : Double);                        NextD

procedure NextX  (var x : Extended; y : Extended);                      NextX
        { x := next representable value from x toward y }

function  ClassS (x : Single;   var sgn : integer) : NumClass;          ClassS

function  ClassD (x : Double;   var sgn : integer) : NumClass;          ClassD

function  ClassX (x : Extended; var sgn : integer) : NumClass;          ClassX
                  { sgn := sign of x (0 for pos, 1 for neg) }

procedure ScalbX (n : integer; var y : Extended);                       ScalbX
        { y := y * 2^n }
procedure LogbX  (var x : Extended);                                    LogbX
        { returns unbiased exponent of x }

{-------------------------------------------------
** Manipulations of the static numeric state.
-------------------------------------------------}

procedure SetRnd  (r : RoundDir);                                       SetRnd

procedure SetEnv  (var e : Environ);                                    SetEnv

function  GetRnd  : RoundDir;                                           GetRnd

procedure GetEnv  (var e : Environ);                                    GetEnv

function  TestXcp (x : Exception) : boolean;                            TestXcp

procedure SetXcp  (x : Exception; OnOff : boolean);                     SetXcp

function  TestHlt (x : Exception) : boolean;                            TestHlt

procedure SetHlt  (x : Exception; OnOff : boolean);                     SetHlt

IMPLEMENTATION

{ ... }

END.
```

APPENDIX D

Pascal Unit for Correctly Rounded Binary-Decimal Conversions

```
UNIT CorrBD;

        {*
        ** Correctly rounded conversions between unpacked binary and
        ** decimal floating-point formats.  Numbers have the form:
        **   (-1)^sign * radix^exp * significand
        ** with an implicit radix point after the first digit (decimal)
        ** or bit (binary).  Numbers need not be normalized in this
        ** unpacked format.  Results are normalized unless underflow
        ** causes denormalization.  Translations between the unpacked
        ** formats are not part of this unit.
        **
        ** Each conversion is governed by an environment record with
        ** rounding and underflow information.  These are dealt with
        ** according to proposed IEEE floating-point standards P754
        ** (binary) and P854 (radix-independent).  That is, underflowed
        ** values are denormalized and overflowed values are set to
        ** either the format's largest value or to the next bigger value
        ** (the latter is intended to represent IEEE infinity).
        **
        ** Version 1.0    17 January 82    Jerome T. Coonen
        *}

INTERFACE

        {*
        ** The constants specify properties of the binary and decimal
        ** formats.  A decimal value is a packed array of BCD digits.
        ** A binary value is a packed array of bytes, with 8 bits per
        ** byte in this implementation.
        **
        ** The constants DEXPMAX and BEXPMAX are not tight bounds.
        ** Rather, they limit the width of the decimal and binary buffers
        ** that must be used to hold input values.  The bounds should
        ** at least cover the range of exponents of all representable
        ** numbers in a NORMALIZED form.
        *}

CONST
        DDIGLEN  =  9;  { max decimal precision }
        DEXPMAX  =  99; { max magnitude of decimal exponent }

        BBITLEN  = 24;  { max binary precision in bits }
        BEXPMAX  = 150; { max magnitude of binary exponent }
        BITSDIG  = 8;   { bits per machine 'digit' (byte) }
        BDIGLEN  = 2;   { max bytes = BBITLEN/ BITSDIG, less 1 }
```

```
        MAXB    = 255; { byte ranges from 0 to 255 }

TYPE
        {*
        ** If space is an issue, these may be redefined as 'packed' records.
        *}
        UnpDec = { unpacked decimal format }
                record
                        sgn : 0..1;
                        exp : -DEXPMAX..DEXPMAX;
                        dig : array [0..DDIGLEN] of 0..9
                end;

        UnpBin = { unpacked binary format }
                record
                        sgn : 0..1;
                        exp : -BEXPMAX..BEXPMAX;
                        dig : array [0..BDIGLEN] of 0..MAXB
                end;

        RDir = (RNEAR, RUP, RDOWN, RZERO);  { rounding directions }

        {*
        ** If style is FloatStyle, pre is the number of significant digits
        ** output; if style is FixedStyle pre is the number, possibly negative,
        ** of fraction digits output.  Because it is presumed that decimal
        ** to binary conversion will only be used to convert to machine types,
        ** type FloatStyle is presumed in the D2BEnv.  In both environment
        ** records, the error flags inexact, uflow, oflow are NOT sticky;
        ** they are set according to the result of the latest conversion.
        *}
        B2DEnv =
                record
                        pre   : integer;
                        style : (FixedStyle, FloatStyle);
                        rnd   : RDir;
                        MinExp : integer;
                        MaxExp : integer;
                        inexact: boolean;
                        uflow : boolean;
                        oflow : boolean
                end;

        D2BEnv =
                record
                        pre   : integer;
                        rnd   : RDir;
                        MinExp : integer;
                        MaxExp : integer;
                        inexact: boolean;
                        uflow : boolean;
                        oflow : boolean
                end;

{*
** Conversions between UnpDec and UnpBin records.  For convenience in
** packing the results of Dec2Bin, if e.pre is not a multiple of
** BITSDIG then the e.pre output bits are right-aligned in the leading
```

```
** ((e.pre div BITSDIG) + 1) bytes of b.dig[]. Of course the implicit
** binary point is still to the right of the first bit of b.dig[0].
*}
procedure Dec2Bin(var e : D2BEnv; d : UnpDec; var b : UnpBin);
procedure Bin2Dec(var e : B2DEnv; b : UnpBin; var d : UnpDec);
```

Dec2Bin

Bin2Dec

```
IMPLEMENTATION

        {*
        ** Constants determining the buffer widths are based on the
        ** interface values. Each buffer must accommodate exactly
        ** any value representable in the respective UnpXXX format,
        ** with several extra digits for rounding.
        *}

CONST
        DBUFLEN = 60;   { DMAXEXP + DDIGLEN + several }
        BBUFLEN = 30;   { (BMAXEXP/ BITSBYT) + BBYTLEN + several }
        MAXB2   = 128;  { MAXB div 2 }

        {*
        ** Binary and decimal values are manipulated in wide byte and
        ** digit buffers. For efficiency, the values head and tail
        ** refer to the most and least significant ends of the 'relevant'
        ** part of the string. An exponent is maintained separately.
        ** Depending on time and space constraints, a DBuf dig may either
        ** be a packed hex nibble (0..4) or a full byte. Though consuming
        ** twice as much space, and unable to take advantage of a computer's
        ** BCD operations in assembly-language support routines, the latter
        ** are much more easily indexed.
        *}

TYPE
        DBuf =
                packed record
                        head : integer;
                        tail : integer;
                        dig : packed array [0..DBUFLEN] of 0..255 { or 0..15 }
                end;

        BBuf =
                packed record
                        head : integer;
                        tail : integer;
                        dig : packed array [0..BBUFLEN] of 0..MAXB
                end;


{*
** Bin2Dec and Dec2Bin employ exactly the same conversion strategies,
** so together they are serviced by corresponding sets of utilities for
** handling DBufs and BBufs. Here is a list of the utilities:
**
** BDZero          -- clear two Bufs to zero.
```

```
** BRight,  DRight  -- shift a Buf right n digs.
** BTimes2, DTimes2 -- Buf * 2.
** BInc           -- add 0-9 in the last dig of a BBuf.
** BTimes10       -- BBuf * 10.
** BWidth         -- find width of a BBuf in bits.
** BUflow,  DUflow -- denormalize a Buf, if necessary, before rounding.
** BRound,  DRound -- round a Buf.
** BOflow,  DOflow -- check and handle Buf overflow, after rounding.
**
** Both Bin2Dec and Dec2Bin require two BBufs and DBufs, a working Buf
** and a temporary for intermediate calculations.  For efficiency, a
** temporary is passed as a var parameter to any utility itself
** requiring a temporary Buf.
*}


{*
** Called by Dec2Bin and Bin2Dec to initialize.
*}
procedure BDZero(var bx : BBuf; var dx : DBuf);                    BDZero
var
        i : integer;
begin
        for i := 0 to BBUFLEN do
                bx.dig[i] := 0;        { set all digs to 0 }
        bx.head := BBUFLEN;        { set head and tail to last dig }
        bx.tail := BBUFLEN;

        for i := 0 to DBUFLEN do
                dx.dig[i] := 0;
        dx.head := DBUFLEN;
        dx.tail := DBUFLEN
end;


{*
** Called by BRound to remove Guard and Sticky bit positions, by BUflow
** to denormalize, and by Dec2Bin to remove excess integer digits.
** bx.head is not updated rightward if all bits are shifted from the
** leading word.  Since bit shifts are only done for the last
** (n mod BITSDIG) bits, this is not a particularly time-consuming
** routine.
*}
procedure BRight(var bx : BBuf; n : integer);                      BRight
var
        i, j, k : integer;
        S : boolean;
begin
        S := false;

        k := n div BITSDIG;  { number of full bytes to be shifted }
        for i := (BBUFLEN - k + 1) to BBUFLEN do
                S := S or (bx.dig[i] <> 0);    { OR doomed bits to S }
        for i := (BBUFLEN - k) downto bx.head do
```

```
                bx.dig[i + k] := bx.dig[i];     { shift right k bytes }
        for i := bx.head to (bx.head + k - 1) do
                bx.dig[i] := 0;                 { clear lead k bytes }

        for i := 1 to (n mod BITSDIG) do
        begin
                S := S or odd(bx.dig[bx.tail]); { record lowest bit }

                for j := BBUFLEN downto (bx.head + k) do
                        if odd(bx.dig[j - 1]) then   { bx.head > 1 here }
                                bx.dig[j] := MAXB2 + (bx.dig[j] div 2)
                        else
                                bx.dig[j] := bx.dig[j] div 2
        end;

        { force sticky bit }
        if S and (not odd(bx.dig[BBUFLEN])) then
                bx.dig[BBUFLEN] := bx.dig[BBUFLEN] + 1
end;


{*
** Called by Bin2Dec to convert integer, Dec2Bin to convert fraction.
** Replace by external assembly-language routine for high speed.
*}
procedure BTimes2(var bx : BBuf);   { external; }                    BTimes2
var
        i, sum, iC : integer;
begin
        iC := 0;  { integer Carry flag }
        for i := bx.tail downto bx.head do
        begin
                sum := bx.dig[i] + bx.dig[i] + iC;
                if sum > MAXB then
                begin
                        iC := 1;
                        bx.dig[i] := sum - (MAXB + 1)
                end
                else
                begin
                        iC := 0;
                        bx.dig[i] := sum
                end
        end;

        if iC <> 0 then  { check for carry out of bx.dig[bx.head] }
        begin
                bx.head := bx.head - 1;
                bx.dig[bx.head] := 1
        end
end;
```

BInc

```
{*
** Called by BRound to add 1 ulp, and by Dec2Bin to add a digit.
** Add  0 <= m <= 9 into BBuf bx by adding m into low byte and
** propagating carry.  Return true if and only if there is a
** carry out of the bx.dig[bx.head].
*}
function BInc(m : integer; var bx : BBuf) : boolean;
var
        i, sum : integer;
        C : boolean;
begin
        BInc := false;          { assume no carry out }
        sum := bx.dig[BBUFLEN] + m;
        if sum <= MAXB then
                bx.dig[BBUFLEN] := sum  { easy case, no carry out }
        else
        begin
                bx.dig[BBUFLEN] := sum - (MAXB + 1);
                C := true;
                i := BBUFLEN;
                while C do
                begin
                        i := i - 1;
                        sum := bx.dig[i] + 1;
                        C := sum > MAXB;
                        if C then
                                bx.dig[i] := 0
                        else
                                bx.dig[i] := sum
                end;

                if i < bx.head then
                begin
                        BInc := true;
                        bx.head := i  { in this case i = bx.head-1 }
                end
        end
end;
```

BTimes10

```
{*
** Called by Bin2Dec to convert fraction digits and by Dec2Bin
** to convert integer digits.  Replace by external assembly-
** language routine for high speed.
*}
procedure BTimes10(var bx : BBuf);  { external; }
var
        i, sum, iC : integer;
begin
        iC := 0;
        for i := bx.tail downto bx.head do
        begin
                sum := (10 * bx.dig[i]) + iC;
                bx.dig[i] := sum mod (MAXB + 1);
                iC := sum div (MAXB + 1)
        end;
        if iC <> 0 then
```

```
        begin
                bx.head := bx.head - 1;
                bx.dig[bx.head] := iC
        end
end;
```

```
{*
** Called by Dec2Bin to determine how many fraction bits to find.
** Lead dig <> 0, since BRight() has not been called yet.
*}
function BWidth(var bx : BBuf) : integer;                    BWidth
var
        i, j : integer;
begin
        { overshoot, as though lead bit of lead dig is 1 }
        i := (BBUFLEN - bx.head + 1) * BITSDIG;

        { correct by decrementing i for leading 0s of leading dig }
        j := bx.dig[bx.head];
        while j < MAXB2 do
        begin
                i := i - 1;
                j := j + j
        end;

        BWidth := i
end;
```

```
{*
** Called by Dec2Bin.
*}
procedure BUflow(var bx : BBuf; var b : UnpBin; var e : D2BEnv);    BUflow
var
        i : integer;
begin
        i := b.exp - e.MinExp;
        if i < 0 then
        begin
                BRight(bx, -i);    { denormalize }
                e.uflow := true;   { mark tiny; BRound determines true Uflow }
                b.exp   := e.MinExp
        end
        else
                e.uflow := false
end;
```

```
{*
** Called by Dec2Bin.
*}
procedure BRound(var bx : BBuf; var b : UnpBin; var e : D2BEnv);    BRound
var
        i, LowDig  : integer;
```

```
        L, G, S, A : boolean;
begin
        { bx has 2 extra trailing bits, Guard and Sticky }
        LowDig := bx.dig[BBUFLEN];
        S := odd(LowDig);
        if S then
                LowDig := LowDig - 1;
        G := odd(LowDig div 2);
        if G then
                LowDig := LowDig - 2;

        L := odd(LowDig div 4);     { least significant bit }
        bx.dig[BBUFLEN] := LowDig;  { replace stripped low byte }
        BRight(bx, 2);              { right-align significand }

        { set inexact flag, and suppress uflow if exact }
        e.inexact := G or S;
        e.uflow   := e.uflow and e.inexact;

        { A := whether to add 1 in L's bit position }
        case e.rnd of
        RZERO: A := false;
        RUP:   A := (b.sgn = 0) and (G or S);
        RDOWN: A := (b.sgn = 1) and (G or S);
        RNEAR: A := G and (S or L)
        end;

        if A then   { add an ULP and check for carry-out }
                if BInc(1, bx) then
                begin
                        BRight(bx, 1);
                        b.exp := b.exp + 1
                end
end;


{*
** Called by Dec2Bin.
** Set to HUGE or INFINITY according to P754/P854 criteria.
** HUGE has maximum exponent and all 1 bits; INFINITY has just
** larger exponent and bits 1000...00
*}
procedure BOflow(var bx : BBuf; var b : UnpBin; var e : D2BEnv);        BOflow
var
        i, fix : integer;
begin
        e.oflow := b.exp > e.MaxExp;
        if e.oflow then
        begin
                e.inexact := true;     { force inexact on any overflow }

                { decide between HUGE and INFINITY }
                if (e.rnd = RNEAR) or ((e.rnd = RUP) and (b.sgn = 0))
                        or ((e.rnd = RDOWN) and (b.sgn = 1)) then
                        fix := 1
                else
                        fix := 0;
```

```
                b.exp := e.MaxExp + fix;    { force excessive exponent }
                BRight(bx, (e.pre - 1));    { clear all but leading 1 }
                for i := 1 to (e.pre - 1) do    { renormalize }
                begin
                        BTimes2(bx);
                        bx.dig[BBUFLEN] := bx.dig[BBUFLEN] + (1 - fix)
                end
        end
end;
```

```
{*
** Called by DUflow to denormalize, by DRound to remove Guard and Sticky
** digit positions, and by Bin2Dec to remove excess integer digits.
** dx.head is not incremented.
*}
procedure DRight(var dx : DBuf; n : integer);                    DRight
var
        i : integer;
        S : boolean;
begin
        S := false;
        for i := (DBUFLEN - n + 1) to DBUFLEN do
                S := S or (dx.dig[i] <> 0);     { OR doomed digits to S }
        for i := (DBUFLEN - n) downto dx.head do
                dx.dig[i + n] := dx.dig[i];     { move right n digits }
        for i := dx.head to (dx.head + n - 1) do
                dx.dig[i] := 0;                 { clear lead n digits }

        if S then
                dx.dig[DBUFLEN] := dx.dig[DBUFLEN] + 1  { OK if > 9 }
end;
```

```
{*
** Called by Bin2Dec to convert integer, by Dec2Bin to convert fraction.
** Replace by external assembly-language routine for high speed.
*}
procedure  DTimes2(var dx : DBuf);  { external; }                DTimes2
var
        i, sum, iC : integer;
begin
        iC := 0;  { integer Carry flag }
        for i := dx.tail downto dx.head do
        begin
                sum := dx.dig[i] + dx.dig[i] + iC;
                if sum > 9 then
                begin
                        iC := 1;
                        dx.dig[i] := sum - 10
                end
                else
                begin
                        iC := 0;
```

```
                        dx.dig[i] := sum
                end
        end;

        if iC <> 0 then { check for carry out of dx.dig[dx.head] }
        begin
                dx.head := dx.head - 1;
                dx.dig[dx.head] := 1
        end
end;
```

DUflow

```
{*
** Called by Bin2Dec.
*}
procedure DUflow(var dx : DBuf; var d : UnpDec; var e : B2DEnv);
var
        i : integer;
begin
        i := d.exp - e.MinExp;
        if i < 0 then
        begin
                DRight(dx, -i);   { denormalize }
                e.uflow := true;  { mark tiny; DRound determines true Uflow }
                d.exp   := e.MinExp
        end
        else
                e.uflow := false
end;
```

DRound

```
{*
** Called by Bin2Dec.
*}
procedure DRound(var dx : DBuf; var d : UnpDec; var e : B2DEnv);
var
        i, iG, sum : integer;
        L, S, A    : boolean;
begin
        { dx has 2 extra trailing digits, Guard and Sticky, to be ignored }
        S := dx.dig[DBUFLEN] <> 0;
        iG := dx.dig[DBUFLEN - 1];
        L := odd(dx.dig[DBUFLEN - 2]);  { low bit of LSD }

        { set inexact flag, and suppress uflow if exact }
        e.inexact := (iG <> 0) or S;
        e.uflow   := e.uflow and e.inexact;

        { A := whether to add 1 in L's bit position }
        case e.rnd of
        RZERO: A := false;
        RUP:   A := (d.sgn = 0) and ((iG <> 0) or S);
        RDOWN: A := (d.sgn = 1) and ((iG <> 0) or S);
        RNEAR: A := (iG > 5) or ((iG = 5) and (L or S))
```

```
        end;

        if A then   { add an ULP and check for carry-out }
        begin
                S := true;          { use to propagate carry }
                i := DBUFLEN - 1;       { will discard low 2 digits }
                while S do
                begin
                        i := i - 1;
                        sum := dx.dig[i] + 1;
                        S := sum > 9;
                        if S then
                                dx.dig[i] := 0
                        else
                                dx.dig[i] := sum
                end;

                if (i < dx.head) then
                        if (e.style = FloatStyle) then
                        begin
                                dx.dig[dx.head] := 1;       { carry out at left }
                                d.exp := d.exp + 1
                        end
                        else
                                dx.head := i
        end
end;


{*
** Called by Bin2Dec.
** Set to HUGE or INFINITY according to P754/P854 criteria.
** HUGE has maximum exponent and all nines; INFINITY has just
** larger exponent and decimal digits 1000...00.
*}
procedure DOflow(var dx : DBuf; var d : UnpDec; var e : B2DEnv);          DOflow
var
        i, fix : integer;
begin
        e.oflow := d.exp > e.MaxExp;
        if e.oflow then
        begin
                e.inexact := true;      { force inexact on any overflow }

                { decide between HUGE and INFINITY }
                if (e.rnd = RNEAR) or ((e.rnd = RUP) and (d.sgn = 0))
                        or ((e.rnd = RDOWN) and (d.sgn = 1)) then
                        fix := 0
                else
                        fix := 1;

                d.exp := e.MaxExp + 1 - fix;       { force big exponent }
                dx.dig[dx.head] := (8 * fix) + 1;  { either 9 or 1 }
                for i := (dx.head + 1) to (DBUFLEN - 2) do
```

```
                    dx.dig[i] := 9 * fix        { either 9 or 0 }
        end
end;


{*
** Both conversions Bin2Dec and Dec2Bin follow the same strategy:
**
** (0) If input has all zero digits, then the resulut is 0; else...
**
** (1) Align input in Buf as 0.XXXXXXX * RADIX^exp, with dig[0] = 0
**    and the significand shifted far enough right that exp >= 0.
**
** (2) Convert integer part, that is until exp = 0.
**
** (3) If no nonzero output digit has been found, then convert
**    the fraction up to the first nonzero digit.
**
** (4) The object is to have exactly p+2 significant digits/bits,
**    the last one sticky in the sense of P754 rounding.  If there
**    are too many already, then right shift and gather lost digits
**    in sticky; otherwise, convert until there are just p+2.
**    Gather unconverted digits/bits into sticky.
**
** (5) If result is tiny in the sense of P754, then right shift
**    (denormalize) it until the exponent is the minimum allowed.
**
** (6) Round the result to p digits/bits.
**
** (7) Deal with overflow according to P754, that is, replacing an
**    overflowed result with either INFINITY or HUGE.
**
** Both conversions align their input to the left of a Buf, up to
** dig[0], and form their output aligned to the right in its Buf.
**
** The conversions set flags inexact, oflow, and uflow in the
** environment record according to P754, except that the flags are
** NOT STICKY.  A full P754 system would 'logically OR' these flags
** into the system's true exception flags after each conversion.
**
** A P754 trapping mechanism is not supported here.
*}

procedure Bin2Dec { (var e : B2DEnv; b : UnpBin; var d : UnpDec) } ;   Bin2Dec
var
        i, j, BExp : integer;
        S : boolean;
        bx : BBuf;
        dx : DBuf;

begin
        d.sgn := b.sgn;    { copy sign }
        for i := 0 to DDIGLEN do    { place all zero digits }
                d.dig[i] := 0;
```

```
{ Step 0: check for all zeros. }
S := true;   { assume the significand is zero }
for i := 0 to BDIGLEN do
        S := S and (b.dig[i] = 0);

if S then   { process zero }
        d.exp := e.MinExp

else
begin
        BExp := b.exp + 1;    { align binary point left of lead bit }
        if BExp >= 0 then     { significand in dig[(0+j)...] }
                j := 1
        else
                j := 2 - (BExp div BITSDIG);

        { Step 1: set bx to input b, aligned. }
        BDZero(bx, dx);
        bx.head := 1;
        bx.tail := BDIGLEN + j;
        for i := 0 to BDIGLEN do
                bx.dig[i+j] := b.dig[i];

        { Adjust BExp < 0, since bx shifted right to the nearest byte. }
        BExp := (BITSDIG * (j - 1)) + BExp;  { j=1 when BExp >= 0 }

        d.exp :=  e.pre + 1;  { dec point after lead dig, then G and S }

        { Step 2: convert integer part of bx. }
        while BExp > 0 do
        begin
                DTimes2(dx);    { make way for the next bit }
                BTimes2(bx);    { get next bit in bx.dig[0] }
                BExp := BExp - 1;
                if bx.dig[0] <> 0 then
                begin
                        dx.dig[DBUFLEN] := dx.dig[DBUFLEN] + 1;
                        bx.dig[0] := 0
                end
        end;

        { Step 3: guarantee some nonzero digit in dx. }
        while dx.dig[dx.head] = 0 do
        begin
                BTimes10(bx);
                dx.dig[DBUFLEN] := bx.dig[0];
                d.exp := d.exp - 1
        end;
        bx.dig[0] := 0;

        { Step 4: check for too many or too few digits. }
        if e.style = FloatStyle then
                j := (DBUFLEN - dx.head + 1) - (e.pre + 2)
        else
                j := -e.pre;  { number of 'fraction' digits }
```

```
                if j < 0 then   { j too few digits }
                begin
                        for i := dx.head to DBUFLEN do
                        begin   { make room for -j more digits }
                                dx.dig[i + j] := dx.dig[i];
                                dx.dig[i]  := 0
                        end;
                        dx.head := dx.head + j;

                        for i := (DBUFLEN + 1 + j) to DBUFLEN do
                        begin   { get -j fraction digits }
                                BTimes10(bx);
                                dx.dig[i] := bx.dig[0];
                                bx.dig[0] := 0
                        end
                end

                else   { j too many digits already }
                begin
                        DRight(dx, j);
                        dx.head := dx.head + j
                end;

                { Fix exp for j-char shift. }
                d.exp  := d.exp + j;

                S := false;
                for i := bx.head to bx.tail do
                        S := S or (bx.dig[i] <> 0);  { unconverted bits --> sticky }
                if S then
                        dx.dig[DBUFLEN] := dx.dig[DBUFLEN] + 1;

                DUflow(dx, d, e);
                DRound(dx, d, e);
                DOflow(dx, d, e);

                for i := dx.head to (DBUFLEN - 2) do
                        d.dig[i - dx.head] := dx.dig[i]
        end
end;


procedure Dec2Bin { (var e : D2BEnv; d : UnpDec; var b : UnpBin) } ;    Dec2Bin
var
        i, j, k, DExp : integer;
        S : boolean;
        bx : BBuf;
        dx : DBuf;

begin
        b.sgn := d.sgn;   { copy sign }
        for i := 0 to BDIGLEN do   { place all zero bits }
                b.dig[i] := 0;
```

```
{ Step 0: check for all zeros. }
S := true;   { assume the significand is zero }
for i := 0 to DDIGLEN do
        S := S and (d.dig[i] = 0);

if S then   { process zero }
        b.exp := e.MinExp

else
begin
        { Steps 1 and 2: convert integer part and align fraction in dx. }
        BDZero(bx, dx);        { initialize bx and dx }
        b.exp :=  e.pre + 1;   { dec point after lead dig, then G and S }
        DExp := d.exp + 1;     { align binary point before dig[0] }

        if DExp >= 0 then
        begin
                for i := 0 to (DExp - 1) do   { compute integer part }
                begin
                        BTimes10(bx);
                        if i <= DDIGLEN then
                                S := BInc(d.dig[i], bx)
                                { but ignore carry-out S }
                end;

                j := DExp    { index of first fraction digit }
        end
        else
                j := 0;      { index of first fraction digit }

        for i := j to DDIGLEN do   { align fraction digits }
                dx.dig[i + 1 - DExp] := d.dig[i];

        dx.head := 1;
        dx.tail := DDIGLEN + 1 - DExp;
        if dx.tail < dx.head then
                dx.tail := dx.head;

        { Step 3: guarantee some nonzero digit in bx. }
        while bx.dig[bx.head] = 0 do
        begin
                DTimes2(dx);
                bx.dig[bx.head] := dx.dig[0];
                b.exp := b.exp - 1
        end;
        dx.dig[0] := 0;

        { Step 4: check for too many or too few bits. }
        j := BWidth(bx) - (e.pre + 2);

        if j < 0 then   { -j too few bits }
        begin
                for i := 1 to -j do
                begin
                        BTimes2(bx);   { make room for fraction bit }
```

```
                                DTimes2(dx);   { next fraction bit in dig[0] }
                                bx.dig[BBUFLEN] := bx.dig[BBUFLEN] + dx.dig[0];
                                dx.dig[0] := 0
                        end
                end

                else   { j too many bits already }
                        BRight(bx, j);

                { Final adjustments according to shift above. }
                b.exp  := b.exp + j;

                S := false;
                for i := dx.head to dx.tail do
                        S := S or (dx.dig[i] <> 0);
                        { unconverted digits --> sticky }
                if S and (not odd(bx.dig[BBUFLEN])) then
                        bx.dig[BBUFLEN] := bx.dig[BBUFLEN] + 1;

                BUflow(bx, b, e);
                BRound(bx, b, e);
                BOflow(bx, b, e);

                { Finally, store trailing e.pre bits, right adjusted. }
                { Fix exponent for possible leading 0s in first byte. }
                j := e.pre mod BITSDIG;
                if j <> 0 then
                        b.exp := b.exp + (BITSDIG - j);
                j := bx.tail - ((e.pre - 1) div BITSDIG);
                for i := j to bx.tail do
                        b.dig[i - j] := bx.dig[i]
        end
end;

END. { of unit CorrBD }
```

```
{*
** Convert between CorrBD Bin and P754 types S, D, E assuming a byte
** ordering in which less significant bytes are at lower addresses.
*}
UNIT FormBD;

INTERFACE

uses FPSoft, CorrBD;

procedure S2Bin(s : Single;   var b : UnpBin);                    S2Bin
procedure D2Bin(d : Double;   var b : UnpBin);                    D2Bin
procedure E2Bin(e : Extended; var b : UnpBin);                    E2Bin
procedure Bin2S(b : UnpBin; var s : Single  );                    Bin2S
procedure Bin2D(b : UnpBin; var d : Double  );                    Bin2D
procedure Bin2E(b : UnpBin; var e : Extended);                    Bin2E


IMPLEMENTATION

type
        SByte =
        record
                case char of
                's' : (s : Single);
                'b' : (b : packed array [0..3] of 0..255)
        end;

        DByte =
        record
                case char of
                'd' : (d : Double);
                'b' : (b : packed array [0..7] of 0..255)
        end;

        EByte =
        record
                case char of
                'e' : (e : Extended);
                'b' : (b : packed array [0..9] of 0..255)
        end;


{*
** Unit CorrBD leaves the bits in UnpBin right aligned so that no shifting
** is required when they are moved to the P754 packed types.  However,
** the exponent field must be modified to account for any leading zeros.
*}
```

D2Bin

```
procedure D2Bin { (d : Double;  var b : UnpBin) };
var
        t : DByte;
        i : integer;
begin
        t.d := d;
        b.sgn := t.b[7] div 128;   { sign }
        b.exp := ((t.b[7] mod 128) * 16) + (t.b[6] div 16) - 1023;

        for i := 0 to BDIGLEN do
                b.dig[i] := 0;
        b.dig[0] := t.b[6] mod 16;
        for i := 1 to 6 do
                b.dig[i] := t.b[6-i];

        if b.exp = -1023 then
                b.exp := b.exp + 1  { correct bias of minimum exp }
        else
                b.dig[0] := b.dig[0] + 16  {force explicit leading 1 }
end;
```

Bin2E

```
procedure Bin2E { (b : UnpBin; var e : Extended) };
var
        t : EByte;
        i, k : integer;
begin
        k := b.exp + 16383;   { biased exponent }
        t.b[9] := (128 * b.sgn) + (k div 256);
        t.b[8] := k mod 256;
        for i := 7 downto 0 do
                t.b[i] := b.dig[7-i];
        e := t.e
end;
```

E2Bin

```
procedure E2Bin { (e : Extended; var b : UnpBin) };
var
        t : EByte;
        i : integer;
begin
        t.e := e;
        b.sgn := t.b[9] div 128;
        b.exp := ((t.b[9] mod 128) * 256) + t.b[8] - 16383;
        for i := 0 to BDIGLEN do
                b.dig[i] := 0;
        for i := 0 to 7 do
                b.dig[i] := t.b[7-i]
end;
```

END. { of unit FormBD }