SPECIFICATIONS FOR A PROPOSED STANDARD FOR FLOATING POINT ARITHMETIC

by

J. T. Coonen

Memorandum No. UCB/ERL M78/72

13 October 1978

SPECIFICATIONS FOR A PROPOSED STANDARD

FOR FLOATING POINT ARITHMETIC

by

J.T. Coonen
Department of Mathematics
University of California

Memorandum No. UCB/ERL M78/72

13 October 1978

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Errata in Memorandum No. UCB/ERL M78/72 October 23, 1978

p.16 Maximum positive normalized:

SINGLE $2^{127} * (2-2^{-23})$ DOUBLE $2^{1023} * (2-2^{-52})$

p. 19 a.

+	-0	+0
-0	-0	A
+0	A	+0

A. +0 in rounding modes RN, RZ, RP and -0 in rounding mode RM.

p. 20 4.i) If all bits of the (unrounded) significand are zero, set the sign to "+" in rounding modes RN, RZ, RP, and set the sign to "-" in mode RM, as in case A of (a) above. The result is true zero, i.e. the exponent is set to its most negative value.

p. 28 Replace the first line of the page with:

FLOATING_TO_INTEGER

This instruction converts a floating point integer X into a binary integer of the host processor.

p. 36 6) Should $\sqrt{-5} = -\sqrt{5}$ if trapped? or NAN?

8) Should the over/underflow bias adjust be $3*2^{n-2}$ rather than $3*2^{n-2} - 2$, where n is the number of bits in the exponent field?

INTRODUCTION

This draft standard is one of several before an IEEE subcommittee whose goal is to standardize floating point arithmetic for mini- and micro-computers. This is the seventh in a series of drafts, the first of which was presented by Harold Stone, William Kahan and Jerome Coonen in May, 1978. The present document benefits from extensive discussions in previous meetings of the IEEE subcommittee. The author wishes to acknowledge the help of Professor William Kahan in preparing this draft.

This proposal specifies meticulously its data formats and its very complete complement of arithmetic operations down to the last bit. It must do so to be of use to the producers of micro-processor hardware and software, who cannot afford to provide the support software and personnel to perform conversions between systems conforming to a less rigid standard. The present standard would allow communication between systems at the data level without conversions.

While this standard is precise in its specification of the results of arithmetic operations, it must be flexible enough to accommodate a variety of computer architectures (e.g. stack, multi-register, single-accumulator, storage-to-storage, dedicated auxiliary processor,...) and several possible levels of implementation (SINGLE precision only, both SINGLE and DOUBLE precisions,...). At the same time it must allow for future developments; some of them, hitherto precluded by ill-advised aspects of current designs, should instead be supported by the new system's design in so far that support is achievable without an excessive burden upon the performance of today's tasks. Among the necessary developments are:

> Interval Arithmetic, which provides a certifiable result despite roundoff and over/underflow and other exceptions; and
>
> A degree of collaboration between numerical (approximate) procedures on the one hand and automatic symbolic algebraic (exact) procedures on the other; and
>
> The use of reserved operands to extend the numerical data structure, say with complex infinities, or pointers into heaps of numbers with extended range and precision.

The new standard takes great care in the handling of exceptional conditions such as over/underflow. An attempt has been made to achieve a higher level of safety than has been customary, with enhanced utility but without excessive cost. This will impact existing software in the following way. Programs, which run now 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.

This work was partially funded by Office of Naval Research Contract N00014-76-C-0013.

PART I: NARRATIVE DESCRIPTION OF THE STANDARD ARITHMETIC

BASIC LEVELS OF PRECISION

Any nonzero real number x may be expressed as $x = \pm 2^e * f$, where e is a signed integer and $1 \leq f < 2$. We call this the "floating point" representation of x, with exponent e and significant digit field f. Our object is to describe a machine representation for real numbers based on this floating point decomposition, and to prescribe rules for arithmetic on these numbers. In this scheme, e and f will be represented by bit strings of prescribed lengths so that necesarily only a finite subset of the real numbers will be representable exactly.

This standard for floating point arithmetic admits two basic levels of precision, SINGLE and DOUBLE. It is possible that a third, QUAD, will be added. An implementation of this standard may provide SINGLE only, or both SINGLE and DOUBLE precisions. We require SINGLE in all standard systems, recognizing both its value as a debugging precision and its efficiency for a wide range of applications where storage economy matters.

We will discuss only SINGLE here, referring the reader to PART II for the analogous details of DOUBLE. A normalized nonzero number X in SINGLE precision has the form

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

S = sign bit = bit string of length one
E = biased exponent = bit string of length 8 encoding integers in the range 0 to 255
F = significand = bit string of length 23 encoding those of X's significant bits that follow the binary point, yielding a 24 bit significant digit field for X that always begins "1.___".

In terms of the above decomposition of X as a normalized number we have the following relations provided $0 < E < 255$:

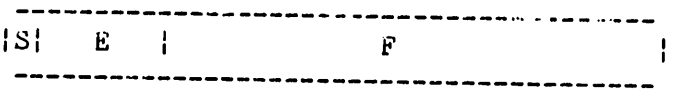
sign of X = $(-1)^S$
e = E-127 = X's unbiased exponent
f = 1.F = X's significant digit field.

Note that the leading 1 bit of X's the significant digit field f is "implicit" in the storage representation. The values 0 and 255 of E are reserved to designate special operands to be discussed in later sections. It may at least be noted here that signed zero is represented by E = F = 0. With this constraint on E, a normalized nonzero SINGLE precision number X can range in magnitude between

$$2^{-126} = 2^{-126} * 1.000...00 \text{ and } 2^{127} * 1.111...11 = 2^{128}$$

inclusive.

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



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

The normalized nonzero SINGLE precision numbers are elements of a finite model for the real number system, a model we shall develop further in the following sections. In this model the normalized nonzero SINGLE precision numbers simply represent themselves as infinite precision real numbers.

ARITHMETIC OPERATIONS

The following arithmetic operations are completely described by this standard:

ADD	REMainder	INTeger_part
SUBtract	CoMPare	Floating_to_INTeger
MULtiply	MOVe	INTeger_to_Floating
DIVide	SQuare_RooT	

binary_INTeger_to_DECimal_string
decimal_STRing_to_BINary_integer
BINary_floating_to_DECimal_floating
DECimal_floating_to_BINary_floating

The description is complete in the sense that, given operands at any levels of precision, and given the precision desired for the result, the result is specified by the standard. We refer the reader to PART II for tables detailing the operations.

An implementation of this standard must at least provide

1. ADD, SUB, MUL, DIV and REM for any two operands of the same precision, for each supported precision, with the result having no less exponent range than the two operands.

2. CMP and MOV for operands at any, perhaps different, supported levels of precision (in the case of MOV the second operand is the destination).

3. INT and SQRT for operands at all supported levels of precision, producing results having no less exponent range than the input operands.

4. Conversions between floating point integers and binary integers in the host processor.

5. Binary_decimal conversions to and from all supported levels of precision. These conversions are supported by conversions between floating point integers and decimal strings. A section of PART II is devoted to the details of these operations.

This standard provides a notably complete set of arithmetic operations 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. SQRT is included as a primitive operation because it appears so often, for example in matrix calculations, and because,

when implemented in hardware according to published algorithms its execution time is comparable with that of a divide. In many current systems, SQRT is implemented too slowly and/or too inaccurately, so that unnecessary effort is expended coding around it. The same applies in the case of REM. We point out that REM is preferable to the MOD function because REM is computed without rounding error. Consider, for example

0.01 MOD (-95) versus 0.01 REM (-95)

on a 2-digit machine. MOD yields the result round(-94.99) = -95 for a complete loss of accuracy while REM yields the correct result 0.01. The standard's insistence upon standard binary-decimal conversions is both a reflection of the delicacy of the calculations involved and an attempt to allow comparison of data from different systems at the decimal output level rather than via hexadecimal dumps.

ACCURACY AND ROUNDING

The operations ADD, SUB, MUL, DIV, REM, INT, and SQRT are presumed to deliver their results to destinations having no less exponent range than their input operands. Besides simplifying the narrative, this constraint avoids unnecessary complexity in the implementation. Systems which implemented the rare operation

DOUBLE * DOUBLE --> SINGLE

directly instead of indirectly via

DOUBLE * DOUBLE --> DOUBLE; MOVE (ROUND) DOUBLE --> SINGLE

would enjoy a slight advantage in speed and rounding error at the cost of a richer instruction set and considerably more complicated responses to over/underflow. There are better ways.

The simplest systems may restrict their instruction sets to allow no mixed-precision operations besides MOVE and COMPARE, providing only

SINGLE * SINGLE --> SINGLE and DOUBLE * DOUBLE --> DOUBLE,

but they would not support efficient numerical computation adequately. There are many occasions when constructions like

SINGLE + SINGLE --> DOUBLE or

(SINGLE * SINGLE --> DOUBLE) + DOUBLE --> DOUBLE

ought to be used in innermost loops, and then the costs of "padding" like

(SINGLE --> DOUBLE) + DOUBLE --> DOUBLE or

(SINGLE --> DOUBLE) * (SINGLE --> DOUBLE) + DOUBLE --> DOUBLE

respectively become unnecessary nuisances. Rather than prohibit mixed-precision operations, the standard is designed to encourage the provision of some such operations, though it is unreasonable to expect operations that cater to every possible combination of input types and yield results at every supported level of precision.

What is needed is control over the precision to which a result is rounded independent of the ostensible precisions of the input operands. This is achieved in a standard system in either or both of two ways. One way is to specify the width of the destination subject to the constraint, mentioned above, that the destination be no narrower than the operation's input operands (except for the operations MOVE and COMPARE). The second way is to specify in advance the precision to which a result is to be rounded subject to the constraint that that precision be not too wide for the intended destination. For instance, the standard allows for operations of the form

((SINGLE * DOUBLE) rounded to SINGLE PRECISION) --> DOUBLE.

Operations like that are appropriate for a system which, in an attempt to economize on the size of its instruction set, delivers all results except those of MOVE and COMPARE only to the widest destinations supported by the system. Such a system would encourage a healthy practice, namely the preservation of intermediate results for a long expression in the widest available precision, with just one serious rounding error at the end when the expression's value is stored in a narrower destination. But that healthy practice is practically precluded at present by ill-considered constraints built into current programming languages. Therefore, the standard's specifications for roundoff control are burdened by the complications necessary to provide, on the one hand, compatibility with past practice however ill-advised, and on the other hand an opportunity for better procedures in the future.

If the result of one of the arithmetic operations, when computed to infinite precision, is exactly representable within the exponent range and precision specified for the result, then it must be given exactly. Otherwise the result must be rounded according to one of the schemes to be presented in this section. In any case, the error will be less than one unit in the last place to which the result is rounded unless Over/Underflow or some other such exception intervenes.

To illustrate the rounding modes we let z be the infinitely precise result of an arithmetic operation. Then we determine $Z1$ and $Z2$, numbers representable exactly with the precision of the intended destination field, such that $Z1$ and $Z2$ most closely bracket z, that is $Z1 \leq z \leq Z2$, barely. Certain details concerning the exponent range and the alignment of the binary point of $Z1$ and $Z2$ are deferred to PART II. We then have:

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_to_Zero(z) = Chop(z) = smaller of Z1 and Z2 in magnitude.

Round_to_Plus_Infinity(z) = Z2.
Round_to_Minus_Infinity(z) = Z1.
} directed roundings

As noted, the latter two rounding modes are often referred to as the "directed roundings". They are intended to support Interval Arithmetic and to calculate certifiable upper and lower bounds, and to control conversion to integers. Round_to_Zero is useful too in controlling conversions to integers in accordance with conventions embedded in certain progamming languages like FORTRAN.

An implementation of the standard may support either of the following combinations of rounding modes:

1. Round_to_Nearest only, with Round_to_Zero for certain specified integer operations.
2. All four rounding modes.

If more than one rounding mode is supported then Round_to_Nearest shall be the default mode for all operations. Despite the apparent redundancy of rounding to Zero, Plus_Infinity and Minus_Infinity, both directed roundings are required if either is implemented. It is a false economy to attempt a saving in the number of rounding modes implemented, because codes are much simpler when given direct access to all rounding modes. In interval arithmetic, for example, one often computes the upper and lower bounds of an interval by executing the same sequence of instructions twice, rounding up during one pass and down the next.

Calculation of Round_to_Nearest requires the so-called Sticky Bit, as shown in PART II. 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, e.g. mode bits or extra opcodes, for exercising the choice of rounding mode. While the standard leaves this mechanism up to the implementer, we remark that the mode bits are usually preferable. For example, consider the interval arithmetic computation of the previous paragraph. This task is greatly expedited if changing rounding modes from one pass to the next is simply a matter of flipping a pair of mode bits.

EXTENDED PRECISION

To perform the arithmetic operations on numbers stored in SINGLE or DOUBLE pecision, a system will generally "unpack" the bit strings into their component fields

S = sign
E = biased exponent
F = significand.

Moreover, the leading bit of the significand will be made explicit, and perhaps the bias will be removed from the exponent. Since most current machines use data paths of widths 4, 8, 16, 32 or 64, the 24-bit SINGLE significand, with explicit leading bit, will probably be unpacked into a 32-bit field. Similarly, the 53-bit DOUBLE significand may be dealt with in a 64-bit field.

The standard provides a way to exploit this unpacked format, by admitting SINGLE_EXTENDED and DOUBLE_EXTENDED precisions; perhaps QUAD_EXTENDED will be added later. Support of the extended precisions is optional. If implemented at all, only one extended precision shall be provided, namely that corresponding to the widest basic precision supported.

As with the basic precisions, we will describe SINGLE_EXTENDED here, referring the reader to PART II for the full details of the extended precisions. SINGLE_EXTENDED is comprised of the fields

S = sign = bit string of length one
E = unbiased exponent = a signed integer of at least 11 bits
F = significand = a bit string of length at least 32 bits, with an explicit leading bit followed by an implicit binary point.

A number X encoded in SINGLE_EXTENDED is then given by

$$X = (-1)^S * 2^E * F \quad ,$$

except possibly when E takes its most positive value. Signed zero is given by E = most-negative-value and F = 0.

The exponent width is so chosen to provide at least the range of DOUBLE precision. The most positive value of the exponent is reserved for the encoding of special operands to be discussed later. Having at least eight extra significand bits greatly simplifies the accurate computation of the trigonometric, logarithm and exponential functions, and the power function Y^X, to full SINGLE precision. Matrix calculations also benefit from SINGLE_EXTENDED accumulations of products of SINGLE precision data. Moreover, the extra bits of precision are so important in binary-decimal conversions that some extended capability must be simulated by system software if extended precision is not implemented; this is discussed in PART II.

If implemented, extended entities are assumed to be few in number, used to evaluate complicated subexpressions, for example. They are not intended to be indexed in arrays in higher-level languages.

Another way to obtain most of the computational benefits of extended precision is by using the "next higher" basic level of precision. Indeed, QUAD may be included in the standard solely as an alternative for those not wishing to implement DOUBLE_EXTENDED in a system with SINGLE and DOUBLE. One important difference between the basic and extended precisions is the leading significand bit, which is explicit only in the extended precisions. The section on treatment of Underflows will indicate how special classes of unnormalized numbers arise in the basic precisions. In extended precision, on the other hand, the explicit leading significand bit allows encoding of unnormalized numbers over the entire exponent range (except, of course, the reserved value). Thus EXTENDED is a more flexible way to get extended range than is the next higher basic level of precision, but it is less precise. Moreover, in most implementations EXTENDED will be as fast as the precision it supports, as compared to the factor of 2 or 4 loss in speed realized by the next higher basic level of precision, if implemented.

EXCEPTIONS

So far we have described the basic and optional extended precisions for encoding real numbers, and we have specified a large family of arithmetic operations on them. This is all quite straightforward, given the word sizes of current machines and the needs of people seeking floating point capability. A much more interesting question remains, with regard to the exceptional conditions that arise during arithmetic operations -- how are the responses to exceptions to be standardized? The remainder of this narrative addresses this question.

The standard organizes the exception conditions under the five headings:

Invalid_Operation
Underflow
Overflow
Invalid_Division
Inexact_Result

The following sections treat these conditions individually, ultimately prescribing the results, if any, to be delivered in each instance. It is important to note that these results, as standard system responses, are independent of whether the standard is implemented entirely in hardware or software, or in a combination of the two. Certain diagnostic information passed via "results" is necessarily implementation-dependent; however, in the context of a given system, the results are uniquely determined by the standard.

For each of the five exceptions, an implementation of the standard may

a. Force a trap to user software.

or

b. Deliver a result specified by the standard and proceed.

or

c. Provide the user with a Trap_Enable bit whereby to choose (a) or (b), i.e. whether or not to trap.

Whenever a choice is given, the default shall be to proceed without a trap. The standard provides a precedence rule to determine which single trap is to be invoked in case several exceptions occur simultaneously.

Associated with each of the exceptions is a "sticky" flag which is set on each occurrence of the corresponding exception, regardless of the system response. Each flag may be tested by a program to determine whether an exception has occurred. Following an exception, a flag remains set until cleared by the user's program (or programming environment). In certain instances, e.g. when the end of a job is obviously at hand, a humane operating system may draw the user's attention to flags still set, thereby perhaps reminding him of exceptions that were overlooked by his program.

To deal effectively with traps, programmers need access to certain vital information, ideally:

What event caused the trap?
Where in the program?
What did the instruction try to do (what opcode)?
What were the operands (source and destination)?

In response, the programmmer will normally either:

Depart from the offending block of code to try something utterly else.
Fix up the aberrant result and resume execution after the offending instruction.
Fix up the aberrant operands and re-execute the offending instruction.

Sometimes the full range of information and responses is not needed, especially when the correct result is available, possibly in encoded form as in the case of Over/Underflow. One might dispense with some of the above

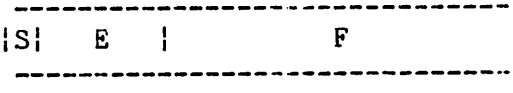
information in these cases.

INVALID_OPERATION

The Invalid_Operation exception encompasses problems arising in a variety of arithmetic operations; it is the blanket covering those errors not frequent or important enough to merit their own fault condition. Here are examples of Invalid_Operations:

-- $\sqrt{-5}$
-- 0/0 with the Invalid_Division trap disabled
-- plus_infinity + minus_infinity
(the infinities will be introduced later)
-- Attempted arithmetic with a designated reserved operand (these "Not-A-Numbers" will be introduced below).

We see that some invalid operations, like 0/0, cannot deliver a numerical result that would be reasonable in all circumstances. For these situations we utilize one class of reserved operands, the Not-A-Number symbols, or NANs. In SINGLE and DOUBLE precisions, with the format



,

NANs are characterized by

S = sign bit (it may be irrelevant)
E = 111...11
F not= 0.

In extended precision, NANs have the most positive exponent. 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 significand field F of a NAN will contain system-dependent information, for example:

a. A distinguished class of NANs, say with two leading zero significand bits, may be used by an operating system to initialize storage. The significand of such a NAN may be a name or a pointer to the region where the NAN is stored. As we will see below, these NANs will propagate through arithmetic operations, ultimately providing pointers to those areas of user-uninitialized storage which are the ancestors of meaningless final results.

b. A NAN generated by an invalid arithmetic operation on numeric data, for example 0 * infinity, 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 infinity as a line rather than a point in the projective plane. A distinguished class of NANs, say those with two leading one bits in the significand, may be used in pairs to provide the relative sizes and signs of the real and imaginary parts of numbers tending to infinity along a

fixed ray emanating from the origin. While these entities have the format of NANs, they contain signed numeric data and would be handled in software invoked by traps.

d. Sometimes an operation could generate an acceptable result 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 as the "result", with a significand pointing to a place, e.g. an extended field or a heap in storage, 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 which point 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 precision NANs to point to DOUBLE numerical data.

As elements of our model of the real numbers, the NANs are extensions of the real number system. Their role in arithmetic operations is quite simple. While certain classes of NANs, for example those in (c) above, will cause an invalid_operation exception when picked up as operands, NANs will generally propagate through arithmetic operations without generating exceptions. For example 5 + n --> n if n is a NAN. If two NANs are picked up as operands, the one with the smaller significand has precedence; this is more precisely specified in the Appendix.

We now specify system action on Invalid_Operation exceptions. If no trap is to be taken then the result of any Invalid_Operation is a NAN bearing some system-dependent information.

If an Invalid_Operaton occurs and a trap is to be taken, the result, if any, to be delivered is highly machine-dependent as well as operation-dependent. In some implementations, the trap will effectively occur before the operation is carried out, so no result need be written into the destination field. On the other hand, the trap may be invoked too late by some machines, i.e. after some result is produced and delivered. In this case the usual result is a NAN, though an implementation may, in certain situations, deliver a numeric result, for example it may make sense to deliver $-\sqrt{5}$ in place of $\sqrt{-5}$; these special cases are noted in PART II.

UNDERFLOW

Exponent Underflow is the most interesting of the exceptions because of the care taken by the standard to provide as much information as possible when proceeding without a trap. In the case of Overflow, on the other hand, a bare minimum of information is passed; this is discussed in the next section. For this reason, the range of normalized numbers in SINGLE and DOUBLE precisions 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 significand in such a way that the product of the largest and smallest positive normalized numbers is roughly 4 in each of the basic levels of precision.

We now discuss the treatment of Underflows. In each case we let z be

the infinitely precise nonzero result of an arithmetic operation and we let z have the form

$$z = \pm 2^{e} * f \qquad \text{where}$$

e is a signed integer and $1 \leq f < 2$. We assume no prior knowledge about e. Note that before rounding z we check whether or not we'll trap on Underflow, if it occurs.

If a trap is to be taken on Underflow, then z is rounded to the precision of the destination field. If the exponent lies below the exponent range of the destination field then Underflow occurs. Because of the restrictions on arithmetic operations presumed under "ACCURACY AND ROUNDING", the exponent can be out of range by at most a factor of two, except for the MOV instruction which is discussed in PART II. The exponent is wrapped around into the desired range with a bias adjust specified in PART II. The result is then delivered to the destination and the trap is invoked.

If no trap is to be taken on Underflow, then the exponent of our infinitely precise result z is tested before rounding. If it lies below the minimum possible exponent of the destination field, then z is "denormalized", that is:

> the significant digit field of z is right-shifted while z's exponent is incremented until it reaches the minimum possible exponent of the destination.

Then z is rounded to the precision of the destination field, and the result is delivered, in a manner to be described presently. Unless z rounds up in magnitude to the smallest nonzero normalized number, Underflow is signalled.

To illustrate the denormalization process let us consider an example: Let $Z = +2^{-130} * 1.01101...$ and suppose that the destination is a SINGLE precision field. As a further simplification let us assume there are only 6 bits of precision to be carried, plus the implicit leading bit, in SINGLE. Then

$Z = 2^{-130} * 1.01101...$ -130 < -126 so we denormalize --

$Z = 2^{-126} * 0.000101101...$ we round (to nearest,say) --

$X = 2^{-126} * 0.000110$ = the result to be delivered.

We call the above result X a "denormalized number" in SINGLE precision; it is a special type of unnormalized number, namely one with the smallest possible exponent for the given basic level of precision. Note that the exponent of a denormalized number links it to a basic level of precision. We will discuss only SINGLE precision denormalized numbers here. DOUBLE precision is essentially the same; see PART II for details of the differences. In terms of the format

a SINGLE precision nonzero denormalized number X is encoded as

S = sign bit
E = 0
F = nonzero string of bits to the right of the binary point of X.

We reconstruct X via the formula

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

observing that E is not the true biased exponent in SINGLE precision. Comparing this formula with its analogue for normalized numbers we see that, when unpacking a denormalized number, the 1-bit that would have gone to the leading bit of the significand for a normalized number is instead added into the unbiased exponent E-127+1.

The denormalized numbers and signed zeros are the family of 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 significand field.

The denormalized numbers and ± 0 join the normalized numbers and NANs as elements of our model of the real numbers. Both +0 and -0 correspond to the real number 0 and are identical in every operation except division; this will be discussed along with Invalid_Divisions. A denormalized number X represents, roughly speaking, all of the real numbers which would round to that bit-string X in the specified rounding mode and precision. We note that the denormalized numbers are designed not so much to extend the exponent at any level of precision, but rather to allow further computation with some sacrifice of precision in order to defer as long as the possible the need to decide whether the underflow will have significant consequences.

In add/subtracts, denormalized numbers behave in much the same way as normalized numbers, with never more than a rounding error committed in any operation. The situation is different in multiply/divides, where multiplying a SINGLE precision denormalized number by a large power of 2 and attempting to store the result in SINGLE is an Invalid_Operation. The unnormalized significand, having suffered loss of precision during some prior Underflow, may not be promoted to normalized status merely by multiplication. If, however, the destination had been an extended field, the unnormalized significand with large exponent would have been a (perhaps temporarily) legitimate result. PART II gives the full details of denormalized numbers in arithmetic opratations.

The implementation of denormalized numbers, whether in hardware or software, is required only of those systems in which, on Underflow, users may proceed without trapping. Implementations not supporting denormalized numbers, and thus forcing a user trap on every Underflow, must nonetheless sense the denormalized numbers as bit strings, when they are picked up as operands, and generate an Invalid_Operation fault. Note that implementations whose default, or only, option upon Underflow is to underflow abruptly (i.e. from anything smaller than the smallest normalized number) to zero, do not conform to the standard.

OVERFLOW

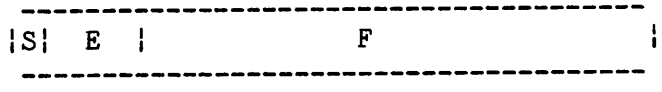
In contrast to the graceful treatment of Underflows in no-trap

situations, Overflows are dealt with swiftly and surely, with a corresponding loss of information. It is noted in the discussion of Underflows that to take the greatest advantage of the treatment of Underflows, the number system is slightly biased away from zero, in the hope of making Overflows more rare than Underflows.

We now discuss the treatment of Overflows. In each case we let X be the normalized result of an arithmetic operation; we assume that X has been rounded to the precision of the destination field and that X has overflowed the exponent range.

If a trap is to be taken then, because every arithmetic operation's result goes to a destination no narrower than its input operands, the exponent can be out of range by at most a factor of two, except for the MOVE operation which is discussed in PART II. The exponent is wrapped around into the desired range with a bias adjust specified in PART II. This result is delivered to the destination field and the trap is invoked. The exponent wrap-around is chosen so that the result, while related in a simple way to the overflowed value, lies somewhere in the middle of the numerical range of representable numbers. This diminishes the risk that a computational response (like scaling) to Overflow will encounter almost immediately a rash of consequent Underflows. The analogous statement holds for the treatment of Underflows when the trap is enabled.

If no trap is to be taken, then infinity with the sign of X is written into the destination field. In SINGLE and DOUBLE precision with format



infinity is encoded as

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

In extended precision E = 0111...11 = the most positive exponent and F = 0.

The signed infinities and NANs thus comprise the family of reserved operands with most positive exponent. As elements of our model of the real numbers, the infinities are given two interpretations. In the Affine Closure,

minus_infinity < {real numbers} < plus_infinity.

But in the Projective Closure the sign of infinity is ignored, i.e.

infinity = minus_infinity = plus_infinity,

and all comparisons between infinity and a real number involving order relations other than = or not= are invalid_operations.

Aside from the compares, all operations on the infinities in the two closures are the same except that both

infinity + infinity and infinity - infinity

are Invalid_Operations in the Projective Closure. Systems supporting the infinities shall provide an Affine/Projective mode bit so that choice of closures can be made under program control. The Affine mode is the default mode, and is appropriate for most engineering calculations involving exponentials or disparate time constants or infinities generated by overflows. The Projective mode is appropriate for real and complex rational arithmetic, for continued fractions, and for infinities generated by division by zeros not generated by underflows.

The infinities interact with ± 0 in a very special way. It was noted in the last section that, aside from a trivial exception noted in PART II, +0 and -0 participate identically in all operations except division. The only way to distinguish +0 and -0 arithmetically is to use the fact that

$$+1/+0 \;=\; \text{plus\_infinity} \;>\; +1/-0 \;=\; \text{minus\_infinity}$$

can be recognized in the Affine mode. In terms of our model of the real numbers, this situation is to be expected. Since we associate the two elements ± 0 with the single real number 0, we should not be able to distinguish machine ± 0 using arithmetic on real numbers; rather, we find that we can disitinguish them only in a proper extension of the real numbers that includes infinities.

As we saw in the case of Underflows, systems forcing traps on Overflow need not support infinities but must recognize them when they are picked up as operands for arithmetic operations, and generate an Invalid_Operation exception.

INVALID_DIVISION

The Invalid_Division exception arises when a zero divisor occurs in a division operation. It also arises when a denormalized divisor is picked up in a system not implementing division by denormalized numbers; see PART II for details.

If the divisor is zero and the dividend is nonzero, the result is infinity with sign according to convention.

If both the divisor and dividend are zero, or if the divisor is (too far) denormalized, then if the invalid_division trap is enabled, it is invoked; if a result must be delivered it is a NAN. If the Invalid_Division trap is disabled then an Invalid_Operation exception arises; if a result must be delivered it is a NAN.

INEXACT_RESULT

The Inexact_Result exception arises when a round-off error is committed in an arithmetic operation. It is intended for essentially integer calculation as in COBOL and to facilitate multiple-precision calculation. The rounded result is delivered to the destination field and the trap is invoked if enabled. When this exception occurs together with Over/Underflow, the latter traps have precedence.

PART II: SPECIFICATIONS FOR STANDARD ARITHMETIC

LEVELS OF PRECISION

SINGLE and DOUBLE are the basic levels of precision. A standard system shall provide either SINGLE only, or both SINGLE and DOUBLE. In addition a system may provide the extended precision corresponding to the widest basic precision supported.

The tables below detail the levels of precision and the data types specified by this standard. Of the reserved operands, the denormalized numbers and infinities need not be implemented in hardware in systems trapping to user software on all Overflow, Underflow and Invalid_Division exceptions, provided these operands cause an Invalid_Operation exception when picked up as operands in an arithmetic operation.

The signed infinities, when implemented, will be interpreted in either the Affine or Projective closures of the real numbers. In the latter case the sign of infinity is ignored by the add, subtract and compare instructions, i.e. "plus" and "minus" infinity are treated as the same, unsigned, infinity. Choice of closures shall be exercised via the Affine/Projective mode bit, which may be sensed and changed by user programs. Affine mode shall be the default for all arithmetic operations. Note that table entries giving specific values for the exponent E of the zeros and reserved operands in EXTENDED precision depend on the number of bits in the exponent field.

BASIC LEVELS OF PRECISION

	SINGLE	DOUBLE
Length in bits	32	64
Fields:		
S = sign	1	1
E = exponent	8	11
F = significand	(1)+23	(1)+52
Storage format:	|S| E | F |	|S| E | F |
Interpretation of sign:		
Positive	0	0
Negative	1	1
Normalized numbers:		
Interp. of E	unsigned integer	unsigned integer
Bias of E	127	1023
Range of E	$1 \leq E \leq 254$	$1 \leq E \leq 2046$
Interp. of F	significant digit field = 1.F	significant digit field = 1.F
Relation to represented real number	$(-1)^S*2^{E-127}*(1.F)$	$(-1)^S*2^{E-1023}*(1.F)$
Signed Zeros:		
E =	0	0
F =	0	0
Reserved Operands:		
Denormalized numbrs:		
E =	0	0
Bias of E	126	1022
Interp. of F	significant digit field = 0.F	significant digit field = 0.F
Range of F	nonzero	nonzero
Relation to represented number	$(-1)^S*2^{-126}*(0.F)$	$(-1)^S*2^{-1022}*(0.F)$
Signed Infinities:		
E =	255	2047
F =	0	0
Not-A-Number, or NAN:		
E =	255	2047
Range of F	nonzero	nonzero
Interp. of F	system-dependent diagnostic and possibly numeric information	
Ranges:		
Max positive normalized	$2^{126}*(2-2^{-23}) = 1.7*10^{38}$	$2^{1022}*(2-2^{-52}) = 9*10^{307}$
Min positive normalized	$2^{-126} = 1.2*10^{-38}$	$2^{-1022} = 2.2*10^{-308}$
Min positive denormalized	$2^{-149} = 1.4*10^{-45}$	$2^{-1074} = 4.9*10^{-324}$

EXTENDED PRECISION

		SINGLE_EXTENDED	DOUBLE_EXTENDED
Length in bits	≥	44	80
Fields:			
S = sign		1	1
E = exponent	≥	11	15
F = significand	≥	32	64
Storage format:		not specified beyond minimum field widths	
Interpretation of sign:			
Positive		0	0
Negative		1	1
Interp. of exponent:		unsigned integer	unsigned integer
Max E	≥	1023	16383
Min E	≤	-1024	-16384
Nonzero numbers:			
Range of E		Min E to (Max E - 1)	Min E to (Max E - 1)
Interp. of F		significant digit field with binary point to the right of the leading bit	
Relation to represented number		$(-1)^S * 2^E * F$	$(-1)^S * 2^E * F$
Signed zeros:		use special indicator or condition bits, or else	
E =		Min E	Min E
F =		0	0
Reserved operands:			
Signed infinities:		use special indicator or condition bits, or else	
E =		Max E	Max E
F =		0	0
Not-A-Number symbols, or NANs:		use special indicator or condition bits, or else	
E =		Max E	Max E
Range of F		nonzero	nonzero
Interp. of F		system-dependent diagnostic and possibly numeric information	
Ranges:			
Max positive normalized	≥	$2^{1022}*(2-2^{-31}) = 9*10^{307}$	$2^{16382}*(2-2^{-63}) = 6*10^{4931}$
Min positive normalized	≤	$2^{-1024} = 5.6*10^{-309}$	$2^{-16384} = 8*10^{-4933}$
Min positive unnormalized	≤	$2^{-1055} = 3*10^{-318}$	$2^{-16447} = 9*10^{-4952}$

ARITHMETIC OPERATIONS

The following arithmetic operations are completely described below:

ADD	REMainder	INTeger_part
SUBtract	CoMPare	Floating_to_INTeger
MULtiply	MOVe	INTeger_to_Floating
DIVide	SQuare_RooT	

BINary_integer_to_decimal_STRing
decimal_STRing_to_binary_INTeger
BINary_floating_to_DECimal_floating
DECimal_floating_to_BINary_floating

An implementation of this standard must at least provide:

1. ADD, SUB, MUL, DIV and REM for any two operands of the same precision, for each supported precision, with the result having no less exponent range than the operands.

2. CMP and MOV for operands at any, perhaps different, supported levels of precision (in the case of MOV the second operand is the destination).

3. INT and SQRT for operands at all supported levels of precision, with the result having no less exponent range than the input operands.

4. Conversions between floating point integers in all supported levels of precision and binary integers in the host processor.

5. Radix conversions, as described in a separate subsection below.

It is assumed that MOV is the only operation whose destination may have a smaller exponent range than its source operand(s). Otherwise Over/Underflow with the corresponding trap enabled entails difficulties which are discussed under "BIAS ADJUST" below.

For simplicity, those arithmetic operations which deliver floating point results rather than strings or binary integers are broken into two steps. In the first step a preliminary result Z is formed and, if numeric, rounded to the required precision. This step is peculiar the the specific operation. In the second step the result Z is delivered to the destination, any exceptions are noted, and any traps invoked. The second step is the same for all operations except REM and MOV; the minor differences are noted.

One or more of five exceptional conditions may arise during an arithmetic operation: Overflow, Underflow, Invalid_Division, Invalid_Operation and Inexact_Result. For each of the exceptions, an implementation of the standard may

a. Force a trap to user software.

or

b. Deliver a specified result and proceed.

or

c. Provide the user with a Trap_Enable bit whereby to choose (a) or (b).

Whenever a choice is given, the default shall be to proceed without a trap. Associated with each of the exceptions is a sticky flag which is set on the

occurrence of the corresponding exception, regardless of the system response. The flags may be sensed and changed by user programs. Following an exception, a flag remains set until cleared by user software.

A system providing a trap on an exceptional condition must give sufficient information to allow correction of the fault. The correct result may be given encoded, as in Over/Underflow with the exponent wrap-around, or in a heap pointed to by a NAN written into the destination. On the other hand, if no numeric result can be given, the opcode and aberrant operands must be provided; in this case if the destination field is the same as one of the source fields then the trap must be taken before any "result" is written over the source operand.

While the specifications of the arithmetic operations indicate that NANs propagate through operations without raising exceptions, a system may raise the Invalid_Operation exception for a system-specified distinguished class of NANs. If the Invalid_Operation trap is enabled it should be invoked at the start of the operation, i.e. before any results are produced; if the trap is disabled a NAN should be generated as in any Invalid_Operation.

In the event that two NANs occur as operands in an arithmetic operation, and neither is designated to cause an Invalid_Operation exception the following precedence rule determines which will be propagated as the result of the operation:

> The sign and exponent are ignored, and the significands are compared as numbers of the form 0.bbbb..., i.e. the leading bit, whether explicit or implicit, is taken to be 0. The NAN which is smaller by this comparison is the result of the operation.

"M": In the tables specifying the arithmetic operations, the entry "M" indicates that the above precedence rule is to be applied to two NANs picked up as operands.

ADD/SUBTRACT

Form a preliminary result Z = X + Y. On a SUBTRACT set Y = -Y and ADD. Z is given by the following table:

		Y			
	X + Y	±0	V	±inf	NAN
X	±0	a	Y	Y	Y
	V	X	b	Y	Y
	±inf	X	X	c	Y
	NAN	X	X	X	M

where V is any nonzero number, possibly unnormalized.

a.

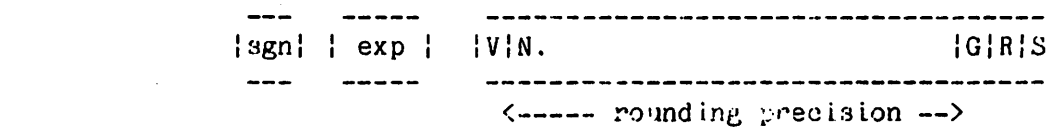
+	-0	+0
-0	-0	A
+0	A	+0

A = +0 in rounding modes RN, RZ, RP and -0 in rounding mode RM.

b.

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 step 4 below).

2) Add the operands, yielding a result which may be viewed as:



where the binary point follows N, and S = sticky bit = logical OR of all bits to the right of R.

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

4) Subtraction of magnitudes: If, after binary point alignment in (1), both operands were unnormalized, then skip to (5). Otherwise...

   i) If all bits of the (unrounded) significand are zero, set the sign to "+" in rounding modes RN, RZ, RP, and set the sign to "-" in mode RM, as in case A of (a) above. The result is true zero, i.e. the exponent is set to its most negative value.

   ii) Otherwise (some significand bit is nonzero)... Normalize the result, i.e. left shift the significand while decrementing the exponent until N=1. S need not participate in the left shifts; either zero or S may be shifted into R from the right.

5) Round, as specified under "ROUNDING".

c. In Affine mode (+infinity) + (+infinity) -> (+infinity) and (-infinity) + (-infinity) -> (-infinity). In Affine mode on (+infinity) + (-infinity) and (-infinity) + (+infinity), and in all cases in the Projective mode, signal Invalid_Operation, and if a result must be delivered set Z to NAN.

MULTIPLY

Form preliminary result Z = X * Y. Z is given by the following table, with sign = exclusive OR of the input signs:

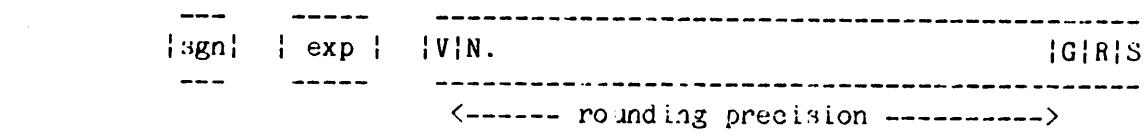
		Y			
	X * Y	±0	V	±inf	NAN
X	±0	g	g	i	Y
	V	g	h	j	Y
	±inf	i	j	j	Y
	NAN	X	X	X	M

where V is any nonzero number, possibly unnormalized. (Perhaps the standard should specify that NAN * 0 = 0.)

g. Z=0 with sign.

h.

1) Generate sign and exponent according to convention. Multiply the significands. The result may be viewed as:



where the binary point follows N, and S = sticky bit = logical OR of all bits to the right of R.

2) If V=1 then right-shift the significand one bit and increment the exponent, and go to (4). Else, when V=0,...

3) If N=0, then left shift the significand one bit and decrement the exponent. S need not participate in the left shift; a zero or S may be shifted into R from the right. (This step is contentious, and may not be included in the standard.)

4) Round, as specified under "ROUNDING".

i. Signal Invalid_Operation. If a result must be delivered, set Z to NAN.

j. Z = infinity with sign according to multiply convention.

DIVIDE

Form a preliminary result Z = X/Y. Z is given by the following table, with sign = exclusive OR of the input signs:

		Y				
	X / Y	±0	unnorm	norm	±inf	NAN
X	±0	m	m*	g	g	Y
	V	k	m*	n	g	Y
	±inf	k	j	j	m	Y
	NAN	X	X	X	X	M

where V is any nonzero number, possibly unnormalized. (Perhaps the standard should specify that NAN / infinity = 0.)

g. Z=0 with sign.

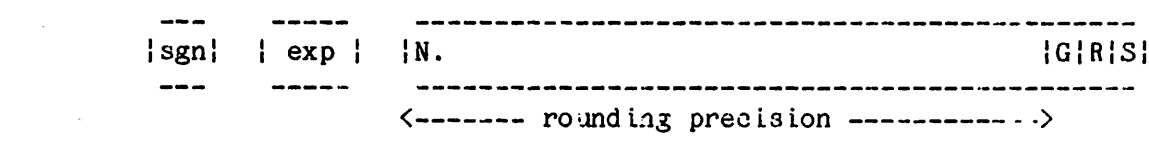
j. Z = infinity with sign.

k. Z = infinity with sign. Signal Invalid_Division.

m. Signal Invalid_Division. If a result must be delivered then set Z to NAN.

n.

1) Generate sign and exponent according to convention. Divide the significands. The result may be viewed as:



2) If N=0 then left shift significand one bit and decrement exponent. S need not participate in the left shift; either a zero or S may be shifted into R from the right. (The standard may allow a second left shift if N=0 after the first.)

3) Round as specified in "ROUNDING".

* These divisions may be implemented provided the result has no or one more significant bit than the operand with the fewer significant bits. The standard may allow divisors whose significant digits have the form 0.1bbb... where the b's are either 0 or 1.

REMAINDER

Form the preliminary result Z = remainder when X is divided by Y, with integer quotient Q. Q does not participate in STEP TWO 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. Z and Q are given by the following table, with the sign of Q given by the exclusive OR of the signs of the input operands.

		Y				
	X REM Y	±0	unnorm	norm	±inf	NAN
X	±0	w1	x	x	x	Y
	V	w2	w	y	x	Y
	±inf	w2	w	w	w	Y
	NAN	X	X	X	X	M

where V is any nonzero number, possibly unnormalized.

w. Signal Invalid_Operation. If results must be delivered then set Z and Q to NAN.

w1. Signal Invalid_Operation. If results must be delivered then set Z to X and Q to NAN.

w2. Signal Invalid_Operation. If results must be delivered then set Z to 0 with the sign of X and set Q to infinity with sign according to divide convention.

x. Q=0 with sign. Z=X. (This is equivalent to (y) when the divisor is 0.)

y. Set Q to the integer part of X/Y, with sign. If Q contains more significant bits than its intended destination, then discard the excessive high order bits and signal Inexact_Result. Set Z to the remainder, with sign

of X. Normalize and round Z as in "ROUNDING".

SQUARE_ROOT

Form a preliminary result $Z = \sqrt{X}$. Z is given by the following table:

X	Z
±0	X
-unnorm	s1
+unnorm	s
-norm	t1
+norm	t
-inf	u
+inf	X
NAN	X

s1. Signal Invalid_Operation. If a result must be delivered, set Z to NAN. The standard may allow $-\sqrt{-X}$ as in s.

s. Compute $Z = \sqrt{X}$ to at least the number of bits required to produce a correctly rounded result*. Then unnormalize Z until it has just one more significant bit than X has. Round, as specified in "ROUNDING". (The standard may classify this as an Invalid_Operation with NAN as the result.)

t1. Signal Invalid_Operation. If a result must be delivered, set Z to NAN. The standard may allow $-\sqrt{-X}$ as in t.

t. Compute $Z = \sqrt{X}$ to the number of bits required to produce a correctly rounded result*. Round as in "ROUNDING".

u. In Projective mode Z=X. In Affine mode signal Invalid_Operation; and if a result must be delivered then set Z to -infinity, if possible, otherwise set Z to NAN.

* To round correctly in all cases, calculate two more bits of X than the precision of the destination, which precision is never less than that of X.

INTEGER_PART

Form a preliminary result Z = integer_part(X). Z is given by the following table:

X	Z
±0	X
V	p
±inf	q
NAN	X

where V is any nonzero number, possibly unnormalized.

q. Signal Invalid_Operation. If a result must be delivered then set Z to X.

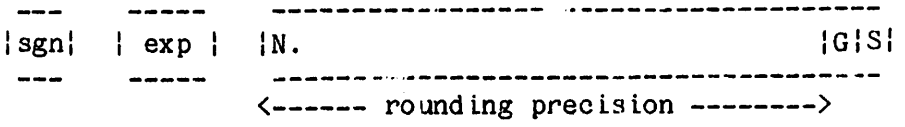
p.

1) If X has no (zero or nonzero) fraction bits in its significand then set Z to X. Otherwise...

2) Right-shift X's significand while incrementing the exponent until no bits (zero or nonzero) of the fractional part of X lie within the rounding precision in effect. The exponent's value will then be:

SINGLE	23
SINGLE_EXTENDED	31
DOUBLE	52
DOUBLE_EXTENDED	63

The result may be viewed as:



where the binary point is to the right of N, and S = sticky bit = logical OR of all bits to the right of G.

3) Round as specified in "ROUNDING".

4) If all of the significand bits of Z are 0 then set Z to zero with the sign of Z. Otherwise normalize Z. S (which was set to zero after rounding in step 2) need not participate in the left-shifts of normalization; zero or S is shifted into G from the right.

MOV

MOV is an operation whose destination may have shorter range and precision than its source operand(s), in which case it performs an arithmetic operation. A preliminary result Z is given by the following table:

X	Z
±0	X
V	r
±inf	X
NAN	X

where V is any nonzero number, possibly unnormalized.

r. Z=X rounded, as specified in "ROUNDING".

STEP TWO of the MOV operation differs from that of the other arithmetic operations in the following way. On Over/Underflow with the corresponding trap enabled, the exponent may be more than a factor of two (i.e. one bit) beyond the exponent range of the destination. In this case the BIAS ADJUST routine is not invoked, rather a NAN is written to the destination field indicating that the correct result is the (unchanged) source operand of the operation.

If the destination field is wider in range and precision than the source

field than the MOV is exact with one exception:

In MOV SINGLE --> DOUBLE, if the SINGLE operand is denormalized then an Invalid_Operation exception arises; deliver a NAN to the destination indicating that the (unchanged) source operand is the correct result.

ROUNDING

Round the preliminary result W of an arithmetic operation to get the rounded result Z. Four rounding modes are described by the standard:

RN -- Round to Nearest
RZ -- Round to Zero
RM -- Round to -infinity
RP -- Round to +infinity.

An implementation of the standard may support either of two combinations of rounding modes:

1. RN only, with RZ for certain specified integer operations.

2. All four rounding modes.

If all four rounding modes are supported then RN shall be the default mode for all arithmetic operations.

Many systems will support more than one level of precision; some as many as three (SINGLE, DOUBLE, DOUBLE_EXTENDED). When a system supports more than one level of precision it must provide users with the option of rounding to a shorter precision results intended for a wider destination. The specification of that option will require at most two bits of information:

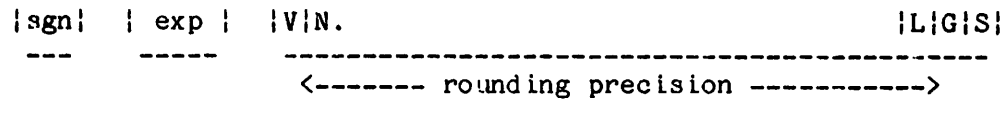
One bit to specify whether to round to EXTENDED or BASIC;

One bit to specify either round to SINGLE or round to DOUBLE, effective only when rounding to BASIC.

If the rounding precision specified is wider than can be held in the intended destination, the latter width will prevail. The standard does not specify how this rounding option will be specified, whether by

preset rounding mode bits
or
rounding mode options in each instruction
or
rounding instructions which can follow the operation whose result is re-rounded
etc.

The number W to be rounded may be viewed as:



where the binary point follows N, and S = sticky bit = logical OR of all bits to the right of G. V=0 at the start of rounding. If the exponent underflows the intended destination and the Underflow trap is disabled, then denormalize W, i.e. shift the significand right while incrementing the exponent until the exponent reaches its most negative allowable value. During each right-shift the G bit is OR-ed into the S bit; the S bit is not shifted.

Determine W1 and W2, numbers representable in the desired rounding precision, as follows:

If G=S=0
then W1 = W2 = W and Z = RN(W) = RZ(W) = RP(W) = RM(W) = W.

Otherwise:
Signal Inexact_Result.
Set t = W with G and S = 0.
Compute T:
Add 1 to the L bit of t's significand.
If V=1 right-shift the significand one bit and increment the exponent.
If sgn=0 then W1 = t and W2 = T; otherwise W1 = T and W2 = t.
Then the rounded values are determined by:
Z = RN(W) = the nearer of W1 and W2 to W;
in case of a tie choose the one of W1 and W2 whose L bit is 0.
RZ(W) = the smaller of W1 and W2 in magnitude.
RM(W) = W1.
RP(W) = W2.

STEP TWO OF ARITHMETIC OPERATIONS

Rounded preliminary result Z was developed in the first step.

1. Special cases involving numeric values of Z:

   a. Test whether Z's exponent over/underflows the intended destination.

   b. If Z is unnormalized...

      i. If the rounding mode is RP or RM then normalize Z as far as possible without allowing Z's exponent to fall below the underflow threshold. Otherwise...

      ii. (In rounding modes RN and RZ...) If the destination is not EXTENDED and Z has not been denormalized by Underflow with the trap disabled, then signal Invalid_Operation and Inexact_Result and if a result must be delivered set Z to NAN.

2. Over/Underflow cases:

   a. On Over/Underflow with the corresponding trap enabled, adjust the exponent bias as specified below.

   b. On Overflow with the trap disabled signal Inexact_Result. Then set Z to infinity with the sign of Z if the rounding mode is:

RN
RZ
RP and Z is positive
RM and Z is negative.

Otherwise set Z to the largest normalized number representable in the destination field, with the sign of Z.

3. Set the exception flags:

   a. Do not signal Over/Underflow signals if the rounding mode is RP or RM and corresponding trap is disabled.

   b. If Invalid_Division (other than by zero) has been signalled but the corresponding trap is disabled, then signal Invalid_Operation.

   c. Set the sticky exception flags corresponding to the exceptions signalled.

4. Deliver Z to its destination. (This may not be required when certain exceptions occur.)

5. Trap if any exception has been signalled for which the corresponding trap is enabled. In the event that more than one signalled exception have their traps enabled, only one trap shall be invoked, according to the following precedence:

   Overflow
   Underflow
   Invalid_Division
   Invalid_Operation
   Inexact_Result.

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 B = 190 in SINGLE, 1534 in DOUBLE, and

$3*2^{n-2}-2$ in (SINGLE or DOUBLE) EXTENDED, where n is the number of bits in the signed exponent.

On Overflow subtract B from Z's exponent; on Underflow add B to Z's exponent.

The foregoing wrap-around scheme works only when the over/underflowed exponent exceeds its destination's range by a factor no larger than two, as is the case when the destination's range is no narrower than the operands' ranges. Such is the case assumed in this document. Otherwise, extreme over/underflows would have to be detected and dealt with in a way similar to what is specified above under "MOV". One way to cope involves a heap onto which is placed that value whose exponent lies beyond the range of its intended destination; into its destination would go a NAN pointing to that value in the heap.

integer of the host processor.

If X is a NAN, infinity, or non-integer then leave the destination unchanged and set the Invalid_Operation bit, trapping if the corresponding trap is enabled. Otherwise...

If X is ±0 then 0 is written into the destination, with the sign of X if the processor supports signed zeros. Otherwise...

Convert nonzero integer X to a binary integer 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.

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.

COMPARES

One of four conditions can result from a floating point compare:

<, =, >, different_and_unordered.

From these conditions there follow:

=	implies	not<,	≤,	≥,	and not>	;
<	implies	not=,	≤,	not>=,	and not>	;
>	implies	not<,	not<=,	≥,	and not=	.

The following table specifies the compare operation:

X \ Y	-inf AFF	n	-0	+0	p	+inf AFF	inf PROJ	NAN
-inf AFF	=	<	<	<	<	<	//// ////	b
n	>	c	<	<	<	<	a	b
-0	>	>	=	=	<	<	a	b
+0	>	>	=	=	<	<	a	b
p	>	>	>	>	c	<	a	b
+inf AFF	>	>	>	>	>	=	//// ////	b
inf PROJ	////	a	a	a	a	////	=	b
NAN	b	b	b	b	b	b	b	b

where n is a negative number and p is a positive number.

a. different_and_unordered.

b. Set the Invalid_Operation exception bit. If X and Y are equal as bit strings then the result is "equal", otherwise the result is "different_and_unordered". Trap if the corresponding trap is enabled.

c. Determine <, =, or > by comparison of X and Y as sign-magnitude bit strings if both have the same level of precision, and by floating point subtraction if either X or Y is EXTENDED or if X and Y are at different levels of precision. The subtraction may not have to be carried out completely. The standard does not yet specify the result of the comparison when the difference is a nonzero number with zero significand, as can be obtained only if either X or Y is EXTENDED and one is unnormalized.

RADIX CONVERSION

A. A system need provide standard conversion to and from only its basic levels of precision. Conversion of EXTENDED numbers, to full precision if desired, is straightforward and intended to be done in auxiliary software if at all.

B. The decimal field widths are:

1. SINGLE: up to 2-digit exponent and up to 9-digit significand.

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

C. Two floating point functions perform conversions between binary floating point integers and signed decimal strings. The latter are character strings consisting of a sign followed by one or more decimal digits. Choice of the character code (BCD, ASCII,...) is left to the implementer.

1. BINSTR converts a binary floating-point integer X to a signed decimal string whose length is at most 9 for SINGLE and 17 for DOUBLE. BINSTR converts zero with its correct sign. In case X is not an integer round X as in "INT". If X is too large to be expressed by a decimal string that fits into the intended destination an Invalid_Operation exception arises and the corresponding trap is invoked if enabled and, if a result must be delivered, the result is a non-decimal string.

2. STRBIN converts a signed decimal string with at most 9 digits in SINGLE, 19 in DOUBLE, to a normalized floating point number X whose value is that of the decimal integer the string represents. If the string contains non-deciaml characters, the standard does not yet specify what happens. If the integer cannot be represented exactly in the intended destination, an Invalid_Operation exception arises and the corresponding trap is invoked if enabled; if a result must be delivered it is a NAN.

D. Conversion over the full range of floating point quantities could be required to be done correctly rounded, but the cost of doing so is probably more than its value. What follows is a compromise designed to ensure that conversion is uniform and in error by appreciably less than one unit

in the last place delivered, at a cost which is nearly minimal. But correctly rounded conversion should also be regarded as conforming to the standard.

E. The function log_base_10 is required and may be computed from the formula

$$\text{log\_base\_10}(X) = \text{log\_base\_2}(X) * \text{log\_base\_10}(2) \quad .$$

It need only be computed to the nearest integer for this calculation.

F. Within the conversion process arithmetic must be done to extended precision. Systems without extended precision must therefore effect extended floating arithmetic using fixed point arithmetic on 32-bit significands (in systems with only SINGLE precision) or 64-bit significands (in systems with DOUBLE precision) while processing signs and exponents separately.

G. Powers of 10 not exactly calculable in the stated precision shall be procured from values stored in tables. Negative powers shall be obtained by dividing by the corresponding positive powers instead of multiplying. The following are suggestions for tables requiring minimal storage.

1. Systems with SINGLE precision only: 10^{13} can be computed exactly using a 32-bit significand. To cover the range up to 10^{38}, a table with the single entry 10^{26} suffices.

2. Systems with DOUBLE or both SINGLE and DOUBLE precisions: 10^{27} can be computed exactly using a 64-bit significand. To cover the range up to 10^{308} a table of 10^{54}, 10^{108}, and 10^{216} suffices.

H. BINary_floating_to_DECimal_floating. Given binary floating point number X and integer k with $1 \le k \le 9$ for SINGLE precision and $1 \le k \le 17$ for DOUBLE precision, we 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.

1. If X is ±infinity or NAN deliver a non-decimal string.

2. If X = ±0 then return I = BINSTR(X) and E = BINSTR(0.). Otherwise...

3. Remember the sign of X. Let Y = absolute_value(X).

4. If Y is normalized compute U = log_base_10(Y), otherwise let U = log_base_10(smallest normalized number).

5. Remember the current rounding mode. Compute V = INT(U)+1-k with mode RZ. Restore the original rounding mode.

6. Compute W = INT(Y / 10^{-V}), drawing powers of 10 from the table if necessary.

7. Adjust W:

   If $W \geq 10^{k} + 1$ then increment V and go to (6).

   If $W = 10^{k}$ then increment V, divide W by 10 (exactly), and go to (8).

   If $W \leq 10^{k-1} - 1$ and Y was normalized in step (3) then decrement V and go to (6).

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

I. DECimal_floation_to_BINary_floating. The decimal floating point number X has the form

$$X = \text{sddddd.DDDDDDD} * 10^{E}.$$

We are given
signed decimal string E
signed decimal string I = sddddddDDDDDDD
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) and W = STRBIN(E).

2. Compute result $X = U * 10^{W-P}$.

OPEN QUESTIONS

1) Should two's-complement representation be allowed, or used instead of sign magnitude?

2) Should the Affine/Projective modes be replaced by three zeros

+0, -0, unsigned 0

and three infinities

+∞, -∞, unsigned ∞

3) Should underflow denormalize to "epsilon" instead of zero; should overflow go to ±HUGE instead of ±∞?

4) Should 0 x NAN = 0 or NAN?

Should NAN/∞ = 0 or NAN?

5) Should denormalized numbers be allowed to be multiplied by numbers bigger than 2, but less than 4? Divided by numbers smaller than ½ but bigger than ¼ ?

6) Should $\sqrt{-5} = -\sqrt{5}$ if not trapped? or NAN?

7) Should division by a denormalized number be allowed if it has only one leading zero?

8) Should division with full quotient and remainder, to expedite multiple-precision division, be required?