✓

Draft of material on a Floating-Point Standard from discussions with H. Stone, W. Kahan, and J. Coonen

Apr. 24, 1978

This paper is a recommendation for a standard floating-point system that can be implemented on a variety of computers. The proposed standard is defined to include a standard storage representation and a functional description of how the floating-point unit carries out computations, what numbers are produced, and which exception conditions are flagged by the unit at the conclusion of an operation.

The purpose of standardization is to benefit many classes of people. Ultimately, standardization should encourage relatively few people to concentrate their talents to create fast, efficient, and accurate floating-point units free from anomalies like those that continue to plague floating-point algorithms even as the computer industry enters its fourth decade. With hardware implementations of the standard considered to be as high-quality as possible, the job of the systems implementers who do the library function routines, conversions, and mathematical interface becomes that much easier. The floating-point interface presented to the applications programmers is then that much more trustworthy so that the applications packages can be developed with a minimum of knowledge of the underlying behavior of the floating-point hardware. Finally, the highest level of users who obtain the computer generated data and make their decisions based on the computational results can do so knowing that their results are as accurate as possible for that particular algorithm for computation.

The floating-point unit and the representation do not contribute any more error than necessary due to finite precision and inherent round-off error. No more accurate answers could be obtained under the given conditions. Anomalies such as uninitialized storage, masked fault indicators, massive intervention of overflow/underflowing and the like cannot go undetected by the standard.

Standardization provides for interchange of encoded floating-point data from machine to machine, and creates a potential market for floating-point modules to attach through standard interfaces to a variety of host machines. Computational results become repeatable from one manufacturers machine to another's, perhaps for the first time in the history of floating point.

✓

This report is organized as follows. Section I is a tutorial description of the standard that communicates the features of the standard and the reasons for including them. Section II describes the storage format for the standard, and Section III treats the machine register format and precision of the arithmetic operations as provided for by the standard. The exact register format is not prescribed by the standard, although minimum field widths are specified for exponent and significand. Results of arithmetic operations described in Section III make use of a guard digit, round bit, and sticky bit so as to obtain the most accurate results possible within the constraints of the register precision. Conversions to and from the binary floating-point format are considered in the next section. SectionlV specifies the faults and modes of the standard, and exhaustively treats all of the combinations of operand pair types, and how they are to be treated by the arithmetic unit and host computer.

Appendices give the detailed specifications for the arithmetic operations and frap handling conditions.

I. A Tutorial Description of the Standard

In this section we give an overview of the provisions of the standard and an intuitive explanation and justification of the features incorporated into the standard.

Representable entities

The standard provides for the representations of:

normalized numbers -- Normalized numbers are real numbers and each floating-point representation of a normalized real number represents a single real number.

denormalized numbers -- Each denormalized number represents an interval of real numbers. Denormalized numbers are produced by arithmetic operations when the results of the operations are two small to represent as a normalized number. The size of the interval associated with a denormalized number is approximately equal to the bound on the rounding error caused by changing the true result of an operation into a denormalized representation.

infinity -- The standard provides for plus and minus infinity which are generated as the result of operations that produce real numbers too large in magnitude to represent as normalized numbers, and as a result of division by 0.

NANs -- A NAN (not a number) is produced as a result of an invalid operation, possibly an operation involving an infinity, denormalized number, or another NAN. The NAN represents no number, but does contain information that tells how and where it was generated.

✓

Entities that are somewhat unusual are the denormalized numbers, infinities, and NANs. The reason for their introduction exposes the underlying philosophy in the design of the floating-point standard. Floating-point operations on normalized representable real numbers must inevitably produce results that are outside the system of representable real numbers. This is inherent because of the finiteness of the set of representable real numbers. When a nonrepresentable result appears, something special must happen. In this standard, one of two things happens--either a trap is generated or the result is a special entity, and the standard provides for program control of whether or not the trap is generated. The standard is specified so carefully in the treatment of special entities that in a wide variety of situations the hardware can deal with the special entities in subsequent calculations and the problems caused by the special results will simply disappear without any intervention or special attention by error handling software. Thus, programmers may be able to avoid special tests for over and underflow in the middle of loops of certain computations, and may be able to incorporate similar efficiencies in a host of computations with the knowledge that should an error condition arise, the calculation will continue with a special entity that may disappear in a subsequent calculation, leaving normalized real answers that are correct to within the rounding error associated with the computation. Thus programs can be written to defer tests for special events as long as possible, and to eliminate the tests completely in some cases.

The standard also provides for extended precision of intermediate results held in registers so that additional protection against over and underflow is obtained when a series of computations can be done in the extended registers before storing the results back in storage in a format with less precision and dynamic range.

The denormalized numbers provided for by the standard represent intervals of numbers rather than individual numbers. Should the result of a computation be a denormalized number, then the true result should be interpreted as lying somewhere within the interval represented by the denormalized number. The intent of introducing these entities is that once produced, they can be combined with normalized numbers and may well produce results that are normalized. Thus a denormalized number may appear only to vanish a short time later with its influence spread during its abbreviated life-time in just the way that a correct representable result should have its influence felt were it possible to represent the true result.

The formats of the entities representable have been chosen so that the entities can be ordered by using a fixed-point signed-magnitude comparision in place of a floating-point comparison. This is much faster, and provides for very simple tests in sorting and searching files of floating-point data, where floating-point arithmetic operations need not be done.

<u>Arithmetic operations</u>

The standard specifies that arithmetic operations are carried
a
out to/precision of roughly half-a- unit in the last place. To achieve this precision, the arithmetic units will require at least a guard digit, round bit, and sticky bit. The guard digit and round bit can be viewed as the significant bits just beyond the significance

of the data held in the register. Should the significand require a left shift to renormalize, then the guard digit and round bit provide the data to be shifted into the register. The sticky bit indicates if there is any nonzero bit off to the right of the round bit in the true representation of the result of an operation. Rounding uses the guard digit, round bit, and sticky bit to determine the nearest representable number above or below the given result.

There are four rounding modes provided by the standard, one mandatory mode which is the default mode, and three additional optional modes that cover, respectively, truncation toward 0, truncation downwards, and truncation upwards. The latter two are included in the standard at this time to provide for future extension of the standard to interval arithmetic. A single mode should suffice for typical operations, but the cost of supplying the additional modes is so small that the gains from including the modes dictate that the optional modes be provided if possible. The modes for interval arithmetic, for example, make possible a quite efficient system for interval arithmetic in which a computation on intervals is a small integer constant times the cost of the same computation performed on simple numbers. Then interval arithmetic is feasible in such a system, whereas today it is far too costly to use. Interval arithmetic provides an almost ideal mechanism for bounding the error of a numerical computation.

The arithmetic system supports two distinct modes for computations involving infinities. The affine mode treats plus and minus infinity as distinct entities, which corresponds to a real number system in which the continuum of numbers is discontinuous at infinity. The projective mode treats the real number system as a system in which plus and minus infinity are the same entity, and the real numbers are continuous at infinity as well as in the range of finite numbers. There are algorithms in real analysis that depend on the affine mode, and similarly there exist algorithms from projective geometry that depend on the projective mode. No single way of dealing with infinities could satisfy both types of algorithms, and both types must run on modern computer systems. Since the standard provides for the representation of infinity, the standard must equally well provide for the proper way of manipulating infinity. Thus it must provide both the affine and projective arithmetic modes. Fortunately, the differences in the two arithmetic modes are quite few, and the complexity required to provide both modes is negligible.

The major philosophy behind the use of infinity as a representable number is that by producing this as a result of an operation, the computer can proceed <u>without</u> a trap, and without requiring special tests for overflow within a tight loop. It is quite conceivable that on a later step the infinity will be divided into a finite real number, producing a zero, so that the infinity will live only a brief time before vanishing. No particular complexity or overhead in the software is required in such an instance. Consequently, the floating-point system definition with infinity can provide an efficient computing environment and can reduce the level of difficulty in creating quality numerical programs.

This philosophy extends into the treatment of underflows as well. Underflows are often treated by setting a result to 0 and continuing. But it is not difficult to construct instances in which this action gives completely erroneous results. The denormalized number scheme replaces the underflow with a result that represents an interval of real numbers. Essentially, the denormalized entity is a symbol that says "the true result is too small to represent to machine precision, but it lies with the interval $a \leq x \leq b$." Should a denormalized number later be added or subtracted from a normalized number, the operation can be done to the precision of the extended registers, and almost surely the result of such an operation is a normalized number. What this says is that no matter which x in the interval $a \leq x \leq b$ represented by denormalized number is used in the operation, the result of the operation rounded to extended register precision is the one delivered. In this way, all of the error attributable to the underflow vanishes.

When you consider the several facilities of the standard it becomes apparent that problems attributable to overflow and underflow may well become negligible. Extended precision in registers keeps such problems from occurring in the first place. But should such conditions arise, then quite often the mechanisms provided by infinity and denormalized numbers are sufficient to deal with the problems without any special programming techniques.

Two arithmetic operations prescribed by the standard include the MOD function and the square root. These are slight variants of the division algorithm, and can be provided rather inexpensively. By insisting on these operations as being primitive operations, the standard forces a guaranteed precision on the operations, and implies that they will be implemented to run at reasonably high speed as well. This makes such primitives quite usable in algorithms for computing elementary functions and other numerical calculations that require fast and precise results.

Traps and faults

Because some operations on real numbers can lead to undefined and invalid results (such as 0 divided by 0), the standard provides for a trap and fault mechanism. Under certain conditions, a trap occurs, with the result being a program interruption to a software trap handler, which then sorts out the problem and takes corrective or diagnostic action. We indicated that the standard is actually designed to avoid traps where possible. Consequently, the traps provided by the standard can be enabled or disabled by program control, and the programmer has the option of invoking the trap mechanism should his results go awry.

Each trap is associated with one or more status bits for that trap. These bits are set when the fault condition occurs, whether or not the corresponding fault is enabled. In this way the programmer can run his program to completion before interrogating the machine to determine if an overflow occurred prior to termination. It is quite possible that all results returned will be correct representable numbers, yet the status bits may indicate that all kinds of terrible things like overflow and underflow occurred en route. This would mark an instance in which the hardware assists dealt with the problems correctly.

The traps include the following: Overflow, underflow, invalid operation, and divide exception. Each can be enabled or disabled. When disabled, each fault condition produces some result, and continues. Overflows generally produce infinities, and underflows generate denormalized numbers. The others produce NANs that indicate the nature and origin of the fault. Each of these traps has at least

✓

one associated status bit that is set when the fault condition occurs, and remains set until cleared by the program. It is quite desirable to have more than one bit per fault condition with the extra bits recording additional information about the fault. Ideally, the status bits contain the program counter value at the point of fault, and possibly the program counter values at the first and last faults recorded. This type of status recording is especially desirable for overflow faults. It is not necessary at all for invalid operation faults because the NAN produced by this fault contains information about how and where it was produced.

There is one additional status bit that is not associated with a fault. This is the <u>inexact</u> bit. If the result of a floating point operation is rounded from the true result, the inexact bit is set to 1, otherwise the bit is reset to 0. A program can therefore test to determine if rounding occurred during an operation. This type of test is extremely useful when performing integer arithmetic in the floating point registers. As long as the integers manipulated contain sufficiently few significant digits to fit within the precision of the significand, the floating-point system is quite useful for this purpose because it automatically takes care of binary point alignment. When a rounding takes place, the number of significant bits have exceeded the precision available. Business applications are quite likely to find the floating-point system attractive for integer arithmetic, but must have the inexact bit to be protected from the loss of significant data.

Conversions

The standard provides for conversions to and from floating-point format, and specifies a method and number of digits to carry that guarantees the accuracy of conversions. With this method it is possible to convert binary floating-point data to a decimal configuration and back again, and recover the identical binary floating-point data. This feature is essential to prevent drift of constants by repeated conversions. It has been observed that some programming systems store files externally in a decimal (or character-oriented)format, and update the files repeatedly. During update, the data on file are converted to binary for processing and reconverted to decimal for output. Some conversion methods lead to a small amount of error in the conversion, which if biased slightly in one direction or the other eventually causes drift in that direction of the constants stored on the file.

Conversions require very few primitive operations. The elementary operations are a conversion from a decimal representation into a binary integer and conversely, and a means for building a floating-point integer from a binary integer. It is also necessary to have some means for finding the integer part of a floating-point number. The conversions depend on the existence of tables that contain accurate representations of powers of 10 to a precision equal to that of the extended registers. A brute force implementation might well store the tables to full precision in computer memory, possibly in a special read-only memory. More sophisticated methods store only parts of the tables, and reconstruct the entries not stored as required at the expense of a small computation.

Future extensions

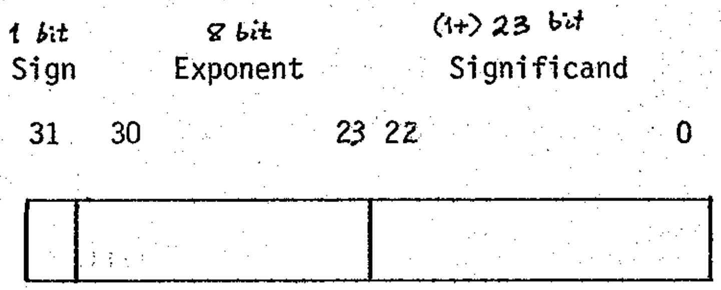
The standard leaves open certain questions that can be resolved at a future time. This standard calls for a binary radix format when one can argue for a decimal radix as well. Recognizing the attractiveness of a decimal readix standard format, if not for now then for some time in the future, we have used the word "digit" rather the word "bit" to describe an entitiy whose magnitude is bounded by the radix. Should there be a decimal standard format, then the word "digit" denots a single decimal digit, and the recommendations here will hold for the decimal standard as well. In the present context the word "digit" denotes one digit in binary radix, and thus denotes one bit.

The results of operations and the inclusion of negative zero are consistent with an extension to interval arithmetic. Optional rounding modes for this standard become mandatory for implementations with interval arithmetic. Given the facilities of this standard it will be possible to implement interval arithmetic on an experimental basis so as to determine its specific requirements. Such experimentation should also establish the benefits of interval arithmetic to provide data on whether or not an extension of the standard to include interval arithmetic is worthwhile.

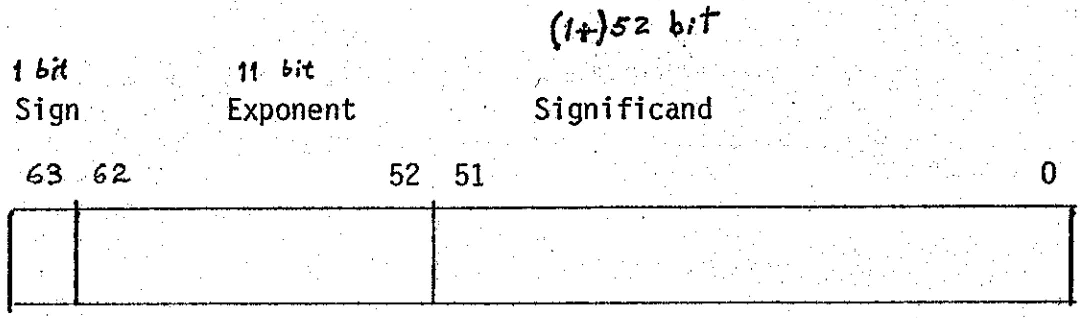
II. Storage Formats

Standard formats are 32 and 64 bits in length as shown in Fig. 1. The 32-bit and 64-bit formats are called short and long, respectively, in the remainder of this report. The intention of standardizing the storage format is to provide a mechanism forinterchanging binary floating-point data from machine-to-machine without requiring conversion, or to enable specialized floating-point processors to manipulate data in the memory of a host machine, independent of the identity of the host.

Each of the formats provides a single bit for sign, a field for a biased exponent, and a field for the significand, which is an encoding of the significant digits of the operand. The formats provide for the encoding of normalized floating-point numbers whose interpretations are real numbers lying within the range of representable numbers. Also encoded in the format are entities that have a special significance and are derived from



Short format



Long format

Fig. 1 Long and short floating point formats.

underflow, overflow, and improper arithmetic operations. The intention of the standard with respect to these special quantities is to provide a mechanism for dealing with anomalous conditions automatically so that in all but the rarest of situations the floating-point processor can treat the unusual conditions so as to obtain the most nearly accurate answer possible. In the very rare instance that cannot be treated by the floating-point processor, an indication of an error condition will be given either through a fault trap or by returning a special nonfloating-point quantity.

In short format, the entities encoded and their interpretation are:

1. Nonzero normalized number: Let p be the value of the sign digit, e the exponent, and s the significand each taken as binary integer encoding. Then the number n encoded by a short floating-point word is:

   $$n = (-1)^p\, 2^{e-127}\,(1 + s\cdot 2^{-23})$$

   This encoding is valid for all exponents e in the range $1 \leq e \leq 254$. Hence $2^{-126} \leq |n| \lneq 2^{128}$.

2. Nonzero denormalized number: When $e = 0$ and $s \neq 0$, the number is assumed to be denormalized and has the value:

   $$n = (-1)^p\, 2^{-126}(s\cdot 2^{-23}) = (-1)^p\, 2^{-149}\, s$$. Hence $2^{-149} \leq |n| < 2^{-126}$

3. Zero: Plus zero is represented by $p = e = s = 0$.

4. Negative zero: There is a negative zero in this representation distinguished by $e = s = 0$, and $p = 1$.

5. Infinity: There are two infinities in the representation, with plus infinity represented as $p = s = 0$, and $e = 11\ldots1_2 = 255_{10}$.

Negative infinity has a negative sign, and is otherwise the same as plus infinity. Thus for negative infinity $p = 1$, $s = 0$, and $e = 11\ldots1_2 = 255_{10}$.

6. Not-a-number (NAN): These quantities are not numbers and are used to indicate improper data or results of improper operations. These quantities have $e = 11\ldots1_2 = 255_{10}$, and $s \neq 0$. The sign bit p may be either plus or minus (0 or 1).

In long format the entities and their representations are:

1. Nonzero normalized number: $n = (-1)^p 2^{e-1023}(1 + s\cdot2^{-52})$ where e is an 11-bit field and s is a 52-bit field. This encoding holds for all bit patterns e except those for which $e = 0$ and $e = 11\ldots1_2 = 2047$. Hence $2^{-1022} \leq |n| \lesssim 2^{1024}$.
2. Nonzero denormalized number: When $e = 0$ and $s \neq 0$, the number represented is $n = (-1)^p\cdot2^{-1022}\cdot(s\cdot2^{-52}) = (-1)^p\cdot2^{-1074}\cdot s$; $2^{-1074} \leq |n| < 2$
3. Zero: Plus zero is represented by $p = e = s = 0$.
4. Negative zero: Minus zero is represented by $p = 1$, $e = s = 0$.
5. Infinity: Plus infinity is represented as $p = s = 0$ and $e = 11\ldots1_2 = 2047_{10}$. Minus infinity is represented as $p = 1$, $s = 0$, and $e = 11\ldots1_2 = 2047_{10}$.
6. NAN: Encodings in which $e = 11\ldots1_2 = 2047_{10}$ and $s \neq 0$ represent entities that are not numbers.

There are a number of comments that explain the reasons for the structure of the format and the choice of representable entities. As the standard is explainedin greater detail later in this report, most of the comments made here will be backed up by the technical discussion.

The formats obviously must encode an exponent, sign, and mantissa. The choice of representation for the mantissa assumes normally that the numbers are normalized, so that the leading digit is nonzero. Since the radix is binary, this digit is known to be a 1, and need not be represented explicitly in storage. Consequently, it is implicit in the representation. (This bit is sometimes called the <u>hidden bit</u>, and this type of representation has been implemented on commercial machines). For denormalized numbers the leading bit is no longer implicit, so one bit of significance (at least) is lost, and the interpretation of the exponent of a denormalized number differs by one from the interpretation for a normalized number.

The storage format and choice of representations guarantees that it is possible to order a collection of floating point numbers by means of fixed-point signed magnitude comparison. In fact, the ordering obtained ranks the entities as follows from smallest to largest:

negative NAN's

minus infinity

negative normalized numbers

negative denormalized numbers

minus 0 } These two bit-strings are not distinguishable

plus 0 } by ordinary arithmetic alone.

positive denormalized numbers

positive normalized numbers

plus infinity

positive NAN's

✓

This is an extremely important characteristic of the representation because signed-magnitude fixed-point comparisons are extremely efficient, requiring simple hardware and very little time in contrast to floating-point comparisons. Consequently, to sort files, to construct histograms, and to perform similar kinds of operations on floating-point data, one need not make use of floating-point arithmetic hardware, and the computations can be done very quickly. Also, NAN's can be detected easily because "$|NAN| > |\infty|$".

The choice of exponent length and significand length are clearly interacting decisions, and at best there is a trade-off between them for the short format. Because of the wide-spread use of computers with a word length equal to a multiple of 8, it is essential to admit a standard format that is a multiple of 8. The shortest practical standard format is 32-bits. Below this length, floating-point operations are still possible, but the precise needs are highly specialized and vary from application to application. No one format can satisfy the special needs because the comprise in choosing an exponent length and significand length that fit in a word shorter than 32 bits inherently must fail to satisfy a large number of the applications that require such short formats.

For the 32-bit length, experience has shown that a 7-bit exponent is simply too small to be acceptable, so the standard requires a format of 8 bits for the exponent. For this length the significand is 23 bits, which is admittedly quite small in itself. However, because of the hidden bit, the format gives essentially the same significance as a 24-bit mantissa, and there have been commercial floating-point formats with this mantissa size that serve a variety of applications.

The long format is clearly the preferred format when adequate storage is available, and when the additional precision improves the accuracy and credibility of the results. Here the exponent length is 11 bits, which gives a very large dynamic range roughly the same as on some widely used commercial computers, and the remainder of the bits are allocated to the significand. The format also satisfies the constraint that the product and quotient of two normalized short numbers cannot overflow the long format.

The exponent bias is slightly different from the bias one might expect given the formats that have been used in the past. Instead of choosing a bias that splits the representable numbers roughly equally between those greater than unity and less than unity, the bias is chosen so that there are about four times as many normalized numbers greater than unity as less than unity. This is done purposely to make overflow less likely than underflow. Overflow, as we shall see later, is more difficult to treat than underflow, and should therefore be allowed to occur less often. Underflow is handled by the standard in a "forgiving" manner so that in all but the rarest of cases underflows that occur can be treated with virtually no more loss of significance in the final results than would otherwise have been caused by roundoff alone. The standard encourages algorithm designers to bias their algorithms to fail by underflow rather than by overflow where the designers can scale their data to create this bias. Since the underflows can be disposed of almost error free in nearly all cases, numerical algorithms can then be made virtually free of problems experienced in present day algorithms because of overflow and underflow.

The inclusion of negative zero is principally for the future

✓

This is an extremely important characteristic of the representation because signed-magnitude fixed-point comparisons are extremely efficient, requiring simple hardware and very little time in contrast to floating-point comparisons. Consequently, to sort files, to construct histograms, and to perform similar kinds of operations on floating-point data, one need not make use of floating-point arithmetic hardware, and the computations can be done very quickly. Also, NAN's can be detected easily because "|NAN| > |∞|".

The choice of exponent length and significand length are clearly interacting decisions, and at best there is a trade-off between them for the short format. Because of the wide-spread use of computers with a word length equal to a multiple of 8, it is essential to admit a standard format that is a multiple of 8. The shortest practical standard format is 32-bits. Below this length, floating-point operations are still possible, but the precise needs are highly specialized and vary from application to application. No one format can satisfy the special needs because the comprise in choosing an exponent length and significand length that fit in a word shorter than 32 bits inherently must fail to satisfy a large number of the applications that require such short formats.

For the 32-bit length, experience has shown that a 7-bit exponent is simply too small to be acceptable, so the standard requires a format of 8 bits for the exponent. For this length the significand is 23 bits, which is admittedly quite small in itself. However, because of the hidden bit, the format gives essentially the same significance as a 24-bit mantissa, and there have been commercial floating-point formats with this mantissa size that serve a variety of applications.

The long format is clearly the preferred format when adequate storage is available, and when the additional precision improves the accuracy and credibility of the results. Here the exponent length is 11 bits, which gives a very large dynamic range roughly the same as on some widely used commercial computers, and the remainder of the bits are allocated to the significand. The format also satisfies the constraint that the product and quotient of two normalized short numbers cannot overflow the long format.

The exponent bias is slightly different from the bias one might expect given the formats that have been used in the past. Instead of choosing a bias that splits the representable numbers roughly equally between those greater than unity and less than unity, the bias is chosen so that there are about four times as many normalized numbers greater than unity as less than unity. This is done purposely to make overflow less likely than underflow. Overflow, as we shall see later, is more difficult to treat than underflow, and should therefore be allowed to occur less often. Underflow is handled by the standard in a "forgiving" manner so that in all but the rarest of cases underflows that occur can be treated with virtually no more loss of significance in the final results than would otherwise have been caused by roundoff alone. The standard encourages algorithm designers to bias their algorithms to fail by underflow rather than by overflow where the designers can scale their data to create this bias. Since the underflows can be disposed of almost error free in nearly all cases, numerical algorithms can then be made virtually free of problems experienced in present day algorithms because of overflow and underflow.

The inclusion of negative zero is principally for the future

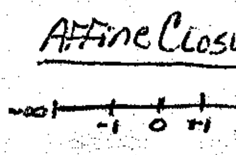
✓

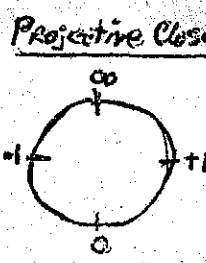
but is also needed to make $+\infty$ and $-\infty$ work as well as they can

inclusion of interval arithmetic operations^. Although this standard does not specify interval arithmetic, interval arithmetic as a mechanism for bounding the precision of a calculation is sufficiently important that we must include the necessary facilities where possible today so that the standard can evolve to incorporate interval arithmetic. The use of a representation to distinguish plus zero from minus zero is one such facility.

The use of an explicit representation of infinity is incorporated to provide a means for dealing with overflows should they occur, and should the programmer prefer an alternative to trapping to a fault handler routine. Operations with infinities are designed so that, in some cases, problems caused by infinities can be disposed automatically. However, there is clearly some loss of information when an overflow is replaced by ^ an infinity, so that computations with infinity cannot give correct results for all algorithms. There is sufficient flexibility in the mechanism for dealing with infinities that the numerical analyst can choose to deal with an infinity through a fault routine or to have the hardware deal with the matter automatically, depending on the sensitivity of the algorithm to the fault handling mechanism.

The inclusion of two infinities, plus and minus infinity, is itself worthy of discussion. In real analysis there are two distinct, and consistent, mathematical systems for the real numbers with infinity. The affine closure of the real numbers has two distinct infinities, plus and minus infinity, and there is a discontinutity between plus and minus infinity. The projective closure of the real numbers treats plus and minus infinity as a single entity, and the real numbers constitute a continuum as you move through the positive numbers to plus infinity, minus infinity, and on to the negative real numbers. An example of a function that is continuous through minus infinity to plus infinity and on around again is the function tan Ø, which obviously arises from projective geometry, and thus is defined on the projective closure of the real numbers.





Since infinities can arise as results of computations, it is essential that the floating-point operations produce the correct representation of infinity. But, unfortunately, with two consistent ways of dealing with infinities, it is not possible to incorporate only a single system into the hardware that deals with both systems. Some algorithms require one system while other algorithms require the other. Fortunately, the differences in the two systems with respect to the behavior of the hardware is so slight that both systems can be incorporated by use of a mode setting to determine which mode is in effect at any given time. The details appear later in this report.

Minus zero is a representable entity, but it must behave exactly as positive zero with respect to arithmetic operations on normalized numbers. In fact, minus zero must be indistinguishable from plus zero in programs with only ordinary arithmetic operations, and only special tests can be used to distinguish the two zeros. The two zeros do behave differently when results fall outside the range of representable numbers since for positive x, x/0 produces an infinity that carries the sign of 0. In projective mode, the two infinities are indistinguishable, but they are distinguishable in affine mode. Except for a few instances like these, the existence of minus zero can largely be ignored until the standard incorporates interval arithmetic.

One of the very important and powerful features of the standard is the specific inclusion of NAN's in the format. The idea of the NAN is to be able to produce and manipulate entities that carry nonnumeric information. Specifically, they are intended to contain codes in the significand field that specify when and/or where they were generated and what caused them to be generated; for instance NAN's can be produced by fault conditions, or can be introduced as unitialized data. If NAN's appear in the output results (possibly because all fault traps have been turned off), the programmer will often be able to discern how and where the NAN's were introduced so that he can eliminate the fault in the program.

Denormalized numbers actually represent intervals rather than individual numbers. For purposes of implementing the standard, a denormalized number with sign p and significand s is treated as if it were the single number $(-1)^p.2^{-149}s$ in short precision and as the number $(-1)^p.2^{-1074}s$ in long precision. Actually, the denormalized number represents the entire interval of numbers in between $(-1)^p\ 2^{-xxx}(s-1/2)$ and $(-1)^p\ 2^{-xxx}\ (s+1/2)$ when $2^{-xxx} = 2^{-149}$ or 2^{-1074}. Note that these intervals can be added and subtracted from much larger individual numbers, with the result being a single representable number to within the permissible rounding error of half a digit in the last place; this is the way denormalized numbers introduced by underflow are intended to disappear, and when they do disappear this way they leave behind numerical results which are scarcely more in error due to underflow than due to unavoidable round-off. When, instead of disappearing, denormalized numbers escape from their zoo and turn up in final results, they serve notice that underflow has caused vital information to be lost. Thus we can usually arrange our calculations to deliver valid inferences about their correctness in the presence of underflow without having to plant "branch on underflow" statements inside inner loops.

Extended formats

The standard specifies the format in storage for floating-point data because of the need to interchange information at this level, and to allow independent floating-point processors to work comfortably with a variety of host processors. When floating-point entities are fetched from storage and placed in registers of the host processor or floating-point processor the format can deviate from the standard because this is not an interface for exchange of information. Indeed, the registers no doubt will insert an explicit leading hidden bit to simplify the floating-point arithmetic.

Although the register format is not fully specified, there is good reason for the registers to have extended precision

✓

so that the results of floating-point operations can be computed to extended precision and over a larger dynamic range than the storage format admits. This minimizes the effects of rounding for computations that are done entirely in registers, since the primary source of rounding error for typical computations is then the single rounding error that occurs when going from extended precision in the register format to the precision of the storage format. Also, overflow and underflow will virtually disappear because of the extended dynamic range of the register format. Consequently the benefit of the extended precision in the register format is that the hardware all but eliminates undesirable floating-point aberrations caused by unexpected round-off, underflow, and overflow.

Table 1 shows the minimum field lengths recommended in the floating point registers. The field length for the significand includes an explicit hidden bit but not the sign of the significand.

Table 1

Recommended Minimum field lengths for register formats

	Exponent	Significand
Extended Short	12 bits	32 bits
Extended Long	16 bits	64 bits

There is no need for extended short if the hardware provides long; there is no point in extended long unless the hardware provides long precision storage formats as well.

The standard is intended to be compatible with the presence of a modest number of registers with extended formats.

Since the precision and range of numbers representable in registers may exceed that of numbers representable in storage, it is possible to compute results in registers that cannot be stored. Any attempt to store them results in a fault condition that is handled either by a trap or by producing a special entity (NAN, infinity, or denormalized number) as described later in this report. Should a fault result in a trap, it is essential that the trap preserve the information in the registers so that the trap handler can make full use of information in the registers when the trap occurs. Obviously, the registers should be accessible to the assembly language programmer, and the format of numbers held in registers should be documented so that the trap handler can rescale or perform a similar sort of computation on the register results to enable a program to proceed. The exact register format and the operations permitted on the registers beyond the use of ordinary floating-point arithmetic are machine dependent and not fixed by the standard.

III. Floating-point arithmetic operations

The standard provides for a guaranteed accuracy of floating-point arithmetic operations, and establishes the rules for rounding results. The algorithm and implementation of the several operations are not specified, and are left to the machine designer, firmware programmer, or systems software programmer to pick the most suitable implementation for each computer system.

All operations require that results be rounded in some fashion whenever the results of the operation might otherwise have more significant bits than can be held in an extended register or storage format.

There are several schemes for rounding possible, and the standard provides for four different schemes. One of these is mandatory, and the other three are optional. If any of the other three are implemented, the mandatory scheme must be the default rounding mode, and the other modes must be selectable under program control.

<u>Mandatory rounding mode</u> (Round to nearest or even number):

1. When rounding a result, the rounding operation must round to the <u>nearest</u> representable number if there is one. (Plus and minus infinity are not considered to be representable numbers when deciding whether to round to infinity or to the largest finite representable number; infinity can be generated only by overflow or by division by zero.)
2. In the event that the result lies midway between two representable numbers, the rounded result is the one with the even significand.
3. In this rounding mode, the rounding error cannot exceed ½ unit in the last digit delivered.

<u>Truncation--Rounding biased toward 0</u>

1. When rounding a result, the rounded result differs from the unrounded result only if the latter is not representable.
2. If the unrounded result is not representable, the rounded result is next representable number closer to 0 (except for over/underflow).
3. The rounding error is always less than 1 unit in the last digit delivered.

<u>Rounding, biased to positive direction</u>:

1. When rounding a result, the rounded result differs from the unrounded result only if the latter is not representable.
2. If the unrounded result is not representable, the rounded result is the next more positive representable number (or $+\infty$).

<u>Rounding, biased to negative direction</u>:

1. When rounding a result, the rounded result differs from the

✓

unrounded result only if the latter is not representable.

2. If the unrounded result is not representable, then the rounded result is the next more negative representable number (or $-\infty$).

The first rule, rounding to the nearest or even representable number, is a rule that creates as accurate a result as any rule, and does not bias the answers in a statistical sense. When this rule is implemented, the rounding errors produced by a sequence of floating-point operations can be bounded *a priori* so that for stable numerical algorithms, it is possible to guarantee the accuracy of the answers returned by the algorithms with a tight error bound. Moreover, this rounding rule preservesas many numerical identities as any other unbiased rounding rule. Consequently, the standard provides for this rule as the mandatory rule if only one rounding mode is implemented, and states that this rule is the default rule if more than one is implemented.

The other three rules are incorporated here because they are useful in certain contexts. For example, the rules that round toward positive numbers and toward negative numbers are both used in interval arithmetic computations. Without these rules, it may be too costly to implement interval arithemtic in a practical sense, so we cite the rules as optional at this time, and expect them to become mandatory options if in the future interval arithmetic is incorporated as part of an extended standard.

All of the rounding rules use the same type of hardware for their implementation; the last three rules use no more information than that which must be provided to affect the first rule.

Before rounding can be effected, the user must have specified the length to which results will be rounded. The lengths available are short (24 significant bits), long (53 significant bits), and/or an extended length.

In the descriptions of the elementary arithmetic operations that follow, we assume that the results produced contain significant data to the right of the significand field, and these data are are shown in Fig. 2. Specifically, there is a guard digit, a round bit, and a so-called "sticky-bit". The guard digit contains the next full digit of significant data (for binary radix operations provided by this standard the guard digit is a single bit; for decimal radix operations of an extension to this standard the guard digit is a full decimal digit). The round bit contains the next bit of information, and the sticky-bit is the logical OR of all the bits in the representation of the result that reside to the right of the round bit. It is usual to conceive of the sticky bit as being reset prior to an arithmetic operation, and it becomes set if any 1 bit is shifted right past the round bit, whence the name "sticky bit."

Arithmetic operations:

Addition and subtraction

1. Compare the exponents of the two addends, and shift the significand with the smaller exponent to the right by by an amount equal to the difference in exponents.
2. Add or subtract the significands.
3. If the unshifted number was normalized before the operation then normalize the results, otherwise do not normalize. Normalization in this case involves left shifts (or the equivalent. A left shift of a single bit-position moves

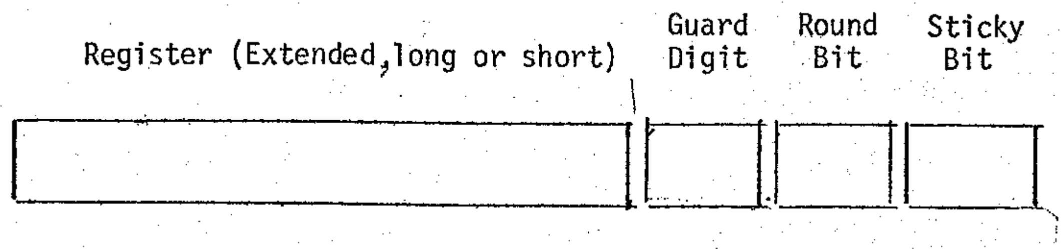


Fig. 2 The guard digit, round bit, and sticky bit.

the guard digit into the extended register, the round bit into guard digit, and moves a 0 into the round bit. The sticky bit is left unchanged by a left shift. Normalize the results after rounding if the rounding produces an unnormalized result.

With addition defined as above, the results of an addition may have to be shifted one digit to the right if the sum of two numbers is too large to be represented by the larger exponent. This right shift and the right shift used to align the significands are the shifts that set the contents of the guard digit and round and sticky bits. The case for which the result is half-way between two representable numbers is identified by a guard digit of 1 (in binary radix or 5 in decimal radix), and a round bit and sticky bit of 0. To round in this case to the nearest even number, inspect the least significant bit of the result. If this is 0 then leave the significand alone. Otherwise add 1.

<u>Multiplication</u>

1. Multiply the significands to obtain a result that occupies at least the full precision of the extended register plus an overflow bit at the left and guard digit, round bit, and sticky bit at the right. The sticky bit is set only if the true product would have non-zero bits to the right of the round bit.
2. Add the exponents of the operands minus a bias to produce a result exponent.
3. The product of the significands could be 2 or larger, almost 4. If the product of the significands is too large to represent (i.e. larger than $1.11111...1111_2$), shift the significand to the right one digit and reduce the exponent by 1.
4. Round the results to the prespecified precision and renormalize if required.

This scheme holds for normalized and denormalized numbers both. The product of two denormalized numbers is surely denormalized (usually 0), and this will be the result produced here. The product of a normalized number and a denormalized number may come out denormalized, but since no left shifts

✓

are provided in this algorithm, the product could be returned as a denormalized number in this instance. No new "significant" figures are introduced as a result of operating with a denormalized number. No particular hardware need be introduced to process denormalized numbers since the algorithm handles both normalized and denormalized numbers in a unified way.

Division

1. Inspect the divisor to determine if the divisor is normalized or denormalized. If denormalized, and the first digit after the point of the significand is 0, terminate the division indicating an invalid operation fault, and producing a NAN if the fault trap is deactivated. If that first digit after the point of the denormalized divisor is not zero, it is optional as to whether the operation should proceed or trap at this point. It is possible to proceed and generate legitimate results if the dividend's significand is not too big.
2. The leading digit produced by dividing the divisor into the dividend can be either 0 or 1. If the digit is a 0, the significand of the quotient is shifted left once upon termination of the operation, and the result's exponent is decremented by 1.
3. The final exponent is the exponent of the dividend minus the exponent of the divisor, possibly with a decrement from Step 2, plus a bias.

If the dividend is denormalized, and the divisor is normalized, the quotient may be denormalized because the algorithm does not provide more than one left shift of the significand to renormalize. Consequently, the algorithm correctly handles both normalized and denormalized numbers, and the only special handling is in the first step where the divisor is tested to determine if it is normalized →

or not. But this test must be built into the division algorithm anyway to determine if the divisor is 0. The only possible difference in handling division by 0 versus division by a denormalized number is the NAN that is produced as a result of the operation. In both cases an appropriate flag must be set.

The standard provides specifically for a floating-point square-root operation because it can be implemented at a cost approximately equal to the cost of a division algorithm. The square root is useful in algorithms for inverse trigonometric and hyperbolic functions, the error function, and other similar functions. With an accurate and fast square root function it is possible to create smooth approximations to the transcendental functions given here and to others at a computational cost approximately the same as or better than the costs for less desirable approximations in use today. Approximations involving the square root are not commonly in use today largely because the square root is not a primitive operation and is too costly to compute in the context of an approximation algorithm. The specifications for the square root are given below:

<u>Square root</u>

1. If the number is negative, trap if a trap is enabled, otherwise set a flag and continue.
2. If the number is plus or minus infinity, and the mode of computation is the projective mode, then return the number itself. If the mode is the affine mode, then the computation must set a fault flag, and trap if a fault trap is enabled, whenever the operand is minus

infinity, and should return plus infinity if the operand is plus infinity.

3. If the operand is plus or minus zero, the square root returns the operand.
4. If the operand is a denormalized number, the operation returns a NAN if the fault trap is inactive, and traps otherwise.
5. Otherwise, return the square root of the magnitude of the operand (accurate to within a half unit in the last place in ◇ mode, et

The specificed accuracy of the square-root is easiest to accomplish by computing the square root by a digit-by-digit algorithm similar to the division algorithm. To be of maximum use as a primitive operation, it is absolutely essential that the square-root be computed to the specified accuracy over the range of numbers.

<u>The MOD function</u>

The standard provides for the computation of x MOD y since this can be done easily at the primitive levels of computation by small changes in the division algorithm. The operation of the function is as follows:

1. Follow the rules for the division algorithm to determine whether or not to initiate the computation.*
2. Carry out the algorithm just far enough to develop all integer digits of the quotient, allowing overflow to occur to the left. Return the quotient and remainder.
3. The sign of the quotient is the sign of x/y. The sign of the remainder is the sign of x. The magnitude of the remainder is always less than y's magnitude.

* If traps are disabled the quotient produces a NAN rather than infinity, and x MOD 0 produces a NAN.

This algorithm for computing x MOD y guarantees that the MOD function is a periodic function of x with period y exactly (no roundoff). This behavior is required to obtain accurate approximations to exponential and trigonometric functions. Note that this definition provides for range reduction in trigonometric functions by permitting the use x MOD pi as a valid operation with pi as accurate an approximation to π as will serve as a divisor.

At this point the discussion of the elementary operations has been completed. Before terminating this section, we explore some of the implications of the specifications given here.

The guard digit, round bit, and sticky bit guarantee that the results delivered are as accurate as possible, and the extended precision of the registers further reduces errors from rounding. Because overflow in the extended registers will be extremely rare (except for invalid operations caused by division by zero), the cost of traps to handle recoverable overflow conditions will be almost negligible. Similarly, underflow will be extremely rare, but should it occur the denormalized number scheme provides an accurate and graceful way of handling denormalized numbers. Addition and subtraction with denormalized numbers is handled in such a way that the loss of significance due to denormalization is given a chance to vanish during subsequent additions. This means that any denormalization , may well vanish after a few further operations without any trap handler or special considerations that greatly increase the complexity and overhead of the operations.

The rounding scheme of rounding to nearest representable number or to the nearest even in case of tie provides an unbiased and accurate rounding scheme. It also prevents "drift" in representations that are produced from other rounding schemes. Consider, for example, a computation that produces $x_1, x_2, \ldots$ from the following formula:

$$x_i = (x_{i-1} + y) - y, \quad y \neq 0$$

where x_0 is a fixed given initial value. Some rounding schemes that have been implemented will cause the value of x_i's to slowly increase away from zero up to a terminal value when the computation is initiated with $x_0 = 0$. With the rounding scheme provided here, no drift is observed past x_1.

In defining the operations, we have indicated that fault conditions must be recognized in specific instances. Because the registers use extended precision, correct computations with normalized numbers will run virtually free of traps when generating results in the registers. The computations may, however, create numbers for which no storage representation exists. It is at this point that the traps are generated by means of a trap on an impossible conversion from register format to storage format. A trap handler can take over at this point and rescale the data as necessary to make representation in storage possible. By deferring the call on the trap handler until the result is actually stored, the overhead associated with overflow and underflow can be diminished substantially and may well disappear from almost all computations.

IV. Binary / decimal conversions

The standard provides for input and output so that the internal storage format can be put into a form interpretable outside the computer. The process of conversion is greatly simplified by defining it in terms of single primitive conversion operation from binary to decimal, and a single primitive conversion in the reverse direction. All other types of co^n^versions are derived from these.

Let the external format for decimal encoding of a floating point number $\overset{x}{/}$be D.DDDDD$\cdot 10^E$, where the leading decimal digit is nonzero for nonzero quantities. Here, for example, the external format uses 6 digits for the significand. The length of the significand is a variable quantity whose bounds are determined from the precision (short or long) of the internal storage format. (The maximum envisaged is 9 digits for short, 17 digits for long.) Proceeding with this illustration, let I be represented by the integer DDDDDD whose digits form the significand of the number, and note that I is an integer between 10^5 and $10^6 - 1$. The value of I can be computed directly from x, and in fact, I is the integer that satisfies

$$I = 10^{5-E} \cdot x$$

since

$$x = I \cdot 10^{E-5}.$$

Conversion from internal format to external format is then simply a matter of computing the value of I and E, given the number of digits to produce in the external format for the mantissa. The specification of the conversion process is then:

1. To produce an external representation with w mantissa digits, compute $E = \lfloor \log_{10} x \rfloor$, using binary floating point arithmetic with the largest available precision.
2. Compute $I = 10^{w-E-1} \cdot x$ rounded to the nearest integer.

3. Check to see if $10^{W-1} \leq I \leq 10^{W} - 1$. If not, adjust E and recalculate I. (This may be required if rounding errors in log and multiply take I out of range.)

4. Convert the integers I and E to strings of decimal digits in external encodings using a binary integer to decimal integer conversion algorithm.

The procedure for floating point conversion requires a table of powers of 10 in floating-point representation, a logarithm algorithm (whose accuracy is not critical), an operation for taking the integer part of floating-point quantity, and an operation for producing the nearest integer to a floating-point quantity.

The table for the powers of 10 if stored in memory must contain sufficient precision to be useful for the extended precision of the registers. Specifically, the powers of 10 table must have/ significands and exponents whose lengths meet the minimum lengths given in Table 1. This implies that the powers of 10 table, if stored in memory, is stored in a format that is different from the standard storage formats, because its precision exceeds the precision of the standard format. It is presumably stored in a format that can easily loaded directly into the floating-point extended registers, or is in a form that permits the extended precision multiplication to be simulated easily. For conversion of external decimal numbers to internal binary numbers, it is most convenient if the mantissas and exponents of the entries for the powers of 10 table be held in separate words.

To produce an internal representation from an external one, given I and E in decimal from the external format, perform the following steps:

1. Convert I from external decimal into an internal binary encoded integer (which we denote as $\underline{i}$).
2. Convert E from external decimal format into the internal binary encoded integer $\underline{e}$.
3. Multiply $\underline{i}$ by the significand of the $\underline{e}$th power of 10.
4. Obtain the significand $\underline{s}$ by normalizing the quantity from Step 3, and determine the number of bit positions shifted in order to normalize.
5. Convert the significand into a floating-point representation derived from the significand and an exponent determined from the number of shifts that occurred during normalization. (The leading digit of the significand will hide in the hidden bit during this operation, unless the number is not representable.)
6. Add the exponent of the $\underline{e}$th power of 10 to the exponent of the representation of x.

This method of conversion is quite suitable for tables of the powers of 10 stored in memory, with exponents separated from mantissas. Essentially it requires the machine to simulate extended precision arithmetic as performed in the registers, but to do so with entities from storage.

It is equally satisfactory to incorporate a primitive operation that coverts a binary encoded integer into its equivalent floating-point representation in an extended register. Once in this format, the floating-point integer can be multiplied by the $\underline{e}$th power of 10, if that power of 10 can be loaded into an extended register with full precision.

The implementation of the powers of 10 table need not store every power of 10 to extended precision. For example, the table might store selected powers of 10 to full precision and compute the remaining ones by multiplying powers of 10 stored in the the table. If, for example, it were convenient to store 10^{10}, 10^{20}, 10^{30}, ..., and $10^1, 10^2, \ldots, 10^9$, then 10^{87} could be computed as $10^{80} \cdot 10^7 = 10^{87}$. (There are faster ways using shorter tables; the standard should specify one of them, for uniformity.)

Among the primitive operations useful to support conversions are operations that access the exponent fields and significand fields of extended registers as separate entities. There should be an instruction that converts a binary fixed-point integer into a floating-point integer, conversion between short and long floating-point precision, and a means of taking the integer part of a floating-point number. The latter should be defined as the operation that truncates downward toward negative infinity. From this operation it is simple to construct the operation that truncates toward 0 when fractions are discarded, but given the latter as a primitive, it is very difficult to construct the former.

To guarantee that each binary floating-point number can be converted into a decimal number and reconverted into the same binary floating-point integer, we require that conversions be capable of producing 9 decimal digits for short precision and 17 decimal digits for long. With this precision and conversion accuracy of half a unit in the last place, numbers will not drift after repreated conversions to and from floating-point.

IV. <u>Treatment of special entities, faults, and traps</u>

This section contains the specification of the special considerations of the floating-point standard including the handling of NAN's, infinities, faults, and traps.

<u>Infinity</u>

There are two infinities, plus infinity and minus infinity, and two different arithmetic modes in which to deal with them, the affine mode and the projective mode. In the affine mode, the two infinities are different entities and are distinguishable. In projective mode, the two infinities are indistinguishable arithmetically despite that they have distinct representations. Arithmetic operations are defined such that when an operation produces an infinity its sign is determined by the conventions required by the affine mode. Since the sign does not matter for the projective mode, there is no problem introduced because of the sign.

All arithmetic involving infinities is the same for both the affine and projective modes except for the addition of infinities with like sign or subtraction of infinities with opposite signs. In the projective mode a NAN is generated while in the affine mode the result is infinity with sign according to convention.

The NAN is introduced because in the projective mode we cannot tell if either operand is positive or negative, so that the result has no meaning.

The results of manipulating infinities in combination with themselves and with other entities are summarized in the tables in this section. In general, the projective closure mode should be the default option for arithmetic, but <u>both</u> the projective and affine modes must be available. Since the only difference concerns addition, subtraction, & compariso of infinite operands, the cost of providing both modes should be small.

As an example of the use of the affine mode consider the expression that occurs frequently in hyperbolic computations:

$$f(x) = \frac{e^x - 1}{e^x + 1} = \tanh\left(\frac{x}{2}\right)$$

When computed in this form, for large x the numerator and denominator both overflow, and the function has no computable value. However, the function is often rewritten in the form

$$f(x) = 1 - \frac{2}{e^x + 1}$$

where now the second term goes to 0 for large x and the function approaches 1. For this function, a programmer takes advantage of the standard by setting the mode to affine mode and disabling the fault for overflow. For large x, the sign of x matters crucially in this computation. Should x overflow and be replaced by infinity, then denominator becomes infinite if x is plus infinity, or becomes 1 if x is minus infinity. Clearly, we have to distinguish between plus and minus infinity in this case, and by so doing and by following the rules for combining infinities, f(x) can be computed correctly (but for round for every x lying in the range of representable numbers <u>without</u> the use of traps.

To give an equally valid example of the use of the projective closure, consider the continued fraction expansion that is frequently encountered in numerical approximation algorithms:

$$f(x) = \cfrac{a}{x + b + \cfrac{c}{x + d + \cfrac{e}{x + f + \ldots}}}$$

In this case, the sign of an infinite intermediate denomoninator makes no difference, for once using it as a divisor, the quotient is 0.

There is an inherent danger in replacing large quantities by a single infinite quantity without trapping. Information must be lost when making this replacement. That loss of information may be extremely damaging when using the infinity later in computations. Consider, for example, the calculation of $(x^2 - y^2)/z$ where x^2 is just a little bit greater than the ~~max~~ overflow threshold, y^2 just a little less than the overflow threshold, and z is a large quantity about the same as the difference $x^2 - y^2$. Then the quotient has a value near unity. But when x^2 overflows and is replaced by plus infinity, subtracting y^2 still gives an answer of infinity, so that upon division by z the result returned is still infinite. There is no way to eliminate this type of problem without rescaling to prevent the loss of information incurred when x^2 is replaced by infinity. Consequently, the standard provides both for the active trap on overflow mechanism to handle unforeseen overflows, and for the disabled trap mechanism to permit computations to proceed with infinite quantities when the programmer knows that computations will give correct results in this instance. Nevertheless, the programmer is urged to scale his problem so that underflows rather than overflows occur, because the standard copes with underflows smoothly through the use of denormalized numbers in which computations can continue with essentially no loss of precision in many cases.

As a final example on the use of infinite quantities, consider the computation

$$x = y$$
$$z = x - y$$
$$t = y - x$$
$$1/z + 1/t = ?$$

If it were not for the first equation, and x differed from y, the last quantity would be a well defined zero. However, with x = y, z and t are both the same zero, and in affine mode the result obtained from the last equation would be plus infinity. In projective mode, the answer obtained is undefined and a NAN should be returned. It is rather interesting that except at x = y, the function defined by the last line is everywhere zero because z and t are negatives of each other. However, at x = y both z and t are zero, so that at this point there is a singularity in the value returned for the function.

<u>NANs</u>

Tables in this section show when to generate NANs and how to propagate them. Because the interpretation of a NAN is implementation dependent, the standard does not provide a specific format for the significand of a NAN. The standard does provide general ground rules for the implementation of NANs.

The purpose of generating a NAN is to be able to determine the point of origin and the reason for origin of the NAN. This information in encoded in the significand. When two NANs are combined by an arithmetic operation, the operation simply propagates the "earlier" of the two, which by convention is the NAN with the algebraically lesser value when the bit-string is interpreted as a sign-magnitude number.

To implement an encoding of error conditions in NANs, an implementation may follow any one of a number of consistent policies. Unitialized storage should be set to NANs whose significands are numerically less than those generated by arithmetic computations. NANs generated arithmetically should contain encoded fields that give the nature of the fault and the program counter at the point of fault. For short precision, the bits available are too few to do this properly, so that an abbreviated pointer to the program counter may be required.

If fault logging can be implemented, the NAN generated by an arithmetic operation can have its fault number inserted in the significand, and information summarizing the fault placed in a fault table at the position specified by the fault number. The implementation of this type of behavior is about equal to the cost of logging faults in error-correcting memory, and is essentially free if the memory-fault logging mechanism is usable for this purpose.

Faults and traps

A full specification of faults and traps appears in the tables in the appendix. The general characteristics of the mechanisms are that the traps can be enabled or disabled, under program control, so that a programmer who knows that the hardware will handle the fault condition properly need not provide a trap handler for the faults. However, the information that faults occurred should set "sticky" flags so the programmer can test afterwards to see if an unusual condition arose, and was corrected by the program. A "sticky" flag is a flag that the program resets prior to a computation, and once set by a fault condition, remains set throughout the computation until it

it is reset by the program. The ideal implementation of a sticky flag has not onl
a sticky bit to indicate the occurrence of the flag, but two registers whose length is equal to the program counter length. The first register is set with the program counter at the point the sticky bit is first set, and the second register is reset to the value of the program counter at each subsequent fault of that type. When a computation terminates, the programmer can find the first and last occurrences of the fault by examining the two registers.

The standard provides for an <u>inexact flag</u> that is a condition bit set after after each floating-point operation in which rounding may occur. The inexact flag is set to 1 if reported result is not equal to the true result, otherwise it is set to 0. If any of the quantities guard digit, round bit, or sticky bit are 1 prior to rounding, the inexact flag is set. The inexact flag allows long-integer calculations to be performed in the floating-point registers without anomalies, like "odd + even = even" caused by integers growing too long, going unnoticed. This will be useful in COBOL programs. Other applications of the inexact flag involve generation of critical test data, and interval arithmetic.

V. Summary

The floating-point standard provides for standard storage formats, minimum bounds on the precision of register formats, a specification of the accuracy of the floating-point operations of addition, subtraction, multiplication, division, square root, and MOD, a specification of the binary/decimal conversion process, and detailed discussions of the treatment of special entities, traps, and faults. The standard provides for evolution to decimal internal representation and for interval arithmetic. No manufacturer yet has implemented the recommendations as described here. Many implementations that fail the accuracy and precision requirements of the standard have through their use shown that such failures are costly and unnecessary. The implementation of the requirements of the standard provide for floating-operations that are done as correctly as possible with a minimum of overhead due to special conditions to the extent that the state-of-the-art can provide today. The cost of a standard floating-point implementation, be it in hardware, firmware, or software, is competitive with the implementations now in common use that have far inferior characteristics.

Appendices

The tables that follow show the results of arithmetic operations, condition code settings, and trap actions for all combinations of operands.

$X + Y$ $\quad$ $X - Y = X + (-Y)$ [If Y is a NAN then follow the procedure for X+Y.]

X \ Y	±0	denormalized	normalized	±∞	NAN
±0	A	Y	Y	Y	Y
denormalized	X	B	E	Y	Y
normalized	X	E	C	Y	Y
±∞	X	X	X	D	Y
NAN	X	X	X	X	min (X,Y)

A -

B - Shift operand with smaller exponent, add and round.

C - Shift operand with smaller exponent, add, normalize and round.

D - In projective mode generate NAN and signal INVALID OPERATION. In affine mode, generate NAN and signal INVALID OPERATION on $-\infty + (+\infty)$ and $+\infty + (-\infty)$. In affine mode $+\infty + (+\infty) \rightarrow +\infty$ while $-\infty + (-\infty) \rightarrow -\infty$.

E - Shift operand with smaller exponent. If unshifted operand (or either operand if neither is shifted) is normalized then add, normalize and round. Otherwise add and round.

$X * Y$

X \ Y	±0	denormalized	normalized	±∞	NAN
±0	F	F	F	N	Y
denormalized	F	G	G	H	Y
normalized	F	G	G	H	Y
±∞	N	H	H	H	Y
NAN	X	X	X	X	min(X,Y)

F – Generate 0 with sign according to convention.

G – Generate sign and exponent according to convention. Multiply significands and if product ≥ 2 then right shift once and increment exponent. Round.

H – Generate ∞ with sign according to convention.

N – Generate NAN. Signal INVALID OPERATION.

X / Y

X \ Y	±0	denormalized	normalized	±∞	NAN
±0	N	F	F	F	Y
denormalized	I	N*	J	F	Y
normalized	I	N*	J	F	Y
±∞	I	H	H	N	Y
NAN	X	X	X	X	min(X,Y)

F - Generate 0 with sign according to convention.

H - Generate ∞ with sign according to convention.

I - Generate ∞ with sign according to convention. Signal DIVIDE EXCEPTION.

J - Generate sign and exponent according to convention. Divide significands and if quotient < 1 then left shift once and decrement exponent. Round.

N - Generate NAN and signal INVALID OPERATION.

* This divide may be implemented if desired.

ROUNDING

The result of an arithmetic computation may lie outside the defined data structure. Rounding is an attempt to fit the result into the data structure before delivery to the destination field.

Let Z be the number to be rounded.
In all of the four rounding modes INEX, the inexact bit, is unchanged if the guard digit, round bit and sticky bit are all 0. Otherwise INEX is set to 1.
All normalization and denormalization occurs before rounding.
The rounding modes fall naturally into two categories, one admitting OVER/UNDERFLOW traps and the other producing some numerical value in each instance (resulting in a closed system).

GROUP 1. If underflow occurred in the computation of Z and if UNDERFLOW trap is disabled then denormalize Z by right shifting while incrementing the exponent until either the exponent is zero (biased) or all nonzero bits of the significand have been shifted into the sticky bit, whichever occurs first. In the latter case, set the exponent to zero.

◇ ROUND TO EVEN

1. (Half-way case) If guard digit=1, round bit= sticky bit=0, and last bit of significand=1 then increment significand. If significand overflows then right shift one bit, increment exponent and test for overflow.
2. (All other cases) Add 1 to guard digit. If significand overflows then right shift one bit, increment exponent and test for overflow.

⧖ ROUND TO ZERO No action (other than determining INEX).

GROUP 2. After rounding and determing INEX, clear any OVER/UNDERFLOW signals for this operation.

$\wedge$ ROUND TO $+\infty$

$Z < 0$	On overflow set Z to most negative number for given destination.
	On underflow set Z to -0.
$Z = 0$	
$Z > 0$	On overflow set Z to $+\infty$.
	On underflow set Z to smallest positive (denormalized) number for given destination.

$\vee$ ROUND TO $-\infty$

$Z < 0$	On overflow set Z to $-\infty$.
	On underflow set Z to largest negative (denormalized) number for given destination.
$Z = 0$	
$Z > 0$	On overflow set Z to largest positive number for given destination.
	On underflow set Z to $+0$.

AN OPERATION PRODUCES A VALUE Z AND EXACTLY ONE OF THE FOLLOWING:			ROUNDING MODES ◇ & ✕ — RESULT DELIVERED TO DESTINATION	CONDITION BITS SET* OFLO	UFLO	INVAL	DIVEX	MODES ∧ & ∨ — RESULT DELIVERED TO DESTINATION	CONDITION BITS SET* OFLO	UFLO	INVAL	DIVEX
NO EXCEPTION			Z					Z				
EXPONENT OVERFLOW	TRAP	NORM	Z with overflow bias adjust †	✓				TRAP: DISABLED				
		DENORM	Z	✓								
	CONTINUE	NORM	∞ with sign of Z	✓				CONTINUE NORM: DISABLED				
		DENORM	Generate NAN. Go to line "INVALID OPERATION"	✓		✓		CONTINUE DENORM: DISABLED				
EXPONENT UNDERFLOW	TRAP	NORM	Z with underflow bias adjust †		✓			TRAP: DISABLED				
		DENORM	Z		✓							
	CONT		Z		✓			CONT: DISABLED				
INVALID OPERATION	TRAP / CONT		Z (a NAN)			✓		TRAP / CONT: Z (a NAN)			✓	
DIVIDE EXCEPTION	TRAP / CONT		Z (±∞)				✓	TRAP / CONT: Z (±∞)				✓

* INEX, the inexact bit, is determined while rounding. If an operation requires no rounding the bit is unchanged.

† Bias adjust — OVERFLOW LONG: −1534 SHORT: −190 EXT. REGISTER: } determined by length of exponent field.
UNDERFLOW LONG: +1534 SHORT: +190 EXT. REGISTER: }