Various features of the proposed standard provide an especially convenient environment for programming numerical procedures such as the familiar elementary functions.

Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic

David Hough
Apple Computer, Inc.

The Floating-Point Working Group, IEEE Task 754, of the IEEE Computer Society's Microprocessor Standards Committee considered several proposals for standard binary floating-point arithmetic for microprocessors. One proposal—called the KCS—was originally developed in 1978 by W. Kahan, J. Coonen, and H. S. Stone. It evolved as an integration of ideas from various computer arithmetic systems, some dating from the 1950's.

Various features of this proposal provide an especially convenient environment for programming numerical procedures such as the familiar elementary functions. In the discussion that follows, we will explain some of these advantages.

The proposed standard (published in this issue) is based upon the KCS proposal, which has been described elsewhere, and on a guide to implementation and a list of applications which have been published.[1-3] The proposal describes binary floating-point formats for single precision, in 32 bits, with 24 significant bits and 8 exponent bits, and for double precision, in 64 bits, with 53 significant bits and 11 exponent bits. In addition to normalized numbers, $+0$, and -0, there are representations for denormalized numbers that can be created by underflow, $+\infty$, $-\infty$, and Not-a-Numbers, or NaNs, that represent various kinds of invalidities.

Extended formats are also defined. *Single extended* has at least 32 significant bits and at least 11 exponent bits. *Double extended* has at least 64 significant bits and 15 exponent bits. Conforming implementations typically provide one of the following four combinations of precisions:

- single,
- single and single extended,
- single and double, or
- single, double, and double extended.

Infinite operands may be handled in affine instead of the default projective mode, according to a switch that may be set by the programmer. Figure 1 illustrates the difference between the projective and affine modes. While the projective mode can be represented as a circle closed at ∞, the affine mode is represented by a line extending through the positive numbers to $+\infty$ on one side and through the negative numbers to $-\infty$ on the other.

Implementations must provide unbiased rounding by default as well as directed roundings.

Computing the elementary functions

Extended format is the most important aspect of the proposed standard for computing elementary functions. A conforming implementation that does not support an extended format would find it unduly costly to provide accurate elementary functions for the widest basic format in that implementation. So, assuming that extended format is available, how can it be exploited?

The idea behind extended format is that it provides a few more bits of significance and exponent range over the basic format that it extends. In execution time, however, single extended will usually be almost as fast as single and much faster than double, since 32-bit registers are likely to be available to accommodate integers.

With the extra precision and range, it is often possible to implement the first algorithm that comes to mind

A preliminary version of this article was prepared for and presented at Electro 80 in Boston in May 1980.

without worrying about problems from roundoff or intermediate overflow or underflow. A single precision function should take a single precision argument and return a result good to single precision, and should not overflow or underflow unless the correct result would overflow or underflow. For instance, when evaluating elementary functions by rational approximation, computation of the approximation in extended, followed by rounding the result to the desired precision, will easily produce as accurate a result as any virtuoso algorithm in the basic precision. If the extended rational approximation is comparable in precision to the extended format, then the basic precision result will probably be better than any basic precision algorithm, even a virtuoso's.

Trigonometric range reduction

Consider now the process of reducing arguments of trigonometric functions. The easiest way to get a satisfactory argument reduction is to use an extended precision value of 2π and an extended precision REM operation. Existing routines, such as those described by Cody,[4,5] use double precision to support single or employ various coding tricks to simulate the effect of extended precision. When angles x are measured in degrees, the process is simple. The remainder operation REM may be used to compute a reduced angle

$$r = x \text{ REM } 360$$

without any error. r will be between -180 and 180 degrees. The only error in $\sin(r)$ will be that due to approximating sin over a restricted range.

For large arguments in radians, however, the principal source of error is often argument reduction with approximate values of 2π. For instance, suppose we wish to compute $\sin(x)$ for $x = n \cdot 2\pi + r$. Assume the only error in the computed result $\text{SIN}(x)$ is due to argument reduction with a contaminated value $2\pi\,(1+\varepsilon)$. Then,

$$\sin(x) = \sin(r)$$

but

$$\begin{aligned}\text{SIN}(x) &= \sin(r - 2\pi n\varepsilon)\\ &= \sin(x)\cos(2\pi\, n\varepsilon)\\ &\quad - \cos(x)\sin(2\pi\, n\varepsilon)\end{aligned}$$

Assume that $2\pi\, n\varepsilon$ is small enough that $\cos(2\pi\, n\varepsilon) \doteq 1$ and $\sin(2\pi\, n\varepsilon) \doteq 2\pi\, n\varepsilon$; then,

$$\sin(x) - \text{SIN}(x) \doteq -2\pi\, n\varepsilon \cos(x)$$

The relative error

$$\left|\frac{\sin(x) - \text{SIN}(x)}{\sin(x)}\right| \doteq 2\pi\, n\varepsilon \cot(x)$$

becomes serious when comparable to the precision desired for $\sin(x)$, say 2^{-s}. Thus,

$$|\tan(x)| < 2^{s}\,|2\pi\, n\varepsilon|$$

defines the set of x for which $\text{SIN}(x)$ can be contaminated by a relative error of 2^{-s} or more, just due to argument reduction. If the approximation of 2π is to the precision 2^{-s}, then $|\varepsilon| < 2^{-s}$ and

$$|\tan(x)| < 2\pi\,|n|$$

defines a very large set of x indeed. But most careful trigonometric argument reduction routines use a value of 2π of greater precision than that of the desired result.

Let 2^{-p} be that greater precision, so that $|\varepsilon| < 2^{-p}$ and

$$|\tan(x)| < \frac{2\pi\,|n|}{2^{p-s}}$$

describes the contaminated intervals. A way to visualize the effect is to compute the percentage of the argument interval $[(n-\frac{1}{2})\,2\pi\;,\;(n+\frac{1}{2})\,2\pi]$ that would result in contaminated values of $\text{SIN}(x)$:

	$n=1$	$n=10$
$p=s$	90%	99%
$p=s+8$	1.6%	15%
$p=s+11$	0.2%	2%

The table displays that percentage for certain values of n and p. In the proposed standard, single extended supporting single corresponds to $p \geqslant s+8$, double extended supporting double corresponds to $p \geqslant s+11$, and double supporting single corresponds to $p = s+29$.

Exponential function x^y

The exponential function x^y provides another example of how extended precision can make the obvious algorithm work well. On a binary machine, x^y is typically computed, for $x > 0$, as

$$x^y = 2^{(y \log_2(x))}$$

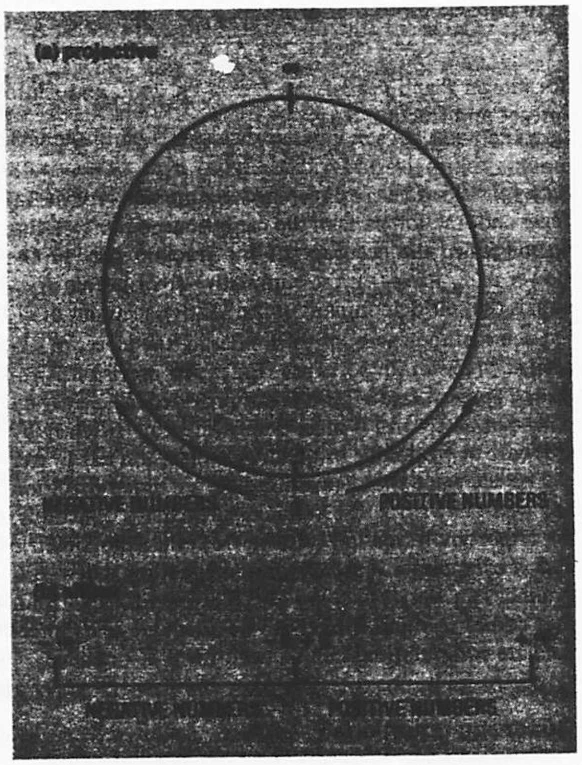


Figure 1. Above, (a) illustrates the projective mode and (b) represents the affine mode.

Assume the only error of importance is the error ε in the computed value LOG2(x):

$$\text{LOG2}(x) = (\log_2(x))\,(1+\varepsilon)$$

Then, the computed value

$$X^Y = 2^{(y\,\text{LOG2}(x))} = x^y \cdot (x^y)^\varepsilon$$

If the desired precision for the result X^Y is 2^{-s} then we consider the computed value to be satisfactory if

$$|(x^y)^\varepsilon - 1| < 2^{-s}$$

and for sufficiently small ε, the inequality can be replaced by

$$|ln(x^y)|\,|\varepsilon| < 2^{-s}$$

If LOG2(x) was computed so $|\varepsilon| < 2^{-p}$ then x^y will be satisfactory if

$$|ln(x^y)| < 2^{p-s}$$

The following table indicates satisfactory ranges for x^y.

$p=s$	$e^{-1} \ldots e^{+1}$	$=$	$.37\ldots2.7$
$p=s+8$	$e^{-256}\ldots e^{+256}$	$=$	$10^{-111}\ldots 10^{+111}$
$p=s+11$	$e^{-2048}\ldots e^{+2048}$	$=$	$10^{-888}\ldots 10^{+888}$

Even 3^3 is likely to come out wrong if $p=s$.

One might be discouraged that, even with extended precision, many values of x^y would be erroneous. However, for single precision, positive normalized numbers lie in the general range 2^{-126} to 2^{+128} so 8 extra bits of precision in single extended are sufficient to calculate in-range x^y satisfactorily to single precision; likewise, 11 bits suffice for double extended to calculate x^y satisfactorily to double precision.

If extended hardware is not available, there are conventional ways of improving the accuracy of x^y. One way is to unpack x into the form

$$x = 2^k \cdot f$$

for integer k and f satisfying $2^{-\frac{1}{2}} < f < 2^{+\frac{1}{2}}$, and y into integer and fraction parts

$$y = \text{int}(y) + g$$

with $|g| \leqslant \frac{1}{2}$. Then, by keeping separate sums for the integer and fraction parts of the various terms, it is possible to maximize the accuracy of the fraction part of the power of 2, $y \log_2(x)$; the cumulative error in that fraction is the sole determinant of the relative accuracy of x^y.

Such methods amount to simulating the effect of extended precision in software. Extended precision is *required* for accurate x^y; if it is not provided efficiently by extended hardware, it must be provided inefficiently by software or by double or quadruple precision hardware.

The software tricks that simulate extended precision are well-known for the familiar elementary functions. An important lesson to be drawn from the trigonometric and x^y examples is that a programmer coding a new, unfamiliar function is likely to find that the first algorithm that comes to mind is likely to be adequate if its intermediate computations can be performed in extended.

Quadratic equations

Nearly equal roots. Lest the unwary be deceived by the previous examples, consider as a counterexample the problem of finding the roots of the quadratic equation

$$ax^2 + bx + c = 0$$

Everyone knows the formula

$$x_\pm = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

However, not everyone has learned that a simpleminded application of this formula will not always yield roots good to single precision if only single precision arithmetic is used. Will extended save the day?

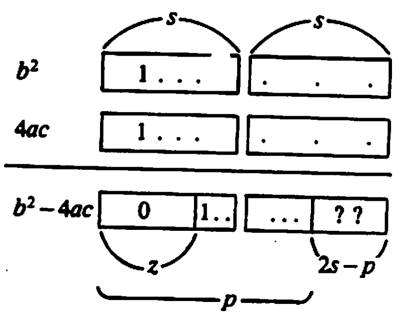
Extended exponent range certainly solves the problem of intermediate underflows and overflows. But extended precision is not enough to save the algorithm implied by the formula above.

To see why, consider the case when

$$|b^2 - 4ac| << |b^2|,\ b > 0$$

In this case there are two nearly equal roots. Suppose the only rounding errors that are made are those in computing b^2 and ac.

If single precision is s bits, the b^2 and $4ac$ terms will each have up to $2s$ significant bits. Because $b^2 \doteq 4ac$, let z be the number of leading bits that cancel. Picture the registers, each s bits in length, as follows:



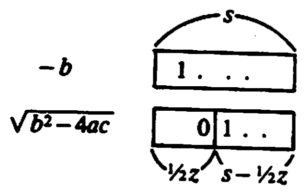
If the computed values of b^2 and $4ac$ are rounded to p bits, then after the subtraction the computed value of b^2-4ac will have $p-z$ significant bits correct, if $p > z$, and none otherwise. Now

$$|b^2-4ac| \doteq 2^{-z}\,|b^2|$$

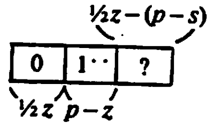
so

$$|\sqrt{b^2-4ac}| \doteq 2^{(-z/2)}\,|b|$$

Thus, when $\sqrt{b^2-4ac}$ is added to $-b$, the operands will align:



If $p-z \geqslant 0$ and $p-z \geqslant s - \frac{1}{2}z$, then $-b \pm \sqrt{b^2-4ac}$ will be correct to about a unit in single precision. But if $p-z \geqslant 0$ and $p-s < \frac{1}{2}z$, then the last $\frac{1}{2}z-(p-s)$ bits of $\sqrt{b^2-4ac}$ will be wrong:



and the corresponding bits of $-b \pm \sqrt{b^2-4ac}$ may be incorrect.

If $p < z$, then the computed value of b^2-4ac will be 0. The uncertain bits in $-b \pm \sqrt{b^2-4ac}$ will be the last $s-½z$.

Combining cases, we find that the number of uncertain bits is the minimum of $(s-p+½z)$ and $(s-½z)$. Under the constraints that $s \leqslant p \leqslant 2s-1$ and $1 \leqslant z \leqslant 2s-1$, we find that the number of uncertain bits in the result is maximized when $p=z$, and that maximum number of uncertain bits in $-b \pm \sqrt{b^2-4ac}$ is $s-½p$ bits. (Note that the case $p=2s$ is not interesting because then b^2, $4ac$, and their computed difference are always exact. Likewise, if $z=2s$, then $b^2=4ac$ exactly, so the rounded values will be the same in any precision and the computed difference will be zero.)

Thus, for single precision of 24 bits and extended precision of 32 bits, it is possible to have as many as 8 incorrect bits in the single precision result $-b \pm \sqrt{b^2-4ac}$. The proof-by-picture arguments above may be formalized to gain rigor at the cost of clarity, but the essential result remains: to compute $-b \pm \sqrt{b^2-4ac}$ correct to single precision, b^2-4ac must be computed to double precision when the roots are nearly equal.

A numerical example may lend credence to the preceding analysis. Consider arithmetic with 5 decimal digits for single precision and 7 decimal digits for single extended. Let $b=70254$, $a=35122$, and $c=35132$. Then,

$$\begin{aligned} b^2 &= 49356\ 24516 \\ ac &= 12339\ 06104 \\ 4ac &= 49356\ 24416 \\ b^2-4ac &= 00000\ 00100 \\ z &= 7 \\ \sqrt{b^2-4ac} &= 00000\ 00010 \end{aligned}$$

resulting in

$$x = 70244;\ 70264.$$

If rounding to extended occurs, however,

$$\begin{aligned} b^2 &= 49356\ 25000 \\ ac &= 12339\ 06000 \\ 4ac &= 49356\ 24000 \\ b^2-4ac &= 00000\ 01000 \\ \sqrt{b^2-4ac} &= \quad 00\ 00031.623 \end{aligned}$$

with the result

$$x = 70222;\ 70286.$$

The computed results are in error by 22 units in the last place of single precision, in accordance with the informal estimate derived previously which predicts about 1½ digits to be uncertain.

Thus, extended precision helps but double precision is really required to make this algorithm work. A more complicated algorithm must be used[3] if extended precision is available and double is not.

Different magnitudes. There are some algorithms that cannot be saved at any reasonable price by extending precision. An example is the obvious quadratic algorithm again, but this time for the case $|b^2| >> |4ac|$, $b > 0$. Here one root is large and may be computed accurately using only single precision. The other root is computed from

$$x = \frac{-b+\sqrt{b^2-4ac}}{2a}$$

The only rounding error that matters is the one due to the subtraction b^2-4ac. Writing this computed value as $(b^2-4ac)(1+\varepsilon)$, we compute a value X

$$X \doteq \frac{-b+\sqrt{b^2-4ac}\,(1+½\varepsilon)}{2a}$$

and find that

$$\frac{X-x}{x} \doteq ½\varepsilon\,\frac{\sqrt{b^2-4ac}\,(-b-\sqrt{b^2-4ac}\,)}{4ac}$$

and

$$\left|\frac{X-x}{x}\right| \doteq \varepsilon\left|\frac{b^2}{4ac}\right|$$

To get X accurate to single precision requires the error less than 2^{-s} so

$$|\varepsilon| < 2^{-s}\,\frac{4ac}{b^2}$$

but if, for instance, a is near the underflow threshold, c is about 1, and b is near the overflow threshold, then the small root x is near, but above, the underflow threshold. In single precision with $s=24$ and the underflow threshold about 2^{-126}, we find

$$|\varepsilon| < 2^{-404}$$

which implies about 404 bits of precision are required to compute b^2-4ac by the algorithm suggested by the well-known formula.

A better solution[3,6,7] is to compute the smaller root x_S from the alternate formula

$$x_s = 2c/(-b-\sqrt{b^2-4ac}\,)$$

The moral here is that extended, or even double, precision cannot save an incurably bad algorithm. Some research has been devoted to finding automatic ways of detecting such algorithms.[8]

Signed zeros and infinities

Signed infinities may seem more natural than signed zeros, but the two are closely related. In the proposal, zeros and infinities may represent underflowed or overflowed quantities. When computing in the affine mode, the rules of the arithmetic preserve many of the relationships that would hold among underflowed or overflowed quantities. By selecting affine mode, the user accepts responsibility for determining that such arithmetic is indeed valid for his algorithm and data.

Thus the expression

$$\frac{1}{\dfrac{1}{x}+\dfrac{1}{y}}$$

yields an invalid result if $x=+0$ and $y=+0$ in projective mode. In affine mode, however, the first expression evaluates to +0:

$$\frac{+1}{\frac{1}{+0}+\frac{1}{+0}} = \frac{+1}{(+\infty)+(+\infty)} = \frac{+1}{+\infty} = +0$$

which is correct if the +0 values of x and y may be thought of as representing positive underflowed quantities, or if they may be thought of as representing limiting values of zero to be attained from the positive direction, since

$$\lim_{\substack{x \to 0^+ \\ y \to 0^+}} \left(\frac{1}{\frac{1}{x}+\frac{1}{y}} \right) = 0$$

in the customary mathematical sense of one-sided limits. But

$$\frac{1}{(+0)} + \frac{1}{(-0)} = (+\infty) + (-\infty) = \text{invalid}$$

even in affine mode, in accordance with the fact that

$$\lim_{\substack{x \to 0^+ \\ y \to 0^-}} \left(\frac{1}{\frac{1}{x}+\frac{1}{y}} \right)$$

does not exist.

Another application of the signs of zero and infinity is in interval arithmetic to denote open and closed intervals. A closed interval includes its endpoints but an open interval does not. Thus,

Machine representation:	Mathematical interval:
$[-\infty,-0]$	$(-\infty,0)$
$[-\infty,+0]$	$(-\infty,0]$
$[-\infty,+\infty]$	$(-\infty,+\infty)$
$[-0,+0]$	$[0,0]$
$[-0,+\infty]$	$[0,+\infty)$
$[+0,+\infty]$	$(0,+\infty)$

The parentheses indicate an open end and the square brackets indicate a closed end.

Now consider the evaluation of the expression

$$\frac{1}{1+\frac{x^2}{y^2}}$$

in interval arithmetic. Suppose we let x represent the set of numbers from μ to 1 with the notation $x=[\mu,1]$, and let $y=[\mu,2]$, where μ is a positive quantity such that μ^2 underflows to zero. Then $x^2=(0,1]$ and $y^2=(0,4]$, so $x^2/y^2=(0,+\infty)$, and $1+(x^2/y^2)=(1,+\infty)$, and finally

$$\frac{1}{1+\frac{x^2}{y^2}} = (0,1)$$

The rules of the proposed standard for signed zeros and infinities will mimic this computation, producing a good result: $[+0,1]$ which is interpreted as $(0,1]$. In a system without open intervals about zero, however, x^2 will be approximated as $[0,1]$ and y^2 as $[0,4]$, so x^2/y^2 will degenerate into the entire real line as it must represent 0/0. Then, the final result will also be the entire line. This example and a full implementation are discussed in detail by Rabinowitz.[9]

The examples given above are intended to demonstrate the applicability of the extended precision and zero/infinity features of the proposed standard. More examples have been given by Kahan and Palmer.[3]

The reader may well conclude that the proposed standard specifies features that are indeed useful in practical programs. Fortunately, other studies have indicated that the implementation cost of the proposal is typically not much greater than the cost of less capable traditional systems of floating-point arithmetic. Some of these studies have borne fruit in the form of the Intel 8087.[10] ■

References

1. J. Coonen, W. Kahan, J. Palmer, T. Pittman, and D. Stevenson, "A Proposed Standard for Floating Point Arithmetic," ACM *SIGNUM Newsletter*, Oct. 1979, pp. 4-12.
2. J. Coonen, "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic," *Computer*, Vol. 13, No. 1, Jan. 1980, pp. 68-79.
3. W. Kahan and J. Palmer, "On a Proposed Floating Point Standard," ACM *SIGNUM Newsletter*, Oct. 1979, pp. 13-21.
4. W. Cody, "Software for the Elementary Functions," in *Mathematical Software*, John R. Rice, ed., Academic Press, New York, 1971, pp. 171-185.
5. W. Cody and W. Waite, *Software Manual for the Elementary Functions*, Prentice-Hall, Englewood Cliffs, N.J., 1980.
6. G. Forsythe, "What is a Satisfactory Quadratic Equation Solver?" in *Constructive Aspects of the Fundamental Theorem of Algebra*, B. Dejon and P. Henrici, eds., John Wiley & Sons, New York, 1969, pp. 53-61.
7. W. Kahan, *Implementation of Algorithms*, Computer Science Technical Report No. 20, University of California, Berkeley, Calif., 1973.
8. W. Miller and C. Wrathall, *Software for Roundoff Analysis of Matrix Algorithms*, Academic Press, New York, 1980.
9. H. Rabinowitz, "Implementation of a More Complete Interval Arithmetic," Master's Thesis, University of California, Berkeley, 1979.
10. Intel Corp., *The 8086 Family User's Manual, Numerics Supplement*, 1980.

David Hough is with Apple Computer, Inc., in Cupertino, California. Previously, he was with Tektronix. Hough participates in the IEEE P754 Floating-Point Working Group. He is a graduate of Carleton College and the University of California.