Advertising Tips for Computer Number System Designers

Jean-Michel Muller
CNRS, Laboratoire LIP, Université de Lyon
France

Work in Progress – Feb. 2021

Abstract

You want to show that your favorite computer number system is far better than anything humankind has designed until now. We survey some tips that can help you to support your (of course, rightful!) claim.

Introduction

Representing real numbers in computers requires finding an adequate compromise between many goals that are frequently antagonistic: accuracy, speed, low power consumption, low memory consumption, large dynamic range, easiness of use, reproducibility, easiness of proof of properties of programs…

Nowadays, Floating-Point arithmetic is by far the most widely used computer number system in scientific computing for representing real numbers with words of fixed size, but many alternative systems have been suggested throughout the history of computer arithmetic (examples can be found in [18, 13, 21, 14, 5, 15, 17, 20, 1, 19, 8]). In the following, we assume that your favorite guru (or maybe yourself) has designed a new number system, where real numbers are represented by a w-bit word (systems with variable-length representations are therefore excluded). You wish to convince the world that it is far better than any other number system invented by humankind since the dawn of time. Let us review some tips that can help you in that purpose. Of course, any resemblance to real and actual situations is purely coincidental.

In this paper, we only focus on accuracy matters. As said above, there are obviously many other aspects that are worth being considered when promoting a number system, for instance, the speed, complexity and power consomption of the arithmetic operators. However these aspects are already very often taken into account, as if they were the only ones that matter (as Kahan once pointed out [12], *The Fast drives out the Slow even if the Fast is wrong*).

Assume that $\mathcal{M}$ is the (finite) set of the numbers that are exactly representable in your system, that $\mathcal{M}_{\min}$ and $\mathcal{M}_{\max}$ are the smallest and largest finite elements of $\mathcal{M}$, respectively. For $x \in \mathcal{M}, x \neq \mathcal{M}_{\max}$, x^+ is the smallest element of $\mathcal{M}$ larger than

x, and For $x \in \mathcal{M}, x \neq \mathcal{M}_{\min}$, x^- is the largest element of $\mathcal{M}$ less than x. Some notions, classical in floating-point arithmetic but easily generalizable to any finite-word-length arithmetic will be used here and there. They are defined as follows.

- if $\hat{x} \in \mathcal{M}$ is the result of a computation that approximates an exact value $x \in \mathbb{R}$, we will say that $\hat{x}$ is a *faithful result* if:
  - $x \in \mathcal{M}$ and $\hat{x} = x$; or
  - $x \notin \mathcal{M}$ and $\hat{x}$ is either the largest element of $\mathcal{M}$ less than x or the smallest element of $\mathcal{M}$ larger than x.
- if $x \notin \mathcal{M}$ then $\mathrm{ulp}(x)$ is the distance between the two consecutive elements of $\mathcal{M}$ that surround x. If $x \in \mathcal{M}$ then $\mathrm{ulp}(x)$ is the largest of $x - x^-$ and $x^+ - x$.
- if $x \in \mathbb{R}$, $\mathrm{RN}(x)$ is the element of $\mathcal{M}$ that is closest to x. If there are two such elements (i.e., x is halfway between two consecutive elements of $\mathcal{M}$), we assume there is a tie-breaking rule for deterministically choosing which one of them is chosen.

These definitions are not necessarily valid when the number x does not lie in the interval $(\mathcal{M}_{\min}, \mathcal{M}_{\max})$ but this does not matter for the purposes of this paper.

1 Build artificial problems whose solution is an exact number in your small format

This is by far the most useful trick. Assume that w is small (say, 8 to 32 bits). There are infinitely many real numbers, and there are at most 2^w numbers that can be represented in a w-bit format. Hence, the probability that the exact solution of a given *real-life* problem of a significant size should be exactly representable in your system (i.e., belongs to $\mathcal{M}$) is 0 or very near 0. But of course, one can always easily build *artificial* problems whose solutions belong to $\mathcal{M}$. With such problems, if we are skillfull (and possibly lucky) enough, we will obtain a faithful result, i.e., in this case, an *exact* result (just because for small values of w, there are not so many elements of $\mathcal{M}$ near that exact result). Then we perform the same calculation in quad precision floating-point arithmetic to obtain (almost certainly) a nonzero error.

You are then able to proclaim that your small w-bit format is far superior to quad precision! (of course, don't mention that for an overwhelming majority of problems quad precision is much more accurate).

An example? Let us consider the evaluation of function

$$f(x) = \sqrt{\frac{1}{\cos^2(\arctan(x))} - 1}, \tag{1}$$

and let us pretend we do not know that $f(x) = |x|$ (so that by definition, the result is exactly representable in your small format). Let us compare binary floating-point formats with various values of the precision p. If we evaluate f by straightforwardly

following (1) in round-to-nearest, ties-to-even arithmetic, and assuming that the arctangent and the cosine are correctly rounded, then, for $x = 513/512$, we will obtain an exact result in IEEE 754 binary16 arithmetic [9] ($p = 11$), while binary64/double precision arithmetic will give an error 2^{-52}. The conclusion is inescapable: binary16 is much more accurate than binary64!

Even better, let me invent the M-numbers: a floating-point format with 3-bit significands, and extremal exponents -2 and 1 (so that with an adequate coding, the exponents fit in 2 bits): appending a sign bit, the whole format requires 6 bits. In that format, with $x = 5/4$, $f(x)$ is computed exactly, whereas binary128 ($p = 113$) arithmetic painfully gives error 2^{-111}. I can proudly claim that the 6-bit M-number system is much more accurate than the 128-bit quad precision format!

In a similar vein, consider the following example (not so infrequent in the literature: it is used to illustrate the non-associativity of floating-point addition). We want to compute

$$(1 + 2^{200}) - 2^{200}.$$

If we perform that calculation in quad precision, we obtain 0. If we perform it in double-double arithmetic we get the correct result. Does it mean that double-double arithmetic is better than quad precision? Of course not: in the overwhelming majority of cases the relative errors of double-double arithmetic operations are of the order of 2^{-106} [11], whereas those of quad precision operations are bounded by 2^{-113}. We just used the highly artificial fact that the result of the first operation $(1 + 2^{200})$ is exactly representable in double-double arithmetic.

A hint: when someone claims a small error, he or she may well be right. If he or she claims *zero* error, this might well be because the trick presented in this section is used (quite possibly not consciously, of course).

2 Build artificial calculations that favor the domains where your number system is more accurate

A nonzero number x is represented by $\mathrm{RN}(x)$ with relative error $|x - \mathrm{RN}(x)|/|x|$, which can be as large as $\frac{1}{2}\mathrm{ulp}(x)/|x|$. Hence, that value—let us call it $\rho(x)$—is a good measurement of the local relative accuracy of a number system. Logarithmic number systems [13, 21] have a constant relative representation error bound $\rho(x)$ in all their range. A radix-β floating-point system has an "almost constant" relative representation error bound: the ratio between the maximum and the minimum value of $\rho(x)$ equals β. More exotic number systems (such as Clenshaw and Olver's Level-index systems [5, 20], Morris' Tapered floating-point arithmetic [18], of which Gustafson and Yonemoto's Posits [8, 7] are a skillfully crafted recent descendent) have a ρ function that varies much, with the effect of "focusing" the accuracy at places where it is considered most important (typically, around moderate orders of magnitude, the idea being that tiny and huge numbers do not need much accuracy and are less likely to appear in calculations).

A good way of building examples for which your number system looks great is to devise them so that all (or almost all) intermediate values of the calculation lie in the

areas where function ρ is small.

To illustrate this, let us pretend we are not aware of the pioneering works on the comparison of floating-point systems by Kuki and Cody [16], Cody [6], and Brent [3], and let us consider two different floating-point arithmetics: radix 2 *without implicit first bit convention*,[1] and radix 16. Fair comparison requires that for both arithmetics, the word length w should be the same, and the largest (and smallest positive) representable numbers should be almost the same. Since radix 16 requires two less exponent bits than radix 2 to represent extremal values of similar order of magnitude (because $16^k = 2^{4k}$), if we allow p bits to the significands of the radix-2 system, we must allow $p + 2$ bits to the significands of the radix-16 system (and since a radix-16 digit fits in 4 bits, $p + 2$ must be a multiple of 4).

Figure 1 presents the respective ρ functions for two such systems: a radix-2 system with p-bit significands, and a radix-16 system with $(p + 2)/4$-hexadecimal-digit significands (it suffices to visualize these functions for x between 1 and 16 because for both systems—barring underflow or overflow—$\rho(16^k \cdot x) = \rho(x)$). Let us divide the set of the real numbers whose absolute values lie between the minimum and maximum representable numbers of both formats into three areas: area A is the reunion of the intervals of the form $16^k \times [1, 2)$, area B contains the intervals of the form $16^k \times [2, 4)$, and area C consists of the union of the intervals of the form $16^k \times [4, 16)$.

From Figure 1, one will infer that the radix-2 system will be more accurate than the radix-16 system in domain A, that the radix-16 system will be better than the radix-2 system in domain C, and that both systems will be of comparable accuracy in domain B.

Once you know that, the trick is easy: if you wish to "prove" that the radix-2 system is the best, just build an artificial example of calculation for which all intermediate results lie in domain A, and if you prefer "proving" than the hexadecimal system is better, build your example so that all intermediate values belong to domain C. This can be done as follows. Let us assume $p = 14$ (binary significands with 14 bits, and hexadecimal significands with 4 digits), and consider the following "algorithm":

Algorithm 1 (TakeRootThenSquare)
$x \leftarrow x_0$
for i *from* 1 *to* n ***do***
  $x \leftarrow \mathrm{RN}(\sqrt{x})$
end for
for i *from* 1 *to* n ***do***
  $x \leftarrow \mathrm{RN}(x^2)$
end for
return x

Of course, if the operations were exact, the returned result should be x_0. To be fair, we must choose a value of x_0 that is exactly representable in both systems.

- An advocate of radix 2 can choose the starting point $x_0 = 1.00390625$ and $n = 5$, ensuring that all intermediate results are in area A. The returned result

[1] With the implicit first bit convention, radix 2 is *always* better.

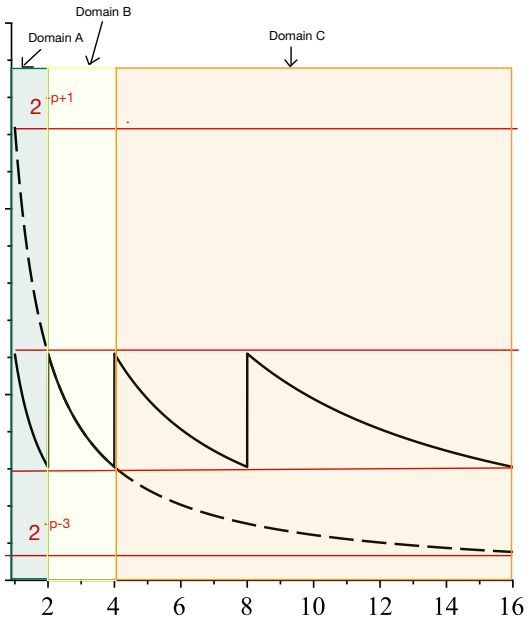

Figure 1: Relative representation error bounds for two different floating-point formats: radix 2 and precision-p without implicit bit convention (plain line); and radix 16 and precision $(p+2)/4$.

will be exact in radix 2, and it will be 1 in radix 16: we therefore conclude that radix 2 is far better than radix 16;

- a radix-16 fanatic can choose $x_0 = 0.8125$ and $n = 10$. The returned result will be 0.7781982421875 in radix 2 and 0.8032989501953125 in radix 16, which is significantly better (the difference is less impressive, probably because in real life, radix 2 *is* better than radix 16).

Similarly, if one wishes to "show" that tapered arithmetics are better than conventional floating-point, it suffices to build a calculation whose intermediate results are of the order of magnitude of 1. Again, Algorithm 1 is a perfect candidate for this because it forces all intermediate variables to be closer to 1 than x_0. Indeed, when running Algorithm 1 in binary32 [9] and Posit32 (with $\mathrm{es} = 2$) [7] arithmetics, with $x_0 = 1.5$, and $n = 15$, the returned result is $1.49967239\cdots$ in Posit32 arithmetic and $1.49517345\cdots$ in binary32 arithmetic. We can see that the Posit32 result is much closer to the exact result than the binary32 result.

Conversely, to "show" the superiority of floating-point, it suffices to have big or tiny intermediate results[2] (not necessarily all: having a very tiny or big intermediate value at a critical place may sometimes suffice). A simple solution for obtaining that is to modify Algorithm 1. In the first **for** loop, Algorithm 1 was building intermediate

[2]Not *too* big or tiny: we need to avoid overflows and subnormal numbers!

values closer and closer to 1 by iteratively taking square roots, and then, in the second **for** loop, by iteratively computing squares we were supposed to retrieve the initial value. If we switch both **for** loops, i.e., if we run Algorithm 2 (which, like Algorithm 1, would return x_0 if the arithmetic was exact), we will start by building intermediate values farther and farther from 1, i.e., closer and closer to places where floating-point arithmetic is better.

Algorithm 2 (TakeSquareThenRoot)
$x \leftarrow x_0$
for i *from* 1 *to* n ***do***
  $x \leftarrow \mathrm{RN}(x^2)$
end for
for i *from* 1 *to* n ***do***
  $x \leftarrow \mathrm{RN}(\sqrt{x})$
end for
return x

With Algorithm 2, assuming $x_0 = 1.0625$ and $n = 10$, we obtain the exact result 1.0625 in binary32 arithmetic, and $1.0624947\cdots$ in Posit32 arithmetic: as expected, on this example, binary32 arithmetic is slightly better.

The point is that with artificially-designed toy examples one can "prove" anything one wants: it is a vain exercise (even if it can be very effective at convincing non-numerically-aware readers). What matters is: *in real life applications, with problems of a significant size, what happens?* Concerning tapered arithmetic for instance, for a given numerical problem, can we really assume that huge or tiny intermediate values will not appear at places where this might hinder the accuracy of the calculation? The disappointing yet inescapable answer is: *it depends on the problem.* Fortunately, careful studies are starting to appear [4] (with mixed conclusions, by the way: not everything is black or white). Of course, when the appearance of huge or tiny intermediate values cannot be excluded, with a very careful preliminary analysis (similar to what our fathers—and still many people from the signal processing community—would do when using the first computers with fixed-point arithmetic), one can try to *scale* the calculations, by putting explicit scale factors here and there to ensure that all intermediate results remain of a moderate order of magnitude. But if we are able and willing to accomplish that long, tedious (and, above all, error-prone!) task, why not just using good old fixed-point? Or, possibly, new floating-point formats with a smaller exponent field?

3 Perform one billion experiments and just show the three ones that support your claims

No need to explain: that trick has been used for decades, sometimes with much success, in almost all areas of science. But nobody does that in computer arithmetic.

4 Ignore exponential growth with disdain

If your number system is nonredundant, you can represent 2^w different numbers with w-bit words. In practice, this will be slightly less: first because redundancy is sometimes a good idea, and maybe more importantly because in practice it is useful to reserve special words for non-fully-numerical information such as infinities and the results of indeterminate or forbidden operations. However, we can reasonably assume that the number of representable numbers grows exponentially with w (the converse would mean that your number system is highly inefficient). Exponential growth implies that some problems that are very easily solved for small w become very difficult, or even intractable, with large w. Let us call these problems *arduously scalable* problems. A typical example is function testing: the best solution to be absolutely certain of the reliability of an implementation of the sine function in binary32 arithmetic is to try it with all 2^{32} possible input values, which takes at most a few hours on a recent laptop. Of course, this becomes impossible with wider formats and certain validation of a binary64 function requires formal proof, which is much more complex.

Here, the tip is subtle: for a problem that is arduously scalable in usual number systems, show with your system a nice solution for small w, with beautiful and impressive figures, formulas and drawings, and just let the reader assume (without you actually needing to say it) that it easily generalizes to larger values of w.

An example of an arduously scalable problem is the Table Maker's Dilemma (TMD). The TMD can be considered with all possible rounding functions, but for the sake of simplicity, let us consider round-to-nearest only (with any choice in case of a tie—indeed for implementing the most usual transcendental functions, Lindemann's Theorem [2] implies that there are no ties). The problem is the following: for a given number system, characterized here by the discrete set $\mathcal{S}$ of its exactly representable numbers) and a given transcendental function f (either one of the usual "elementary functions" $\sin$, $\cos$, $\exp$, $\arctan$, …, or a more complex "special" function), we wish to design a program that always returns, when being given $x \in \mathcal{S}$ in input, the element of $\mathcal{S}$ that is nearest to $f(x)$. There are two facts that cannot be escaped here:

- a transcendental function cannot be exactly expressed as a finite sequence of arithmetic operations. Hence, f can only be *approximated* (usually, by a piecewise polynomial or rational function). The approximation can (and in practice must) be computed in a somehow wider format (this can be explicit, or implicit, e.g. using tricks such as representing some intermediate results as unevaluated sums or products of elements of $\mathcal{S}$, i.e., as "double-words" or "triple-words"), so that the information we have is that $f(x)$ belongs to some (hopefully small) interval I_x. Typically, we obtain an approximation $\hat{f}(x)$ that belongs to a set wider than $\mathcal{S}$ and an error bound. Beware: in some cases (e.g. using an "alternate" series and carefully playing with directed roundings in intermediate calculations) we know the *sign* of the error. This does not solve the problem:[3] this just means that $\hat{f}(x)$ is one of the ends of I_x, but still the only information we have on $f(x)$ is that it belongs to I_x;

[3] This is a frequent misunderstanding.

- $\mathcal{S}$ is a *discrete* set. This implies that, unless we are in the special cases where $\min I_x \leq \min \mathcal{S}$ or $\max I_x \geq \max \mathcal{S}$, if I_x is small enough, it contains 0 or 1 middle of two consecutive elements of $\mathcal{S}$ (in the following, we call such elements "midpoints").

The TMD is the problem of *i)* determining all the $x \in \mathcal{S}$ such that $f(x)$ is a midpoint, and *ii)* the minimum distance between $f(x)$ and a midpoint when $f(x)$ is *not* a midpoint (that distance can be a *relative* distance if this is more convenient for the number system and function under consideration). Solving that problem is necessary if one wishes to know what the accuracy of the approximation $\hat{f}(x)$ must be to always guarantee correct rounding.

Point i) belongs to the realm of algebra: for many functions ($\sin(x)$, $\cos(x)$, $\arctan(x)$, $\exp(x)$, $\ln(x)$) Lindemann's Theorem implies that $f(x)$ is never a midpoint; and for several algebraic functions, we have a partial answer [10]. For more general functions (e.g., Bessel functions), almost nothing is known. For Point ii) the choice of the number system will not change much the problem: we need to consider a number of input points that grows exponentially with the width w of the number format, and the only possible improvements come from properties of the function itself (for instance in FP arithmetic, this is the case of function 2^x) that may allow one to somehow reduce the considered input domain.

But the TMD is *not* a difficult problem…when w is small. It is very simple for $w = 16$, and not-so-difficult for $w = 32$. Let me give just an example. It takes 9 seconds only (on my Macbook Pro) to find that the hardest to round case for the arctangent of a normal number in Binary16 arithmetic is attained with the input number:

$$x = \frac{823}{128} = 110.0110111_2.$$

We have (in binary)

$$\arctan(x) = 1.01101010101000000000000000010001_2 \cdots$$

which implies that $\arctan(x)$ is within 8.4778×10^{-6} ulps from the exact middle of two consecutive Binary16 numbers. The toy Maple program used for that is ridiculously simple:

```
checkp := proc(emin,emax,p);
maxulpdist := 7890; Digits := 90;
twotopminusone := 2^(p-1); twotop := 2*twotopminusone;
Mmin := twotopminusone; Mmax := twotop-1;
for exponent from emin to emax do
   scalex := 2^(exponent-p+1);
   scaley := 1;
   for mantissa from Mmin to Mmax do
      x := mantissa * scalex;
      y := evalf(arctan(x));
      if y > scaley then
              while y >= 2*scaley do scaley := 2*scaley end do
        else while y < scaley do scaley := scaley*0.5 end do
      end if;
      Y := y*Mmin/scaley;
      if abs(frac(Y)-0.5) <= maxulpdist then
```

```
            maxulpdist := abs(frac(Y)-0.5);
            xmax := x
      end if;
   end do;
      end do;
printf("minimum distance to a midpoint %a ulps, attained for
                     x = %a\n", evalf(maxulpdist,15), xmax);
end:
```

Unfortunately, a rapid calculation shows that the same program would take around 80 million years to do the same thing in binary64 arithmetic.

Conclusion

I hope I have convinced you that one can "show" just anything one wants by carefully crafting small examples (or by using the heavy artillery, as suggested in Section 3). This is sterile: *toy examples are not proofs*. They are extremely useful for *pedagogical* purposes, because they make it possible to illustrate various situations that may arise in computing. They can also sometimes serve as *counterexamples* to wrong claims. However, really showing the interest of a number system, a numerical method, etc., requires much more work: an in-depth analysis whenever possible, and when analysis is not possible or not fully concluding, heavy testing on real-life, real-size, problems, with real-life input data.

References

[1] Aqil M. Azmi and Fabrizio Lombardi. On a tapered floating point system. In *9th Symposium on Computer Arithmetic, ARITH 1989, Santa Monica, CA, USA, September 6-8, 1989*, pages 2–9. IEEE, 1989.

[2] A. Baker. *Transcendental Number Theory*. Cambridge University Press, 1990.

[3] R. P. Brent. On the precision attainable with various floating point number systems. *IEEE Transactions on Computers*, C-22(6):601–607, June 1973.

[4] N. Buoncristiani, S. Shah, D. Donofrio, and J. Shalf. Evaluating the numerical stability of posit arithmetic. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 612–621, 2020.

[5] C. W. Clenshaw and F. W. J. Olver. Beyond floating point. *Journal of the ACM*, 31:319–328, 1985.

[6] W. J. Cody. Static and dynamic numerical characteristics of floating-point arithmetic. *IEEE Transactions on Computers*, C-22(6):598–601, June 1973.

[7] Posit Working Group. Posit standard documentation, release 3.2-draft. Technical report, June 2018. https://posithub.org/docs/posit_standard.pdf.

[8] J.L. Gustafson and I. Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations: an International Journal*, 4(2), june 2017.

[9] IEEE. *IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2019)*. July 2019.

[10] C.-P. Jeannerod, N. Louvet, J.-M. Muller, and A. Panhaleux. Midpoints and exact points of some algebraic functions in floating-point arithmetic. *IEEE Transactions on Computers*, 60(2), February 2011.

[11] M. Joldes, J.-M. Muller, and V. Popescu. Tight and rigorous error bounds for basic building blocks of double-word arithmetic. *ACM Transactions on Mathematical Software*, 44(2), 2017.

[12] W. Kahan. The baleful effect of computer benchmarks upon applied mathematics, physics and chemistry. https://people.eecs.berkeley.edu/ wkahan/ieee754status/baleful.pdf, 1996.

[13] N. G. Kingsbury and P. J. W. Rayner. Digital filtering using logarithmic arithmetic. *Electronic Letters*, 7:56–58, 1971. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[14] P. Kornerup and D. W. Matula. Finite-precision rational arithmetic: an arithmetic unit. *IEEE Transactions on Computers*, C-32:378–388, 1983.

[15] P. Kornerup and D. W. Matula. Finite precision lexicographic continued fraction number systems. In *Proceedings of the 7th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, Los Alamitos, CA, 1985. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 2, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[16] H. Kuki and W. J. Cody. A statistical study of the accuracy of floating-point number systems. *Communications of the ACM*, 16(14):223–230, April 1973.

[17] D. W. Matula and P. Kornerup. Finite precision rational arithmetic: Slash number systems. *IEEE Transactions on Computers*, 34(1):3–18, 1985.

[18] R. Morris. Tapered floating point: A new floating-point representation. *IEEE Transactions on Computers*, C-20(12):1578–1579, 1971.

[19] J.-M. Muller, A. Scherbyna, and A. Tisserand. Semi-logarithmic number systems. *IEEE Transactions on Computers*, 47(2), February 1998.

[20] F. W. J. Olver and P. R. Turner. Implementation of level-index arithmetic using partial table look-up. In *Proceedings of the 8th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, Los Alamitos, CA, May 1987.

[21] E. E. Swartzlander and A. G. Alexpoulos. The sign-logarithm number system. *IEEE Transactions on Computers*, December 1975. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.