New directions in floating-point arithmetic

Nelson H. F. Beebe

University of Utah
Department of Mathematics, 110 LCB
155 S 1400 E RM 233
Salt Lake City, UT 84112-0090
USA

Abstract. This article briefly describes the history of floating-point arithmetic, the development and features of IEEE standards for such arithmetic, desirable features of new implementations of floating-point hardware, and discusses work-in-progress aimed at making decimal floating-point arithmetic widely available across many architectures, operating systems, and programming languages.

Keywords: binary arithmetic; decimal arithmetic; elementary functions; fixed-point arithmetic; floating-point arithmetic; interval arithmetic; `mathcw` library; range arithmetic; special functions; validated numerics
PACS: 02.30.Gp Special functions

DEDICATION

This article is dedicated to N. Yngve Öhrn, my mentor, thesis co-advisor, and long-time friend, on the occasion of his retirement from academia. It is also dedicated to William Kahan, Michael Cowlishaw, and the late James Wilkinson, with much thanks for inspiration.

WHAT IS FLOATING-POINT ARITHMETIC?

Floating-point arithmetic is a technique for storing and operating on numbers in a computer where the base, range, and precision of the number system are usually fixed by the computer design.

Conceptually, a floating-point number has a *sign*, an *exponent*, and a *significand* (the older term *mantissa* is now deprecated), allowing a representation of the form $(-1)^{\text{sign}} \times \text{significand} \times \text{base}^{\text{exponent}}$. The *base point* in the significand may be at the left, or after the first digit, or at the right. The point and the base are implicit in the representation: neither is stored.

The sign can be compactly represented by a single bit, and the exponent is most commonly a biased unsigned bitfield, although some historical architectures used a separate exponent sign and an unbiased exponent. Once the sizes of the sign and exponent fields are fixed, all of the remaining storage is available for the significand, although in some older systems, part of this storage is unused, and usually, ignored. On modern systems, the storage order is conceptually sign, exponent, and significand, but addressing conventions on byte-addressable systems (the *big endian* versus *little endian* theologies) can alter that order, and some historical designs reordered them, and sometimes split the exponent and significand fields into two interleaved parts. Except when the low-level storage format must be examined by software, such as for binary data exchange, these differences are handled by hardware, and are rarely of concern to programmers.

The data size is usually closely related to the computer word size. Indeed, the venerable Fortran programming language mandates a *single-precision* floating-point format occupying the same storage as an integer, and a *double-precision* format occupying exactly twice the space. This requirement is heavily relied on by Fortran software for array dimensioning, argument passing, and in `COMMON` and `EQUIVALENCE` statements for storage alignment and layout. Some vendors later added support for a third format, called *quadruple-precision*, occupying four words. Wider formats have yet to be offered by commercially-viable architectures, although we address this point later in this article.

Floating-point arithmetic can be contrasted with *fixed-point* arithmetic, where there is no exponent field, and it is the programmer's responsibility to keep track of where the base point lies. Address arithmetic, and signed and unsigned integer arithmetic, are special cases of fixed-point arithmetic where the base point is always at the right, so only whole numbers can be represented, and for address and unsigned integer arithmetic, the storage occupied by the sign is given to the number field.

Floating-point number systems have limited precision and range, and their arithmetic is *not* associative. These properties are at odds with mathematical arithmetic, and often require great care to handle correctly in software. They often produce large gaps between a problem's mathematical solution and a practical, and accurate, computational solution.

Also, compiler optimizations, instruction reordering, and use of higher precision for intermediate computations, can all produce unpleasant surprises in floating-point software. Consequently, numerical programmers must be highly experienced, eternally vigilant, and fanatic about testing on every accessible platform.

EARLY FLOATING-POINT ARITHMETIC

Konrad Zuse designed and built the Z1, Z2, Z3, and Z4 computers in Germany during the period 1936–1945. The Z3 is believed to be the world's first working programmable computer, albeit electromechanical rather than fully electronic. The Z1, Z2, and Z3 had a 22-bit memory word, and the Z4 had a 32-bit word. Memory size was only 64 words, and was reserved for data: instructions were read from punched cards. All but the Z2 had floating-point arithmetic, with an approximate nonzero magnitude range of $2^{\pm 63} \approx 10^{\pm 19}$. In addition, the arithmetic had special representations for *infinity* and *undefined*, so that the machine could continue working when results were generated that could not be represented as normal floating-point numbers.

Zuse's pioneering work was not widely known until decades later, and most of the computers built in Britain and the US during the 1940s and 1950s had fixed-point arithmetic, rather than floating-point arithmetic. This has been attributed to an influential early study of computer arithmetic [1] that argued

> There is, of course, no denying the fact that human time is consumed in arranging for the introduction of suitable scale factors. We only argue that the time so consumed is a very small percentage of the total time we will spend in preparing an interesting problem for our machine. In order to have such a floating point, one must waste memory capacity which could otherwise be used for carrying more digits per word.
> Arthur W. Burks, Herman H. Goldstine, and John von Neumann,
> *Preliminary discussion of the logical design of an electronic computing instrument* (1946).

Later experience demonstrated the error of this view [2]:

> It is difficult today to appreciate that probably the biggest problem facing programmers in the early 1950s was scaling numbers so as to achieve acceptable precision from a fixed-point machine.
> Martin Campbell-Kelly,
> *Programming the Mark I: Early Programming Activity at the University of Manchester* (1980).

Consequently, by the early 1960s, most computers built for engineering and scientific computation included floating-point arithmetic, and for the majority, this arithmetic used base 2, although a few designs used base 3, 4, 8, 10, 16, or 256. The most important from this second group is the ground-breaking, and extraordinarily long-lived and commercially successful, IBM System/360 family, introduced in 1964, and still marketed today as the IBM zSeries mainframes. Except as noted later, this architecture uses hexadecimal (base-16) floating-point arithmetic, with 32-bit, 64-bit, and 128-bit formats, each with a 7-bit power-of-16 exponent, corresponding to a nonzero magnitude range of $[16^{-64}, 16^{63}] \approx [10^{-77}, 10^{75}]$. An excellent description of many of these early architectures can be found in the treatise by IBM System/360 architects Blaauw and Brooks [3], which covers computers up to about 1980, or roughly, the start of the desktop microcomputer era.

Decimal (base-10) floating-point arithmetic was offered by only a few vendors, and largely disappeared from the market by the mid 1960s, although it reappeared in hand-held electronic calculators made possible by the microprocessor revolution that began with the Intel 4004 in 1971.

WHY THE BASE MATTERS

Most programmers give little thought to the choice of the floating-point base, and indeed, on most systems, that choice has already been made for them by the computer architects. Nevertheless, the base *does* matter, for at least these reasons:

- complexity, efficiency, and speed of chip circuitry;
- accuracy and run-time cost of conversion between internal and external (usually decimal) bases;
- effective precision varies when the floating-point representation uses a radix larger than 2 or 10;
- reducing the exponent width makes digits available for increased precision;
- for a fixed number of exponent digits, larger bases provide a wider exponent range;
- for a fixed storage size, granularity (the spacing between successive representable numbers) increases as the base increases;
- in the absence of underflow and overflow (that is, results outside of the representable range), multiplication by a power of the base is an *exact* operation, and this feature is *essential* for many computations, in particular, for accurate elementary and special functions.

THE BASE-CONVERSION PROBLEM

When the internal and external bases differ, data conversion is required, and it may be impossible to represent the target result exactly (for example, decimal 0.9 has an infinite binary expansion $0.1110\,0110\,0110\,0110\ldots_2$). Goldberg [4] and Matula [5, 6] showed in the late 1960s how many digits are required for accurate round-trip conversion between two bases, although few programmers or textbook authors appear to be aware of that work. Producing an *exact* decimal output representation of an internal binary floating-point number, and the correctly-rounded reverse conversion, can require a surprisingly large number of digits: more than 11,000 decimal digits in the quadruple-precision format. The base-conversion problem seems to be viewed by many as trivial, but it is not: it took more than 50 years of computer use before the problem was properly solved in the 1990s [7, 8, 9, 10, 11, 12, 13]. Even today, more than 15 years after the earliest of those papers, most vendors, and most programming languages, fail to guarantee correct base conversion in source code, input, and output. This in turn makes accurate specification of numeric data difficult [13]:

> While working on the BEEF tests for transcendental functions, it was discovered that the Turbo C 1.0 compiler did not convert 11.0 exactly into a floating point number equal to 11!
> Guy L. Steele Jr. and Jon L. White,
> *Retrospective: How to Print Floating-Point Numbers Accurately* (2003).

This author has experienced similar flawed behavior in a few other compilers and run-time libraries that refuse to handle a number with more than some limited number of digits, or produce a wildly-incorrect conversion when additional digits are appended.

Since the external base is almost always 10, the floating-point conversion problem can be eliminated if computers use decimal floating-point arithmetic internally. We return to this later.

FLAWS OF EARLY FLOATING-POINT DESIGNS

There have been commercially-successful historical computers with floating-point arithmetic that produced unexpected behavior like these examples:

- $u \neq 1.0 \times u$;
- $u + u \neq 2.0 \times u$;
- $u \times 0.5 \neq u/2.0$;
- $u \neq v$ but $u - v = 0.0$, and $1.0/(u - v)$ raises a zero-divide error;
- $u \neq 0.0$ but $1.0/u$ raises a zero-divide error;
- $u \times v \neq v \times u$;
- underflow wraps to overflow, and vice versa.

In older floating-point designs, rounding behavior was often suboptimal, using truncation, rather than rounding-to-nearest, increasing cumulative rounding error.

THE IEEE 754 AND 854 STANDARDS

The deficiencies of older computer arithmetic systems, and the important leadership of distinguished mathematician, numerical analyst, and computer scientist, William Kahan at the University of California, Berkeley [14, 15] [Kahan received the prestigious 1990 *ACM Turing Award* for this work] resulted in the collaborative development in the late 1970s of a new floating-point design with these important features:

- binary (base-2) arithmetic;
- four dynamically-accessible rounding modes, defaulting to round-to-nearest, with ties going to the adjacent even value;
- representations of signed zero, signed *Infinity*, and two kinds of *NaN* (Not a Number, Zuse's *undefined*);
- 32-bit, 64-bit, and 80-bit formats offering 24, 53, and 64 bits of significand precision (about 7, 15, and 19 decimal digits), and nonzero magnitude ranges of about $[10^{-45}, 10^{38}]$, $[10^{-324}, 10^{308}]$, and $[10^{-4951}, 10^{4932}]$;
- gradual underflow to *subnormal* (formerly, *denormalized*, but that term is now deprecated) numbers, by relaxation of the normalization requirement when the smallest exponent is reached;
- hidden (not stored) bit for normalized values in the 32- and 64-bit formats;
- default nonstop computation in the presence of exceptional values, instead of the traditional behaviors of raising an exception and terminating the job, or simple getting a radically-wrong answer and continuing without warning;
- program-accessible sticky exception flags that record the occurrence of *divide by zero*, *invalid* operation or operand, *inexact* result, *overflow*, and *underflow*, and on some processors, as many as a dozen more.

Almost all new CPU architectures introduced since the mid 1980s with floating-point arithmetic are based on this design.

Rounding-direction control allows implementation of *interval arithmetic*, a computational method that retains rigorous upper and lower bounds for all numerical results. It is sometimes called *range arithmetic* or *validated numerics*. This small, but increasingly important, area deserves to be much more widely appreciated, taught, and used; there has even been a serious call [16] for mandating its use in US government projects and general scientific research. In late 2005, Sun Microsystems released freely-available C++ and Fortran compilers with support for interval arithmetic; they run on Solaris on SPARC, AMD64, and IA-32 CPUs, and also on GNU/Linux on AMD64 and IA-32 CPUs.

A preliminary version of the new floating-point design was first implemented in the Intel 8087 numeric coprocessor chip in 1980, and continues today in the Intel IA-32, EM64T, and IA-64 architecture families, and the AMD AMD64 family whose success forced Intel to introduce the compatible EM64T family that is now provided by most new Intel processors. The final version was issued as IEEE Standard 754 in 1985 [17]. IEEE Standard 854-1987 later extended the design to a general base [18]. A significant revision that combines these standards has been underway for over a decade, and may be completed in 2008. One significant change in the 754 design after the 8087 chip was in wide use was the introduction of two kinds of NaN: quiet and signaling. The IA-32 processors provide only a single NaN, although the AMD64 and EM64T in 32-bit and 64-bit formats, and the IA-64 in all three formats, provide both NaNs.

Some vendors subsequently added support for a 128-bit format, providing a 113-bit significand (about 34 decimal digits), and about the same exponent range as the 80-bit format. In most cases, this is implemented in software, rather than hardware.

In 1999, the new IBM mainframe G5 processor added hardware support for IEEE 754 arithmetic (32-bit, 64-bit, and 128-bit formats) [12], while retaining the older hexadecimal arithmetic with its problems of *wobbling precision* and limited exponent range.

Sadly, despite the existence of hundreds of millions of desktop computers, mobile phones, and other embedded devices offering IEEE 754 arithmetic (although sometimes only subsets thereof), standardized vendor-independent software access to many of the feature of the system remains unavailable. The 1999 ISO C Standard defines a complete interface, but compilers and libraries conforming to that standard are still inaccessible to many users, and few other languages provide any access whatsoever, pretending that hardware details are an irrelevant, or at least nonstandard, abstraction.

Equally sadly, the lack of understanding of the behavior and needs of floating-point computation on the part of some language designers, notably those of Java and C#, resulted in severely, and utterly needlessly, crippled floating-point systems in those languages, which deserve the scathing commentary offered by Kahan and Darcy [19].

FUSED MULTIPLY-ADD OPERATION

The IBM POWER architecture introduced in 1990 included a new floating-point instruction that has since been found to be of critical importance for efficiency, accuracy, and correctness of many important algorithms: *fused multiply-add (FMA)*. This operation computes the value of $u \times v + w$ with an *exact* double-length product $u \times v$, followed by the addition of w with a *single* rounding to a normal-length result.

Sequences of interleaved multiplications and additions are extremely common in numerical computation, including matrix multiplication, solving linear equations and eigenvalue problems, computing dot products, and evaluating polynomials.

The FMA operation makes possible dot products that are provably correct to the next-to-last bit [20], software-pipelined division and square root that are provably correctly rounded [21], and significantly simplifies many elementary and special functions [21].

The 1999 ISO C Standard includes a library function, `fma(u,v,w)`, to provide platform-independent access to the fused multiply-add operation. Unfortunately, the function is difficult to implement in software, and then likely to be slow. It is not yet widely available, and on most GNU/Linux systems, is defective. There are also incorrect hardware implementations, such as in some CPU models of the MIPS architecture.

Hardware implementations of FMA are fast, and can sometimes double performance of numerical code. Some CPUs even implement multiply as `fma(u,v,0)` and add as `fma(u,1,w)`, dispensing with separate implementations of those two instructions entirely.

The biggest floating-point deficiency of several current architectures based on the IEEE 754 model is their lack of hardware FMA, and it is regrettable that now that its supreme importance is widely understood by floating-point experts and numerical analysts, it has not been scheduled for the next revision of every CPU chip.

DECIMAL ARITHMETIC

The Rexx and NetRexx scripting languages [22, 23] developed by IBM Fellow Michael Cowlishaw provide a software implementation of decimal floating-point arithmetic with up to a billion (10^9) digits, and a huge exponent range of $10^{\pm 999,999,999}$. Based on long experience with those languages, in 2006, IBM researchers developed a firmware implementation of IEEE-like decimal arithmetic for the z9 mainframe [24, 25], and on May 21, 2007, IBM announced the POWER6 processor with the first hardware implementation since the 1960s, outside of handheld calculators, of decimal floating-point arithmetic. For a survey of historical decimal systems, see [26].

In 2005, ISO committees for the standardization of the C and C++ languages received proposals [27, 28, 29, 30] for the incorporation of decimal floating-point arithmetic. In late 2006, GNU developers of compilers for those languages added preliminary support on one CPU platform for the new data types, and in 2007, more CPU platforms followed. The underlying arithmetic is supplied by Cowlishaw's `decNumber` library [31], but no debugger or library support for I/O and mathematical functions is included. The decimal formats are 32-bit, 64-bit, and 128-bit, with precisions of 7, 16, and 34 digits, respectively. Their nonzero magnitude ranges are approximately $[10^{-101}, 10^{97}]$, $[10^{-398}, 10^{385}]$, and $[10^{-6176}, 10^{6145}]$, somewhat wider than the corresponding binary formats.

The `decNumber` library provides a superset of the features of the IBM hardware implementation, including support for *eight* rounding modes (additional modes are needed to meet rounding rules in financial computations mandated by various legal jurisdictions), along with all of the other features of IEEE 754 arithmetic.

Unlike the binary format, where the point lies after the first digit, the decimal significand is an *integer coefficient*, and thus, trailing zeros allow multiple representations of the same numeric value. This design choice was intentional, because it allows decimal fixed-point arithmetic, historically required for financial computations, to be done in decimal floating-point arithmetic, and additional library functions make it possible to get and set, the scale, or *quantization*. In financial work, these operations are expected to be common.

One important ramification for programmers is that multiplication by 1., 1.0, 1.00, … each produce different quantization, and similarly, multiplication by 1000 differs in quantization from multiplication by 1×10^3. Since most people are taught in school to write at least one digit following a decimal point, programmers of decimal arithmetic need to learn to avoid introducing unwanted trailing zeros so as to preserve quantization.

Another significant point is that financial computations need large numbers of digits: older COBOL standards required support for 18 decimal digits, and the 2002 ISO COBOL Standard [32] mandates 32 digits. Thus, the 128-bit format is expected to be widely used, and market pressure will force it to be in hardware, rather than in software as is currently the case for the binary format of that size on several platforms.

The IBM hardware and software implementations of decimal arithmetic use an encoding called *DPD (Densely-Packed Decimal)* [31, 33], which represents three decimal digits in ten bits. DPD is more compact than older schemes, such as Binary-Coded Decimal (BCD), that have long been used for fixed-point decimal arithmetic in many processors.

Intel researchers have developed a competing encoding called *BID (Binary Integer Decimal)* and implemented it in a prototype reference library [34]. BID is designed to allow chip designers to reuse circuitry from integer arithmetic units, but appears to make correct decimal rounding difficult. Fortunately, detailed analysis has made it possible to develop computational algorithms that can guarantee correct decimal rounding for the basic operations of add, subtract, multiply, and divide. From the programmer's point of view, the two encodings offer identical floating-point characteristics, but there are small differences in the representable range that could make it possible for software to distinguish between them, and thus, make assumptions that limit portability.

In mid 2007, the GNU compilers were extended to generate code for both the IBM and Intel libraries, and it is expected that Intel's own compilers will soon have support for decimal arithmetic as well. It seems likely that future Intel processors will provide decimal arithmetic in hardware, once sufficient software experience with both BID and DPD encodings has been accumulated to guide the final choice of encoding. Since most desktop computers use CPUs from the Intel architecture families, in just a few years, decimal floating-point arithmetic may be widespread, and competitive with binary floating-point arithmetic in efficiency.

Since 2005, this author has been developing a large highly-portable elementary function library, called the `mathcw` library, that includes all of the mathematical repertoire of the 1999 ISO C Standard, plus a great deal more. This library has been designed to provide a comfortable C99 environment on all current platforms, as well as to run on historical architectures of the 1970s, such as the PDP-10, PDP-11, and VAX, which remain available via software virtual machines that, thanks to advances in processor technology, can be an order of magnitude faster than the hardware ever was. The library currently supports six binary types, and four decimal types, covering almost every significant computing platform of the last thirty years.

In addition, the `mathcw` library is designed to pave the way for future *octuple-precision* arithmetic in a 256-bit format offering a significand of 235 bits in binary, and 70 digits in decimal. The approximate nonzero magnitude ranges are $[10^{-315\,653}, 10^{315\,652}]$ in binary and $[10^{-1\,572\,932}, 10^{1\,572\,863}]$ in decimal.

While it is relatively straightforward to add support for new data types and machine instructions (or interfaces to software implementations thereof) in modern compilers, there is a huge hurdle to overcome in providing a powerful run-time library for I/O and mathematical functions. The `mathcw` library removes that obstacle, and makes decimal arithmetic as comfortable, and accessible, as binary arithmetic.

Fast and compact implementations of many of the elementary and special functions are based on a combination of range reduction, and evaluation of a rational polynomial that accurately represents the function itself, or more commonly, an auxiliary function from which the desired function can be easily obtained, but only over a limited argument interval. In particular, this means that the polynomial approximation must be adapted to both the interval, and the required precision, and it is then impossible to port the software to systems with differing precision, or to modify it for use on a different interval, without having the means to generate new polynomial approximations. To eliminate this problem, which is widespread in previous mathematical software libraries, the `mathcw` library uses polynomials output in formats suitable for use in C, Fortran, and `hoc` by auxiliary software written in the Maple symbolic-algebra language. The Maple output is manually copied into the library's header files. To facilitate algorithm modification, and use with even higher precision in the distant future, all of the Maple programs are included in the `mathcw` software distribution.

The I/O part of the library includes important extensions to allow handling of numbers in any base from 2 to 36 (e.g., `8@3.11037554@e+0`, `10@3.14159265@e+0`, `16@3.243f6c@e+0`, and `36@3.53i5g@e+0` all represent 32-bit approximations to π), control over exponent widths (a feature available in Fortran since the 1978 ISO Standard, but still lacking in the most-recent C, C++, and Java run-time libraries), and digit grouping with separating underscores, making long digit strings much more readable, so that

```
static const decimal_long_double PI = 3.141_592_653_589_793_238_462_643_383_279_503;
```

becomes a valid initialization in C or C++. Such numbers can be used in input and output files, and once compilers are trivially extended, also in programming-language source code.

The `mathcw` library includes an extensive collection of functions for *pair-precision arithmetic*, which simulates double-length significands in every supported precision. It is sometimes the case that somewhat higher precision in just a small part of a large computation can make a dramatic improvement in overall accuracy, and these functions fill that need.

The mathcw library can interfaced to many other programming languages, and interfaces to Ada, C#, C++, Fortran, Java, and Pascal are provided with the system.

As additional proof of concept, *five* scripting language compilers/interpreters (three for awk, plus hoc and lua) have been adapted to use decimal, rather than binary, floating-point arithmetic, and allow digit-separating underscores in source code. All of them pass their full validation suites as well as they do for the original versions that use binary arithmetic. In each case, less than 3% of the source code needed changes for decimal arithmetic, so this author is confident that the scores of other popular scripting languages implemented in C or C++, such as icon, javascript, perl, php, python, ruby, the Unix shells, and so on, can be similarly updated, each in less than a day's work, making decimal floating-point arithmetic the norm almost everywhere within just a few years. With highly-portable underpinnings in the form of the mathcw library, these languages could easily offer a much richer mathematical function repertoire, including access to all of the features in the IEEE 754 Standard, instead of the rather limited function set chosen for Fortran more than 50 years ago.

The mathcw library is described in a forthcoming book [35], and is to be released under a software license that ensures free and unrestricted source-code access to everyone, in the tradition of the numerical mathematics community, and the free software movement.

FURTHER READING

Overton's 100-page book [36] provides a useful introduction to more details of floating-point arithmetic than we can provide in this short article. This author's book [35] gives much more information, including guidance for creating numerical programs that are independent of precision and range, and usually independent of base.

There are numerous other books and technical papers on the subject of computer arithmetic, and the best advice is to consult the comprehensive on-line bibliography that is actively maintained by this author at http://www.math.utah.edu/pub/tex/bib/index-table.html#fparith.

REFERENCES

1. A. W. Burks, H. H. Goldstine, and J. von Neumann, Preliminary discussion of the logical design of an electronic computing instrument, Report to the U.S. Army Ordnance Department (1946), reprinted in [37, pp. 221–259] and [38, pp. 97–146].
2. M. Campbell-Kelly, *Annals of the History of Computing* **2**, 130–168 (1980), ISSN 0164-1239, URL http://dlib.computer.org/an/books/an1980/pdf/a2130.pdf;http://www.computer.org/annals/an1980/a2130abs.htm, see minor correction [39].
3. G. A. Blaauw, and F. P. Brooks, Jr., *Computer architecture: concepts and evolution*, Addison-Wesley, Reading, MA, USA, 1997, ISBN 0-201-10557-8.
4. I. B. Goldberg, *Communications of the Association for Computing Machinery* **10**, 105–106 (1967), ISSN 0001-0782.
5. D. W. Matula, *Proceedings of the American Mathematical Society* **19**, 716–723 (1968), ISSN 0002-9939.
6. D. W. Matula, *Communications of the Association for Computing Machinery* **11**, 47–50 (1968), ISSN 0001-0782.
7. W. D. Clinger, *ACM SIGPLAN Notices* **25**, 92–101 (1990), ISSN 0362-1340, URL http://www.acm.org:80/pubs/citations/proceedings/pldi/93542/p92-clinger/, see also output algorithms in [9, 10, 11, 12, 13].
8. D. Gries, "Binary to Decimal, One More Time," in [40], chap. 16, pp. 141–148, this paper presents an alternate proof of Knuth's algorithm [9] for conversion between decimal and fixed-point binary numbers.
9. D. E. Knuth, "A Simple Program Whose Proof Isn't," in [40], chap. 27, pp. 233–242, this paper discusses the algorithm used in TEX for converting between decimal and scaled fixed-point binary values, and for guaranteeing a minimum number of digits in the decimal representation. See also [7, 41] for decimal to binary conversion, [10, 13] for binary to decimal conversion, and [8] for an alternate proof of Knuth's algorithm.
10. G. L. Steele Jr., and J. L. White, *ACM SIGPLAN Notices* **25**, 112–126 (1990), ISSN 0362-1340.
11. R. G. Burger, and R. K. Dybvig, *ACM SIGPLAN Notices* **31**, 108–116 (1996), ISSN 0362-1340, URL http://www.acm.org:80/pubs/citations/proceedings/pldi/231379/p108-burger/, this paper offers a significantly faster algorithm than that of [10], together with a correctness proof and an implementation in Scheme. See also [7, 12, 13, 41].
12. P. H. Abbott, D. G. Brush, C. W. Clark III, C. J. Crone, J. R. Ehrman, G. W. Ewart, C. A. Goodrich, M. Hack, J. S. Kapernick, B. J. Minchau, W. C. Shepard, R. M. Smith, Sr., R. Tallman, S. Walkowiak, A. Watanabe, and W. R. White, *IBM Journal of Research and Development* **43**, 723–760 (1999), ISSN 0018-8646, URL http://www.research.ibm.com/journal/rd/435/abbott.html, besides important history of the development of the S/360 floating-point architecture, this paper has a good description of IBM's algorithm for exact decimal-to-binary conversion, complementing earlier ones [7, 9, 10, 11, 13].

13. G. L. Steele Jr., and J. L. White, *ACM SIGPLAN Notices* **39**, 372–389 (2004), ISSN 0362-1340, best of PLDI 1979–1999. Reprint of, and retrospective on, [10].
14. C. Severance, An interview with the old man of floating-point. Reminiscences elicited from William Kahan, World-Wide Web document. (1998), URL `http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html`, a shortened version appears in [15].
15. C. Severance, *Computer* **31**, 114–115 (1998), ISSN 0018-9162, URL `http://pdf.computer.org/co/books/co1998/pdf/r3114.pdf`.
16. J. Gustafson, *IEEE Computational Science & Engineering* **5**, 36–45 (1998), ISSN 1070-9924, URL `http://www.computer.org/cse/cs1998/c1036abs.htm;http://dlib.computer.org/cs/books/cs1998/pdf/c1036.pdf`, discusses recent progress in interval arithmetic and its relevance to error estimation in very large computations of the type envisioned for the U.S. ASCI (Advanced Strategic Computing Initiative) project.
17. IEEE Task P754, *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*, IEEE, New York, NY, USA, 1985, ISBN 1-55937-653-8, URL `http://standards.ieee.org/reading/ieee/std_public/description/busarch/754-1985_desc.html;http://standards.ieee.org/reading/ieee/std/busarch/754-1985.pdf;http://www.iec.ch/cgi-bin/procgi.pl/www/iecwww.p?wwwlang=E&wwwprog=cat-det.p&wartnum=019113;http://ieeexplore.ieee.org/iel1/2355/1316/00030711.pdf`.
18. ANSI/IEEE, *ANSI/IEEE Std 854-1987: An American National Standard: IEEE Standard for Radix-Independent Floating-Point Arithmetic*, IEEE, New York, NY, USA, 1987, ISBN 0-7381-1167-8, URL `http://ieeexplore.ieee.org/xpl/standardstoc.jsp?isnumber=1121&isYear=1987;http://ieeexplore.ieee.org/iel1/2502/1121/00027840.pdf`, revised 1994. INSPEC Accession Number: 3095617.
19. W. Kahan, and J. D. Darcy, How Java's floating-point hurts everyone everywhere, Technical report, Department of Mathematics and Department of Electrical Engineering and Computer Science, University of California, Berkeley, Berkeley, CA, USA (1998), URL `http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf;http://www.cs.berkeley.edu/~wkahan/JAVAhurt.ps`.
20. Y. Nievergelt, *ACM Transactions on Mathematical Software* **29**, 27–48 (2003), ISSN 0098-3500, URL `http://doi.acm.org/10.1145/641876.641878`.
21. P. Markstein, *IA-64 and elementary functions: speed and precision*, Hewlett-Packard professional books, Prentice-Hall, Upper Saddle River, NJ 07458, USA, 2000, ISBN 0-13-018348-2, URL `http://www.markstein.org/`.
22. M. F. Cowlishaw, *The REXX language: a practical approach to programming*, Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1985, ISBN 0-13-780735-X (paperback).
23. M. F. Cowlishaw, *The NetRexx language*, Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1997, ISBN 0-13-806332-X, see also supplement [42].
24. *Preliminary Decimal-Floating-Point Architecture*, IBM Corporation, San Jose, CA, USA (2006), URL `http://publibz.boulder.ibm.com/epubs/pdf/a2322320.pdf;http://www-03.ibm.com/servers/eserver/zseries/zos/bkserv/r3pdf/zarchpops.html`, form number SA23-2232-00.
25. A. Y. Duale, M. H. Decker, H.-G. Zipperer, M. Aharoni, and T. J. Bohizic, *IBM Journal of Research and Development* **51**, 217–227 (2007), ISSN 0018-8646, URL `http://www.research.ibm.com/journal/rd/511/duale.html`.
26. M. F. Cowlishaw, "Decimal floating-point: algorism for computers," in *16th IEEE Symposium on Computer Arithmetic: ARITH-16 2003: proceedings: Santiago de Compostela, Spain, June 15–18, 2003*, edited by J. C. Bajard, and M. Schulte, IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2003, pp. 104–111, ISBN 0-7695-1894-X, ISSN 1063-6889, URL `http://www.dec.usc.es/arith16/papers/paper-107.pdf`.
27. R. Klarer, Decimal types for C++: Second draft, Report C22/WG21/N1839 J16/05-0099, IBM Canada, Ltd., Toronto, ON, Canada (2005), URL `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1839.html`.
28. ISO, ISO/IEC JTC1 SC22 WG14 N1154: Extension for the programming language C to support decimal floating-point arithmetic, World-Wide Web document (2006), URL `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1154.pdf`.
29. ISO, ISO/IEC JTC1 SC22 WG14 N1161: Rationale for TR 24732: Extension to the programming language C: Decimal floating-point arithmetic, World-Wide Web document (2006), URL `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1161.pdf`.
30. ISO, ISO/IEC JTC1 SC22 WG14 N1176: Extension for the programming language c to support decimal floating-point arithmetic, World-Wide Web document (2006), URL `http://open-std.org/jtc1/sc22/wg14/www/docs/n1176.pdf`.
31. M. Cowlishaw, *The decNumber C library*, IBM Corporation, San Jose, CA, USA (2007), URL `http://download.icu-project.org/ex/files/decNumber/decNumber-icu-340.zip`, version 3.40.
32. International Organization for Standardization, *ISO/IEC 1989:2002: Information technology — Programming languages — COBOL*, International Organization for Standardization, Geneva, Switzerland, 2002, ISBN ????, URL `http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=28805`.
33. M. F. Cowlishaw, *IEE Proceedings. Computers and Digital Techniques* **149**, 102–104 (2002), ISSN 1350-2387.
34. Anonymous, Reference software implementation of the IEEE 754R decimal floating-point arithmetic, World-Wide Web document (2006), URL `http://cache-www.intel.com/cd/00/00/29/43/294339_294339.pdf`.
35. N. H. F. Beebe, *The* `mathcw` *Portable Elementary Function Library*, 2008, in preparation.
36. M. Overton, *Numerical Computing with IEEE Floating Point Arithmetic, Including One Theorem, One Rule of Thumb, and One Hundred and One Exercises*, Society for Industrial and Applied Mathematics, Philadelphia,

PA, USA, 2001, ISBN 0-89871-482-6, URL http://www.siam.org/catalog/mcc07/ot76.htm,http://www.cs.nyu.edu/cs/faculty/overton/book/.

37. E. E. Swartzlander, Jr., *Computer design development: principal papers*, Hayden Book Co., Rochelle Park, NJ, USA, 1976, ISBN 0-8104-5988-4.
38. W. Aspray, and A. Burks, editors, *Papers of John von Neumann on computing and computer theory*, vol. 12 of *Charles Babbage Institute reprint series for the history of computing*, The MIT Press, Cambridge, MA, 1987, ISBN 0-262-22030-X.
39. Anonymous, *Annals of the History of Computing* **2**, 274–280 (1980), ISSN 0164-1239, URL http://dlib.computer.org/an/books/an1980/pdf/a3274.pdf;http://www.computer.org/annals/an1980/a3274abs.htm, see [2].
40. W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our business: a birthday salute to Edsger W. Dijkstra*, Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1990, ISBN 0-387-97299-4.
41. W. D. Clinger, *ACM SIGPLAN Notices* **39**, 360–371 (2004), ISSN 0362-1340, best of PLDI 1979–1999. Reprint of, and retrospective on, [7].
42. M. F. Cowlishaw, *NetRexx Language Supplement*, IBM UK Laboratories, Hursley Park, Winchester, England (2000), URL http://www-306.ibm.com/software/awdtools/netrexx/nrlsupp.pdf, version 2.00. This document is a supplement to [23].