Decimal Floating-Point: Algorism for Computers

Michael F. Cowlishaw
IBM UK Ltd., P.O. Box 31, Birmingham Road, Warwick CV34 5JL, UK
or Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK
mfc@uk.ibm.com

Abstract

Decimal arithmetic is the norm in human calculations, and human-centric applications must use a decimal floating-point arithmetic to achieve the same results.

Initial benchmarks indicate that some applications spend 50% to 90% of their time in decimal processing, because software decimal arithmetic suffers a 100× to 1000× performance penalty over hardware. The need for decimal floating-point in hardware is urgent.

Existing designs, however, either fail to conform to modern standards or are incompatible with the established rules of decimal arithmetic. This paper introduces a new approach to decimal floating-point which not only provides the strict results which are necessary for commercial applications but also meets the constraints and requirements of the IEEE 854 standard.

A hardware implementation of this arithmetic is in development, and it is expected that this will significantly accelerate a wide variety of applications.

1. Introduction

Algorism, the decimal system of numeration, has been used in machines since the earliest days of computing. Mechanical computers mirrored the manual calculations of commerce and science, and were almost all decimal. Many early electronic computers, such as the ENIAC[1], also used decimal arithmetic (and sometimes even decimal addressing). Nevertheless, by 1961 the majority were binary, as shown by a survey of computer systems in the USA[2] which reported that "131 utilize a straight binary system internally, whereas 53 utilize the decimal system (primarily binary coded decimal)...". Today, few computing systems include decimal hardware.

The use of binary arithmetic in computing became ascendant after Burks, Goldstine, and von Neumann[3] highlighted simplicity as the primary advantage of binary hardware (and by implication, greater reliability due to the reduced number of components). They concluded that for a general-purpose computer, used as a scientific research tool, the use of binary was optimal. However, Bucholtz[4] pointed out later that they

> "did not consider the equally important data processing applications in which but few arithmetic steps are taken on large volumes of input-output data. If these data are expressed in a form different from that used in the arithmetic unit, the conversion time can be a major burden."

and suggested that the combination of binary addressing with decimal data arithmetic was more powerful, a conclusion echoed by many other authors (see, for example, Schmid[5]). Inevitably, this implied that computers needed at least two arithmetic units (one for binary address calculations and the other for decimal computations) and so, in general, there was a natural tendency to economize and simplify by providing only binary arithmetic units.

The remainder of this section explains why decimal arithmetic and hardware are still essential. In section 2, the variety of decimal datatypes in use is introduced, together with a description of the arithmetic used on these types, which increasingly needs floating-point. In section 3, earlier designs are summarized, with a discussion of why they have proved inadequate. Section 4 introduces the new design, which allows integer, fixed-exponent, and floating-point numbers to be manipulated efficiently in a single decimal arithmetic unit.

1.1. The need for decimal arithmetic

Despite the widespread use of binary arithmetic, decimal computation remains essential for many applications. Not only is it required whenever numbers are presented for human inspection, but it is also often a necessity when fractions are involved.

Decimal fractions (rational numbers whose denominator is a power of ten) are pervasive in human endeavours, yet most cannot be represented by binary fractions;

the value 0.1, for example, requires an infinitely recurring binary number. If a binary approximation is used instead of an exact decimal fraction, results can be incorrect even if subsequent arithmetic is exact.

For example, consider a calculation involving a 5% sales tax on an item (such as a $0.70 telephone call), rounded to the nearest cent. Using double-precision binary floating-point, the result of multiplying 0.70 by 1.05 is a little under 0.73499999999999999 whereas a calculation using decimal fractions would yield exactly 0.735. The latter would be rounded up to $0.74, but using the binary fraction the result returned would be the incorrect $0.73.

For this reason, financial calculations (or, indeed, any calculations where the results achieved are required to match those which might be calculated by hand), are carried out using decimal arithmetic.

Further, numbers in commercial databases are predominately decimal. During a survey of commercial databases (the survey reported by Tsang[6]) the column datatypes of databases owned by 51 major organizations were analyzed. These databases covered a wide range of applications, including airline systems, banking, financial analysis, insurance, inventory control, management reporting, marketing services, order entry and processing, pharmaceuticals, and retail sales. In these databases, over 456,420 columns contained identifiably numeric data, and of these 55% were decimal (the SQL NUMERIC type[7]). A further 43.7% were integer types which could have been stored as decimals[8].

This extensive use of decimal data suggested that it would be worthwhile to study how the data are used and how decimal arithmetic should be defined. These investigations showed that the nature of commercial computation has changed so that decimal floating-point arithmetic is now an advantage for many applications.

It also became apparent that the increasing use of decimal floating-point, both in programming languages and in application libraries, brought into question any assumption that decimal arithmetic is an insignificant part of commercial workloads.

Simple changes to existing benchmarks (which used incorrect binary approximations for financial computations) indicated that many applications, such as a typical Internet-based ‘warehouse’ application, may be spending 50% or more of their processing time in decimal arithmetic. Further, a new benchmark, designed to model an extreme case (a telephone company’s daily billing application), shows that the decimal processing overhead could reach over 90%[9].

These applications are severely compute-bound rather than I/O-bound, and would clearly benefit from decimal floating-point hardware. Such hardware could be two to three orders of magnitude faster than software.

The rest of this paper discusses the requirements for decimal arithmetic and then introduces a proposed design for floating-point decimal arithmetic. This design is unique in that it is based on the strongly typed decimal representation and arithmetic required for commercial and financial applications, yet also meets the constraints and requirements of the IEEE 854 standard[10]. The design has been implemented as a C library in software, and a hardware implementation is in development.

2. Decimal arithmetic in practice

In early computers decimal floating-point in hardware was unstandardized and relatively rare. As a result, programming languages with decimal types almost invariably describe a decimal number as an integer which is scaled (divided) by a power of ten (in other words, effectively encoding decimal values as rational numbers). The number 2.50, for example, is held as the integer 250 with a scale of 2; the scale is therefore simply a negative exponent.

Depending on the language, the scale might be fixed (as in Ada fixed point[11], PL/I fixed decimal[12], or SQL NUMERIC) or it might be variable, hence providing a simple floating-point type (as in COBOL numerics[13], Rexx strings[14], Java BigDecimal[15], Visual Basic currency[16], and C# decimal[17]). Whether fixed or floating, this scaled approach reflects common ways of working with numbers on paper, especially in school, in commerce, and in engineering, and is both effective and convenient.

For many applications, a floating scale is especially advantageous. For example, European regulations[18,19], dictate that exchange rates must be quoted to 6 digits instead of to a particular scale. All the digits must be present, even if some trailing fractional digits are zero.

Preserving the scale associated with a number is also important in engineering and other applications. Here, the original units of a measurement are often indicated by means of the number of digits recorded, and the scale is therefore part of the datatype of a number.

If the scale is not preserved, measurements and contracts may appear to be more vague (less precise) than intended, and information is lost. For example, the length of a beam might be specified as 1.200 meters; if this value is altered to 1.2 meters then a contractor could be entitled to provide a beam that is within 5 centimeters of that length, rather than measured to the nearest millimeter. Similarly, if the scale is altered by computation, it must be possible to detect that change (even if the value of the result is exact) so that incorrect changes to algorithms can be readily detected.

For these and other reasons, the scaled integer representation of decimal data is pervasive. It is used in all commercial databases (where the scale is often an attribute of a column) as well as in programming. The integer coefficient is encoded in various forms (including binary, binary coded decimal (BCD), and base 100), and the scale is usually encoded in binary.

2.1. Arithmetic on decimal numbers

Traditionally, calculation with decimal numbers has used exact arithmetic, where the addition of two numbers uses the largest scale necessary, and multiplication results in a number whose scale is the sum of the scales of the operands (1.25 × 3.42 gives 4.2750, for example).

However, as applications and commercial software products have become increasingly complex, simple rational arithmetic of this kind has become inadequate. Repeated multiplications require increasingly long scaled integers, often dramatically slowing calculations as they soon exceed the limits of any available binary or decimal integer hardware.

Further, even financial calculations need to deal with an increasingly wide range of values. For example, telephone calls are now often costed in seconds rather than minutes, with rates and taxes specified to six or more fractional digits and applied to prices quoted in cents. Interest rates are now commonly compounded daily, rather than quarterly, with a similar requirement for values which are both small and exact. And, at the other end of the range, the Gross National Product of a country such as the USA (in cents) or Japan (in Yen) needs 15 digits to the left of the decimal point.

The manual tracking of scale over such wide ranges is difficult, tedious, and very error-prone. The obvious solution to this is to use a floating-point arithmetic.

The use of floating-point may seem to contradict the requirements for exact results and preservation of scales in commercial arithmetic; floating-point is perceived as being approximate, and normalization loses scale information.

However, if unnormalized floating-point is used with sufficient precision to ensure that rounding does not occur during simple calculations, then exact scale-preserving (type-preserving) arithmetic is possible, and the performance and other overheads of normalization are avoided. Rounding occurs in the usual manner when divisions or other complex operations are carried out, or when a specific rounding operation (for example, rounding to a given scale or precision) is applied.

This eclectic approach especially benefits the very common operations of addition or subtraction of numbers which have the same scale. In this case, no alignment is necessary, so these operations continue to be simple decimal integer addition or subtraction, with consequent performance advantages. For example, 2.50 can be stored as 250×10^{-2}, allowing immediate addition to 12.25 (stored as 1225×10^{-2}) without requiring shifting. Similarly, after adding 1.23 to 1.27, no normalization shift to remove the ‘extra’ 0 is needed.

The many advantages of scaled-integer decimal floating-point arithmetic have led to its being widely adopted in programming languages (including COBOL, Basic, Rexx, Java, and C#) and in many other software libraries. However, these implementations have not in the past provided the full floating-point arithmetic facilities which are now the norm in binary floating-point libraries and hardware.

To see how these can be supported too, it is helpful to consider some earlier decimal floating-point designs.

3. Previous decimal floating-point designs

Decimal floating-point (DFP) arithmetics have been proposed, and often implemented, for both hardware and software. A particular dichotomy of these designs is the manner by which the coefficient (sometimes called the mantissa or significand) is represented. Designs derived from scaled arithmetic use an integer for this, whereas those built for mathematical or scientific use generally use a normalized fraction.

Both of these approaches appear in the following designs. (Here, the notation {p,e} gives the maximum precision and exponent range where known.)

- In 1955, Perkins[20] described the EASIAC, a wholly DFP virtual computer implemented on the University of Michigan’s MIDAC machine {7+, ±20}. The form of the coefficient seems to have been a fraction.
- The Gamma 60 computer[21], first shipped by Bull in 1960, had a DFP calculation unit with fractional coefficients {11–19, ±40}.
- In 1962, Jones and Wymore[22] gave details of the normalized variable-precision DFP Feature on the IBM Type 1620 computer {100, ±99}.
- The Burroughs B5500 computer[23], shipped in 1964, used an integer or fixed-point coefficient {21–22, ±63}. Neely[24] noted later that this “permits mixed integer and real arithmetic without type conversion”.
- Mazor, in 1966[25], designed the Fairchild Symbol II DFP unit. This used normalized variable-precision and most-significant-digit–first processing {99, ±99?}.
- In 1969, Duke[26] disclosed a hardware design using unnormalized binary integers of unspecified length for both coefficient and exponent.
- Also in 1969, Taub, Owen, and Day[27] built the IBM

Schools computer; this experimental teaching computer used a scaled integer format {6, −6 to 0}.

- Fenwick, in 1972[28], described a representation similar to Duke's, and highlighted the advantages of unnormalized DFP.
- By the early 1970s, the mathematical requirements for decimal floating-point were becoming understood. Ris, working with Gustavson, Kahan, and others, proposed a unified DFP architecture[29] which had many of the features of the later binary floating-point standard. It was a normalized DFP with a fractional coefficient, three directed rounding modes, and a trap mechanism. Three precisions were defined, up to {31, ±9999}. Ris's representations were the first to use the encoding invented by Chen and Ho[30], which allowed the 31-digit DFP numbers, with 4-digit exponent, to be represented in 128 bits.
- In 1975, Keir[31] described the advantages of an unnormalized integer coefficient, noting that it is "exactly as accurate as normalized arithmetic".
- Hull, in 1978[32], proposed a DFP with controlled (variable) precision and a fractional coefficient. This was later refined and implemented in the controlled-precision CADAC arithmetic unit by Cohen, Hull, and Hamacher[33,34].
- In 1979, Johannes, Pegden, and Petry[35] discussed the problems of efficient decimal shifts in a two-binary-integer DFP representation (problems which were revisited by Bohlender in 1991[36]).
- In 1981, Cowlishaw added arbitrary-precision DFP {10^9, ±10^9;} to the Rexx programming language[37]; this was unnormalized. Later implementations of the Rexx family of languages have used a variety of representations, all with integer coefficients.
- In 1982, Sacks-Davis[38] showed that the advantage of redundant number representations (addition time is independent of operand length) can be applied to DFP, though no implementation of this is known.
- Also in the early 1980s, the standardization of binary floating-point arithmetic was completed, with the publication of the IEEE 754 standard[39]. A later generalization of that standard, IEEE 854, extended the principles to DFP, as explained by Cody *et al*[40].

  These standards formalized earlier work, recorded by Kahan[41], and, unlike earlier designs using integer coefficients, prescribe *gradual underflow* as well as infinities and NaNs.

  Compliance with IEEE 854 does not require either normalization or fractional coefficients. Values may be encoded redundantly, as in the Burroughs B5500, and hence may use integer coefficients.

  IEEE 854 was first implemented in the HP 71B calculator[42,43]; this held numbers as {12, ±499} fractions, expanded to {15, ±49999} for calculations.
- In 1987, Bohlender and Teufel[44] described a bit-slice DFP unit built for PASCAL-SC (a version of Pascal designed for scientific computation). This used a BCD fractional coefficient {13, −98 to +100}.
- *Circa* 1990, Visual Basic added a floating-point currency class with a 64-bit binary integer coefficient. This was later extended to 96 bits and formed the basis of the C# decimal type {28–29, −28 to 0}.
- In 1996, Java added the exact arithmetic BigDecimal class. This is arbitrary-precision, with a binary integer coefficient.
- Finally, hand calculators almost all use decimal floating-point arithmetic. For example, the Texas Instruments TI-89[45,46] uses a BCD fractional coefficient {14, ±999}, Hewlett Packard calculators continue to use a 12-digit decimal format, and Casio calculators have a 15-digit decimal internal format.

Looking back at these designs, it would seem that those which assumed a fractional coefficient were designed with mathematical rather than commercial uses in mind. These did not meet the strong type requirements of scaled decimal arithmetic in commercial applications, and for other applications they were eclipsed by binary floating-point, which provides better performance and accuracy for a given investment in hardware. (The notable exception to this generalization is the hand-held calculator, where performance is rarely an issue and decimal floating-point is common.)

Those designs which use integer coefficients, however, have survived, and are widely used in software, probably due to their affinity with decimal data storage and exact rational arithmetic. These designs tend to follow the traditional rules of decimal arithmetic, and although this is in effect floating-point, little attempt was made to incorporate the improvements and advantages of the floating-point system defined in IEEE 854 until recently.

At first reading it may seem that the rules of IEEE 854, which appear to assume a fractional coefficient, must be incompatible with unnormalized scaled-integer arithmetic. However, a fractional coefficient is not necessary to meet the requirements of IEEE 854. All the mathematical constraints of the standard can be met when using an integer coefficient; indeed, it turns out that subnormal values are simpler to handle in this form.

Normalization, too, is not required by IEEE 854. It does offer an advantage in binary floating-point, where in effect it is a compression scheme that increases the length of the coefficient by one bit. However, in decimal arithmetic (where in any case only one value in ten could be implied in the same way) normalized arithmetic is harmful, because it precludes the exact and strongly typed arithmetic with scale preservation which is essential for many applications.

With these observations, it becomes possible to extend the relatively simple software DFP in common use today to form a richer arithmetic which not only meets the commercial and financial arithmetic requirements but also has the advantages of the IEEE 854 design: a formally closed arithmetic with gradual underflow and defined exception handling.

4. The decimal arithmetic design

There is insufficient space here to include every detail of the decimal floating-point design; this is available at `http://www2.hursley.ibm.com/decimal`. Also, the arithmetic is independent of specific concrete representations, so these are not discussed here.

In summary, the core of the design is the abstract model of finite numbers. In order to support the required exact arithmetic on decimal fractions, these comprise an integer coefficient together with a conventional sign and signed integer exponent (the exponent is the negative of the scale used in scaled-integer designs). The numerical *value* of a number is given by $(-1)^{sign} \times coefficient \times 10^{exponent}$.

It is important to note that even though the concept of scale is preserved in numbers (the numbers are essentially two-dimensional), they are not the *significant numbers* deprecated by Delury[47]. Each number has an exact value and, in addition, an exact exponent which indicates its type; the number may be thought of as the sum of an integral number of discrete values, each of magnitude $10^{exponent}$. The arithmetic on these numbers is exact (unless rounding to a given precision is necessary) and is in no sense a 'significance' arithmetic.

Given the parameters just described, much of the arithmetic is obvious from either the rules of mathematics or from the requirements of IEEE 854 (the latter, for example, define the processing of NaNs, infinities, and subnormal values). The remainder of this section explains some less obvious areas.

4.1. Context

In this design, the concept of a *context* for operations is explicit. This corresponds to the concept of a 'floating-point control register' in hardware or a context object instance in software.

This context includes the flags and trap-enablers from IEEE 854 §7 and the rounding modes from §4. There is also an extra rounding mode and a precision setting.

4.1.1. Commercial rounding. The extra rounding mode is called *round-half-up*, which is a requirement for many financial calculations (especially for tax purposes and in Europe). In this mode, if the digits discarded during rounding represent greater than or equal to half (0.5) of the value of a one in the next left position then the result should be rounded up. Otherwise the discarded digits are ignored. This is in contrast to *round-half-even*, the default IEEE 854 rounding mode, where if the discarded digits are exactly half of the next digit then the least significant digit of the result will be even.

It is also recommended that implementations offer two further rounding modes: *round-half-down* (where a 0.5 case is rounded down) and *round-up* (round away from zero). The rounding modes in IEEE 854 together with these three are the same set as those available in Java.

4.1.2. Precision. The *working precision* setting in the context is a positive integer which sets the maximum number of significant digits that can result from an arithmetic operation. It can be set to any value up to the maximum length of the coefficient, and lets the programmer choose the appropriate working precision.

In the case of software (which may well support unlimited precision), this lets the programmer set the precision and hence limit computation costs. For example, if a daily interest rate multiplier, R, is 1.000171 (0.0171%, or roughly 6.4% per annum), then the exact calculation of the yearly rate in a non-leap year is R^{365}. To calculate this to give an exact result needs 2191 digits, whereas a much shorter result which is correct to within one unit in the last place (ulp) will almost always be sufficient and could be calculated very much faster.

In the case of hardware, precision control has little effect on performance, but allows the hardware to be used for calculations of a different precision from the available 'natural' register size. For example, one proposal[48] for a concrete representation suggests a maximum coefficient length of 33 digits; this would be unsuitable for implementing the new COBOL standard (which specifies 32-digit intermediate results) if precision control in some form were not available.

Note that to conform to IEEE 854 §3.1 the working precision should not be set to less than 6.

4.2. Arithmetic rules

The rules for arithmetic are the traditional exact decimal rational arithmetic implemented in the languages and databases described earlier, subject to the context used for the operation. These rules can all be described in terms of primitive integer operations, and are defined in such a way that integer arithmetic is itself a subset of the full floating-point arithmetic. The lack of automatic normalization is essential for this to be the case.

It is the latter aspect of the design which permits both integer and floating-point arithmetic to be carried out in the same processing unit, with obvious economies in either a hardware or a software implementation.

The ability to handle integers as easily as fractions avoids conversions (such as when multiplying a cost by a number of units) and permits the scale (type) of numbers to be preserved when necessary. Also, since the coefficient is a 'right-aligned' integer, conversions to and from other integer representations (such as BCD or binary) are simplified.

To achieve the necessary results, every operation is carried out as though an infinitely precise mathematical result is first computed, using integer arithmetic on the coefficient where possible. This intermediate result is then coerced to the precision specified in the context, if necessary, using the rounding algorithm also specified in the context. Rounding, the processing of overflow and underflow conditions, and the production of subnormal results are defined in IEEE 854.

The following subsections describe the required operators (including some not defined in IEEE 854), and detail the rules by which their initial result (before any rounding) is calculated.

The notation {*sign*, *coefficient*, *exponent*} is used here for the numbers in examples. All three parameters are integers, with the third being a signed integer.

4.2.1. Addition and subtraction. If the exponents of the operands differ, then their coefficients are first aligned; the operand with the larger exponent has its original coefficient multiplied by 10^n, where n is the absolute difference between the exponents.

Integer addition or subtraction of the coefficients, taking signs into account, then gives the exact result coefficient. The result exponent is the minimum of the exponents of the operands.

For example, {0, 123, −1} + {0, 127, −1} gives {0, 250, −1}, as does {0, 50, −1} + {0, 2, +1}.

Note that in the common case where no alignment or rounding of the result is necessary, the calculations of coefficient and exponent are independent.

4.2.2. Multiplication. Multiplication is the simplest operation to describe; the coefficients of the operands are multiplied together to give the exact result coefficient, and the exponents are added together to give the result exponent.

For example, {0, 25, 3} × {0, 2, 1} gives {0, 50, 4}.

Again, the calculations of coefficient and exponent are independent unless rounding is necessary.

4.2.3. Division. The rules for division are more complex, and some languages normalize all division results. This design, however, uses exact integer division where possible, as in C#, Visual Basic, and Java. Here, a number such as {0, 240, −2} when divided by two becomes {0, 120, −2} (not {0, 12, −1}).

The precision of the result will be no more than that necessary for the exact result of division of the integer coefficient. For example, if the working precision is 9 then {0, 241, −2} ÷ 2 gives {0, 1205, −3} and {0, 241, −2} ÷ 3 gives, after rounding, {0, 803333333, −9}.

This approach gives integer or same-scale results where possible, while allowing post-operation normalization for languages or applications which require it.

4.2.4. Comparison. A comparison compares the numerical values of the operands, and therefore does not distinguish between redundant encodings. For example, {1, 1200, −2} compares equal to {1, 12, 0}. The actual values of the coefficient or exponent can be determined by conversion to a string (or by some unspecified operation). For type checking, it is useful to provide a means for extracting the exponent.

4.2.5. Conversions. Conversions between the abstract form of decimal numbers and strings are more straightforward than with binary floating-point, as conversions can be exact in both directions.

In particular, a conversion from a number to a string and back to a number can be guaranteed to reproduce the original sign, coefficient, and exponent.

Note that (unless deliberately rounded) the length of the coefficient, and hence the exponent, of a number is preserved on conversion from a string to a number and vice versa. For example, the five-character string `"1.200"` will be converted to the number {0, 1200, −3}, not {0, 12, −1}.

One consequence of this is that when a number is displayed using the defined conversion there is no hidden information; "what you see is exactly what you have". Further, the defined conversion string is in fact a valid and complete concrete representation for decimal numbers in the arithmetic; it could be used directly in an interpreted scripting language, for example.

4.2.6. Other operations. The arithmetic defines a number of operations in addition to those already described. **abs**, **max**, **min**, **remainder-near**, **round-to-integer**, and **square-root** are the usual operations as defined in IEEE 854. Similarly, **minus** and **plus** are defined in order to simplify the mapping of the prefix **−** and prefix **+** operators present in most languages.

divide-integer and **remainder** are operators which provide the truncating remainder used for integers (and for floating-point in Java, Rexx, and other languages). If the operands x and y are given to the divide-integer and remainder operations, resulting in i and r respectively, then the identity $x = (i \times y) + r$ holds.

An important operator, **rescale**, sets the exponent of a number and adjusts its coefficient (with rounding, if necessary) to maintain its value. For example, rescaling the number {0, 1234567, –4} so its exponent is –2 gives {0, 12346, –2}. This example is the familiar, and very heavily used, 'round to cents' operation, although rescale has many other uses (**round-to-integer** is a special case of rescale, for example).

Finally, the **normalize** operator is provided for reducing a number to its most succinct form. Unlike the binary equivalent, this normalization removes trailing rather than leading zeros from the coefficient; this means that it generalizes to arbitrary-precision implementations.

5. Conclusion

The new data type described here combines the advantages of algorism and modern floating-point arithmetic. The integer coefficient means that conversions to and from fixed-point data and character representations are fast and efficient. The lack of normalization allows strongly typed decimal numbers and improves the performance of the most common operations and conversions. The addition of the IEEE 854 subnormal and special values and other features means that full floating-point facilities are available on decimal numbers without costly and difficult conversions to and from binary floating-point. These performance and functional advantages are complemented by easier programming and the reduced risk of error due to the automation of scaling and other operations.

6. Acknowledgements

The author is indebted to Kittredge Cowlishaw, Roger Golliver, Michael Schulte, Eric Schwarz, and the anonymous referees for their many helpful suggestions for improvements to this paper.

References

[1] H. H. Goldstine and Adele Goldstine, "The Electronic Numerical Integrator and Computer (ENIAC)", *IEEE Annals of the History of Computing, Vol. 18 #1*, pp10–16, IEEE, 1996.

[2] Martin H. Weik, "A Third Survey of Domestic Electronic Digital Computing Systems, Report No. 1115", 1131pp, Ballistic Research Laboratories, Aberdeen Proving Ground, Maryland, March 1961.

[3] Arthur W. Burks, Herman H. Goldstine, and John von Neumann, "Preliminary discussion of the logical design of an electronic computing instrument", 42pp, Inst. for Advanced Study, Princeton, N. J., June 28, 1946.

[4] Werner Buchholz, "Fingers or Fists? (The Choice of Decimal or Binary representation)", *Communications of the ACM, Vol 2 #12*, pp3–11, ACM, December 1959.

[5] Hermann Schmid, "Decimal Computation", ISBN 047176180X, 266pp, Wiley, 1974.

[6] Annie Tsang and Manfred Olschanowsky, "A Study of DataBase 2 Customer Queries", *IBM Technical Report TR 03.413*, 25pp, IBM Santa Teresa Laboratory, San Jose, CA, April 1991.

[7] Jim Melton *et al*, "ISO/IEC 9075:1992: Information Technology – Database Languages – SQL", 626pp, ISO, 1992.

[8] Akira Shibamiya, "Decimal arithmetic in applications and hardware", 2pp, *pers. comm.*, 14 June 2000.

[9] M. F. Cowlishaw, "The 'telco' benchmark", URL: `http://www2.hursley.ibm.com/decimal`, 3pp, IBM Hursley Laboratory, May 2002.

[10] W. J. Cody *et al*, "IEEE 854-1987 IEEE Standard for Radix-Independent Floating-Point Arithmetic", 14pp, IEEE, March 1987.

[11] S. Tucker Taft and Robert A. Duff, "ISO/IEC 8652:1995: Information Technology – Programming Languages – Ada (Ada 95 Reference Manual: Language and Standard Libraries)", ISBN 3-540-63144-5, 552pp, Springer-Verlag, July 1997.

[12] J. F. Auwaerter, "ANSI X3.53-l976: American National Standard – Programming Language PL/I", 421pp, ANSI, 1976.

[13] JTC-1/SC22/WG4, "Proposed Revision of ISO 1989:1985 Information technology – Programming languages, their environments and system software interfaces – Programming language COBOL", 905pp, INCITS, December 2001.

[14] Brian Marks and Neil Milsted, "ANSI X3.274-1996: American National Standard for Information Technology – Programming Language REXX", 167pp, ANSI, February 1996.

[15] Sun Microsystems, "BigDecimal (Java 2 Platform SE v1.4.0)", URL: `http://java.sun/com/products`, 17pp, Sun Microsystems Inc., 2002.

[16] Microsoft Corporation, "MSDN Library Visual Basic 6.0 Reference", URL: `msdn.microsoft.com/library`, Microsoft Corporation, 2002.

[17] Rex Jaeschke, "C# Language Specification", *ECMA-TC39-TG2-2001*, 520pp, ECMA, September 2001.

[18] European Commission, "The Introduction of the Euro and the Rounding of Currency Amounts", 29pp, European Commission Directorate General II Economic and Financial Affairs, 1997.

[19] European Commission Directorate General II, "The Introduction of the Euro and the Rounding of Currency Amounts", *II/28/99-EN Euro Papers No. 22.*, 32pp, DGII/C-4-SP(99) European Commission, March 1998, February 1999.

[20] Robert Perkins, "EASIAC, A Pseudo-Computer", *Communications of the ACM, Vol. 3 #2*, pp65–72, ACM, April 1956.

[21] M. Bataille, "The Gamma 60: The computer that was ahead of its time", *Honeywell Computer Journal Vol 5 #3*, pp99–105, Honeywell, 1971.

[22] F. B. Jones and A. W. Wymore, "Floating Point Feature On The IBM Type 1620", *IBM Technical Disclosure Bulletin, 05-62*, pp43–46, IBM, May 1962.

[23] Burroughs Corporation, "Burroughs B5500 Information Processing Systems Reference Manual", 224pp, Burroughs Corporation, Detroit, Michigan, 1964.

[24] Peter M. Neely, "On conventions for systems of numerical representation", *Proceedings of the ACM annual conference, Boston, Massachusetts*, pp644–651, ACM, 1972.

[25] Stan Mazor, "Fairchild decimal arithmetic unit", 9pp, *pers. comm.*, July–September 2002.

[26] K. A. Duke, "Decimal Floating Point Processor", *IBM Technical Disclosure Bulletin, 11-69*, pp862–862, IBM, November 1969.

[27] D. M. Taub, C. E. Owen, and B. P.. Day, "Experimental Computer for Schools", *Proceedings of the IEE, Vol 117 #2*, pp303–312, IEE, February 1970.

[28] Peter M. Fenwick, "A Binary Representation for Decimal Numbers", *Australian Computer Journal, Vol 4 #4 (now Journal of Research and Practice in Information Technology)*, pp146–149, Australian Computer Society Inc., November 1972.

[29] Frederic N. Ris, "A Unified Decimal Floating-Point Architecture for the Support of High-Level Languages", *ACM SIGNUM Newsletter, Vol. 11 #3*, pp18–23, ACM, October 1976.

[30] Tien Chi Chen and Irving T. Ho, "Storage-Efficient Representation of Decimal Data", *CACM Vol 18 #2*, pp49–52, ACM, January 1975.

[31] R. A. Keir, "Compatible number representations", *Conf. Rec. 3rd Symp. Comp. Arithmetic CH1017-3C*, pp82–87, IEEE Computer Society, 1975.

[32] T. E. Hull, "Desirable Floating-Point Arithmetic and Elementary Functions for Numerical Computation", *ACM Signum Newsletter, Vol. 14 #1 (Proceedings of the SIGNUM Conference on the Programming Environment for Development of Numerical Software)*, pp96–99, ACM, 1978.

[33] Marty S. Cohen, T. E. Hull, and V. Carl Hamacher, "CADAC: A Controlled-Precision Decimal Arithmetic Unit", *IEEE Transactions on Computers, Vol. 32 #4*, pp370–377, IEEE, April 1983.

[34] T. E. Hull and M. S. Cohen, "Toward an Ideal Computer Arithmetic", *Proceedings of the 8th Symposium on Computer Arithmetic*, pp131–138, IEEE, May 1987.

[35] J. D. Johannes, C. Dennis Pegden, and F. E. Petry, "Decimal Shifting for an Exact Floating Point Representation", *Computers and Electrical Engineering, Vol. 7 #3*, pp149–155, Elsevier, September 1980.

[36] Gerd Bohlender, "Decimal Floating-Point Arithmetic in Binary Representation", *Computer arithmetic: Scientific Computation and Mathematical Modelling (Proceedings of the Second International Conference, Albena, Bulgaria, 24-28 September 1990)*, pp13–27, J. C. Baltzer AG, 1991.

[37] M. F. Cowlishaw, "The Design of the REXX Language", *IBM Systems Journal, Vol 23 #4*, pp326–335, IBM (Offprint # G321-5228), 1984.

[38] R. Sacks-Davis, "Applications of Redundant Number Representations to Decimal Arithmetic", *The Computer Journal, Vol 25 #4*, pp471–477, November 1982.

[39] David Stevenson *et al*, "IEEE 754-1985 IEEE Standard for Binary Floating-Point Arithmetic", 20pp, IEEE, July 1985.

[40] W. J. Cody *et al*, "A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic", *IEEE Micro magazine*, pp86–100, IEEE, August 1984.

[41] W. Kahan, "Mathematics Written in Sand", *Proc. Joint Statistical Mtg. of the American Statistical Association*, pp12–26, American Statistical Association, 1983.

[42] Hewlett Packard Company, "Math Reference", *HP-71 Reference Manual, Mfg. #0071-90110, Reorder #0071-90010*, pp317–318, Hewlett Packard Company, October 1987.

[43] Hewlett Packard Company, "Chapter 13 – Internal Data Representations", *Software Internal Design Specification for the HP-71, Vol. 1 Part #00071-90068*, pp13.1–13.17, Hewlett Packard Company, December 1983.

[44] G. Bohlender and T. Teufel, "A Decimal Floating-Point Processor for Optimal Arithmetic", *Computer arithmetic: Scientific Computation and Programming Languages*, ISBN 3-519-02448-9, pp31–58, B. G. Teubner Stuttgart, 1987.

[45] Texas Instruments, "TI-89/TI-92 Plus Developers Guide, Beta Version .02", 1356pp, Texas Instruments, 2001.

[46] Texas Instruments, "TI-89/TI-92 Plus Sierra C Assembler Reference Manual, Beta Version .02", 322pp, Texas Instruments, 2001.

[47] Daniel B. Delury, "Computation with Approximate Numbers", *The Mathematics Teacher 51*, pp521–530, November 1958.

[48] Michael F. Cowlishaw, Eric M. Schwarz, Ronald M. Smith, and Charles F. Webb, "A Decimal Floating-Point Specification", *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, ISBN 0-7695-1150-3, pp147–154, IEEE, June 2001.