Science and Technology Publishing (SCI & TECH)
ISSN: 2632-1017
Vol. 5 Issue 8, August - 2021

Analysis Of Decimal Floating-Point Algorithms For Arithmetic Logic Unit

Subhash Kumar Sharma[1] Shri Prakash Dubey[2] Anil Kumar Mishra[3]
Department Of Electronics Department Of Physics Department Of Physics
MGPG College, Gorakhpur MGPG College, Gorakhpur MGPG College, Gorakhpur
273001,UP India 273001,UP India 273001 ,UP India
Email: sksharma13@yahoo.co.uk

Abstract: 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. Keywords: Arithmetic on Decimal Numbers & Design, Rounding, Precision, Addition, Subtraction, Multiplication, Division, Compare, Radix Conversion.

1. Introduction: Many early electronic computers, such as "The Electronic Numerical Integrator and Computer" (ENIAC), also used decimal arithmetic (and sometimes even decimal addressing). 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. The variety of decimal data types in use is introduced, together with a description of the arithmetic used on these types, which increasingly needs floating-point.

2.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 within an increasingly wide range of values. 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. 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.

3.The Decimal Arithmetic Design: 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* × *coefficient* × $10^{exponent}$.

3.1 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).

3.2 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. 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.

3.3 Arithmetic Rules: The design which permits both integer and floating-point arithmetic to be carried out in the same processing unit, with obvious economies in either 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. 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.

3.4 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.

3.5 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.

3.6 Division: The rules for division are more complex, and some languages normalize all division results. 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.

3.7 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.

3.8 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}.

3.9 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. 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}.

4.0 Radix Conversion Algorithms:

In this section, we present two very fast radix conversion algorithms that do not always return a 1 Our study is easily generalizable to directed roundings: we focus on round-to-nearest for the sake of brevity. correctly-rounded result. These algorithms require the availability of a fused multiply-add (fma) instruction in binary FP arithmetic. Their accuracy will suffice for our purpose (implementing decimal functions using the binary ones), but they cannot be directly used for implementing the (correctly rounded) radix conversions specified by the IEEE 754-2008 standard for FP arithmetic. And yet, we can fairly easily precompute the very few input values for which these algorithms do not provide correctly-rounded conversions, and use this information to design variants that always return correctly rounded results. Early works on radix conversion were done by Goldberg [17] and by Matula [26]. At that time, it was assumed that the processors' arithmetic was binary, and that the user wanted to enter and read data in decimal. Assuming a radix-2 underlying arithmetic and

a radix-10 user interface, algorithms for input and output radix conversion can be found in the literature [11]–[13], [29], [30]. The IEEE 754–2008 standard [19] specifies two encoding systems for decimal floating-point arithmetic, called the decimal and binary encodings. The reason for that is that the binary encoding makes a software implementation of decimal arithmetic easier, whereas the decimal encoding is more suited for a hardware implementation. The set of representable floating-point numbers is the same for both encoding systems, so that this additional complexity is transparent for most users. We focus here on the binary encoding. In that encoding, the exponent as well as 3 to 4 leading bits of the significand are stored in a “combination field”, and the remaining significand bits are stored in a “trailing significand field”. We can easily assume here (packing to and unpacking from the combination and trailing significand fields is simple) that a decimal number x10 is represented by an exponent e10 and an integral significand X10, |X10| ≤ 10p10 − 1 such that x10 = X10 · 10e10−p10+1 . From this, one can easily deduce that converting from decimal to binary essentially consists in performing, in binary arithmetic, the multiplication X10 × 10e10−p10+1 , where X10 is already available in binary, and the binary representation of 10e10−p10+1 (or, merely, a suitable approximation to that number) is precomputed and stored in memory. Conversion from binary to decimal will essentially consist in performing a multiplication by the inverse constant (or, merely, a suitable approximation to it), with some additional difficulty linked with decimal exponent guess and rounding. In a very comprehensive study [14], Cornea et al. give constraints on the accuracy of the approximation to the powers of ten used in conversions, suggest ways of performing decimal roundings, and give algorithms for implementing decimal arithmetic in software, assuming the ensl-00463353, version 1 - 11 Mar 2010 binary encoding is used. Our goal is to implement conversions using, for performing the multiplications by the factors 10e10−p10+1, a very fast FP multiply-by-a-constant algorithm suggested by Brisebarre and Muller [10], and then to use these fast conversions for implementing functions in decimal arithmetic using already existing binary functions.

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.

ACKNOWLEDGEMENT: Every success stands as a testimony not only to the hardship but also to hearts behind it. Likewise, the present work has been undertaken and completed with direct and indirect help from many people, my friends ,My wife ,My elder daughter Dr.Vaishnavi Sharma (MBBS) and I would like to acknowledge all of them for the same.

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].Hermann Schmid, “Decimal Computation”, ISBN 047176180X, 266pp, Wiley, 1974.

[4].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.

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

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

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

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

[9].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.

[10].N. Brisebarre and J.-M. Muller. Correctly rounded multiplication by arbitrary precision constants. IEEE Transactions on Computers, 57(2):165–174, February 2008.

[11]. R. G. Burger and R. Kent Dybvig. Printing floating point numbers quickly and accurately. In Proceedings of the SIGPLAN’96 Conference on Programming Languages Design and Implementation, pages 108–116, June 1996.

[12].W. D. Clinger. How to read floating-point numbers accurately. ACM SIGPLAN Notices, 25(6):92–101, June 1990. ensl-00463353, version 1 - 11 Mar 2010

[13]. W. D. Clinger. Retrospective: how to read floating-point numbers accurately. ACM SIGPLAN Notices, 39(4):360– 371, April 2004.

[14]. M. Cornea, J. Harrison, C. Anderson, P. T. P. Tang, E. Schneider, and E. Gvozdev. A software implementation of the IEEE 754R decimal floating-point arithmetic using the binary encoding format. IEEE Transactions on Computers, 58(2):148–162, 2009.

[15]. M. Daumas, C. Mazenc, X. Merrheim, and J.-M. Muller. Modular range reduction: A new algorithm for fast and accurate computation of the elementary functions. Journal of Universal Computer Science, 1(3):162–175, March 1995.

[16]. L. Fousse, G. Hanrot, V. Lefevre, P. P ` elissier, and P. Zim- ´ mermann. MPFR: A Multiple-Precision Binary FloatingPoint Library with Correct Rounding. ACM Transactions on Mathematical Software, 33(2), 2007. available at http: //www.mpfr.org/.

[17]. I. B. Goldberg. 27 bits are not enough for 8-digit accuracy. Commun. ACM, 10(2):105–106, 1967.

[18]. J. Harrison. Decimal transcendentals via binary. In Proceedings of the 19th IEEE Symposium on Computer Arithmetic (ARITH-19). IEEE Computer Society Press, June 2009.

[19].IEEE Computer Society. IEEE Standard for FloatingPoint Arithmetic. IEEE Standard 754-2008, August 2008. available at http://ieeexplore.ieee.org/servlet/opac?punumber=4610933.

[20]. Francisco J. Jaime, Julio Villalba, Javier Hormigo, and Emilio L. Zapata. Pipelined architecture for additive range reduction. J. Signal Process. Syst., 53(1-2):103–112, 2008.

[21].V. Lefevre and J.-M. Muller. Worst cases for correct ` rounding of the elementary functions in double precision. In N. Burgess and L. Ciminiera, editors, Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH16), Vail, CO, June 2001.

[22]. V. Lefevre, J.-M. Muller, and A. Tisserand. Towards cor- ` rectly rounded transcendentals. In Proceedings of the 13th IEEE Symposium on Computer Arithmetic. IEEE Computer Society Press, Los Alamitos, CA, 1997.

[23].V. Lefevre, D. Stehl ` e, and P. Zimmermann. Worst cases for ´ the exponential function in the IEEE 754r decimal64 format. In Reliable Implementation of Real Number Algorithms: Theory and Practice, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008.

[24].P. Markstein. IA-64 and Elementary Functions: Speed and Precision. Hewlett-Packard Professional Books. PrenticeHall, Englewood Cliffs, NJ, 2000.

[25].Peter Markstein. The new IEEE-754 standard for floating point arithmetic. In Numerical Validation in Current Hardware Architectures, number 08021 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. Internationales Begegnungs- und Forschungszentrum fur Informatik (IBFI), ¨ Schloss Dagstuhl, Germany.

[26].D. W. Matula. In-and-out conversions. Communications of the ACM, 11(1):47–50, January 1968.

[27].J.-M. Muller. Elementary Functions, Algorithms and Implementation. Birkhauser Boston, MA, 2nd edition, 2006. ¨

[28]. Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, ` Guillaume Melquiond, Nathalie Revol, Damien Stehle,´ and Serge Torres. Handbook of Floating-Point Arithmetic . Birkhauser Boston, 2010. ACM G.1.0; G.1.2; G.4; B.2.0; ¨ B.2.4; F.2.1., ISBN 978-0-8176-4704-9.

[29].S. Rump. Solving algebraic problems with high accuracy (habilitationsschrift). In Kulisch and Miranker, editors, A New Approach to Scientific Computation, pages 51–120. Academic Press, New york, NY, 1983.

[30].G. L. Steele Jr. and J. L. White. Retrospective: how to print floating-point numbers accurately. ACM SIGPLAN Notices , 39(4):372–389, april 2004.

[31].Julio Villalba, Tomas Lang, and Mario A. Gonzalez. Double-residue modular range reduction for floatingpoint hardware implementations. IEEE Trans. Comput., 55(3):254–267, 2006.