

Implementing Decimal Floating-Point Arithmetic through Binary: some Suggestions

Nicolas Brisebarre, Nicolas Louvet, Érik Martin-Dorel,
Jean-Michel Muller, Adrien Panhaleux
CNRS, ENSL, INRIA, UCBL, Univ. Lyon, Lab. LIP (UMR 5668)
École Normale Supérieure de Lyon, LIP
46 allée d'Italie, 69364 Lyon Cedex 07, France
firstname.lastname@ens-lyon.fr

Miloš D. Ercegovac
4731H Boelter Hall
Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90024, USA
Email: milos@cs.ucla.edu

Abstract—We propose algorithms and provide some related results that make it possible to implement decimal floating-point arithmetic on a processor that does not have decimal operators, using the available binary floating-point functions. In this preliminary study, we focus on round-to-nearest mode only. We show that several functions in decimal32 and decimal64 arithmetic can be implemented using binary64 and binary128 floating-point arithmetic, respectively. We discuss the decimal square root and some transcendental functions. We also consider radix conversion algorithms.

Keywords—decimal floating-point arithmetic; square root; transcendental functions; radix conversion.

I. INTRODUCTION

The IEEE 754-2008 standard for floating-point (FP) arithmetic [8], [13] specifies decimal formats. Decimal arithmetic is mainly used in financial applications. This has the following implications:

- An implementation must be *correct* (especially, the arithmetic operations must round correctly), but on most platforms (except those specialized for finance applications), it does not necessarily need to be *fast*.
- Because some functions, such as the trigonometric functions, may be rarely used, user reporting will be infrequent, so that bugs may remain hidden for years.

A natural solution would be to implement the decimal functions using the binary ones and radix conversions. If the conversions are overlapped with the binary functions, good throughput would be kept. Also, validating the “new” set of decimal functions would just require validating once and for all the conversion algorithms. Note that using this approach in a naive way could sometimes lead to poor accuracy.

We focus on the “binary encoding” format of decimal floating-point arithmetic. Cornea et al. describe a software implementation of IEEE 754 decimal arithmetic using that encoding [4]. Harrison [7] suggests re-using binary functions as much as possible, and to use radix conversions, with ad-hoc improvements when needed, to implement decimal transcendental functions. He notices that for implementing a function f , the naive method fails when the “condition number” $|x \cdot f'(x)/f(x)|$ is large. A typical example is the evaluation of trigonometric functions. We will partially

circumvent that problem by merging a kind of “modular range reduction” with the radix conversion. In the following, we follow the Harrison’s approach, mainly focusing on low-level aspects. We assume that we wish to implement a precision- p_{10} , rounded to nearest, decimal arithmetic, and that the underlying binary arithmetic is of precision p_2 . One of our goals is to estimate what value of p_2 will allow for good quality precision- p_{10} decimal arithmetic. We denote $\text{RN}_{\beta}^p(x)$ the number x rounded to the nearest radix- β , precision- p FP number. We assume that the binary format is wide enough, so that there are no over/underflow issues to be considered when converting from decimal to binary. We call a *midpoint* the exact middle of two consecutive FP numbers.

Even if our goal is correctly rounded *functions*, fulfilling that goal will not necessarily require correctly-rounded *conversions*. The conversion algorithms presented in Section II sometimes return a result within slightly more than $1/2$ ulp from the exact value. We assume that, when converting a precision- p_{10} decimal FP number x_{10} to precision- p_2 binary, we get a result

$$x_2 = R_{10 \rightarrow 2}(x_{10}) = x_{10}(1 + \epsilon), \quad (1)$$

with $|\epsilon| \leq 2^{-p_2} + 3 \cdot 2^{-2p_2}$;

and when converting a precision- p_2 binary FP number z_2 to precision- p_{10} decimal, we get a result

$$z_{10} = R_{2 \rightarrow 10}(z_2) = \text{RN}_{10}^{p_{10}}(z_2^*), \quad (2)$$

with $z_2^* = z_2(1 + \epsilon)$ and $|\epsilon| \leq 2^{-p_2} + 3 \cdot 2^{-2p_2}$.

The conversion algorithms presented in Section II satisfy these requirements. If the conversions are correctly rounded (to the nearest), then (1) and (2) are also satisfied.

II. RADIX CONVERSION ALGORITHMS

We now present two conversion algorithms that *do not always* return a correctly-rounded result. They require the availability of a fused multiply-add (FMA) instruction in binary FP arithmetic. Their accuracy suffices for our purpose (implementing decimal functions), but they cannot be directly used for implementing the conversions specified by the IEEE 754-2008 standard. One may precompute the input

values for which these algorithms do not provide correctly-rounded conversions, and use this information to design correctly-rounding variants.

Early works on radix conversion were done by Goldberg [6] and by Matula [11]. Assuming a radix-2 internal arithmetic, algorithms for input and output radix conversion can be found in the literature [2], [3], [14].

IEEE 754–2008 specifies *two* encoding systems for decimal floating-point arithmetic, called the *decimal* and *binary* encodings. We focus here on the *binary* 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 assume here that a decimal number x_{10} is represented by an exponent e_{10} and an integral significand X_{10} , $|X_{10}| \leq 10^{p_{10}} - 1$ such that $x_{10} = X_{10} \cdot 10^{e_{10}-p_{10}+1}$.

Converting from decimal to binary essentially consists in performing, in binary arithmetic, the multiplication $X_{10} \times 10^{e_{10}-p_{10}+1}$, where X_{10} is already in binary, and the binary representation of a suitable approximation to $10^{e_{10}-p_{10}+1}$ is precomputed and stored. Conversion from binary to decimal will essentially consist in performing a multiplication by an approximation to the inverse constant.

In [4], Cornea et al. give constraints on the accuracy of the approximation to the powers of ten used in conversions.

Our goal is to implement conversions using, for performing the multiplications by the factors $10^{e_{10}-p_{10}+1}$, a fast FP multiply-by-a-constant algorithm suggested by Brisebarre and Muller [1], and then to use these conversions for implementing functions in decimal arithmetic using already existing binary functions.

A. Multiplication by a constant

We want to compute $C \cdot x$ with correct rounding (to nearest even) in binary, precision- p_2 , FP arithmetic, where C is a high-precision constant, and x is a FP number. We assume that an FMA instruction is available. We also assume that the two following FP numbers are precomputed:

$$C_h = \text{RN}_2^{p_2}(C) \text{ and } C_\ell = \text{RN}_2^{p_2}(C - C_h). \quad (3)$$

We use the following multiplication algorithm:

Algorithm 1: (Multiplication by C). From x , compute

$$\begin{cases} u = \text{RN}_2^{p_2}(C_\ell x), \\ v = \text{RN}_2^{p_2}(C_h x + u). \end{cases} \quad (4)$$

The result to be returned is v .

Brisebarre and Muller give methods that allow one to check, for a given C and a given precision p_2 , whether Algorithm 1 always returns a correctly-rounded product or not. For the constants C for which it does not always return a correctly-rounded result, their methods also give the (in general, very few) values of x for which it does not.

Even when the multiplication is not correctly rounded, one can easily show that for $p_2 \geq 2$:

$$v = C \cdot x \cdot (1 + \alpha), \text{ with } |\alpha| \leq 2^{-p_2} + 3 \cdot 2^{-2p_2}. \quad (5)$$

B. Decimal to binary conversion, possibly with range reduction

Converting $x_{10} = X_{10} \cdot 10^{e_{10}-p_{10}+1}$ to binary consists in getting $10^{e_{10}-p_{10}+1}$ in binary, and then to multiply it by X_{10} . In all the cases considered in this paper, X_{10} is exactly representable in precision- p_2 binary arithmetic. Hence, our problem is to multiply, in binary, precision- p_2 arithmetic, the exact floating-point number X_{10} by $10^{e_{10}-p_{10}+1}$, and to get the product possibly rounded-to-nearest. To perform the multiplication, we use Algorithm 1: we assume that two precomputed tables T_H and T_L , addressed by e_{10} , of binary, precision- p_2 FP numbers contain the following values:

$$\begin{cases} T_H[e_{10}] = \text{RN}_2^{p_2}(10^{e_{10}-p_{10}+1}), \\ T_L[e_{10}] = \text{RN}_2^{p_2}(10^{e_{10}-p_{10}+1} - T_H[e_{10}]). \end{cases}$$

The multiplication algorithm consists in computing, using a multiplication followed by an FMA, $u = \text{RN}_2^{p_2}(T_L[e_{10}] \cdot X_{10})$, and $x_2 = \text{RN}_2^{p_2}(T_H[e_{10}] \cdot X_{10} + u)$. (5) implies that $x_2 = x_{10} \cdot (1 + \alpha)$, with $|\alpha| \leq 2^{-p_2} + 3 \cdot 2^{-2p_2}$.

A potential benefit of our approach is that range reduction can be partly merged with conversion, which avoids the loss of accuracy one might expect with the trigonometric functions of an argument close to a large multiple of π . For that, we run the same algorithm, replacing the values T_H and T_L given above by

$$\begin{cases} T_H[e_{10}] = \text{RN}_2^{p_2}(10^{e_{10}-p_{10}+1} \bmod (2\pi)) \\ T_L[e_{10}] = \text{RN}_2^{p_2}(10^{e_{10}-p_{10}+1} \bmod (2\pi) - T_H[e_{10}]). \end{cases}$$

The obtained binary result is equal to x_{10} plus or minus a multiple of 2π , and is of absolute value $\leq 2 \cdot (10^{p_{10}} - 1)\pi$. To estimate accuracy obtained using this process, one must compute the “hardest to range-reduce” number of that decimal format. This is done using a continued-fraction based algorithm due to Kahan. See [12] for details. This range reduction algorithm process is close to Daumas et al’s “modular” range reduction [5] (with the addition of the radix conversion).

C. Binary to decimal conversion

Assume the input binary FP value z_2 to be converted to decimal is of exponent k . Let us call z_{10} the decimal floating-point value we wish to obtain. Again, we suggest a conversion strategy that almost always provides a correctly rounded result—namely, $\text{RN}_{10}^{p_{10}}(z_2)$, and, when it does not, has error bounds that allow some correctly rounded decimal functions. We assume that we have tabulated $T'_H[k] = \text{RN}_2^{p_2}(10^{p_{10}-1-m})$ and $T'_L[k] = \text{RN}_2^{p_2}(10^{p_{10}-1-m} - T'_H[k])$, where $m = \lfloor k \cdot \ln(2)/\ln(10) \rfloor$. We will have $1 \leq z_2 \cdot 10^{-m} < 20$, so that m is the “tentative” exponent of z_{10} . This gives the following method:

- 1) approximate $z_2 \cdot 10^{p_{10}-1-m}$ using again Algorithm 1, i.e., compute $u = \text{RN}_2^{p_2}(T'_L[k] \cdot z_2)$ and $v = \text{RN}_2^{p_2}(T'_H[k] \cdot z_2 + u)$, using an FMA. The returned

value v satisfies $v = z_2 \cdot 10^{p_{10}-1-m}(1 + \alpha)$, where $|\alpha| \leq 2^{-p_2} + 3 \cdot 2^{-2p_2}$,

- 2) round v to the nearest integer, say Z_{10}^{tent} ;
- 3) if $|Z_{10}^{\text{tent}}| < 10^{p_{10}}$, then return $Z_{10} = Z_{10}^{\text{tent}}$ as the integral significand, and m as the exponent, of z_{10} . We have $z_{10} = z_2(1 + \beta)(1 + \alpha)$, with $|\beta| \leq \frac{1}{2}10^{-p_{10}+1}$;
- 4) if $|Z_{10}^{\text{tent}}| \geq 10^{p_{10}}$, then the right exponent for z_{10} was $m + 1$: repeat the same calculation with m replaced by $m + 1$.

When the product $z_2 \times 10^{p-1-m}$ is correctly rounded, then we get a correctly rounded conversion, provided that when rounding v to the nearest integer, we follow the same rule in case of a tie as that specified by the rounding direction attribute being chosen.

D. Correctly rounded conversions

When e_{10} is such that multiplication by $10^{e_{10}-p_{10}+1}$ using Algorithm 1 is correctly rounded, the decimal-to-binary conversion of any decimal number of exponent e_{10} will be correctly rounded; and when $m = \lfloor k \ln(2)/\ln(10) \rfloor$ is such that multiplication by $10^{p_{10}-1-m}$ and by $10^{p_{10}-2-m}$ using Algorithm 1 are correctly rounded, the binary-to-decimal conversion of any binary number of exponent k will be correctly rounded. To the tables T_H and T'_H , we may add a one-bit information saying if with their corresponding inputs the conversions are always correctly rounded (except possibly when the exact result is a midpoint of the target format: such cases are easily filtered out).

III. IMPLEMENTING SQUARE ROOT

Assume we wish to implement decimal square root, using the available binary square root. We assume that the binary square root is correctly rounded (to the nearest), and that the radix conversion functions $R_{10 \rightarrow 2}$ and $R_{2 \rightarrow 10}$ satisfy (1) and (2). The input is a decimal number x_{10} with precision- p_{10} . We would like to obtain the decimal number $z_{10 \rightarrow 10}$ nearest to its square root, namely:

$$z_{10 \rightarrow 10} = \text{RN}_{10}^{p_{10}}(\sqrt{x_{10}}).$$

We successively compute $x_2 = R_{10 \rightarrow 2}(x_{10})$ (conversion), $z_2 = \text{RN}_2^{p_2}(\sqrt{x_2})$ (square root evaluation in binary arithmetic), and $z_{10 \rightarrow 2 \rightarrow 10} = R_{2 \rightarrow 10}(z_2)$ (final conversion). Hence, for a given value of p_{10} , we wish to find the smallest value of p_2 for which we always have $z_{10 \rightarrow 2 \rightarrow 10} = z_{10 \rightarrow 10}$.

One can show (see http://prunel.ccsd.cnrs.fr/enl-00463353_v1/ for details), that

Theorem 1 (Decimal sqrt through binary arithmetic): If the precisions and extremal exponents of the decimal and binary arithmetics satisfy $2^{p_2} \geq 3 \cdot 10^{2p_{10}+1}$, $10^{e_{10,\max}+1} < 2^{e_{2,\max}+1}$, $10^{e_{10,\min}-p_{10}+1} > 2^{e_{2,\min}}$, and $p_2 \geq 5$, then $\text{RN}_{10}^{p_{10}}(\sqrt{x_{10}}) = R_{2 \rightarrow 10}(\text{RN}_2^{p_2}(\sqrt{R_{10 \rightarrow 2}(x_{10})}))$ for all decimal, precision- p_{10} , FP numbers x_{10} . ■

Table I gives, for the basic decimal formats of the IEEE 754-2008 standard, the smallest value of p_2 such that, from

$$p_{10} \min\{p_2; \forall \text{ decimal } x, R_{2 \rightarrow 10}(\text{RN}_2^{p_2}(\sqrt{R_{10 \rightarrow 2}(x_{10})})) = \text{RN}_{10}^{p_{10}}(\sqrt{x})\}$$

7	52
16	112
34	231

Table I
SMALLEST p_2 GIVING A CORRECTLY ROUNDED DECIMAL SQUARE ROOT

Theorem 1, the method proposed here is shown to always produce a correctly-rounded square root. Interestingly enough, these results show that to implement a correctly rounded square root in the decimal32 format, using the binary64 ($p_2 = 53$) format suffices; and to implement a correctly rounded square root in the decimal64 format, using the binary128 ($p_2 = 113$) format suffices.

IV. OTHER ARITHMETIC OPERATIONS

It is possible to design algorithms for $+$, $-$, \times , and \div using our approach. They are more complex than square root, because the result can be a decimal midpoint. Concerning $+$, $-$, and \times , it is likely that the algorithms given in [4] have better performance. Division is a case we wish to investigate.

V. SOME RESULTS ON TRANSCENDENTAL FUNCTIONS

We wish to evaluate a function f using the approach we used for the square root: decimal-to-binary conversion, evaluation of f in precision- p_2 binary arithmetic, and then binary-to-decimal conversion. Estimating what value of p_2 —and what accuracy of the binary function—guarantees a correctly-rounded decimal function requires to solve the Table maker’s dilemma (TMD) for f in radix 10. Up to now, authors have mainly focused on that problem in binary arithmetic (see e.g. [9]). The first authors to get all the hardest-to-round cases for a nontrivial function in the decimal64 format were Lefèvre, Stehlé and Zimmerman [10].

A. A simple example: the exponential function

Assume we wish to evaluate $e^{x_{10}}$. We successively compute $x_2 = R_{10 \rightarrow 2}(x_{10})$, and (assuming a correctly-rounded exponential function in binary), $z_2 = \text{RN}_2^{p_2}(e^{x_2})$, and, finally, $z_{10 \rightarrow 2 \rightarrow 10} = R_{2 \rightarrow 10}(z_2)$. We would like to obtain

$$z_{10 \rightarrow 2 \rightarrow 10} = \text{RN}_{10}^{p_{10}}(e^{x_{10}}).$$

Using the bound of the decimal-to-binary conversion algorithm, and the relative error bound of the binary exponential function, we find $R_{2 \rightarrow 10}(z_2) = \text{RN}_{10}^{p_{10}}(z_2^*)$, where

$$z_2^* = e^{x_{10}}(1 + \eta), \quad \text{with } |\eta| \leq 4 \cdot 2^{-p_2}. \quad (6)$$

If the hardest-to-round case is within w^* ulp of the decimal format from a midpoint of that decimal format, then for any midpoint m , $|e^{x_{10}} - m| \geq w^* \cdot x_{10} \cdot 10^{-p_{10}+1}$, which implies, combined with (6) that if $w^* \cdot 10^{-p_{10}+1} \geq 4 \cdot 2^{-p_2}$, then our strategy will always produce correctly rounded

results. For instance, for the decimal32 format ($p_{10} = 7$) and the exponential function, we get $w^* = 5.35 \dots \times 10^{-9}$ ulp, so that $w^* \cdot 10^{-p_{10}+1} = 5.35 \dots \times 10^{-15}$. If $p_2 = 53$, we find $4 \cdot 2^{-p_2} = 4.44 \times 10^{-15}$. From this we deduce

Theorem 2: If the decimal format is the decimal32 format of IEEE 754-2008 and the binary format is the binary64 format, then our strategy always produces correctly rounded decimal exponentials. ■

In the decimal64 format ($p_{10} = 16$), the hardest-to-round case for the exponential function with round-to-nearest is known, and allows us to conclude that

Theorem 3: If the decimal format is the decimal64 format and the binary format is the binary128 format, then our strategy always produces correctly rounded decimal exponentials. ■

B. Preliminary results on the logarithm function

Performing an analysis similar to that of the exponential function, we can show that if the hardest-to-round case is within w^* (decimal) ulp from a midpoint of the decimal format, then our strategy will produce a correctly-rounded result, for $0 < x_{10} < 1/e$ or $x_{10} > e$, as soon as

$$w^* \cdot 10^{-p_{10}+1} \geq 5 \cdot 2^{-p_2}.$$

The hardest-to-round case for logarithms of decimal32 numbers is within $w^* \approx 0.235 \times 10^{-8}$ ulp from a midpoint of the decimal format. This gives

Theorem 4: If the decimal format is the decimal32 format and the binary format is the binary64 format, then our strategy always produces correctly rounded logarithms for input numbers $x_{10} \in [10^{-28}, 1/e] \cup [e, 10^{22}]$. ■

The hardest-to-round case for the logarithm of decimal32 numbers corresponds to $x = 6.436357 \times 10^{-29}$. For that value, we have $w^* \cdot 10^{-p_{10}+1} = 0.8112 \dots \times 10^{-16}$: our method may not work on that value. And yet, still using our computed tables of hardest-to-round cases, we can show that the only decimal32 input values not in $[1/e, e]$ for which our method may not work are 3.3052520E-83, 6.436357E-29, 6.2849190E+22, and 4.2042920E+44: these four values could easily be processed separately.

Implementing logarithms for decimal inputs close to 1 would require a different approach.

VI. CONCLUSION AND FUTURE WORK

We have analyzed a way of implementing decimal floating-point arithmetic on a processor that does not have decimal operators. We have introduced fast conversion algorithms that, without providing correctly rounded conversions, guarantee correctly rounded decimal arithmetic for several functions. Our future work will consist in improving the conversion algorithms, and getting hardest-to-round cases in the decimal64 format.

REFERENCES

- [1] N. Brisebarre and J.-M. Muller. Correctly rounded multiplication by arbitrary precision constants. *IEEE Transactions on Computers*, 57(2):165–174, February 2008.
- [2] 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.
- [3] W. D. Clinger. Retrospective: how to read floating-point numbers accurately. *ACM SIGPLAN Notices*, 39(4):360–371, April 2004.
- [4] 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.
- [5] 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.
- [6] I. B. Goldberg. 27 bits are not enough for 8-digit accuracy. *Commun. ACM*, 10(2):105–106, 1967.
- [7] J. Harrison. Decimal transcendentals via binary. In *Proceedings of the 19th IEEE Symposium on Computer Arithmetic (ARITH-19)*. IEEE Computer Society Press, June 2009.
- [8] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, August 2008. available at <http://ieeexplore.ieee.org/servlet/opac?punumber=4610933>.
- [9] V. Lefèvre 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 (ARITH-16)*, Vail, CO, June 2001.
- [10] V. Lefèvre, D. Stehlé, 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.
- [11] D. W. Matula. In-and-out conversions. *Communications of the ACM*, 11(1):47–50, January 1968.
- [12] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhäuser Boston, MA, 2nd edition, 2006.
- [13] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.
- [14] 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.