Implementing decimal floating-point arithmetic through binary: some suggestions

Nicolas Brisebarre, Milos Ercegovac, Nicolas Louvet, Erik Martin-Dorel, Jean-Michel Muller, Adrien Panhaleux

Implementing decimal floating-point arithmetic through binary: some suggestions

Nicolas Brisebarre, Milos Ercegovac, Nicolas Louvet,
Erik Martin-Dorel, Jean-Michel Muller, and Adrien Panhaleux

Abstract

We propose several 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. Specifically, we discuss the decimal square root and some transcendental functions. We also consider radix conversion algorithms.

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

I. Introduction

The recent IEEE 754-2008 standard for floating-point (FP) arithmetic [10], [16], [19] specifies decimal formats. The main parameters of the decimal interchange formats of that standard are listed in Table I.

Name	decimal32	decimal64 (basic)	decimal128 (basic)
precision	7	16	34
$e_{10,\max}$	$+96$	$+384$	$+6144$
$e_{10,\min}$	-95	-383	-6143

Table I. Main parameters of the decimal interchange formats of size up to 128 bits specified by the 754-2008 standard [10], [19].

Decimal arithmetic is mainly used in financial applications. This has the following implications:

- An implementation must be *correct* (especially, the arithmetic operations must round according to what is specified by the standard), but on most platforms (except those specialized for finance and business applications), it does not necessarily need to be *fast* (that is, one may sacrifice latency if a good throughput is guaranteed).
- Because some functions, such as the trigonometric functions, may be very rarely used, user reporting will be infrequent, so that possible bugs may remain hidden for years.

Hence, a natural solution would be to implement the decimal functions using the binary ones and radix conversions. If the radix conversions are overlapped with the binary functions, good throughput would be kept. Also, this would avoid having to restart from scratch in decimal arithmetic the considerable effort done in the last 20 years on developing high quality binary functions, and 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 (i.e., just converting to binary, computing the function in binary, and converting back the result to decimal) could sometimes lead to poor accuracy.

In this paper, we will focus on the "binary encoding" format of decimal floating-point arithmetic. Cornea, Harrison, Anderson, Tang, and Gvozdev describe a software implementation of IEEE 754-2008 arithmetic using that encoding [5]. Harrison [9] suggests re-using binary transcendental functions as much as possible, and to use radix conversions, with ad-hoc improvements when needed, to implement decimal transcendental functions. As noticed by Harrison, for implementing a function f, the naive method fails when the "condition number" $|x \cdot f'(x)/f(x)|$ becomes large. A typical example of that is the evaluation of trigonometric functions. Consider computing, using the naive method, the sine of the decimal32 number $x_{10} = 1.234567 \times 10^{22}$. The binary64 number nearest x_{10} is $x_2 = 12345669999999999541248$. The binary64 number nearest the sine of x_2 is $8305399354678595/9007199254740992$. Rounding that number to decimal32 gives 0.9220846, whereas the decimal32 number nearest $\sin(x_{10})$ is -0.8600666. 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 (implementing decimal functions through binary arithmetic), mainly focusing on more low-level aspects (arithmetic operations, conversions, and range reduction). We assume that we wish to implement a precision-p_{10}, rounded to nearest,[1] decimal arithmetic. We assume the underlying binary arithmetic is of precision p_2. One of our major goals will be to estimate what value of p_2 will allow for good quality precision-p_{10} decimal arithmetic. We will denote

$$\mathrm{RN}_\beta^p(x)$$

the number x rounded to the nearest radix-β, precision-p floating-point number (when this is not specified in the text, we assume round-to-nearest even in case of a tie). We assume that the binary format is such that $10^{e_{10,\max}+1} < 2^{e_{2,\max}+1}$, and $10^{e_{10,\min}-p_{10}+1} > 2^{e_{2,\min}}$, 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 (to nearest) *functions*, we will see in the following that fulfilling that goal will not necessarily require correctly-rounded *conversions*. Indeed, the conversion algorithms presented in Section II will sometimes—although very rarely—return a result within slightly more than $1/2$ ulp from the exact value. We will assume that, when converting:

- from a precision-p_{10} decimal FP number x_{10} to precision-p_2 binary, we get a result

$$\begin{aligned} x_2 = R_{10\to 2}(x_{10}) = x_{10}(1+\epsilon),\\ \text{with } |\epsilon| \le 2^{-p_2} + 3\cdot 2^{-2p_2}; \end{aligned} \tag{1}$$

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

$$\begin{aligned} z_{10} = R_{2\to 10}(z_2) = \mathrm{RN}_{10}^{p_{10}}(z_2^*),\\ \text{with } z_2^* = z_2(1+\epsilon),\\ \text{and } |\epsilon| \le 2^{-p_2} + 3\cdot 2^{-2p_2}. \end{aligned} \tag{2}$$

The conversion algorithms presented in Section II satisfy these requirements. Of course, if the conversions are correctly rounded (to the nearest), then (1) and (2) are also satisfied (in that case, we may even get smaller bounds on p_2).

II. Radix Conversion Algorithms

In this section, we present two very fast radix-conversion algorithms that *do not always* return a 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 [8] and by Matula [17]. 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 [2]–[4], [20], [21].

The IEEE 754–2008 standard [10] 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 x_{10} is represented by an exponent e_{10} and an integral significand X_{10}, $|X_{10}| \le 10^{p_{10}} - 1$ such that

$$x_{10} = X_{10}\cdot 10^{e_{10}-p_{10}+1}.$$

From this, one can easily deduce that 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 available in binary, and the binary representation of $10^{e_{10}-p_{10}+1}$ (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 [5], 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

[1] Our study is easily generalizable to directed roundings: we focus on round-to-nearest for the sake of brevity.

binary encoding is used.

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

A. Multiplication by a constant

We want to compute $C \cdot x$ with correct rounding (assuming rounding to nearest even) in binary, precision-p_2, FP arithmetic, where C is a constant (i.e., C is known at compile time), and x is a floating-point number. C is not an FP number, nor the middle of two consecutive FP numbers (otherwise the problem would be straightforward). We assume that an fma instruction is available. We also assume that the two following FP numbers are precomputed:

$$\left\{ \begin{array}{rcl} C_h & = & \mathrm{RN}_2^{p_2}(C), \\ C_\ell & = & \mathrm{RN}_2^{p_2}(C - C_h), \end{array} \right. \tag{3}$$

We use the following multiplication algorithm:

Algorithm 1: (Multiplication by C with a multiplication and an fma). From x, compute

$$\left\{ \begin{array}{rcl} u & = & \mathrm{RN}_2^{p_2}(C_\ell x), \\ v & = & \mathrm{RN}_2^{p_2}(C_h x + u). \end{array} \right. \tag{4}$$

The result to be returned is v.

It is worth pointing out that without the use of an fma instruction, Algorithm 1 would fail to return a correctly rounded result for all but a few simple (e.g., powers of 2) values of x. Brisebarre and Muller give methods [1] that allow one to check, for a given constant C and a given precision p_2, whether Algorithm 1 always returns a correctly-rounded product—i.e., $v = \mathrm{RN}_2^{p_2}(C \cdot x)$ for all FP numbers x—or not. For the constants C for which the algorithm 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

$$\begin{array}{rcl} v & = & C \cdot x \cdot (1+\alpha), \\ & & \text{with} \\ |\alpha| & \leq & 2^{-p_2} + 2 \cdot 2^{-2p_2} + 3 \cdot 2^{-3p_2} + 2^{-4p_2}. \end{array} \tag{5}$$

or, more simply, $|\alpha| \leq 2^{-p_2} + 3 \cdot 2^{-2p_2}$ for $p_2 \geq 2$ (which always holds in practice).

B. Decimal to binary conversion, possibly with range reduction

Converting $x_{10} = X_{10} \cdot 10^{e_{10}-p_{10}+1}$ to binary FP arithmetic consists in getting $10^{e_{10}-p_{10}+1}$ in binary, and then to multiply it by X_{10}. Notice that, in all the cases considered later on in this paper (for which $2^{p_2} \approx 10^{2p_{10}}$), X_{10} is exactly representable in precision-p_2 binary FP 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. Notice that when $10^{e_{10}-p_{10}+1}$ is exactly representable in precision-p_2 binary arithmetic, this will be straightforward, so we assume we are not in that case.

To perform the multiplication, we will use Algorithm 1. More precisely, 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:

$$\left\{ \begin{array}{rcl} T_H[e_{10}] & = & \mathrm{RN}_2^{p_2}\left(10^{e_{10}-p_{10}+1}\right), \\ T_L[e_{10}] & = & \mathrm{RN}_2^{p_2}\left(10^{e_{10}-p_{10}+1} - T_H[e_{10}]\right). \end{array} \right.$$

The multiplication algorithm consists in computing, using a multiplication followed by an fma:

$$\left\{ \begin{array}{rcl} u & = & \mathrm{RN}_2^{p_2}(T_L[e_{10}] \cdot X_{10}) \\ x_2 & = & \mathrm{RN}_2^{p_2}(T_H[e_{10}] \cdot X_{10} + u). \end{array} \right.$$

Notice that (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 big loss of accuracy one might expect, for instance, with the trigonometric functions, when the decimal input value is large and close to a multiple of π. For that, it suffices to notice that we can run exactly the same algorithm, replacing the values T_H and T_L given above by

$$\left\{ \begin{array}{rcl} T_H[e_{10}] & = & \mathrm{RN}_2^{p_2}\left(10^{e_{10}-p_{10}+1} \mod (2\pi)\right) \\ T_L[e_{10}] & = & \mathrm{RN}_2^{p_2}\left(10^{e_{10}-p_{10}+1} \mod (2\pi) - T_H[e_{10}]\right) \end{array} \right.$$

The obtained binary result will be equal to x_{10} plus or minus a multiple of 2π, and it will be of absolute value less than

$$2 \cdot (10^{p_{10}} - 1)\pi.$$

For estimating, depending on p_{10} and $e_{10,\max}$ which accuracy will be obtained using that process, one must compute the "hardest to range-reduce" number of that decimal format. This is easily done using a continued-fraction based algorithm due to Kahan. See [18] for details.

This range reduction algorithm process is similar to Daumas et al's "modular" range reduction [6] (with the addition of the radix conversion). The improvements brought to modular range reduction by Villalba et al. [22] and Jaime et al. [11] might possibly be adapted to this new context.

C. Binary to decimal conversion

Assume the input binary FP value z_2 to be converted to decimal is of exponent 2^k. Let us call z_{10} the decimal floating-point value we wish to obtain.

Again, we will suggest a conversion strategy that almost always provides a correctly rounded result—namely, $\mathrm{RN}_{10}^{p_{10}}(z_2)$, and, when it does not, has error bounds that allow some correctly rounded decimal functions.

We assume that we have tabulated

$$\begin{cases} T'_H[k] &= \mathrm{RN}_2^{p_2}\left(10^{p_{10}-1-m}\right) \\ T'_L[k] &= \mathrm{RN}_2^{p_2}\left(10^{p_{10}-1-m} - T'_H[k]\right) \end{cases}$$

where

$$m = \left\lfloor \log_{10} 2^k \right\rfloor = \left\lfloor k \cdot \frac{\ln(2)}{\ln(10)} \right\rfloor.$$

The best way of doing that is probably to tabulate function $k \mapsto \left\lfloor \log_{10} 2^k \right\rfloor$ and to re-use the tables T_H and T_L of the decimal-to-binary conversion.

We will have

$$1 \le z_2 \cdot 10^{-m} < 20,$$

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
$$\begin{cases} u &= \mathrm{RN}_2^{p_2}(T'_L[k] \cdot z_2), \\ v &= \mathrm{RN}_2^{p_2}(T'_H[k] \cdot z_2 + u), \end{cases}$$
using an fma. The returned value v satisfies
$$v = z_2 \cdot 10^{p_{10}-1-m}(1+\alpha),$$
where α is the same as in the previous section, i.e., it satisfies $|\alpha| \le 2^{-p_2} + 3 \cdot 2^{-2p_2}$,
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 $|\beta| \le \frac{1}{2}10^{-p_{10}+1}$;
4) if $|Z_{10}^{\text{tent}}| \ge 10^{p_{10}}$, then the right exponent for z_{10} was $m+1$: repeat the same calculation with m replaced by $m+1$.

Notice that 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.

Several variants are possible. For instance, at the price of larger tables, one may considerably lower the probability of having to re-do the calculations (step 4 of the algorithm) with $m+1$ because of a wrong guess of m: it suffices to use a few most significant bits of the significand of z_2 (in addition to the bits of the binary exponent k) to address the table that returns the decimal exponent guess m.

D. Correctly rounded conversions

As one may easily infer from the presentation of the conversion algorithms:

- when e_{10} is such that multiplication by $10^{e_{10}-p_{10}+1}$ using Algorithm 1 in precision-p_2 binary arithmetic is correctly rounded, the decimal-to-binary conversion of any decimal number of exponent e_{10} will be correctly rounded;
- when $m = \lfloor \log_{10}(2^k) \rfloor = \lfloor k \ln(2)/\ln(10) \rfloor$ is such that multiplication by $10^{p_{10}-1-m}$ and by $10^{p_{10}-2-m}$ using Algorithm 1 in precision-p_2 binary arithmetic are correctly rounded, the binary-to-decimal conversion of any binary number of exponent k will be correctly rounded.

Hence, to the tables T_H and T'_H, we may add a one-bit information saying if with their corresponding inputs the conversions will always be correctly rounded. When this is not the case, we may also store the cases (in general, only one value of the significand) for which they are not, along with the corresponding correct product. We give below, for the combinations that are most useful in this paper (namely, $p_{10} = 7$ and $p_2 = 53$ on the one hand; and $p_{10} = 16$ and $p_2 = 113$ on the other hand), for all values of n that correspond to a conversion algorithm, information on whether multiplication by 10^n using Algorithm 1 is correctly rounded or not.

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 $R_{10\to 2}$ and $R_{2\to 10}$ satisfy (1) and (2). The input is a decimal number x_{10} with precision-p_{10}. We would like to obtain the decimal number $z_{10\to 10}$ nearest to its square-root, namely:

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

To do that, we will successively compute

$$x_2 = R_{10\to 2}(x_{10}),$$

Table II. Only values of k of the form $e_{10} - p_{10} + 1$, where e_{10} is a possible exponent of the decimal32 format and $p_{10} = 7$, for which Algorithm 1 does not provide $\mathrm{RN}_2^{53}(10^k \cdot x)$, in binary64 arithmetic, for all binary64 numbers x. For each of these k, we also give the only value of the integral significand X of x for which the product is not correctly rounded.

k	X
-89	6601914299527020
-88	5281531439621616
-80	7442610212143378
-75	5775274921417125
-74	4620219937133700
-39	8862054676683570
-27	6322612303128019
44	5303153036887306
58	8642445784927644
81	6901257826767179
88	5651538526623358

(radix conversion)

$$z_2 = \mathrm{RN}_2^{p_2}(\sqrt{x_2}),$$

(square root evaluation—correctly rounded—in binary arithmetic), and

$$z_{10\to 2\to 10} = R_{2\to 10}(z_2)$$

(final radix conversion). This process is illustrated in Fig. 1. Hence, for a given value of p_{10}, we wish to find the smallest value of p_2 for which we always have

$$z_{10\to 2\to 10} = z_{10\to 10}.$$

Since the conversion from decimal to binary satisfies the above-given bound (5) and there is no underflow, we have

$$x_2 = x_{10}(1+\epsilon_1), \text{ with } |\epsilon_1| \le 2^{-p_2} + 3\cdot 2^{-2p_2}.$$

Therefore, using the Taylor expansion of the square root, one easily shows that, for $p_2 \ge 4$ (which of course always holds),

$$\sqrt{x_2} = \sqrt{x_{10}}(1+\alpha), \text{ with } |\alpha| < 5\cdot 2^{-p_2-3}.$$

Also, since the binary square root is correctly rounded, we have,

$$\begin{aligned} z_2 &= \sqrt{x_2}(1+\epsilon_2), \text{ with } |\epsilon_2| \le 2^{-p_2} \\ &= \sqrt{x_{10}}(1+\alpha)(1+\epsilon_2), \end{aligned}$$

Table III. Only values of k of the form $e_{10} - p_{10} + 1$, where e_{10} is a possible exponent of the decimal64 format and $p_{10} = 16$, for which Algorithm 1 does not provide $\mathrm{RN}_2^{113}(10^k \cdot x)$, in binary128 arithmetic, for all binary128 numbers x. For each of these k, we give the only value of the integral significand of x for which the product is not correctly rounded.

k	X
-386	91127342379322186908533996967743 99
-378	72421991641007116789647959217636 75
-370	73877801479327112818462134078608 27
-366	90305354484075230938611021232335 82
-341	88766224497953533854254134013728 81
-337	63988007295625826430935638058683 18
-321	97330585207188513952401932227889 51
-299	72540715695076980674474096484783 58
-255	84200371884307500898749671075660 84
-216	70922228966843715773290110085091 26
-213	59781045147463166985229498621142 6
-208	78192016118170332932503482105213 62
-205	91764999264064774417517315100526 09
-201	61475468046845547288062051129169 12
-170	67411155015275519918763873298284 38
-147	70470779621186629073013018928214 41
-135	64134595844519324430840693323663 59
-135	88801748092411372288856344601995 74
-98	91314658892599178879105279025080 07
-94	61072213623421020989578754458414 55
-93	97715541797473633583326007133463 28
-91	52457726480184454093107117054445 53
-79	82141724092248030967151717938246 55
50	88304956287087816165007198162696 37
53	89771301642545151394380980977189 14
66	82421817424297411541071844437796 15
67	65937453939437929232857475550236 92
68	60822378535440293912290651997066 47
72	66479862803162394710027150435163 49
97	78286829686557732692751532719894 61
104	85426214030399103606984315182302 41
114	81275860598572799455624432404235 37
117	91401288149788078932087587068991 01
136	10040263224243486040560969864754 721
168	88445163418940393528279577361642 63
196	79240135838062675109920346610170 08
199	89603781100170085659920246361234 30
200	71683024880136068527936197088987 44
209	96398585436946138314456856108401 27
211	86012843110935467064691648650911 53
228	99060475204266019771633342934967 94
232	72348038411697063731238807072540 55
233	57878430729357650984991045658032 44
271	67416329862234880596146261869383 07
275	71506148682714236588846467678564 06
304	96330294142315187964241268493736 38
311	83807589312919513360607120770391 87
313	10261688674644756765606840220687 138
316	60234549813344751458072726724236 66
320	89040562460384492673739357284478 95
327	67329195943316058412307042906403 99
331	77211026742331379849820333397264 63
336	59477062031607629692888473979126 50
337	95163299250572207508621558366602 40
361	76370477515325858049454356957701 013
362	91794282472487467437257721717679 59

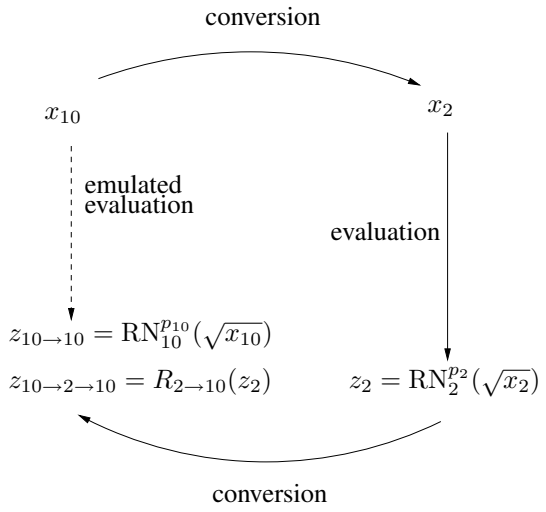

Figure 1. Implementing decimal square root through binary: we first convert the decimal number x_{10}, and get a binary number x_2. $\sqrt{x_2}$ is evaluated, which gives z_2, and z_2 is converted to decimal, which gives $z_{10\to2\to10}$. We wish to know which conditions on p_2 need to be satisfied so that $z_{10\to2\to10}$ is always equal to $z_{10\to10} = \mathrm{RN}_{10}^{p_{10}}(\sqrt{x_{10}})$.

from which we deduce, after some straightforward calculations, that for $p_2 \geq 4$,

$$z_2 = \sqrt{x_{10}}(1+\eta) \text{ with } |\eta| < \frac{7}{4}2^{-p_2}. \tag{6}$$

Now, we determine if $R_{2\to10}(z_2)$ is equal to $\sqrt{x_{10}}$ rounded to the nearest precision-p_{10} decimal number. Since $R_{2\to10}(z_2) = \mathrm{RN}_{10}^{p_{10}}(z_2 \cdot (1+\epsilon))$, where $|\epsilon| \leq 2^{-p_2} + 3 \cdot 2^{-2p_2}$, this will be the case if and only if there is no exact midpoint between $z_2^* = z_2 \cdot (1+\epsilon)$ and $\sqrt{x_{10}}$. Therefore, we now have to estimate the minimum possible relative distance between the square root of a precision-p_{10} decimal number x_{10} and a decimal midpoint. To that purpose, without loss of generality, we can assume that $1 \leq x_{10} < 100$, so that $1 \leq \sqrt{x_{10}} < 10$. The middle m of two consecutive decimal FP numbers in that domain has the form

$$\left(M + \frac{1}{2}\right) \cdot 10^{-p_{10}+1},$$

where M is an integer satisfying $10^{p_{10}-1} \leq M \leq 10^{p_{10}} - 1$. The FP number x_{10} has the form

$$X \cdot 10^{-p_{10}+\delta+1},$$

where X is an integer satisfying $10^{p_{10}-1} \leq X \leq 10^{p_{10}}-1$, and $\delta \in \{0,1\}$. Let us try to find a lower bound on $|\beta|$, where $m = \sqrt{x_{10}} \cdot (1+\beta)$.

From $m^2 = x_{10} \cdot (1+\beta)^2$, we deduce

$$\left(M + \frac{1}{2}\right)^2 \cdot 10^{-2p_{10}+2} = X \cdot 10^{-p_{10}+\delta+1} \cdot (1+\beta)^2,$$

hence,

$$(2M+1)^2 = X \cdot 2^{p_{10}+\delta+1} \cdot 5^{p_{10}+\delta-1} \cdot (1+\beta)^2. \tag{7}$$

From (7), we can easily find again the well-known fact [15] that the square-root of an FP number cannot be a midpoint of the same format: $\beta = 0$ is impossible since the left-hand part of (7) is an odd integer, and the right-hand part would be an even integer. Now if $\beta \neq 0$, then (7) implies that $(1+\beta)^2$ is an integer multiple of

$$\frac{1}{X \cdot 2^{p_{10}+\delta+1} \cdot 5^{p_{10}+\delta-1}}.$$

Moreover, $(1+\beta)^2 \neq 1$, thus

$$\left|(1+\beta)^2 - 1\right| \geq \frac{1}{X \cdot 2^{p_{10}+\delta+1} \cdot 5^{p_{10}+\delta-1}} > \frac{1}{2^{2p_{10}+2} \cdot 5^{2p_{10}}}.$$

Since m is the closest midpoint to $\sqrt{x_{10}}$, one has $|\beta| \leq 1/2 \cdot 10^{1-p_{10}} \leq 1/2$ as soon as $p_{10} \geq 1$, hence

$$|(1+\beta)^2 - 1| \leq |\beta| \cdot (2 + |\beta|) \leq 5/2 \cdot |\beta|.$$

As a consequence,

$$\beta > \frac{1}{10^{2p_{10}+1}}. \tag{8}$$

Now, (6) implies that the number z_2^* such that $z_{10\to2\to10} = \mathrm{RN}_{10}^{p_{10}}(z_2^*)$ satisfies $z_2^* = \sqrt{x_{10}}(1+\eta)(1+\epsilon)$ with $|\eta| \leq \frac{7}{4}2^{-p_2}$ and $|\epsilon| \leq 2^{-p_2} + 3 \cdot 2^{-2p_2}$. From this, we easily deduce that for $p_2 \geq 5$, $z_2^* = \sqrt{x_{10}}(1+\eta^*)$, with $|\eta^*| < 3 \cdot 2^{-p_2}$.

Now, by combining this with (8), we deduce that for

$$3 \cdot 2^{-p_2} \leq \frac{1}{10^{2p_{10}+1}},$$

we will always have $z_{10\to2\to10} = z_{10\to10}$. This gives the following result.

Theorem 1 (Decimal sqrt through binary arithmetic): If the precisions and extremal exponents of the decimal and binary arithmetics satisfy

$$2^{p_2} \geq 3 \cdot 10^{2p_{10}+1},$$

and $10^{e_{10,\max}+1} < 2^{e_{2,\max}+1}$, and $10^{e_{10,\min}-p_{10}+1} > 2^{e_{2,\min}}$, and $p_2 \geq 5$, then

$$\mathrm{RN}_{10}^{p_{10}}(\sqrt{x_{10}}) = R_{2\to10}\left(\mathrm{RN}_2^{p_2}\sqrt{R_{10\to2}(x_{10})}\right)$$

for all decimal, precision-p_{10}, FP numbers x_{10} (i.e., our method provides a correctly-rounded square root). ■

Table IV gives, for the basic decimal formats of the IEEE 754-2008 standard, the smallest value of p_2 such that, from Theorem 1, the method proposed here is shown to always produce a correctly-rounded square-root. Interestingly enough, these results show that

p_{10}	Smallest p_2 s.t. $R_{2\to 10}\left(\mathrm{RN}_2^{p_2}\sqrt{R_{10\to 2}(x_{10})}\right)$ $= \mathrm{RN}_{10}^{p_{10}}(\sqrt{x}), \forall$ decimal x
7	52
16	112
34	231

Table IV. Smallest values of p_2 that allow one to obtain a correctly rounded decimal square root.

- to implement a correctly rounded square root in the decimal32 format, using the binary64 ($p_2 = 53$) format[2] suffices; and
- to implement a correctly rounded square root in the decimal64 format, using the binary128 ($p_2 = 113$) format[3] suffices.

Implementing square root in the decimal128 format would require the use of a multiple-precision library such as MPFR [7].

IV. Other arithmetic operations

It is possible to design algorithms for addition, subtraction, multiplication, and division using our approach. They are somewhat more complex than square root, because the sum, product, or quotient of two decimal FP numbers can be a decimal midpoint. Concerning addition/subtraction and multiplication, it is very likely that the algorithms given in [5] have better performance. Division is a case we wish to investigate in future studies.

V. Some results on transcendental functions

Dealing with the transcendental functions is easier than dealing with the arithmetic operations, because for the most common ones (sine, cosine, exponentials, logarithms, arctangents), the value of the function at a FP number cannot be a midpoint. We wish to evaluate a function f using the approach depicted by Figure 1 for the square-root: decimal-to-binary conversion (using the algorithm outlined in Section II), evaluation of the function 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 function f in radix 10. Up to now, authors have mainly focused on that problem in binary arithmetic (see e.g. [12], [13]). We are currently working on computing hardest-to-round cases for the most common functions in the basic decimal formats of the IEEE 754-2008 standard. Getting the hardest-to-round cases for a given function in decimal32 arithmetic only requires a few hours of computations. The first authors to get all the hardest-to-round cases for a nontrivial function in the decimal64 formats were Lefèvre, Stehlé and Zimmerman [14].

A. A simple example: the exponential function

Assume we wish to evaluate $e^{x_{10}}$. We successively compute

$$x_2 = R_{10\to 2}(x_{10}),$$

and (assuming a correctly-rounded exponential function in binary, precision-p_2, FP arithmetic)[4],

$$z_2 = \mathrm{RN}_2^{p_2}\left(e^{x_2}\right),$$

and, finally,

$$z_{10\to 2\to 10} = R_{2\to 10}(z_2).$$

We would like to obtain

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

Using the bound of the decimal-to-binary conversion algorithm, and the relative error bound of the binary exponential function, we find

$$z_2 = (1+\beta)\cdot e^{x_{10}}\cdot e^{(1+\alpha)},$$

with $|\alpha| \le 2^{-p_2} + 3\cdot 2^{-2p_2}$ and $|\beta| \le 2^{-p_2}$. Also,

$$R_{2\to 10}(z_2) = \mathrm{RN}_{10}^{p_{10}}(z_2^*),$$

where $z_2^* = z_2(1+\epsilon)$, with $|\epsilon| \le 2^{-p_2} + 3\cdot 2^{-2p_2}$. Combining all these bounds, we find

$$\begin{array}{rcl} z_2^* & = & e^{x_{10}}(1+\eta), \\ & & \text{with} \\ |\eta| & \le & 4\cdot 2^{-p_2} \end{array} \qquad (9)$$

If the hardest-to-round case is within w^* ulp of the decimal format from a midpoint of that decimal format, then it means that for any midpoint m,

$$|e^{x_{10}} - m| \ge w^*\cdot x_{10}\cdot 10^{-p_{10}+1},$$

which implies, combined with (9) that if

$$w^*\cdot 10^{-p_{10}+1} \ge 4\cdot 2^{-p_2},$$

then our strategy will always produce correctly rounded results. For instance, for the decimal32 format ($p_{10} = 7$)

[2]Formerly called "double precision".

[3]Formerly called "quad precision".

[4]But in many cases—as we will see for the decimal32 format, we will have some "margin", so that if the maximum error of the binary exponential function is slightly more than $1/2$ ulp, we will anyway have a correctly rounded decimal exponential.

and the exponential function, we get $w^* = 5.35\cdots \times 10^{-9}$ ulp (it is attained for $x = 2.408597 \times 10^{-3}$), so that $w^* \cdot 10^{-p_{10}+1} = 5.35\cdots \times 10^{-15}$. 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 (i.e., $p_{10} = 7$) and the binary format is the binary64 format (i.e., $p_2 = 53$), then our strategy always produces correctly rounded decimal exponentials, that is,

$$\mathrm{RN}_{10}^{p_{10}}(x_{10}) = R_{2\to 10}\left(\mathrm{RN}_2^{p_2}\left(\exp(R_{10\to 2}(x_{10}))\right)\right),$$

for all decimal32 FP numbers x_{10}. ■

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

$$\exp(9.407822313572878 \times 10^{-2})$$
$$= 1.0986456820663385000000000000000278\cdots$$

for which $w^* \approx 2.78 \times 10^{-18}$ ulp, so that

$$w^* \cdot 10^{-p_{10}+1} \approx 2.78 \times 10^{-33} \text{ ulp}$$

If $p_2 = 113$, we find $4 \cdot 2^{-p_2} \approx 3.85 \times 10^{-34}$.

Hence, we get

Theorem 3: If the decimal format is the decimal64 format of IEEE 754-2008 (i.e., $p_{10} = 16$) and the binary format is the binary128 format (i.e., $p_2 = 113$), then our strategy always produces correctly rounded decimal exponentials, that is,

$$\mathrm{RN}_{10}^{p_{10}}(x_{10}) = R_{2\to 10}\left(\mathrm{RN}_2^{p_2}\left(\exp(R_{10\to 2}(x_{10}))\right)\right),$$

for all decimal64 FP numbers x_{10}. ■

B. Preliminary results on the logarithm function

Performing an analysis very similar to that of the exponential function, we can show that if the hardest-to-round case is within w^* ulp of the decimal format 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 between 10^{-28} and 10^{+22} is within $w^* \approx 0.235 \times 10^{-8}$ ulp from a midpoint of the decimal format. That value is reached for $x_{10} = 1.192327 \times 10^{-20}$, whose logarithm is $-45.875794999999976472\cdots$. We have $w^* \cdot 10^{-p_{10}+1} = 0.235 \times 10^{-14}$, whereas (with $p_2 = 53$), $5 \cdot 2^{-p_2} = 0.555 \times 10^{-15}$, therefore:

Theorem 4: If the decimal format is the decimal32 format of IEEE 754-2008 (i.e., $p_{10} = 7$) and the binary format is the binary64 format (i.e., $p_2 = 53$), then our strategy always produces correctly rounded logarithms, that is,

$$\mathrm{RN}_{10}^{p_{10}}(x_{10}) = R_{2\to 10}\left(\mathrm{RN}_2^{p_2}\left(\ln(R_{10\to 2}(x_{10}))\right)\right),$$

for all decimal32 FP numbers $x_{10} \in [10^{-28}, 1/e] \bigcup [e, 10^{22}]$. ■

The hardest-to-round case for the logarithm of decimal32 numbers corresponds to $x = 6.436357 \times 10^{-29}$, whose logarithm is $-64.9130049999999991880307\cdots$. For that value, we have $w^* \cdot 10^{-p_{10}+1} = 0.8112\cdots 10^{-16}$: 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 (it is probably better to use function $x \to \ln(1 + x)$).

*(**Note to the referees:** in the final version, we will have hardest-to-round cases for all common functions in decimal32, so that we will be able to derive results similar to Theorem 2 for these functions. We will also try to have some results for decimal64, but we cannot be sure: this requires months of CPU)*

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 two conversion algorithms that do not provide correctly rounded conversions, but are fast and suffice to guarantee correctly rounded decimal arithmetic, at least for several functions. Our future work will consist in improving the conversion algorithms (with the aim of deriving algorithms that always produce correctly rounded conversions), getting hardest-to-round cases in the decimal64 format, and trying (for decimal64 functions) to use as much as possible the "double extended" binary format, instead of the binary128 one.

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. How to read floating-point numbers accurately. *ACM SIGPLAN Notices*, 25(6):92–101, June 1990.

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

[5] 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.

[6] 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.

[7] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Transactions on Mathematical Software*, 33(2), 2007. available at http://www.mpfr.org/.

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

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

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] P. Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice-Hall, Englewood Cliffs, NJ, 2000.

[16] 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 für Informatik (IBFI), Schloss Dagstuhl, Germany.

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

[18] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhäuser Boston, MA, 2nd edition, 2006.

[19] 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. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.

[20] 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.

[21] 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.

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