

On the Computation of Correctly-Rounded Sums

Peter Kornerup, Vincent Lefèvre, Nicolas Louvet, Jean-Michel Muller

▶ **To cite this version:**

Peter Kornerup, Vincent Lefèvre, Nicolas Louvet, Jean-Michel Muller. On the Computation of Correctly-Rounded Sums. 19th IEEE Symposium on Computer Arithmetic - Arith'19, Jun 2009, Portland, Oregon, United States. inria-00367584v2

HAL Id: inria-00367584

https://inria.hal.science/inria-00367584v2

Submitted on 23 Mar 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On the Computation of Correctly-Rounded Sums

P. Kornerup V. Lefèvre N. Louvet J.-M. Muller
SDU, Odense, Denmark *LIP, CNRS/ENS Lyon/INRIA/Université de Lyon, Lyon, France*

Abstract

This paper presents a study of some basic blocks needed in the design of floating-point summation algorithms. In particular, we show that among the set of the algorithms with no comparisons performing only floating-point additions/subtractions, the 2Sum algorithm introduced by Knuth is minimal, both in terms of number of operations and depth of the dependency graph. Under reasonable conditions, we also prove that no algorithms performing only round-to-nearest additions/subtractions exist to compute the round-to-nearest sum of at least three floating-point numbers. Starting from an algorithm due to Boldo and Melquiond, we also present new results about the computation of the correctly-rounded sum of three floating-point numbers.

Keywords: Floating-point arithmetic, summation algorithms, correct rounding, 2Sum and Fast2Sum algorithms.

1. Introduction

The computation of sums appears in many domains of numerical analysis. They occur when performing numerical integration, when evaluating dot products, means, variances and many other functions. When computing the sum of n floating-point numbers $a_1, a_2, \ldots, a_n$, the best one can hope is to get $\circ(a_1 + a_2 + \cdots a_n)$, where $\circ$ is the desired rounding mode. On current architectures this can always be done in software using multiple-precision arithmetic. This could also be done using a long accumulator, but such accumulators are not yet available on current processors.

In radix-2 floating-point arithmetic, it is well known that the rounding error generated by a round-to-nearest addition is itself a floating-point number. Many accurate and efficient summation algorithms published in the literature (see for instance [12, 13, 11, 3]) are based on this property and use basic blocks such as the Fast2Sum and the 2Sum algorithms (Algorithms 1 and 2 below) to compute the rounding error generated by a floating-point addition. Since efficiency is one of the main concerns in the design of floating-point programs, we focus on algorithms using only floating-point additions and subtractions in the target format and without conditional branches. The computation of the correctly-rounded sum of three floating-point numbers is also a basic task needed in many different contexts: in [3], Boldo and Melquiond have presented a new algorithm for this task, with an application in the context of the computation of elementary functions. Hence, it is of great interest to study the properties of these basic blocks.

In this paper, we assume an IEEE 754 [1, 7] arithmetic. We show that among the set of the algorithms with no comparisons performing only floating-point operations, the 2Sum algorithm introduced by Knuth is minimal, both in terms of number of operations and depth of the dependency graph. Under reasonable assumptions, we also show that it is impossible to always obtain the correctly round-to-nearest sum of $n \geq 3$ floating-point numbers with an algorithm performing only round-to-nearest additions/subtractions. The algorithm proposed by Boldo and Melquiond for computing the round-to-nearest sum of three floating-point numbers relies on a non-standard rounding mode, *rounding to odd*. We show that rounding to odd can be emulated in software using only floating-point additions/subtractions in the standard rounding modes and a multiplication by the constant 0.5, thus allowing the round-to-nearest sum of three floating-point numbers to be determined without tests. We also propose algorithms to compute the correctly-rounded sum of three floating-point values for directed roundings.

1.1. Assumptions and notations

We assume a *radix-2 and precision-p floating-point arithmetic* as defined in the original IEEE 754-1985 standard [1] as well as in its followers, the IEEE 854-1987 radix-independent standard [2] and the newly IEEE 754-2008 revised standard [7]. The precision p can be either 11, 24, 53 or 113, corresponding to the four binary floating-point formats specified in the IEEE 754-2008 standard. The user can choose an *active rounding mode*, also called *rounding direction attribute* in IEEE 754-2008: round toward $-\infty$, round toward $+\infty$, round toward 0, and round to nearest even, which is the default rounding mode. Given a real number x, we denote respectively by $RD(x)$, $RU(x)$,

$RZ(x)$ and $RN(x)$ these rounding modes.

Let us also recall that *correct rounding* is required for the four elementary arithmetic operations and the square root by the above cited IEEE standards: an arithmetic operation is said to be correctly rounded if for any inputs its result is the infinitely precise result rounded according to the active rounding mode. Correct rounding makes arithmetic deterministic, provided all computations are done in the same format, which might be sometimes difficult to ensure [10]. Correct rounding allows one to design portable floating-point algorithms and to prove their correctness, as the results summarized in the next subsection.

1.2. Previous results

The following result is due to Dekker [5].

Theorem 1 (Fast2Sum algorithm)**.** *Assume a radix-2 floating-point arithmetic providing correct rounding with rounding to nearest. Let a and b be finite floating-point numbers such that the exponent of a is larger than or equal to that of b. The following algorithm computes floating-point numbers s and t such that $s = RN(a + b)$ and $s + t = a + b$ exactly.*

Algorithm 1 (Fast2Sum(a,b))**.**

$$\begin{aligned} s &= RN(a+b); \\ z &= RN(s-a); \\ t &= RN(b-z); \end{aligned}$$

Note that the condition "the exponent of a is larger than or equal to that of b" may be slow to check in a portable way. But if $|a| \geq |b|$, then this condition is fulfilled. If no information on the relative orders of magnitude of a and b is available, there is an alternative algorithm due to Knuth [8] and Møller [9], called 2Sum.

Algorithm 2 (2Sum(a,b))**.**

$$\begin{aligned} s &= RN(a+b); \\ b' &= RN(s-a); \\ a' &= RN(s-b'); \\ \delta_b &= RN(b-b'); \\ \delta_a &= RN(a-a'); \\ t &= RN(\delta_a+\delta_b); \end{aligned}$$

2Sum requires 6 operations instead of 3 for the Fast2Sum algorithm, but on current pipelined architectures, a wrong branch prediction may cause the instruction pipeline to drain. As a consequence, using 2Sum instead of a comparison followed by Fast2Sum will usually result in much faster programs [11]. The names "2Sum" and "Fast2Sum" seem to have been coined by Shewchuk [14]. We call these algorithms *error-free additions* in the sequel.

The IEEE 754-2008 standard [7] describes new operations with two floating-point numbers as operands:

- minNum and maxNum that deliver respectively the minimum and the maximum;
- minNumMag, which delivers the one with the smaller magnitude (the minimum in case of equal magnitudes);
- maxNumMag, which delivers the one with the larger magnitude (the maximum in case of equal magnitudes).

The operations minNumMag and maxNumMag can be used to sort two floating-point numbers by order of magnitude. This leads to the following alternative to the 2Sum algorithm.

Algorithm 3 (Mag2Sum(a,b))**.**

$$\begin{aligned} s &= RN(a+b); \\ a' &= \mathrm{maxNumMag}(a,b); \\ b' &= \mathrm{minNumMag}(a,b); \\ z &= RN(s-a'); \\ t &= RN(b'-z); \end{aligned}$$

Algorithm Mag2Sum consists in sorting the inputs by magnitude before applying Fast2sum. It requires 5 floating-point operations, but notice that the first three operations can be executed in parallel. Mag2Sum can already be implemented efficiently on the Itanium processor, thanks to the instructions famin and famax available on this architecture [4, p. 291].

2. Algorithm 2Sum is minimal

In the following, we call an *RN-addition algorithm* an algorithm only based on additions and subtractions in the round-to-nearest mode: at step i the algorithm computes $x_i = RN(x_j \pm x_k)$, where x_j and x_k are either one of the input values or a previously computed value. An RN-addition algorithm must not perform any comparison or conditional branch, but may be enhanced with minNum, maxNum, minNumMag or maxNumMag as in Theorem 3.

For instance, 2Sum is an RN-addition algorithm that requires 6 floating-point operations. To estimate the performance of an algorithm, only counting the operations is a rough estimate. On modern architectures, pipelined arithmetic operators and the availability of several FPUs make it possible to perform some operations in parallel, provided they are independent. Hence the depth of the dependency graph of the instructions of the algorithm is an important criterion. In the case of Algorithm 2Sum, two operations only can be performed in parallel, $\delta_b = RN(b - b')$ and $\delta_a = RN(a - a')$. Hence the depth of Algorithm 2Sum is 5. In Algorithm Mag2Sum the first three operations can be executed in parallel, hence this algorithm has depth 3.

In this section, we address the following question: are there other RN-addition algorithms producing the same results as 2Sum, i.e., computing both $RN(a+b)$ and the rounding error $a+b-RN(a+b)$ for any floating-point inputs a and b?

We show the following result, proving that among the RN-addition algorithms, 2Sum is minimal in terms of number of operations as well as in terms of depth of the dependency graph. The results of this section are proved for the four binary floating-point formats defined in the IEEE 754-2008 standard [7].

Theorem 2. *Among the RN-addition algorithms computing the same results as 2Sum,*

- *each one requires at least* 6 *operations;*
- *each one with* 6 *operations reduces to 2Sum;*
- *each one has depth at least* 5*.*

Proof. To prove Theorem 2, we have proceeded as follows. We enumerated all possible RN-addition algorithms with 6 additions or less, and each algorithm was tried with 3 pairs of well chosen inputs a and b. The only algorithm that delivered the correct results was essentially 2Sum (by "essentially" we mean that some algorithms reduce to 2Sum through obvious transformations). We also enumerated all possible results at depth 4 or less on 3 pairs of well chosen inputs a and b, showing that the expected results could not be obtained on the 3 pairs at the same time. The whole process has been carried out with the four IEEE 754-2008 binary floating-point formats. □

As previously mentioned an RN-addition algorithm can also be enhanced with minNum, maxNum, minNumMag and maxNumMag operations [7], which is the case for Algorithm Mag2Sum. The following result proves the minimality of this algorithm.

Theorem 3. *We consider the set of all the RN-addition algorithms enhanced with minNum, maxNum, minNumMag and maxNumMag. Among all the algorithms that computes the same results as 2Sum,*

- *each one requires at least* 5 *operations;*
- *each one with* 5 *operations reduces to Mag2Sum;*
- *each one has depth at least* 3*.*

As for the proof of Theorem 2, we used exhaustive enumeration to prove this result.

The C programs used to prove all these results are based on the MPFR library [6] (in order to control the precision and the roundings) and can be found on `http://hal.inria.fr/inria-00367584/`.

3. On the impossibility of computing a round-to-nearest sum

In this section, we are interested in the computation of the sum of n floating-point numbers, correctly rounded to nearest. We prove the following result.

Theorem 4. *Let $a_1, a_2, \ldots, a_n$ be $n \geq 3$ floating-point numbers of the same format. Assuming an* unbounded exponent range*, an RN-addition algorithm cannot always return $RN(a_1 + a_2 + \cdots + a_n)$.*

If there exists an RN-addition algorithm to compute the round-to-nearest sum of n floating-point numbers, with $n \geq 3$, then this algorithm must also compute the round-to-nearest sum of 3 floating-point values. As a consequence we only consider the case $n = 3$ in the proof of this theorem. We show how to construct for any RN-algorithm a set of input data such that the result computed by the algorithm differs from the round-to-nearest result.

Proof of Theorem 4. An RN-addition algorithm can be represented by a directed acyclic graph[1] (DAG) whose nodes are the arithmetic operations. Given such an algorithm, let r be the depth of its associated graph. First we consider the input values a_1, a_2, a_3 defined as follows.

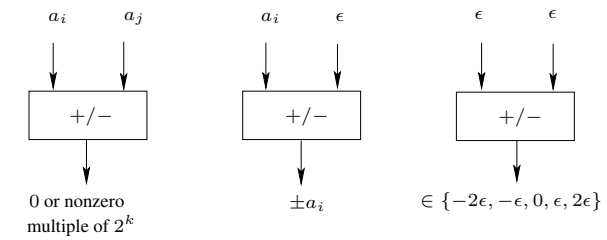
- $a_1 = 2^{k+p}$ and $a_2 = 2^k$: For any[2] integer k, a_1 and a_2 are two nonzero multiples of 2^k whose sum is the exact middle of two consecutive floating-point numbers;
- $a_3 = \varepsilon$, with $0 \leq 2^{r-1}|\varepsilon| \leq 2^{k-p-1}$ for $r \geq 1$.

Note that when $\varepsilon \neq 0$,

$$RN\left(a_1 + a_2 + a_3\right) = \begin{cases} RD(a_1 + a_2 + a_3) & \text{if} \quad \varepsilon < 0 \\ RU(a_1 + a_2 + a_3) & \text{if} \quad \varepsilon > 0, \end{cases}$$

where we may also conclude that $RN\left(a_1 + a_2 + a_3\right) \neq 0$.

The various computations that can be performed "at depth 1", i.e., immediately from the entries of the algorithm are illustrated below. The value of ε is so small that after rounding to nearest, every operation with ε in one of its entries will return the same value as if ε were zero, unless the other entry is 0 or ε.



[1] Such an algorithm cannot have "while" loops, since tests are prohibited. It may have "for" loops that can be unrolled.

[2] Here k is arbitrary. When considering a limited exponent range, we have to assume that $k + p$ is less than the maximum exponent.

An immediate consequence is that after these computations "at depth 1", the possible available variables are nonzero multiples of 2^k that are the same as if ε were 0, and values taken from $\mathcal{S}_1 = \{-2\varepsilon, -\varepsilon, 0, \varepsilon, 2\varepsilon\}$. By induction one easily shows that the available variables after a computation of depth m are either nonzero multiples of 2^k that are the same as if ε were 0 or values taken from $\mathcal{S}_m = \{-2^m\varepsilon, \cdots, 0, \cdots, +2^m\varepsilon\}$.

Now, consider the very last addition/subtraction, at depth r in the DAG of the RN-addition algorithm. If at least one of the inputs of this last operation is a nonzero multiple of 2^k that is the same as if ε were 0, then the other input is either also a nonzero multiple of 2^k or a value belonging to $\mathcal{S}_{r-1} = \{-2^{r-1}\varepsilon, \cdots, 0, \cdots, +2^{r-1}\varepsilon\}$. In both cases the result does not depend on the sign of ε, hence it is always possible to choose the sign of ε so that the round-to-nearest result differs from the computed one. If both entries of the last operation belong to $\mathcal{S}_{r-1}$, then the result belongs to $\mathcal{S}_r = \{-2^r\varepsilon, \cdots, 0, \cdots, +2^r\varepsilon\}$. If one sets $\varepsilon = 0$ then the computed result is 0, contradicting the fact that the round-to-nearest sum must be nonzero. □

In the proof of Theorem 4, it was necessary to assume an unbounded exponent range to make sure that with a computational graph of depth r, we can always build an ε so small that $2^{r-1}\varepsilon$ vanishes when added to any multiple of 2^k. This constraint can be transformed into a constraint on r related to the extremal exponents $e_{\min}$ and $e_{\max}$ of the floating-point system. Indeed, assuming $\varepsilon = \pm 2^{e_{\min}}$ and $a_1 = 2^{k+p} = 2^{e_{\max}}$, the inequality $2^{r-1}|\varepsilon| \leq 2^{k-p-1}$ gives the following theorem.

Theorem 5. *Let $a_1, a_2, \ldots, a_n$ be $n \geq 3$ floating-point numbers of the same format. Assuming the extremal exponents of the floating-point format are $e_{\min}$ and $e_{\max}$, an RN-addition algorithm of depth r cannot always return $RN(a_1 + a_2 + \cdots + a_n)$ as soon as*

$$r \leq e_{\max} - e_{\min} - 2p.$$

For instance, with the IEEE 754-1985 double precision format ($e_{\min} = -1022$, $e_{\max} = 1023$, $p = 53$), Theorem 5 shows that an RN-addition algorithm able to always evaluate the round-to-nearest sum of at least 3 floating-point numbers (if such an algorithm exists!) must have depth at least 1939.

4. Correctly-rounded sums of three floating-point numbers

We have proved in the previous section that there exist no RN-addition algorithms of acceptable size to compute the round-to-nearest sum of $n \geq 3$ floating-point values. In [3], Boldo and Melquiond presented an algorithm to compute $RN(a+b+c)$ using a round-to-odd addition. Rounding to odd is defined as follows:

- if x is a floating-point number, then $RO(x) = x$;
- otherwise, $RO(x)$ is the value among $RD(x)$ and $RU(x)$ whose least significant bit is a one.

The algorithm of Boldo and Melquiond for computing of $RN(a + b + c)$ is depicted on Fig. 1. Rounding to odd is not a rounding mode available on current architectures, hence a software emulation is proposed in [3]: this software emulation requires accesses to the binary representation of the floating-point numbers and conditional branches, both of which are costly on pipelined architectures.

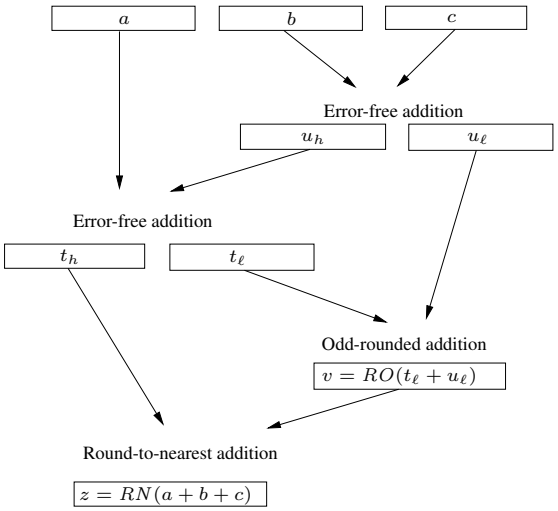


Figure 1. The Boldo-Melquiond algorithm.

In the next section, we propose a new algorithm for simulating the round-to-odd addition of two floating-point values. This algorithm uses only available IEEE-754 rounding modes and a multiplication by the constant 0.5, and can be used to avoid access to the binary representation of the floating-point numbers and conditional branches in the computation of $RN(a + b + c)$ with the Boldo-Melquiond algorithm. We also study a modified version of the Boldo-Melquiond algorithm to compute $DR(a+b+c)$, where DR denotes any of the IEEE-754 directed rounding modes.

4.1. A new method for rounding to odd

If we allow the multiplication by the constant 0.5 and choosing the rounding mode for each operation, the following algorithm can be used to implement the round-to-odd addition.

Algorithm 4 (OddRoundSum(a,b))**.**

$$\begin{array}{lcll} d & = & RD(a+b); & \\ u & = & RU(a+b); & \\ e' & = & RN(d+u); & \\ e & = & e' \times 0.5; & \textit{\{exact\}} \\ o' & = & u-e; & \textit{\{exact\}} \\ o & = & o'+d; & \textit{\{exact\}} \end{array}$$

Theorem 6. *Let us assume that no underflows nor overflows occur in intermediate operations. Given a and b two floating-point numbers, Algorithm 4 computes $o = RO(a+b)$.*

Proof. The desired result is either d or u. The remaining operations are to identify which of d and u is $a+b$ rounded to odd. □

Algorithm 4 can be used in the algorithm depicted on Fig. 1 to implement the round-to-odd addition. Then we obtain an algorithm using only basic floating-point operations and the IEEE-754 rounding modes to compute $RN(a+b+c)$ for all floating-point numbers a, b and c.

In Algorithm 4, note that d and u may be calculated in parallel and that the calculation of e and o' may be combined if a fused multiply-add (FMA) instruction is available. On most floating-point units, the rounding mode is dynamic and changing it requires to flush the pipeline, which is expensive. However, on some processors such as Intel's Itanium, the rounding mode of each floating-point operation can be chosen individually [4, Chap. 3]. In this case, choosing the rounding mode has no impact on the running time of a sequence of floating-point operations. Moreover the Itanium provides an FMA instruction, hence the proposed algorithm can be expected to be a very efficient alternative to compute round-to-odd additions on this processor.

4.2. Computation of $DR(a+b+c)$

We now focus on the problem of computing $DR(a+b+c)$, where DR denotes one of the directed rounding modes (RZ, RD or RU). Our algorithm is depicted on Fig. 2: it is a variant of the Boldo-Melquiond algorithm. The only difference is that the last two operations use a directed rounding mode. We will show that it computes $DR(a+b+c)$ for rounding downward or upward, but may give an incorrect answer for rounding to zero. In the sequel, we denote by s the exact sum $a+b+c$.

Theorem 7. *Let a, b and c be three numbers in a radix-2 floating-point system in precision $p \geq 3$ and DR be either RD (round toward $-\infty$) or RU (round toward $+\infty$). Assuming that no underflows nor overflows occur in intermediate operations, the algorithm given in Fig. 2 computes $DR(a+b+c)$.*

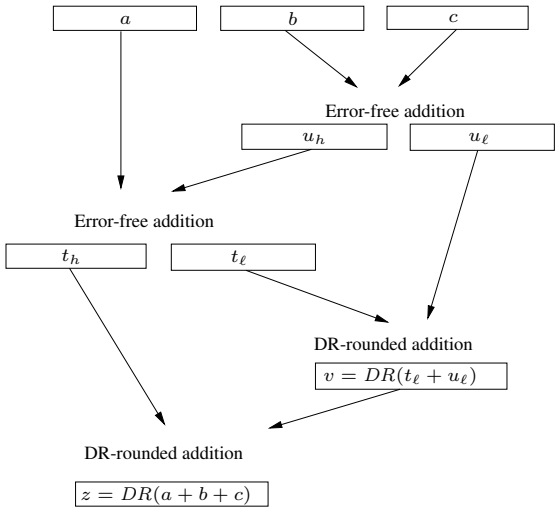


Figure 2. Algorithm to compute $DR(a+b+c)$ with $DR = RD$ or RU.

Proof. Without loss of generality, assume that $DR = RD$: the case $DR = RU$ is symmetric.

For $x \neq 0$, $\mathrm{ulp}(x)$ will denote the *unit in the last place* of x, i.e., if $2^{k-1} \leq |x| < 2^k$ then $\mathrm{ulp}(x) = 2^{k-p}$.

The error-free additions yield $s = a+b+c = t_h+t_\ell+u_\ell$. After the first rounding $v = RD(t_\ell + u_\ell)$, one gets $s' = t_h + v$, and the result is $z = RD(s')$. In general, these two roundings can be regarded as some kind of *double rounding*, which is known, in directed rounding, to be equivalent to a single rounding. The proof is based on the same idea.

We first need to exclude a particular case: If $a + u_h$ is exactly representable in precision p, then $t_\ell = 0$ and v is computed exactly ($v = u_\ell$), so that $z = RD(a+b+c)$. Now let us assume that $a + u_h$ is not exactly representable.

Let us show that $|t_\ell + u_\ell|$ is significantly smaller than $|t_h|$. First, $|u_h| \leq 2\ |t_h|$. Indeed, if a and u_h have the same sign, then $|u_h| \leq |t_h|$; otherwise since $a + u_h$ is not exactly representable, either $|a| < 1/2\ |u_h|$, in which case

$$|a + u_h| \geq |u_h| - |a| > 1/2\ |u_h|\,,$$

or $|a| > 2\ |u_h|$, in which case

$$|a + u_h| \geq |a| - |u_h| > |u_h|\,,$$

then $|t_h| \geq RN(1/2\ |u_h|) = 1/2\ |u_h|$. As a consequence, $|u_\ell| \leq 2^{-p}\,|u_h| \leq 2^{1-p}\,|t_h|$. Since $|t_\ell| \leq 2^{-p}\,|t_h|$ we have $|t_\ell + u_\ell| \leq 3 \cdot 2^{-p}\,|t_h| \leq 2^{2-p}\,|t_h|$.

Now let us show that $RD(s) \leq s'$. We have

$$|v| \leq RU(|t_\ell + u_\ell|) \leq RU(2^{2-p}\,|t_h|) = 2^{2-p}\,|t_h|\,,$$

hence $|s'| = |t_h + v| \geq |t_h| - |v| \geq (1 - 2^{2-p})\ |t_h|$, and

$$|t_\ell + u_\ell| \leq 2^{2-p}\,|t_h| \leq 2^{3-p}\,|s'|\,.$$

Since $p \geq 3$, it follows that $|t_\ell + u_\ell| \leq |s'|$. As a consequence, $\mathrm{ulp}(s')\,\mathbb{Z} \subset \mathrm{ulp}(t_\ell + u_\ell)\,\mathbb{Z}$ and $RD(s) \leq s'$.

Thus $RD(s) \leq RD(s')$. Moreover, since we have $s' = t_h + v \leq t_h + t_\ell + u_\ell = s$, it follows that $RD(s') \leq RD(s)$. Therefore $z = RD(s') = RD(a+b+c)$. □

Note that the proof cannot be extended to RZ, due to the fact that the two roundings can be done in opposite directions. For instance, if $s > 0$ (not exactly representable) and $t_\ell + u_\ell < 0$, then one has $RD(s) \leq RD(s')$ as wanted, but $t_\ell + u_\ell$ rounds upward and s' can be $RU(s)$, so that $z = RU(s)$ instead of $RZ(s) = RD(s)$, as shown on the following counter-example. In precision 7, with $a = -3616$, $b = 19200$ and $c = -97$, we have $s = 15487$, $RZ(s) = 15360$ and $RU(s) = 15488$. Running the algorithm depicted on Fig. 2 on this instance gives

$$\begin{aligned}(u_h, u_l) &= (19200, -97)\\ (t_h, t_l) &= (15616, -32)\\ v &= RZ(-129) = -128\\ z &= RZ(15488) = 15488\end{aligned}$$

and $RU(s)$ has been computed instead of $RZ(s)$.

Nevertheless $RZ(s)$ can be obtained by computing both $RD(s)$ and $RU(s)$, then selecting the one closer to zero thanks to the minNumMag instruction [7], as shown in the following algorithm.

Algorithm 5 (RZ3(a,b,c))**.**

$$\begin{aligned}(u_h, u_\ell) &= \mathrm{2Sum}(b, c);\\ (t_h, t_\ell) &= \mathrm{2Sum}(a, u_h);\\ v_d &= RD(u_\ell + t_\ell);\\ z_d &= RD(t_h + v_d);\\ v_u &= RU(u_\ell + t_\ell);\\ z_u &= RU(t_h + v_u);\\ z &= \mathrm{minNumMag}(z_d, z_u);\end{aligned}$$

This algorithm for computing $RZ(a+b+c)$ without branches can already be implemented on the Itanium architecture thanks to the famin instruction [4].

5. Conclusions

We have proved that Knuth's 2Sum algorithm is minimal, both in terms of the number of operations and the depth of the dependency graph. We have also shown that, just by performing round-to-nearest floating-point additions and subtractions without any testing, it is impossible to compute the round-to-nearest sum of $n \geq 3$ floating-point numbers. If changing the rounding mode is allowed, we can implement without testing the nonstandard *rounding to odd* defined by Boldo and Melquiond, which makes it indeed possible to compute the sum of three floating-point numbers rounded to nearest. We finally proposed an adaptation of the Boldo-Melquiond algorithm for calculating $a+b+c$ rounded according to the standard directed rounding modes.

6. Acknowledgement

We thank Damien Stehlé, who actively participated in our first discussions on these topics.

References

[1] American National Standards Institute and Institute of Electrical and Electronic Engineers. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. New York, 1985.

[2] American National Standards Institute and Institute of Electrical and Electronic Engineers. *IEEE Standard for Radix Independent Floating-Point Arithmetic, ANSI/IEEE Standard 854-1987*. New York, 1987.

[3] S. Boldo and G. Melquiond. Emulation of a FMA and correctly-rounded sums: proved algorithms using rounding to odd. *IEEE Transactions on Computers*, 57(4), Apr. 2008.

[4] M. Cornea, J. Harrison, and P. T. P. Tang. *Scientific computing on Itanium based systems*. Intel Press, 2002.

[5] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 3 1971.

[6] 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/`.

[7] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, Aug. 2008. Available at `http://ieeexplore.ieee.org/servlet/opac?punumber=4610933`.

[8] D. Knuth. *The Art of Computer Programming, 3rd edition*, volume 2. Addison-Wesley, Reading, MA, 1998.

[9] O. Møller. Quasi double-precision in floating-point addition. *BIT*, 5:37–50, 1965.

[10] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM TOPLAS*, 30(3):1–41, 2008. Available at `http://hal.archives-ouvertes.fr/hal-00128124`.

[11] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

[12] M. Pichat. Correction d'une somme en arithmétique à virgule flottante (in French). *Numerische Mathematik*, 19:400–406, 1972.

[13] D. Priest. *On Properties of Floating-Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. PhD thesis, University of California at Berkeley, 1992.

[14] J. R. Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete Computational Geometry*, 18:305–363, 1997.