



**HAL**  
open science

**Emulating round-to-nearest-ties-to-zero "augmented"  
floating-point operations using  
round-to-nearest-ties-to-even arithmetic**

Sylvie Boldo, Christoph Q. Lauter, Jean-Michel Muller

► **To cite this version:**

Sylvie Boldo, Christoph Q. Lauter, Jean-Michel Muller. Emulating round-to-nearest-ties-to-zero "augmented" floating-point operations using round-to-nearest-ties-to-even arithmetic. 2019. hal-02137968v2

**HAL Id: hal-02137968**

**<https://hal.science/hal-02137968v2>**

Preprint submitted on 15 Oct 2019 (v2), last revised 13 Mar 2020 (v4)

**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.

# Emulating Round-to-Nearest-Ties-to-Zero “Augmented” Floating-Point Operations Using Round-to-Nearest-Ties-to-Even Arithmetic

Sylvie Boldo\*, Christoph Lauter†, Jean-Michel Muller‡

\* Inria, LRI, CNRS & Univ. Paris-Sud, Université Paris-Saclay, F-91405 Orsay Cedex, France

† University of Alaska Anchorage, USA

‡ Univ Lyon, CNRS, ENS de Lyon, Inria, Université Claude Bernard Lyon 1, LIP UMR 5668, F-69007 Lyon, France

**Abstract**—The 2019 version of the IEEE 754 Standard for Floating-Point Arithmetic recommends that new “augmented” operations should be provided for the binary formats. These operations use a new “rounding direction”: round to nearest *ties-to-zero*. We show how they can be implemented using the currently available operations, using round-to-nearest *ties-to-even*.

**Keywords.** Floating-point arithmetic, Numerical reproducibility, Rounding error analysis, Error-free transforms, Rounding mode.

## I. INTRODUCTION AND NOTATION

The new IEEE 754-2019 Standard for Floating-Point Arithmetic [1] supersedes the 2008 version. It recommends that new “augmented” operations should be provided for the binary formats [14]. These operations are called **augmentedAddition**, **augmentedSubtraction**, and **augmentedMultiplication**. They use a new “rounding direction”: round to nearest *ties-to-zero*. The reason behind this recommendation is that these operations would significantly help to implement reproducible summation and dot product, using an algorithm due to Demmel, Ahrens, and NGuyen [5]. Obtaining very fast reproducible summation with that algorithm will certainly require a direct hardware implementation of these operations. However, having these operations available on common processors will certainly take time. The purpose of this paper is to show that, in the meantime, one can emulate these operations with conventional floating-point operations (with the usual round to nearest “ties to even” rounding direction), with reasonable efficiency.

In the following, we assume radix-2, precision- $p$  floating-point (FP) arithmetic [12]. The minimum floating-point exponent is  $e_{\min}$ , so that  $2^{e_{\min}}$  is the smallest positive normal number and  $2^{e_{\min}-p+1}$  is the smallest positive floating-point number. The maximum floating-point exponent is  $e_{\max}$ . The largest positive floating-point number is  $\Omega = (2 - 2^{-p+1}) \cdot 2^{e_{\max}}$ . We will assume

$$3p \leq e_{\max}, \quad (1)$$

which is satisfied by all formats of the IEEE 754 Standard. The usual round to nearest, ties-to-even function (which is the default in the IEEE-754 Standard) will be noted  $\text{RN}_e$ . We recall its definition [1]:

$\text{RN}_e(t)$  (where  $t$  is a real number) is the floating-point number nearest to  $t$ . If the two nearest floating-point numbers bracketing  $t$  are equally near,  $\text{RN}_e(t)$  is the one whose least significant bit is zero. If  $|t| \geq \Omega + 2^{e_{\max}-p}$  then  $\text{RN}_e(t) = \infty$ , with the same sign as  $t$ .

We will also assume that an FMA (fused multiply-add) instruction is available. This is the case on all recent floating-point units.

As said above, the new recommended operations use a new “rounding direction”: round to nearest *ties-to-zero*. It corresponds to the rounding function  $\text{RN}_0$  defined as follows [1]:

$\text{RN}_0(t)$  (where  $t$  is a real number) is the floating-point number nearest  $t$ . If the two nearest floating-point numbers bracketing  $t$  are equally near,  $\text{RN}_0(t)$  is the one with smaller magnitude. If  $|t| > \Omega + 2^{e_{\max}-p}$  then  $\text{RN}_0(t) = \infty$ , with the same sign as  $t$ .

This is illustrated in Fig. 1. As one can infer from the definitions,  $\text{RN}_e(t)$  and  $\text{RN}_0(t)$  can differ in only two circumstances (called *halfway cases* in the following): when  $t$  is halfway between two consecutive floating-point numbers, and when  $t = \pm(\Omega + 2^{e_{\max}-p})$ .

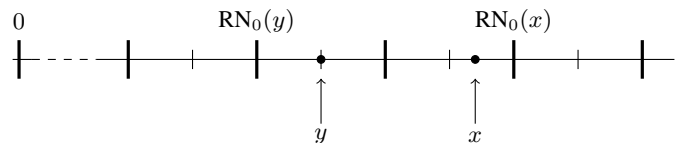


Fig. 1. Round to nearest ties-to-zero (assuming we are in the positive range). Number  $x$  is rounded to the (unique) FP number nearest to  $x$ . Number  $y$  is a halfway case: it is exactly halfway between two consecutive FP numbers: it is rounded to the one that has the smallest magnitude.

The augmented operations are required to behave as follows [1], [14]:

- **augmentedAddition**( $x, y$ ) delivers  $(a_0, b_0)$  such that  $a_0 = \text{RN}_0(x + y)$  and, when  $a_0 \notin \{\pm\infty, \text{NaN}\}$ ,  $b_0 = (x + y) - a_0$ . When  $b_0 = 0$ , it is required to have the same sign as  $a_0$ . One easily shows that  $b_0$  is a floating-point number. For special rules when  $a_0 \in \{\pm\infty, \text{NaN}\}$ , see [14];

- **augmentedSubtraction**( $x, y$ ) is exactly the same as **augmentedAddition**( $x, -y$ ), so we will not discuss that operation further;
- **augmentedMultiplication**( $x, y$ ) delivers  $(a_0, b_0)$  such that  $a_0 = \text{RN}_0(x \cdot y)$  and, where  $a_0 \notin \{\pm\infty, \text{NaN}\}$ ,  $b_0 = \text{RN}_0((x \cdot y) - a_0)$ . When  $b_0 = 0$ , it is required to have the same sign as  $a_0$ . Note that in some corner cases (an example is given in Section IV-A),  $b_0$  may differ from  $(x \cdot y) - a_0$  (in other words,  $(x \cdot y) - a_0$  is not always a floating-point number). Again, rules for handling infinities, NaNs and the signs of zeroes are given in [1], [14].

Because of the different rounding function, these augmented operations differ from the well-known Fast2Sum, 2Sum, and Fast2Mult algorithms (Algorithms 1, 2 and 3 below). As said above, the goal of this paper is to show that one can implement these augmented operations just by using rounded-to-nearest-even floating-point operations and with reasonable efficiency on a system compliant with IEEE 754-2008,

Let  $t$  be the exact sum  $x + y$  (if we consider implementing **augmentedAddition**) or the exact product  $x \cdot y$  (if we consider implementing **augmentedMultiplication**). To implement the augmented operations, in the general case (i.e., the sum or product does not overflow, and in the case of **augmentedMultiplication**, the floating-point exponents  $e_x$  and  $e_y$  of  $x$  and  $y$  satisfy  $e_x + e_y \geq e_{\min} + p - 1$ ), we first use the classical Fast2Sum, 2Sum, or Fast2Mult algorithms to generate two floating-point numbers  $a_e$  and  $b_e$  such that  $a_e = \text{RN}_e(t)$  and  $b_e = t - a_e$ . We explain how **augmentedAddition**( $x, y$ ) and **augmentedMultiplication**( $x, y$ ) can be obtained from  $a_e$  and  $b_e$  in Sections III and IV, respectively, using a “recomposition” algorithm presented in Section II.

In the following, we need to use a definition inspired from Harrison’s definition [6] of function  $\text{ulp}$  (“unit in the last place”). If  $x$  is a floating-point number different from  $-\Omega$ , first define  $\text{pred}(x)$  as the floating-point predecessor of  $x$ , i.e., the largest floating-point number  $< x$ . We define  $\text{ulp}_H(x)$  as follows.

**Definition 1** (Harrison’s  $\text{ulp}$ ). *If  $x$  is a floating-point number, then  $\text{ulp}_H(x)$  is*

$$|x| - \text{pred}(|x|).$$

Notation  $\text{ulp}_H$  is to avoid confusion with the usual definition of function  $\text{ulp}$ . The usual  $\text{ulp}$  and function  $\text{ulp}_H$  differ at powers of 2, except in the subnormal domain. For instance,  $\text{ulp}(1) = 2^{-p+1}$ , whereas  $\text{ulp}_H(1) = 2^{-p}$ . One easily checks that if  $|t|$  is not a power of 2, then  $\text{ulp}(t) = \text{ulp}_H(t)$ , and if  $|t| = 2^k$ , then  $\text{ulp}(t) = 2^{k-p+1} = 2\text{ulp}_H(t)$ , except in the subnormal range where  $\text{ulp}(t) = \text{ulp}_H(t) = 2^{e_{\min}-p+1}$ .

The reason for choosing function  $\text{ulp}_H$  instead of function  $\text{ulp}$  is twofold:

- if  $t > 0$  is a real number, each time  $\text{RN}_0(t)$  differs from  $\text{RN}_e(t)$ ,  $\text{RN}_0(t)$  will be the floating-point predecessor of  $\text{RN}_e(t)$ , because  $\text{RN}_0(t) \neq \text{RN}_e(t)$  implies that  $t$  is a halfway case: it is exactly halfway between two

consecutive floating-point numbers, and in that case,  $\text{RN}_0(t)$  is the one of these two FP numbers which is closest to zero and  $\text{RN}_e(t)$  is the other one. Hence, in these cases, to obtain  $\text{RN}_0(t)$  we will have to subtract from  $\text{RN}_e(t)$  a number which is exactly  $\text{ulp}_H(\text{RN}_e(t))$  (for negative  $t$ , for symmetry reasons, we will have to add  $\text{ulp}_H(\text{RN}_e(t))$  to  $\text{RN}_e(t)$ ); and

- there is a very simple algorithm for computing  $\text{ulp}_H(t)$  in the range where we need it (Algorithm 4 below).

Let us now briefly recall the classical Algorithms Fast2Sum, 2Sum, and Fast2Mult.

---

**ALGORITHM 1: Fast2Sum**( $x, y$ ). The Fast2Sum algorithm [4].

---

$$\begin{aligned} a_e &\leftarrow \text{RN}_e(x + y) \\ y' &\leftarrow \text{RN}_e(a_e - x) \\ b_e &\leftarrow \text{RN}_e(y - y') \end{aligned}$$


---

If  $x = 0$  or  $y = 0$ , or if the floating-point exponents  $e_x$  and  $e_y$  satisfy  $e_x \geq e_y$ , then the two variables  $a_e$  and  $b_e$  returned by Algorithm 1 (Fast2Sum) satisfy  $a_e + b_e = x + y$ . Hence,  $b_e$  is the error of the floating-point addition  $a_e \leftarrow \text{RN}_e(x + y)$ . Another property that will be useful in Section IV-C is that  $y' = a_e - x$  (i.e., there is no rounding error at line 2 of the algorithm, see for instance [12] for a proof). In practice, condition “ $e_x \geq e_y$ ” may be hard to check. However, if  $|x| \geq |y|$  then that condition is satisfied. Algorithm 1 is immune to spurious overflow: it was proved in [2] that if the addition  $\text{RN}_e(x + y)$  does not overflow then the other two operations cannot overflow.

---

**ALGORITHM 2: 2Sum**( $x, y$ ). The 2Sum algorithm [11], [10].

---

$$\begin{aligned} a_e &\leftarrow \text{RN}_e(x + y) \\ x' &\leftarrow \text{RN}_e(a_e - y) \\ y' &\leftarrow \text{RN}_e(a_e - x') \\ \delta_x &\leftarrow \text{RN}_e(x - x') \\ \delta_y &\leftarrow \text{RN}_e(y - y') \\ b_e &\leftarrow \text{RN}_e(\delta_x + \delta_y) \end{aligned}$$


---

Algorithm 2 (2Sum) gives the same results as Algorithm 1, but without any requirement on the exponents of  $x$  and  $y$ . It is *almost* immune to spurious overflow: if  $|x| \neq \Omega$  and the addition  $\text{RN}_e(x + y)$  does not overflow then the other five operations cannot overflow [2].

Let  $x$  and  $y$  be two floating-point numbers, with exponents  $e_x$  and  $e_y$ , such that  $e_x + e_y \geq e_{\min} + p - 1$ . Define  $a_e = \text{RN}_e(x \cdot y)$ . The number  $b_e = x \cdot y - a_e$  is a floating-point number (see [13] for a proof). An immediate consequence is that Algorithm 3 (Fast2Mult) delivers these numbers  $a_e$  and  $b_e$ . Checking if  $e_x + e_y \geq e_{\min} + p - 1$  may be difficult, however, a sufficient condition for that is  $|\text{RN}_e(x \cdot y)| \geq (1 - 2^{-p}) \cdot 2^{e_{\min} + p}$ .

We will also use the following results, due to Hauser [7] and Sterbenz [16] (the proofs are straightforward, see [12]).

---

**ALGORITHM 3: Fast2Mult**( $x, y$ ). The Fast2Mult algorithm (see for instance [9], [13], [12]). It requires the availability of a fused multiply-add (FMA) instruction for computing  $\text{RN}_e(x \cdot y - a_e)$ .

---

$a_e \leftarrow \text{RN}_e(x \cdot y)$   
 $b_e \leftarrow \text{RN}_e(x \cdot y - a_e)$

---

**Lemma 1** (Hauser). *If  $x$  and  $y$  are floating-point numbers, and if the number  $\text{RN}_e(x + y)$  is subnormal, then  $x + y$  is a floating-point number, which implies  $\text{RN}_e(x + y) = x + y$ .*

**Lemma 2** (Sterbenz). *If  $x$  and  $y$  are floating-point numbers that satisfy  $x/2 \leq y \leq 2x$ , then  $x - y$  is a floating-point number, which implies  $\text{RN}_e(x - y) = x - y$ .*

As said above, when  $\text{RN}_0(t)$  and  $\text{RN}_e(t)$  differ,  $\text{RN}_0(t)$  is obtained by subtracting  $\text{sign}(t) \cdot \text{ulp}_H(\text{RN}_e(t))$  from  $\text{RN}_e(t)$ . Therefore, we need to be able to compute  $\text{sign}(a) \cdot \text{ulp}_H(a)$ . If  $|a| > 2^{e_{\min}}$ , this can be done using Algorithm 4 below, which is a variant of an algorithm introduced by Rump [15].

---

**ALGORITHM 4:** Computing  $\text{sign}(a) \cdot \text{ulp}_H(a)$  for  $|a| > 2^{e_{\min}}$ . Uses the FP constant  $\psi = 1 - 2^{-p}$ .

---

$z \leftarrow \text{RN}_e(\psi a)$   
 $\delta \leftarrow \text{RN}_e(a - z)$   
**return**  $\delta$

---

The fact that Algorithm 4 returns  $\text{sign}(a) \cdot \text{ulp}_H(a)$  when  $|a| > 2^{e_{\min}}$  is a direct consequence of [15, Lemma 3.6]. See also [8]. Note that when  $a > 2^{e_{\min}}$ ,  $z$  equals  $\text{pred}(a)$ . If  $a$  is subnormal or zero (i.e.,  $|a| < 2^{e_{\min}}$ ), then Algorithm 4 returns 0. Interestingly enough, Algorithm 4 almost always returns the same result if we change the tie-breaking rule: the only exception is  $|a| = 2^{e_{\min}}$ , for which  $\delta = 0$  if the rounding function is  $\text{RN}_e$ , and  $\delta = 2^{e_{\min} - p + 1}$  if the rounding function is  $\text{RN}_0$ . Another remark is that the fact that the radix is 2 is important here (a counterexample in radix 10 is  $p = 3$  and  $a = 101$ ). This means that our work cannot be straightforwardly generalized to decimal floating-point arithmetic.

## II. RECOMPOSITION

In this section, we start from two floating-point numbers  $a_e$  and  $b_e$ , that satisfy  $a_e = \text{RN}_e(t)$ , with  $t = a_e + b_e$ , and we assume  $|a_e| > 2^{e_{\min}}$ . These numbers may have been preliminarily generated by the 2Sum, Fast2Sum or Fast2Mult algorithms. We want to obtain two floating-point numbers  $a_0$  and  $b_0$  such that  $a_0 = \text{RN}_0(t)$  and  $a_0 + b_0 = t$ .

One easily notes that  $a_e \neq \text{RN}_0(t)$  only when  $b_e = -\frac{1}{2}\text{sign}(a_e) \cdot \text{ulp}_H(a_e)$ . In that case,

$$\text{RN}_0(t) = a_e - \text{sign}(a_e)\text{ulp}_H(a_e),$$

and

$$t - \text{RN}_0(t) = -b_e.$$

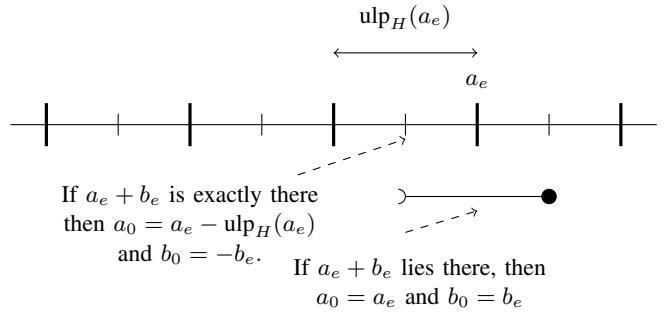


Fig. 2. Illustration of the transformation to be performed in the case  $a_e + b_e > 0$  (the case  $a_e + b_e < 0$  is symmetrical). The thick vertical lines represent the floating-point numbers. The numbers  $a_e$  and  $b_e$  may have been previously obtained using 2Sum, Fast2Sum, or Fast2Mult.

---

**ALGORITHM 5: Recomp**( $a_e, b_e$ ). From two FP numbers  $a_e$  and  $b_e$  such that  $a_e = \text{RN}_e(a_e + b_e)$  and  $|a_e| > 2^{e_{\min}}$ , computes  $a_0$  and  $b_0$  such that  $a_0 + b_0 = a_e + b_e$  and  $a_0 = \text{RN}_0(a_e + b_e)$ . Uses the FP constant  $\psi = 1 - 2^{-p}$ .

---

$z \leftarrow \text{RN}_e(\psi \cdot a_e)$   
 $\delta \leftarrow \text{RN}_e(z - a_e)$   
**if**  $2 \cdot b_e = \delta$  **then**  
 $a_0 \leftarrow z$   
 $b_0 \leftarrow -b_e$   
**else**  
 $a_0 \leftarrow a_e$   
 $b_0 \leftarrow b_e$   
**end if**  
**return** ( $a_0, b_0$ )

---

This is illustrated by Figure 2, and this leads to Algorithm 5 below.

In Algorithm 5, when  $2 \cdot b_e = \delta$ , we must return  $a_0 = a_e - \delta$ . Lemma 2 applied to the second line of the algorithm implies  $\delta = z - a_e$ . This explains why in that case the value of  $a_0$  returned by the algorithm is  $z$ .

Note that if  $|a_e| \leq 2^{e_{\min}}$ , Algorithm 5 always returns  $a_0 = a_e$  and  $b_0 = b_e$ . This is not a problem for augmentedAddition thanks to Lemma 1, as we are going to see in Section III. For augmentedMultiplication this will require a special handling (see Sections IV-C and IV-D).

In the next two sections, we examine how Algorithm 5 can be used to compute augmentedAddition( $x, y$ ) and augmentedMultiplication( $x, y$ ).

## III. USE OF ALGORITHM RECOMP FOR IMPLEMENTING AUGMENTEDADDITION

From two input floating-point numbers  $x$  and  $y$ , we wish to compute  $\text{RN}_0(x + y)$  and  $(x + y) - \text{RN}_0(x + y)$ . Let us first give a simple algorithm (Algorithm 6, below) that returns a correct result when no exception occurs (i.e, the returned values are finite floating-point numbers).

---

**ALGORITHM 6:** AA-Simple( $x, y$ ): computes augmentedAddition( $x, y$ ) when no exception occurs.

---

```

1: if  $|y| > |x|$  then
2:   swap( $x, y$ )
3: end if
4:  $(a_e, b_e) \leftarrow$  Fast2Sum( $x, y$ )
5:  $(a_0, b_0) \leftarrow$  Recomp( $a_e, b_e$ )
6: return  $(a_0, b_0)$ 

```

---

**Theorem 1.** *The values  $a_0$  and  $b_0$  returned by Algorithm 6 satisfy:*

- 1) if  $a_0$  and  $b_0$  are finite numbers then  $(a_0, b_0) = \text{augmentedAddition}(x, y)$ ;
- 2) when  $x + y = 0$ ,  $a_0$  and  $b_0$  are equal to zero too (as expected), but possibly with signs that differ from the ones specified in the standard;
- 3) if  $|x+y| = \Omega + 2^{e_{\max}-p} = (2-2^{-p}) \cdot 2^{e_{\max}}$  then  $a_0 = \pm\infty$  and  $b_0$  is  $\pm\infty$  (with a sign different from the one of  $a_0$ ), whereas the correct values would have been  $a_0 = \pm\Omega$  and  $b_0 = \pm 2^{e_{\max}-p}$  (with the appropriate signs);
- 4) if  $|x+y| > \Omega + 2^{e_{\max}-p}$  then  $a_0 = \pm\infty$  (with the appropriate sign) and  $b_0$  is either NaN or  $\pm\infty$  (possibly with a wrong sign), whereas the standard requires  $a_0 = b_0 = \infty$  (with the same sign as  $x + y$ ).

The first item in Theorem 1 is an immediate consequence of the properties of the Fast2Sum and Recomp algorithms. More precisely: we have  $a_e = \text{RN}_e(x + y)$  and  $a_e + b_e = x + y$ . Hence,

- if  $|a_e| > 2^{e_{\min}}$  then Recomp( $a_e, b_e$ ) gives the expected result;
- if  $|a_e| \leq 2^{e_{\min}}$  then from Lemma 1, we know that the floating-point addition of  $x$  and  $y$  is exact, hence  $b_e = 0$ . We easily deduce that Recomp( $a_e, b_e$ ) =  $(a_e, b_e)$  which is the expected result. In particular, if  $a_e = 0$  then we obtain  $a_0 = b_0 = 0$  (possibly with wrong signs, as indicated in the second item in Theorem 1, see below for an explanation).

Note that if we are certain that  $|x| \neq \Omega$  (so that 2Sum( $x, y$ ) can be called without any risk of spurious overflow) we can replace lines 1 to 4 of the algorithm by a simple call to 2Sum( $x, y$ ).

Now, consider the second item in Theorem 1. Note that Lemma 1 implies that  $x+y = 0$  and  $\text{RN}_e(x+y) = 0$  are equivalent. In that case, the standard requires that  $a_0 = \text{RN}_0(x+y)$  should be  $+0$  except when  $x = y = -0$  (and in that case,  $a_0$  should be  $-0$ ), and that  $b_0$  should be equal to  $a_0$  [14]. However, the signs of the zero values delivered by Algorithm 6 may differ from these specifications:

- if  $(x = -y$  and  $|x| \neq 0)$  or  $(x = -0$  and  $y = +0)$  or  $(x = +0$  and  $y = +0)$  then Algorithm 6 returns  $a_0 = +0$  and  $b_0 = -0$ , whereas the desired result is  $a_0 = b_0 = +0$ ;
- if  $x = +0$  and  $y = -0$  then Algorithm 6 returns the desired result, namely  $a_0 = b_0 = +0$  (note that if we

replace Fast2Sum by 2Sum in the algorithm, we obtain  $a_0 = +0$  and  $b_0 = -0$ );

- if  $x = -0$  and  $y = -0$  then Algorithm 6 returns  $a_0 = -0$  and  $b_0 = +0$ , whereas the desired result is  $a_0 = b_0 = -0$  (note that if we replace Fast2Sum by 2Sum in the algorithm, we obtain  $a_0 = b_0 = -0$ ).

Hence, if the signs of the zero variables matter in the target application, one has to add the following lines to Algorithm 6 after Line 5:

```

if  $b_0 = 0$  then
   $b_0 \leftarrow (+0) \times a_0$ 
end if

```

The third item in Theorem 1 follows immediately by applying Algorithm 6 to the corresponding input value.

Concerning the 4th item in Theorem 1, Table I gives the values returned by Algorithm 6 when  $x + y > \Omega + 2^{e_{\max}-p}$  (the case  $x + y < -\Omega - 2^{e_{\max}-p}$  is symmetrical).

TABLE I  
VALUES OBTAINED USING ALGORITHM 6 (POSSIBLY WITH A REPLACEMENT OF FAST2SUM BY 2SUM) WHEN  $x + y > 2^{e_{\max}}(2 - 2^{-p})$  (RESP. ALGORITHM 8 WHEN  $x \cdot y > 2^{e_{\max}}(2 - 2^{-p})$ ). THE CASE WHERE  $x + y$  (RESP.  $x \cdot y$ ) IS NEGATIVE IS SYMMETRICAL.

	$(a_e, b_e)$ obtained through 2Sum	$(a_e, b_e)$ obtained through Fast2Sum	$(a_e, b_e)$ obtained through Fast2Mult	Result required by the standard
$a_0$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$b_0$	NaN	$-\infty$	$-\infty$	$+\infty$

If the considered applications only require augmentedAddition to follow the specifications when no exception occurs, Algorithm 6 (possibly with the above given additional lines if the signs of zeros matter) is a good candidate. If we wish to always follow the specifications, we suggest using Algorithm 7 below.

**Theorem 2.** *The output  $(a_0, b_0)$  of Algorithm 7 is equal to augmentedAddition( $x, y$ ).*

We just give a sketch of the proof.

*Proof.*

- when  $b_0 \neq 0$  at Line 6 of the algorithm and  $a_e \neq \pm\infty$ , Algorithm 7 behaves exactly as Algorithm 6. A quick look at Algorithm 1 shows that if  $a_e = \pm\infty$  then  $b_0 = \pm\infty$  too;
- we have just explained the case  $a_0 = 0$  before;
- when  $a_e = \pm\infty$ , there are two possibilities (as discussed in cases 3 and 4 of Theorem 1): either  $|x + y| = \Omega + 2^{e_{\max}-p} = (2 - 2^{-p}) \cdot 2^{e_{\max}}$ , in which case we must return  $a_0 = \pm\Omega$  and  $b_0 = \pm 2^{e_{\max}-p}$  (with the appropriate signs), or  $|x + y| > \Omega + 2^{e_{\max}-p}$ , in which case we must return  $a_0 = b_0 = \pm\infty$  (with the appropriate sign, namely the sign of  $a_e$ ). This issue is dealt with at Lines 8 to 16 of Algorithm 7: we divide  $x$  and  $y$  by 2 so that if  $|x + y| = \Omega + 2^{e_{\max}-p}$ , then  $x/2 + y/2$  is computed by Fast2Sum without overflow, which makes it possible to compare it with  $\pm(\Omega + 2^{e_{\max}-p})/2$ .

---

**ALGORITHM 7:** AA-Full( $x, y$ ): computes augmentedAddition( $x, y$ ) in all cases.

---

```

1: if  $|y| > |x|$  then
2:   swap( $x, y$ )
3: end if
4:  $(a_e, b_e) \leftarrow \text{Fast2Sum}(x, y)$ 
5:  $(a_0, b_0) \leftarrow \text{Recomp}(a_e, b_e)$ 
6: if  $b_0 = 0$  then
7:    $b_0 \leftarrow (+0) \times a_0$ 
8: else if  $|a_e| = +\infty$  then
9:    $(a'_e, b'_e) \leftarrow \text{Fast2Sum}(0.5x, 0.5y)$ 
10:  if  $(a'_e = 2^{e_{\max}}$  and  $b'_e = -2^{e_{\max}-p-1})$  or
     $(a'_e = -2^{e_{\max}}$  and  $b'_e = +2^{e_{\max}-p-1})$  then
11:     $a_0 \leftarrow \text{RN}_e(a'_e \cdot (2 - 2^{-p+1}))$ 
12:     $b_0 \leftarrow -2b'_e$ 
13:  else
14:     $a_0 \leftarrow a_e$  (infinity with right sign)
15:     $b_0 \leftarrow a_e$ 
16:  end if
17: end if
18: return  $(a_0, b_0)$ 

```

---

□

#### IV. USE OF ALGORITHM RECOMP FOR IMPLEMENTING AUGMENTEDMULTIPLICATION

##### A. General case

From two input floating-point numbers  $x$  and  $y$ , we wish to compute  $\text{RN}_0(x \cdot y)$  and  $x \cdot y - \text{RN}_0(x \cdot y)$  (or, merely,  $\text{RN}_0[x \cdot y - \text{RN}_0(x \cdot y)]$  when  $x \cdot y - \text{RN}_0(x \cdot y)$  is not a floating-point number). As we did for augmentedAddition, let us first present a simple algorithm. Unfortunately, it will be less general than the simple addition algorithm: this is due to the fact that when the absolute value of the product of two floating-point numbers is less than or equal to  $2^{e_{\min}+p}$ , it may not be exactly representable by the sum of two floating-point numbers (an example is  $x = 1 + 2^{-p+1}$  and  $y = 2^{e_{\min}} + 2^{e_{\min}-p+1}$ ; their product  $2^{e_{\min}} + 2^{e_{\min}-p+2} + 2^{e_{\min}-2p+2}$  cannot be a sum of two FP numbers, since such a sum is necessarily a multiple of  $2^{e_{\min}-p+1}$ ).

---

**ALGORITHM 8:** AM-Simple( $x, y$ ): computes augmentedMultiplication( $x, y$ ) when  $2^{e_{\min}+p} < |\text{RN}_e(x \cdot y)| < +\infty$ .

---

```

1:  $(a_e, b_e) \leftarrow \text{Fast2Mult}(x, y)$ 
2:  $(a_0, b_0) \leftarrow \text{Recomp}(a_e, b_e)$ 
3: return  $(a_0, b_0)$ 

```

---

**Theorem 3.** *If  $2^{e_{\min}+p} < |\text{RN}_e(x \cdot y)| < +\infty$  (i.e.,  $2^{e_{\min}+p} + 2^{e_{\min}+1} \leq |\text{RN}_e(x \cdot y)| \leq \Omega$ ) then the output  $(a_0, b_0)$  of Algorithm 8 is equal to augmentedMultiplication( $x, y$ ).*

*Proof.* If  $2^{e_{\min}+p} + 2^{e_{\min}+1} \leq |\text{RN}_e(x \cdot y)| \leq \Omega$  then we know that

- $(a_e, b_e) = \text{Fast2Mult}(x, y)$  gives  $a_e + b_e = x \cdot y$ ;
- $|a_e| > 2^{e_{\min}}$ ;

therefore  $\text{Recomp}(a_e, b_e)$  returns the expected result. □

The lower bound  $2^{e_{\min}+p} + 2^{e_{\min}+1}$  in Theorem 3 comes from the fact that if  $|\text{RN}_e(x \cdot y)|$  is below that value,  $\text{Fast2Mult}(x, y)$  may not deliver a correct result.

As for the addition algorithm, when  $b_0 = 0$ , it may have the wrong sign. Again, if the signs of the zero variables matter in the target application, one has to add the following lines to Algorithm 8 after Line 2:

```

if  $b_0 = 0$  then
   $b_0 \leftarrow (+0) \times a_0$ 
end if

```

Let us now examine how the cases  $\text{RN}_e(x \cdot y) = \pm\infty$  and  $|\text{RN}_e(x \cdot y)| \leq 2^{e_{\min}+p}$  can be addressed.

##### B. First special case: if $\text{RN}_e(x \cdot y) = \pm\infty$

In this case, in a way very similar to what we did for augmented addition,

- either  $|x \cdot y| = \Omega + 2^{e_{\max}-p} = (2 - 2^{-p}) \cdot 2^{e_{\max}}$ , in which case we must return  $a_0 = \pm\Omega$  and  $b_0 = \pm 2^{e_{\max}-p}$  (with the appropriate signs), whereas one easily checks that Algorithm 8 delivers a wrong result;
- or  $|x \cdot y| > \Omega + 2^{e_{\max}-p}$ , in which case we must return  $a_0 = b_0 = \pm\infty$ , whereas Table I shows that Algorithm 8 delivers a wrong result for  $b_0$ .

The problem is addressed easily. It suffices to compute  $(a'_e, b'_e) = \text{Fast2Mult}(0.5 \cdot x, y)$ . If  $|x \cdot y| = \Omega + 2^{e_{\max}-p}$ , then  $x \cdot y/2$  is computed by  $\text{Fast2Mult}$  without overflow, which makes it possible to compare it with  $\pm(\Omega + 2^{e_{\max}-p})/2$ . If it turns out that  $|x \cdot y/2| \neq (\Omega + 2^{e_{\max}-p})/2$  we must return  $a_0 = b_0 = \text{RN}_e(x \cdot y)$ .

The case  $|\text{RN}_e(x \cdot y)| \leq 2^{e_{\min}+p}$  is more complex. We will separately examine the case  $|\text{RN}_e(x \cdot y)| \leq 2^{e_{\min}+1} - 2^{e_{\min}-p+1}$  (for which  $b_0$  is always zero) and the case  $2^{e_{\min}+1} \leq |\text{RN}_e(x \cdot y)| \leq 2^{e_{\min}+p}$ .

##### C. Second special case: if $|\text{RN}_e(x \cdot y)| \leq 2^{e_{\min}+1} - 2^{e_{\min}-p+1}$

In that case,  $|x \cdot y - \text{RN}_0(x \cdot y)| \leq 2^{e_{\min}-p}$  and thus  $\text{RN}_0(x \cdot y - \text{RN}_0(x \cdot y)) = 0$ , so we only have to focus on the computation of  $\text{RN}_0(x \cdot y)$ . We also assume that  $\text{RN}_e(x \cdot y) \neq 0$  (otherwise, it suffices to return the pair  $(0, 0)$ ). We therefore have

$$2^{e_{\min}-p} < |x \cdot y| < 2^{e_{\min}+1} - 2^{e_{\min}-p}. \quad (2)$$

Let  $a_e$  be  $\text{RN}_e(x \cdot y)$ , and let us successively compute (using FMA instructions)

$$\begin{aligned}
t_1 &= \text{RN}_e(x \cdot y \cdot 2^{2p}) \\
t_2 &= \text{RN}_e(x \cdot y \cdot 2^{2p} - t_1) = x \cdot y \cdot 2^{2p} - t_1 \\
t_3 &= \text{RN}_e(t_1 - a_e \cdot 2^{2p}).
\end{aligned}$$

One easily checks that (1) implies that  $t_1$  can be computed without overflow. Let us show that  $\theta_3 = t_1 - a_e \cdot 2^{2p}$

is a floating-point number. This will imply  $t_3 = \theta_3 = t_1 - a_e \cdot 2^{2p}$  (hence,  $\theta_3$  can be computed with an FMA, or with a multiplication followed by a subtraction). Note that (2) implies  $|2^{2p}x \cdot y| < 2^{e_{\min}+2p+1} - 2^{e_{\min}+p}$ , so that  $|t_1| \leq 2^{e_{\min}+2p+1} - 2^{e_{\min}+p+1}$  and  $\text{ulp}(t_1) \leq 2^{e_{\min}+p+1}$ . Also, we have  $|x \cdot y \cdot 2^{2p}| > 2^{e_{\min}+p}$ , which implies  $|t_1| \geq 2^{e_{\min}+p}$ .

Finally, since  $a_e$  is a multiple of  $2^{e_{\min}-p+1}$ , the number  $2^{2p} \cdot a_e$  is a multiple of  $2^{e_{\min}+p+1}$ . Therefore,  $\theta_3$  is a multiple of  $\text{ulp}(t_1)$ .

Now, from  $x \cdot y - 2^{e_{\min}-p} \leq |a_e| \leq x \cdot y + 2^{e_{\min}-p}$ , we deduce  $x \cdot y \cdot 2^{2p} - 2^{e_{\min}+p} \leq |a_e| \cdot 2^{2p} \leq x \cdot y \cdot 2^{2p} + 2^{e_{\min}+p}$ , which implies

$$t_1 - \frac{1}{2}\text{ulp}(t_1) - 2^{e_{\min}+p} \leq |a_e| \cdot 2^{2p} \leq t_1 + \frac{1}{2}\text{ulp}(t_1) + 2^{e_{\min}+p},$$

so that

$$|t_1 - a_e \cdot 2^{2p}| \leq \frac{1}{2}\text{ulp}(t_1) + 2^{e_{\min}+p} \leq \frac{1}{2}\text{ulp}(t_1) + |t_1|.$$

Hence,  $\theta_3$  is a multiple of  $\text{ulp}(t_1)$  of magnitude less than or equal to  $\frac{1}{2}\text{ulp}(t_1) + |t_1|$ . An immediate consequence is that  $\theta_3$  is a floating-point number, which implies  $t_3 = \theta_3$ .

Now, we wish to compute  $a_0 = \text{RN}_0(x \cdot y)$ . If  $x \cdot y = a_e - \text{sign}(a_e) \cdot 2^{e_{\min}-p}$  then  $a_0 = a_e - \text{sign}(a_e) \cdot 2^{e_{\min}-p+1}$  (computed without error), otherwise  $a_0 = a_e$ . Hence we have to decide whether  $x \cdot y = a_e - \text{sign}(a_e) \cdot 2^{e_{\min}-p}$ . This is equivalent to checking if  $t_2 + t_3 = -\text{sign}(a_e) \cdot 2^{e_{\min}+p}$ . This can be done as follows: first note that since  $t_3$  is a multiple of  $\text{ulp}(t_1)$  and  $|t_2| \leq \frac{1}{2}\text{ulp}(t_1)$ , either  $t_3 = 0$  or  $|t_3| > |t_2|$ . In any case, it follows from the properties of Algorithm 1 (Fast2Sum) that checking if

$$t_2 + t_3 = -\text{sign}(a_e) \cdot 2^{e_{\min}+p}$$

is equivalent to checking if

$$\begin{aligned} z := \text{RN}_e(t_2 + t_3) &= -\text{sign}(a_e) \cdot 2^{e_{\min}+p} \\ &\text{and} \\ \text{RN}_e(z - t_3) &= t_2. \end{aligned}$$

*D. Last special case: if  $2^{e_{\min}+1} \leq |\text{RN}_e(x \cdot y)| \leq 2^{e_{\min}+p}$*

In that case, we know that  $x \cdot y - \text{RN}_0(x \cdot y)$  is of magnitude less than or equal to  $2^{e_{\min}}$ , but is not necessarily a floating-point number. The standard requires that we return  $\text{RN}_0(x \cdot y)$  and  $\text{RN}_0(x \cdot y - \text{RN}_0(x \cdot y))$ .

First, we apply Algorithm 8 to the product  $(2^p x) \cdot y$ . One easily checks that (1) implies that  $2^p x$  and  $\text{RN}_e((2^p x) \cdot y)$  can be computed without overflow. This gives two values, say  $a'$  and  $b'$ , such that  $a' = \text{RN}_0(2^p x \cdot y)$  and  $b' = 2^p x \cdot y - a'$ . We immediately deduce that  $2^{-p}a'$  is the expected  $\text{RN}_0(x \cdot y)$ . Obtaining  $\text{RN}_0(x \cdot y - 2^{-p}a') = \text{RN}_0(2^{-p}b')$  is slightly more tricky. We first compute  $\beta = \text{RN}_e(2^{-p}b')$ . The number  $\beta$  is equal to the expected  $\text{RN}_0(2^{-p}b')$  unless

$$\beta - (2^{-p}b') = \text{sign}(\beta) \cdot 2^{e_{\min}-p} \quad (3)$$

in which case, one should replace  $\beta$  by  $\beta - \text{sign}(\beta) \cdot 2^{e_{\min}-p+1}$ . Equation (3) is implied by

$$2^p \beta - b' = \text{sign}(\beta) \cdot 2^{e_{\min}},$$

a condition which is easy to test since the subtraction is exact:  $2^p \beta - b'$  is a multiple of  $2^{e_{\min}-p+1}$ , of magnitude less than or equal to  $2^{e_{\min}}$ , hence it is a floating-point number.

All this gives Algorithm 9 and Theorem 4, below.

---

**ALGORITHM 9:** AM-Full( $x, y$ ): computes augmentedMultiplication( $x, y$ ) in all cases.

---

```

1:  $a_e \leftarrow \text{RN}_e(x \cdot y)$ 
2: if  $|a_e| = +\infty$  then
3:    $x' \leftarrow 0.5 \cdot x$ 
4:    $(a'_e, b'_e) \leftarrow \text{Fast2Mult}(x', y)$ 
5:   if  $(a'_e = 2^{e_{\max}}$  and  $b'_e = -2^{e_{\max}-p+1})$  or
      $(a'_e = -2^{e_{\max}}$  and  $b'_e = +2^{e_{\max}-p+1})$  then
6:      $a_0 \leftarrow \text{RN}_e(a'_e \cdot (2 - 2^{-p+1}))$ 
7:      $b_0 \leftarrow -2b'_e$ 
8:   else
9:      $a_0 \leftarrow a_e$  (infinity with right sign)
10:     $b_0 \leftarrow a_e$ 
11:   end if
12: else if  $|a_e| \leq 2^{e_{\min}+p}$  then
13:   if  $a_e = 0$  then
14:      $a_0 \leftarrow a_e$ 
15:      $b_0 \leftarrow a_e$ 
16:   else if  $|a_e| \leq 2^{e_{\min}+1} - 2^{e_{\min}-p+1}$  then
17:      $b_0 \leftarrow 0$ 
18:      $(t_1, t_2) \leftarrow \text{Fast2Mult}((x \cdot 2^{2p}), y)$ 
19:      $t_3 \leftarrow \text{RN}_e(t_1 - a_e \cdot 2^{2p})$ 
20:      $z \leftarrow \text{RN}_e(t_2 + t_3)$ 
21:     if  $(z = -\text{sign}(a_e) \cdot 2^{e_{\min}+p})$  and
        $(\text{RN}_e(z - t_3) = t_2)$  then
22:        $a_0 \leftarrow a_e - \text{sign}(a_e) \cdot 2^{e_{\min}-p+1}$ 
23:     else
24:        $a_0 \leftarrow a_e$ 
25:     end if
26:   else
27:      $(a', b') \leftarrow \text{AM-Simple}(2^p x, y)$ 
28:      $a_0 \leftarrow \text{RN}_e(2^{-p} \cdot a')$ 
29:      $\beta \leftarrow \text{RN}_e(2^{-p} \cdot b')$ 
30:     if  $\text{RN}_e(2^p \beta - b') = \text{sign}(\beta) \cdot 2^{e_{\min}}$  then
31:        $b_0 \leftarrow \beta - \text{sign}(\beta) \cdot 2^{e_{\min}-p+1}$ 
32:     else
33:        $b_0 \leftarrow \beta$ 
34:     end if
35:   end if
36: else
37:    $b_e \leftarrow \text{RN}_e(x \cdot y - a_e)$ 
38:    $(a_0, b_0) \leftarrow \text{Recomp}(a_e, b_e)$ 
39: end if
40: return  $(a_0, b_0)$ 

```

---

**Theorem 4.** The output  $(a_0, b_0)$  of Algorithm 9 is equal to augmentedMultiplication( $x, y$ ).

## V. FORMAL PROOF

Arithmetic algorithms can be used in critical applications. Their proof can be somehow complex, with many particular cases to be considered. This makes them a good candidate for formal proof. Using Coq and the Flocq library [3], we have formally proven Theorem 1 and Theorem 4, with the following important limitation (due to the nature of the formalization of the FP numbers in Flocq): overflows, infinite numbers, and the signs of zeroes are not considered.

## VI. IMPLEMENTATION AND COMPARISON

We have implemented the algorithms presented in this paper in binary64 (a.k.a. *double precision*) arithmetic, as well as emulation algorithms based on integer arithmetic. We used an x86\_64 processor under GNU/Linux (Debian 4.9.144-3), and the programs were compiled using GCC (Debian 6.3.0-18+deb9u1) 6.3.0 20170516, with the option `-O3 -march=native`.

The statistical distribution of the number of cycles (using  $10^6$  samples, assuming uniform distribution of the significands and the exponents, and no overflows but including subnormal results) is given in Figures 3 (for our augmentedAddition algorithm, Algorithm 7), 4 (for an integer-based emulation of augmentedAddition), 5 (for our augmentedMultiplication algorithm, Algorithm 9), and 6 (for an integer-based emulation of augmentedMultiplication). The average timings are given in the first half of Table II. The second half of Table II gives average timings for halfway cases.

Concerning augmentedAddition, Algorithm 7 is slightly better than the integer-based emulation in the general case, and significantly better in the bad cases. Concerning augmentedMultiplication, Algorithm 9 is significantly better, except on very rare cases (at the extreme right of Figure 5).

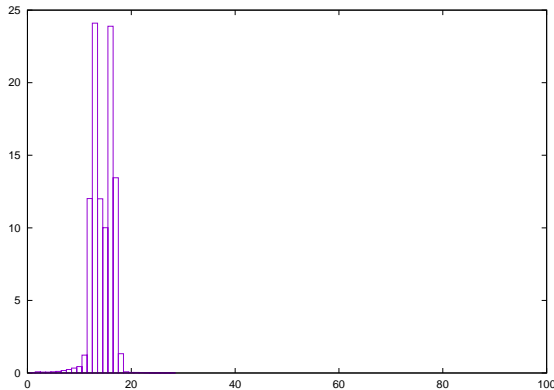


Fig. 3. Statistical distribution of the number of cycles for our augmentedAddition algorithm (Algorithm 7).

## CONCLUSION

We have presented and implemented algorithms that allow one to emulate the newly suggested “augmented” floating-point operations using the classical, rounded-ties-to-even operations. The algorithms are very simple in the general case. Special cases are slightly more involved but will remain infrequent

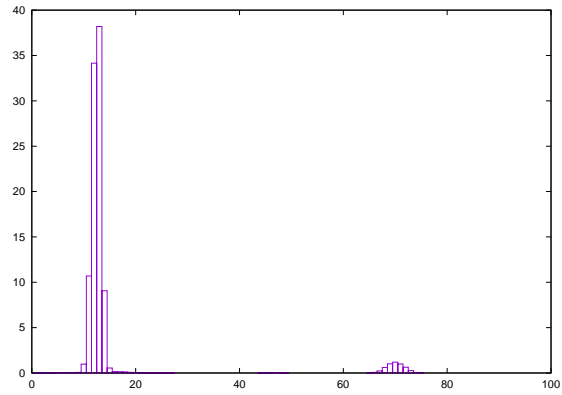


Fig. 4. Statistical distribution of the number of cycles for an integer-based emulation of augmentedAddition.

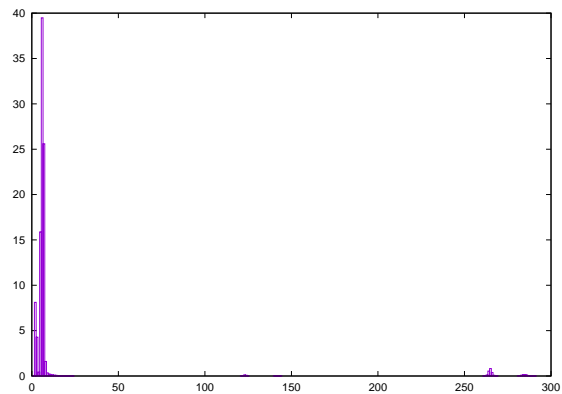


Fig. 5. Statistical distribution of the number of cycles for our augmentedMultiplication algorithm (Algorithm 9).

in most applications. These algorithms compare favorably with an integer-based emulation of the augmented operations. Furthermore, the availability of formal proofs (despite the limitations presented in Section V) increases confidence in them.

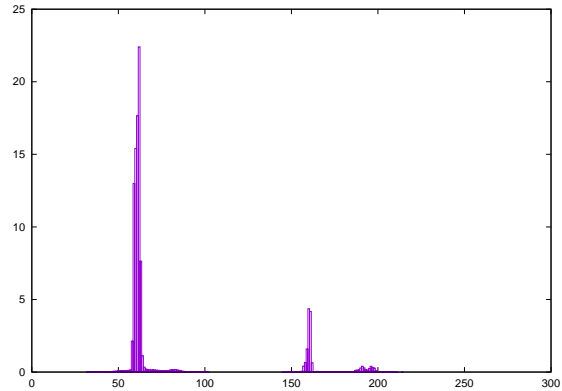


Fig. 6. Statistical distribution of the number of cycles for an integer-based emulation of augmentedMultiplication.



TABLE II  
AVERAGE TIMINGS IN CYCLES

Algorithm	# of cycles
Algorithm 7 (addition, all cases)	14.62
Integer-based emulation (addition, all cases)	15.67
Algorithm 9 (multiplication, all cases)	13.97
Integer-based emulation (multiplication, all cases)	78.23
Algorithm 7 (addition, halfway cases)	14.44
Integer-based emulation (addition, halfway cases)	70.46
Algorithm 9 (multiplication, halfway cases)	7.41
Integer-based emulation (multiplication, halfway cases)	60.72

#### ACKNOWLEDGEMENT

We thank Claude-Pierre Jeannerod for his very useful suggestions.

#### REFERENCES

- [1] IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, July 2019.
- [2] Sylvie Boldo, Stef Graillat, and Jean-Michel Muller. On the robustness of the 2Sum and Fast2Sum algorithms. *ACM Transactions on Mathematical Software*, 44(1):4:1–4:14, July 2017.
- [3] Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs*. ISTE Press – Elsevier, 2017.
- [4] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.
- [5] James Demmel, Peter Ahrens, and Hong Diep Nguyen. Efficient reproducible floating point summation and BLAS. Technical Report UCB/EECS-2016-121, EECS Department, University of California, Berkeley, Jun 2016.
- [6] John Harrison. A machine-checked theory of floating point arithmetic. In *12th International Conference in Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130, Nice, France, September 1999. Springer-Verlag, Berlin.
- [7] J. R. Hauser. Handling floating-point exceptions in numeric programs. *ACM Transactions on Programming Languages and Systems*, 18(2):139–174, 1996.
- [8] C.-P. Jeannerod, J. Muller, and P. Zimmermann. On various ways to split a floating-point number. In *25th IEEE Symposium on Computer Arithmetic, Amherst, MA, USA*, pages 53–60, June 2018.
- [9] W. Kahan. Lecture notes on the status of IEEE-754. Available at <http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF>, 1997.
- [10] D. E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, Reading, MA, 3rd edition, 1998.
- [11] O. Møller. Quasi double-precision in floating-point addition. *BIT*, 5:37–50, 1965.
- [12] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2018.
- [13] Y. Nievergelt. Scalar fused multiply-add instructions produce floating-point matrix arithmetic provably accurate to the penultimate digit. *ACM Transactions on Mathematical Software*, 29(1):27–48, 2003.
- [14] E. Jason Riedy and James Demmel. Augmented arithmetic operations proposed for IEEE-754 2018. In *25th IEEE Symposium on Computer Arithmetic, Amherst, MA, USA*, pages 45–52, June 2018.
- [15] Siegfried M. Rump. Ultimately fast accurate summation. *SIAM Journal on Scientific Computing*, 31(5):3466–3502, January 2009.
- [16] P. H. Sterbenz. *Floating-Point Computation*. Prentice-Hall, Englewood Cliffs, NJ, 1974.