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

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‡

* Université Paris-Saclay, Univ. Paris-Sud, CNRS, Inria, Laboratoire de recherche en informatique, 91405 Orsay, France † University of Alaska Anchorage, College of Engineering, Computer Science Department, Anchorage, AK, 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* with a partial formal proof of correctness.**

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

I. Introduction and notation

The new IEEE 754-2019 Standard for Floating-Point (FP) Arithmetic [8] supersedes the 2008 version. It recommends that new "augmented" operations should be provided for the binary formats (see [15] for history and motivation). 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 may require a direct hardware implementation of these operations. However, having these operations available on common processors will certainly take time, and they may not be available on all platforms. The purpose of this paper is to show that, in the meantime, one can emulate these operations with conventional FP operations (with the usual round-to-nearest *ties-to-even* rounding direction), with reasonable efficiency. In this paper, we present the first proposed emulation algorithms, with proof of their correctness and experimental results. This allows, for instance, the design of programs that use these operations, and that will be ready for use with full efficiency as soon as the augmented operations are available in hardware. Also, when these operations are available in hardware on some systems, this will improve the portability of programs using these operations by allowing them to still work (with degraded performance, however) on other systems.

In the following, we assume radix-2, precision-p floating-point arithmetic [13]. The minimum floating-point exponent is $e_{\min} < 0$ and the maximum exponent is $e_{\max}$. A floating-point number is a number of the form

$$x = M_x \times 2^{e_x - p+1}, \tag{1}$$

where

$$M_x \text{ is an integer satisfying } |M_x| \le 2^p - 1, \tag{2}$$

and

$$e_{\min} \le e_x \le e_{\max}. \tag{3}$$

If $|M_x|$ is maximum under the constraints (1), (2), and (3), then e_x is the *floating-point exponent* of x. The number $2^{e_{\min}}$ is the smallest positive normal number (a FP number of absolute value less than $2^{e_{\min}}$ is called *subnormal*), and $2^{e_{\min}-p+1}$ is the smallest positive FP number. The largest positive FP number is

$$\Omega = (2 - 2^{-p+1}) \cdot 2^{e_{\max}}.$$

We will assume

$$3p \le e_{\max} + 1, \tag{4}$$

which is satisfied by all binary formats of the IEEE 754 Standard, *with the exception of binary16* (which is an *interchange format* but not a *basic format* [8]). The usual round-to-nearest, ties-to-even function (which is the default in the IEEE-754 Standard) will be noted RN_e. We recall its definition [8]:

> $\mathrm{RN}_e(t)$ (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, $\mathrm{RN}_e(t)$ is the one whose least significant bit is zero. If $|t| \ge \Omega + 2^{e_{\max}-p}$ then $\mathrm{RN}_e(t) = \infty$, 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 FP 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 RN_0 defined as follows [8]:

> $\mathrm{RN}_0(t)$ (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, $\mathrm{RN}_0(t)$ is the one with smaller magnitude. If $|t| > \Omega + 2^{e_{\max}-p}$ then $\mathrm{RN}_0(t) = \infty$, with the same sign as t.

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

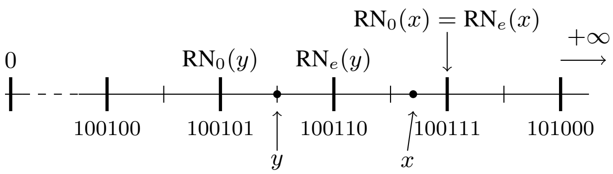


Fig. 1: Round-to-nearest ties-to-zero (assuming we are in the positive range). Number x is rounded to the (unique) FP number nearest to x. Number y is a halfway case: it is exactly halfway between two consecutive FP numbers: it is rounded to the one that has the smallest magnitude.

The augmented operations are required to behave as follows [8], [15]:

- **augmentedAddition**(x, y) delivers (a_0, b_0) such that $a_0 = \mathrm{RN}_0(x + y)$ and, when $a_0 \notin \{\pm\infty, \mathrm{NaN}\}$, $b_0 = (x+y) - a_0$. When $b_0 = 0$, it is required to have the same sign as a_0. One easily shows that b_0 is a FP number. For special rules when $a_0 \in \{\pm\infty, \mathrm{NaN}\}$, see [15];
- **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 $a_0 = \mathrm{RN}_0(x \cdot y)$ and, where $a_0 \notin \{\pm\infty, \mathrm{NaN}\}$, $b_0 = \mathrm{RN}_0((x \cdot y) - a_0)$. When $(x \cdot y) - a_0 = 0$, the floating-point number b_0 (equal to zero) 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 [8], [15].

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 ties-to-even FP 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 $x \cdot y$ for 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, x and y satisfy the requirements of Lemma 2 below), we first use the classical Fast2Sum, 2Sum, or Fast2Mult algorithms to generate two FP numbers a_e and b_e such that $a_e = \mathrm{RN}_e(t)$ 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 ulp ("unit in the last place"). If x is a floating-point number different from $-\Omega$, first define pred(x) as the floating-point predecessor of x, i.e., the largest floating-point number $< x$. We define $\mathrm{ulp}_H(x)$ as follows.

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

$$|x| - \mathrm{pred}\,(|x|)\,.$$

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

The reason for choosing function ulp_H instead of function ulp is twofold:

- if $t > 0$ is a real number, each time $\mathrm{RN}_0(t)$ differs from $\mathrm{RN}_e(t)$, $\mathrm{RN}_0(t)$ will be the floating-point predecessor of $\mathrm{RN}_e(t)$, because $\mathrm{RN}_0(t) \neq \mathrm{RN}_e(t)$ implies that t is what we call a "halfway case" in Section II: it is exactly halfway between two consecutive floating-point numbers, and in that case, $\mathrm{RN}_0(t)$ is the one of these two FP numbers which is closest to zero and $\mathrm{RN}_e(t)$ is the other one. Hence, in these cases, to obtain $\mathrm{RN}_0(t)$ we will have to subtract from $\mathrm{RN}_e(t)$ a number which is exactly $\mathrm{ulp}_H(\mathrm{RN}_e(t))$ (for negative t, for symmetry reasons, we will have to add $\mathrm{ulp}_H(\mathrm{RN}_e(t))$ to $\mathrm{RN}_e(t)$); and
- there is a very simple algorithm for computing $\mathrm{ulp}_H(t)$ 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].

$a_e \leftarrow \mathrm{RN}_e(x + y)$
$y' \leftarrow \mathrm{RN}_e(a_e - x)$
$b_e \leftarrow \mathrm{RN}_e(y - y')$

Lemma 1. *If $x = 0$ or $y = 0$, or if the floating-point exponents e_x and e_y of x and y satisfy $e_x \geq e_y$, then*

1) *the two variables a_e and b_e returned by Algorithm 1 (Fast2Sum) satisfy $a_e + b_e = x + y$;*
2) *the operations performed at lines 2 and 3 of Algorithm 1 are exact operations: $y' = a_e - x$ and $b_e = y - y'$.*

See for instance [13] for a proof. Hence, the variable b_e returned by Algorithm 1 is the error of the floating-point addition $a_e \leftarrow \mathrm{RN}_e(x + y)$. The second property given in Lemma 1 will be useful in Section IV-C. 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 [1] that if the addition $\mathrm{RN}_e(x + y)$ does not overflow then the other two operations cannot overflow.

ALGORITHM 2: 2Sum(x, y). The 2Sum algorithm [12], [11].

$a_e \leftarrow \mathrm{RN}_e(x+y)$
$x' \leftarrow \mathrm{RN}_e(a_e - y)$
$y' \leftarrow \mathrm{RN}_e(a_e - x')$
$\delta_x \leftarrow \mathrm{RN}_e(x - x')$
$\delta_y \leftarrow \mathrm{RN}_e(y - y')$
$b_e \leftarrow \mathrm{RN}_e(\delta_x + \delta_y)$

Algorithm 2 (2Sum) gives the same results a_e and b_e 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 $\mathrm{RN}_e(x+y)$ does not overflow then the other five operations cannot overflow [1].

A similar algorithm, Algorithm 3 (Fast2Mult), makes it possible to express the exact product of two floating-point numbers x and y as the sum of the rounded product $\mathrm{RN}_e(xy)$ and an error term. It requires the availability of an FMA (*fused multiply-add*) instruction. To be exactly representable as the sum of two floating-point numbers, the exact product must not be too tiny. Several sufficient conditions appear in the literature (such as the exponents e_x and e_y satisfying $e_x + e_y \geq e_{\min} + p - 1$, see [14] for a proof). We will use a slightly different condition, given by Lemma 2 below.

ALGORITHM 3: Fast2Mult(x, y)**.** The Fast2Mult algorithm (see for instance [10], [14], [13]). It requires the availability of a fused multiply-add (FMA) instruction for computing $\mathrm{RN}_e(x \cdot y - a_e)$.

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

Lemma 2. *If* $2^{e_{\min}+p} \leq |x \cdot y| < \Omega + 2^{e_{\max}-p}$ *(which in particular holds if* $2^{e_{\min}+p} + 2^{e_{\min}+1} \leq |\mathrm{RN}_e(x \cdot y)| \leq \Omega$*) then the numbers* a_e *and* b_e *returned by Algorithm 3 satisfy*

$$a_e + b_e = x \cdot y.$$

Proof. Let k be the integer such that $2^k \leq |xy| < 2^{k+1}$. Since xy is a $2p$-bit number, it is a multiple of 2^{k-2p+1}, so that $xy - \mathrm{RN}_e(xy)$ is a multiple of 2^{k-2p+1} too. $|xy - \mathrm{RN}_e(xy)|$ is less than or equal to $\frac{1}{2}\mathrm{ulp}(xy) = 2^{k-p}$. Also, Condition $2^{e_{\min}+p} \leq |x \cdot y|$ implies $k \geq e_{\min} + p$. All this implies that $xy - \mathrm{RN}_e(xy)$ can be written $M \times 2^{k-2p+1}$, where M is an integer of absolute value less than or equal to 2^{p-1} and $k - 2p + 1 \geq e_{\min} - p + 1$, which implies that $xy - \mathrm{RN}_e(xy)$ is a floating-point number, which implies that $b_e = xy - \mathrm{RN}_e(xy) = xy - a_e$. ☐

We will also use the following, classical results, due to Hauser [7] and Sterbenz [17] (the proofs are straightforward, see for instance [13]).

Lemma 3 (Hauser Lemma)**.** *If x and y are floating-point numbers, and if the number* $\mathrm{RN}_e(x+y)$ *is subnormal, then $x+y$ is a floating-point number, which implies* $\mathrm{RN}_e(x+y) = x+y$.

Lemma 4 (Sterbenz Lemma)**.** *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* $\mathrm{RN}_e(x-y) = x - y$.

Finally, we will sometimes use the following lemmas, whose proofs are straightforward.

Lemma 5. *If a is a nonzero floating-point number. If t is a real number such that $|t| \leq |a|$ and t is a multiple of* $\mathrm{ulp}(a)$, *then t is a floating-point number.*

Lemma 6. *If t_1 and t_2 are real numbers such that*

1) $2^{e_{\min}} \leq |t_1|,\ |t_2| < \Omega + 2^{e_{\max}-p}$*;*
2) there exists an integer k such that $t_1 = 2^k \cdot t_2$;

then $\mathrm{RN}_e(t_1) = 2^k \cdot \mathrm{RN}_e(t_2)$ *and* $\mathrm{RN}_0(t_1) = 2^k \cdot \mathrm{RN}_0(t_2)$.

As explained in Section II (where it corresponds to "Halfway Case 1"), when $\mathrm{RN}_0(t)$ and $\mathrm{RN}_e(t)$ differ, $\mathrm{RN}_0(t)$ is obtained by subtracting $\mathrm{sign}(t)\cdot\mathrm{ulp}_H(\mathrm{RN}_e(t))$ from $\mathrm{RN}_e(t)$. Therefore, we need to be able to compute $\mathrm{sign}(a)\cdot\mathrm{ulp}_H(a)$. If $|a| > 2^{e_{\min}}$, this can be done using Algorithm 4 below, which is a variant of an algorithm introduced by Rump [16].

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

$z \leftarrow \mathrm{RN}_e(\psi a)$
$\delta \leftarrow \mathrm{RN}_e(a - z)$
return (z, δ)

Lemma 7. *The numbers z and δ returned by Algorithm 4 satisfy:*

- *if* $|a| > 2^{e_{\min}}$ *then* $z = \mathit{sign}(a) \cdot \mathrm{pred}(|a|)$ *and* $\delta = \mathit{sign}(a) \cdot \mathrm{ulp}_H(a)$*;*
- *If* $|a| \leq 2^{e_{\min}}$ *then $z = a$ and $\delta = 0$.*

Proof.

- The fact that when $|a| > 2^{e_{\min}}$ the number z returned by Algorithm 4 equals $\mathrm{sign}(a) \cdot \mathrm{pred}(|a|)$ is a direct consequence of [16, Lemma 3.6] (see also [9]). The value of δ immediately follows from that.
- If $|a| < 2^{e_{\min}}$ (i.e., a is subnormal or zero), then $|2^{-p}a| < 2^{e_{\min}-p} = \frac{1}{2}\mathrm{ulp}(a)$, from which we obtain $|\psi a - a| < \frac{1}{2}\mathrm{ulp}(a)$, thus $z = \mathrm{RN}_e(\psi a) = a$ and $\delta = 0$.
- Finally, if $|a| = 2^{e_{\min}}$, the ties-to-even rule implies $z = \mathrm{RN}_e(\psi a) = a$ and $\delta = 0$.

☐

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 FP numbers a_e and b_e, that satisfy $a_e = \mathrm{RN}_e(t)$, 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

(Algorithms 1, 2, and 3). We want to obtain from a_e and b_e two FP numbers a_0 and b_0 such that $a_0 = \mathrm{RN}_0(t)$ and $a_0 + b_0 = t$. Before giving the algorithm, let us present the basic principle in the case $2^{e_{\min}} < t < \Omega$ (t is thus assumed positive to simplify the presentation). If t is not halfway between two consecutive FP numbers, we know that $a_0 = a_e$ and $b_0 = b_e$. If t is halfway between two FP numbers (one of them being a_e), then two cases may occur:

- **Halfway case 1:** $t = a_e - \frac{1}{2}\mathrm{ulp}_H(a_e)$ (i.e., $b_e = -\frac{1}{2}\mathrm{ulp}_H(a_e)$);
- **Halfway case 2:** $t = a_e + \frac{1}{2}\mathrm{ulp}_H(a_e)$ (i.e., $b_e = +\frac{1}{2}\mathrm{ulp}_H(a_e)$).

In the second case, a_e is already equal to t rounded to zero, so we must choose $a_0 = a_e$ and $b_0 = b_e$. In the first case, a_0 is the floating-point predecessor of a_e, and $b_0 = \frac{1}{2}\mathrm{ulp}_H(a_e) = -b_e$.

Hence, to find a_0 and b_0 we must first detect if we are in Halfway case 1: it is the only case where (a_0, b_0) differs from (a_e, b_e). That detection is done using Algorithm 4 (MyulpH).

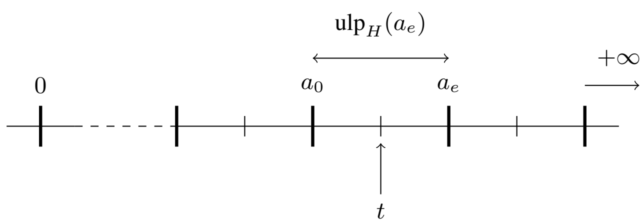


Fig. 2: Halfway case 1: $t = a_e - (1/2)\mathrm{ulp}_H(a_e)$, where $t = a_e + b_e$. We have $a_0 = a_e - \mathrm{ulp}_H(a_0)$ and $b_0 = -b_e$.

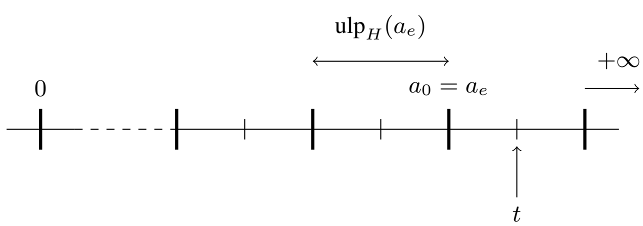


Fig. 3: Halfway case 2: $t = a_e + (1/2)\mathrm{ulp}_H(a_e)$, where $t = a_e + b_e$. We have $a_0 = a_e$ and $b_0 = b_e$.

This is illustrated by Figures 2 and 3, and this leads to Algorithm 5 below. In Algorithm 5, when the number $-2 \cdot b_e$ is equal to δ (i.e., when Halfway case 1 occurs), we must return $a_0 = a_e - \delta = \mathrm{sign}(a_e) \cdot \mathrm{pred}(|a_e|)$. This explains why in that case the value of a_0 returned by the algorithm is z. We obtain Lemma 8 below.

Lemma 8. *If $2^{e_{\min}} < |a_e| \leq \Omega$ then the two floating-point numbers a_0 and b_0 returned by Algorithm 5 satisfy*

$$\begin{aligned} a_0 &= \mathrm{RN}_0(a_e + b_e), \\ a_0 + b_0 &= a_e + b_e. \end{aligned}$$

Condition $2^{e_{\min}} < |a_e|$ in Lemma 8 is necessary: if $|a_e| \leq 2^{e_{\min}}$, an immediate consequence of Lemma 7 is that Algorithm 5 returns $a_0 = a_e$ and $b_0 = b_e$. This is not a problem for implementing augmentedAddition thanks to Lemma 3, as we are going to see in Section III. For augmentedMultiplication this will require a special handling (see Sections IV-C and IV-D).

ALGORITHM 5: Recomp(a_e, b_e). From two FP numbers a_e and b_e such that $a_e = \mathrm{RN}_e(a_e + b_e)$ and $|a_e| > 2^{e_{\min}}$, computes a_0 and b_0 such that $a_0 + b_0 = a_e + b_e$ and $a_0 = \mathrm{RN}_0(a_e + b_e)$.

$(z, \delta) \leftarrow \mathrm{MyulpH}(a_e)$
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)

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 $\mathrm{RN}_0(x+y)$ and $(x+y) - \mathrm{RN}_0(x+y)$. We recall that when $(x+y) - \mathrm{RN}_0(x+y)$ equals zero, the IEEE 754-2019 Standard requires that it should be returned with the sign of $\mathrm{RN}_0(x+y)$. Let us first give a simple algorithm (Algorithm 6, below) that returns a correct result (possibly with a wrong sign for b_0 when it is zero) 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.

if $|y| > |x|$ **then**
  swap(x, y)
end if
$(a_e, b_e) \leftarrow \mathrm{Fast2Sum}(x, y)$
$(a_0, b_0) \leftarrow \mathrm{Recomp}(a_e, b_e)$
return (a_0, b_0)

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

1) *if $|x+y| < \Omega + 2^{e_{\max}-p} = (2 - 2^{-p}) \cdot 2^{e_{\max}}$ then (a_0, b_0) is equal to augmentedAddition(x, y), with the possible exception that if $b_0 = 0$ it may have a sign that differs from the one specified in the IEEE 754-2019 Standard;*
2) *if $|x+y| = \Omega + 2^{e_{\max}-p}$ 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);*
3) *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 $x + y$).*

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). Note also that Theorem 1 implies that each time a_0 is a finite floating-point number, Algorithm 6 returns a correct result (with a possible wrong sign for b_0 when it is zero).

The first item in Theorem 1 is an immediate consequence of the properties of the Fast2Sum and Recomp algorithms. Let us momentarily ignore the signs of zero variables. We have $a_e = \mathrm{RN}_e(x + y)$ 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 $|a_e| \le 2^{e_{\min}}$ then from Lemma 3, 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 $a_0 = b_0 = 0$.

Now, let us reason about the signs of zero variables. Note that $a_0 = 0$ is possible only when $x + y = 0$. A quick look at Fast2Sum and MyulpH shows that when $x + y = 0$, $a_0 = 0$ with the same sign as a_e, which corresponds to what is requested by IEEE 754-2019. Hence, when $a_0 = 0$, it has the right sign.

When $b_0 = 0$, this may come from two possible cases: either $x + y$ is a nonzero floating-point number (in which case, a_0 is that number), or $x + y = 0$. In both cases b_0 should be zero with the same sign as a_0. Tables I and II give the values of b_0 returned by Algorithm 6 in these two cases. One can see that when $b_0 = 0$, its sign is not always correct.

TABLE I: Value of b_0 computed by Algorithm 6 and value of b_0 specified by the IEEE-754 Standard when $x + y$ is a nonzero floating-point number (i.e., $b_e = \pm 0$).

Case			computed b_0	correct b_0
$y \neq 0$	and	$\|a_e\| > 2^{e_{\min}}$	$+0$	$+0 \times \mathrm{sign}(a_e)$
		$\|a_e\| \le 2^{e_{\min}}$	-0	
$y = +0$	and	$\|a_e\| > 2^{e_{\min}}$	$+0$	$+0 \times \mathrm{sign}(a_e)$
		$\|a_e\| \le 2^{e_{\min}}$	-0	
$y = -0$	and	$\|x\| = \|a_e\| > 2^{e_{\min}}$	-0	$+0 \times \mathrm{sign}(a_e)$
		$\|x\| = \|a_e\| \le 2^{e_{\min}}$	$+0$	

TABLE II: Value of b_0 computed by Algorithm 6 and value of b_0 specified by the IEEE-754 Standard when $x + y = 0$).

Case			computed b_0	correct b_0
$x = -y$	and	$x \neq 0$	-0	$+0$
$x = +0$	and	$y = +0$	-0	$+0$
$x = +0$	and	$y = -0$	$+0$	$+0$
$x = -0$	and	$y = +0$	-0	$+0$
$x = -0$	and	$y = -0$	$+0$	-0

However, if the signs of the zero variables matter in the target application, there is a simple solution. Since the sign of a_0 is always correct, and since when $b_0 = 0$ it must be returned with the sign of a_0, it suffices to add to add the following lines to Algorithm 6 after Line 5:

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

Alternatively, one can also use the **copySign** instruction specified by the IEEE 754 Standard [8] if it is faster than a floating-point multiplication on the system being used: copySign(x, y) has the absolute value of x and the sign of y.

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

Concerning the third item in Theorem 1, Table III gives the values returned by Algorithm 6 when $|x + y| > \Omega + 2^{e_{\max}-p}$, and compares them with the correct values.

TABLE III: Values obtained using Algorithm 6 (possibly with a replacement of Fast2Sum by 2Sum) when $|x + y| > 2^{e_{\max}}(2 - 2^{-p})$.

	Algorithm 6	Variant of Algorithm 6 with (a_e, b_e) obtained through Fast2Sum	Result required by the standard
a_0	$+\infty \cdot \mathrm{sign}(x + y)$	$+\infty \cdot \mathrm{sign}(x + y)$	$+\infty \cdot \mathrm{sign}(x + y)$
b_0	NaN	$-\infty \cdot \mathrm{sign}(x + y)$	$+\infty \cdot \mathrm{sign}(x + y)$

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 zeroes matter) is a good candidate. If we wish to always follow the specifications, we suggest using Algorithm 7 below.

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) ← Fast2Sum(x, y)
5:  (a_0, b_0) ← Recomp(a_e, b_e)
6:  if b_0 = 0 then
7:      b_0 ← (+0) × a_0
8:  else if |a_e| = +∞ then
9:      (a'_e, b'_e) ← 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 ← RN_e(a'_e · (2 − 2^{−p+1}))
12:         b_0 ← −2b'_e
13:     else
14:         a_0 ← a_e    (infinity with right sign)
15:         b_0 ← a_e
16:     end if
17: end if
18: return (a_0, b_0)
```

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

Proof.

1) if $|x + y| < \Omega + 2^{e_{\max}-p}$ then Item 1 of Theorem 1 tells us that the values a_0 and b_0 computed at Line 5 of Algorithm 7 are equal to augmentedAddition(x, y), with the possible exception that if $b_0 = 0$ it may have a sign that differs from the one specified in the IEEE 754-2019

Standard. This possible error in the sign of b_0 is corrected at Lines 6-7.

2) if $|x+y| = \Omega + 2^{e_{\max}-p}$ then $|a_e| = +\infty$. In that case, since $\frac{1}{2}\cdot|x+y| = \frac{\Omega}{2} + 2^{e_{\max}-p-1} = 2^{e_{\max}} - 2^{e_{\max}-p-1}$, at Line 9 we obtain

$$a' = \mathrm{sign}(x+y)\cdot 2^{e_{\max}}$$

and

$$b' = -\mathrm{sign}(x+y)\cdot 2^{e_{\max}-p-1}.$$

In that case, Lines 10-12 of the algorithm return the correct values

$$a_0 = \mathrm{sign}(x+y)\cdot\Omega$$

and

$$b_0 = \mathrm{sign}(x+y)\cdot 2^{e_{\max}-p-1}.$$

3) if $|x+y| > \Omega + 2^{e_{\max}-p}$ then $|a_e| = +\infty$, and the sum $(x+y)/2$ computed using Fast2Sum at Line 9 (without overflow since $|x+y|/2$ is less than or equal to the maximum of $|x|$ and $|y|$) will be or absolute value (strictly) larger than $2^{e_{\max}} - 2^{e_{\max}-p-1}$, hence Lines 14-15 of the algorithm will be executed, and we will obtain $a_0 = b_0 = \mathrm{sign}(x+y)\cdot\infty$, as expected.

□

IV. Use of Algorithm Recomp for implementing augmentedMultiplication

A. General case

From two input floating-point numbers x and y, we wish to compute $\mathrm{RN}_0(x\cdot y)$ and $x\cdot y - \mathrm{RN}_0(x\cdot y)$ (or, merely, $\mathrm{RN}_0\big[x\cdot y - \mathrm{RN}_0(x\cdot y)\big]$ when $x\cdot y - \mathrm{RN}_0(x\cdot y)$ is not a floating-point number). As we did for augmentedAddition, let us first present a simple algorithm (Algorithm 8 below). 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

$$2^{e_{\min}+p} + 2^{e_{\min}} < |x\cdot y| < \Omega + 2^{e_{\max}-p}.$$

```
1: (a_e, b_e) ← Fast2Mult(x, y)
2: (a_0, b_0) ← Recomp(a_e, b_e)
3: return (a_0, b_0)
```

Theorem 3. *The values a_0 and b_0 returned by Algorithm 8 satisfy:*

1) *If $2^{e_{\min}+p} + 2^{e_{\min}} < |x\cdot y| < \Omega + 2^{e_{\max}-p}$ (which implies $2^{e_{\min}+p} + 2^{e_{\min}+1} \leq |\mathrm{RN}_e(x\cdot y)| \leq \Omega$) then (a_0, b_0) is equal to augmentedMultiplication(x,y), with the possible exception that if $b_0 = 0$ it may have a sign that differs from the one specified in the IEEE 754-2019 Standard;*
2) *if $|x\cdot y| = \Omega + 2^{e_{\max}-p} = (2-2^{-p})\cdot 2^{e_{\max}}$, then $a_0 = \pm\infty$ (with the sign of $x\cdot y$) and $b_0 = \pm\infty$ (with the opposite sign) whereas the correct values would have been $a_0 = \pm\Omega$ and $b_0 = \pm 2^{e_{\max}-p}$ (both with the sign of $x\cdot y$);*
3) *if $|x\cdot y| > \Omega + 2^{e_{\max}-p}$, then $a_0 = \pm\infty$ (with the sign of $x\cdot y$) and $b_0 = \pm\infty$ (with the opposite sign) whereas the correct values would have been $a_0 = b_0 = \pm\infty$ (with the sign of $x\cdot y$).*

Proof. The first item in Theorem 3 is a consequence of Lemma 2 and Lemma 8. If

$$2^{e_{\min}+p} + 2^{e_{\min}} < |x\cdot y| < \Omega + 2^{e_{\max}-p}$$

then $2^{e_{\min}+p} + 2^{e_{\min}+1} \leq |\mathrm{RN}_e(x\cdot y)| \leq \Omega$, therefore

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

therefore Recomp(a_e, b_e) returns the expected result.

The second item in Theorem 3 follows immediately by applying Algorithm 8 to the corresponding input value. Concerning the third item in Theorem 3, Table IV gives the values returned by Algorithm 8 when $|x\cdot y| > \Omega + 2^{e_{\max}-p}$.

□

TABLE IV: Values obtained using Algorithm 8 when $|x\cdot y| > 2^{e_{\max}}(2-2^{-p})$.

	Algorithm 8	Result required by the standard
a_0	$+\infty\cdot\mathrm{sign}(x\cdot y)$	$+\infty\cdot\mathrm{sign}(x\cdot y)$
b_0	$-\infty\cdot\mathrm{sign}(x\cdot y)$	$+\infty\cdot\mathrm{sign}(x\cdot y)$

As with the addition algorithm, if the signs of the zero variables matter in the target application and if Condition 1 of Theorem 3 is satisfied, it suffices 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

(and, again, function **copySign** can be used if it is faster than a floating-point multiplication). Another solution is to notice that in Case 1 of Theorem 3, $xy - a_0$ is a floating-point number, therefore one can just compute a_0 with Algorithm 8 and obtain b_0 with one FMA instruction, as $\mathrm{RN}_e(xy - a_0)$. As we are going to see that solution is useful even when Condition 1 of Theorem 3 is not satisfied.

Let us now build another augmented multiplication algorithm, Algorithm 9 below, that returns a correct result even if the condition of Case 1 of Theorem 3 (i.e., $2^{e_{\min}+p} + 2^{e_{\min}} < |x\cdot y| < \Omega + 2^{e_{\max}-p}$) is not satisfied. We need to be able to address the cases

$$\mathrm{RN}_e(x\cdot y) = \pm\infty$$

(that correspond to items 2 and 3 in Theorem 3) and

$$|\mathrm{RN}_e(x\cdot y)| \leq 2^{e_{\min}+p}.$$

This will be done by *scaling* the calculation, i.e., by finding a suitable power of 2, say 2^k, such that $2^k x$ is computed without over/underflow, and the requested calculations (in particular those in Algorithm 8) can be done safely with inputs $x' = 2^k x$ and y. We need to consider three cases:

- when $\mathrm{RN}_e(x \cdot y) = \pm\infty$, we choose $2^k = 1/2$. This case is dealt with in Section IV-B;
- when $|\mathrm{RN}_e(x \cdot y)| \leq 2^{e_{\min}+1} - 2^{e_{\min}-p+1}$, we choose $2^k = 2^{2p}$. This case is dealt with in Section IV-C;
- when $2^{e_{\min}+1} \leq |\mathrm{RN}_e(x \cdot y)| \leq 2^{e_{\min}+p}$, we choose $2^k = 2^p$. This case is dealt with in Section IV-D.

Note that when $|\mathrm{RN}_e(x \cdot y)| \leq 2^{e_{\min}+p}$ (i.e., in the last two cases mentioned just above), $\mathrm{RN}_0(xy - \mathrm{RN}_0(xy)) = 0$ *does not* mean that $xy - \mathrm{RN}_0(xy) = 0$. This slightly complicates the choice of the sign of b_0 when it is equal to zero. More precisely, when $b_0 = 0$, the sign of b_0 must be:

- the sign of a_0 (i.e., the sign of xy) when $xy - a_0 = 0$;
- the real sign of $xy - a_0$ otherwise.

Fortunately, in both cases, this is the same sign as the one of $\mathrm{RN}_e(xy - a_0)$, which is obtained using an FMA instruction. Hence, when $b_0 = 0$, one can return $(+0) \cdot \mathrm{RN}_e(xy - a_0)$ (the multiplication by $(+0)$ is necessary to handle the case $xy - a_0 = 2^{e_{\min}-p}$, for which functions RN_0 and RN_e differ).

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

In this case, which corresponds to Lines 2–11 in Algorithm 9, we need to know if we are in Case 2 (i.e., $|x \cdot y| = \Omega + 2^{e_{\max}-p}$) or Case 3 (i.e., $|x \cdot y| > \Omega + 2^{e_{\max}-p}$) of Theorem 3. Hence our problem reduces to checking if $|x \cdot y| = \Omega + 2^{e_{\max}-p}$. That problem is addressed easily. It suffices to compute $(a'_e, b'_e) = \mathrm{Fast2Mult}(0.5 \cdot x, y)$:

- If $|x \cdot y| = \Omega + 2^{e_{\max}-p}$, then $(x/2) \cdot y$ is computed by Fast2Mult without overflow, which allows one to check its equality with $\pm\left(\Omega + 2^{e_{\max}-p}\right)/2$;
- If it turns out that $|x \cdot y/2| \neq \left(\Omega + 2^{e_{\max}-p}\right)/2$ it suffices to return $a_0 = b_0 = \mathrm{RN}_e(x \cdot y)$: they will be infinities with the right sign.

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

In that case,

$$|x \cdot y - \mathrm{RN}_0(x \cdot y)| \leq 2^{e_{\min}-p}, \tag{5}$$

and thus $\mathrm{RN}_0\left(x \cdot y - \mathrm{RN}_0(x \cdot y)\right) = 0$, so we have to return $b_0 = 0$ (with the sign of $\mathrm{RN}_e(xy - a_0)$), and we only have to focus on the computation of $a_0 = \mathrm{RN}_0(x \cdot y)$. We also assume that $\mathrm{RN}_e(x \cdot y) \neq 0$ (otherwise, it suffices to return the pair $(0, 0)$, with the sign of xy). We therefore have

$$2^{e_{\min}-p} < |x \cdot y| < 2^{e_{\min}+1} - 2^{e_{\min}-p}. \tag{6}$$

Note that (6) implies

$$2^{e_{\min}+p} < |2^{2p} x \cdot y| < 2^{e_{\min}+2p+1} - 2^{e_{\min}+p}. \tag{7}$$

Let us first give the general reasoning behind the calculations of Lines 16–25 of Algorithm 9 (detailed proof will follow). Let a_e be $\mathrm{RN}_e(x \cdot y)$. Since the distance between consecutive floating-point numbers in the vicinity of xy is $2^{e_{\min}-p+1}$ (we are in the subnormal range), we have the following property:

- if $xy = a_e - \mathrm{sign}(a_e) \cdot 2^{e_{\min}-p}$ (i.e., we are in what we call "Halfway case 1" in Section II), then
$$a_0 = a_e - \mathrm{sign}(a_e) \cdot 2^{e_{\min}-p+1};$$
- otherwise, $a_0 = a_e$.

Therefore, we need to compare xy with $a_e - \mathrm{sign}(a_e) \cdot 2^{e_{\min}-p}$. This cannot be done straightforwardly, because xy is not necessarily representable exactly as the sum of two FP numbers (Lemma 2 does not hold). Instead, we will compare $2^{2p}xy$ with $2^{2p}a_e - \mathrm{sign}(a_e) \cdot 2^{e_{\min}+p}$. The first step for doing that will be to express $2^{2p}xy$ as the sum of two FP numbers t_1 and t_2 using Algorithm Fast2Mult. Then, to compare $t_1 + t_2$ with $2^{2p}a_e - \mathrm{sign}(a_e) \cdot 2^{e_{\min}+p}$, we will first show that the subtraction $t_3 = t_1 - 2^{2p}a_e$ is performed exactly, so that it will suffice to compare $t_2 + t_3$ with $-\mathrm{sign}(a_e) \cdot 2^{e_{\min}+p}$.

So, we successively compute (using FMA instructions)

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

First, t_1 can be computed without overflow:

- $|x \cdot y| < 2^{e_{\min}+1}$ and, since $x \cdot y \neq 0$, $|y| \geq 2^{e_{\min}-p+1}$. Therefore, $|x| \leq 2^{e_{\min}+1}/2^{e_{\min}-p+1} = 2^p$, therefore $|x| \cdot 2^{2p} < 2^{3p}$. Using (4), this implies that $|x \cdot 2^{2p}| < 2^{e_{\max}+1}$, hence $x \cdot 2^{2p}$ is a floating-point number;
- now, $|(2^{2p} \cdot x) \cdot y| < 2^{e_{\min}+1+2p} < 2^{3p} < 2^{e_{\max}+1}$ since $e_{\min} < 0$.

Therefore, $|2^{2p} \cdot x \cdot y|$ is below the overflow threshold, and

$$\left|t_1 - 2^{2p}xy\right| \leq \frac{1}{2}\mathrm{ulp}(t_1). \tag{8}$$

The fact that $t_2 = x \cdot y \cdot 2^{2p} - t_1$ comes from Lemma 2 and (7).

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, θ_3 can be computed with an FMA, or with an exact multiplication by 2^{2p} followed by a subtraction). From (7) we obtain

$$2^{e_{\min}+p} \leq |t_1| \leq 2^{e_{\min}+2p+1} - 2^{e_{\min}+p+1}$$

and $\mathrm{ulp}(t_1) \leq 2^{e_{\min}+p+1}$.

Since a_e (as any FP number) 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, θ_3 is a multiple of $\mathrm{ulp}(t_1)$.

Now, (5) gives $x \cdot y - 2^{e_{\min}-p} \leq a_e \leq x \cdot y + 2^{e_{\min}-p}$, from which 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, using (8),

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

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

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

so that

$$\left|t_1 - a_e \cdot 2^{2p}\right| \le \frac{1}{2}\mathrm{ulp}(t_1) + 2^{e_{\min}+p} \le \frac{1}{2}\mathrm{ulp}(t_1) + |t_1|.$$

Hence, θ_3 is a multiple of ulp(t_1) of magnitude less than or equal to $\frac{1}{2}$ulp$(t_1) + |t_1|$. Since t_1 is a multiple of ulp(t_1), we deduce that θ_3 is a multiple of ulp(t_1) of magnitude less than or equal to $|t_1|$. An immediate consequence (using Lemma 5) is that θ_3 is a floating-point number, which implies $t_3 = \theta_3$.

Now, we can compute $a_0 = \mathrm{RN}_0(x \cdot y)$. If

$$x \cdot y = a_e - \mathrm{sign}(a_e) \cdot 2^{e_{\min}-p}$$

then $a_0 = a_e - \mathrm{sign}(a_e) \cdot 2^{e_{\min}-p+1}$ (computed without error), otherwise $a_0 = a_e$. Hence we have to decide whether $x \cdot y = a_e - \mathrm{sign}(a_e) \cdot 2^{e_{\min}-p}$. This is equivalent to checking if $t_2 + t_3 = -\mathrm{sign}(a_e) \cdot 2^{e_{\min}+p}$. This can be done as follows: first note that since t_3 is a multiple of ulp(t_1) and $|t_2| \le \frac{1}{2}$ulp(t_1), either $t_3 = 0$ or $|t_3| > |t_2|$. Therefore, Lemma 1 can be applied to the addition of t_2 and t_3. Item 2 of that lemma tells us that if we define $z = \mathrm{RN}_e(t_2 + t_3)$, then $\mathrm{RN}_e(z - t_3) = z - t_3$. Therefore, checking if

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

is equivalent to checking if

$$z = -\mathrm{sign}(a_e) \cdot 2^{e_{\min}+p}$$
and
$$\mathrm{RN}_e(z - t_3) = t_2.$$

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

That case corresponds to Lines 26–34 of Algorithm 9. In that case, we know that $x \cdot y - \mathrm{RN}_0(x \cdot y)$ 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 $a_0 = \mathrm{RN}_0(x \cdot y)$ and $b_0 = \mathrm{RN}_0\left(x \cdot y - \mathrm{RN}_0(x \cdot y)\right)$.

We start by applying Algorithm 8 to the product $(2^p x) \cdot y$. That product can be computed without overflow:

- first, $|\mathrm{RN}_e(x \cdot y)| \le 2^{e_{\min}+p}$ implies
$$|xy| \le 2^{e_{\min}+p} + 2^{e_{\min}}.$$
Also, $2^{e_{\min}+1} \le |\mathrm{RN}_e(x \cdot y)|$ implies $y \ne 0$, therefore $|y| \ge 2^{e_{\min}-p+1}$. Thus
$$|x| = \left|\frac{xy}{y}\right| \le \frac{2^{e_{\min}+p} + 2^{e_{\min}}}{2^{e_{\min}-p+1}} = 2^{2p-1} + 2^{p-1}.$$
Therefore $|2^p x| \le 2^{3p-1} + 2^{2p-1} < 2^{e_{\max}}$ using (4). Thus $(2^p x)$ is below the overflow threshold.
- finally, $|xy| \le 2^{e_{\min}+p} + 2^{e_{\min}}$ implies
$$|(2^p x) \cdot y| \le 2^{e_{\min}+2p} + 2^{e_{\min}+p},$$
which is less than $2^{e_{\max}}$ from (4) and the fact that $e_{\min}$ is negative.

Algorithm 8 applied to $(2^p x) \cdot y$ returns two values, say a' and b', such that $a' = \mathrm{RN}_0(2^p x \cdot y)$ and $b' = 2^p x \cdot y - a'$. We immediately deduce using Lemma 6 that $2^{-p}a'$ is the expected $\mathrm{RN}_0(x \cdot y)$. Obtaining $\mathrm{RN}_0(x \cdot y - 2^{-p}a') = \mathrm{RN}_0(2^{-p}b')$ is slightly more tricky (Lemma 6 cannot be used because $|2^{-p}b'|$ can be strictly less than $2^{e_{\min}}$). We first compute $\beta = \mathrm{RN}_e(2^{-p}b')$. The number β is equal to the expected $\mathrm{RN}_0(2^{-p}b')$ unless we are in Halfway Case 1 of Section II, i.e., unless

$$\beta - (2^{-p}b') = \mathrm{sign}(\beta) \cdot 2^{e_{\min}-p} \tag{9}$$

in which case, one should replace β by $\beta - \mathrm{sign}(\beta) \cdot 2^{e_{\min}-p+1}$. Equation (9) is equivalent to

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

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.

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. The proof presented here is complex, with many particular cases to be considered. We have used the Coq proof assistant and the Flocq library [2] for our development towards Theorems 1 and 4.

Our formal proof is available as electronic appendix.

Note that we have aimed at genericity. In particular, we have tried to generalize the tie-breaking rule when possible. The precision and minimal exponent are hardly constrained as we only require $p > 1$ and $e_{\min} < 0$. As explained above, the radix must be 2 as Algorithm 4 does not hold for radix 10 (the definitions and first properties of ulp_H and RN_0 are generic though).

The formal proof quite follows the mathematical proof described above. Of course, we had to add several lemmas and to define RN_0 and its properties. This definition was very similar to the definition of rounding-to-nearest with tie-breaking away from zero defined by the standard for decimal arithmetic [8], and most of the proofs were nearly identical.

We then proved the correctness of Algorithm 4. In this case for $|a| > 2^{e_{\min}}$, the two RN_e roundings may be replaced with a rounding to nearest with any tie-breaking rule (they may even differ). Algorithm 5 is also proven. Similarly, the two RN_e roundings may in fact use any tie-breaking rule. The proof of Theorem 1 is then easily deduced, with Recomp using any two tie-breaking rules.

As on paper, the proof of Theorem 4 is more intricate, with many subcases, even if we handle only cases A (without the zeroes), C, and D. Here, the case split depends on the tie-breaking rule: the equalities may be either strict or large depending upon the tie-breaking rule. For the sake of simplicity, we chose to stick to the pen-and-paper proof and share the same case split. We then require some roundings to use tie-breaking to even. We were not able to generalize the proof at a reasonable cost to handle all tie-breaking rules. Nevertheless, the proof was formally done and we were able to prove the correctness of Theorems 1 and 4 (without considering overflows and signs of zeroes). The Coq statements are as follows (with few simplifications for the sake of readability). Note that `c1...c7` are arbitrary tie-breaking rules.

```
Definition Recomp := fun c1 c2 a b ⇒
  let z:= round_flt c1 (psi∗a) in
  let d:= round_flt c2 (z−a) in
    if (Req_bool (2∗b) d) then (z,−b) else (a,b).

Definition AA_Simple := fun c1 c2 x y ⇒
  let (x',y') := if (Rlt_bool (Rabs x) (Rabs y))
    then (y,x) else (x,y) in
  let (ae,be) := Fast2Sum x' y' in
     Recomp c1 c2 ae be.

Definition AM_Full := fun c1 c2 c3 c4 c5 c6 c7 x y ⇒
 let ae := round_flt ZnearestE (x∗y) in
  if (Rle_bool (Rabs ae) (bpow (emin+prec))) then
(* zero *)
  if (Req_bool ae 0) then (0,0) else
(* very small *)
   if (Rle_bool (Rabs ae) (bpow (emin+1) −
                   bpow (emin−prec+1))) then
        let t1 := round_flt c1 (x∗(y∗bpow (2∗prec))) in
        let t2 := round_flt c2 (x∗(y∗bpow (2∗prec)) − t1) in
        let t3 := round_flt c3 (t1 − ae∗bpow (2∗prec)) in
        let z := round_flt ZnearestE (t2+t3) in
        if (andb (Req_bool z (−sign(ae)∗bpow (emin+prec)))
                (Req_bool (round_flt ZnearestE (z−t3)) t2))
          then (ae−sign(ae)∗bpow (emin−prec+1),0)
          else (ae,0)
(* medium small*)
    else let t1 := round_flt c1 (x∗(y∗bpow prec)) in
        let t2 := round_flt c2 (x∗(y∗bpow prec) − t1) in
        let A' := Recomp emin prec c3 c4 t1 t2 in
        let a0 := round_flt c5 (bpow (−prec)∗fst A') in
        let beta := round_flt c6 (bpow (−prec)∗snd A') in
        let z:= round_flt c7 (bpow prec∗beta−snd A') in
        if (Req_bool z (sign beta∗bpow emin))
          then (a0, beta − sign(beta)∗bpow (emin−prec+1))
          else (a0,beta)
(*big*)
    else
      let be := round_flt ZnearestE (x∗y−ae) in
         Recomp c1 c2 ae be.

 Lemma AA_Simple_correct : forall c1 c2 x y,
    format_flt x → format_flt y →
    let (a0,b0) := AA_Simple c1 c2 x y in
       x+y = a0 + b0 ∧ a0 = round_flt Znearest0 (x+y).

 Lemma AM_Full_correct : forall c1 c2 c3 c4 c5 c6 c7 x y,
     format_flt x → format_flt y →
     let (a0,b0) := AM_Full c1 c2 c3 c4 c5 c6 c7 x y in
       a0 = round_flt Znearest0 (x∗y)
         ∧ b0 = round_flt Znearest0 (x∗y−a0).
```

A very important limitation of these proofs is that overflows, infinite numbers, and the signs of zeroes are not considered. We relied on the Flocq formalization that considers floating-point numbers as a subset of real numbers. Therefore, zeroes are merged and there are neither infinities, nor NaNs. It allows us to state the final theorems in the most understandable way: $a_0 = \mathrm{RN}_0(t)$ and $a_0 + b_0 = t$ or at least $b_0 = \mathrm{RN}_0(t - a_0)$ (with t being either the sum or product of two floating-point numbers).

We have tried to develop additional formal proofs taken all exceptional behaviors into account (especially NaNs and overflows). We have relied on a modified version of the `Binary` definitions of Flocq and we have defined the full algorithms, with comparisons on FP numbers and possible overflows. It made both the algorithms and their specifications more complicated and less readable. Moreover, the comprehensive formal proofs were out of reach, both by lack of support lemmas and by combinatorial explosion of the subcases for each and every operation (NaN, overflow, signed zero, and so on). This really calls for automations in Coq for handling FP numbers with exceptional behaviors, that is out of scope of this paper.

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, described below. We used an Intel Core i7-7500U x86_64 processor clocked at 2.7GHz under GNU/Linux (Debian 4.19.0-8-amd64), and the programs were compiled using GCC `(Debian 8.3.0-6) 8.3.0`, with the option `-O3 -march=native`. Our implementation, together with all testing and performance evaluation code, is available as additional material coming with this article; it can be downloaded at https://gitlab.com/cquirin/ieee754-2019-augmented-operations-reference-implementation and is archived at https://hal.archives-ouvertes.fr/hal-02137968.

Having an integer-based version of the augmented operations was important for comparison purposes, since there are no other implementations of these operations at the time we are writing this paper. Importantly enough, as we are going to see below, the integer-based algorithms are *not simpler* than the floating-point based algorithms presented in this paper. This is because the floating-point operations somehow automatically handle most special cases.

The integer-based emulation code proceeds as follows (assuming here we use the binary64 format):

1) The inputs x and y are checked for special values, such as NaN or infinities. For these special values, a regular FP addition or multiplication is executed, in order to get correct setting of flags and special values for the high order result a. For b, the same value is used. Similar logic is used for zero inputs.
2) Finite, non-zero, regular FP inputs x and y are decomposed into $(-1)^{s_x}\, 2^{E_x}\, M_x$ and $(-1)^{s_y}\, 2^{E_y}\, M_y$ where M_x and M_y are normalized integer significands stored on 64-bit integer variables. This step includes a normalization step using the integer hardware `lzc` (leading-zero count) instruction whenever x or y are subnormal. The hidden bit is made explicit.
3) For addition, the two couples (s_x, E_x, M_x) and (s_y, E_y, M_y) are ordered for exponents: $E_x \geq E_y$. When the exponent difference exceeds a certain limit, addition returns the ordered x and y as a and b as-is.
4) For all other cases, a temporary *exact* intermediate result $(-1)^s\, 2^E\, M$ is formed, where the integer significand M is stored on a 128-bit variable, which the compiler emulates on two 64-bit registers and the appropriate use of addition-with-carry and high/low-part multiplication machine instructions. For addition, this intermediate result is obtained by shifting the ordered M_x integer significand by $E_x - E_y$ places to the left and adding or subtracting M_y. For multiplication, it suffices to multiply M_x and M_y with a 64-by-64-gives-128 bit multiplication. The integer significand M is normalized (unless it becomes zero due to cancellation), which requires a branch and a 1-bit shift for multiplication and a leading-zero count and a larger, 128-bit shift (across 64-bit registers) for addition.
5) The intermediate result $(-1)^s\, 2^E\, M$ is rounded to the nearest IEEE754 binary64 value a, applying round-to-nearest-ties-to-zero rules. This rounding step is implemented as a “rounding to odd” [3] (with sticky bit) to $(-1)^s\, 2^{E'}\, M'$, where M' is a 64-bit integer significand, followed by the actual rounding to the binary64 format. This code sequence is quite complicated as it must cope with a multitude of possible cases, such as overflow, gradual or complete underflow as well as exact zeroes. A trace of overflow, underflow and inexact rounding is kept during this rounding step.
6) The high-order word a is decomposed again into $(-1)^{s_a}\, 2^{E_a}\, M_a$. If it is finite, this value is subtracted from the intermediate result $(-1)^s\, 2^E\, M$, which, after appropriate leading-zero count and normalization shift, yields to $(-1)^{s_\ell}\, 2^{E_\ell}\, M_\ell$, where M_ℓ is an integer significand stored on a 64-bit integer. This value $(-1)^{s_\ell}\, 2^{E_\ell}\, M_\ell$ is given to the same rounding code as the one used above, which yields b with round-to-nearest-ties-to-zero and a trace of overflow[1], underflow and inexact rounding.
7) Out of both traces for overflow, underflow and inexact a global IEEE754 flag setting is computed and applied to IEEE754 flag registers by executing a dummy FP operation that make the appropriate flags be raised.

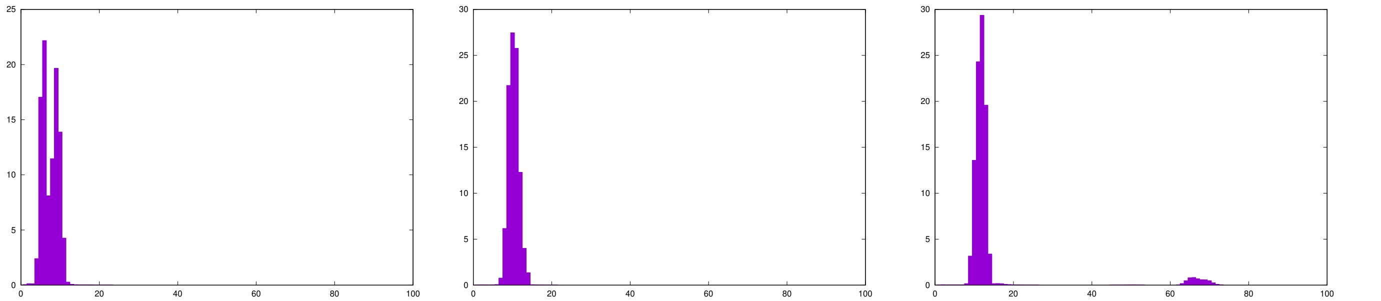
The emulation code has the advantage of being the only version of our algorithms that is able to set the IEEE754 flags correctly and to be insensible to the prevailing IEEE754 rounding direction attribute. The FP-based algorithms may set the inexact flag as well as other flags spuriously. They do require the IEEE754 rounding direction attribute to be round-to-nearest-ties-to-even, which is the default. However, these advantages of the emulation code come at a significant cost: as we are going to see, the emulation code is on average 1.5 to 20 times slower than the FP-based algorithms. The emulation code also has the disadvantage of being rather complex. The precise rounding logic for round-to-nearest-ties-to-zero for example is quite complicated, which required extra care at its development to overcome its error-prone nature.

The statistical distribution of the number of cycles used by our algorithms (using 10^6 samples, with the distributions of the inputs described below) is given:

- for the augmentedAddition algorithms, in Figures 4a for Algorithm 6 (AA-Simple), 4b for Algorithm 7 (AA-Full), and 4c for the integer-based emulation of augmentedAddition;
- for the augmentedMultiplication algorithms, in Figures 5a for Algorithm 8 (AM-Simple), 5b for Algorithm 9 (AM-Full), and 5c for the integer-based emulation of augmentedMultiplication.

For each input sample, formed by an input couple (x, y), the number of cycles is measured as follows: the function implementing one of our algorithms is run on x and y. Its execution time is measured by reading off the x86 Time

[1] which is impossible in this step



(a) Algorithm 6 (all simple inputs) (b) Algorithm 7 (all inputs) (c) Emulation algorithm (all inputs)

Fig. 4: Statistical distribution of the number of cycles for the addition algorithms

Step Counter with the `rdtsc` instruction before and after the execution of the function. Before reading the Time Step Counter, the CPU's pipeline is serialized by executing a dummy `cpuid` instruction, which Intel documents as having this serialization effect. As the time measurements obtained by subtracting the *before* Time Step Counter's value from the *after* counter's value also include the time for pipeline serialization, execution of the `rdtsc` instruction and the function call itself, the same measurement is repeated with an empty function that has the same signature as the actual function to time. The empty function's measured execution time is subtracted off the actual function's measured execution time. This yields a raw execution time sample in cycles. All raw execution times that are less than 1 cycle are discarded and the timing procedure is repeated. The measurement yielding to positive raw execution times is repeated 100 times, an average and maximum value is computed. If the average value does not differ from the maximum value by more than a certain amount (typically 25%), the average value is taken as the *execution time in cycles* for this input sample (x, y). The measurements are taken on a CPU not executing any other heavy jobs, after a preheating phase for the instruction cache. The Linux scheduler is configured to keep the process on one CPU core as long as possible. Core migration is anyway filtered out by our testing strategy as it yields either negative raw timings or raw timings that are clear outliers. Overall, 10^6 different samples are timed, which yields to the histograms illustrated in Figures 4 and 5 as well as the average values (averaging over all 10^6 inputs) reported in Tables V and VI.

The different input samples (x, y) are produced as follows:

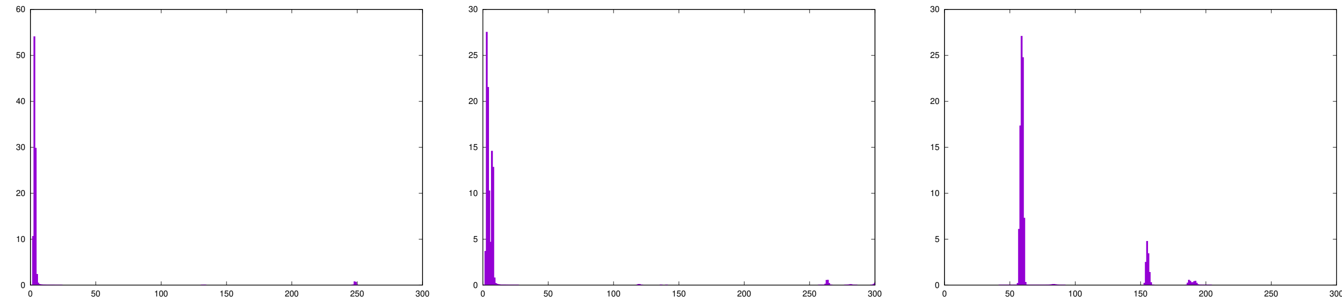
- The *all cases* input samples are produced with a pseudo-random number generator such that all signs, exponents and significands are uniformly distributed among all binary64 FP values (x, y) that are finite numbers such that the resulting outputs (a, b) are finite as well. The filtering of whether or not a candidate (x, y) produces finite outputs (a, b) uses our integer-based emulation code to compute (a, b) out of the candidate (x, y).
- The *all simple cases* input samples are produced by filtering from the set of the *all cases* samples the ones for which the simple FP-based Algorithms 6 and 8 produce bit-correct results (including correct signs for zeroes). The decision of whether or not a candidate sample (x, y) is an *all simple case* is taken by comparing the output of the respective FP-based simple algorithm with the one of our integer-based emulation code.
- The *halfway cases* input samples (x, y) are such that the respective outputs (a, b) are finite FP numbers (zero and non-zero but no overflows) and $x + y$ resp. $x \times y$ is precisely in the middle between two binary64 FP numbers. They are produced as follows:
  - For addition in binary64, a 54 bit odd integer number N is produced[2] along with a uniformly distributed exponent E. A uniformly distributed split-point $\sigma \in \{1, \dots, 53\}$ is then produced. Using σ, N is cut into two parts which, along with E and σ, yield to the exponents and significands of candidates x and y. Half of the values x and y are swapped. The "cutting" process is not actually just a bit cut but done in such a way that both x and y can be both negative or positive.
  - For multiplication on binary64, two uniformly distributed 27 bit odd integers are produced along with uniformly distributed exponents.

  The candidate (x, y) samples are checked for finiteness and whether or not they produce finite outputs (a, b); all candidates that do not satisfy these constraints are filtered out.
- The *halfway simple cases* are obtained by filtering from the *halfway cases* the ones for which the simple FP-based algorithms produce correct result, similarly as to how the *all simple cases* are obtained.

The average timings are given in the first half of Table V for the augmentedAddition algorithms, and the first half of Table VI for the augmentedMultiplication multiplication algorithms. In each of these tables, the second half 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 augmented-Multiplication, Algorithm 9 is significantly better, except on very rare cases (at the extreme right of Figure 5b). In all cases, the "simple" versions of the algorithms (Algorithms 6 and 8) are significantly faster in the cases when they work. They may also be significantly slower when they do not work, due to

[2]This actually means that only 52 bits are random, as the 53rd bit and the 0th bit must be set.



(a) Algorithm 8 (all simple inputs) (b) Algorithm 9 (all inputs) (c) Emulation algorithm (all inputs)

Fig. 5: Statistical distribution of the number of cycles for the multiplication algorithms

hardware constraints; for instance, Algorithm 8 may make the hardware produce and consume subnormal FP values, which is a slow process on some architectures.

TABLE V: Average timings in cycles for the augmentedAddition algorithms

Algorithm	♯ of cycles
Algorithm 6 (addition, all simple cases)	7.66
Algorithm 7 (addition, all cases)	10.46
Integer-based emulation of addition (all cases)	14.70
Algorithm 6 (addition, halfway simple cases)	7.74
Algorithm 7 (addition, halfway cases)	9.82
Integer-based emulation of addition (halfway cases)	4.99

TABLE VI: Average timings in cycles for the augmentedMultiplication algorithms

Algorithm	♯ of cycles
Algorithm 8 (multiplication, all simple cases)	7.62
Algorithm 9 (multiplication, all cases)	13.45
Integer-based emulation of multiplication (all cases)	75.65
Algorithm 8 (multiplication, halfway simple cases)	3.15
Algorithm 9 (multiplication, halfway cases)	4.99
Integer-based emulation of multiplication (halfway cases)	58.34

Conclusion

We have presented and implemented algorithms that allow one to emulate the newly suggested "augmented" floating-point operations using the classical, rounded-to-nearest ties-to-even, operations. The algorithms are very simple in the general case. Special cases are slightly more involved but will remain infrequent in most applications. These algorithms compare favorably with an integer-based emulation of the augmented operations. Furthermore, the availability of both tests for special cases and formal proofs covering normal and underflow cases (despite the limitations presented in Section V) gives much confidence in these algorithms.

Acknowledgement

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

References

[1] S. Boldo, S. Graillat, and J.-M. Muller. On the robustness of the 2Sum and Fast2Sum algorithms. *ACM Transactions on Mathematical Software*, 44(1):4:1–4:14, July 2017.

[2] S. Boldo and G. Melquiond. *Computer Arithmetic and Formal Proofs*. ISTE Press – Elsevier, 2017.

[3] Sylvie Boldo and Guillaume Melquiond. Emulation of FMA and correctly rounded sums: proved algorithms using rounding to odd. *IEEE Transactions on Computers*, 57(4):462–471, April 2008.

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

[5] J. Demmel, P. Ahrens, and H. D. Nguyen. Efficient reproducible floating point summation and BLAS. Technical Report UCB/EECS-2016-121, EECS Department, University of California, Berkeley, June 2016.

[6] J. 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. Springer-Verlag, Berlin, September 1999.

[7] J. R. Hauser. Handling floating-point exceptions in numeric programs. *ACM Transactions on Programming Languages and Systems*, 18(2):139–174, 1996.

[8] IEEE. *IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2019)*. July 2019.

[9] C.-P. Jeannerod, J.-M. 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.

[10] W. Kahan. Lecture notes on the status of IEEE-754. Available at http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF, 1997.

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

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

[13] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2018.

[14] 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.

[15] E. J. Riedy and J. Demmel. Augmented arithmetic operations proposed for IEEE-754 2018. In *25th IEEE Symposium on Computer Arithmetic, Amherst, MA, USA*, pages 45–52, June 2018.

[16] S. M. Rump. Ultimately fast accurate summation. *SIAM Journal on Scientific Computing*, 31(5):3466–3502, January 2009.

[17] P. H. Sterbenz. *Floating-Point Computation*. Prentice-Hall, Englewood Cliffs, NJ, 1974.