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

To cite this version:
Sylvie Boldo, Christoph 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-02137968

HAL Id: hal-02137968
https://hal.archives-ouvertes.fr/hal-02137968
Submitted on 23 May 2019

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 IEEE 754 Standards Committee for Floating-Point Arithmetic is recommending, in its draft standard, that new “augmented” operations should be added in the next release of the standard. 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 IEEE 754 Standards Committee for Floating-Point Arithmetic is recommending¹, in its draft standard, that new “augmented” operations should be added in the next release of the standard [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 described in [4]. 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. The minimum floating-point exponent is \(\epsilon_{\text{min}}\), so that \(2^{\epsilon_{\text{min}}}\) is the smallest positive normal number and \(2^{\epsilon_{\text{min}} - p + 1}\) is the smallest positive floating-point number. The maximum floating-point exponent is \(\epsilon_{\text{max}}\). The largest positive floating-point number is \(\Omega = (2 - 2^{-p+1}) \cdot 2^{\epsilon_{\text{max}}}\). 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 [7]:

\[
\text{RN}_e(t) = \begin{cases} 
\text{the one whose rightmost bit of the significand is a zero. If } |t| \geq \Omega + 2^{\epsilon_{\text{max}}-p} \text{ then } \\
\text{RN}_e(t) = \infty, \text{ with the same sign as } t.
\end{cases}
\]

We will also assume that an FMA (fused multiply-add) instruction is available.

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:

\[
\text{RN}_0(t) \text{ (where } t \text{ is a real number) is the floating-point number nearest } t. \text{ If } t \text{ is exactly halfway between two consecutive floating-point numbers, } \\
\text{RN}_0(t) \text{ is the one with smaller magnitude. If } |t| > \Omega + 2^{\epsilon_{\text{max}}-p} \text{ then } \text{RN}_0(t) = \infty, \text{ with the same sign as } t.
\]

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

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure1.png}
\caption{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 exactly halfway between two consecutive FP numbers: it is rounded to the one that has the smallest magnitude.}
\end{figure}

The augmented operations are required to behave as follows:

- **augmentedAddition** 
  \(x, y\) delivers \((a_0, b_0)\) such that \(a_0 = \text{RN}_0(x + y)\) and, where \(a_0 \notin \{\pm\infty, \text{NaN}\}\), \(b_0 = (x+y) - a_0\) (with the same sign as \(a_0\) when both are 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 \(a_0 \notin \{\pm\infty, \text{NaN}\}\), \(b_0 = \text{RN}_0((x \cdot y) - a_0)\). Note that in some corner cases

¹Information can be found at http://754r.uchtest.org.
(see below), \( b_0 \) 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 [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 on a system compliant with IEEE 754-2008, just by using rounded-to-nearest-even floating-point operations.

Let \( t \) be the exact sum \( x + y \) (if we consider implementing augmentedAddition) or the exact product \( xy \) (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 \( a_e = RN_e(t) \) 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.

In the following, we need to use a definition inspired from Harrison’s definition [5] 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 \( \text{ulp}(t) = \text{ulp}_H(t) = 2^{e_{\text{min}}-p+1} \).

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 \( \text{RN}_e(t) \). 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 \( t \), we will have to add \( \text{ulp}_H(\text{RN}_e(t)) \)); 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 present the classical Algorithms Fast2Sum, 2Sum, and Fast2Mult.

If \( x = 0 \) or \( y = 0 \), or if the floating-point exponents \( e_x \) and \( e_y \) satisfy \( e_x + e_y \geq e_{\min} + p - 1 \), then the two variables \( a_e \) and \( b_e \) returned by Algorithm 1 (Fast2Sum) satisfy \( a_e + b_e = x + y \). Hence, \( b_e \)

**Algorithm 1: – Fast2Sum\((x,y)\).** The Fast2Sum algorithm [3].

\[
\begin{align*}
a_e &\leftarrow \text{RN}_e(x + y) \\
y' &\leftarrow \text{RN}_e(a_e - x) \\
b_e &\leftarrow \text{RN}_e(y - y')
\end{align*}
\]

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 “\( e_x + e_y \geq e_{\min} + p - 1 \)” may be hard to check. However, if \( |x| \geq |y| \) then that condition is satisfied. Algorithm 1 is immune from spurious overflow: it was proved in [1] 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].

\[
\begin{align*}
a_e &\leftarrow \text{RN}_e(x + y) \\
x' &\leftarrow \text{RN}_e(a_e - y) \\
y' &\leftarrow \text{RN}_e(a_e - x) \\
\delta_x &\leftarrow \text{RN}_e(x - x') \\
\delta_y &\leftarrow \text{RN}_e(y - y') \\
b_e &\leftarrow \text{RN}_e(\delta_x + \delta_y)
\end{align*}
\]

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 from spurious overflow: if \( |x| \neq \Omega \) and the addition \( \text{RN}_e(x + y) \) does not overflow then the other five operations cannot overflow [1].

Let \( x \) and \( y \) be two floating-point numbers, with exponents \( e_x \) and \( e_y \), respectively, such that \( e_x + e_y \geq e_{\min} + p - 1 \). Define \( a_e = \text{RN}_e(xy) \). The number \( b_e = xy - a_e \) 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 \( |\text{RN}_e(x \cdot y)| > 2^{e_{\min}+p} \).

**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 \( \text{RN}_e(xy - a_e) \).

\[
\begin{align*}
a_e &\leftarrow \text{RN}_e(xy) \\
b_e &\leftarrow \text{RN}_e(x \cdot y - a_e)
\end{align*}
\]

We will also use the following results, due to Hauser [6] and Sterbenz [16] (the proofs are straightforward, see [12]).

**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 \( \text{RN}_H(x - y) = x - y \).

As said above, when \( \text{RN}_0(t) \) and \( \text{RN}_e(t) \) differ, \( \text{RN}_0(t) \) is obtained by subtracting \( \text{sign}(t) \cdot \text{ulp}_H(H) \) from \( \text{RN}_e(t) \). Therefore, we need to be able to compute function \( \text{sign}(a) \cdot \text{ulp}_H(a) \). If \( |a| > 2^\text{emin} \), this can be done using Algorithm 4 below.

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

\[
z \leftarrow \text{RN}_e(\psi a) \\
\delta \leftarrow \text{RN}_e(a - z) \\
\text{return } \delta
\]

The fact that Algorithm 4 returns \( \text{sign}(a) \cdot \text{ulp}_H(a) \) when \(|a| > 2^\text{emin} \) is a direct consequence of [15, Lemma 3.6]. See also [8]. Note that when \( a > 2^\text{emin} \), \( z \) equals \( \text{pred}(a) \). If \( a \) is subnormal (i.e., \(|a| < 2^\text{emin} \)), then Algorithm 4 returns 0. Interestingly enough, Algorithm 4 returns the same result if we change the tie-breaking rule. 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 \(|a_e| > 2^\text{emin} \). These numbers may have been preliminarily generated by the 2Sum, Fast2Sum or Fast2Mult algorithms. We want to deduce from them two floating-point numbers \( a_0 \) and \( b_0 \) such that \( a_0 = \text{RN}_0(t) \), and \( a_0 + b_0 = t \).

![Image of transformation](image)

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

**ALGORITHM 5:** \( \text{Recomp}(a_e, b_e) \). From two FP numbers \( a_e \) and \( b_e \) such that \( a_e = \text{RN}_e(a_e + b_e) \) and \(|a_e| > 2^\text{emin} \), 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 \( \psi = 1 - 2^{-p} \).

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

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

In the next sections, we examine how Algorithm 5 can be used to compute \( \text{augmentedAddition}(x, y) \) and \( \text{augmentedMultiplication}(x, y) \).

III. USE OF ALGORITHM RECOMP FOR IMPLEMENTING AUGMENTED ADDITION

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, that returns a correct result when no exception occurs.

**ALGORITHM 6:** \( \text{AA-Simple}(x, y) \): computes \( \text{augmentedAddition}(x, y) \) when no exception occurs.

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

We have,

**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 draft standard;
3) if \(|x + y| = \Omega + 2^\text{emin} \cdot \psi = (2 - 2^{-p}) \cdot 2^\text{emax} \) then \( a_0 = \pm \infty \) and \( b_0 \) is either NaN or \( \pm \infty \), whereas the correct values...
would have been $a_0 = \pm \Omega$ and $b_0 = \pm 2^{\text{max} - p}$ (with the appropriate signs);
4) if $|x + y| > \Omega + 2^{\text{max} - p}$ 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 draft standard requires $a_0 = b_0 = \infty$ (with the same sign as $x + y$).

The first property listed 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 $|a_e| > 2^{\text{min}}$ then Recomp($a_e, b_e$) gives the expected result;
- if $|a_e| \leq 2^{\text{min}}$ 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 claimed in the second property listed 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 property listed 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 draft 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 are important in the target application, one has to add to add the following lines to Algorithm 6 after Line 5:

```java
if $a_0 = 0$ then
    $b_0 \leftarrow a_0$
end if
```

Property 3 of Theorem 1 is immediate by applying Algorithm 6 to the corresponding input value.

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

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

**Table I**

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

<table>
<thead>
<tr>
<th>$(a_e, b_e)$ obtained through 2Sum</th>
<th>$(a_e, b_e)$ obtained through Fast2Sum</th>
<th>$(a_e, b_e)$ obtained through Fast2Mult</th>
<th>Result required by draft standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0$</td>
<td>+∞</td>
<td>+∞</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>$b_0$</td>
<td>NaN</td>
<td>$+\infty$</td>
<td>$+\infty$</td>
</tr>
</tbody>
</table>

**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 $a_0 = 0$ then
7: $b_0 \leftarrow a_0$
8: else if $a_e = \pm \infty$ then
9: $(a'_e, b'_e) \leftarrow \text{Fast2Sum}(0.5x, 0.5y)$
10: if $(a'_e = 2^{\text{max}}$ and $b'_e = -2^{\text{min} - p + 1})$ or $(a'_e = -2^{\text{max}}$ and $b'_e = +2^{\text{min} - p + 1})$ then
11: $a_0 \leftarrow \text{RN}_e(2^{\text{min}} \cdot (2 - 2^{-p + 1}))$
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 $(a_0, b_0)$

We have,

**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 $a_0 \neq 0$ at Line 6 of the algorithm and $a_e \neq \pm \infty$, Algorithm 7 behaves exactly as Algorithm 6;
- 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 $|x + y| = \Omega + 2^{\text{max} - p} = (2 - 2^{-p}) \cdot 2^{\text{max}}$, in which case we must return $a_0 = \pm \Omega$ and $b_0 = \pm 2^{\text{max} - p}$ (with the appropriate signs), or $|x + y| > \Omega + 2^{\text{max} - p}$, 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 in Lines 8 to 16 of Algorithm 7: we divide $x$ and $y$ by 2 so that if
$|x + y| = \Omega + 2^{\text{max} - p}$, then $x/2 + y/2$ is computed by Fast2Sum without overflow, which makes it possible to compare it with $\pm (\Omega + 2^{\text{max} - p})/2$.

IV. USE OF ALGORITHM RECOMP FOR IMPLEMENTING AUGMENTEDMULTIPLICATION

A. General case

From two input floating-point numbers $x$ and $y$, we wish to compute $\text{RN}_0(xy)$ and $xy - \text{RN}_0(xy)$ (or, merely, $\text{RN}_0[xy - \text{RN}_0(xy)]$ when $xy - \text{RN}_0(xy)$ is not a floating-point number). As we did for FastAdd, 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 product of two floating-point numbers is less than or equal to $2^{\text{min} + p}$, it may not be exactly representable by the sum of two floating-point numbers.

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

1. $(a_c, b_c) \leftarrow \text{Fast2Mult}(x, y)$
2. $(a_0, b_0) \leftarrow \text{Recomp}(a_c, b_c)$
3. return $(a_0, b_0)$

We have,

**Theorem 3.** If $2^{\text{min} + p} < |\text{RN}_e(xy), \| < +\infty$ (i.e., $2^{\text{min} + p} + 2^{\text{min} + 1} < |\text{RN}_e(xy), \| \leq \Omega$) then AM-Simple($x, y$) = augmentedMultiplication($x, y$).

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

- $(a_c, b_c) = \text{Fast2Mult}(x, y)$ gives $a_c + b_c = xy$;
- $a_c > 2^{\text{min}}$

therefore Recomp($a_c, b_c$) returns the expected result.

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

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

B. First special case: if $\text{RN}_e(xy) = \pm \infty$

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

- either $|xy| = \Omega + 2^{\text{max} - p} = (2 - 2^{-p}) \cdot 2^{\text{max}}$, in which case we must return $a_0 = \pm \Omega$ and $b_0 = \pm 2^{\text{max} - p}$ (with the appropriate signs) whereas one easily checks that Algorithm 8 delivers a wrong result;
- or $|xy| > \Omega + 2^{\text{max} - p}$, 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 (and very similarly to what we did for augmented addition). It suffices to compute $(a_c', b_c') = \text{Fast2Mult}(0.5 \cdot x, y)$. If $|xy| = \Omega + 2^{\text{max} - p}$, then $xy/2$ is computed by Fast2Mult without overflow, which makes it possible to compare it with $\pm (\Omega + 2^{\text{max} - p})/2$. If it turns out that $|xy/2| \neq (\Omega + 2^{\text{max} - p})/2$ we must return $a_0 = b_0 = \text{RN}_e(xy)$.

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

C. Second special case: if $|\text{RN}_e(xy), \| \leq 2^{\text{min} + 1} - 2^{\text{min} - p + 1}$

In that case, the floating-point number nearest $xy - \text{RN}_0(xy)$ is zero, so we only have to focus on the computation of $\text{RN}_0(xy)$. We also assume that $\text{RN}_e(xy) \neq 0$ (otherwise, it suffices to return $\text{RN}_0(xy) = 0$). We therefore have

$$2^{\text{min} - p} < |xy| < 2^{\text{min} + 1} - 2^{\text{min} - p}.$$  

(1)

Let $a_c$ be $\text{RN}_e(xy)$, and let us successively compute (using FMA instructions)

$$t_1 = \text{RN}_e(xy \cdot 2^{2p});$$
$$t_2 = \text{RN}_e(xy \cdot 2^{2p} - t_1) = xy \cdot 2^{2p} - t_1;$$
$$t_3 = \text{RN}_e(t_1 - a_c \cdot 2^{2p}).$$

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

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

Now, from $xy - 2^{\text{min} - p} \leq |a_c| \leq xy + 2^{\text{min} - p}$, we deduce $xy \cdot 2^{2p} - 2^{\text{min} - p} \leq |a_c| \cdot 2^{2p} \leq xy \cdot 2^{2p} + 2^{\text{min} + p}$, which implies $t_1 - \frac{1}{2} \text{ulp}(t_1) - 2^{\text{min} - p} \leq |a_c| \cdot 2^{2p} \leq t_1 + \frac{1}{2} \text{ulp}(t_1) + 2^{\text{min} + p}$, so that

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

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 $a_0 = \text{RN}_0(xy)$. If $xy = a_c - \text{sign}(a_c) \cdot 2^{\text{min} - p}$ then $a_0 = a_c - \text{sign}(a_c) \cdot 2^{\text{min} - p + 1}$ (computed without error), otherwise $a_0 = a_c$. Hence we have to check if $xy = a_c - \text{sign}(a_c) \cdot 2^{\text{min} - p}$. This is equivalent to checking if $t_2 + t_3 = -\text{sign}(a_c) \cdot 2^{\text{min} + p}$. 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, if follows from the properties of Algorithm 1 (Fast2Sum) that checking if $t_2 + t_3 = -\text{sign}(a_c) \cdot 2^{\text{min} + p}$ is equivalent to checking if $z := \text{RN}_e(t_2 + t_3) = -\text{sign}(a_c) \cdot 2^{\text{min} + p}$ and $\text{RN}_e(z - t_3) = t_2$. 


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

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

First, we apply Algorithm 8 to the product $(2^p x) \cdot y$. This gives two values, say $a'$ and $b'$, such that $a' = \text{RN}_0(2^p xy)$ and $b' = 2^p xy - a'$. We immediately deduce that $2^{-p}a'$ is the expected $\text{RN}_0(xy)$. Obtaining $\text{RN}_0(xy - 2^{-p}a') = \text{RN}_0(2^{-p}b')$ 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

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

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

All this gives Algorithm 9 and Theorem 4, below.

Theorem 4. Algorithm 9 always returns augmented-Multiplication$(x, y)$.

Conclusion

We have presented algorithms that allow one to implement 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 unfrequent in most applications. Note that formal proofs of the general case using Coq and the Flocq library [2] are under development.

References

