Emulation of the FMA in rounded-to-nearest floating-point arithmetic

Stef Graillat, Jean-Michel Muller

Emulation of the FMA in rounded-to-nearest floating-point arithmetic

Stef Graillat* Jean-Michel Muller†

May 14, 2024

Abstract

We present an algorithm that allows to emulate the fused multiply-add (FMA) instruction in binary floating-point arithmetic, using only rounded-to-nearest floating-point additions, multiplications, and comparisons.

Keywords. Floating-point arithmetic, FMA, fused multiply-add, Error-free transforms, double-word arithmetic.

1 Introduction

The *fused multiply-add (FMA)* instruction evaluates an expression of the form $ab + c$, where a, b, and c are floating-point numbers, with one final rounding only. It appeared in 1990 in the IBM POWER instruction set [6], and its specification was incorporated in the 2008 version of the IEEE-754 Standard for Floating-Point Arithmetic [1]. It facilitates the software implementation of correctly rounded division and square root [14, 7], and, in general, allows for faster and more accurate evaluation of dot products and polynomials.

To be able to run programs that use FMAs on architectures without an FMA instruction, it may be interesting to have algorithms that emulate that instruction. We are interested in emulating the FMA for *rounded-to-nearest* arithmetic. It is always possible to do that using integer arithmetic and masks. However, for portability and performance reasons, one may wish to use "high level" algorithms, that only use floating-point operations. This could be done using an algorithm devised by Boldo and Melquiond [5]. Unfortunately, their algorithm requires a "round to odd" rounding function that is not yet available on current processors and is not specified by the current version of the IEEE 754 Standard for Floating-Point Arithmetic [1, 4]. Kornerup et al. [12] show how one could emulate rounded-to-odd additions/subtractions using arithmetic

*Sorbonne Université, CNRS, LIP6, Paris, France
†CNRS, Laboratoire LIP, Université de Lyon, Lyon, France

operations with round-to-$+\infty$ and round-to-$-\infty$ rounding functions. In principle, this makes it possible to use the Boldo-Melquiond algorithm and emulate rounded-to-nearest FMAs. However, changing the rounding function remains a complex and costly procedure.

The purpose of this paper is to find how an FMA instruction can be evaluated using only rounded-to-nearest floating-point additions, subtractions, multiplications and tests.

Throughout the paper, we assume a binary, precision-p floating-point (FP) arithmetic. Unless otherwise specified, it is assumed that the exponent range is unbounded. This implies that the results presented here apply to conventional binary floating-point arithmetic provided that underflow and overflow do not occur. We assume that the rounding function is *round-to-nearest, ties-to-even*, noted RN, which is the default in IEEE 754 arithmetic. The *unit round-off* is $u = 2^{-p}$. It is an upper bound on the relative error due to rounding. This implies that when an arithmetic operation $x \top y$ is performed (with $\top \in \{+, -, \times, \div\}$) , the computed result z satisfies

$$(1 - u) \cdot |(x \top y)| \leq z = |\mathrm{RN}(x \top y)| \leq (1 + u) \cdot |(x \top y)|. \tag{1}$$

We will say that x is a *double-word*[1] (DW) number if it is the unevaluated sum $x_h + x_\ell$ of two floating-point numbers x_h and x_ℓ such that $x_h = \mathrm{RN}(x)$. Some algorithms for manipulating double-word numbers are presented and analyzed in [10].

1.1 Some classical results of floating-point arithmetic used in this paper

In this section, we just briefly present the results needed in the sequel of the paper. More detailed presentations and proofs can be found in [16].

In order to emulate an FMA instruction using FP multiplications and additions, it is necessary to conduct an analysis of the errors associated with these operations. Although very useful, the relative error bound (1) is not the final word:

- First, some operations are *exact*. A straightforward example is the case of multiplications and divisions by powers of 2. Another, less intuitive, example is the case of the subtraction of two numbers that are close enough to each other, as presented in Section 1.1.1;
- Second, a simple analysis shows that the error of an FP addition or multiplication is an FP number.[2] See for instance [2, 3]. Furthermore, these

[1] We frequently see the name "double double" in the literature. We prefer "double word" because there is no reason to systematically assume that the underlying format is double precision/binary64.

[2] Concerning addition, this is true only when the rounding function is *round-to-nearest*, which we have assumed here.

errors can be computed, using relatively simple algorithms, called *Error-Free Transforms* in the literature [17], presented in §1.1.2 (for addition) and §1.1.3 (for multiplication).

1.1.1 Sterbenz's theorem

Sterbenz's theorem is extremely useful in error analysis. For instance, the proof of the double-word algorithms presented in [10] heavily relies on Sterbenz's theorem.

Theorem 1.1 (Sterbenz Theorem [18])**.** *Let a, b be FP numbers. If $\frac{a}{2} \le b \le 2a$ then $a - b$ is an FP number. This implies that the subtraction $a - b$ will be performed exactly in FP arithmetic.*

1.1.2 The Fast2Sum and 2Sum algorithms

Algorithm 1 – **Fast2Sum**(a, b). The Fast2Sum algorithm [8].

$s \leftarrow \text{RN}(a + b)$
$z \leftarrow \text{RN}(s - a)$
$t \leftarrow \text{RN}(b - z)$
return (s, t)

If the floating-point exponents e_a and e_b of a and b are such that $e_a \ge e_b$ then t is the error of the floating-point addition $\text{RN}(a+b)$ (i.e., the double word (s, t) is exactly equal to $a + b$). The condition on the exponents may be difficult to check, but it is satisfied if $|a| \ge |b|$.

Algorithm 2 – **2Sum**(a, b). The 2Sum algorithm [15, 11].

$s \leftarrow \text{RN}(a + b)$
$a' \leftarrow \text{RN}(s - b)$
$b' \leftarrow \text{RN}(s - a')$
$\delta_a \leftarrow \text{RN}(a - a')$
$\delta_b \leftarrow \text{RN}(b - b')$
$t \leftarrow \text{RN}(\delta_a + \delta_b)$
return (s, t)

For all FP numbers a and b, t is the error of the floating-point addition $\text{RN}(a + b)$.

1.1.3 The Dekker-Veltkamp multiplication algorithm

If an FMA instruction is available, then the error of an FP multiplication is very easy and fast to compute: the error of the multiplication $\pi_h = \text{RN}(ab)$ is $\pi_\ell = \text{RN}(ab - \pi_h)$. Since our goal here is to emulate an FMA instruction, we obviously cannot assume that such an instruction is already available, so we must use a more complex algorithm, Algorithm 4 below, due to Dekker and Veltkamp [8]. In order to compute the product ab "exactly", Algorithm 4 must first "split" the input operands a and b into sub-operands of precision around $p/2$, so that the product of two such sub-operands can be representable exactly in precision-p floating-point arithmetic (and is therefore obtained by a simple floating-point multiplication). This preliminary splitting is done by Algorithm 3. For a proof of these algorithms, see [16].

Algorithm 3 – **Split**(x, s). Veltkamp's splitting algorithm. Returns a pair (x_h, x_ℓ) of FP numbers such that the significand of x_h fits in $s - p$ bits, the significand of x_ℓ fits in $s - 1$ bits, and $x_h + x_\ell = x$.

Require: $K = 2^s + 1$
Require: $2 \le s \le p - 2$
$\gamma \leftarrow \mathrm{RN}(K \cdot x)$
$\delta \leftarrow \mathrm{RN}(x - \gamma)$
$a_h \leftarrow \mathrm{RN}(\gamma + \delta)$
$a_\ell \leftarrow \mathrm{RN}(x - a_h)$
return (x_h, x_ℓ)

Algorithm 4 – **DekkerProd**(a, b). Dekker's product. Returns a pair (π_h, π_ℓ) of FP numbers such that $\pi_h = \mathrm{RN}(ab)$ and $\pi_h + \pi_\ell = ab$.

Require: $s = \lceil p/2 \rceil$
$(a_h, a_\ell) \leftarrow \mathrm{Split}(a, s)$
$(b_h, b_\ell) \leftarrow \mathrm{Split}(b, s)$
$\pi_h \leftarrow \mathrm{RN}(a \cdot b)$
$t_1 \leftarrow \mathrm{RN}(-\pi_h + \mathrm{RN}(a_h \cdot b_h))$
$t_2 \leftarrow \mathrm{RN}(t_1 + \mathrm{RN}(a_h \cdot b_\ell))$
$t_3 \leftarrow \mathrm{RN}(t_2 + \mathrm{RN}(a_\ell \cdot b_h))$
$\pi_\ell \leftarrow \mathrm{RN}(t_3 + \mathrm{RN}(a_\ell \cdot b_\ell))$
return (π_h, π_ℓ)

1.2 Workplan

Assume we wish to compute

$$d = \mathrm{RN}(\hat{d}), \text{ with } \hat{d} = ab + c,$$

using only rounded-to-nearest floating-point additions and multiplications, and (if needed) comparisons. Algorithm 4 (DekkerProd) makes it possible to express the product ab as a double word (π_h, π_ℓ) such that $\pi_h + \pi_\ell = ab$. We are therefore reduced to computing the sum of a double-word and an FP number.

We will start from the algorithm implemented in Hida, Li and Bailey's QD library [9], that returns a double-word number very close to the sum of a double-word number and a floating-point number. It is Algorithm 5 below, analyzed in [10]. It will not suffice for our purpose since it does not return a correctly-rounded result, so modifications will be necessary.

Algorithm 5 – **DWPlusFP**(x_h, x_ℓ, y). Computes $(x_h, x_\ell) + y$ in binary, precision-p, floating-point arithmetic. Implemented in the QD library. The number $x = x_h + x_\ell$ is a double-word number (i.e., it satisfies $x_h = \mathrm{RN}(x_h + x_\ell)$).

1: $(s_h, s_\ell) \leftarrow \mathrm{2Sum}(x_h, y)$
2: $v \leftarrow \mathrm{RN}(x_\ell + s_\ell)$
3: $(z_h, z_\ell) \leftarrow \mathrm{Fast2Sum}(s_h, v)$
4: **return** (z_h, z_ℓ)

The following result is proven in [10].

Theorem 1.2. *The pair* (z_h, z_ℓ) *returned by Algorithm 5 is a DW number. it safisfies:*

$$|(z_h + z_\ell) - (x + y)| \leq 2u^2 \cdot |x + y| . \quad (2)$$

In Section 2.1 we analyze the various cases that may occur when trying to compute $\mathrm{RN}(\pi_h + \pi_\ell + c)$. We will find that the calculation will be simple, unless some intermediate variable (variable w in Algorithm 7) is a power of 2. We explain how that case can be detected in Section 2.2, and how it can be dealt with in Section 2.3.

2 Building the algorithm

2.1 Reduction to the computation of the sum of 3 FP numbers

As stated in the previous section, we aim at computing $d = \mathrm{RN}(\hat{d})$, with $\hat{d} = ab + c$, by first expressing the product ab as a double word (π_h, π_ℓ) (this is done by using the Dekker-Veltkamp product, i.e., Algorithm 4). Hence we are reduced to computing

$$\mathrm{RN}(\pi_h + \pi_\ell + c).$$

Lauter [13] shows that for the IEEE754 binary formats, this can be done using 128-bit integer operations. Here, we are going to use floating-point operations only (with the advantage of being able to handle any possible binary FP format, and the inconvenient of not handling underflows, overflows and the various IEEE flags). Interestingly enough, Algorithm 5 *almost always* computes d: the correct result will often be the most significant term of the pair returned by the call to $\mathrm{DWPlusFP}(\pi_h, \pi_\ell, c)$. More precisely, let us modify that algorithm and compute

$$\begin{cases} (s_h, s_\ell) & = & \mathrm{2Sum}(\pi_h, c) \\ (v_h, v_\ell) & = & \mathrm{2Sum}(\pi_\ell, s_\ell) \\ (z_h, z_\ell) & = & \mathrm{Fast2Sum}(s_h, v_h) \end{cases}$$

(one easily sees that v_h is the variable "v" of Algorithm 5). We obviously have

$$z_h + z_\ell + v_\ell = ab + c = \hat{d}, \tag{3}$$

and Theorem 1.2 tells us that

- (z_h, z_ℓ) is a double-word, i.e., $z_h = \mathrm{RN}(z_h + z_\ell)$,
- $|v_\ell| = |(z_h + z_\ell) - \hat{d}| \leq 2u^2|\hat{d}|$.

Note that when $ab + c = 0$ this implies $z_h = z_\ell = v_\ell = 0$, so we will not need to consider that case in the following. From

$$|\hat{d}| \leq \frac{|z_h + z_\ell|}{1 - 2u^2} \leq \frac{|z_h|(1+u)}{1 - 2u^2},$$

we obtain

$$|v_\ell| \leq 2u^2|\hat{d}| \leq \frac{2u^2(1+u)}{1-2u^2}\,|z_h|\,. \tag{4}$$

Two cases may occur,

- If $|z_h|$ is not a power of 2 then $|z_\ell| \leq \frac{1}{2}\mathrm{ulp}(z_h)$ and, as $\mathrm{ulp}(z_h) > u|z_h|$, (4) implies

  $$|v_\ell| \leq \frac{2u(1+u)}{1-2u^2}\mathrm{ulp}(z_h),$$

  which is strictly less than $\frac{1}{2}\mathrm{ulp}(z_h)$ as soon as $u \leq 1/8$, so that

  $$|z_\ell + v_\ell| < \mathrm{ulp}(z_h).$$

- If $|z_h|$ is a power of 2 then

  $$-\frac{1}{4}\mathrm{ulp}(z_h) \leq z_\ell \times \mathrm{sign}(z_h) \leq \frac{1}{2}\mathrm{ulp}(z_h),$$

  and, as $\mathrm{ulp}(z_h) = 2u|z_h|$, (4) implies

  $$|v_\ell| \leq \frac{u(1+u)}{1-2u^2}\mathrm{ulp}(z_h),$$

  which is strictly less than $\frac{1}{4}\mathrm{ulp}(z_h)$ as soon as $u \leq 1/8$, so that

  $$-\frac{1}{2}\mathrm{ulp}(z_h) < (z_\ell + v_\ell) \times \mathrm{sign}(z_h) < \frac{3}{4}\mathrm{ulp}(z_h),$$

The consequence of this is that, as soon as $u \leq 1/8$, $\hat{d} = z_h + z_\ell + v_\ell$ satisfies $z_h^- < \hat{d} < z_h^+$, where z_h^- and z_h^+ are the floating-point predecessor and successor of z_h, respectively.

In the (by far most frequent) case where $|\mathrm{RN}(v_\ell + z_\ell)|$ is *not* a power of 2, the number $|v_\ell + z_\ell|$ is not a power of 2 either (otherwise it would round to itself), and in that case $|\mathrm{RN}(v_\ell + z_\ell)|$ is larger than $\frac{1}{2}\mathrm{ulp}(z_h)$ (resp. $\frac{1}{4}\mathrm{ulp}(z_h)$) iff $|v_\ell + z_\ell|$ is larger than $\frac{1}{2}\mathrm{ulp}(z_h)$ (resp. $\frac{1}{4}\mathrm{ulp}(z_h)$).

Therefore,

if $u \leq 1/8$ then when $|\mathrm{RN}(v_\ell + z_\ell)|$ is not a power of 2, d is equal to $\mathrm{RN}(z_h + \mathrm{RN}(z_\ell + v_\ell))$.

We will examine later on what must be done when $|\mathrm{RN}(v_\ell + z_\ell)|$ is a power of 2, but in the meanwhile, we have to find a simple way of determining if the absolute value of a FP number is a power of 2.

2.2 Determining if the absolute value of a FP number is a power of 2

We have,

Theorem 2.1. *In binary, precision-p, floating-point arithmetic, assuming no underflow/overflow occurs, the absolute value of the nonzero FP number x is a power of 2 if and only if*

$$\mathrm{RN}\left[\mathrm{RN}\left(\left(2^{p-1}+1\right)\cdot x\right) - 2^{p-1}x\right] = x. \tag{5}$$

Proof. If $|x|$ is a power of 2, then multiplying by x is an exact operation and therefore (5) boils down to $\mathrm{RN}(x) = x$, which obviously holds since x is a FP number. If $|x|$ is not a power of 2 then there exist integers N and e such that N is odd, $N > 1$, and $|x| = N \cdot 2^e$. Let $P = 2^{p-1} + 1$. The number $P \cdot N$ is an odd integer of absolute value strictly larger than 2^p. Therefore $P \cdot x = P \cdot N \cdot 2^e$ is not exactly representable in FP arithmetic. Hence $\mathrm{RN}(P \cdot x) \neq P \cdot x$.

From

$$x(2^{p-1}+1)(1-u) \leq \mathrm{RN}(P \cdot x) \leq x(2^{p-1}+1)(1+u),$$

we deduce (remember: $u = 2^{-p}$) that

$$\frac{\frac{1}{2u}+1}{\frac{1}{2u}}(1-u) \leq \frac{\mathrm{RN}(P \cdot x)}{2^{p-1}x} \leq \frac{\frac{1}{2u}+1}{\frac{1}{2u}}(1+u),$$

so that (as soon as $u \leq 1/4$)

$$1 \leq 1 + u - 2u^2 \leq \frac{\mathrm{RN}(P \cdot x)}{2^{p-1}x} \leq 1 + 3u + 2u^2 < 2.$$

Therefore, we can apply Sterbenz Theorem (Theorem 1.1) to the subtraction $\mathrm{RN}(P \cdot x) - 2^{p-1}x$, and deduce that that subtraction is exact. We therefore obtain that the left-hand part of (5) is exactly equal to $\mathrm{RN}(P \cdot x) - 2^{p-1}x$, which differs from $P \cdot x - 2^{p-1}x = x$. □

This gives the following algorithm

Algorithm 6 IsPowerOf2(x).

Require: $P = 2^{p-1} + 1$
Require: $Q = 2^{p-1}$
 $L \leftarrow \mathrm{RN}(P \cdot x)$
 $R \leftarrow \mathrm{RN}(Q \cdot x)$
 $\Delta \leftarrow \mathrm{RN}(L - R)$
 return $(\Delta = x)$

2.3 The difficult case: when $|\mathrm{RN}(v_\ell + z_\ell)|$ is a power of 2

Define $w = \mathrm{RN}(z_\ell + v_\ell)$. We are in the case $|w| = 2^k$ for some $k \in \mathbb{Z}$. We need to determine if $\mathrm{RN}(z_h + w)$ differs from $\mathrm{RN}(z_h + z_\ell + v_\ell)$. An easy case is when $|w|$ is less than the "critical" power of 2, defined as

- $\frac{1}{2}\mathrm{ulp}(z_h)$ if $|z_h|$ is not a power of 2; or if $|z_h|$ is a power of 2 and z_h and w have the same sign;
- $\frac{1}{4}\mathrm{ulp}(z_h)$ if $|z_h|$ is a power of 2 and z_h and w have opposite signs.

This is easily determined: Let $w' = \mathrm{RN}\left(\frac{3}{2}w\right) = \frac{3}{2}w$, the number $|w|$ is (strictly) less than the critical power of 2 if and only if $\mathrm{RN}(z_h + w') = z_h$. In such a case, we are done: the result to be returned is z_h.

Now, when $|w|$ is equal to the "critical" power of 2, we need to determine if $|z_\ell + v_\ell|$ is equal to, above, or below that power of two. This can be done using the Fast2Sum algorithm (as Property 2.2 below shows that $|z_\ell| \geq |v_\ell|$ as soon as $u \leq 1/16$). More precisely, if we compute

$$\begin{cases} \delta & = & \mathrm{RN}(w - z_\ell) \\ t & = & \mathrm{RN}(v_\ell - \delta), \end{cases}$$

then $w + t = z_\ell + v_\ell$. The choice is now simple:

- if $t = 0$ then $w = z_\ell + v_\ell$, so that $d = \mathrm{RN}(z_h + w)$;
- if $t \neq 0$ and w have opposite signs, then $z_\ell + v_\ell$ is below the critical power of 2, so that $d = z_h$;
- if $t \neq 0$ and w have the same sign, then d is the FP predecessor or successor of z_h (depending on the sign of w), which can be obtained as $d = \mathrm{RN}(z_h + w')$, using $w' = \mathrm{RN}\left(\frac{3}{2}w\right) = \frac{3}{2}w$, as previously.

The following property shows that we can use the Fast2Sum algorithm for adding z_ℓ and v_ℓ.

Property 2.2. *When w is the critical power of 2, we have $|v_\ell| \leq |z_\ell|$ as soon as $u \leq 1/16$.*

Proof.

- If $|z_h|$ is not a power of 2, or if $|z_h|$ is a power of 2 and w has the same signe as z_h, then w being critical means that $|w| = \frac{1}{2}\text{ulp}(z_h)$, and therefore, we have
$$|v_\ell + z_\ell| \geq \frac{1}{2}\left(1 - \frac{u}{2}\right)\text{ulp}(z_h).$$
As in that case $|v_\ell|$ is less than
$$\frac{2u(1+u)}{1-2u^2}\text{ulp}(z_h),$$
we have
$$|z_\ell| \geq \left(\frac{1}{2} - \frac{u}{4} - \frac{2u(1+u)}{1-2u^2}\right)\text{ulp}(z_h),$$
so that $|z_\ell| \geq |v_\ell|$ as soon as $u \leq 1/16$;

- if $|z_h|$ is a power of 2 and the signs of w and z_h differ, then $|w| = \frac{1}{4}\text{ulp}(z_h)$, and therefore, we have
$$|v_\ell + z_\ell| \geq \frac{1}{4}\left(1 - \frac{u}{2}\right)\text{ulp}(z_h).$$
As in that case $|v_\ell|$ is less than
$$\frac{u(1+u)}{1-2u^2}\text{ulp}(z_h),$$
we have
$$|z_\ell| \geq \left(\frac{1}{4} - \frac{u}{8} - \frac{u(1+u)}{1-2u^2}\right)\text{ulp}(z_h),$$
so that $|z_\ell| \geq |v_\ell|$ as soon as $u \leq 1/16$.

□

3 Putting all this together

Algorithm 7 below derives from the analysis given in Section 2.

Algorithm 7 EmulFMA(a, b, c).

Require: $P = 2^{p-1} + 1$
Require: $Q = 2^{p-1}$
  $(\pi_h, \pi_\ell) \leftarrow \mathrm{DekkerProd}(a, b)$
  $(s_h, s_\ell) \leftarrow \mathrm{2Sum}(\pi_h, c)$
  $(v_h, v_\ell) \leftarrow \mathrm{2Sum}(\pi_\ell, s_\ell)$
  $(z_h, z_\ell) \leftarrow \mathrm{Fast2Sum}(s_h, v_h)$
  $w \leftarrow \mathrm{RN}(v_\ell + z_\ell)$
  $L \leftarrow \mathrm{RN}(P \cdot w)$
  $R \leftarrow \mathrm{RN}(Q \cdot w)$
  $\Delta \leftarrow \mathrm{RN}(L - R)$
  $d^1_{\mathrm{temp}} \leftarrow \mathrm{RN}(z_h + w)$
  if $\Delta \neq w$ **then**
    return d^1_{temp}
  else
    $w' \leftarrow \mathrm{RN}\left(\frac{3}{2} \cdot w\right)$
    $d^2_{\mathrm{temp}} \leftarrow \mathrm{RN}(z_h + w')$
    if $d^2_{\mathrm{temp}} = z_h$ **then**
      return z_h
    else
      $\delta \leftarrow \mathrm{RN}(w - z_\ell)$
      $t \leftarrow \mathrm{RN}(v_\ell - \delta)$
      if $t = 0$ **then**
        return d^1_{temp}
      else
        $g \leftarrow \mathrm{RN}(t \cdot w)$
        if $g < 0$ **then**
          return z_h
        else
          return d^2_{temp}
        end if
      end if
    end if
  end if

We have,

Theorem 3.1. *In a binary, precision-p, floating-point arithmetic with an unbounded exponent range, if $p \geq 4$, then Algorithm 7 returns* $\mathrm{RN}(ab + c)$ *for all floating-point numbers a, b, and c.*

Proof. The theorem immediately follows from the analysis of Section 2 and the fact that $p \geq 4$ implies $u \leq 1/16$. □

The primary disadvantage of our algorithm is the presence of tests. In the event that the branch prediction mechanism of the processor fails, these tests may result in a significant reduction in performance. However, it is important to note that the value of $|\mathrm{RN}(v_\ell + z_\ell)|$ is very unlikely to be a power of 2. Consequently, when a large number of FMAs are computed, the branch prediction should function effectively, whereas when a small number of FMAs are computed, the performance loss is of minimal consequence. Secondly, and more importantly, we hypothesize that tests cannot be entirely avoided. In fact, we make the following conjecture.

Conjecture 3.2. *An algorithm that only uses rounded-to-nearest additions, subtractions and multiplications, without tests, cannot evaluate* $\mathrm{RN}(ab + c)$ *for all possible FP numbers* a*,* b*, and* c*.*

The rationale behind Conjecture 3.2 is as follows:

- The authors of [12] have shown that an algorithm that only uses rounded-to-nearest additions and subtractions cannot evaluate $\mathrm{RN}(x + y + z)$ for all possible FP numbers x, y, and z (Theorem 8 in [12]);
- When computing $\mathrm{RN}(ab + c)$, if we first convert the product ab into a "double word" (π_h, π_ℓ), as described in Section 2.1, we are reduced to computing $\mathrm{RN}(\pi_h + \pi_\ell + c)$. It is not possible to apply Theorem 8 in [12] because the sum $\pi_h + \pi_\ell + c$ is not an "arbitrary" sum: as $\pi_h + \pi_\ell = ab$, the FP numbers π_h and π_ℓ cannot have an exponent difference larger than $2p$. Nevertheless, one easily verifies that the proof of the theorem remains valid in that specific case;
- Consequently, we conclude that Conjecture 3.2 is valid if we restrict our consideration to algorithms that first convert ab into the sum of two FP numbers and then perform only rounded to nearest additions and subtractions. Although it seems hard to see how it could be any other way, we have no proof of that.

Conclusion

We have presented a novel approach to emulate the fused multiply-add (FMA) instruction using standard, rounded-to-nearest floating-point arithmetic operations. Our method builds on the foundation laid by previous research but eliminates the need for less commonly supported rounding functions such as round-to-odd, thereby increasing the practical applicability and portability of the algorithm across different computing architectures.

Future work could further explore whether Conjecture 3.2 holds.

Acknowledgement

This work was partly supported by the NuSCAP (ANR-20-CE48-0014) project of the French National Agency for Research (ANR).

References

[1] IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, July 2019.

[2] G. Bohlender, W. Walter, P. Kornerup, and D.W. Matula. Semantics for exact floating point operations. In *10th IEEE Symposium on Computer Arithmetic*, pages 22–26, 1991.

[3] Sylvie Boldo and Marc Daumas. Representable correcting terms for possibly underflowing floating point operations. In *16th IEEE Symposium on Computer Arithmetic (ARITH-16)*, pages 79–86, Santiago de Compostela, Spain, 2003.

[4] Sylvie Boldo, Claude-Pierre Jeannerod, Guillaume Melquiond, and Jean-Michel Muller. Floating-point arithmetic. *Acta Numerica*, 32:203–290, 2023.

[5] 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.

[6] John Cocke and V. Markstein. The evolution of RISC technology at IBM. *IBM Journal of Research and Development*, 34(1):4–11, January 1990.

[7] Marius A. Cornea-Hasegan, Roger A. Golliver, and Peter Markstein. Correctness proofs outline for Newton–Raphson based floating-point divide and square root algorithms. In *14th IEEE Symposium on Computer Arithmetic (ARITH-14)*, pages 96–105, April 1999.

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

[9] Y. Hida, X. S. Li, and D. H. Bailey. C++/fortran-90 double-double and quad-double package, release 2.3.17, March 2012. Accessible electronically at `http://crd-legacy.lbl.gov/~dhbailey/mpdist/`.

[10] Mioara Joldeş, Jean-Michel Muller, and Valentina Popescu. Tight and rigourous error bounds for basic building blocks of double-word arithmetic. *ACM Transactions on Mathematical Software*, 44(2), 2017.

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

[12] Peter Kornerup, V. Lefèvre, N. Louvet, and J.-M. Muller. On the computation of correctly-rounded sums. *IEEE Transactions on Computers*, 61(2):289–298, March 2012.

[13] Christoph Lauter. An efficient software implementation of correctly rounded operations extending FMA: $a + b + c$ and $a \times b + c \times d$. In *ACSSC Proc.*, 2017.

[14] P. Markstein. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1):111–119, January 1990.

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

[16] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic, 2nd edition.* Birkhäuser Boston, 2018. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-3-319-76525-9.

[17] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

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