Emulation of a FMA and correctly-rounded sums: proved algorithms using rounding to odd

Sylvie Boldo, Guillaume Melquiond

Emulation of a FMA and correctly-rounded sums: proved algorithms using rounding to odd

Sylvie Boldo and Guillaume Melquiond

Abstract

Rounding to odd is a non-standard rounding on floating-point numbers. By using it for some intermediate values instead of rounding to nearest, correctly rounded results can be obtained at the end of computations. We present an algorithm to emulate the fused multiply-and-add operator. We also present an iterative algorithm for computing the correctly rounded sum of a set floating-point numbers under mild assumptions. A variation on both previous algorithms is the correctly rounded sum of any three floating-point numbers. This leads to efficient implementations, even when this rounding is not available. In order to guarantee the correctness of these properties and algorithms, we formally proved them using the Coq proof checker.

Index Terms

Floating-point, rounding to odd, accurate summation, FMA, formal proof, Coq.

I. INTRODUCTION

Floating-point computations and their roundings are described by the IEEE-754 standard [17], [18] followed by every modern general-purpose processors. This standard was written to ensure the coherence of the result of a computation whatever the environment. This is the "correct rounding" principle: the result of an operation is the same as if it was first computed with an infinite precision and then rounded to the precision of the destination format. There may exist higher precision formats though, and it would not be unreasonable for a processor to store all kinds of floating-point result in a single kind of register instead of having as many register sets as it supports floating-point formats. In order to ensure IEEE-754 conformance, care must then be taken that a result is not first rounded to the extended precision of the registers and then rounded to the precision of the destination format.

This "double rounding" phenomenon may happen on processors built around the Intel x86 instruction set for example. Indeed, their floating-point units use 80-bit long registers to store the results of their computations, while the most common format used to store in memory is only 64-bit long (IEEE double precision). To prevent double rounding, a control register allows to set the floating-point precision, so that the results are not first rounded to the register precision. Unfortunately, setting the target precision is a costly operation as it requires the processor pipeline to be flushed. Moreover, thanks to the extended precision, programs generally seem to produce

more accurate results. As a consequence, compilers usually do not generate the additional code that would ensure that each computation is correctly rounded in its own precision.

Double rounding can however lead to unexpected inaccuracy and as such it is considered a dangerous feature. So writing robust fbating-point algorithms requires extra care in order to ensure that this potential double rounding will not produce incorrect results [15]. Nevertheless, double rounding is not necessarily a threat. For example, if the extended precision is at least twice as big, then it can be used to emulate correctly rounded basic operations for a smaller precision [9]. Double rounding can also be made innocuous by introducing a new rounding mode and using it for the first rounding. When a real number is not representable, it will be rounded to the adjacent fbating-point number with an odd mantissa. In this article, this rounding will be named rounding to odd.

Von Neumann was considering this rounding when designing the arithmetic unit of the EDVAC [19]. Goldberg later used it when converting binary fbating-point numbers to decimal representations [10]. The properties of this rounding operator are also close to the ones that appear when implementing rounded fbating-point operators with guard bits [8]. This rounding was never more than an implementation detail though. Our work aims at showing that it is worth making it a rounding mode in its own rights and using it in higher-level algorithms to produce more accurate results.

Section II will detail a few characteristics of double rounding and why rounding to nearest is failing us. Section III will introduce the formal definition of rounding to odd, how it solves the double rounding issue, and how to implement this rounding. Its property with respect to double rounding will then be extended to two applications. Section IV will describe an algorithm that emulates the fbating-point fused-multiply-and-add operator. Section V will then present algorithms for performing accurate summation.

II. Double rounding

A. Floating-point definitions

Our formal proofs are based on the fbating-point formalization [4] of Daumas, Rideau, and Théry in Coq [1], and on the corresponding library by Théry and one of the authors [2]. Floating-point numbers are represented by pairs (n, e) that stand for $n \times 2^e$. We use both an integral signed mantissa n and an integral signed exponent e for sake of simplicity.

A fbating-point format is denoted by $\mathbb{B}$ and is a pair composed by the lowest exponent $-E$ available and the precision p. We do not set an upper bound on the exponent as overfbws do not matter here (see below). We define a representable pair (n, e) such that $|n| < 2^p$ and $e \geq -E$. We denote by $\mathbb{F}$ the subset of real numbers represented by these pairs for a given format $\mathbb{B}$. Now only the representable fbating-point numbers will be referred to; they will simply be denoted as fbating-point numbers.

All the IEEE-754 rounding modes are also defined in the Coq library, especially the default rounding: the rounding to nearest even, denoted by $\circ$. We have $f = \circ(x)$ if f is the fbating-point number closest to x; when x is half way between two consecutive fbating-point numbers, the one with an even mantissa is chosen.

A rounding mode is defined in the Coq library as a relation between a real number and a fbating-point number, and not a function from real values to fbats. Indeed, there may be several fbats corresponding to the same real value. For a relation, a weaker property than being a rounding mode is being a faithful rounding. A fbating-point number f is a faithful rounding of a real x if it is either the rounded up or rounded down of x, as shown on Figure 1. When x is a fbating-point number, it is its own and only faithful rounding. Otherwise there always are two faithful rounded values bracketing the real value when no overfbw occurs.

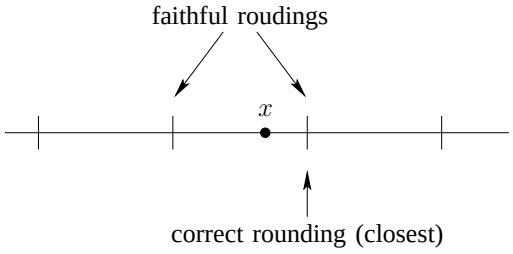


Fig. 1

FAITHFUL ROUNDINGS.

B. Double rounding accuracy

As explained before, a fbating-point computation may first be done in an extended precision, and later rounded to the working precision. The extended precision is denoted by $\mathbb{B}_e = (p+k, E_e)$

and the working precision is denoted by $\mathbb{B}_w = (p, E_w)$. If the same rounding mode is used for both computations (usually to nearest even), it can lead to a less precise result than the result after a single rounding.

For example, see Figure 2. When the real value x is in the neighborhood of the midpoint of two consecutive fbating-point numbers g and h, it may first be rounded in one direction toward this middle t in extended precision, and then rounded in the same direction toward f in working precision. Although the result f is close to x, it is not the closest fbating-point number to x, as h is. When both rounding directions are to nearest, we formally proved that the distance between the given result f and the real value x may be as much as

$$|f - x| \leq \left(\frac{1}{2} + 2^{-k-1}\right) \mathrm{ulp}(f).$$

When there is only one rounding, the corresponding inequality is $|f - x| \leq \frac{1}{2}\mathrm{ulp}(f)$. This is the expected result for a IEEE-754 compatible implementation.

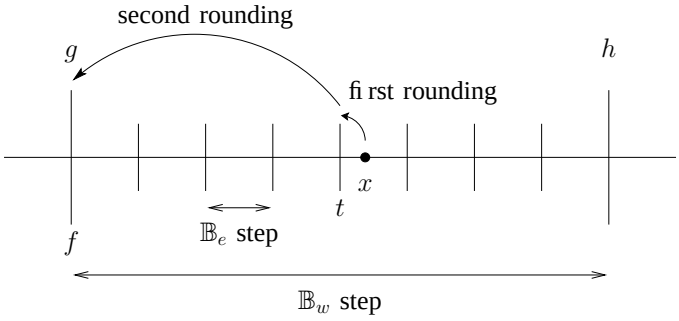


Fig. 2

BAD CASE FOR DOUBLE ROUNDING.

Section IV-C.1 will show that, when there is only one single fbating-point format but many computations, trying to get a correctly rounded result is somehow similar to avoiding incorrect double rounding.

C. Double rounding and faithfulness

Another interesting property of double rounding as defined previously is that it is a faithful rounding. We even have a more generic result.

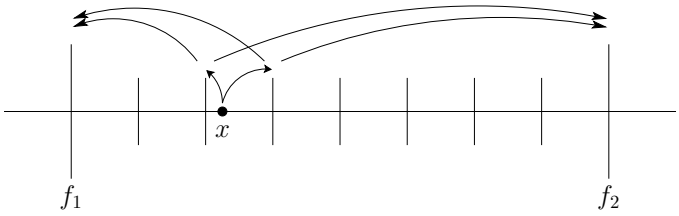


Fig. 3

DOUBLE ROUNDINGS ARE FAITHFUL.

Let us consider that the relations are not required to be rounding modes but only faithful roundings. We formally certified that the rounded result f of a double faithful rounding is faithful to the real initial value x, as shown in Figure 3. The requirements are $k \geq 0$ and $k \leq E_e - E_w$ (any normal fbat in the working format is normal in the extended format).

Let R_e be a faithful rounding in extended precision $\mathbb{B}_e = (p + k, E_e)$ and let R_w be a faithful rounding in the working precision $\mathbb{B}_w = (p, E_w)$. If $k \geq 0$ and $k \leq E_e - E_w$, then for all real value x, the fbating-point number $R_w(R_e(x))$ is a faithful rounding of x in the working precision.

Theorem 1: DblRndStable

This is a very powerful result as faithfulness is the best result we can expect as soon as we at least consider two roundings to nearest. And this result can be applied to any two successive IEEE-754 rounding modes (to zero, toward $+\infty$...).

This means that any sequence of successive roundings in decreasing precisions gives a faithful rounding of the initial value.

III. ROUNDING TO ODD

As seen in the previous section, rounding two times to nearest induces a bigger round-off error than one single rounding to nearest and may then lead to unexpected incorrect results. By rounding to odd first, the second rounding will correctly round to nearest the initial value.

A. Formal description

This rounding does not belong to the IEEE-754's or even 754R[1]'s rounding modes. It should not be mixed up with the rounding to the nearest odd (similar to the default rounding: rounding to the nearest even).

We denote by $\triangle$ the rounding toward $+\infty$ and by $\triangledown$ the rounding toward $-\infty$. The rounding to odd is defined by:

$$\square_{\mathrm{odd}}(x) = \begin{cases} x & \text{if } x \in \mathbb{F}, \\ \triangle(x) & \text{if the mantissa of } \triangle(x) \text{ is odd,} \\ \triangledown(x) & \text{otherwise.} \end{cases}$$

Note that the result of x rounded to odd can be even only when x is a representable fbating-point number. Note also that when x is not representable, $\square_{\mathrm{odd}}(x)$ is not necessarily the nearest fbating-point number with an odd mantissa. Indeed, this is wrong when x is close to a power of two. This partly explains why the formal proof of Algorithm 1 needs to separate powers of two from other fbating-point numbers.

The first proofs we formally checked were basic properties of this operator.

This operator is a rounding mode as defined in our formalization [4]:

- Each real can be rounded to odd.
- Any odd rounding is a faithful rounding.
- Odd rounding is monotone.

We also certified that:

- Odd rounding is unique (meaning that it can be expressed as a function).
- Odd rounding is symmetric, meaning that if $f = \square_{\mathrm{odd}}(x)$, then $-f = \square_{\mathrm{odd}}(-x)$.

Theorem 2: To_Odd {Total, MinOrMax, Monotone, RoundedModeP, UniqueP, and SymmetricP}

B. Implementing the rounding to odd

Rounding to odd the real result x of a fbating-point computation can be done in two steps. First round it to zero into the fbating-point number $\mathcal{Z}(x)$ with respect to the IEEE-754 standard.

[1]See `http://www.validlab.com/754R/`.

And then perform a logical or between the inexact flag ι (or the sticky bit) of the first step and the last bit of the mantissa.

If the mantissa of $\mathcal{Z}(x)$ is already odd, this floating-point number is the rounding to odd of x too; the logical or does not change it. If the floating-point computation is exact, $\mathcal{Z}(x)$ is equal to x and ι is not set; consequently $\square_{\mathrm{odd}}(x) = \mathcal{Z}(x)$ is correct. Otherwise the computation is inexact and the mantissa of $\mathcal{Z}(x)$ is even, but the final mantissa must be odd, hence the logical or with ι. In this last case, this odd float is the correct one, since the first rounding was toward zero.

Computing ι is not a problem *per se*, since the IEEE-754 standard requires this flag to be implemented, and hardware already uses sticky bits for the other rounding modes. Furthermore, the value of ι can directly be reused to flag the odd rounding of x as exact or inexact. As a consequence, on an already IEEE-754 compliant architecture, adding this new rounding has no significant cost.

Another way to round to odd with precision $p + k$ is the following. We first round x toward zero with $p + k - 1$ bits. We then concatenate the inexact bit of the previous operation at the end of the mantissa in order to get a $p + k$-bit float. The justification is similar to the previous one.

Both previous methods are aimed at hardware implementation. They may not be efficient enough to be used in software. Paragraph V-B will present a third way of rounding to odd, more adapted to current architectures and actually implemented. It is portable and available in higher level languages as it does not require changing the rounding direction and accessing the inexact flag.

C. Correct double rounding

Algorithm 1 first computes the rounding to odd of the real value x in the extended format (with $p + k$ bits). It then computes the rounding to the nearest even of the previous value in the working format (with p bits). We here consider a real value x but an implementation does not need to really handle x: it can represent the abstract exact result of an operation between floating-point numbers.

Although there is a double rounding, we here guarantee that the computed final result is correct. The explanation is in Figure 4 and is as follow.

Algorithm 1 Correct double rounding algorithm.

$$t = \square_{\text{odd}}^{p+k}(x)$$
$$f = \circ^{p}(t)$$

Assuming $p \geq 2$, $k \geq 2$, and $E_e \geq 2 + E_w$, Algorithm 1 has the following property:

$$f = \circ^{p}(x).$$

Theorem 3: To_Odd_Even_is_Even

When x is exactly equal to the middle of two consecutive fbating-point numbers f_1 and f_2 (case 1), then t is exactly x and f is the correct rounding of x. Otherwise, when x is slightly different from this midpoint (case 2), then t is different from this midpoint: it is the odd value just greater or just smaller than the midpoint depending on the value of x. The reason is that, as $k \geq 2$, the midpoint is even in the $p + k$ precision, so t cannot be rounded into it if it is not exactly equal to it. This obtained t value will then be correctly rounded to f, which is the closest p-bit fbat from x. The other cases (case 3) are far away from the midpoint and are easy to handle.

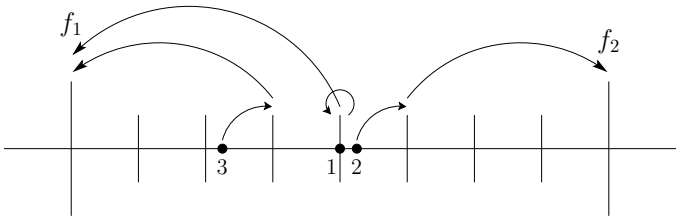


Fig. 4

DIFFERENT CASES OF ALGORITHM 1.

Note that the hypothesis $E_e \geq 2 + E_w$ is not a requirement hard to satisfy. Indeed it does not mean that the exponent range (as defined in the IEEE-754 standard) of the extended format

must be greater by 2. Due to our definition of E, a sufficient condition is simply: Any normal fbating-point numbers with respect to $\mathbb{B}_w$ should be normal with respect to $\mathbb{B}_e$.

The pen and paper proof is a bit technical but does seem easy (see Figure 4). But it does not consider the special cases: especially the ones where $\circ^p(x)$ is a power of two, and subsequently where $\circ^p(x)$ is the smallest normal fbat. And we must look into all these special cases in order to be sure that the algorithm can always be applied, even when underfbw occurs. We formally proved this result using the Coq proof assistant in order to be sure not to forget any case or hypothesis and not to make mistakes in the numerous computations. There are many splittings into subcases that made the final proof rather long: 7 theorems and about one thousand lines of Coq, but we are now sure that every cases (normal/subnormal, power of the radix or not) are supported. Details on this proof were presented in a previous work [3].

IV. Emulating the FMA

The fused-multiply-and-add is a recent fbating-point operator that is already present on modern processors like PowerPC or Itanium. This operation will be standardized thanks to the revision of the IEEE-754 standard. Given three fbating-point numbers a, b, and c, it computes the value $z = \circ(a \cdot b + c)$ with one single rounding at the end of the computation. There is no rounding after the product $a \cdot b$. This operator is very useful as it increases the accuracy of the dot product and matrix multiplication, but it is available on few processors.

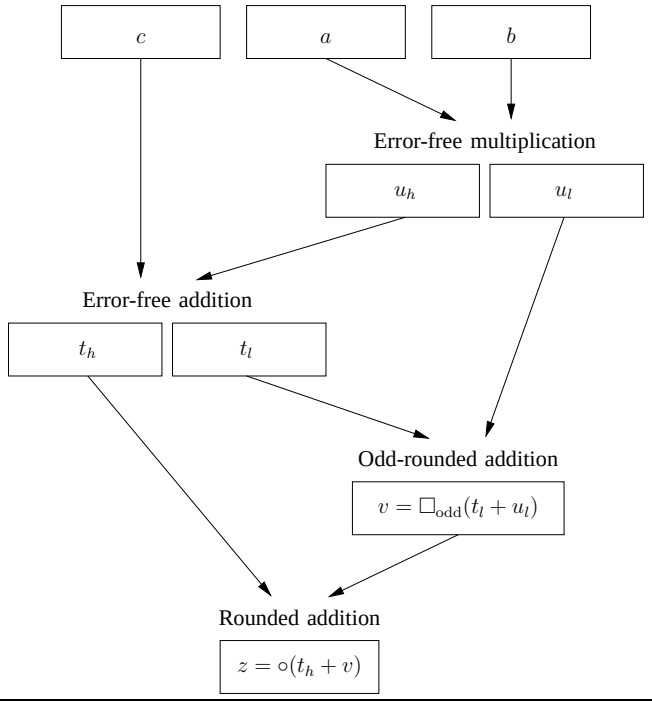
This section will show how an FMA can be emulated thanks to rounding to odd.

A. The algorithm

Algorithm 2 relies on error-free transformations (ExactAdd and ExactMult) to perform some of the operations exactly. These transformations described below return two fbating-point values. One is the usual result: the exact sum or product rounded to nearest. The other is the error term. For addition and multiplication, this term happens to be exactly representable by a fbating-point number and computable using only fbating-point operations provided neither underfbw (for the multiplication) nor overfbw occurs. As a consequence, these equalities hold: $a \cdot b = u_h + u_l$ and $c + u_h = t_h + t_l$. And the rounded result is stored in the higher word: $u_h = \circ(a \cdot b)$ and $t_h = \circ(c + u_h)$.

Algorithm 2 Emulating the FMA.

$$
\begin{aligned}
(u_h, u_l) &= \text{ExactMult}(a, b) \\
(t_h, t_l) &= \text{ExactAdd}(c, u_h) \\
v &= \square_{\text{odd}}(t_l + u_l) \\
z &= \circ(t_h + v)
\end{aligned}
$$



A fast method for computing the error term of the multiplication is the use of the FMA: $u_l = \circ(a \cdot b + (-u_h))$. Unfortunately, we are trying to emulate a FMA here, so we will have to use another method. In IEEE-754 double precision, Dekker's algorithm first splits the 53-bit fbating-point inputs into 26-bit parts thanks to the sign bit. These parts can then be multiplied exactly and subtracted in order to get the error term [6]. For the error term of the addition, since we do not know the relative order of $|c|$ and $|u_h|$, we use Knuth's unconditional version of the algorithm [12].

Our emulated FMA first computes an approximation of the correct result: $t_h = \circ(\circ(a \cdot b) + c)$. It also computes an auxiliary term v that will be added to t_h to get the final result. All the computations are done at the working precision, there is no need for an extended precision. v is computed by adding the neglected terms u_l and t_l, and by rounding the result to odd. If the value was rounded to nearest instead, it could lead to a wrong answer.

As an example, let us consider $a = 1 + 2^{-27}$ and $b = 1 - 2^{-27}$. The exact product is $a \cdot b = 1 - 2^{-54}$. This real number is exactly the midpoint between 1 and its representable predecessor in double precision. If c is small enough, it means that the value $\circ(a \cdot b + c)$ will purely depend

on the sign of c. For example, $|c| \leq 2^{-150}$ is small enough. If c is negative, then the result of the FMA operation will be $1 - 2^{-53}$. Otherwise it will be 1.

Let us get back to our algorithm: the results of the error-free multiplication are $u_h = 1$ and $u_l = -2^{-54}$. Since c is negligible with respect to u_h, we have $t_h = u_h = 1$ and $t_l = c$. So the final result is $\circ(1+t)$. If t was rounded to nearest, then t would be equal to $u_l = -2^{-54}$, since c is also negligible with respect to u_l. As a consequence, the final result would be 1, irrespective of the sign of c. In particular it would be wrong when c is negative. By rounding v to odd, we get $v = -2^{-54}(1+2^{-52})$ instead, when c is negative. This is enough to get the correctly rounded result: $z = 1 - 2^{-53}$.

B. Theorem of correctness

Under the notations of Algorithm 2, if $p \geq 5$ and u_l is representable, then

$$z = \circ(a \times b + c).$$

Theorem 4: FmaEmul

We have formally proved in Coq that the previous algorithm will indeed emulate a FMA. Two hypotheses were necessary. First, the value u_l has to indeed be the error term of the multiplication $a \cdot b$. This hypothesis is just here to avoid some degenerate underfbw cases where the error term is so small that its exponent falls outside the admitted range. The second hypothesis is $p \geq 5$. The mantissa must contain at least 5 bits. This requirement is reasonable since even the smallest format of the IEEE-754 standard has a 24-bit mantissa (23 explicit bits and the most significant bit is implicit).

Listing 1 shows the theorem as it was proved in Coq. First it assumes that the precision is $p \geq 5$. Then it states that a, b, c, and u_l are representable fbating-point numbers in the format `bo` thanks to the predicate `Fbounded`.

Next comes the description of the algorithm. u_h and t_h are the nearest (`Closest`) fbating-point numbers in format `bo` to the potentially non-representable real numbers $a \cdot b$ and $c + u_h$. Note that the `radix` is 2; we are dealing with binary fbating-point numbers.

Listing 1 Coq theorem certifying the correctness of Algorithm 2

```
Hypotheses pGe: (5 <= p).
Hypothesis Fa : Fbounded bo a.
Hypothesis Fb : Fbounded bo b.
Hypothesis Fc : Fbounded bo c.
Hypothesis Ful: Fbounded bo ul.

Hypothesis uhDef: Closest bo radix (a*b)%R uh.
Hypothesis ulDef: (FtoRradix ul = a*b-uh)%R.
Hypothesis thDef: Closest bo radix (c+uh)%R th.
Hypothesis tlDef: (FtoRradix tl = c+uh-th)%R.
Hypothesis vDef : To_Odd bo radix p (tl+ul)%R v.
Hypothesis zDef : EvenClosest bo radix p (th+v)%R z.

Theorem FmaEmul: EvenClosest bo radix p (a*b+c)%R z.
```

The fbating-point numbers u_l and t_l are defined as the error terms. The algorithms that are used to compute them do not matter. We just have to assume that $u_l = a \cdot b - u_h$ and $t_l = c + u_h - t_h$. Then there are the definitions of $v = \square_{\mathrm{odd}}(t_l + u_l)$ and $z = \circ(t_h + v)$. While the precise rounding mode used to compute u_h and t_h does not matter, the last operation of the algorithm has to be described with more details: when $t_h + v$ is the midpoint between two consecutive fbating-point numbers, it is rounded to the one fbating-point number with an even mantissa.

All these hypotheses define the algorithm. The last line of the script can now state our theorem: z is indeed $\circ(a \cdot b + c)$.

C. Proof

This theorem was once again formally proved with a proof checker. It was especially important since the theorem is quite generic. In particular, it does not contain any hypothesis regarding subnormal numbers. The algorithm is still correct even if some computed values are not normal numbers.

1) Adding a negligible yet odd value: We need an intermediate lemma for simplicity and reusability. Let μ be the smallest positive normal number.

By uniqueness of the rounding to nearest even, it is enough to prove that $\circ(x + y)$ is a correct rounding to nearest of $x + z$ with tie breaking to even. This will be proved using the correctness

Let x and y be fbating-point numbers such that $5|y| \leq |x|$ and $5\mu \leq |x|$.
Assume there is a real value z such that $y = \square_{\text{odd}}(z)$.
Then $\circ(x + y) = \circ(x + z)$.

Theorem 5: AddOddEven

of Algorithm 1. It is now enough to prove that $x + y$ is equal to $\square^{p+k}_{\text{odd}}(x + z)$ for a k that we might choose as we want (as soon as it is greater than 1).

The choice of k is done such that there exists a fbat f equal to $x + y$, normal with respect to an extended format on precision $p+k$ and having for exponent the exponent of y. For that, we set $f = (n_x 2^{e_x - e_y} + n_y, e_y)$. As $|y| \leq |x|$, we know that $e_y \leq e_x$ and this definition is indeed a fbat with the wanted exponent. We then choose k to be the integer such that $2^{p+k-1} \leq |n_f| < 2^{p+k}$. The fact that $k \geq 2$ is guaranteed as $5|y| \leq |x|$. The underfbw threshold for the extended format is defined as needed and this works fine thanks to the $5\mu \leq |x|$ hypothesis. These ponderous details are handled in the machine-checked proof.

We have defined an extended format where $x + y$ is representable. There is left to prove that $x + y = \square^{p+k}_{\text{odd}}(x + z)$. We know that $y = \square_{\text{odd}}(z)$, thus we have 2 cases. First, $y = z$, then $x + y = x + z$ and the result holds. Second, y is odd and is a faithful rounding of z. Then we prove (several possible cases and many computations later), that $x + y$ is odd and is a faithful rounding of $x + z$ with respect to the extended format. That ends the proof.

2) Emulate a FMA: First, we can eliminate the case where v was computed without rounding error. Indeed, it means that $z = \circ(t_h + v) = \circ(t_h + t_l + u_l)$. Since $u_l = a \cdot b - u_h$ and $t_h + t_l = c + u_h$, we have $z = \circ((c + u_h) + (a \cdot b - u_h)) = \circ(a \cdot b + c)$.

Now, if v is rounded, it means that v is not a subnormal number. Indeed, if the result of a fbating-point addition is a subnormal number, then the addition is exact. It also means that neither u_l nor t_l are zero: neither the product $a \cdot b$ nor the sum $c + u_h$ are representable fbating-point numbers.

Since $c + u_h$ is not representable, the inequality $2 \cdot |t_h| \geq |u_h|$ holds. Moreover, since u_l is the error term in $u_h + u_l$, $|u_l| \leq 2^{-p} \cdot |u_h|$ holds. Similarly, $|t_l| \leq 2^{-p} \cdot |t_h|$. As a consequence, both $|u_l|$ and $|t_l|$ are bounded by $2^{1-p} \cdot |t_h|$. So their sum $|u_l + t_l|$ is bounded by $2^{2-p} \cdot |t_h|$. Since v is not a subnormal number, the inequality still holds when rounding $u_l + t_l$ to v. So we have

proved that $|v| \leq 2^{2-p} \cdot |t_h|$ when the computation of v is inexact.

To summarize, either v is equal to $t_l + u_l$, either v is negligible with respect to t_h, meaning that $|v| \leq 2^{2-p} \cdot |t_h|$. Theorem 5 can then be applied with $x = t_h$, $y = v$ and $z = t_l + u_l$. Indeed $x + z = t_h + t_l + u_l = a \cdot b + c$. We have to verify two inequalities in order to apply it though. First, we must prove that $5|y| \leq |x|$, meaning that $5|v| \leq |t_h|$. We have just shown that $|v| \leq 2^{2-p} \cdot |t_h|$. As $p \geq 5$, this inequality easily holds.

Second, we must prove that $5\mu \leq |x|$, meaning that $5\mu \leq |t_h|$. We prove it by reducing it to the absurd. We assume that $|t_h| < 5\mu$. So t_l is subnormal. More, t_h must be normal: if t_h is subnormal, then $t_l = 0$, which is impossible. We then look into u_l. If u_l is subnormal, then $v = \square_{\text{odd}}(u_l + t_l)$ is computed correctly, which is impossible. So u_l is normal. We then prove that both $t_l = 0$ and $t_l \neq 0$. First, $t_l \neq 0$ as $v \neq u_l + t_l$. Second, we will prove that $t_l = 0$ by proving that the addition $c + u_h$ is computed exactly (as $t_h = \circ(c + u_h)$). For that, we will prove that $e_{t_h} < e_{u_h} - 1$ as that implies a cancellation in the computation of $c + u_h$ and therefore the exactness of t_h. There is then left to prove that $2^{e_{t_h}} < 2^{e_{u_h}-1}$. As t_h is normal, $2^{e_{t_h}} \leq |t_h|2^{1-p}$ and then, as $p \geq 5$ and u_l is normal: $2^{e_{t_h}} \leq |t_h|2^{1-p} < 5\mu 2^{1-p} \leq \mu \leq |u_l| \leq 2^{e_{u_h}-1}$. We have a contradiction in all cases, therefore $5\mu \leq |t_h|$ and Theorem 5 can be applied and the result holds.

V. Accurate summation

The last steps of the algorithm for emulating the FMA actually computes the correctly rounded sum of three fbating-point numbers at once. Although there is no particular assumption on two of the numbers (c and u_h), there is a strong hypothesis on the third one: $|u_l| \leq 2^{-p} \cdot |u_h|$. We will generalize this summation scheme to an iterated scheme that will compute the correctly rounded sum of a set of fbating-point numbers under some mild assumptions. We will then describe an adder for three fbating-point numbers that rely on the rounding to odd to produce the correctly rounded result.

A. Iterated summation

We will consider the problem of adding a sequence of fbating-point numbers $(f_i)_{1 \leq i \leq n}$ in order to get the correctly rounded sum $s = \circ\left(\sum_{1 \leq i \leq n} f_i\right)$. This problem is not new: adding

several fbating-point numbers with good accuracy is an important problem of scientific computing [11]. Demmel and Hida presented a simple algorithm that yields almost correct summation results [7]. And recently Oishi, Rump, and Ogita, presented some other algorithms for accurate summation [16]. Our algorithm requires stronger assumptions, but it is simple, very fast, and will return the correctly rounded result thanks to the rounding to odd.

Once again we will rely on the double rounding property: $s = \circ(\sum f_i) = \circ(\square^{p+k}_{\text{odd}}(\sum f_i))$. Two approaches are possible. The first one was described in a previous work [3]. Its algorithm computes the partial sums in an extended precision format with rounding to odd. The correctness of the algorithm relies on the following property:

$$\forall j < n, \quad \square^{p+k}_{\text{odd}}\left(\sum_{i=1}^{j+1} f_i\right) = \square^{p+k}_{\text{odd}}\left(f_{j+1} + \square^{p+k}_{\text{odd}}\left(\sum_{i=1}^{j} f_i\right)\right)$$

While this property holds, it is easy to iteratively compute $\square^{p+k}_{\text{odd}}(\sum f_i)$. Then this number just has to be rounded to the working precision in order to obtain the correctly rounded result s thanks to the double rounding property.

Let us now present a different approach. We will rely on the property described in Section IV-C.1 in order to compute a correctly rounded value. Algorithm 3 is close to the one obtained with the first approach, but it does not need the intermediate values to be computed with an extended precision.

For the algorithm to return the correct value, we want to compute $g_{n-1} = \square_{\text{odd}}(\sum_{1 \le i \le n-1} f_i)$. Listing 2 shows the Coq theorem that is used to prove that the following equality stands at each iteration

$$\forall j < n-1, \quad g_{j+1} = \square_{\text{odd}}\left(f_{j+1} + \square_{\text{odd}}\left(\sum_{i=1}^{j} f_i\right)\right) = \square_{\text{odd}}\left(\sum_{i=1}^{j+1} f_i\right)$$

This theorem states that $\square_{\text{odd}}(x+y) = \square_{\text{odd}}(x+z)$ with x a fbating-point number, z a dyadic real number, and $y = \square_{\text{odd}}(z)$. The first line is an hypothesis on the fbating-point format, but it is true for any common format: the number $\frac{1}{2}$ must be a normal number. There are also two hypotheses on $|x|$. First $|x| \ge 2 \cdot |z|$. And second it should be bigger than twice the smallest positive normal fbating-point number μ. As a consequence, in order to apply this theorem at each iteration, we just need the following hypothesis:

$$\text{for } 2 \le j \le n-2, \qquad |f_{j+1}| \ge 2 \cdot \left|\sum_{i=1}^{j} f_i\right| \quad \text{and} \quad |f_{j+1}| \ge 2 \cdot \mu.$$

Algorithm 3 Iterated summation

Input: the $(f_i)_{1\leq i\leq n}$ are suitably ordered and spaced out.

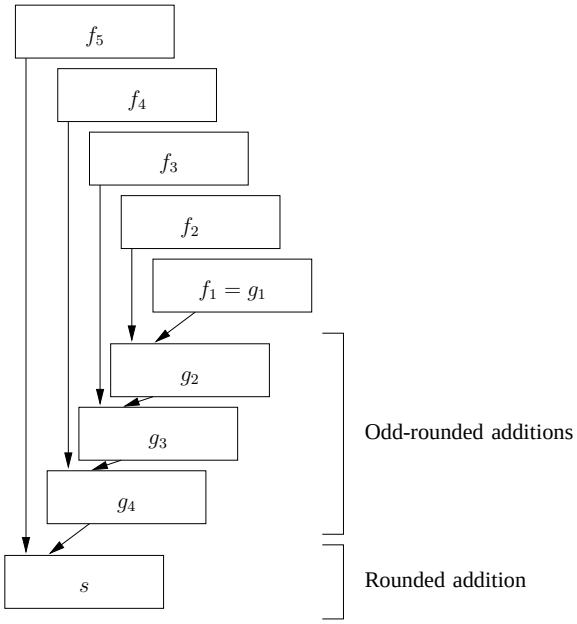
$g_1 = f_1$

For i from 2 to $n-1$,

$g_i = \square_{\text{odd}}(g_{i-1} + f_i)$

$s = \circ(g_{n-1} + f_n)$

Output: $s = \circ(\sum f_i)$.



Now for the final rounding, we apply a slightly modified version of the result of Section IV-C.1. Listing 3 shows its formal specification. In order to apply this theorem on the last operation of Algorithm 3, we need the following hypothesis, just for the biggest value f_n:

$$|f_n| \geq 6 \cdot \left|\sum_{i=1}^{n-1} f_i\right| \quad \text{and} \quad |f_n| \geq 5 \cdot \mu$$

Listing 2 Coq theorem showing how addition preserves rounding to odd

```
Hypothesis dExpBig: p <= dExp bo.
Theorem AddOddOdd2:
  forall (x y f1 f2: float) (z: R),
  Fbounded bo x ->
  (2*(Rabs z) <= Rabs x)%R ->
  (2*(firstNormalPos radix bo p) <= Rabs x)%R ->
  To_Odd bo radix p z y ->
  To_Odd bo radix p (x+y)%R f1 ->
  To_Odd bo radix p (x+z)%R f2 ->
  (FtoRradix f1 = f2)%R.
```

It may generally be a bit difficult to verify that the previous hypotheses hold. So it is interesting

Listing 3 Coq theorem showing how rounding to odd helps computing nearest sum

```
Theorem AddOddEven2:
  forall (x y f1 f2: float) (z: R),
  (3 < p) ->
  (6*(Rabs z) <= Rabs x)%R ->
  (5*(firstNormalPos radix bo p) <= Rabs x)%R ->
  Fcanonic radix bo y -> Fcanonic radix bo x ->
  To_Odd bo radix p z y ->
  EvenClosest bo radix p (x+y)%R f1 ->
  EvenClosest bo radix p (x+z)%R f2 ->
  (FtoRradix f1 = f2)%R.
```

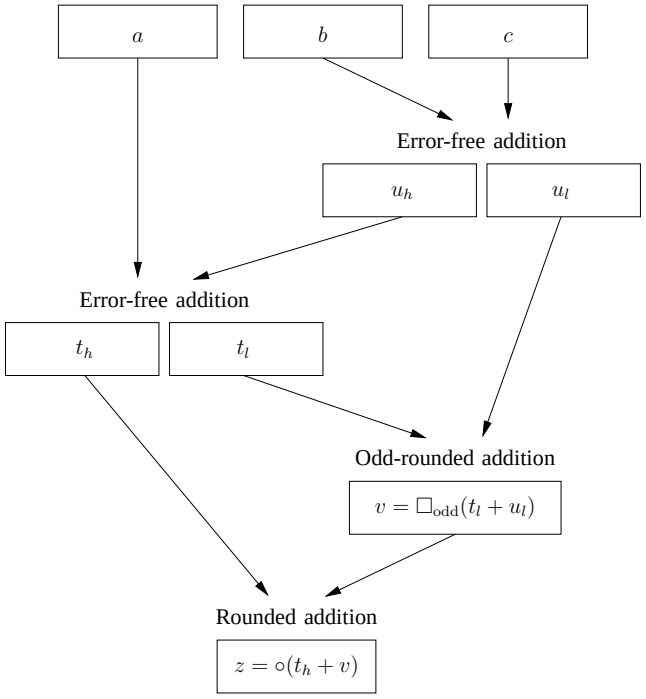
to have a criteria that can be checked with fbating-point numbers only:

$$|f_1| \geq \frac{2}{9} \cdot \mu \quad \text{and} \quad |f_n| \geq 9 \cdot |f_{n-1}| \quad \text{and} \quad \text{for } 1 \leq i \leq n-2,\ |f_{i+1}| \geq 3 \cdot |f_i|$$

B. Adding three numbers

Algorithm 4 Adding three numbers.

$$\begin{aligned} (u_h, u_l) &= \text{ExactAdd}(b, c) \\ (t_h, t_l) &= \text{ExactAdd}(a, u_h) \\ v &= \square_{\text{odd}}(t_l + u_l) \\ z &= \circ(t_h + v) \end{aligned}$$



Let us now consider a simpler situation. We still want to compute a correctly-rounded sum, but there are only three numbers left. In return, we will remove all the requirements on the

relative ordering of the inputs. Algorithm 4 shows how to compute this correctly-rounded sum of three numbers.

Its graph looks similar to the graph of Algorithm 2 for emulating the FMA. The only difference lies in its first error-free transformation. Instead of computing the exact product of two of its inputs, this algorithm computes their exact sum. As a consequence, its proof of correctness can directly be derived from the one for the FMA emulation. Indeed, the correctness of the emulation is not relying on the properties of an exact product. The only property that matters is: $u_h + u_l$ is a normalized representation of a number u. As a consequence, both Algorithm 2 and Algorithm 4 are special cases of a more general algorithm that would compute the correctly rounded sum of a fbating-point number with a real number exactly represented by the sum of two fbating-point numbers.

C. A practical use case

CRlibm[2] is an efficient library for computing correctly rounded results of elementary functions in IEEE-754 double precision. Let us consider the logarithm function [5]. In order to be efficient, the library will first try a fast algorithm. This will usually give the correctly rounded result, but in some situations it may be off by one unit in the last place. When the library detects such a situation, it starts again with a slower yet accurate algorithm in order to get the correct final result.

When computing the logarithm $\circ(\log f)$, the slow algorithm will use triple-double arithmetic [13] in order to first compute an approximation of $\log f$ stored on three double precision numbers $x_h + x_m + x_l$. Thanks to results provided by the table-maker dilemma [14], this approximation is known to be sufficiently accurate for the equality $\circ(\log f) = \circ(x_h + x_m + x_l)$ to hold. This means the library just has to compute the correctly rounded sum of the three fbating-point numbers x_h, x_m, and x_l.

Computing this sum is exactly the point of Algorithm 4. Unfortunately, rounding to odd is not available on any architecture targeted by CRlibm, so it will have to be emulated. Although such an emulation is costly in software, rounding to odd will still allow for a speed up here. Indeed $x_h + x_m + x_l$ is the result of a sequence of triple-double fbating-point operations. As a

[2]See `http://lipforge.ens-lyon.fr/www/crlibm/`.

Listing 4 Correctly rounded sum of three ordered values

```
double CorrectRoundedSum3(double xh, double xm, double xl) {
  double th, tl;
  db_number thdb; // thdb.l is the binary representation of th

  // Dekker's error-free adder of two ordered numbers
  Add12(th, tl, xm, xl);

  // round to odd th if tl is not zero
  if (tl != 0.0) {
    thdb.d = th;
    // if the mantissa of th is odd, there is nothing to do
    if (!(thdb.l & 1)) {
      // choose the rounding direction
      // depending on the signs of th and tl
      if ((tl > 0.0) ^ (th < 0.0))
        thdb.l++;
      else
        thdb.l--;
      th = thdb.d;
    }
  }

  // final addition rounded to nearest
  return xh + th;
}
```

consequence, the operands are ordered in such a way that some parts of Algorithm 4 are not necessary. In fact, theorem `Add0ddEven2` presented in Listing 3 implies the following equality:

$$\circ(x_h + x_m + x_l) = \circ(x_h + \square_{\text{odd}}(x_m + x_l)).$$

This means that, at the end of the logarithm function, we just have to compute the rounded-to-odd sum of x_m and x_l, and then do a standard fbating-point addition with x_h. Now, all that is left is computing $\square_{\text{odd}}(x_m + x_l)$. This will be done by first computing $t_h = \circ(x_m + x_l)$ and $t_l = x_m + x_l - t_h$ thanks to an error-free adder. If t_l is zero or if the mantissa of t_h is odd, then t_h is already equal to $\square_{\text{odd}}(x_m + x_l)$. Otherwise t_h is off by one unit in the last place. We replace it either by its successor or by its predecessor depending on the signs of t_l and t_h.

Listing 4 shows a cleaned version of the implementation of a macro that is internally used by CRlibm: `ReturnRoundToNearest3Other`. The macro `Add12` is an implementation of Dekker's error-free adder. It is only 3 addition long, and is correct since the inequality $|x_m| \geq |x_l|$ holds. The successor or the predecessor of t_h is directly computed by incrementing or decrementing the integer `thdb.l` that holds its binary representation. Working on the integer is correct, since t_h cannot be zero when t_l is not zero.

CRlibm already contained some code at the end of the logarithm function in order to compute the correctly rounded sum of three fbating-point numbers. When the code of Listing 4 is used instead, the slow step of this elementary function gets 25 cycles faster on an AMD Opteron processor. While we only looked at the very last operation of the logarithm, it still amounts to a 2% speed-up on the whole function.

The performance increase would obviously be even greater if we had not to emulate a rounded-to-odd addition. Moreover, this speed-up is not restricted to logarithm: it is available for every other rounded elementary functions, since they all rely on triple-double arithmetic at the end of their slow step.

VI. Conclusion

We first considered rounding to odd as a way of performing intermediate computations in an extended precision and yet still obtaining correctly rounded results at the end of the computations. This is expressed by the properties of Algorithm 1. This algorithm is even more general than what is presented here. It can also be applied to any realistic rounding to the closest (meaning

that the result of a computation is uniquely defined by the value of the infinitely precise result and does not depend on the machine state). In particular, it handles the new rounding to nearest, ties away from zero, defined by the revision of the IEEE-754 standard.

Rounding to odd then leads us to consider algorithms that could benefit from its robustness. We first considered an iterated summation algorithm that was using extended precision and rounding to odd in order to perform the intermediate additions. The FMA emulation however showed that the extended precision only has to be virtual. As long as we prove that the computations are done as if an extended precision was used, the working precision can be used. This is especially useful when we already compute with the highest available precision. This gives constraints on the inputs such that Algorithm 3 computes the correctly rounded sum of a set of fbating-point numbers.

Algorithm 2 for emulating the FMA and Algorithm 4 for adding numbers are similar. They both allow to compute $\circ(a \diamond b + c)$ with a, b, and c three fbating-point numbers, as long as $a \diamond b$ is exactly representable as the sum of two fbating-point numbers that can be computed. These algorithms rely on rounding to odd to ensure that the result is correctly rounded. Although this rounding is not available in current hardware, our changes to CRlibm have shown that reasoning on it opens the way to some efficient new algorithms for computing correctly rounded results.

In this paper, we did not tackle at all the problem of overfbwing operations. The reason is that overfbw does not matter here: on all the algorithms presented, overfbw can be detected afterward. Indeed, any of these algorithms will produce an infinity or a NaN as a result in case of overfbw. The only remaining problem is that they may create an infinity or a NaN although the result could be represented. For example, let M be the biggest positive fbating-point number, and let $a = -M$, $b = c = M$ in Algorithm 4. Then $u_h = t_h = +\infty$, $u_l = t_l = v = -\infty$ and $z = NaN$ whereas the correct result is M. This can be misleading, but this is not a real problem when adding three numbers. Indeed, the crucial point is that we cannot create inexact finite results: when the result is finite, it is correct. When emulating the FMA, it also requires the error-term of the product to be correctly computed. This property can be checked by verifying that the magnitude of the product is big enough.

All the algorithms presented here look short and simple, but their correctness is far from trivial. When rounding to odd is replaced by a standard rounding to nearest in them, there exist inputs such that the final results are no longer correctly rounded. So great care has to be taken

when asserting that simply changing one intermediate rounding is enough to fix an algorithm. So we have written formal proofs of their correctness and used the Coq proof-checker to guarantee their validity. This approach is essential to ensure that the algorithms are correct, even in the unusual cases.

REFERENCES

[1] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development. Coq'Art : the Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[2] Sylvie Boldo. *Preuves formelles en arithmétiques à virgule flottante*. PhD thesis, École Normale Supérieure de Lyon, November 2004.

[3] Sylvie Boldo and Guillaume Melquiond. When double rounding is odd. In *Proceedings of the 17th IMACS World Congress on Computational and Applied Mathematics*, Paris, France, 2005.

[4] Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library of floating-point numbers and its application to exact computing. In *14th International Conference on Theorem Proving in Higher Order Logics*, pages 169–184, Edinburgh, Scotland, 2001.

[5] Florent de Dinechin, Christoph Q. Lauter, and Jean-Michel Muller. Fast and correctly rounded logarithms in double-precision. *Theoretical Informatics and Applications*, 2006. To appear.

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

[7] James W. Demmel and Yozo Hida. Fast and accurate floating point summation with applications to computational geometry. In *Proceedings of the 10th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN 2002)*, January 2003.

[8] Miloš D. Ercegovac and Tomàs Lang. *Digital Arithmetic*. Morgan Kaufmann Publishers, 2004.

[9] Samuel A. Figueroa. When is double rounding innocuous? *SIGNUM Newsletter*, 30(3):21–26, 1995.

[10] David Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–47, 1991.

[11] Nicholas J. Higham. *Accuracy and stability of numerical algorithms*. SIAM, 1996.

[12] Donald E. Knuth. *The art of computer programming: Seminumerical Algorithms*, volume 2. Addison Wesley, 1969.

[13] Christoph Q. Lauter. Basic building blocks for a triple-double intermediate format. Technical Report RR2005-38, LIP, September 2005.

[14] Vincent Lefèvre and Jean-Michel Muller. Worst cases for correct rounding of the elementary functions in double precision. In Neil Burgess and Luigi Ciminiera, editors, *Proceedings of the 15th Symposium on Computer Arithmetic*, pages 111–118, Vail, Colorado, 2001.

[15] Guillaume Melquiond and Sylvain Pion. Formally certified floating-point filters for homogeneous geometric predicates. *Theoretical Informatics and Applications*, 2006. To appear.

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

[17] David Stevenson et al. A proposed standard for binary floating point arithmetic. *IEEE Computer*, 14(3):51–62, 1981.

[18] David Stevenson et al. An American national standard: IEEE standard for binary fbating point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, 1987.

[19] John von Neumann. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.