Correctly Rounded Multiplication by Arbitrary Precision Constants

Nicolas Brisebarre and Jean-Michel Muller, *Senior Member*, *IEEE*

Abstract—We introduce an algorithm for multiplying a floating-point number x by a constant C that is not exactly representable in floating-point arithmetic. Our algorithm uses a multiplication and a fused multiply and add instruction. Such instructions are available in some modern processors such as the IBM Power PC and the Intel/HP Itanium. We give three methods for checking whether, for a given value of C and a given floating-point format, our algorithm returns a correctly rounded result for any x. When it does not, some of our methods return all of the values x for which the algorithm fails. The three methods are complementary: The first two do not always allow one to conclude, yet they are simple enough to be used at compile time, while the third one always either proves that our algorithm returns a correctly rounded result for any x or gives all of the counterexamples. We generalize our study to the case where a wider internal format is used for the intermediate calculations, which gives a fourth method. Our programs and some additional information (such as the case where an arbitrary nonbinary even radix is used), as well as examples of runs of our programs, can be downloaded from http://perso.ens-lyon.fr/jean-michel.muller/MultConstant.html.

Index Terms—Floating-point arithmetic, computer arithmetic, multiplication by constants, fused multiply-add instruction, correct rounding.

1 Introduction

Many numerical algorithms require multiplication by constants that are not exactly representable in floating-point (FP) arithmetic. Typical constants that are used [1], [4] are π, $1/\pi$, $\ln(2)$, e, $B_k/k!$ (Euler-McLaurin summation), and $\cos(k\pi/N)$ and $\sin(k\pi/N)$ (Fast Fourier Transforms). Some numerical integration formulas such as [4]:

$$\int_{x_0}^{x_1} f(x)dx \approx h\left(\frac{55}{24}f(x_1) - \frac{59}{24}f(x_2) + \frac{37}{24}f(x_3) - \frac{9}{24}f(x_4)\right),$$

also naturally involve multiplication by constants. In the following, we call a number that is exactly representable in the FP format being used an *FP number* (unless it is clearly stated—as in Section 9—we assume that only one format is used so that there is no ambiguity).

For approximating Cx, where C is an infinite-precision constant and x is an FP number, the desirable result would be the best possible one, namely, $\circ(Cx)$, where $\circ(u)$ is u rounded to the nearest FP number (in this paper, we assume round to nearest even mode only). In practice, one usually defines a constant C_h, equal to the FP number that is closest to C, and actually computes C_hx (that is, what is returned is $\circ(C_hx)$). Due to the extra approximation of C by C_h, the obtained result is frequently different from $\circ(Cx)$, as shown in Section 2. Our goal here is to be able—at least for some constants and some FP formats—to return $\circ(Cx)$ for all input FP numbers x (provided no overflow or underflow occurs) and at a low cost (that is, using a very few arithmetic operations only). This will lead to more accurate numerical computations. To do that, we will use *fused multiply and add* (FMA) instructions. The FMA instruction is available on some current processors such as the IBM Power PC or the Intel/HP Itanium. It evaluates an expression $xy + z$, where x, y, and z are FP numbers, with one final rounding error only. This makes it possible to perform correctly rounded division using Newton-Raphson Division [3], [12], [13]. Li et al. use it for argument reduction [11]. Also, this makes evaluation of scalar products and polynomials faster and, generally, more accurate than with conventional (addition and multiplication) FP operations. We will examine two possible cases: In the first case, all intermediate calculations are performed in the "target" format. In the second case, the intermediate calculations are performed in a somewhat larger format (a frequent example is when the target format is the IEEE-754 double precision format and the intermediate calculations are performed in the double-extended precision format that is available on INTEL processors).

Our proofs will use the ulp function. Several slightly different definitions of $\mathrm{ulp}(x)$ do exist in the literature [5], [7], [8], [12], [15]. These definitions and their properties are compared in [14]. In this paper, we will use the following definition, given by Cornea-Hasegan et al. [3]: In a radix-2 FP system with n-bit mantissas, if $x \in [2^e, 2^{e+1})$, then $\mathrm{ulp}(x) = 2^{e-n+1}$. With this definition, we have the following properties (see [14]):

Assume that the binary FP number X approximates the real number x. Then,

- *N. Brisebarre is with the Laboratoire MUSE, Université Jean Monnet, Saint-Étienne, France, and Projet Arénaire of INRIA. E-mail: Nicolas.Brisebarre@ens-lyon.fr.*
- *J.-M. Muller is with LIP, ENS Lyon, 46 Allee d'Italie, 69364 Lyon Cedex 07, France. E-mail: jean-michel.muller@ens-lyon.fr.*

Manuscript received 7 July 2006; revised 27 Apr. 2007; accepted 3 Aug. 2007; published online 5 Sept. 2007.
Recommended for acceptance by M. Schulte.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0264-0706.
Digital Object Identifier no. 10.1109/TC.2007.70813.

TABLE 1
Proportion of Input Values x for which $\circ(C_h x) = \circ(Cx)$ for $C = \pi$ and Various Values of the Number n of Mantissa Bits

n	Proportion of correctly rounded results
5	0.93750
6	0.78125
7	0.59375
...	...
16	0.86765
17	0.73558
...	...
24	0.66805

$$|X - x| < \frac{1}{2}\mathrm{ulp}(x) \Rightarrow X = \circ(x),$$

$$X = \circ\,(x) \Rightarrow |X - x| \leq \frac{1}{2}\mathrm{ulp}(x),$$

$$X = \circ\,(x) \Rightarrow |X - x| \leq \frac{1}{2}\mathrm{ulp}(X),$$

$$|X - x| < \frac{1}{2}\mathrm{ulp}(X) \text{ does not imply } X = \circ(x).$$

An example for which $|X - x| < \frac{1}{2}\mathrm{ulp}(X)$ and yet $X \neq \circ(x)$ can be obtained by choosing $X = 1$ and $1 - 2^{-n} < x < 1 - 2^{-n-1}$. In such a case, $\circ(x) = 1 - 2^{-n} \neq 1$ and $|x - 1| < 2^{-n} = \frac{1}{2}\mathrm{ulp}(1)$. One can easily show the following result:

If one of the two following conditions is satisfied

- x is not of the form $2^k - \beta$, with $2^{k-n-1} < \beta < 2^{k-n}$ or
- X is not a power of 2,

then,

$$|X - x| < \frac{1}{2}\mathrm{ulp}(X) \Rightarrow X = \circ(x).$$

In Section 3, we give an algorithm for multiplying by a constant C. For most values of C but not all (see Section 4), it will return a correctly rounded value of Cx for all FP variables x. Hence, for a given value of C, we have to check whether our algorithm returns a correctly rounded result for all x. In Section 6, we present three methods to do that. The three methods are complementary: The first two do not always allow one to conclude, yet they are simple enough to be used at compile time, whereas the third always either proves that our algorithm returns a correctly rounded result for any x or gives all of the counterexamples. In Section 9, we extend our algorithm to the very frequent case where the calculations can be performed in an internal format that is larger than the "target" FP format.

2 Some Statistics

Let n be the number of mantissa bits of the considered FP format (usual values of n are 24, 53, 64, 113). For small values of n, one can compute $\circ(C_h x)$ and $\circ(Cx)$ for all possible values of the mantissa of x. The proportion of correctly rounded results is given in Table 1, for $C = \pi$ (it was obtained through a Maple simulation of these FP arithmetics). These results show that the "naive" method that consists of computing $\circ(C_h x)$ often returns an incorrectly rounded result (in around 41 percent of the cases for $n = 7$).

3 First Case: The Intermediate Calculations Are Performed in the "Target" Format

We want to compute Cx with correct rounding (assuming rounding to nearest even), where C is a constant (that is, C is known at compile time). C is not an FP number (otherwise, the problem would be straightforward). We assume that an FMA instruction is available. We assume that the operands are stored in a binary FP format with n-bit mantissas. We also assume that the two following FP numbers are precomputed:

$$\begin{cases} C_h = \circ(C), \\ C_\ell = \circ(C - C_h). \end{cases} \tag{1}$$

In the next sections of the paper, we analyze the behavior of the following algorithm. Specifically, we want to know for which values of C and n will it return a correctly rounded result for any x. When it does not, we wish to know for which values of x it does not.

Algorithm 1: Multiplication by C with a multiplication and an FMA). From x, compute

$$\begin{cases} u_1 = \circ(C_\ell x), \\ u_2 = \circ(C_h x + u_1). \end{cases} \tag{2}$$

The result to be returned is u_2.

It is worth pointing out that, without the use of FMA instruction, Algorithm 1 would fail to return a correctly rounded result for all but a few simple (for example, powers of 2) values of x.

When C is the exact reciprocal of an FP number, this algorithm coincides with an algorithm for division by a constant given in [2]. Obviously (provided no overflow/underflow occurs), if Algorithm 1 gives a correct result with a given constant C and a given input variable x, it will work as well with a constant $2^p C$ and an input variable $2^q x$, where p and q are integers. Also, if x is a power of 2 or if C is exactly representable (that is, $C_\ell = 0$) or if $C - C_h$ is a power of 2 (so that u_1 is exactly $(C - C_h)x$), it is straightforward to show that $u_2 = \circ(Cx)$. Hence, *without loss of generality, we assume in the following that* $1 < x < 2$ *and* $1 < C < 2$*, that* C *is not exactly representable, and that* $C - C_h$ *is not a power of 2.*

In Section 6, we give three methods for analyzing the behavior of Algorithm 1 for a given constant C. The first two certify that Algorithm 1 always returns a correctly rounded result, give a "bad case" (that is, a number x for which $u_2 \neq \circ(Cx)$), or are unable to infer anything. The third one is able to return all "bad cases," or certify that there are none. These methods use the following property, which bounds the maximum possible distance between u_2 and Cx in Algorithm 1. Of course, Algorithm 1 works for a given constant C and precision n if all FP numbers x are such that $|u_2 - Cx| < (1/2)\mathrm{ulp}(Cx)$. In most cases, this will be equivalent to $|u_2 - Cx| < (1/2)\mathrm{ulp}(u_2)$, which leads us to establish Property 2. Therefore, we need to prove Property 1, which gives the at most four possible values of x (between 1 and 2) for which $\mathrm{ulp}(u_2)$ may differ from $\mathrm{ulp}(Cx)$ or

$\mathrm{ulp}(C_h x + u_1)$. Algorithm 1 will need to be separately checked with these four values (this is done easily and quickly).

Property 1 (Equivalence of $\mathrm{ulp}(Cx)$, $\mathrm{ulp}(u_2)$, and $\mathrm{ulp}(C_h x + u_1)$). *Define $x_{\mathrm{cut}} = 2/C$.*

- *If $x < x_{\mathrm{cut}} - 2^{-n+2}$, then Cx, $C_h x + u_1$, and u_2 are in the same binade (hence, they have the same ulp).*
- *If $x > x_{\mathrm{cut}} + 2^{-n+2}$, then Cx, $C_h x + u_1$, and u_2 are in the same binade (hence, they have the same ulp).*

Proof. We just consider the case $x > x_{\mathrm{cut}} + 2^{-n+2}$ (the proof for the other case is very similar). If $x > x_{\mathrm{cut}} + 2^{-n+2}$, then $Cx > 2 + 2^{-n+2}$. Moreover, $C_h > C - 2^{-n}$; hence, $C_h x > (2 + 2^{-n+2}) - 2^{-n+1} \geq 2 + 2^{-n+1}$.

From $|C_\ell| \leq 2^{-n}$, we deduce $|C_\ell x| < 2^{-n+1}$; hence, $|u_1| \leq 2^{-n+1}$. Therefore, $C_h x + u_1 \geq 2$. Hence, $u_2 \geq 2$. □

Property 2. *Define $\epsilon_1 = |C - (C_h + C_\ell)|$:*

- *if $x < x_{\mathrm{cut}} - 2^{-n+2}$, then*

$$|u_2 - Cx| < (1/2)\mathrm{ulp}(u_2) + \eta,$$

- *if $x \geq x_{\mathrm{cut}} + 2^{-n+2}$, then*

$$|u_2 - Cx| < (1/2)\mathrm{ulp}(u_2) + \eta',$$

where

$$\begin{cases} \eta = \frac{1}{2}\mathrm{ulp}(C_\ell x_{\mathrm{cut}}) + \epsilon_1 x_{\mathrm{cut}}, \\ \eta' = \mathrm{ulp}(C_\ell) + 2\epsilon_1. \end{cases}$$

Proof. From $1 < C < 2$ and $C_h = \circ(C)$, we deduce $|C - C_h| < 2^{-n}$ (since $C - C_h$ is not a power of 2), which gives

$$|\epsilon_1| \leq \frac{1}{2}\mathrm{ulp}(C - C_h) \leq 2^{-2n-1}.$$

Now, we have

$$\begin{aligned} |u_2 - Cx| &\leq |u_2 - (C_h x + u_1)| \\ &\quad + |(C_h x + u_1) - (C_h x + C_\ell x)| \\ &\quad + |(C_h + C_\ell)x - Cx| \\ &\leq \frac{1}{2}\mathrm{ulp}(C_h x + u_1) \\ &\quad + |u_1 - C_\ell x| + \epsilon_1 |x| \\ &\leq \frac{1}{2}\mathrm{ulp}(u_2) + \frac{1}{2}\mathrm{ulp}(C_\ell x) \\ &\quad + \epsilon_1 |x|, \end{aligned} \tag{3}$$

and $\frac{1}{2}\mathrm{ulp}(C_\ell x) + \epsilon_1 |x|$ is less than $\frac{1}{2}\mathrm{ulp}(C_\ell x_{\mathrm{cut}}) + \epsilon_1 |x_{\mathrm{cut}}|$ if $|x| < x_{\mathrm{cut}}$ and less than $\mathrm{ulp}(C_\ell) + 2\epsilon_1$ if $x_{\mathrm{cut}} \leq x < 2$. □

Under the conditions of Property 1, if $|u_2 - Cx| < (1/2)\mathrm{ulp}(u_2)$, then $|u_2 - Cx| < (1/2)\mathrm{ulp}(Cx)$ so that u_2 is the FP number that is closest to Cx. Hence, our problem is to know if Cx can be at a distance larger than or equal to $\frac{1}{2}\mathrm{ulp}(u_2)$ from u_2. From (3), this would imply that Cx would be at a distance less than $\frac{1}{2}\mathrm{ulp}(C_\ell x) + \epsilon_1 |x| < 2^{-2n+1}$ from the midpoint of two consecutive FP numbers (see Fig. 1).

If $x < x_{\mathrm{cut}}$ then $Cx < 2$, then the midpoint of two consecutive FP numbers around Cx is of the form $(2A+1)/2^n$, where A is an integer between 2^{n-1} and

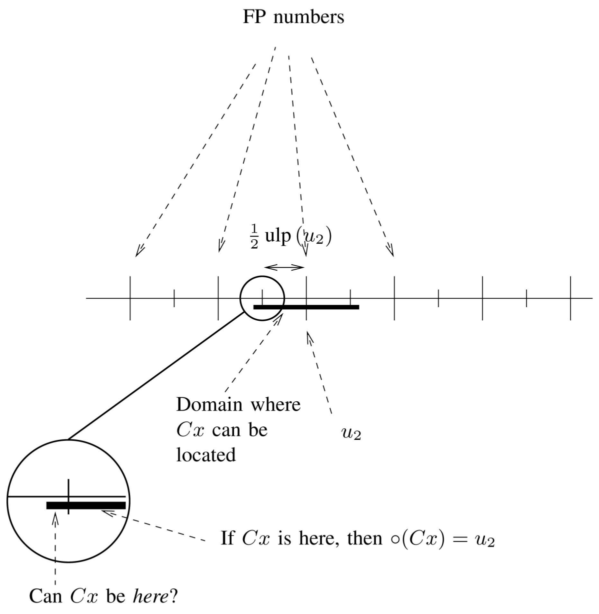
FP numbers
$\frac{1}{2}$ ulp (u_2)
Domain where Cx can be located
u_2
If Cx is here, then $\circ(Cx) = u_2$
Can Cx be *here*?

Fig. 1. From (3), we know that Cx is within $1/2\mathrm{ulp}(u_2) + \eta$ (or η') of the FP number u_2, where η is less than 2^{-2n+1}. If we can show that Cx cannot be at a distance less than or equal to η (or η') from the midpoint of two consecutive FP numbers, then u_2 will be the FP number that is closest to Cx.

$2^n - 1$. If $x \geq x_{\mathrm{cut}}$, then the midpoint of two consecutive FP numbers around Cx is of the form $(2A+1)/2^{n-1}$. For the sake of clarity of the proofs, we assume that x_{cut} is not an FP number (if x_{cut} is an FP number, it suffices to separately check Algorithm 1 with $x = x_{\mathrm{cut}}$).

4 Building Counterexamples

We do not claim that Algorithm 1 works for all values of C. Although the various examples we give at the end of the paper show that, in practice, for most usual values of C and most formats, Algorithm 1 returns a correctly rounded value for all x, it is extremely easy to build constants C for which the algorithm fails. Consider an integer A between 2^{n-1} and $2^n - 1$. The integer $2A+1$ is exactly representable with $n+1$ bits of mantissa but cannot be exactly represented with n bits. Also consider an integer X, $2^{n-1} < X < 2^n$ such that $(2A+1)/X$ is not an FP number. The integer X is exactly representable in our FP format.

Then, consider the following two constants:

$$C^+ = (2A + 1 + \delta)/X$$

and

$$C^- = (2A + 1 - \delta)/X,$$

where δ is chosen small enough so that we get the same values C_h and C_ℓ for C^+ and C^-. Adequate values of δ will depend on the bit patterns of $(2A+1)/X$ but will be less than 2^{-n}.

Now, consider Algorithm 1, with constant C^+ or C^- and input value X. Since both constants lead to the same values C_h and C_ℓ, Algorithm 1 will return the same value in both cases. Nevertheless, since $C^- X < 2A+1 < C^+ X$ and $2A+1$ is the exact middle of two FP numbers, the correctly rounded values of $C^- X$ and $C^+ X$ are necessarily different.

Therefore, either C^- or C^+ is a counterexample to Algorithm 1.

For instance, assume $n = 53$ (double precision format) and consider

$$2A + 1 = 2^{53} + 5 = 9007199254740997,$$
$$X = 3 \times 2^{51} = 6755399441055744,$$

and $\delta = 2^{-57}$. One can easily check that, called with constant

$$C^+ = (2A + 1 + \delta)/X = \frac{43269140487790254256952148719479 5}{32451855365842672678315602057625 6}$$

and variable X, Algorithm 1 fails to return a correctly rounded value of C^+X.

5 A Reminder on Continued Fractions

We just recall here the elementary results that we need in the following. For more information on continued fractions, see [6], [9], [16], [17].

Let α be a real number. From α, consider the two sequences (a_i) and (r_i) defined by

$$\begin{cases} r_0 &= \alpha, \\ a_i &= \lfloor r_i \rfloor, \\ r_{i+1} &= \frac{1}{r_i - a_i}. \end{cases} \tag{4}$$

If α is irrational, then these sequences are defined for any i (that is, r_i is never equal to a_i) and the rational number

$$\frac{p_i}{q_i} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots + \frac{1}{a_i}}}}}$$

is called the ith *convergent* of α. If α is rational, then these sequences finish for some k, and $p_k/q_k = \alpha$ exactly. The p_is and the q_is can be deduced from the a_i using the following recurrences:

$$\begin{cases} p_0 = a_0, \\ p_1 = a_1 a_0 + 1, \\ q_0 = 1, \end{cases} \qquad \begin{cases} q_1 = a_1, \\ p_n = p_{n-1} a_n + p_{n-2}, \\ q_n = q_{n-1} a_n + q_{n-2}. \end{cases}$$

The major interest of the continued fractions lies in the fact that p_i/q_i is the best rational approximation to α among all rational numbers of denominator less than or equal to q_i (Theorem 1 states an even stronger result).

We will use the following two results [6]:

Theorem 1. *Let $(p_j/q_j)_{j \geq 1}$ be the convergents of α. If q_{n+1} exists, then, for any $(p, q) \in \mathbb{Z} \times \mathbb{N}^*$, with $q < q_{n+1}$, we have*

$$|p - \alpha q| \geq |p_n - \alpha q_n|.$$

If q_{n+1} does not exist (which implies that α is rational), then the previous inequality holds for any $(p, q) \in \mathbb{Z} \times \mathbb{N}^*$.

Theorem 2. *Let p, q be integers, $q \neq 0$. If*

$$\left| \frac{p}{q} - \alpha \right| < \frac{1}{2q^2},$$

then p/q is a convergent of α.

6 Three Methods for Analyzing Algorithm 1

We assume that Algorithm 1 is preliminarily checked with the at most four values of x that are between $x_{\text{cut}} - 2^{-n+2}$ and $x_{\text{cut}} + 2^{-n+2}$. Hence, in the rest of this section, all FP numbers x considered are such that $\text{ulp}(Cx) = \text{ulp}(C_h x + u_1) = \text{ulp}(u_2)$. Our algorithms may be slow in the case where C is equal or extremely close to a rational number of small (in front of 2^n) denominators. Nevertheless, in the case $C = 2^k/p$, where $|p| \leq 2^n - 1$, the method given in [2] can be applied. For instance, it allows us to conclude that the algorithm always works with constants 10^{-k}, with $0 \leq k \leq 22$.

6.1 Method 1: Use of Theorem 1

Define $X = 2^{n-1}x$ and $X_{\text{cut}} = \lfloor 2^{n-1} x_{\text{cut}} \rfloor$. X and X_{cut} are integers. X is between $2^{n-1} + 1$ and $2^n - 1$ and X_{cut} is between 2^{n-1} and $2^n - 1$. We separate the cases $x < x_{\text{cut}}$ and $x > x_{\text{cut}}$.

6.1.1 If $x < x_{\text{cut}}$

We want to know if there is an integer A between 2^{n-1} and $2^n - 1$ such that

$$\left| Cx - \frac{2A+1}{2^n} \right| < \eta, \tag{5}$$

where η is defined in Property 2. Inequality (5) is equivalent to

$$|2CX - 2A - 1| < 2^n \eta. \tag{6}$$

Define $(p_i/q_i)_{i \geq 1}$ as the convergents of $2C$. Let k be the smallest integer such that $q_{k+1} > X_{\text{cut}}$ and define $\delta = |p_k - 2Cq_k|$. Theorem 1 implies that, for any $A, X \in \mathbb{Z}$, with $0 < X \leq X_{\text{cut}}$, we have

$$|2CX - 2A - 1| \geq \delta.$$

Therefore,

1. if $\delta \geq 2^n \eta$, then $|Cx - (2A + 1)/2^n| < \eta$ is impossible. *In that case, Algorithm 1 returns a correctly rounded result for any $x < x_{\text{cut}}$;*
2. if $\delta < 2^n \eta$, then we try Algorithm 1 with $y = q_k 2^{-n+1}$. If the obtained result is not $\circ(yC)$, then *we know that Algorithm 1 fails for at least one value.*[1] Otherwise, we cannot infer anything.

6.1.2 If $x > x_{\text{cut}}$

We want to know if there is an integer A between 2^{n-1} and $2^n - 1$ such that

$$\left| Cx - \frac{2A+1}{2^{n-1}} \right| < \eta', \tag{7}$$

where η' is defined in Property 2. Inequality (7) is equivalent to

$$|CX - 2A - 1| < 2^{n-1} \eta'. \tag{8}$$

Define $(p'_i/q'_i)_{i \geq 1}$ as the convergents of C. Let k' be the smallest integer such that $q'_{k'+1} \geq 2^n$ and define

1. It is possible that y may be not between 1 and x_{cut}. It will be a counterexample anyway, that is, an n-bit number for which Algorithm 1 fails.

$\delta' = |p'_{k'} - Cq'_{k'}|$. Theorem 1 implies that, for any $A, X \in \mathbb{Z}$, with $X_{\text{cut}} \le X < 2^n$, $|CX - 2A - 1| \ge \delta'$. Therefore,

1. if $\delta' \ge 2^{n-1}\eta'$, then $|Cx - (2A+1)/2^{n-1}| < \eta'$ is impossible. *In that case, Algorithm 1 returns a correctly rounded result for any $x > x_{\text{cut}}$;*
2. if $\delta' < 2^{n-1}\eta'$, then we try Algorithm 1 with $y = q'_{k'}2^{-n+1}$. If the obtained result is not $\circ(yC)$, then *we know that Algorithm 1 fails for at least one value.* Otherwise, we cannot infer anything.

6.2 Method 2: Use of Theorem 2

Again, we use $X = 2^{n-1}x$ and $X_{\text{cut}} = \lfloor 2^{n-1}x_{\text{cut}} \rfloor$ and we separate the cases $x < x_{\text{cut}}$ and $x > x_{\text{cut}}$.

6.2.1 If $x < x_{\text{cut}}$

If

$$\left|Cx - \frac{2A+1}{2^n}\right| < \epsilon_1 x_{\text{cut}} + \frac{1}{2}\text{ulp}(C_\ell x_{\text{cut}}),$$

then

$$\left|2C - \frac{2A+1}{X}\right| < 2^n \times \frac{\epsilon_1 x_{\text{cut}} + \frac{1}{2}\text{ulp}(C_\ell x_{\text{cut}})}{X}.$$

Therefore, since $X \le X_{\text{cut}}$, if

$$\epsilon_1 x_{\text{cut}} + \frac{1}{2}\text{ulp}(C_\ell x_{\text{cut}}) \le \frac{1}{2^{n+1}X_{\text{cut}}}, \tag{9}$$

then we can apply Theorem 2: If $|Cx - (2A+1)/2^n| < \epsilon_1 x_{\text{cut}} + \frac{1}{2}\text{ulp}(C_\ell x_{\text{cut}})$, then $(2A+1)/X$ is a convergent of $2C$.

In that case, we have to check the convergents of $2C$ of denominator less than or equal to X_{cut}. A given convergent p/q (with $\gcd(p,q)=1$) is a candidate for generating a value X for which Algorithm 1 does not work if there exist $X = mq$ and $2A+1 = mp$ such that

$$\begin{cases} 2^{n-1}+1 \le X \le X_{\text{cut}}, \\ 2^{n-1} \le A \le 2^n - 1, \\ \left|\frac{CX}{2^{n-1}} - \frac{2A+1}{2^n}\right| < \epsilon_1 x_{\text{cut}} + \frac{1}{2}\text{ulp}(C_\ell x_{\text{cut}}). \end{cases}$$

This would mean

$$\left|C\frac{mq}{2^{n-1}} - \frac{mp}{2^n}\right| < \epsilon_1 x_{\text{cut}} + \frac{1}{2}\text{ulp}(C_\ell x_{\text{cut}}),$$

which would imply

$$|2Cq - p| < \frac{2^n}{m^*}\left(\epsilon_1 x_{\text{cut}} + \frac{1}{2}\text{ulp}(C_\ell x_{\text{cut}})\right), \tag{10}$$

where $m^* = \lceil 2^{n-1}/q \rceil$ is the smallest possible value of m. Hence, if (10) is not satisfied, convergent p/q cannot generate a bad case for Algorithm 1.

Now, if (10) is satisfied, we have to check Algorithm 1 with all values $X = mq$, with $m^* \le m \le \lfloor X_{\text{cut}}/q \rfloor$.

6.2.2 If $x > x_{\text{cut}}$

If

$$\left|Cx - \frac{2A+1}{2^{n-1}}\right| < \epsilon_1 x + \frac{1}{2}\text{ulp}(C_\ell x),$$

then

$$\left|C - \frac{2A+1}{X}\right| < \epsilon_1 + \frac{2^{n-2}}{X}\text{ulp}(C_\ell x). \tag{11}$$

Now, if

$$2^{2n+1}\epsilon_1 + 2^{2n-1}\text{ulp}(2C_\ell) \le 1, \tag{12}$$

then, for any $X < 2^n$ (that is, $x < 2$),

$$\epsilon_1 + \frac{2^{n-2}}{X}\text{ulp}(C_\ell x) < \frac{1}{2X^2}.$$

Hence, if (12) is satisfied, then (11) implies (from Theorem 2) that $(2A+1)/X$ is a convergent of C. This means that if (12) is satisfied to find the possible bad cases for Algorithm 1, it suffices to examine the convergents of C of denominator less than 2^n. We can eliminate most of them. A given convergent p/q (with $\gcd(p,q)=1$) is a candidate for generating a value X for which Algorithm 1 does not work if there exist $X = mq$ and $2A+1 = mp$ such that

$$\begin{cases} X_{\text{cut}} < X \le 2^n - 1, \\ 2^{n-1} \le A \le 2^n - 1, \\ \left|\frac{CX}{2^{n-1}} - \frac{2A+1}{2^{n-1}}\right| < \epsilon_1 \frac{X}{2^{n-1}} + \frac{1}{2}\text{ulp}(C_\ell x). \end{cases}$$

This would mean

$$\left|C\frac{mq}{2^{n-1}} - \frac{mp}{2^{n-1}}\right| < \epsilon_1 \frac{mq}{2^{n-1}} + \frac{1}{2}\text{ulp}(2C_\ell),$$

which would imply that

$$|Cq - p| < \epsilon_1 q + \frac{2^{n-1}}{m^*}\text{ulp}(C_\ell), \tag{13}$$

where $m^* = \lceil X_{\text{cut}}/q \rceil$ is the smallest possible value of m. Hence, if (13) is not satisfied, convergent p/q cannot generate a bad case for Algorithm 1.

Now, if (13) is satisfied, we have to check Algorithm 1 with all values $X = mq$, with $m^* \le m \le \lfloor (2^n - 1)/q \rfloor$.

6.2.3 Conclusion

This last result and (3) make it possible to deduce the following:

Theorem 3 (Conditions on C and n). *Assume that $1 < C < 2$. Let $x_{\text{cut}} = 2/C$ and $X_{\text{cut}} = \lfloor 2^{n-1}x_{\text{cut}} \rfloor$:*

- *If $X = 2^{n-1}x \le X_{\text{cut}}$ and $\epsilon_1 x_{\text{cut}} + 1/2\text{ulp}(C_\ell x_{\text{cut}}) \le 1/(2^{n+1}X_{\text{cut}})$, then Algorithm 1 will always return a correctly rounded result, except possibly if X is a multiple of the denominator of a convergent p/q of $2C$ for which $|2Cq - p| < \frac{2^n}{\lceil 2^{n-1}/q \rceil}\left(\epsilon_1 x_{\text{cut}} + \frac{1}{2}\text{ulp}(C_\ell x_{\text{cut}})\right)$.*
- *If $X = 2^{n-1}x > X_{\text{cut}}$ and $2^{2n+1}\epsilon_1 + 2^{2n-1}\text{ulp}(2C_\ell) \le 1$, then Algorithm 1 will always return a correctly rounded result, except possibly if X is a multiple of the denominator of a convergent p/q of C for which $|Cq - p| < \epsilon_1 q + \frac{2^{n-1}}{\lceil X_{\text{cut}}/q \rceil}\text{ulp}(C_\ell)$.*

6.3 Method 3: Refinement of Method 2

When Method 2 fails to return an answer, we can use the following method.

We have $|C - C_h| < 2^{-n}$; hence, $\text{ulp}(C_\ell) \le 2^{-2n}$ or $C_\ell = 2^{-n}$.

6.3.1 If $x < x_{\mathrm{cut}}$

If $\mathrm{ulp}(C_\ell) \leq 2^{-2n-2}$, then we have, from Property 2:

$$|u_2 - Cx| < \frac{1}{2}\mathrm{ulp}(u_2) + 2^{-2n-1}.$$

For any integer A, the inequality

$$\left|Cx - \frac{2A+1}{2^n}\right| \leq \frac{1}{2^{2n+1}}$$

implies that

$$|2CX - 2A - 1| \leq \frac{1}{2^{n+1}} < \frac{1}{2X}.$$

$(2A+1)/X$ is a convergent of $2C$ from Theorem 2. It then suffices to check (as in Method 2) the convergents of $2C$ of the denominator less than or equal to X_{cut}.

Now, assume that $\mathrm{ulp}(C_\ell) \geq 2^{-2n-1}$. We have

$$-\mathrm{ulp}(C_\ell) + C_\ell\frac{X}{2^{n-1}} \leq u_1 \leq \mathrm{ulp}(C_\ell) + C_\ell\frac{X}{2^{n-1}},$$

that is,

$$\begin{aligned} &-2^{2n}\mathrm{ulp}(C_\ell) + 2^{n+1}C_\ell X \leq u_1 2^{2n} \\ &\leq 2^{2n}\mathrm{ulp}(C_\ell) + 2^{n+1}C_\ell X. \end{aligned} \tag{14}$$

The proof of Property 2 leads us to look for the integers X, $2^{n-1}+1 \leq X \leq X_{\mathrm{cut}}$, such that there exists an integer A, $2^{n-1} \leq A \leq 2^n - 1$, with

$$\left|C_h\frac{X}{2^{n-1}} + u_1 - \frac{2A+1}{2^n}\right| < 2\mathrm{ulp}(C_\ell),$$

that is,

$$\left|\frac{C_h X}{2^n\mathrm{ulp}(C_\ell)} + \frac{u_1}{2\mathrm{ulp}(C_\ell)} - \frac{2A+1}{2^{n+1}\mathrm{ulp}(C_\ell)}\right| < 1.$$

Since $u_1/(2\mathrm{ulp}(C_\ell))$ is half an integer and $\frac{C_h X}{2^n\mathrm{ulp}(C_\ell)}$ and $\frac{2A+1}{2^{n+1}\mathrm{ulp}(C_\ell)}$ are integers, we have

$$\frac{C_h X}{2^n\mathrm{ulp}(C_\ell)} + \frac{u_1}{2\mathrm{ulp}(C_\ell)} - \frac{2A+1}{2^{n+1}\mathrm{ulp}(C_\ell)} = 0, \pm 1/2.$$

Then, combining these three equations with (14), we get the following three pairs of inequalities:

$$\begin{gathered} 0 \leq 2X(C_h + C_\ell) - (2A+1) + 2^n\mathrm{ulp}(C_\ell) \\ \leq 2^{n+1}\mathrm{ulp}(C_\ell), \\ 0 \leq 2X(C_h + C_\ell) - (2A+1) \leq 2^{n+1}\mathrm{ulp}(C_\ell), \\ 0 \leq 2X(C_h + C_\ell) - (2A+1) + 2^{n+1}\mathrm{ulp}(C_\ell) \\ \leq 2^{n+1}\mathrm{ulp}(C_\ell). \end{gathered}$$

For $y \in \mathbb{R}$, let $\{y\}$ be the fractional part of y: $\{y\} = y - \lfloor y \rfloor$. These three inequalities can be rewritten as

$$\begin{gathered} \{X(C_h + C_\ell) + 2^{n-1}\mathrm{ulp}(C_\ell) - 1/2\} \leq 2^n\mathrm{ulp}(C_\ell), \\ \{X(C_h + C_\ell) - 1/2\} \leq 2^n\mathrm{ulp}(C_\ell), \\ \{X(C_h + C_\ell) + 2^n\mathrm{ulp}(C_\ell) - 1/2\} \leq 2^n\mathrm{ulp}(C_\ell). \end{gathered}$$

We use an efficient algorithm due to Lefèvre [10] to determine the integers X solution of each inequality.

6.3.2 If $x > x_{\mathrm{cut}}$

If $\mathrm{ulp}(C_\ell) \leq 2^{-2n-1}$, then we have

$$|u_2 - Cx| < \frac{1}{2}\mathrm{ulp}(u_2) + 2^{-2n}.$$

Therefore, for any integer A, the inequality

$$\left|Cx - \frac{2A+1}{2^{n-1}}\right| \leq \frac{1}{2^{2n}}$$

is equivalent to

$$|CX - 2A - 1| \leq \frac{1}{2^{n+1}} < \frac{1}{2X}.$$

$(2A+1)/X$ is necessarily a convergent of C from Theorem 2. It suffices then to check, as indicated in Method 2, the convergents of C of denominator less than or equal to $2^n - 1$.

Now, assume that $\mathrm{ulp}(C_\ell) = 2^{-2n}$ or $C_\ell = 2^{-n}$. The proof of Property 2 leads us to look for the integers X, $X_{\mathrm{cut}} + 1 \leq X \leq 2^n - 1$, such that there exists an integer A, $2^{n-1} \leq A \leq 2^n - 1$, with

$$\left|C_h\frac{X}{2^{n-1}} + u_1 - \frac{2A+1}{2^{n-1}}\right| < \frac{1}{2^{2n-1}},$$

that is,

$$\left|2^{n+1}C_h X + u_1 2^{2n} - 2^{n+1}(2A+1)\right| < 2.$$

Since $u_1 2^{2n}$, $2^{n+1}C_h X$, and $2^{n+1}(2A+1) \in \mathbb{Z}$, we have

$$2^{n+1}C_h X + u_1 2^{2n} - 2^{n+1}(2A+1) = 0, \pm 1.$$

Then, combining this equation with (14), we get the three pairs of inequalities

$$\begin{gathered} 0 \leq X(C_h + C_\ell) - (2A+1) + \frac{1}{2^{n+1}} \leq \frac{1}{2^n}, \\ 0 \leq X(C_h + C_\ell) - (2A+1) \leq \frac{1}{2^n}, \\ 0 \leq X(C_h + C_\ell) - (2A+1) + \frac{1}{2^n} \leq \frac{1}{2^n}, \end{gathered}$$

that is to say,

$$\begin{gathered} \left\{X\frac{C_h + C_\ell}{2} - \frac{1}{2} + \frac{1}{2^{n+2}}\right\} \leq \frac{1}{2^{n+1}}, \\ \left\{X\frac{C_h + C_\ell}{2} - \frac{1}{2}\right\} \leq \frac{1}{2^{n+1}}, \\ \left\{X\frac{C_h + C_\ell}{2} - \frac{1}{2} + \frac{1}{2^{n+1}}\right\} \leq \frac{1}{2^{n+1}}. \end{gathered}$$

Here again, we use Lefèvre's algorithm [10] to determine the integers X solution of these inequalities.

7 EXAMPLES

7.1 Example 1: Multiplication by π in Double Precision

Consider the case $C = \pi/2$ (which corresponds to multiplication by any number of the form $2^{\pm j}\pi$) and $n = 53$ (double precision) and assume we use Method 1. We find

$$\begin{cases} C_h & = & 884279719003555/562949953421312, \\ C_\ell & = & 6.123233996\cdots \times 10^{-17}, \\ \epsilon_1 & = & 1.497384905\cdots \times 10^{-33}, \\ x_{\text{cut}} & = & 1.2732395447351626862\cdots, \\ \text{ulp}(C_\ell x_{\text{cut}}) & = & 2^{-106}, \\ \text{ulp}(C_\ell) & = & 2^{-106}. \end{cases}$$

Hence,

$$\begin{cases} 2^n\eta & = & 7.268364390 \times 10^{-17}, \\ 2^{n-1}\eta' & = & 6.899839541 \times 10^{-17}. \end{cases}$$

Computing the convergents of $2C$ and C, we find

$$\frac{p_k}{q_k} = \frac{6134899525417045}{1952799169684491}$$

and $\delta = 9.495905771 \times 10^{-17} > 2^n\eta$ (which means that Algorithm 1 works for $x < x_{\text{cut}}$) and

$$\frac{p'_{k'}}{q'_{k'}} = \frac{12055686754159438}{7674888557167847}$$

and $\delta' = 6.943873667 \times 10^{-17} > 2^{n-1}\eta'$ (which means that Algorithm 1 works for $x > x_{\text{cut}}$). We therefore deduce the following:

Theorem 4 (Correctly rounded multiplication by π). *Algorithm 1 always returns a correctly rounded result in double precision with $C = 2^j\pi$, where j is any integer, provided that no under/overflow occurs.*

Hence, in that case, multiplying by π with correct rounding only requires two consecutive FMAs.

7.2 Example 2: Multiplication by $\ln(2)$ in Double Precision

Consider the case $C = 2\ln(2)$ (which corresponds to multiplication by any number of the form $2^{\pm j}\ln(2)$), and $n = 53$, and assume we use Method 2. We find that

$$\begin{cases} C_h & = & \frac{6243314768165359}{4503599627370496}, \\ C_\ell & = & 4.638093628\cdots \times 10^{-17}, \\ x_{\text{cut}} & = & 1.442695\cdots, \\ \epsilon_1 & = & 1.141541688\cdots \times 10^{-33}, \\ \epsilon_1 x_{\text{cut}} & & \\ +\frac{1}{2}\text{ulp}(C_\ell x_{\text{cut}}) & = & 7.8099\cdots \times 10^{-33}, \\ 1/(2^{n+1}X_{\text{cut}}) & = & 8.5437\cdots \times 10^{-33}. \end{cases}$$

Since $\epsilon_1 x_{\text{cut}} + (1/2)\text{ulp}(C_\ell x_{\text{cut}}) \le 1/(2^{n+1}X_{\text{cut}})$, to find the possible bad cases for Algorithm 1 that are less than x_{cut}, it suffices to check the convergents of $2C$ of denominator less than or equal to X_{cut}. These convergents are

$2, 3, 11/4, 25/9, 36/13, 61/22, 890/321, 2731/985,$
$25469/9186, 1097898/395983, 1123367/405169,$
$2221265/801152, 16672222/6013233, 18893487/6814385,$
$35565709/12827618, 125590614/45297239,$
$161156323/58124857, 609059583/219671810,$
$1379275489/497468477, 1988335072/717140287,$
$5355945633/1931749051, 7344280705/2648889338,$
$27388787748/9878417065, 34733068453/12527306403,$
$62121856201/22405723468, 96854924654/34933029871,$
$449541554817/162137842952,$
$2794104253556/1007760087583,$
$3243645808373/1169897930535,$
$6037750061929/2177658018118,$
$39470146179947/14235846039243,$
$124448188601770/44885196135847,$
$163918334781717/59121042175090,$
$288366523383487/104006238310937,$
$6219615325834944/2243252046704767.$

None of them satisfies (10). Therefore, there are no bad cases less than x_{cut}. Processing the case $x > x_{\text{cut}}$ is similar and gives the same result; hence, we have the following:

Theorem 5 (Correctly rounded multiplication by $\ln(2)$). *Algorithm 1 always returns a correctly rounded result in double precision with $C = 2^j\ln(2)$, where j is any integer, provided that no under/overflow occurs.*

7.3 Example 3 Multiplication by $1/\pi$ in Double Precision

Consider the case $C = 4/\pi$ and $n = 53$ and assume we use Method 1. We find

$$\begin{cases} C_h & = & \frac{5734161139222659}{4503599627370496}, \\ C_\ell & = & -7.871470670\cdots \times 10^{-17}, \\ \epsilon_1 & = & 4.288574513\cdots \times 10^{-33}, \\ x_{\text{cut}} & = & 1.570796\cdots, \\ C_\ell x_{\text{cut}} & = & -1.236447722\cdots \times 10^{-16}, \\ \text{ulp}(C_\ell x_{\text{cut}}) & = & 2^{-105}, \\ 2^n\eta & = & 1.716990939\cdots \times 10^{-16}, \\ p_k/q_k & = & \frac{15486085235905811}{6081371451248382}, \\ \delta & = & 7.669955467\cdots \times 10^{-17}. \end{cases}$$

Consider the case $x < x_{\text{cut}}$. Since $\delta < 2^n\eta$, there can be bad cases for Algorithm 1. We try Algorithm 1 with X equal to the denominator of p_k/q_k, that is, 6081371451248382, and we find that it does not return $\circ(cX)$ for that value. Hence, *there is at least one value of x for which Algorithm 1 does not work.*

Method 3 certifies that $X = 6081371451248382$, that is, $6081371451248382 \times 2^{\pm k}$ are the *only* FP numbers for which Algorithm 1 fails.

7.4 Example 4: Multiplication by $\sqrt{2}$ in Single Precision

Consider the case $C = \sqrt{2}$ and $n = 24$ (which corresponds to single precision) and assume we use Method 1. We find

$$\begin{cases} C_h & = 11863283/8388608, \\ C_\ell & = 2.420323497 \cdots \times 10^{-8}, \\ \epsilon_1 & = 7.628067479 \cdots \times 10^{-16}, \\ X_{\text{cut}} & = 11863283, \\ \text{ulp}(C_\ell x_{\text{cut}}) & = 2^{-48}, \\ 2^n \eta & = 4.790110735 \cdots \times 10^{-8}, \\ p_k/q_k & = 22619537/7997214, \\ \delta & = 2.210478490 \cdots \times 10^{-8}, \\ 2^{n-1}\eta' & = 2.769893477 \cdots \times 10^{-8}, \\ p_{k'}/q_{k'} & = 22619537/15994428, \\ \delta' & = 2.210478490 \cdots \times 10^{-8}. \end{cases}$$

Since $2^n\eta > \delta$ and $X = q_k = 7997214$ is not a bad case, we cannot infer anything in the case $x < x_{\text{cut}}$. Also, since $2^{n-1}\eta' > \delta'$ and $X = q_{k'} = 15994428$ is not a bad case, we cannot infer anything in the case $x \geq x_{\text{cut}}$. Hence, in the case $C = \sqrt{2}$ and $n = 24$, Method 1 does not allow us to know if the multiplication algorithm works for any input FP number x. In that case, Method 2 also fails. Nevertheless, Method 3 or exhaustive testing (which is possible since $n = 24$ is reasonably small) show that Algorithm 1 always works.

7.5 Example 5: Powers of 10

Consider constants C of the form 10^k, in double precision. If $0 \leq k \leq 22$, C is exactly representable so that a simple FP multiplication is used. Otherwise, for $-100 \leq k \leq 100$, the only values of k for which Algorithm 1 fails to always return a correctly rounded result are -89, -88, -80, -75, -74, -39, -27, 44, 58, 81, 88, and 97 (the case $-22 \leq k \leq -1$ is addressed using that in [2]).

8 Implementation and Results

As the reader will have guessed from the previous examples, using our Methods by paper and pencil calculation is tedious and error prone. We have written Maple programs that implement Methods 1, 2, and 3, and a GP/PARI[2] program that implements Method 3. They allow any user to quickly check, for a given constant C and a given number n of mantissa bits, if Algorithm 1 works for any x, and Method 3 gives all values of x for which it does not work (if there are such values). These programs and some additional information (such as the case where an arbitrary nonbinary even radix is used) can be downloaded from http://perso.ens-lyon.fr/jean-michel.muller/MultConstant.html

These programs, along with some examples, are given in the appendix. Table 2 presents some obtained results. They show that implementing Method 1, Method 2, *and* Method 3 is necessary: Methods 1 and 2 do not return a result (either a bad case or the fact that Algorithm 1 always works) for the same values of C and n. For instance, in the case $C = \pi/2$ and $n = 53$, we know that, thanks to Method 1, the multiplication algorithm always works, whereas Method 2 fails to give an answer. On the contrary, in the case $C = 1/\ln(2)$ and $n = 24$, Method 1 does not give an answer, whereas Method 2 makes it possible to show that the algorithm always works. Method 3 always returns an answer but is more complicated to implement: This is not a problem for getting a result such as Theorem 4 in advance for a general constant C. Nevertheless, this might make Method 3 difficult to implement in a compiler, to decide at compile time if we can use our algorithm.

2. http://pari.math.u-bordeaux.fr/.

TABLE 2
Some Results Obtained Using Methods 1, 2, and 3

C	n	method 1	method 2	method 3
π	8	Does not work for 226	Does not work for 226	AW (c) unless $X =$ 226
π	24	unable	unable	AW
π	53	AW	unable	AW
π	64	unable	AW	AW (c)
π	113	AW	AW	AW (c)
$1/\pi$	24	unable	unable	AW
$1/\pi$	53	Does not work for 6081371451248382	unable	AW unless $X =$ 6081371451248382
$1/\pi$	64	AW	AW	AW (c)
$1/\pi$	113	unable	unable	AW
$\ln 2$	24	AW	AW	AW (c)
$\ln 2$	53	AW	unable	AW (c)
$\ln 2$	64	AW	unable	AW (c)
$\ln 2$	113	AW	AW	AW (c)
$\frac{1}{\ln 2}$	24	unable	AW	AW (c)
$\frac{1}{\ln 2}$	53	AW	AW	AW (c)
$\frac{1}{\ln 2}$	64	unable	unable	AW
$\frac{1}{\ln 2}$	113	unable	unable	AW
$\ln 10$	24	unable	AW	AW (c)
$\ln 10$	53	unable	unable	AW
$\ln 10$	64	unable	AW	AW (c)
$\ln 10$	113	AW	AW	AW (c)
$\frac{1}{\ln 10}$	24	unable	unable	AW
$\frac{1}{\ln 10}$	53	unable	AW	AW (c)
$\frac{1}{\ln 10}$	64	unable	AW	AW (c)
$\frac{1}{\ln 10}$	113	unable	unable	AW
$\cos\frac{\pi}{8}$	24	unable	unable	AW
$\cos\frac{\pi}{8}$	53	AW	AW	AW (c)
$\cos\frac{\pi}{8}$	64	AW	unable	AW
$\cos\frac{\pi}{8}$	113	unable	AW	AW (c)

The results given for constant C hold for all values $2^{\pm j}C$ "AW" means "always works" and "unable" means "the method is unable to conclude." For Method 3, "(c)" means that we have needed to check the convergents.

9 Second Case: Assuming Intermediate Calculations in a Larger Format

Intermediate calculations will frequently be performed in an internal format that is significantly larger than the "target" format. For instance, on Intel processors, the intermediate calculations can be performed in a double-extended precision (with 64-bit mantissas), with the final result being converted to double precision. This is done at no cost since the only operators that are actually implemented are double-extended precision operators. Hence, it is important to see what changes in Algorithm 1 and its properties in this case.

Therefore, in the following, we still assume a "target" n-bit format, but we assume that C_h and C_ℓ are computed in a larger format, with $n+g$ bits of mantissa. We also assume that Algorithm 1 is performed in that $n+g$-bit format, before its final result is rounded to the nearest number with n-bit mantissa (using the standard round-to-nearest-even rounding mode). We still assume that an FMA instruction is available. We will denote $\mathrm{ulp}_t(x)$ as the value of $\mathrm{ulp}(x)$ in a format with t bits of mantissa (the relevant values of t will be n and $n+g$).

One might be intuitively convinced that we will get a correctly rounded result more frequently than when only using n-bit arithmetic. This, of course, will be true in general, as shown by the examples. Nevertheless, it is still easy to build cases for which we get an incorrect result: It suffices to use the method presented in Section 4, with a smaller value of δ. Here again, the case where C or $2C$ is a rational number of denominator less than $2^n - 1$ (remember that $1 < C < 2$) is separately handled (as in Section 6).

Define $\circ_k(w)$ as w rounded to the nearest FP number with k bits of mantissa. We will now use the constants

$$\begin{cases} C_h = \circ_{n+g}(C), \\ C_\ell = \circ_{n+g}(C - C_h), \end{cases} \tag{15}$$

and the following algorithm.

Algorithm 2: Multiplication by C using an intermediate format with $n+g$ bits of mantissa. From x, compute

$$\begin{cases} u_1 = \circ_{n+g}(C_\ell x), \\ u_2 = \circ_n(C_h x + u_1). \end{cases} \tag{16}$$

The result to be returned is u_2.

Analyzing this algorithm will be done in a way that is very similar to what we did for Algorithm 1, so we will skip the details. First, we preliminarily check the values of x between $x_{\text{cut}} - 2^{-n+2}$ and $x_{\text{cut}} + 2^{-n+2}$. We can then show that, in all remaining cases,

$$\begin{aligned} |u_2 - Cx| &< \frac{1}{2}\mathrm{ulp}_n(u_2) + \mathrm{ulp}_{n+g}(C_\ell) + 2^{-2(n+g)} \\ &< \frac{1}{2}\mathrm{ulp}_n(u_2) + 2^{-2(n+g)+1}. \end{aligned} \tag{17}$$

Therefore,

1. If $x < x_{\text{cut}} - 2^{-n+2}$. We want to know if there exists an integer A between 2^{n-1} and $2^n - 1$ such that

$$\left| Cx - \frac{2A+1}{2^n} \right| \le 2^{-2(n+g)+1}. \tag{18}$$

This would imply that

$$|2CX - 2A - 1| \le 2^{-n-2g+1},$$

where $X = 2^{n-1}x$. Since $X < 2^n$, $2X$ will always be strictly less than 2^{n+2g-1} as soon as $g \ge 1$. Hence, if $g \ge 1$, then (18) implies

$$\left| 2C - \frac{2A+1}{X} \right| < \frac{1}{2X^2}.$$

Hence, it follows from Theorem 2 that $(2A+1)/X$ is a convergent of $2C$. To find if such A and X do exist, we proceed very similarly to Method 2. A convergent (p/q) of $2C$ (with $\gcd(p,q) = 1$) is a candidate if there exist $X = mq$ and $2A + 1 = mp$ such that

$$\begin{aligned} 2^{n-1} + 1 &\le X \le X_{\text{cut}}, \\ 2^{n-1} \quad &\le A \le 2^n - 1, \end{aligned}$$

and if

$$|2Cq - p| \le \frac{1}{2^{n+2g-1} m^*},$$

where $m^* = \lceil 2^{n-1}/q \rceil$ is the smallest possible value of m.

2. If $x > x_{\text{cut}} + 2^{-n+2}$. We want to know if there exists an integer A between 2^{n-1} and $2^n - 1$ such that

$$\left| Cx - \frac{2A+1}{2^{n-1}} \right| \le 2^{-2(n+g)+1}. \tag{19}$$

This would imply that

$$|CX - 2A - 1| \le 2^{-n-2g}.$$

Since $X < 2^n$, $2X$ will always be strictly less than 2^{n+2g} as soon as $g \ge 1$. Hence, if $g \ge 1$, then (19) implies

$$\left| C - \frac{2A+1}{X} \right| < \frac{1}{2X^2}.$$

Hence, it follows from Theorem 2 that $(2A+1)/X$ is a convergent of C. Again, to find if such A and X do exist, we proceed very similarly to Method 2. A convergent (p/q) of C (with $\gcd(p,q) = 1$) is a candidate if there exist $X = mq$ and $2A + 1 = mp$ such that

$$\begin{aligned} X_{\text{cut}} &< X \le 2^n - 1, \\ 2^{n-1} &\le A \le 2^n - 1, \end{aligned}$$

and if

$$|Cq - p| \le \frac{1}{2^{n+2g} m^*},$$

where $m^* = \lceil X_{\text{cut}}/q \rceil$ is the smallest possible value of m.

Method 4 consists of checking whether Algorithm 2 works with the values of X obtained from the convergents of C and $2C$. Method 4 is very similar to Method 2. Nevertheless, Method 4 is always able to conclude: Either it gives all counterexamples to Algorithm 2 or it proves that Algorithm 2 always works with the chosen values of C, n, and g.

A Maple program implementing this method is given in the appendix. Using this program, we are able to conclude that, in the case $n = 53$ and $g = 11$ (which corresponds to the very useful case of double precision as the "target" format and double-extended precision as the "internal" format), Algorithm 2 always returns a correctly rounded product when used with constants C equal to any power of 2 times π, $1/\pi$, $\ln(2)$, $1/\ln(2)$, $\ln(10)$, $1/\ln(10)$, and

$\cos(\pi/8)$. In practice, to find counterexamples for these values of n and g, one must build them, for instance, using the method given in Section 4.

10 Conclusion

The four methods we have proposed allow one to check whether correctly rounded multiplication by an "infinite precision" constant C is feasible at a low cost (one multiplication and one FMA). For instance, in double precision arithmetic, we can multiply by π or $\ln(2)$ with correct rounding. When the multiplication algorithm does not work, Method 3 returns all counterexamples. Notice that, when the calculations are performed in an internal format with at least one more fraction bit than the target format, we have an even simpler method (Method 4) that also gives all counterexamples. Interestingly enough, although it is always possible to build ad hoc values of C for which Algorithm 1 fails, for "general" values of C, our experiments show that Algorithm 1 works for most values of n.

References

[1] M. Abramowitz and I.A. Stegun, *Handbook of Mathematical Functions with Formulas, Graphs and Mathematical Tables,* Applied Math. Series 55, US Nat'l Bureau of Standards, 1964.

[2] N. Brisebarre, J.-M. Muller, and S. Raina, "Accelerating Correctly Rounded Floating-Point Division When the Divisor Is Known in Advance," *IEEE Trans. Computers,* vol. 53, no. 8, pp. 1069-1072, Aug. 2004.

[3] M.A. Cornea-Hasegan, R.A. Golliver, and P. Markstein, "Correctness Proofs Outline for Newton-Raphson Based Floating-Point Divide and Square Root Algorithms," *Proc. 14th IEEE Symp. Computer Arithmetic,* I. Koren and P. Kornerup, eds., pp. 96-105, Apr. 1999.

[4] B.P. Flannery, W.H. Press, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes in C,* second ed. Cambridge Univ. Press, 1992.

[5] D. Goldberg, "What Every Computer Scientist Should Know about Floating-Point Arithmetic," *ACM Computing Surveys,* vol. 23, no. 1, pp. 5-47, Mar. 1991.

[6] G.H. Hardy and E.M. Wright, *An Introduction to the Theory of Numbers.* Oxford Univ. Press, 1979.

[7] J. Harrison, "A Machine-Checked Theory of Floating-Point Arithmetic," *Proc. 12th Int'l Conf. Theorem Proving in Higher Order Logics,* Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, eds., pp. 113-130, Sept. 1999.

[8] W. Kahan, "A Logarithm Too Clever by Half," http://http.cs.berkeley.edu/ wkahan/LOG10HAF.TXT, 2004.

[9] A.Y. Khinchin, *Continued Fractions.* Dover, 1997.

[10] V. Lefèvre, "An Algorithm that Computes a Lower Bound on the Distance between a Segment and Z^2," *Developments in Reliable Computing,* pp. 203-212, Kluwer Academic, 1999.

[11] R.-C. Li, S. Boldo, and M. Daumas, "Theorems on Efficient Argument Reductions," *Proc. 16th IEEE Symp. Computer Arithmetic,* 2003.

[12] P. Markstein, *IA-64 and Elementary Functions: Speed and Precision,* Hewlett-Packard Professional Books. Prentice Hall, 2000.

[13] P.W. Markstein, "Computation of Elementary Functions on the IBM Risc System/6000 Processor," *IBM J. Research and Development,* vol. 34, no. 1, pp. 111-119, Jan. 1990.

[14] J.-M. Muller, "On the Definition of $\mathrm{ulp}(x)$," Technical Report 2005-09, LIP Laboratory, ENS Lyon, ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR2005/RR2005-09.pdf, 2005.

[15] M.A. Overton, *Numerical Computing with IEEE Floating-Point Arithmetic.* SIAM, 2001.

[16] O. Perron, *Die Lehre von den Kettenbrüchen, 3. verb. und erweiterte Aufl.,* pp. 1954-1957. Teubner,

[17] H.M. Stark, *An Introduction to Number Theory.* MIT Press, 1981.

Nicolas Brisebarre received the PhD degree in pure mathematics from the Université Bordeaux I, France, in 1998. He has been a *maître de conférences* (associate professor) in pure mathematics at the Laboratoire d'Arithmétique et d'Algèbre (LArAl), Université de Saint-Étienne, France, since 1999. His research interests are in computer arithmetic and number theory.

Jean-Michel Muller received the PhD degree from the Institut National Polytechnique de Grenoble in 1985. He is the *directeur de recherches* (senior researcher) at CNRS, France, and he is the former head of the LIP Laboratory (LIP is a joint laboratory of CNRS, the Ecole Normale Supérieure de Lyon, INRIA, and the Université Claude Bernard Lyon 1). His research interests are in computer arithmetic. He was the coprogram chair of the 13th IEEE Symposium on Computer Arithmetic (1997) and the general chair of the 14th IEEE Symposium on Computer Arithmetic (1999). He is the author of several books, including *Elementary Functions, Algorithms and Implementation* (second edition, Birkhäuser, 2006). He served as an associate editor of the *IEEE Transactions on Computers* from 1996 to 2000. He is a senior member of the IEEE and a member of the IEEE Computer Society.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.