Correctly rounded multiplication by arbitrary precision constants
Jean-Michel Muller, Nicolas Brisebarre

To cite this version:

HAL Id: ensl-00000010
https://ens-lyon.hal.science/ensl-0000010
Submitted on 13 Apr 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Correctly rounded multiplication by arbitrary precision constants

Nicolas Brisebarre
Laboratoire LARAL, Université Jean Monnet (Saint-Étienne) and Laboratoire LIP
Jean-Michel Muller
CNRS, Laboratoire LIP (CNRS/ENS Lyon/INRIA/Univ. Lyon 1), Projet Arénaire, 46 allée d’Italie, 69364 Lyon Cedex 07, FRANCE
Nicolas.Brisebarre@ens-lyon.fr, Jean-Michel.Muller@ens-lyon.fr

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. We give 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, our methods give the values \(x\) for which it does not.

Introduction

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

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 practice one usually defines a constant \(Ch\), equal to the FP number that is closest to \(C\), and actually computes \(Chx\) (i.e., what is returned is \(\circ(Chx)\)). The obtained result is frequently different from \(\circ(Cx)\) (see Section 1 for some statistics).

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 occur), and at a low cost (i.e., using a very few arithmetic operations only). To do that, we will use fused multiply and add instructions. The fused multiply and add instruction (FMA for short) is available on some current processors such as the IBM Power PC or the Intel/HP Itanium. It evaluates an expression \(ax + b\) with one final rounding error only. This makes it possible to perform correctly rounded division using Newton-Raphson division [9, 3, 8]. Also, this makes evaluation of scalar products and polynomials faster and, generally, more accurate than with conventional (addition and multiplication) floating-point operations.

1 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 \(\circ(Chx)\) and \(\circ(Cx)\) for all possible values of the mantissa of \(x\). The obtained results are given in Table 1, for \(C = \pi\). They show that the “naive” method that consists in computing \(\circ(Chx)\) often returns an incorrectly rounded result (in around 41% of the cases for \(n = 7\)).

2 The algorithm

We want to compute \(Cx\) with correct rounding (assuming rounding to nearest even), where \(C\) is a constant (i.e., \(C\) is known at compile time). \(C\) is not an FP number (otherwise the problem would be straightforward). We assume that a 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 pre-computed:

\[
\begin{align*}
Ch &= \circ(C), \\
Cx &= \circ(C - Ch),
\end{align*}
\]

(1)

where \(\circ(t)\) stands for \(t\) rounded to the nearest FP number.
In the sequel of the paper, we analyze the behavior of the following algorithm. We aim at being able to know for which values of $C$ and $n$ it will 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 a FMA). From $x$, compute

\[
\begin{align*}
   u_1 &= o(C_1 x), \\
   u_2 &= o(C_h x + u_1). 
\end{align*}
\]

The result to be returned is $u_2$.

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 occur) 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 (i.e., $C_h = 0$), 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 $u_2 = o(C x)$. 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 4, we give three methods. The first two ones either certify that Algorithm 1 always returns a correctly rounded result, or give a “bad case” (i.e., a number $x$ for which $u_2 \neq o(C x)$), or are not able 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, that bounds the maximum possible distance between $u_2$ and $C x$ in Algorithm 1. Of course, Algorithm 1 works for a given constant $C$ and precision $n$ if all floating-point values of $x$ are such that $|u_2 - C x| < 1/2 \text{ulp} (u_2)$. It is worth being noticed that without the use of a FMA instruction (that is, if Algorithm 1 was executed using ordinary FMUL and FADD), except for a few very simple values of the constant $C$ – e.g., powers of 2 –, Algorithm 1 would fail to return a correctly rounded result for all values of $x$.

### Property 1

Define $x_{\text{cut}} = 2/C$ and

\[
\epsilon_1 = |C - (C_h + C_f)|
\]

- If $x < x_{\text{cut}}$ then $|u_2 - C x| < 1/2 \text{ulp} (u_2) + \eta$,
- If $x \geq x_{\text{cut}}$ then $|u_2 - C x| < 1/2 \text{ulp} (u_2) + \eta'$,

where

\[
\begin{align*}
   \eta &= \frac{1}{2} \text{ulp} (C_{x_{\text{cut}}}) + \epsilon_1 x_{\text{cut}}, \\
   \eta' &= \text{ulp} (C_f) + 2\epsilon_1.
\end{align*}
\]

**Proof.**

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

\[
|\epsilon_1| \leq \frac{1}{2} \text{ulp} (C - C_h) \leq 2^{-2n - 1}.
\]

Now, we have,

\[
|u_2 - C x| \leq |u_2 - (C_h x + u_1)| + |(C_h x + u_1) - (C_h x + C_f x)| + |C_h + C_f x - C x| \leq \frac{1}{2} \text{ulp} (u_2) + |u_1 - C_h x| + \epsilon_1 |x|
\]

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

If $|u_2 - C x|$ is less than $1/2 \text{ulp} (u_2)$, then $u_2$ is the FP number that is closest to $C x$. Hence our problem is to know if $C x$ can be at a distance larger than or equal to $1/2 \text{ulp} (u_2)$ from $u_2$. From (4), this would imply that $C x$ would be at a distance less than $1/2 \text{ulp} (C_{x_{\text{cut}}}) + \epsilon_1 |x| < 2^{-2n + 1}$ from the midpoint of two consecutive FP numbers (see Figure 1). If $x < x_{\text{cut}}$ then $C x < 2$, then the midpoint of two consecutive FP numbers around $C x$ is of the form $A/2^n$, where $A$ is an odd integer between $2^n + 1$ and $2^{n+1} - 1$. If $x \geq x_{\text{cut}}$, then the midpoint of two consecutive FP numbers around $C x$ is of the form $A/2^{n-1}$. For the sake of clarity of the proofs we assume that $x_{\text{cut}}$ is not an FP number (if $x_{\text{cut}}$ is an FP number, it suffices to separately check Algorithm 1 with $x = x_{\text{cut}}$).

<table>
<thead>
<tr>
<th>$n$</th>
<th>Proportion of correctly rounded results</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.93750</td>
</tr>
<tr>
<td>6</td>
<td>0.78125</td>
</tr>
<tr>
<td>7</td>
<td>0.59375</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>16</td>
<td>0.86765</td>
</tr>
<tr>
<td>17</td>
<td>0.73558</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>24</td>
<td>0.66805</td>
</tr>
</tbody>
</table>

Table 1. Proportion of input values $x$ for which $o(C_h x) = o(C x)$ for $C = \pi$ and various values of the number $n$ of mantissa bits.
3 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 [5, 11, 10, 6].

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

\[
\begin{align*}
    r_0 &= \alpha, \\
    a_i &= \lfloor r_i \rfloor, \\
    r_{i+1} &= \frac{1}{r_i - a_i}.
\end{align*}
\]

If \( \alpha \) is irrational, then these sequences are defined for any \( i \) (i.e., \( r_i \) is never equal to \( a_i \)), and the rational number

\[
\frac{p_i}{q_i} = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \cdots + \frac{1}{a_i}}}}
\]

is called the \( i \)th convergent to \( \alpha \). If \( \alpha \) is rational, then these sequences finish for some \( k \), and \( p_k/q_k = \alpha \) exactly. The \( p_i \)'s and the \( q_i \)'s can be deduced from the \( a_i \) using the following recurrences,

\[
\begin{align*}
    p_0 &= a_0, \\
    p_1 &= a_1a_0 + 1, \\
    p_n &= p_{n-1}a_n + p_{n-2}, \\
    q_0 &= 1, \\
    q_n &= q_{n-1}a_n + q_{n-2}.
\end{align*}
\]

The major interest of the continued fractions lies in the fact that \( p_i/q_i \) is the best rational approximation to \( \alpha \) among all rational numbers of denominator less than or equal to \( q_i \).

We will use the following two results [5]

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

\[
|p - \alpha q| \geq |p_n - \alpha q_n|.
\]

**Theorem 2** Let \( p, q \) be nonzero integers, with \( \gcd(p, q) = 1 \). If

\[
\left| \frac{p}{q} - \alpha \right| < \frac{1}{2q^2}
\]

then \( p/q \) is a convergent of \( \alpha \).

4 Three methods for analyzing Algorithm 1

4.1 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 \( X_{\text{cut}} \) are integers between \( 2^{n-1} + 1 \) and \( 2^n - 1 \). We separate the cases \( x < x_{\text{cut}} \) and \( x > x_{\text{cut}} \).

4.1.1 If \( x < x_{\text{cut}} \)

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

\[
|Cx - A| < \frac{\eta}{2^n} \tag{6}
\]

where \( \eta \) is defined in Property 1. (6) is equivalent to

\[
|2CX - A| < 2^n \eta \tag{7}
\]

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 \( 0 < X \leq X_{\text{cut}}, |2CX - A| \geq \delta \). Therefore

1. if \( \delta \geq 2^n \eta \) then \( |Cx - A|/2^n \eta \) 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 \( y = q_k2^{-n+1} \).

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.

\(^1\)It is possible that \( y \) not be between 1 and \( x_{\text{cut}} \). It will anyway be a counterexample, i.e., an \( n \)-bit number for which Algorithm 1 fails.
4.2.1 If \( x > x_{\text{cut}} \)

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

\[
|Cx - \frac{A}{2^{n-1}}| < \eta'
\]

where \( \eta' \) is defined in Property 1. (8) is equivalent to

\[
|CX - A| < 2^{-n} \eta'
\]

Define \((q'_i/\ell'_i)_{i \geq 1}\) as the convergents of \( C \). Let \( k' \) be the smallest integer such that \( q'_{k'+1} \geq 2^n \), and define \( \delta' = |q'_k - Cq'_k| \). Theorem 1 implies that for any \( A, X \in \mathbb{Z} \), with \( X_{\text{cut}} \leq X < 2^n \), \( |CX - A| \geq \delta' \). Therefore

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

4.2 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 \( x > x_{\text{cut}} \).

4.2.1 If \( x > x_{\text{cut}} \)

If 

\[
|Cx - \frac{A}{2^{n-1}}| < \epsilon_1 x + \frac{1}{2} \text{ulp}(Cx)
\]

then,

\[
|C - \frac{A}{X}| < \epsilon_1 + \frac{2^{n-2}}{X} \text{ulp}(Cx).
\]

Now, if

\[
2^{2n+1} \epsilon_1 + 2^{2n-1} \text{ulp}(2Cx) \leq 1,
\]

then for any \( X < 2^n \) (i.e., \( x < 2^n \)),

\[
\epsilon_1 + \frac{2^{n-2}}{X} \text{ulp}(Cx) < \frac{1}{2X^2}.
\]

Hence, if (11) is satisfied, then (10) implies (from Theorem 2) that \( A/X \) is a convergent of \( C \). This means that if (11) 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 quickly 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 \( A = mp \) such that

\[
\begin{cases}
X_{\text{cut}} < X \leq 2^n - 1, \\
2^n + 1 \leq A \leq 2^{n+1} - 1, \\
|\frac{CX}{2^n} - \frac{A}{2^{n-1}}| < \epsilon_1 \frac{X}{2^n} + \frac{1}{2} \text{ulp}(Cx).
\end{cases}
\]

This would mean

\[
|\frac{Cmq}{2^{n-1}} - \frac{mp}{2^n}| < \epsilon_1 x_{\text{cut}} + \frac{1}{2} \text{ulp}(Cx_{\text{cut}}),
\]

where \( m^* = \lfloor X_{\text{cut}}/q \rfloor \) is the smallest possible value of \( m \). Hence, if Condition (12) is not satisfied, convergent \( p/q \) cannot generate a bad case for Algorithm 1.

Now, if Condition (12) is satisfied, we have to check Algorithm 1 with all values \( X = mq \), with \( m^* \leq m \leq \lfloor (2^n - 1)/q \rfloor \).
which would imply
\[
\frac{2^n}{m^*} \left( |2Cq - p| + \frac{1}{2} \ulp(C \times x_{cut}) \right), \tag{14}
\]
where \(m^* = \left\lceil \frac{2^n - 1}{q} \right\rceil\) is the smallest possible value of \(m\). Hence, if \((14)\) is not satisfied, convergent \(p/q\) cannot generate a bad case for Algorithm 1.

Now, if \((14)\) is satisfied, we have to check Algorithm 1 with all values \(X = mq\), with \(m^* \leq m \leq \left\lfloor X_{cut} / q \right\rfloor\).

This last result and \((4)\) make it possible to deduce:

**Theorem 3 (Conditions on \(C\) and \(n\))** Assume \(1 < C < 2\). Let \(x_{cut} = 2/C\), and \(X_{cut} = \left\lceil 2^n x_{cut} \right\rceil\).

- If \(X = 2^n x > X_{cut}\) and \(2^{n+1} + 2^{n-1} \ulp(2C) \leq 1\) 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 \(|Cq - p| < \epsilon_x + \frac{1}{2} \ulp(C)\);
- if \(X = 2^n x \leq X_{cut}\) and \(\epsilon_x x_{cut} + 1/2 \ulp(C \times x_{cut}) \leq 1/(2^{n+1} X_{cut})\) 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 \(2Cq - p| < \frac{1}{2^n Q} (\epsilon_x x_{cut} + 1/2) \ulp(C \times x_{cut})\).

**4.3 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 \(\ulp(C_t) \leq 2^{-2n}\).

**4.3.1 If \(x < x_{cut}\)**

if \(\ulp(C_t) \leq 2^{-2n-2}\) then we have
\[
\left| u_x - Cx \right| < \frac{1}{2} \ulp(u_x) + 2^{-2n-1}.
\]
For any integer \(A\), the inequality
\[
\left| Cx - \frac{2A + 1}{2^n} \right| \leq \frac{1}{2^{n+1}}
\]
implies
\[
|2CX - 2A - 1| \leq \frac{1}{2^{n+1}} < \frac{1}{2X}.
\]
\((2A + 1)/X\) is a convergent of \(2C\) from Theorem 2. It suffices then to check (as in Method 2) the convergents of \(2C\) of denominator less or equal to \(X_{cut}\).

Now, assume \(\ulp(C_t) \geq 2^{-2n-1}\). We have,
\[
-\ulp(C_t) + \frac{X}{2^n-1} \leq u_1 \leq \ulp(C_t) + \frac{X}{2^{n-1}}
\]
i.e.,
\[
-2^{2n} \ulp(C_t) + 2^{n+1} C_t X \leq u_1 2^{2n}
\]
\[
\leq 2^{2n} \ulp(C_t) + 2^{n+1} C_t X.
\]
We look for the integers \(X, 2^n-1 \leq X \leq X_{cut}\), such that there exists an integer \(A, 2^n-1 \leq A \leq 2^n-1\), with
\[
\left| C_h \frac{X}{2^n-1} + u_1 - \frac{2A + 1}{2^n}\right| < 2 \ulp(C_t)
\]
i.e.,
\[
\left| C_h \frac{X}{2^n \ulp(C_t)} + u_1 \left( \frac{2A + 1}{2^n \ulp(C_t)} \right) \right| < 1.
\]
Since \(u_1 / (2 \ulp(C_t))\) is half an integer and \(\frac{C_h X}{2^n \ulp(C_t)}\) and \(\frac{2A + 1}{2^n \ulp(C_t)}\) are integers, we have
\[
\frac{C_h X}{2^n \ulp(C_t)} + \frac{u_1}{2 \ulp(C_t)} - \frac{2A + 1}{2^n \ulp(C_t)} = 0, \pm 1/2.
\]
Then, combining these three equations with inequalities \((15)\), we get the following three pairs of inequalities
\[
0 \leq 2X(C_h + C_t) - (2A + 1) + 2^n \ulp(C_t) \leq 2^{n+1} \ulp(C_t),
\]
\[
0 \leq 2X(C_h + C_t) - (2A + 1) \leq 2^{n+1} \ulp(C_t),
\]
\[
0 \leq 2X(C_h + C_t) - (2A + 1) + 2^{n+1} \ulp(C_t) \leq 2^{n+1} \ulp(C_t).
\]
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
\[
\{2X(C_h + C_t) + 2^n \ulp(C_t)\} \leq 2^{n+1} \ulp(C_t),
\]
\[
\{2X(C_h + C_t)\} \leq 2^{n+1} \ulp(C_t),
\]
\[
\{2X(C_h + C_t) + 2^{n+1} \ulp(C_t)\} \leq 2^{n+1} \ulp(C_t).
\]
We use an efficient algorithm due to V. Lefèvre [7] to determine the integers \(X\) solution of each inequality.
4.3.2 If \( x > x_{\text{cut}} \) if \( \text{ulp} (C_t) \leq 2^{-2n-1} \) then we have
\[
|u_2 - CX| < \frac{1}{2} \text{ulp} (u_2) + 2^{-2n}.
\]
Therefore, for any integer \( A \), the inequality
\[
|CX - 2A + 1| \leq \frac{1}{2^{2n}}
\]
is equivalent to
\[
|CX - 2A - 1| \leq \frac{1}{2^{2n+1}} < \frac{1}{2X},
\]
(\( 2A+1 \))/\( X \) 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 or equal to \( 2^n - 1 \).

Now, assume \( \text{ulp} (C_t) = 2^{-2n} \). We look for the integers \( X \), \( X_{\text{cut}} + 1 \leq X \leq 2^n - 1 \), such that there exists an integer \( A \), \( 2^n - 1 \leq A \leq 2^n - 1 \), with
\[
\left| Ch \frac{X}{2^n-1} + u_1 - \frac{2A + 1}{2^n-1} \right| < \frac{1}{2^{2n}}
\]
i.e.,
\[
\left| 2^{n+1}ChX + u_12^{2n} - 2^{n+1} (2A + 1) \right| < 1.
\]
Since \( u_12^{2n} \), \( 2^{n+1}ChX \) and \( 2^{n+1}(2A + 1) \in \mathbb{Z} \), we have
\[
2^{n+1}ChX + u_12^{2n} - 2^{n}(2A + 1) = 0.
\]
Then, combining this equation with inequalities (15), we get the inequalities
\[
0 \leq X(Ch + Ct) - (2A + 1) + \frac{1}{2^{n+1}} \leq \frac{1}{2^n},
\]
that is to say
\[
\{ X(Ch + Ct) + \frac{1}{2^{n+1}} \} \leq \frac{1}{2^n}.
\]
Here again, we use Lefèvre’s algorithm [7] to determine the integers \( X \) solution of this inequality.

5 Examples

5.1 Example 1: multiplication by \( \pi \) in double precision

Consider the case \( C = \pi /2 \) (which corresponds to multiplication by any number of the form \( 2^{k\pi /2} \)), and \( n = 53 \) (double precision), and assume we use Method 1. We find:
\[
\begin{align*}
C_h &= 884279719003555/562949953421321, \\
C_t &= 6.123233996 \ldots \times 10^{-17}, \\
\epsilon_1 &= 1.497384905 \ldots \times 10^{-33}, \\
x_{\text{cut}} &= 1.2732395447351626862 \ldots, \\
\text{ulp} (C_t x_{\text{cut}}) &= 2^{-106}, \\
\text{ulp} (C_t) &= 2^{-106}.
\end{align*}
\]

Hence,
\[
\begin{align*}
2^n \eta &= 7.268364390 \times 10^{-17}, \\
2^{n-1} \eta' &= 6.899839541 \times 10^{-17}.
\end{align*}
\]

Computing the convergents of \( 2C \) and \( C \) we find
\[
\begin{align*}
\frac{p_k}{q_k} &= 6134899525417045, \\
\frac{p_k'}{q_k'} &= 7647888557167847,
\end{align*}
\]
and \( \delta = 9.495905771 \times 10^{-17} > 2^n \eta \) (which means that Algorithm 1 works for \( x < x_{\text{cut}} \)), and
\[
\begin{align*}
\frac{p_k}{q_k} &= 12055686754159438, \\
\frac{p_k'}{q_k'} &= 7647888557167847,
\end{align*}
\]
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:

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

Hence, in that case, multiplying by \( \pi \) with correct rounding only requires 2 consecutive FMAs.

5.2 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 \( 2^{k\ln(2)} \)), and \( n = 53 \), and assume we use Method 2. We find:
\[
\begin{align*}
C_h &= 6243314708165559, \\
C_t &= 4.638093628 \ldots \times 10^{-17}, \\
x_{\text{cut}} &= 1.44269 \ldots, \\
\epsilon_1 &= 1.141541688 \ldots \times 10^{-33}, \\
\epsilon_1 x_{\text{cut}} + \frac{1}{2} \text{ulp} (C_t x_{\text{cut}}) &= 7.89099 \ldots \times 10^{-33}, \\
1/(2^{n+1}X_{\text{cut}}) &= 8.5437 \ldots \times 10^{-33}.
\end{align*}
\]
Since \( \epsilon_1 x_{\text{cut}} + \frac{1}{2} \text{ulp} (C_t x_{\text{cut}}) \leq 1/(2^{n+1}X_{\text{cut}}) \), to find the possible bad cases for Algorithm 1 that are less than \( x_{\text{cut}} \), it suffices to check the convergents of \( 2C \) of denominator less than or equal to \( X_{\text{cut}} \). These convergents are:
\[
\begin{align*}
2, 3, 11/4, 25/9, 36/13, 61/22, 2731/985, 25469/9186, 1097898/395983, 1123667/405169, \\
22221265/801152, 16672222/6013233, 18893487/6814385, 35657097/12827618, 125596614/49297239, \\
161156323/58124857, 609095893/219671810, 1379275497/497468477, 1998355702/717140287, \\
3355845633/1931749051, 7344280705/264889938, 2738879748/987842065, 3473306843/12527306403, \\
622185860122/240575234689, 96854924654/34933029871, 449541558417/162137842952, \\
27941204253556/1007760875893, 3243868083737/116987930551, 603775061929/217758018118, \\
39470146779947/1423854039243, 124448186801770/4885196153847, 163913834781717/59121042175090, \\
288366523383847/104006238310937, 6219615325834944/2243252046704767.
\end{align*}
\]
None of them satisfies condition (14). Therefore there are no bad cases less than $x_{\text{cut}}$. Processing the case $x > x_{\text{cut}}$ is similar and gives the same result, hence:

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 no under/overflow occurs.

5.3 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:

\[
\begin{align*}
C_h &= \frac{5774161139222559}{4503599627370496}, \\
C_t &= -7.871470670\ldots \times 10^{-17}, \\
\epsilon_1 &= 4.28857513\ldots \times 10^{-33}, \\
x_{\text{cut}} &= 1.570796\ldots, \\
C_l x_{\text{cut}} &= -1.236447722\ldots \times 10^{-16}, \\
\text{ulp}(C_l x_{\text{cut}}) &= 2^{-105}, \\
2^n\eta &= 1.710990939\ldots \times 10^{-16}, \\
p_k/q_k &= \frac{75486032237905811}{7081371451248382}, \\
\delta &= 7.66955467\ldots \times 10^{-17}.
\end{align*}
\]

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 $X = 6081371451248382$, i.e., $6081371451248382 \times 2^{\pm k}$ are the only FP values for which Algorithm 1 fails.

5.4 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:

\[
\begin{align*}
C_h &= \frac{11863283/11863283}{8388608}, \\
C_t &= 2.40323497\ldots \times 10^{-8}, \\
\epsilon_1 &= 7.628067479\ldots \times 10^{-16}, \\
x_{\text{cut}} &= 11863283, \\
\text{ulp}(C_l x_{\text{cut}}) &= 2^{-48}, \\
2^n\eta &= 4.790110735\ldots \times 10^{-8}, \\
p_k/q_k &= 22619537/7997214, \\
\delta &= 2.210478490\ldots \times 10^{-8}, \\
p_k'/q_k' &= 22619537/15094428, \\
\delta' &= 2.210478490\ldots \times 10^{-8}.
\end{align*}
\]

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 $X = q_k' = 15094428$ 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. And yet, Method 3 or exhaustive testing (which is possible since $n = 24$ is reasonably small) show that Algorithm 1 always works.

6 Implementation and results

As the reader will have guessed from the previous examples, using our Methods by paper and pencil calculation is fastidious and error-prone. We have written Maple programs that implement Methods 1 and 2, and a GP/PARI\textsuperscript{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 can be downloaded from the url

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 thanks to Method 1 that 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 in advance a result such as Theorem 4, for a general constant $C$. And yet, this might make method 3 difficult to implement in a compiler, to decide at compile-time if we can use our algorithm.

7 Conclusion

The three 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 $\pi$ or $\ln(2)$ with correct rounding. 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 $n$.

\textsuperscript{2}http://pari.math.u-bordeaux.fr/
<table>
<thead>
<tr>
<th>$C$</th>
<th>$n$</th>
<th>method 1</th>
<th>method 2</th>
<th>method 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi$</td>
<td>8</td>
<td>Does not work for 226</td>
<td>Does not work for 226</td>
<td>AW (c)</td>
</tr>
<tr>
<td>$\pi$</td>
<td>24</td>
<td>unable</td>
<td>unable</td>
<td>AW</td>
</tr>
<tr>
<td>$\pi$</td>
<td>53</td>
<td>AW</td>
<td>unable</td>
<td>AW</td>
</tr>
<tr>
<td>$\pi$</td>
<td>64</td>
<td>unable</td>
<td>AW</td>
<td>AW (c)</td>
</tr>
<tr>
<td>$\pi$</td>
<td>113</td>
<td>AW</td>
<td>AW</td>
<td>AW (c)</td>
</tr>
<tr>
<td>$1/\pi$</td>
<td>24</td>
<td>unable</td>
<td>unable</td>
<td>AW</td>
</tr>
<tr>
<td>$1/\pi$</td>
<td>53</td>
<td>Does not work for 6081371451248382</td>
<td>unable</td>
<td>AW unless $X = 6081371451248382$</td>
</tr>
<tr>
<td>$1/\pi$</td>
<td>64</td>
<td>AW</td>
<td>AW</td>
<td>AW (c)</td>
</tr>
<tr>
<td>$1/\pi$</td>
<td>113</td>
<td>unable</td>
<td>unable</td>
<td>AW</td>
</tr>
<tr>
<td>$\ln 2$</td>
<td>24</td>
<td>AW</td>
<td>AW</td>
<td>AW (c)</td>
</tr>
<tr>
<td>$\ln 2$</td>
<td>53</td>
<td>AW</td>
<td>unable</td>
<td>AW (c)</td>
</tr>
<tr>
<td>$\ln 2$</td>
<td>64</td>
<td>AW</td>
<td>unable</td>
<td>AW (c)</td>
</tr>
<tr>
<td>$\ln 2$</td>
<td>113</td>
<td>AW</td>
<td>AW</td>
<td>AW (c)</td>
</tr>
<tr>
<td>$\ln 10$</td>
<td>24</td>
<td>unable</td>
<td>AW</td>
<td>AW (c)</td>
</tr>
<tr>
<td>$\ln 10$</td>
<td>53</td>
<td>unable</td>
<td>AW</td>
<td>AW (c)</td>
</tr>
<tr>
<td>$\ln 10$</td>
<td>64</td>
<td>unable</td>
<td>AW</td>
<td>AW (c)</td>
</tr>
<tr>
<td>$\ln 10$</td>
<td>113</td>
<td>AW</td>
<td>AW</td>
<td>AW (c)</td>
</tr>
<tr>
<td>$1/\ln 10$</td>
<td>24</td>
<td>unable</td>
<td>unable</td>
<td>AW</td>
</tr>
<tr>
<td>$1/\ln 10$</td>
<td>53</td>
<td>unable</td>
<td>AW</td>
<td>AW (c)</td>
</tr>
<tr>
<td>$1/\ln 10$</td>
<td>64</td>
<td>unable</td>
<td>AW</td>
<td>AW (c)</td>
</tr>
<tr>
<td>$1/\ln 10$</td>
<td>113</td>
<td>unable</td>
<td>unable</td>
<td>AW</td>
</tr>
<tr>
<td>$\cos \frac{\pi}{2}$</td>
<td>24</td>
<td>unable</td>
<td>unable</td>
<td>AW</td>
</tr>
<tr>
<td>$\cos \frac{\pi}{2}$</td>
<td>53</td>
<td>AW</td>
<td>AW</td>
<td>AW (c)</td>
</tr>
<tr>
<td>$\cos \frac{\pi}{2}$</td>
<td>64</td>
<td>AW</td>
<td>unable</td>
<td>AW</td>
</tr>
<tr>
<td>$\cos \frac{\pi}{2}$</td>
<td>113</td>
<td>unable</td>
<td>AW</td>
<td>AW (c)</td>
</tr>
</tbody>
</table>

Table 2. Some results obtained using methods 1, 2 and 3. 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.

References


