A Correctly Rounded
Mixed-Radix Fused-Multiply-Add

Clothilde Jeangoudoux
Safran Electronics & Defense
21, avenue du Gros Chêne
95610 Eragny-sur-Oise
Email: clothilde.jeangoudoux@safrangroup.com

Christoph Lauter
Sorbonne Université, CNRS,
Laboratoire d'Informatique de Paris 6, LIP6,
F-75005 Paris, France
Email: christoph.lauter@lip6.fr

***Abstract*—The IEEE 754-2008 Standard governs Floating-Point Arithmetic in all types of Computer Systems. The Standard provides for two radices, 2 and 10. It specifies conversion operations between these radices, but does not allow floating-point formats of different radices to be mixed in computational operations. In contrast, the Standard does provide for mixing formats of one radix in one operation.**

In order to enhance the Standard and make it closed under all basic computational operations, we propose an algorithm for a correctly rounded mixed-radix Fused-Multiply-and-Add (FMA). Our algorithm takes any combination of IEEE754 binary64 and decimal64 numbers in argument and provides a result in IEEE754 binary64 and decimal64, rounded according to any for the five IEEE754 rounding modes.

Our implementation does not require any dynamic memory allocation; its runtime can be bounded statically. We compare our implementation to a basic mixed-radix FMA implementation based on the GMP Multiple Precision library.

I. Introduction

When developing software using floating-point arithmetic, the user is presented with a choice between two radices: binary floating-point arithmetic (radix-2) or decimal floating-point arithmetic (radix-10). The IEEE 754-2008 Standard for Floating-Point Arithmetic [1] defines the format and operations for both radices, and states that *a programming environment may conform to this standard in one radix or in both*. Quite a few studies have been undertaken on decimal floating-point arithmetic in an effort to provide accurate and efficient decimal operations using binary formats [2] and conversion operations [3]. Those algorithms provide faithfully (or comparably accurate) rounded results. We would like to go one step further and perform correctly rounded "mixed-radix" floating-point arithmetic, that is, arithmetic operations with operands and result represented in a binary or a decimal floating-point format.

To perform mixed-radix operations, the user may be tempted to combine binary and decimal floating-point arithmetic such as in the following C code:

```
double a=2000.0, c;
_Decimal64 b=0.001D;
c = a * b;
```

When trying to compile this code e.g. with gcc 6.3.0, the compiler refuses to translate the code, emitting an error message which states that floating-point types of different radices cannot be mixed. The user, who might have no other choice than using and mixing two scientific libraries, one of which produces results in binary floating-point and the other produces decimal ones, might hence be inclined to force the compiler to translate this piece of code, e.g. by adding explicit casts (conversions). This attempt may also be misleading, as these conversions induce errors, which may be insignificant but may also accumulate, amplify and become important.

Extending our example code as follows showcases this dangerous [4] behavior:

```
double a=2000.0, c=-2.0, d;
_Decimal64 b=0.001D;
/* d = a * b + c */
d = __builtin_fma(a,(double)b,c);
```

This code compiles, but does not set the correct result to `d`, i.e. $2000 \times 0.001 - 2 = 0$, but $4.16333634 \cdot 10^{-17}$. This error is due to the conversion of the decimal64 number `b` into a binary floating-point number.

All these reasons point toward the need for a study and conception of a specific mixed-radix framework. However, as of now, the applicability of mixed-radix operations has sparsely been studied and very few of them are available to the user. Previous work investigated the feasibility of exact comparison between floating-point formats of different radices [5], as a way to enrich the current floating-point environment and make numerical software safer. We would like to extend this approach to the basic computational operations defined by the IEEE754 Standard: $+, -, \times, /, \sqrt{}$ and FMA. The FMA operation, added in 2008, computes $a \times b + c$ with one, single, correct rounding. We wish to provide algorithms for a mixed-radix implementation of the computational operations, i.e. algorithms that take floating-point numbers of both radices–two and ten–supported by the IEEE754-2008 Standard and compute correctly rounded floating-point results, also in one of these two radices.

A naive approach would be to perform the FMA computation in one of those two radices and perform a rounding of the result into the output format. Suppose we have

$$\begin{aligned} a &= 2^{-55} \cdot 3242591731706757, \\ b &= 2^{-52} \cdot 4548635623644201, \\ c &= 2^{-52} \cdot 5902958103587057. \end{aligned}$$

All these three numbers are IEEE754 binary64 numbers. The binary64 result of the FMA operation is then $\circ(a \times b + c) = 2^{-52} \cdot 5902958103587057$ that rounds to the

XXX-X-XXXXXXX-X-X/ARITH18/©2018 IEEE

decimal64 $10^{-17} \cdot 9090999999999999$, where in a correctly rounded mixed-radix arithmetic the decimal64 result would have been $10^{-17} \cdot 9091000000000000$. It is easy to construct an example for which the mixed-radix result of an operation will be hard to round. Yet we have no simple systematic way to seek inputs for which this will be the case, hence our need for a study and specific implementation of a correctly rounded mixed-radix FMA. Our work is based on a previous study [6], where an attempt to compute such hard-to-round cases was made, without actually succeeding.

There are several binary and decimal floating-point formats available, as defined in the IEEE754-2008 such as binary32, binary64 and decimal64 or decimal128, and in principle even mixed-radix heterogeneous operations like a *binary64 plus decimal128 gives binary32*-addition, might eventually to be considered. To get started, it is however easier to look at one binary and one decimal format, with comparable accuracy, i.e. precisions that are comparable provided appropriate conversion. Hence to perform those algorithms, we choose the IEEE754 binary64 format, which provides $k = 53$ bits of binary precision, which is equivalent to a basic relative error of $u = 2^{-53}$, and the IEEE754 decimal64 that provides $k = 16$ digits of decimal precision, which is equivalent to a basic relative error of $u = 1/2 \times 10^{-15} = 2^{-50.82...}$.

We will suppose that the IEEE754 decimal formats, such as the decimal64 format we use, are encoded in the IEEE754 binary encoding (BID) [3], i.e. that access to a binary representation of the decimal significand is straightforward. Given this setting, let us admit the following:

- Binary, resp. decimal, operands are denoted $2^E \cdot m$, resp. $10^F \cdot n$, with a bounded signed binary exponent $E \in \mathbb{Z}$, resp. decimal exponent $F \in \mathbb{Z}$, and $m \in \mathbb{Z}$, a signed integer significand satisfying $2^{k-1} \leq |m| \leq 2^k - 1$, where k is the binary precision, resp. $n \in \mathbb{Z}$ with $1 \leq |n| \leq 10^l - 1$ and l the decimal precision.
- The result of the computation of the mixed-radix FMA is either a binary output $2^G \cdot p$, with $G \in \mathbb{Z}$ appropriately bounded and $p \in \mathbb{Z}$ a signed integer significand bounded, or a decimal output $10^H \cdot q$ with $H \in \mathbb{Z}$ appropriately bounded and $q \in \mathbb{Z}$ a signed integer significand bounded.

The IEEE 754 Standard defines the notion of “quantum” for decimal operations [1]. However, whenever an operation’s input is binary (floating-point or integer), the quantum is defined in an ad hoc manner. We shall hence not consider it.

Throughout this paper, we will first highlight the challenges due to the computation of correctly rounded mixed-radix operations (Section II). We will then describe the internal mechanisms involved in the computation of a correctly rounded mixed-radix FMA (Section III), and then discuss the implementation and test strategy developed to measure the efficiency of our approach (Section IV).

II. Challenges and Settings

Provided a correctly rounded mixed-radix FMA such as $\circ(a \times b + c)$, we can easily compute the correctly rounded mixed-radix multiplication by setting the c operand to zero, and the addition/subtraction by setting either a or b to one. Furthermore both correctly rounded mixed-radix division and square root can be determined with a single mixed-radix FMA operation, provided it allows for slightly larger precision: given a binary precision k, resp. a decimal precision l, we call *midpoint*, noted f, a number that is exactly halfway between two consecutive floating-point numbers.

The rounding of the result of the mixed-radix division $\circ_k\left(\frac{x}{y}\right)$ of binary precision k, resp. decimal precision l, can be determined with the help of the closest midpoint of the real result $\frac{x}{y}$ [7]. This midpoint f can be represented with a floating-point number of slightly larger precision k', resp. l'. Thus the rounding decision boils down to decide whether

- $\frac{x}{y} < f \Leftrightarrow 0 < y \times f - x$,
- $\frac{x}{y} > f \Leftrightarrow 0 > y \times f - x$,
- or $\frac{x}{y} = f \Leftrightarrow 0 = y \times f - x$.

This implies that identifying the rounding of the division $\frac{x}{y}$ and thus computing the correctly rounded mixed-radix division operation in the output format of precision k, resp. l, is equivalent to compute a mixed-radix FMA operation $y \times f - x$ that allows for operands of slightly larger precision k', resp. l', and compare its result to zero.

Similarly, the computation of the correctly rounded mixed-radix square root of precision k, resp. l, boils down to determine where the result falls according to a midpoint f. Thus to determine if $\sqrt{x} < f$ we can compute $0 < f \times f - x$ with a mixed-radix FMA operation that allows for operands of slightly larger precision k', resp. l'.

For the binary64 format, we can represent all binary floating-point numbers, normal and subnormal, and their midpoints by increasing the binary precision from $k = 53$ to $k' = 55$. The modification of the precision of the binary64 format has an impact on the bounds of its significand and exponent, thus it can be defined as follows

$$\begin{aligned} &2^E \cdot m;\ \text{with } 2^{54} \leq |m| < 2^{55}; \\ &-1130 \leq E \leq 969;\ m, E \in \mathbb{Z}. \end{aligned} \tag{1}$$

Since 10 is divisible by 2, the representation of the midpoints is also possible in the decimal64 format, by adding only one binary bit of precision, for the midpoint between two decimal floating-point numbers can be represented in the same binade instead of decade. This means that we can represent the significand n with the same 55-bit precision as the binary64 format earlier and adapt the exponent, by factorizing 10^F as $2^J \cdot 5^K$, with $K = F$. We can then adapt the decimal64 input representation such as

$$\begin{aligned} &10^F \cdot n = 2^J \cdot 5^K \cdot r;\ \text{with } 2^{54} \leq |r| < 2^{55};\ r \in \mathbb{Z} \\ &-452 \leq J \leq 385;\ -421 \leq K \leq 385;\ J, K \in \mathbb{Z}. \end{aligned} \tag{2}$$

There is however not one mixed-radix FMA operation, but several operations, depending on the combination of formats given as inputs and output. As a matter of fact, the FMA operation handles as inputs the three numbers a, b and c, and returns one output, all four numbers being representable in two different formats: binary64 or decimal64. We can naively consider that we will need to implement $2^4 = 16$ versions of the mixed-radix FMA. Yet as we are studying mixed-radix arithmetic, we can easily discard the all binary and all decimal

FMA operations that will not benefit from our approach. Furthermore we will consider the *decimal64-to-binary64*-multiplication and the *binary64-to-decimal64*-multiplication to be equivalent, whether we want a binary or decimal output. Hence this excludes four more versions of the mixed-radix FMA operation as redundancies. In the end, we will only consider ten different versions of the mixed-radix FMA operation.

The real challenge of mixed-radix operations lies into the computation of the addition or subtraction inside the FMA operation. As floating-point subtraction is just floating-point addition with a change in sign of the subtrahend, we are going to continue the discussion only for "addition", subsuming subtraction. Typically, for a floating-point addition, two cases can be distinguished: the case when the two input operands are sufficiently close in terms of exponent that the subtraction of their aligned significands will lead to cancellation, and the other case when they are far enough in terms of magnitude so that the result's order of magnitude is essentially the one of the bigger of the operands. The first of these two cases is commonly called *near path*, the latter *far path* [8].

Uniform-radix correctly rounded floating-point addition is based, for the near path, on the observation that when cancellation occurs, the floating-point operation becomes naturally exact, i.e. does not require any rounding [8]. This case corresponds to the case when Sterbenz' lemma can be applied [9]. For mixed-radix addition, there is no reason why this cancellation case should become exact: e.g. when both input operands are in decimal floating-point arithmetic, both 10 and -9.9 are representable. There sum of course is 0.1 which clearly is not representable in binary floating-point arithmetic. A decimal plus decimal to binary mixed-radix addition can hence not suppose that no rounding occurs in near path cases. Furthermore, when the two operands to a mixed-radix addition are not in the radix, some sort of conversion of the binary operand to the decimal radix or vice versa will be needed. While cancellation in the near-path makes the operation become exact for uniform-radix cases, the same cancellation in the mixed-radix case will amplify this conversion error:

$$a + b \cdot (1 + \varepsilon) = (a + b) \cdot \left(1 + \frac{b}{a+b} \cdot \varepsilon\right), \quad |a + b| \ll |b| .$$

And even if we find ways to compute lower bounds on this catastrophic cancellation results of mixed-radix addition on its near path, i.e. if we find ways to make the conversion sufficiently accurate that some accuracy is left once the cancellation amplifies the error, the result of this (approximate) mixed-radix addition may itself be close to a rounding boundary of the output radix, preventing us to return the correctly rounded result with certainty.

On the far path, mixed-radix addition is no less difficult, as absorption can occur and lead to the same rounding issues. Even though we can construct exemplary cases where correct rounding of mixed-radix addition is hard to perform, we have no simple, systematic way to look for inputs to mixed-radix addition for which rounding is hard in round-to-nearest or a directed rounding mode. We therefore propose another approach to implementing a correctly rounded mixed-radix FMA, without precomputing such lower bounds.

III. Mixed-Radix FMA algorithm

The input of the mixed-radix FMA algorithm are either binary64 or decimal64 with the possibility of representing the midpoints by slightly increasing the precision. Those two formats are defined by the equations (1) and (2), and can be unified with a single representation of the mixed-radix unified format, with pessimist bounds on the exponents such as, given a an input of the FMA operation, we have:

$$\begin{aligned} &a = 2^{N_a} \cdot 5^{P_a} \cdot t_a;\ \text{with } 2^{54} \leq |t_a| < 2^{55};\ t_a \in \mathbb{Z} \\ &-1130 \leq N_a \leq 969;\ -421 \leq P_a \leq 385;\ N_a, P_a \in \mathbb{Z}. \end{aligned} \tag{3}$$

The algorithm that performs a fast and correctly rounded FMA defined in Figure 1 can be split in several parts. The first step is the decomposition of the binary64 and decimal64 inputs into the explicit format of sign, exponents and significand. Throughout this algorithm, we will analyze the bounds of those variables since we want to ensure that they can be stored on machine words: for the exponent on a signed 32-bit integer machine word and for the significand on one or several unsigned 64-bit integer machine words. With this internal format, we compute an error-free mixed-radix multiplication. Then we perform a test to decide which addition/subtraction algorithm to use, described in more detail later on. According to the result of this test, we convert the operands into an internal binary format and perform either a *far-path* addition/subtraction, or a *near-path* subtraction. We will then test if the final rounding in the output radix is possible with this fast-computed result. If the result is too close to a midpoint f, we cannot decide what is the correct floating-point value that should take the result according to the current rounding mode [7]. In that case we will have an extra computation step, called the recovery phase, to get those last bits of precision needed to decide how to round the result.

In the following Section, we will describe further the mechanisms of the mixed-radix FMA algorithm. We will first describe the *far-path* and *near-path* addition/subtraction algorithms of the fast computation, and then explain how work the rounding test and the recovery phase.

A. Multiplication and Ratio Test

The first operation performed by the FMA is pretty simple: the multiplication can be computed exactly in the mixed-radix unified format. According the definition of the mixed-radix unified format of the input operands a and b given in equation (3), we can compute the result of the multiplication such as

$$\begin{aligned} &\psi = a \times b = 2^{L} \cdot 5^{M} \cdot s;\ \text{with } 2^{109} \leq |s| < 2^{110};\ s \in \mathbb{Z} \\ &-2261 \leq L \leq 1938;\ -842 \leq M \leq 770;\ L, M \in \mathbb{Z}. \end{aligned} \tag{4}$$

We then want to perform the addition/subtraction between ψ and c, the third input operand of the FMA that is represented in the format given in equation (3). Previously, in Section II, we outlined two cases for the floating-point addition, according to the proximity of the two operands in terms of exponents. Those two cases are the far-path and near-path addition algorithms.

Deciding whether the operands are close or far enough to perform either one of those algorithm can determined by

```
Require: $a, b$ and $c$, binary64 or decimal64 numbers
Ensure: $R = 2^G \cdot p$ in binary or $R = 10^H \cdot q$ in decimal
 1: $\psi = a \times b$
 2: if it is an “addition” or $\frac{\psi}{c} \notin [\frac{1}{2}; 2]$ then
 3:     $T_1 \leftarrow 2^{\gamma_1} \cdot v_1 = \psi \cdot (1 + \varepsilon_{T1_{FP}})$
 4:     $T_2 \leftarrow 2^{\gamma_2} \cdot v_2 = c \cdot (1 + \varepsilon_{T2_{FP}})$
 5:     $\phi = (T_1 \pm T_2)(1 + \varepsilon_{\phi_{FP}})$ (“far-path”)
 6: else
 7:     $T_1 \leftarrow 2^{\Gamma_1} \cdot w_1 = \psi \cdot (1 + \varepsilon_{T1_{NP}})$
 8:     $T_2 \leftarrow 2^{\Gamma_2} \cdot w_2 = c \cdot (1 + \varepsilon_{T2_{NP}})$
 9:     $\phi = (T_1 - T_2)(1 + \varepsilon_{\phi_{NP}})$ (“near-path”)
10: end if
11: $\rho \leftarrow 2^C \cdot g = \phi$ or $\rho \leftarrow 10^H \cdot 2^{-10} \cdot q = \phi \cdot (1 + \varepsilon_{\rho_D})$
12: if $\rho = (a \times b + c)(1 + \varepsilon_\rho)$ can round correctly then
13:     return $R \leftarrow \rho$ correctly rounded into output format
14: else
15:     Compute integer rounding boundary significand $f$ such
        as $2^C \cdot 2^{10} \cdot 1/2 \cdot f$ or $10^H \cdot 2^{-10} \cdot 2^{10} \cdot 1/2 \cdot f$
16:     Let $\alpha \leftarrow 2^{L-Z_{min}} \cdot 5^{M-F_{min}} \cdot (a \times b) + 2^{N_c-Z_{min}} \cdot$
        $5^{P_c-F_{min}} \cdot c - 2^{F_T-Z_{min}} \cdot 5^{F_F-F_{min}} \cdot f)$ with $\alpha \in \mathbb{Z}$
17:     Correct $\rho$ using $f$ and the sign of $\alpha$
18:     return $R \leftarrow \rho$ correctly rounded into output format
19: end if
```

Fig. 1. Correctly Rounded Mixed-Radix FMA Algorithm

a simple test. If the ratio $\frac{\psi}{c}$ is clearly outside of $[\frac{1}{2}; 2]$, the operands are far enough to perform the faster far-path algorithm, otherwise the more accurate near-path algorithm is executed. This test ensures that when the result states that the far-path algorithm should be used the ratio is surely outside of $[\frac{1}{2}; 2]$.

Given the definitions of ψ and c given in equations (3) and (4), we want to ensure that either we have $2^{L-N_c} \cdot 5^{M-P_c} \cdot \frac{s}{t_c} < \frac{1}{2}$ or $2^{L-N_c} \cdot 5^{M-P_c} \cdot \frac{s}{t_c} > 2$. Thus certifying that the ratio $\frac{\psi}{c}$ is clearly outside of $[\frac{1}{2}; 2]$ is equivalent to evaluating the following inequation

$$\begin{aligned} &(L - N_c) + \lfloor (M - P_c) \cdot \log_2(5) \rfloor + 1 + 56 < -1 \\ &\text{or } (L - N_c) + \lfloor (M - P_c) \cdot \log_2(5) \rfloor + 54 > 1 \end{aligned} \tag{5}$$

Given the precomputed value $\log_2(5)$, $\lfloor A \cdot \log_2(5) \rfloor$ with $A \in \mathbb{Z}$ in a small domain can be computed correctly as $\lfloor A \cdot \lfloor \log_2(5) \cdot 2^{48} \rfloor \cdot 2^{-48} \rfloor$ which does not overflow on a 64-bit signed integer variable [5]. Hence we can easily compute on a 64-bit signed integer the two values given in equation (6) and perform ratio test that gives us two cases.

B. When no cancellation occurs (far-path)

1) Far-path algorithm: When the ratio test states that the result is clearly outside of $[\frac{1}{2}; 2]$, no cancellation can occur during the the addition/subtraction, hence we can use the *far-path* addition algorithm. For simplicity reasons, this algorithms will take binary floating-point inputs and return a binary floating-point result. As the operands ψ and c are represented in the mixed-radix unified format with precisions of respectively 110-bit and 55-bit precisions. Thus we first want to perform a conversion of those operands into a 64-bit binary format. This means that we want to represent the significand v stored on a 64-bit unsigned integer such as $2^{63} \leq |v| \leq 2^{64} - 1$, and adapt the binary exponent γ, as described by the lines 3 and 4 of algorithm 1. The next step being the computation of the correctly rounded in the current rounding mode binary addition/subtraction and leave the rounding of the result into the output format for further considerations.

As the operands ψ and c do not have the same significand precision, we might first think that we need two conversion algorithms from unified mixed-radix format to a 64-bit binary format. Yet it can easily be shown that we can first unify those two operands into a 64-bit mixed-radix format such as

$$\begin{aligned} &2^\lambda \cdot 5^\mu \cdot \omega;\ \text{with } 2^{63} \leq |\omega| \leq 2^{64} - 1;\ \omega \in \mathbb{Z} \\ &-2215 \leq \lambda \leq 1284;\ -842 \leq \mu \leq 770;\ \lambda, \mu \in \mathbb{Z}, \end{aligned} \tag{6}$$

an then apply a unique conversion algorithm from this mixed-radix format to a binary variable T represented with a 64-bit unsigned integer significand such as

$$\begin{aligned} &T = 2^\gamma \cdot v;\ \text{with } 2^{63} \leq |v| \leq 2^{64} - 1;\ v \in \mathbb{Z} \\ &-4171 \leq \gamma \leq 3772;\ \gamma \in \mathbb{Z}. \end{aligned} \tag{7}$$

Applying this conversion algorithm on ψ and c gives us the two variables $T_1 = 2^{\gamma_1} \cdot v_1$ and $T_2 = 2^{\gamma_2} \cdot v_2$ defined as in equation (7) as inputs for the *far-path* addition/subtraction algorithm, described in Figure 2.

The overall philosophy of this algorithm is that when neither of the operands are zero, we can order T_1 and T_2 according to their exponents such as $\gamma_1' \geq \gamma_2'$. Absorption occurs when $\gamma_1' \geq \gamma_2' + 64$, in that case the result of the far-path addition is $2^{\gamma_1'} \cdot v_1'$. Otherwise, to perform the addition of the two 64-bit unsigned integer exponents v_1' and v_2', we pose the intermediate 128-bit unsigned integer variables V_1 and V_2 and align the exponents by shifting v_1' to the left such as$V_1 = 2^{\gamma_1' - \gamma_2'} \cdot v_1'$. After performing the addition or subtraction on a 128-bit unsigned integer variable $V = V_1 \pm V_2$, the significand is normalized with the function `lzc`(V) that counts the number of leading zeros and truncated to a 64-bit unsigned integer variable g. The exponent C is set accordingly.

The result of the *far path* addition algorithm is then returned in the following binary format

$$\begin{aligned} &2^C \cdot g;\ \text{with } 2^{63} \leq |g| \leq 2^{64} - 1;\ g \in \mathbb{Z} \\ &-4174 \leq C \leq 3836;\ C \in \mathbb{Z}. \end{aligned} \tag{8}$$

2) Far-path error analysis: We will analyze the global relative error made during the computation of the far-path addition/subtraction algorithm, from the conversion of the inputs ψ and c until the acquisition of the result $\phi = 2^C \cdot g$. We shall denote this global error $\varepsilon_{\phi_{FP}}$.

The first operation producing an error occurs during the computation of the equation (6). In the case of ψ, as the precision decreased from 110 bits to 64 bits, an error arises, $2^{\lambda_\psi} \cdot 5^{\mu_\psi} \cdot \omega_\psi = 2^L \cdot 5^M \cdot s \cdot (1 + \varepsilon_\psi)$, and it can be bounded by $|\varepsilon_\psi| \leq 2^{-63}$. Then, given an input in the format described in equation (6), we compute the exponent γ and significand v as $\gamma = \lambda + \lfloor \mu \cdot \log_2(5) \rfloor$; $v = \lfloor 2^{-64} \cdot \omega \cdot \tau_0(\mu) \rfloor$, with the precomputed table $\tau_0(\mu) = \lfloor 5^\mu \cdot 2^{-\lfloor \mu \cdot \log_2(5) \rfloor + 63} \rfloor$, represented by a table of 64-bit unsigned integers. The relative error arising during the computation of the table can be bounded as $|\varepsilon_{\tau_0(\mu)}| \leq 2^{-63}$.

```
**Require:** $T_1 = 2^{\gamma_1} \cdot v_1$ and $T_2 = 2^{\gamma_2} \cdot v_2$
**Ensure:** $\phi = (T_1 \pm T_2)(1 + \varepsilon_{\phi_{FP}}) = 2^C \cdot g$
Order $T_1$ and $T_2$ so that $\gamma_1' \geq \gamma_2'$
**if** $\gamma_1' \geq \gamma_2' + 64$ **then**
    $g \leftarrow v_1'$
    $C \leftarrow \gamma_1'$
**else**
    $V_1 \leftarrow 2^{\gamma_1' - \gamma_2'} \cdot v_1'$
    $V_2 \leftarrow v_2'$
    $V \leftarrow V_1 \pm V_2$
    $\sigma \leftarrow \texttt{lzc}(V)$
    $\pi \leftarrow 2^{\sigma} \cdot V$
    $g \leftarrow \lfloor 2^{-64} \cdot \pi \rfloor$
    $C \leftarrow \gamma_2' + 64 - \sigma$
**end if**
**return** $\phi = 2^C \cdot g$
```

Fig. 2. Far-Path Addition/Subtraction Algorithm

The global relative error attached to the computation of the 64-bit precision binary variables T_1 and T_2 is $2^{\gamma} \cdot v = 2^{\lambda} \cdot 5^{\mu} \cdot \omega \cdot (1+\varepsilon_v) \cdot (1+\varepsilon_{\tau_0(\mu)})$. The relative error of v is a combination of the truncation error and the error attached to the computation of the table $\tau_0(\mu)$ such as ε_v is bounded by $|\varepsilon_v| \leq 2^{-62}$. The global relative error for this part of the mixed-radix FMA algorithm includes the conversion error ε_{ψ}, the error on the precomputed table $\varepsilon_{\tau_0(\mu)}$ and the rounding error ε_v. With those three errors combined, we deduce a pessimist bound on the global relative error $|\varepsilon_{T_{FP}}| \leq 2^{-60.5}$. Given T_1 and T_2 the inputs of the *far-path* addition, we want to bound the relative error $\varepsilon_{\phi_{FP}}$ as $\phi = (T_1 \pm T_2)(1 + \varepsilon_{\phi_{FP}})$.

The algorithm can compute the result in two different ways, firstly if $\gamma_1' \geq \gamma_2' + 64$, the result is $\phi = 2^{\gamma_1'} \cdot v_1' \cdot (1 + \varepsilon_{\phi_{FP}})$ with $|\varepsilon_{\phi_{FP}}| \leq 2^{-63}$. Secondly, if we are in the case when $\gamma_1' < \gamma_2' + 64$, hence $\gamma_2' \leq \gamma_1' \leq \gamma_2' + 63$ because $\gamma_1', \gamma_2' \in \mathbb{Z}$, we can deduce the bounds on V_1 and V_2 defined in the algorithm 2 such as $2^{63} \leq V_1 < 2^{127}$ and $2^{63} \leq V_2 < 2^{64}$. We want to compute bounds on $V = V_1 \pm V_2$. Either it is an addition and we have $2^{64} \leq V_1 + V_2 < 2^{127} + 2^{64} < 2^{128}$, or it is a subtraction, and we can show, with a *reductio ad absurdum*, that $2^{61} \leq \frac{1}{2} \cdot (1 - 2^{-59.49}) \cdot 2^{63} \leq V_1 + V_2 < 2^{127} + 2^{64} < 2^{128}$, with the fact that $1 - 2^{-59.49} \leq \frac{1+\varepsilon_{T1_{FP}}}{1+\varepsilon_{T2_{FP}}} \leq 1 + 2^{-59.49}$.

Following the steps of the algorithm, we then bound σ according to its definition such as $0 \leq \sigma \leq 66$; thus π is between $2^{127} \leq \pi \leq 2^{128}$ and finally we have $g = \lfloor 2^{-64} \cdot \pi \rfloor = 2^{-64} \cdot \pi \cdot \varepsilon_{\phi_{FP}}$. In the end we have $|\varepsilon_{\phi_{FP}}| \leq 2^{-63}$.

C. When cancellation does occur (near-path)

When the ratio test states that the result could be inside $[\frac{1}{2}; 2]$, cancellation may occur during the computation of the subtraction. Hence the use of the *near-path* subtraction.

A solution for this near-path algorithm is to use the same method as the far-path one and perform the subtraction on binary operands. To do so, we need to determine the number of bits of precision necessary for the binary inputs to compute the near-path subtraction in the worst case of cancellation and avoid loosing all accuracy.

1) Worst case of cancellation: The worst case occurs when, without being zero, the subtraction between ψ and c such as $2^L \cdot 5^M \cdot s - 2^{N_c} \cdot 5^{P_c} \cdot t_c$ is relatively small. This is equivalent to $2^L \cdot 5^M \cdot s \approx 2^{N_c} \cdot 5^{P_c} \cdot t$, thus $\frac{s}{t} \approx 2^{N_c - L} \cdot 5^{P_c - M}$, with $2^{54} \leq |t| < 2^{55}$ and $2^{109} \leq |s| < 2^{110}$. Let us define X and Y such as $X = N_c - L$ and $Y = P_c - M$; $X, Y \in \mathbb{Z}$. With this definition and the bounds of M, L, P_c and N_c given in equations (3) and (4), we can find the bounds of X and Y.

Lemma 1. *For all $X, Y \in \mathbb{Z}$ with $-3068 \leq X \leq 3230$ and $-1191 \leq Y \leq 1227$, for $s, t \in \mathbb{Z}$ with with $2^{54} \leq |t| < 2^{55}$ and $2^{109} \leq |s| < 2^{110}$*

$$\left| \frac{s}{t} - 2^X \cdot 5^Y \right| \geq \eta = 2^{-177.61}.$$

Proof. By application of the algorithm found in [5]. □

2) Near-path algorithm: Subsequently, we deduce that to compute the near-path subtraction with at least 60 bits of precision, we need to represent the binary significands with at least $178 + 60 = 228$ bits of precision. This means that we can represent the significands of the operands of the near-path subtraction on a 256-bit unsigned integer, i.e. four 64-bit unsigned integers machine words.

Hence we first compute the conversion from the mixed-radix unified format variables ϕ and c to 256-bit precision variables, with which we perform the near-path subtraction.

As in the far-path algorithm, we want to perform only one conversion into the binary internal format, therefore adapt the significand and exponent of c by increasing to the 110-bit precision of ψ as in equation (4). Then perform the conversion from the 110-bit precision mixed-radix format to the 256-bit precision binary format such as

$$\begin{aligned} &T = 2^{\Gamma} \cdot w; \text{ with } 2^{255} \leq |w| \leq 2^{256} - 1; \; w \in \mathbb{Z} \\ &-4364 \leq \Gamma \leq 3578; \; \Gamma \in \mathbb{Z}. \end{aligned} \tag{9}$$

The near-path subtraction algorithm is quite similar to the far-path except that the alignment of the significands yields to an error. We first order the two operands $T_1 = 2^{\Gamma_1} \cdot w_1$ and $T_2 = 2^{\Gamma_2} \cdot w_2$ such as $\Gamma_1 > \Gamma_2$. With the result T_1' and T_2', we align the significands so that $z_2 = \lfloor 2^{\Gamma_2' - \Gamma_1'} \cdot w_2' \rfloor$; $z_1 = w_1'$. After ordering z_1 and z_2 such as $z_1' > z_2'$, we can compute the subtraction $d = z_1' - z_2'$, perform the count of leading zeros in d with $\texttt{lzc}(d)$ such as we can normalize the significand and exponent of the result $g = \lfloor d \cdot 2^l \cdot 2^{-192} \rfloor$; and $C = \Gamma_1' - l + 192$. The result of the *near-path* subtraction takes the same form as the result of the *far-path* addition defined in equation (8), except for the bounds of the exponent that are $-2296 \leq C \leq 3773$.

3) Near-path error analysis: The first error that occurs during the computation of the near-path algorithm comes from the conversion from the mixed-radix unified format to the 256-bit precision binary format. We can denote this error $\varepsilon_{T_{NP}}$ and define it such as $2^{\Gamma} \cdot w = 2^L \cdot 5^M \cdot s \cdot (1 + \varepsilon_{T_{NP}})$, computed as $\Gamma = L - 18 + \lfloor M \cdot \log_2(5) \rfloor + \sigma$ with $0 \leq \sigma \leq 1$; $\sigma \in \mathbb{N}$ and $w = \lfloor 2^{-129} \cdot \tau_{0-3}(M) \cdot s \cdot 2^{18} \rfloor \cdot 2^{\sigma}$.

The table $\tau_{0-3}(M)$ is the 256-bit precision version of the table $\tau_0(\mu)$ used previously. We can show that $\tau_0(M) = \lfloor 2^{-192} \cdot \lfloor 2^{255 - \lfloor M \cdot \log_2(5) \rfloor} \cdot 5^M \rfloor \rfloor = \lfloor 2^{-192} \cdot \tau_{0-3}(M) \rfloor$. We can then bound the relative error induced by the computation of the table such as $|\varepsilon_{\tau_{0-3}}| \leq 2^{-255}$, and the global relative error of the conversion algorithm such as $|\varepsilon_{T_{NP}})| \leq 2^{-251}$.

The result ϕ of the near-path binary subtraction is subject to an error such as $\phi = (T_1 - T_2)(1 + \varepsilon_{\phi_{NP}})$. A first error appears during the computation of $z_2 = (2^{\Gamma'_2 - \Gamma'_1} \cdot w'_2) \cdot (1 + \varepsilon_{z_2})$. By noticing that $-2 \leq \Gamma'_2 - \Gamma'_1 \leq 0$ we can bound ε_{z_2} such as $|\varepsilon_{z_2}| \leq 2^{-253}$. This implies that $g = (d \cdot 2^l \cdot 2^{-192}) \cdot (1 + \varepsilon_{\phi_{NP}})$ with $2^{255} \leq d \cdot 2^l \leq 2^{256} - 1$. Finally we have $|\varepsilon_{\phi_{NP}}| \leq 2^{-63}$.

D. Rounding Test and recovery phase

1) Conversion to the output format: Once the multiplication and addition or subtraction are performed, we are left with a 64-bit binary significand with a binary exponent bounded such as $2^C \cdot g$ with $2^{63} \leq |g| \leq 2^{64} - 1$ and $-4174 \leq C \leq 3836$. If we want a binary output for the FMA operation, we can directly perform the rounding test that informs us whether the result can be correctly rounded into a binary64 floating-point number. If we want a decimal64 output, we must convert this 64-bit binary number into a decimal one such as

$$\begin{aligned} &10^H \cdot 2^{-10} \cdot q \text{ with } -1253 \leq H \leq 1159; H, q \in \mathbb{Z} \\ &10^{15} \cdot 2^{10} \leq q \leq (10^{16} - 1) \cdot 2^{10} < 2^{64} - 1. \end{aligned} \tag{10}$$

This algorithm computes the exponent $H = \lfloor (C + 63) \cdot \log_{10}(2) - 15 + \mu(C, g)$ where $\mu(C, g) \in \mathbb{N};\ \mu(C, g) = \lfloor C \cdot \log_{10}(2) + \log_{10}(g) - \lfloor (C + 63) \cdot \log_{10}(2) \rfloor \rfloor$. We can show that $0 \leq \mu(C, g) \leq 1$, implying that $\mu(C, g) \in \{0, 1\}$, by observing that $\mu(C, g)$ is an increasing function in g. Thus for a given C, there exists $g^*(C) \in \mathbb{Z};\ 2^{63} \leq g^*(C) < 2^{64}$,

$$\mu(C, g) = \left\{ \begin{array}{l} 0 \text{ if } g \leq g^*(C) \\ 1 \text{ if } g > g^*(C). \end{array} \right. \tag{11}$$

We can precompute such a $g^*(C)$ with a table of 64-bit unsigned integer words.

Once the exponent H is set, we can compute the significand q with a bounded precomputed table $t(H) = \lfloor 10^{-H} \cdot 2^{\lfloor H \cdot \log_2(10) \rfloor + 127} \rfloor$ and multiply it by g. The value is then normalized and truncated properly, conforming to the definition, in equation 10, of this operation's output.

We skipped the detailed error analysis of this algorithm for the sake of conciseness but the full pencil-and-paper proof will soon be available under the form of a research report. The relative error of this conversion is bounded such as $10^H \cdot 2^{-10} \cdot q = (2^C \cdot g) \cdot (1 + \varepsilon_{\rho_D})$ with $|\varepsilon_{\rho_D}| \leq 2^{-59.82}$.

2) Rounding test: The purpose of this test is to determine whether the 64-bit precision output of the addition/subtraction algorithm can be rounded into a a binary64, resp. decimal64, with any sign and rounding mode.

In the binary64 case, we have as input $2^C \cdot g = (\psi \pm c)(1 + \varepsilon^*_B)$ with ε^*_B the global error combining all the previous errors such as $|\varepsilon^*_B| \leq 2^{-58.74}$ and $2^{63} \leq |g| \leq 2^{64} - 1$. The rounding test boils down to compute the "magic bound" $\beta_B = \lfloor (2^{64} \cdot \frac{\varepsilon^*_B}{1 + \varepsilon^*_B} \rfloor$ and test if $|g - \lfloor 2^{-10} \cdot g \rceil 2^{10}| \geq \beta_B$. If this condition is not met, rounding to a 53-bit precision binary64 number is not possible. Therefore we compute f, the binary rounding boundary closest to $2^C \cdot g$ such as $2^{F_T} \cdot 5^{F_F} \cdot f = 2^C \cdot 2^{10} \cdot \frac{1}{2} \cdot f$ and $2^{54} \leq f \leq 2^{55} - 1$, and step into the recovery phase.

In the decimal64 case, we have as input $2^H \cdot 2^{-10} \cdot q = (\psi \pm c)(1 + \varepsilon^*_D)$ with ε^*_D the global error combining all the previous errors such as $|\varepsilon^*_D| \leq 2^{-58.18}$ and $10^{15} \cdot 2^{10} \leq |q| \leq (10^{16} - 1) \cdot 2^{10}$. The rounding test boils down to compute the "magic bound" $\beta_D = \lfloor ((10^{16} - 1) \cdot 2^{10} \cdot \frac{\varepsilon^*_D}{1 + \varepsilon^*_D} \rfloor$ and test if $|q - \lfloor 2^{-9} \cdot q \rceil 2^9| \geq \beta_D$. If this condition is not satisfied, rounding to a 16-digit precision decimal64 number is not possible. Therefore we compute f, the binary rounding boundary closest to $2^H \cdot q$ such as $2^{F_T} \cdot 5^{F_F} \cdot f = 10^H \cdot 2^{-10} \cdot 2^{10} \cdot \frac{1}{2} \cdot f$ and $2 \cdot 10^{15} \leq f \leq 2 \cdot (10^{16} - 1)$, and proceed to the recovery phase.

3) Recovery phase: If the rounding test fails, we try to round a result that is too close to the rounding border f to be able to decide towards which nearest floating-point number it must be rounded. Similarly to the computation of this floating-point rounding border's value in the previous step, the recovery phase boils down to computing the sign of $\alpha' = (-1)^{sa} \cdot 2^{L - Z_{min}} \cdot 5^{M - F_{min}} \cdot s + (-1)^{sb} \cdot 2^{N_c - Z_{min}} \cdot 5^{P_c - F_{min}} \cdot t_c - 2^{F_T - Z_{min}} \cdot 5^{F_F - F_{min}} \cdot f$ with bounds on the exponents such as $0 \leq F_T - Z_{min} \leq 2162;\ 0 \leq F_F - F_{min} \leq 786;\ 0 \leq L - Z_{min} \leq 2282;\ 0 \leq M - F_{min} \leq 785;\ 0 \leq N_c - Z_{min} \leq 2342;\ 0 \leq P_c - F_{min} \leq 786$.

Given those bounds, we know that we need to compute positive or zero integer powers of 5, i.e. 5^k, with $0 \leq k \leq 786$. Furthermore, we know that every 5^k holds on $\lceil \log_2(5^{786}) \rceil = 1826$ bits, hence 29 words of 64 bits. Finally, we determine that each term of α holds on respectively on 4218, 4223 and 4043 bits, so whatever their sign is, they hold on a maximum of 4225 bits. So to compute exactly α and decide its sign, we need 67 words of 64 bits, leaving 63 "free" bits.

We have three cases: if $\alpha = 0$ then the result is exactly the rounding border f, and will be rounded in the output format correctly by our final rounding algorithm according to the current rounding mode. If $\alpha < 0$, then the result is below the rounding border f and will round to the lower nearest floating-point number in the output format. By analogy, if $\alpha > 0$ then the result shall round to the higher nearest floating-point number in the output format. Computing this recovery phase is therefore very costly, but fortunately this part of the FMA algorithm will only be computed sparsely.

4) Final rounding to the output format: Final rounding to the supported output formats, binary64 and decimal64, takes either $(-1)^s \cdot 2^C \cdot g$, $2^{63} \leq g < 2^{64}$, or $(-1)^s \cdot 10^H \cdot 2^{-10} \cdot q$, $10^{15} \cdot 2^{10} \leq g < 10^{16} \cdot 2^{10}$, as input and rounds them to a binary64 or decimal64 floating-point number respectively, according to the current IEEE754 rounding mode. Binary rounding is done in the IEEE754 rounding mode for binary floating-point arithmetic, decimal rounding in the decimal mode. This final rounding also sets the inexact, underflow or overflow flags, when the rounding is respectively inexact, underflowed (tiny and inexact) or overflowed.

The difficulty in implementing this final operation is twofold: first, the rounding must address all special cases, such as subnormal rounding, normal rounding, overflow, decimal denormalization at the least exponent etc. Second, the rounding must be performed with the appropriate rounding mode, access to which is difficult. It also must set the flags. We overcome both challenges by exactly computing binary64, resp. decimal64, numbers that, provided to a IEEE754 floating-point operation, round to the same value as the input data.

For binary64 rounding, we decompose $(-1)^s \cdot 2^C \cdot g$ into a normal binary64 value 2^A, another normal binary64 value $2^B \cdot r$ and a normal or subnormal value $2^{C'} \cdot g'$ such that a binary64 FMA computes $2^A \times 2^B \cdot r + 2^{C'} \cdot g'$ and rounds to the same binary64 value as $(-1)^s \cdot 2^C \cdot g$. Typically $C' = C + 11$, $g' = \lfloor g \cdot 2^{-11} \rfloor$, $A + B = C$, $r = g - 2^{11} \cdot g'$.

For decimal64 rounding, we represent $(-1)^s \cdot 10^H \cdot 2^{-10} \cdot q$ exactly as an IEEE754 decimal128 floating-point value and perform an IEEE754 decimal128 to decimal64 format conversion, which rounds correctly and sets the flags. This way of performing final decimal rounding is known to be sub-optimal in terms of performance but has the advantage of being easy to implement. We consider changing this part of our algorithm and perform decimal64 rounding manually, while detecting the appropriate rounding mode by executing the avatar decimal64 instruction $t = (-1)^s \cdot (10^{15} + 1)$ resp. $t' = (-1)^s \cdot (10^{15} + 2)$ added to δ, with $\delta \in \{-3/4, -1/2, -1/4, 0, 1/4, 1/2, 3/4\}$, choosing t, t' and δ depending on the parity of a truncated significand and the value of the rest that got truncated off.

IV. Implementation and test result

A. Reference implementations

In order to perform software testing on our mixed-radix FMA implementation, we designed two other implementations of a mixed-radix FMA: one, based on the GNU Multiple Precision Library (GMP) [10], with the same C calling interface as our optimized FMA implementation, and one other, written in Sollya [11], with an ad hoc text-based interface. While designing these two reference implementations, we set the following design goals: the GMP-based implementation should be as close as possible to the result of a mixed-radix FMA designed in a limited timeframe, reusing existing software library. It should be reasonably fast, but easy to design.

We addressed this design goal by converting all input operands, be they IEEE754 binary64 or decimal64 numbers, into GMP rational numbers, i.e. fractions of (long) integers. We then performed the multiplication and addition steps of the FMA in GMP rational numbers. GMP can perform these operations on rational numbers without any error, computing exact long products, sums and gcds as required, while allocating memory dynamically. Our reference implementation then rounds the rational number, call it $\frac{p}{q}$, by first determining the output exponent for a binary64, resp. decimal64, number, call it E, and then "rounding" the rational number $\frac{r}{s} = 2^{-E} \cdot \frac{p}{q}$ (resp. $\frac{r}{s} = 10^{-E} \cdot \frac{p}{q}$) to an integer significand by long division of the long integer r by s, adapting the division rest's sign as required by the rounding mode. As the input/output's exponents may grow relatively large, GMP will manipulate integers of size of about a couple of thousand bits in the worst case.

In the next Section IV-B, we give an overview of the testing of our optimized FMA implementation we performed, comparing its results to the GMP-based reference implementation, and also compare timing results. While performing software development and intermediate testing of both our optimized implementation and the GMP-based one, we found certain input operands where the two algorithms did not agree. In order to have a way to decide where to look out for issues, we designed yet another implementation of a mixed-radix FMA, in the interpreted numerical language Sollya [11]. In this tool, numerical expressions can be written exactly, and evaluated to any precision without being hurt by spurious roundings. It is rather easy to give definitions of the output exponent and significand of a mixed-radix FMA in terms of the input exponents and integer significands. Particularities due to subnormal rounding, underflow and overflow can be addressed through minimum/maximum computations.

We used our Sollya reference implementations to generate "gold" result test vectors, with which we then verified both our optimized FMA implementation and the GMP-based reference. After our software testing campaign, all our three implementations agreed. While we spent quite some time on testing the implementation of our mixed-radix FMA algorithm against reference implementations, we would like to insist that we consider our algorithm to be correct only because we were able to prove its correctness, the detailed pencil-and-paper proof made available under the form of a research report.

B. Test and Timing Results

We tested our mixed-radix FMA implementation in all its 14 different operand-result-format variants on a system based on an Intel i7-7500U quad-core processor, clocked at maximally 2.7GHz, running Debian/GNU Linux 4.9.0-5 in x86-64 mode. We compiled our code using gcc version 6.3.0 with optimization level 3 and setting the `-march=native` flag. On our system, this results in AVX instructions being used, including hardware binary64 FMA. The IEEE754-2008 decimal types and instructions come from our gcc's libgcc. The GMP-based reference implementation was based on GMP version 6.1.2. Sollya was at git commit 12bf2006c.

We tested each of the 14 variants on an appropriate test vector file containing at least 115000 test points, covering all possible cases of signs, signed zeros, infinities, NaNs, subnormal and normal numbers, as well as all possible binary and decimal rounding modes[1]. We found all our 14 mixed-radix FMA implementations to be correct with respect to their numerical outputs for all the outlined cases.

We also performed testing of the implementations concerning the correct setting of the IEEE754-2008 flags, as they are inexact, overflow, underflow, divide-by-zero and invalid [1]. We found all flags to be set correctly, i.e. our implementations correctly raised the appropriate IEEE754-2008 when the operation actually was inexact (i.e. implied rounding), overflowed, underflowed or was invalid (as for multiplication of zero by infinity). The implementation also was checked for correctly leaving the flag unchanged if it did not encounter the suited IEEE754-defined condition to raise the flag. One issue was found about this last check for the mixed-radix FMA implementations for certain decimal exact results, the inexact flag was raised spuriously. We traced this undue raising of the inexact flag back to the implementation of the IEEE754 decimal128 to decimal64 format conversion that is contained in our version of libgcc. A bug has been filed against libgcc.

Concerning performance testing, we ran our 14 implementations on each of the entries of our result test vectors, timing the function call time in terms of machine cycles using the

[1] and rounding mode combinations, as IEEE754-2008 specifies separate current rounding modes for the binary and decimal operations' subsets [1]

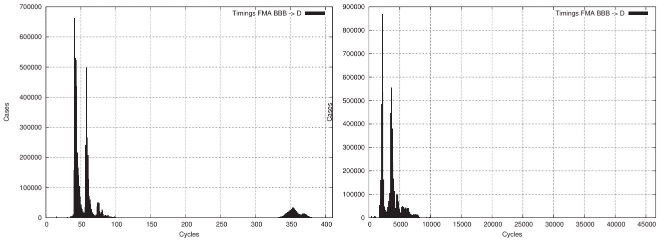

Fig. 3. Histogram of timing measurements for our implementation (left) and a reference implementation (right)

Intel `rdtsc` instruction after serialization with `cpuid`. We deducted the measured calling-time for an empty function with the same prototype. We ran the timing measurements repeatedly and with preheated caches, so to smooth out effects of modern superscalar processors. We made sure that the test vectors contained a representative, random subset of floating-point numbers, i.e. we made sure that special cases, such as addition and multiplication of zeros, NaNs etc. were represented but not over-represented.

We present our timing measurements under the form of histograms which relate a certain cycle count interval to the number of times this cycle count interval was encountered. We performed the same performance testing on our optimized FMA implementation and on our GMP-based mixed-radix FMA one. For the sake of succinctness, we report only the performance testing results of one FMA function, called DBBB. This function takes all its arguments as IEEE754 binary64 numbers and returns a decimal64 number.

On the left of Figure 3 is the cycle count histograms for our DBBB FMA functions. It is easy to see that our implementations handle most cases in less than 50 cycles. Our implementations present a maximum call time of about 375 cycles, corresponding to cases when the rounding recovery phase gets used. As our algorithm is statically bounded in terms of memory consumption and makes no dynamic memory allocation, these maximum timing counts are hard bounds.

On the right of Figure 3 is the cycle count histograms of the GMP-based implementations for DBBB. For most cases, the GMP-based implementations is about 10 times slower than the implementation based on our algorithm presented in this article. The GMP-based maximum call times reach up to 46000 cycles in very rare cases, which is about 100 times slower than ours. This is due to the cost incurred by GMP's dynamical memory allocation mechanism.

V. Conclusion

In this article, we have presented an algorithm to compute a correctly-rounded IEEE754 binary64 or decimal64 result to a mixed-radix Fused-Multiply-and-Add operation, taking any combination of IEEE754 binary64 and decimal64 numbers in argument. Our algorithm trivially allows for covering correctly rounded mixed-radix addition, subtraction and multiplication and is designed in a way that allows mixed-radix division and square root to be computed with that FMA, too.

Our algorithm is based on a mixture of exact significand multiplication, finely precision-tuned radix conversion, binary addition and an exact rounding recovery phase that is launched if rounding of an approximation would not allow a correctly rounded result to be computed. The recovery phase is based on the observation that all IEEE754 binary64 and decimal64 floating-point numbers and their mid-points are integer multiples of a fixed $2^Q \cdot 5^W$, with constant $Q, W \in \mathbb{Z}, Q, W \leq 0$.

Our implementation can handle most input cases in about 50 machine cycles, which is a reasonable performance timing for a software-implemented floating-point operation. When the recovery phase gets used, latency stays below 400 cycles. Our implementation does not require any dynamic memory allocation, hence being suitable for integration into environments where dynamic memory management is not available.

Our implementation has been thoroughly tested against two other reference implementations, which we developed. Besides an issue with an undue IEEE754 inexact flag setting, which is outside our control, it tests correct. A detailed pencil-and-paper proof for our algorithm has been written and will be made available under the form of a research report.

While we are convinced that our implementation is one of the first step to paving the road to generalized mixed-radix floating-point arithmetic in IEEE754, quite a few points need to be addressed by future work: we subsumed all 14 possible mixed-radix binary and decimal input/output format combinations for the FMA under one single core algorithm. This eased development and the correctness proof; but it induces certain overhead due to exponent range over-estimation. Future work might perform a more detailed one-by-one analysis.

We showcased our mixed-radix FMA for the IEEE754 formats binary64 and decimal64 only. All possible combinations taking into account also the binary32 and decimal128 formats need to be covered by future development.

The necessity to fall back to exact arithmetic with pretty large integer-based accumulators for hard-to-round cases is not very intellectually satisfactory. Future work should try to address the issue of precomputing worst-case precision bounds for mixed-radix FMA again.

References

[1] *IEEE Standard for Floating-Point Arithmetic*, 2008, IEEE Std 754-2008.

[2] J. Harrison, "Decimal Transcendentals via Binary," in *2009 19th IEEE Symposium on Computer Arithmetic*, June 2009, pp. 187–194.

[3] M. Cornea *et al.*, "A Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic Using the Binary Encoding Format," in *18th IEEE Symposium on Computer Arithmetic*, June 2007, pp. 29–37.

[4] W. E. Wong *et al.*, "Recent Catastrophic Accidents: Investigating How Software was Responsible," in *4th Int. Conf. on Secure Software Integration and Reliability Improvement*, 2010, pp. 14–22.

[5] N. Brisebarre *et al.*, "Comparison Between Binary and Decimal Floating-Point Numbers," *IEEE Transactions on Computers*, vol. 65, 2016.

[6] O. Kupriianova, "Towards a Modern Floating-Point Environment," Theses, Université Pierre et Marie Curie - Paris VI, Dec. 2015.

[7] F. De Dinechin *et al.*, "On Ziv's Rounding Test," *ACM Transactions on Mathematical Software*, vol. 39, no. 4, p. 26, 2013.

[8] J.-M. Muller *et al.*, *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[9] P. H. Sterbenz, *Floating-Point Computation*, ser. Prentice-Hall series in automatic computation. Prentice-Hall, 1974.

[10] T. Granlund and the GMP development team, *GNU MP: The GNU Multiple Precision Arithmetic Library*, 2016, http://gmplib.org/.

[11] S. Chevillard, M. Joldeş, and C. Lauter, "Sollya: An Environment for the Development of Numerical Codes," in *Mathematical Software - ICMS 2010*, ser. LNCS, vol. 6327, Sept 2010, pp. 28–31.