A Correctly-Rounded Fixed-Point-Arithmetic Dot-Product Algorithm

Sylvie Boldo*, Diane Gallois-Wong* and Thibault Hilaire‡
*Université Paris-Saclay, CNRS, Inria, Laboratoire de recherche en informatique, F-91405, Orsay, France
‡Sorbonne Université, LIP6 UMR 7606, F-75005 Paris, France

***Abstract*—Dot products (also called sums of products) are ubiquitous in matrix computations, for instance in signal processing. We are especially interested in digital filters, where they are the core operation. We therefore focus on fixed-point arithmetic, used in embedded systems for time and energy efficiency. Common dot product algorithms ensure faithful rounding. For the sake of accuracy and reproducibility, we want to ensure correct rounding. This article describes an algorithm that computes a correctly-rounded sum of products from inputs whose format is known in advance. This algorithm relies on odd rounding (that is easily implemented in hardware) and comes with a careful proof and some cost analysis.**

***Index Terms*—Dot Product, Sum-of-Products, Correct Rounding, Odd Rounding, Fixed-Point Arithmetic**

I. Introduction

Many algorithms used in embedded systems rely on dot product algorithms, for example in linear algebra based systems. Notably, many applications in automotive, communication, aerospace domains use signal processing and control algorithms named filters and controllers that have dot product evaluation as core.

Because of cost, power efficiency, or performance constraints, these low-level algorithms are often implemented using fixed-point arithmetic (eg. based on the integer arithmetic unit) rather than floating-point arithmetic. Moreover, in hardware implementation (ASIC or FPGA), the multiple word-length paradigm [1] (ie. the use of dedicated word-length for each computation) is key to minimize the area and/or the power consumption while preserving a good performance, and bounding the error due to the finite precision by an acceptable level.

In this article, we only focus on the dot product (also called sum-of-products) in fixed-point arithmetic. We also assume that the fixed-point formats involved (the Most Significant Bit and Least Significant Bit positions, as defined in Section II) are fixed and given by the user. We propose an efficient algorithm that ensures correct rounding and we prove its correctness. It is based on odd rounding, an unusual rounding, defined in Section II-A, which is easy to implement (the proof will be useful for the proof of our fixed-point dot product, and then of a complete filter or controller implementation).

A correctly rounded dot product is known to be difficult to achieve, as we cannot dismiss the least significant bits. A solution is to rely on a Kulisch accumulator [2], but this may be costly when the exponent range is large. Indeed, it uses the complete bit-range for all the accumulation in order to perform only exact operations before the last rounding. Our algorithm is more efficient and still ensures correct rounding.

Other practical solutions for the dot product exist, but only provide last-bit accuracy dot product (also denoted faithful rounding: it is not a correct rounding, only a round up or round down of the exact result). The main idea is to use a slightly extended precision for the internal computation of the sum, using some extra guard bits. The number of them depends on the log2 of the number of terms to add [3].

Section II gives a brief overview of fixed-point arithmetic and odd rounding, and introduces related notations. The correctly-rounded dot product is detailed in Section III: a simple version is first presented, then an optimized one illustrated by two examples. The proof of the correct rounding is given in Section IV. Some elements of comparison of the proposed algorithm to classical ones are given in Section V. Section VI features a conclusion.

II. Odd Rounding and Fixed-Point Arithmetic

As for the pre-requisite, we first describe odd rounding and how to implement it, and then fixed-point arithmetic, and the used notations.

A. *Odd rounding*

Correct rounding is notably difficult to achieve for some functions due to the Table Maker Dilemma [4]. In particular, common algorithms for sums or dot products usually only ensure faithful rounding [4], [5]. As explained above, we want to ensure correct rounding, that requires more care to ensure.

A solution is to rely on a specific rounding mode, called rounding to odd, and using it for intermediary roundings. This odd rounding is also called von Neumann rounding because it has been first designed in the 40's by Von Neumann for the design of the arithmetic unit of the EDVAC computer [6, section 9.4] [7]. It was lately used by Goldberg when converting binary FP numbers to decimal representations [8] and formally studied later, notably to emulate the FMA operator [9].

The informal definition of odd rounding is the following one. When a real number is not in the set of representable numbers $\mathbb{F}$, it is rounded to the adjacent number in the format with an odd integer significand. More precisely, let $\bigtriangleup$ be the

This work is supported by a grant from the "Fondation CFM pour la Recherche".

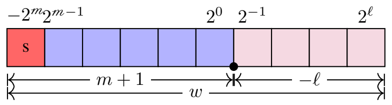

Fig. 1: Fixed-point number in format (m, ℓ).

rounding up and $\bigtriangledown$ the rounding down to $\mathbb{F}$. Then rounding to odd $\square$ is defined by:

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

Note that the result of x rounded to odd can be even only when x is already a representable number.

This rounding belongs neither to the IEEE-754 required roundings for floating-point arithmetic, nor to the IEEE-1666 roundings for fixed-point arithmetic in SystemC [10]. Nevertheless, implementations are available. An effective software implementation is given in [9]. As explained, a hardware one is much simpler: round to zero (with the IEEE-754 flags), and perform a logical or between the inexact flag (or the sticky bit) of this first rounding step and the last bit of the mantissa to get the result (or equivalently it forces the last bit of the mantissa to 1 when the deleted bits are not all zero). From an IEEE-compliant architecture, the additional cost is therefore slight.

Numerous properties are given in [9]. They are mostly dedicated to floating-point arithmetic, while we are interested in fixed-point arithmetic as explained above. Fortunately, many theorems are generic enough to be used both in floating- and fixed-point arithmetics. In particular, we will use here Theorem 1, that ensures correct rounding to nearest, provided a previous odd rounding with sufficient precision:

Theorem 1 (round_N_odd, from [9])**.** *Assume an even radix β and two different formats: a working format and an extended format. Assume that the extended format has at least two more digits at all points. If $\square_{ext}$ denotes the rounding to odd in the extended format, and if $\bigcirc$ denotes a rounding to nearest with any tie-breaking rule in the working format, we have*

$$\forall x \in \mathbb{R}, \quad \bigcirc(\square_{ext}(x)) = \bigcirc(x).$$

We refer to Corollary 2 for an explicit version of this theorem applied to fixed-point arithmetic (see Section IV).

B. Fixed-Point Arithmetic

We consider here numbers and computations based on two's complement fixed-point arithmetic. Let x be a w-bit long fixed-point number, with bits $\{x_i\}_{m \le i \le \ell}$:

$$x = -2^m x_m + \sum_{i=\ell}^{m-1} 2^i x_i \tag{1}$$

where m and ℓ are the positions of the most significant bit (MSB) and the least significant bit (LSB) of x, respectively. So the word-length w and the Most and Least Significant Bit positions, m and ℓ are linked with

$$w = m - \ell + 1. \tag{2}$$

The couple (m, ℓ) denotes the fixed-point format of x. Figure 1 exhibits a fixed-point number and its binary representation.

Internally, the fixed-point numbers are represented with an integer X (denoted *mantissa*) formed by the w-bit $\{x_j\}$ and an implicit scaling factor 2^ℓ, ie,

$$X = -2^{w-1} x_m + \sum_{i=0}^{w-2} 2^i x_{i+\ell} \in \mathbb{Z} \tag{3}$$

or equivalently $x = X 2^\ell$. All the operations are then integer operations on the mantissa X.
We denote $\mathbb{F}_\ell$ the set of fixed-point integers with ℓ as LSB (or equivalently $\mathbb{F}_\ell = \mathbb{Z}2^\ell$), and we do not consider overflow here.

We will also need the following notations: $\bigtriangledown_\ell(x)$, $\triangle_\ell(x)$, $\square_\ell(x)$, $\bigcirc_\ell(x)$ are the rounding down, rounding up, rounding to odd and rounding to nearest (given a tie-breaking rule) of x at the bit ℓ, respectively. Moreover, we denote $x \overset{+}{\bigtriangledown}_\ell y$ and $x \boxplus_\ell y$ the addition of x and y rounded at bit ℓ, using rounding down and rounding odd respectively. We assume correct rounding for addition, so we have $x \overset{+}{\bigtriangledown}_\ell y = \bigtriangledown_\ell(x+y)$ and $x \boxplus_\ell y = \square_\ell(x+y)$.

III. Algorithm

Let us now describe our algorithm. A first point is that we decide to compute exactly all multiplications. The fixed-point numbers obtained each have an LSB equal to the sum of the LSBs of both corresponding operands. What is left to do is to compute a correct sum of fixed-point numbers with LSBs given in advance. Therefore, we just consider that we have n non-null inputs noted $x_0, \ldots, x_{n-1}$ with respective LSBs $\ell_0, \ldots, \ell_{n-1}$ known in advance. We are also given a desired LSB ℓ_f for the output. The algorithm computes an output noted *result* which is a correct rounding of the sum of the inputs:

$$\textit{result} = \bigcirc_{\ell_f}(x_0 + \cdots + x_{n-1}).$$

This summation relies on odd rounding presented in Section II-A. Then, the intuition is that, to obtain a correct rounding to nearest at LSB ℓ_f, we need an odd rounding at LSB $\ell_f - 2$. Indeed, Theorem 1 will then guarantee a correct rounding of the sum. Of course, we do not use a Kulisch accumulator for computing this odd rounding as it would defeat our efficiency wish. Instead, we add the values sequentially, using odd rounding with a well-chosen LSB at

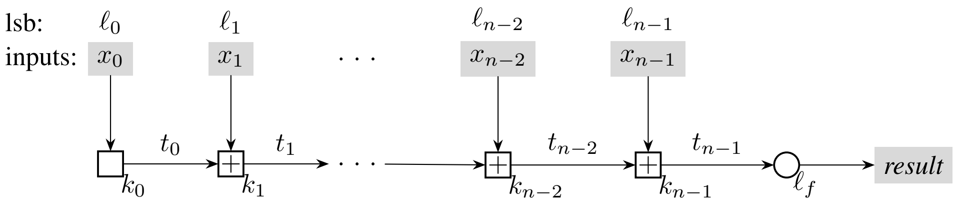

Fig. 2: General algorithm for any input LSBs.

each step to ensure that we keep the following invariant: the current computed value is the odd rounding of the sum of all the previously-seen values.

The key ingredient is the choice of the LSB at each step of the algorithm. Too large, and we cannot ensure correct rounding; too small, and the algorithm will be less efficient. For instance, if we used the smallest LSB of all inputs for all computations, we would get a correct result *à la* Kulisch.

A first version of the algorithm is described in Section III-A. Section III-B presents an optimized version, where inputs are sorted by LSB (the LSBs are known in advance so there is no runtime cost) to allow for overall larger computing LSBs. Examples are given in Section III-C. Note that all the proofs will be presented in Section IV; only the intuition is given here.

A. General algorithm

The general algorithm described by Algorithm 1 computes a correct rounding of the sum for any input LSBs. First of all, the LSBs k_i used in intermediary computations are chosen recursively. Then, we sum the inputs x_i from left to right using odd rounded additions to $\mathbb{F}_{k_i}$ as illustrated by Fig. 2. The t_i could all be replaced with a single variable t that is updated at each step of the loop. However, using indices to identify the value at each iteration will be useful for the proof in Section IV-B.

Algorithm 1: General algorithm for any input LSBs.

```
  Inputs: $x_0, \dots, x_{n-1}, \ell_0, \dots, \ell_{n-1}$
  // $x_i$ has LSB $\ell_i$
  Output: result $= \bigcirc_{\ell_f}(x_0 + \cdots + x_{n-1})$
1 $k_{n-1} \leftarrow \ell_f - 2$
2 for $i \leftarrow n-1$ downto $1$ do
3     $k_{i-1} \leftarrow \min(k_i, \ell_i) - 1$
4 $t_0 \leftarrow \square_{k_0}(x_0)$
5 for $i \leftarrow 0$ to $n-2$ do
6     $t_{i+1} \leftarrow t_i \boxplus_{k_{i+1}} x_{i+1}$
7 result $\leftarrow \bigcirc_{\ell_f}(t_{n-1})$
```

The larger the LSBs used in computations, the more efficient the algorithm. Note that for $i \leq n-1$, we define k_{i-1} as the larger LSB that verifies both $k_{i-1} < k_i$ and $k_{i-1} < \ell_i$. Indeed, these inequalities are required to keep on with odd rounding. For that, we will rely on Corollary 6 in the proof in Section IV. It is easy to notice that all these computation LSBs are maximized when the input LSBs ℓ_i are sorted in increasing order. The optimized algorithm presented in the next section builds on this remark and similar considerations to maximize the LSB at each step.

B. Optimized algorithm

The optimized algorithm is derived from the general algorithm above through a few successive optimizations. As explained previously, the first one consists in sorting the inputs by LSB, as this maximizes the computing LSBs k_i. As the input LSBs are known in advance, sorting by LSB has no runtime cost. Therefore, we can assume that $\ell_0 \leq \ell_1 \leq \cdots \leq \ell_{n-1}$.

Yet to maximize efficiency, the hypothesis we need is $\ell_0 < \cdots < \ell_{n-1}$. Therefore, we do a pre-processing where inputs with the same LSB are grouped together as described by Algorithm 2. This may decrease the number of inputs n and modify the values of the x_i and ℓ_i. But it preserves the exact sum $x_0 + \cdots + x_{n-1}$, as well as the fact that each x_i has LSB ℓ_i. And afterwards, we do have $\ell_0 < \cdots < \ell_{n-1}$.

Algorithm 2: Pre-processing to make input LSBs distinct.

```
1 while there is an $i$ such that $\ell_i = \ell_{i+1}$ do
2     $x_i \leftarrow x_i \boxplus\!\!\!\!\bigtriangledown_{\ell_i} x_{i+1}$
3     for $j \leftarrow i+1$ to $n-2$ do
4         $x_j \leftarrow x_{j+1}$
5         $\ell_j \leftarrow \ell_{j+1}$
6     $n \leftarrow n-1$
```

The optimized algorithm for strictly increasing input LSBs is described in Algorithm 3 and illustrated by Fig. 3. Inputs are separated into two groups: on the left hand side $x_0, \dots, x_{m-1}$ whose LSBs are at most $\ell_f - 2$, and on the right hand side $x_m, \dots, x_{n-1}$ whose LSBs are greater than $\ell_f - 2$. The left hand side is handled similarly to the previous general algorithm. The main difference is that thanks to the LSBs being strictly increasing, we do not need to define computation LSBs recursively. Instead, each former k_i is simply equal to $\ell_{i+1} - 1$ (except for $k_{m-1} = \ell_f - 2$). On the right hand side, inputs with large LSBs are added together from right to left. At each step, we use the smaller LSB of both operands so that the operation is exact. Finally, the results for both groups of inputs are added together at $\ell_f - 2$ (the operation is once again guaranteed to be exact) then rounded to nearest at ℓ_f.

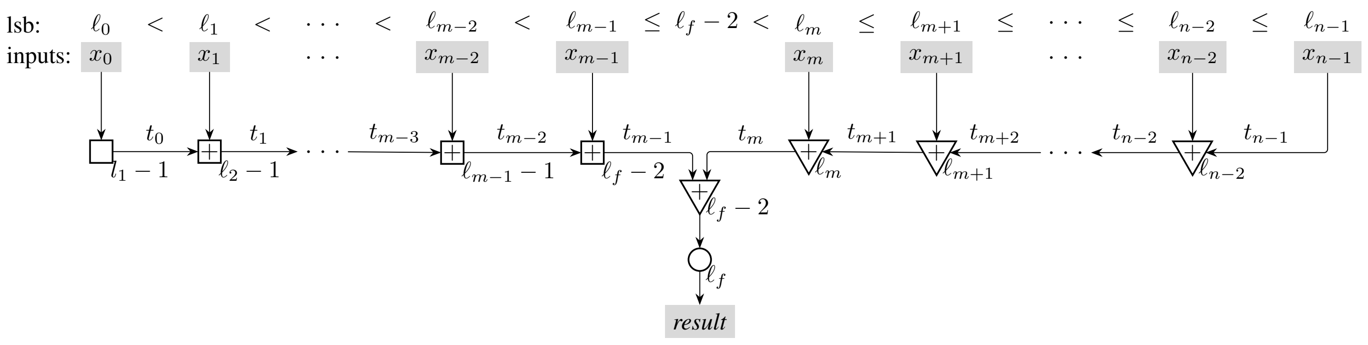
lsb:
$\ell_0 < \ell_1 < \cdots < \ell_{m-2} < \ell_{m-1} \le \ell_f - 2 < \ell_m \le \ell_{m+1} \le \cdots \le \ell_{n-2} \le \ell_{n-1}$
inputs:
x_0 x_1 $\cdots$ x_{m-2} x_{m-1} x_m x_{m+1} $\cdots$ x_{n-2} x_{n-1}
t_0 t_1 t_{m-3} t_{m-2} t_{m-1} t_m t_{m+1} t_{m+2} t_{n-2} t_{n-1}
$\ell_1 - 1$ $\ell_2 - 1$ $\ell_{m-1} - 1$ $\ell_f - 2$ ℓ_m ℓ_{m+1} ℓ_{n-2}
$\ell_f - 2$
ℓ_f
result

Fig. 3: Optimized algorithm.

Note that every time we use a rounded down addition $\triangledown_\ell$, we know that the operation is actually exact. This means we could have used any rounding mode. We choose rounding down for its efficiency: it is a simple truncation in two's complement fixed-point arithmetic.

It may be the case that all inputs are either in the set with large LSBs or the one with small LSBs, making the other set empty. Then, m is respectively 0 or n. The same algorithm holds, provided we consider all t_i for i outside of the bounds $\{0, \ldots, n-1\}$ to be null. When $m = 0$, this means that we can ignore Lines 2 to 5 (both included), and we use $t_{m-1} = 0$ at Line 9. When $m = n$, we should ignore Lines 6 to 8, and we use $t_m = 0$ at Line 9. Note that this convention also affects the case $m = 1$. Indeed, Line 5 becomes $t_0 = 0 \boxplus_{\ell_f-2} x_0 = \square_{\ell_f-2}(x_0)$ which overrides Line 2.

Algorithm 3: Optimized algorithm.

```
Inputs: $x_0, \ldots, x_{n-1}, \ell_0, \ldots, \ell_{n-1}$
// $x_i$ has LSB $\ell_i$ and $\ell_0 < \cdots < \ell_{n-1}$
Output: result $= \bigcirc_{\ell_f}(x_0 + \cdots + x_{n-1})$
1 let $m$ such that $\ell_{m-1} \le \ell_f - 2 < \ell_m$
2 $t_0 \leftarrow \square_{\ell_1-1}(x_0)$
3 for $i \leftarrow 0$ to $m-3$ do
4     $t_{i+1} \leftarrow t_i \boxplus_{\ell_{i+2}-1} x_{i+1}$
5 $t_{m-1} \leftarrow t_{m-2} \boxplus_{\ell_f-2} x_{m-1}$
6 $t_{n-1} \leftarrow x_{n-1}$
7 for $i \leftarrow n-1$ downto $m+1$ do
8     $t_{i-1} \leftarrow t_i \triangledown_{\ell_{i-1}} x_{i-1}$ // addition is exact
9 result $\leftarrow \bigcirc_{\ell_f}(t_{m-1} \triangledown_{\ell_f-2} t_m)$ // add. is exact
```

C. Example

Let us now illustrate the optimized algorithm on two examples with explicit LSBs. The first one is a toy example that showcases every part of the algorithm. The second one was generated from an actual digital filter as explained below.

As a first example, we are given inputs with respective LSBs 0, 4, 4, 5, 8, 9, 9, 12, and 20. The desired output LSB is $\ell_f = 10$. After pre-processing, x_0 is the input with LSB $\ell_0 = 0$, x_1 is the exact sum of the two inputs with LSB $\ell_1 = 4$ (computed with an addition truncated to $\mathbb{F}_4$), and so on: $\ell_2 = 5$, $\ell_3 = 8$, $\ell_4 = 9$, $\ell_5 = 12$, and $\ell_6 = 20$. Then $m = 4$ since $\ell_3 \le \ell_f - 2 < \ell_4$. We can now apply the algorithm as illustrated by Fig. 4.

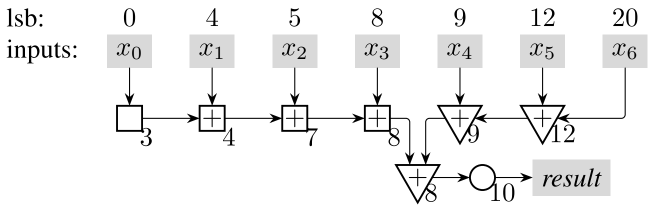

Fig. 4: Optimized algorithm: toy example.

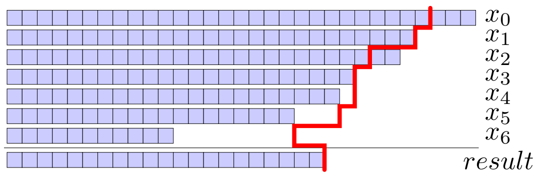

Fig. 5: Bit view: toy example.

Another point of view is given in Fig. 5. All bits of the inputs are shown. The red line shows the intermediary LSBs used for computations (that is to say the LSBs of the t_i) to give an idea on how many bits are kept and dismissed at each step. More precisely, each line i shows both x_i and a vertical red line at the LSB of t_i (remember $t_i \leftarrow t_{i-1} \boxplus_{\ell_{i+1}-1} x_i$). In particular, Fig. 5 pictures the dropped bits and we clearly see that the smallest LSBs are quickly dropped.

Let us now consider an example from signal processing. We consider a 7-th order low-pass Butterworth filter with normalized cutoff frequency equal to 0.41. Its evaluation using a Direct Form algorithm[1] [11], [12] requires the use of a dot

[1]Other possibilities for the implementation of this filter exist, implying the use of multiple-but-shorter dot products. The comparison of these different algorithms and their implementation is outside of the scope of this article.

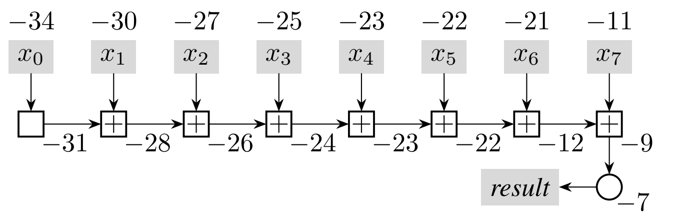

Fig. 6: Optimized algorithm: example from signal processing.

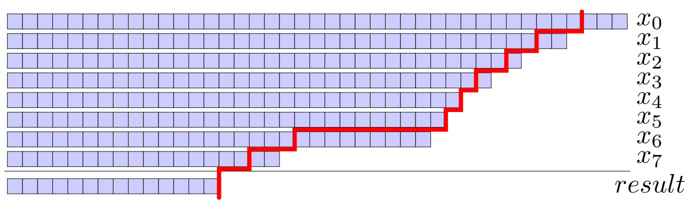

Fig. 7: Bit view: example from signal processing.

product that sums 10 terms[2] with respective LSBs -34, -30, -27, -25, -23, -22, -22, -22, -21, and -11. Moreover, the output should have LSB $\ell_f = -7$.

Once again, inputs are grouped by LSB then labeled $x_0, \dots, x_7$, where x_5 is the exact sum of the three inputs with LSB -22. But this time, all input LSBs are smaller than $\ell_f - 2$. Technically, this means that $m = n = 8$ and the group of inputs with large LSBs is empty so their sum is zero. In practice, we can remove all the truncated additions from Fig. 3. We obtain Fig. 6.

As before, the bit view is given in Fig. 7 with the red line showing the intermediary LSBs used for computations. Again, the dropped bits are clear, even if we need to consider them all to ensure correct rounding.

IV. Proof

Now let us prove the correctness of these algorithms. This Section is organized as follows: Section IV-A provides general lemmas about odd rounding. Section IV-B tackles the general algorithm and Section IV-C the optimized one.

A. Useful properties about odd rounding

Odd rounding was defined in Section II-A. We have put here all the lemmas about odd rounding that we need to prove the correctness of the algorithm.

Corollary 2. *For all $x \in \mathbb{R}$ and $k, \ell \in \mathbb{Z}$, if $k \le \ell - 2$ then $\bigcirc_\ell(\square_k(x)) = \bigcirc_\ell(x)$.*

This is only an instance of Theorem 1, where the format is fixed-point arithmetic.

Lemma 3. *For all $x \in \mathbb{R}$ and $k, \ell \in \mathbb{Z}$, if $k \le \ell$ then $\square_\ell(\square_k(x)) = \square_\ell(x)$.*

[2]The coefficients of this filter can be obtained with the `butter` function of the Scipy library (or equivalently with Matlab) with the command `butter(7,0.41)`.

Proof. When $k = \ell$, this is trivial: as $\square_k(x)$ has LSB $k = \ell$, then $\square_\ell(\square_k(x)) = \square_k(x) = \square_\ell(x)$.

When $x \in \mathbb{F}_k$, we have $\square_k(x) = x$, therefore $\square_\ell(\square_k(x)) = \square_\ell(x)$.

Let us now assume $k < \ell$ and $x \notin \mathbb{F}_k$. To prove that $\square_\ell(\square_k(x))$ is the odd rounding of x to $\mathbb{F}_\ell$, it is sufficient to prove that it is a faithful rounding of x to $\mathbb{F}_\ell$ and it is odd in $\mathbb{F}_\ell$. As a double rounding is always a faithful rounding, the former holds. Moreover, $x \notin \mathbb{F}_k$ implies that $\square_k(x)$ is odd in $\mathbb{F}_k$ so $\square_k(x) \notin \mathbb{F}_\ell$ then $\square_\ell(\square_k(x))$ is odd in $\mathbb{F}_\ell$. This means that $\square_\ell(\square_k(x))$ is indeed an odd faithful rounding of x to $\mathbb{F}_\ell$, therefore equal to $\square_\ell(x)$. □

Lemma 4. *For all $x \in \mathbb{R}$ and $y \in \mathbb{F}_{\ell+1}$, $\square_\ell(x) + y = \square_\ell(x+y)$.*

Proof. The idea is quite simple: as $y \in \mathbb{F}_{\ell+1}$, it is even with LSB ℓ. Therefore adding it to an odd rounding makes an odd rounding of the sum. Getting a clean proof is slightly more complicated as many subcases have to be tackled.

If $x \in \mathbb{F}_\ell$, then $\square_\ell(x) = x$. We also have that $x + y \in \mathbb{F}_\ell$, so $\square_\ell(x+y) = x + y$. Therefore, $\square_\ell(x) + y = x + y = \square_\ell(x+y)$.

Let us now assume that $x \notin \mathbb{F}_\ell$, meaning that $\square_\ell(x)$ is odd in $\mathbb{F}_\ell$. Then $\square_\ell(x)+y$ is also odd in $\mathbb{F}_\ell$. There is left to prove, by uniqueness of odd rounding, that $\square_\ell(x) + y$ is a faithful rounding of $x + y$. It is sufficient to prove that $\bigtriangledown_\ell(x+y) \le \square_\ell(x) + y \le \bigtriangleup_\ell(x+y)$.

When $\square_\ell(x) = \bigtriangledown_\ell(x)$, we easily have $\square_\ell(x) + y = \bigtriangledown_\ell(x) + y \le x + y \le \bigtriangleup_\ell(x+y)$. The other inequality is slightly more complicated to get: we want to prove that $\bigtriangledown_\ell(x+y) \le \bigtriangledown_\ell(x) + y$, that is $\bigtriangledown_\ell(x+y) - y \le \bigtriangledown_\ell(x)$. By the properties of the rounding down of x, it is enough to prove that $\bigtriangledown_\ell(x+y) - y \le x$ and $\bigtriangledown_\ell(x+y) - y \in \mathbb{F}_\ell$, and both easily hold. The case where $\square_\ell(x) = \bigtriangleup_\ell(x)$ is similar. □

Corollary 5. *For all $x, y \in \mathbb{R}$ and $k, \ell \in \mathbb{Z}$, if $y \in \mathbb{F}_\ell$ and $k < \ell$ then $\square_\ell(\square_k(x) + y) = \square_\ell(x+y)$.*

Proof. As $y \in \mathbb{F}_\ell$ with $\ell > k$, then $y \in \mathbb{F}_{k+1}$. Therefore:

$$\begin{aligned} \square_\ell(\square_k(x) + y) &= \square_\ell(\square_k(x+y)) && \text{(Lemma 4)} \\ &= \square_\ell(x+y) && \text{(Lemma 3)} \end{aligned}$$

□

Corollary 6. *For all $x, y \in \mathbb{R}$ and $k, \ell, \ell_y \in \mathbb{Z}$, if $y \in \mathbb{F}_{\ell_y}$ and $k < \ell$ and $k < \ell_y$ then $\square_\ell(\square_k(x) + y) = \square_\ell(x+y)$.*

Proof.

- If $\ell \le \ell_y$ then $y \in \mathbb{F}_\ell$ and we conclude using Corollary 5.
- If $\ell > \ell_y$ then:

$$\begin{aligned} \square_\ell(\square_k(x) + y) &= \square_\ell(\square_{\ell_y}(\square_k(x) + y)) && \text{(Lemma 3)} \\ &= \square_\ell(\square_{\ell_y}(x+y)) && \text{(Corollary 5)} \\ &= \square_\ell(x+y) && \text{(Lemma 3)} \end{aligned}$$

□

B. Proof of the general algorithm

We want to prove that the general algorithm presented in Section III-A computes a correct rounding of the sum of the inputs. Variables k_i and t_i and output *result* are defined by Algorithm 1 as illustrated by Fig. 2.

Lemma 7. *For all* $0 \le i \le n-1$*,* $t_i = \square_{k_i}(x_0+x_1+\cdots+x_i)$*.*

Proof. We proceed by induction over i.

- For $i = 0$ we get $t_0 = \square_{k_0}(x_0)$ by construction.
- For $0 \le i \le n-2$ we have:

$$\begin{aligned} t_{i+1} &= \square_{k_{i+1}}(t_i + x_{i+1}) \\ &= \square_{k_{i+1}}(\,\square_{k_i}(x_0 + \cdots + x_i) + x_{i+1}) \quad \text{(I.H.)} \\ &= \square_{k_{i+1}}(x_0 + \cdots + x_i + x_{i+1}) \end{aligned}$$

  using Corollary 6: indeed $x_{i+1} \in \mathbb{F}_{\ell_{i+1}}$ and by construction $k_i = \min(k_{i+1}, \ell_{i+1}) - 1$ so $k_i < k_{i+1}$ and $k_i < \ell_{i+1}$.

$\square$

Theorem 8. *The output of the general algorithm verifies:*

$$\textit{result} = \bigcirc_{\ell_f}(x_0 + \cdots + x_{n-1}).$$

Proof. Using Lemma 7 for $i = n-1$ then Corollary 2 with $k_{n-1} = \ell_f - 2$:

$$\begin{aligned} \textit{result} = \bigcirc_{\ell_f}(t_{n-1}) &= \bigcirc_{\ell_f}(\,\square_{k_{n-1}}(x_0 + \cdots + x_{n-1})) \\ &= \bigcirc_{\ell_f}(x_0 + \cdots + x_{n-1}) \end{aligned}$$

$\square$

C. Proof of the optimized algorithm

Let us now prove the correctness of the optimized algorithm presented in Section III-B. First of all, we must show that the pre-processing described by Algorithm 2 has the right properties. Its inputs are once again $x_0, \ldots, x_{n-1}$ with respective LSBs $\ell_0, \ldots, \ell_{n-1}$. But here, we require the LSBs to have been sorted: $\ell_0 \le \cdots \le \ell_{n-1}$. Pre-processing can modify the x_j and ℓ_j and even the number of inputs n. However, the fact that $x_j \in \mathbb{F}_{\ell_j}$ is obviously preserved through each step of the loop. Each step also preserves the sum $x_0+\cdots+x_{n-1}$. Indeed, all the x_j are preserved (though indices get shifted down for $j > i+1$) except for x_i and x_{i+1}, which are replaced by a new x_i with value $x_i \mathbin{\triangledown\mkern-12mu+}_{\ell_i} x_{i+1}$. This is where the LSBs being sorted is important: as $\ell_i \le \ell_{i+1}$, both x_i and x_{i+1} are in $\mathbb{F}_{\ell_i}$, so the addition is exact. Finally, exiting the loop means that $\ell_0 < \cdots < \ell_{n-1}$ at the end.

From now on, we just assume that we have inputs $x_0, \ldots, x_{n-1}$ with respective LSBs $\ell_0 < \cdots < \ell_{n-1}$. We want to prove that the *result* of Algorithm 3 (illustrated by Fig. 3) is a correct rounding to ℓ_f of the sum of the inputs. Recall that m is chosen such that $\ell_{m-1} \le \ell_f - 2 < \ell_m$. With the convention that $\ell_{-1} = -\infty$ and $\ell_n = +\infty$, this means that m is well defined and verifies $0 \le m \le n$.

Lemma 9. *For all* $0 \le i \le m-2$*,* $t_i = \square_{\ell_{i+1}-1}(x_0+\cdots+x_i)$*.*

Proof. As in the proof of Lemma 7, by induction over i:

- For $i = 0$ we get $t_0 = \square_{\ell_1-1}(x_0)$ by construction.
- For $0 \le i \le m-3$ we have:

$$\begin{aligned} t_{i+1} &= \square_{\ell_{i+2}-1}(t_i + x_{i+1}) \\ &= \square_{\ell_{i+2}-1}(\,\square_{\ell_{i+1}-1}(x_0 + \cdots + x_i) + x_{i+1}) \quad \text{(I.H.)} \\ &= \square_{\ell_{i+2}-1}(x_0 + \cdots + x_i + x_{i+1}) \end{aligned}$$

  using Corollary 6 (indeed $\ell_{i+1} - 1 < \ell_{i+2} - 1$ since the LSBs are strictly increasing; and of course $\ell_{i+1} - 1 < \ell_{i+1}$ with $x_{i+1} \in \mathbb{F}_{\ell_{i+1}}$).

$\square$

Lemma 10. *If* $m \ge 1$ *then* $t_{m-1} = \square_{\ell_f-2}(x_0+\cdots+x_{m-1})$*.*

Proof. If $m \ge 2$, we use Lemma 9 for $i = m-2$ then Corollary 6 with $x_{m-1} \in \mathbb{F}_{\ell_{m-1}}$ and $\ell_{m-1} - 1 < \ell_{m-1} \le \ell_f - 2$:

$$\begin{aligned} t_{m-1} &= \square_{\ell_f-2}(t_{m-2} + x_{m-1}) \\ &= \square_{\ell_f-2}(\,\square_{\ell_{m-1}-1}(x_0 + \cdots + x_{m-2}) + x_{m-1}) \\ &= \square_{\ell_f-2}(x_0 + \cdots + x_{m-2} + x_{m-1}) \end{aligned}$$

If $m = 1$, recall that Line 5 of Algorithm 3 overrides Line 2 so that $t_0 = \square_{\ell_f-2}(x_0)$. $\square$

Lemma 11. *If* $m \le n-1$ *then* $t_m = x_m + \cdots + x_{n-1} \in \mathbb{F}_{\ell_m}$*.*

Proof. By induction over i decreasing from $n-1$ to m, it is easy to prove that $t_i \in \mathbb{F}_{\ell_i}$, the additions $t_i \mathbin{\triangledown\mkern-12mu+}_{\ell_{i-1}} x_{i-1}$ are all exact and $t_i = x_i + x_{i+1} + \cdots + x_{n-1}$. $\square$

Theorem 12. *The output of the algorithm verifies:*

$$\textit{result} = \bigcirc_{\ell_f}(x_0 + \cdots + x_{n-1}).$$

Proof. Let us assume for now that $1 \le m \le n-1$, so that we can use both Lemma 10 and Lemma 11 (the special cases $m = 0$ and $m = n$ are handled further below). A corollary of Lemma 10 is that $t_{m-1} \in \mathbb{F}_{\ell_f-2}$. Lemma 11 states that $t_m \in \mathbb{F}_{\ell_m}$. By construction of m we have $\ell_f - 2 < \ell_m$ so $t_m \in \mathbb{F}_{\ell_f-2}$ and the addition $t_{m-1} \mathbin{\triangledown\mkern-12mu+}_{\ell_f-2} t_m$ is exact. This also means that $\ell_f - 1 \le \ell_m$ so $t_{m-1} \in \mathbb{F}_{\ell_f-1}$, which allows us to use Lemma 4 thereafter.

$$\begin{aligned} \textit{result} &= \bigcirc_{\ell_f}(t_{m-1} \mathbin{\triangledown\mkern-12mu+}_{\ell_f-2} t_m) = \bigcirc_{\ell_f}(t_{m-1} + t_m) \\ &= \bigcirc_{\ell_f}(\,\square_{\ell_f-2}(x_0 + \cdots + x_{m-1}) + t_m) \quad \text{(Lemma 10)} \\ &= \bigcirc_{\ell_f}(\,\square_{\ell_f-2}(x_0 + \cdots + x_{m-1} + t_m)) \quad \text{(Lemma 4)} \\ &= \bigcirc_{\ell_f}(\,\square_{\ell_f-2}(x_0 + \cdots + x_{m-1} + x_m + \cdots + x_{n-1})) \\ &\qquad \text{(Lemma 11)} \\ &= \bigcirc_{\ell_f}(x_0 + \cdots + x_{n-1}) \quad \text{(Corollary 2)} \end{aligned}$$

Let us now handle the special cases $m = 0$ and $m = n$. As mentioned in Section III-B, any t_i for i outside of the range $0, \ldots, n-1$ is considered to be null.

When $m = 0$, Lemma 11 still gives us $t_m \in \mathbb{F}_{\ell_m}$ so $t_m \in \mathbb{F}_{\ell_f-2}$ then:

$$\begin{aligned} \textit{result} &= \bigcirc_{\ell_f}(0 \mathbin{\triangledown\mkern-12mu+}_{\ell_f-2} t_m) = \bigcirc_{\ell_f}(t_m) \\ &= \bigcirc_{\ell_f}(x_m + \cdots + x_{n-1}) \quad \text{(Lemma 11)} \\ &= \bigcirc_{\ell_f}(x_0 + \cdots + x_{n-1}) \end{aligned}$$

When $m = n$, we still have $t_{m-1} \in \mathbb{F}_{\ell_f - 2}$ and:

$$\begin{aligned}
\textit{result} &= \bigcirc_{\ell_f}(t_{m-1} \triangledown_{\ell_f-2} 0) = \bigcirc_{\ell_f}(t_{m-1}) \\
&= \bigcirc_{\ell_f}(\square_{\ell_f-2}(x_0 + \cdots + x_{m-1})) \quad \text{(Lemma 10)} \\
&= \bigcirc_{\ell_f}(x_0 + \cdots + x_{m-1}) \quad \text{(Corollary 2)} \\
&= \bigcirc_{\ell_f}(x_0 + \cdots + x_{n-1})
\end{aligned}$$

□

V. Comparison with other algorithms

Let us now give some intuition about the cost of this algorithm, compared to the literature. Note this is a very approximated cost, guessing the number of and/or gates involved. A first approximation is that we do not consider multiplications: we only consider additions. One reason is that both Kulisch and our algorithm compute exactly all multiplications. This notably disfavors the faithful algorithm that is allowed to round the multiplication and therefore drop bits.

A second approximation is that we are not taking overflow into account: all algorithms use a MSB sufficient to ensure that no overflow will occur. If the x_i have a MSB worth m_i, we may use as computation MSB: $m_f = \max_i m_i + (n-1)$. This could be improved if we have prior knowledge, but it will be sufficient for this rough analysis.

The chosen cost model is as follows: for the exact sum of two values x_1 and x_2 with LSBs ℓ_1 and ℓ_2 with $\ell_1 \leq \ell_2$, the cost is $2\,(m_f - \ell_1 + 2)$. When a subsequent rounding to LSB ℓ is done, the additional cost is 0 for rounding down (assuming two's complement arithmetic); it is $\ell - \ell_1$ for rounding to odd (as it only needs an or of the trailing bits, see Section II-A); and it is $m_f - \ell_1$ for rounding to nearest as there may be a carry.

The Kulisch accumulator [2] is a sum of all x_i computed with LSB ℓ_0, followed by a rounding to nearest. (In this section, we assume the input LSBs to be increasing, though they do not have to be stricly increasing.) The cost is then:

$$\mathcal{C}_K = (n-1) * 2\,(m_f - \ell_0 + 2) + (m_f - \ell_0).$$

The faithful algorithm [3] is also a sum of all x_i with the same LSB. Instead of ℓ_0, the computation LSB is $\ell_c = \ell_f - \lceil \log(n) \rceil - 1$, followed by a rounding to nearest. The cost is then:

$$\mathcal{C}_F = (n-1) * 2\,(m_f - \ell_c + 2) + (m_f - \ell_c).$$

Our Algorithm 3 based on odd rounding has a more complex cost. First, sorting the numbers by their LSB has cost 0 as the LSBs are known in advance. Then, let us consider the addition $t_{i-1} \boxplus_{\ell_{i+1}-1} x_i$ for $1 \leq i \leq n-1$ (with $\ell_n = \ell_f - 1$ by convention). The LSB of t_{i-1} is $\ell_i - 1$ and the LSB of x_i is ℓ_i. Then then cost of this addition is $2(m_f - (\ell_i - 1) + 2) + (\ell_{i+1} - 1 - (\ell_i - 1)) = 2(m_f - \ell_i + 3) + (\ell_{i+1} - \ell_i)$. The cost is then:

$$\begin{aligned}
\mathcal{C}_O &= \left(\sum_{i=1}^{n-1} (2(m_f - \ell_i + 3) + (\ell_{i+1} - \ell_i))\right) + (m_f - (\ell_f - 2)) \\
&= \left(\sum_{i=1}^{n-1} 2(m_f - \ell_i + 3)\right) + \ell_n - \ell_0 + m_f - (\ell_f - 2) \\
&= \left(\sum_{i=1}^{n-1} 2(m_f - \ell_i + 3)\right) + m_f - \ell_0 + 1.
\end{aligned}$$

Note that we have considered the worst case where we only have rounding to odd: the cases of equal LSBs and of large LSBs that involve correct rounding (that will be computed with rounding down) have a smaller cost.

An abstract view of the previous formulas is given by Fig. 8 and Fig. 9: they exhibit the bits involved in each of the three algorithms. The cost of the Kulisch accumulator is roughly the area of the bigger rectangle - - with width $m_f - \ell_0$. The cost of the faithful algorithm is the area of the smaller rectangle --- with width $m_f - \ell_c$. The cost of our algorithm is the area of the roughly trapezoidal region —— delimited on the right hand side by the LSBs $\ell_{i+1} - 1$ used in intermediary computations.

VI. Conclusion

We have presented an efficient algorithm for computing correctly-rounding dot products in the context of two's complement fixed-point arithmetic. Note that it easily applies to sums too: it is basically a summation algorithm that takes correct results of multiplications.

As explained, it is clearly more efficient than Kulisch accumulator (even if this effiency depends upon the input

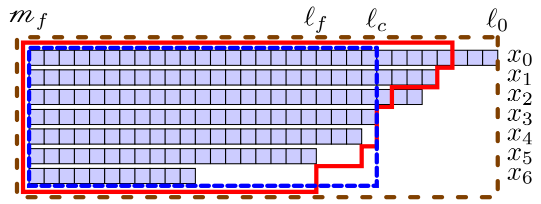

Fig. 8: Toy example: bit view and comparison of the involved bits in the Kulisch accumulator - -, the faithful algorithm ---, and our algorithm ——.

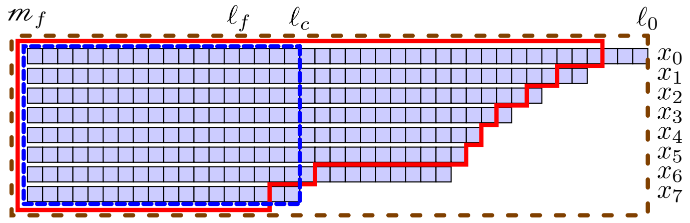

Fig. 9: Signal Processing example: bit view and comparison of the involved bits in the Kulisch accumulator - -, the faithful algorithm ---, and our algorithm ——.

LSBs). Moreover, it is more accurate than the classical faithful rounding algorithms. Indeed, correct rounding means more accuracy (the error bound is twice as small). This could be useful in further computations. More importantly, the result is now unique, allowing reproducibility of the computation.

An immediate application of this algorithm is the sum of products used inside numerical filters. Relying on this algorithm inside numerical filters iterations will improve the accuracy of the computations. It could be used for the word-length allocation problem where the word-lengths are minimized while guarantying a final accuracy [13].

Another perspective would be the formal proof of this algorithm. Most of the basic ingredients are already available in the Flocq library [14] (odd rounding, fixed-point algorithm, Theorem 1), but other lemmas are not. Filters have already been formalized in a previous work [15]. Then, the formal description of the algorithm(s) will need to be done with care, to be as readable as possible, in order to be as convincing as possible. And then this proof will allow the formal proof of more larger implementations like filters or controllers.

A more challenging perspective is the handling of overflow. We assumed here that fixed-point numbers are big enough (in the sense of a big enough MSB) in all our operations, but this has a cost. Two directions are possible. As for LSBs, we could provide correct MSBs for each operation to ensure no overflow may happen. We could also try to have smaller MSBs inside the computations by using modulo overflows if we know in advance a maximal MSB for the final result. Both directions would need careful proofs, or even formal ones, as overflow proofs are quite error-prone.

Acknowledgments

We are thankful to the anonymous referees for their remarks and contructive suggestions.

References

[1] G. Constantinides, P. Cheung, and W. Luk, “The multiple wordlength paradigm,” in *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[2] G. Bohlender and U. Kulisch, “Comments on fast and exact accumulation of products,” in *Applied Parallel and Scientific Computing*, K. Jónasson, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 148–156.

[3] A. Volkova, M. Istoan, F. De Dinechin, and T. Hilaire, “Towards hardware iir filters computing just right: Direct form i case study,” *IEEE Transactions on Computers*, vol. 68, no. 4, pp. 597–608, April 2019.

[4] V. Lefèvre and J.-M. Muller, “Worst cases for correct rounding of the elementary functions in double precision,” in *Proceedings of the 15th Symposium on Computer Arithmetic*, N. Burgess and L. Ciminiera, Eds., Vail, Colorado, 2001, pp. 111–118. [Online]. Available: http://www.computer.org/proceedings/arith/1150/1150toc.htm

[5] T. Ogita, S. Rump, and S. Oishi, “Accurate sum and dot product,” *SIAM Journal on Scientific Computing*, vol. 26, pp. 1955–1988, 01 2005.

[6] J. v. Neumann, “First draft of a report on the edvac,” the United States Army Ordinance Department and the University of Pennsylvania Moore School of Electrical Engineering, Tech. Rep., 1945.

[7] A. W. Burks, H. H. Goldstine, and J. von Neumann, “Preliminary discussion of the logical design of an electronic computing instrument (1946),” 1946, p. Published in 1989.

[8] D. Goldberg, “What every computer scientist should know about floating point arithmetic,” *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–47, 1991. [Online]. Available: http://doi.acm.org/10.1145/103162.103163

[9] S. Boldo and G. Melquiond, “Emulation of FMA and Correctly-Rounded Sums: Proved Algorithms Using Rounding to Odd,” *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 462–471, 2008. [Online]. Available: http://hal.inria.fr/inria-00080427

[10] “IEEE Standard for Standard SystemC Language Reference Manual,” *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, Jan 2012.

[11] H. Hanselmann, “Implementation of digital controllers - a survey,” *Automatica*, vol. 23, no. 1, pp. 7–32, January 1987.

[12] B. Porat, *A course in digital signal processing*. John Wiley, 1997. [Online]. Available: https://books.google.fr/books?id=ZgYfAQAAIAAJ

[13] T. Hilaire, H. Ouzia, and B. Lopez, “Optimal Word-Length Allocation for the Fixed-Point Implementation of Linear Filters and Controllers,” in *ARITH 2019 - IEEE 26th Symposium on Computer Arithmetic*. Kyoto, Japan: IEEE, Jun. 2019, pp. 175–182. [Online]. Available: https://hal.sorbonne-universite.fr/hal-02393851

[14] S. Boldo and G. Melquiond, *Computer Arithmetic and Formal Proofs: Verifying Floating-point Algorithms with the Coq System*. ISTE Press - Elsevier, Dec. 2017.

[15] D. Gallois-Wong, S. Boldo, and T. Hilaire, “A Coq formalization of digital filters,” in *Intelligent Computer Mathematics*, F. Rabe, W. M. Farmer, G. O. Passmore, and A. Youssef, Eds. Cham: Springer International Publishing, Aug. 2018, pp. 87–103. [Online]. Available: https://hal.inria.fr/hal-01728828