Design-space exploration for the Kulisch accumulator

Yohann Uguen, Florent de Dinechin

▶ **To cite this version:**

Yohann Uguen, Florent de Dinechin. Design-space exploration for the Kulisch accumulator . 2017. hal-01488916v2

HAL Id: hal-01488916

https://hal.science/hal-01488916v2

Preprint submitted on 20 Mar 2017

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

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Design-space exploration for the Kulisch accumulator

Yohann Uguen
Univ Lyon, INSA Lyon, Inria, CITI
F-69621 Villeurbanne, France
Yohann.Uguen@insa-lyon.fr

Florent de Dinechin
Univ Lyon, INSA Lyon, Inria, CITI
F-69621 Villeurbanne, France
Florent.de-Dinechin@insa-lyon.fr

ABSTRACT

Floating-point sums and dot products accumulate rounding errors that may render the result very inaccurate. To address this, Kulisch proposed to use an internal accumulator large enough to cover the full exponent range of floating-point. With it, sums and dot products become exact operations. This idea failed to materialize in general purpose processors, as it was considered to slow and/or too expensive in terms of resources. It may however be an interesting option in reconfigurable computing, where a designer may use use smaller, more resource-efficient floating-point formats, knowing that sums and dot products will be exact. Another motivation of this work is that these exact operations, contrary to classical floating point ones, are associative, which enables better compiler optimizations. This work therefore compares, in the context of modern FPGAs, several implementations of the Kulisch accumulator: three proposed by Kulisch, and two novel ones. These architectures are implemented in a VivadoHLS-compliant C++ generator that is fully customizable. Comparisons targeting Xilinx's Kintex 7 FPGAs show improvement over Kulisch' proposal in both area and speed. In single precision, compared with a naive use of classical operators, the proposed accumulator runs at similar frequency, consumes 10x more resource, but reduces the overall latency of a large dot product by 25x while vastly improving accuracy.

1. INTRODUCTION

Field-Programmable Gate Arrays (FPGAs) have demonstrated their potential as accelerators for many applications. FPGAs may bypass some of the bottlenecks of general purpose processors, such as instruction decoding, register file contention, register spilling, parallelism limitation, restricted datatypes. This freedom of design comes with a more complex programming model, but high-level programming languages are becoming a reliable option for hardware design thanks to recent improvements in the High-Level Synthesis (HLS) tools.

Ironically, the "high level" languages used for these tools are derivatives of C, which is about the lowest possible language for processors, just above assembly code. This entails all sorts of restriction, in particular when computing with real numbers. Then the datatypes of choice are the few IEEE-754 floating-point types [1], and HLS tools strive to comply with the C11 and IEEE-754 standards. Floating-point addition is not associative, which leaves a designer with two choices: either preserve the sequential semantics of the floating-point program being compiled, which prevents many optimisations that an HLS compiler could do; or, enable such optimisations, and obtain numerically different results due to a different accumulation of rounding errors. This article addresses this dilemma in the very common case of sums and sums of products, which is at the core of linear algebra and can become very inaccurate when the size of the vectors increases [9].

It is an active research field. Some previous work address the parallelism issue using standard floating-point adders [14, 7, 16] or modified ones [12, 8, 11]. A variant of the floating-point adder that outputs the rounding error allows to handle also the accuracy issue, but in non-deterministic time [10, 3]. A third line of research, which we follow here, is to use internal formats that depart from traditional floating-point [2, 15, 13, 6].

The present work revisits in this context the exact accumulator design proposed by Kulisch [4]. Its core idea, detailed in Section 2, is to remove all sources of rounding errors in a sum of products. This is achieved thanks to an internal accumulator that contains the entire floating-point range. This is quite area-demanding, but the potential benefits are single-cycle accumulation, more aggressive optimizations, and guaranteed accuracy.

Several implementations of this idea are possible, each with several parameters. Section 2 reviews the implementations initially proposed by Kulisch. Section 3 introduces a novel technique for managing one-cycle accumulation of signed numbers that improves both speed and area over Kulisch' proposal. Section 4 introduces a novel pipelined architecture. Section 5 compares these implementations in a fully customizable generator, and Section 6 concludes.

2. OVERVIEW OF KULISCH PROPOSAL

The main idea of Kulisch was to remove all sources of rounding errors in a dot product. First, one needs an exact multiplier. As reviewed in 2.1 below, it is a classical multiplier with the rounding and normalization logic removed, so is actually simpler a classical floating-point multiplier. Then, the classical floating-point adder is replaced with a very wide fixed-point accumulator whose bits cover the full exponent range of a product. For instance, in double-precision, the weights of the bits in a floating-point mantissa can range from $2^{-1022-53} = 2^{-1075}$ to 2^{1023}. Therefore, the weights of the bits of the exact product of two such mantissa can range

from 2^{-2150} to 2^{2046}. Kulisch even proposed [4] to add even more bits to absorbe possible temporary overflows, leading to 4288 bits of accumulator for double precision.

In the remainder of the article, the accumulator width is denoted w_a. As the previous example shows, the minimum size of w_a can be deduced from the exponent width w_e and significand width w_f of the input format. Table 1 shows the minimum size of exact accumulators for mainstream formats.

In his book, Kulish suggested three ways of implementing the long accumulator [4]. The first uses a long adder and a long shift, and is reviewed in 2.2. It is impractical but for very small formats. The second uses only one short adder, with on-chip RAM containing the segmented accumulator. It is reviewed in 2.4. The third uses small parallel adders, and is reviewed in 2.5.

format	$b_{\min}$	$b_{\max}$	$\min w_a$
binary16 (half)	2^{-24}	2^{15}	80 bits
binary32 (single)	2^{-149}	2^{127}	554 bits
binary64 (double)	2^{-1074}	2^{1023}	4196 bits

Table 1: Accumulator sizes for the IEEE-754 formats

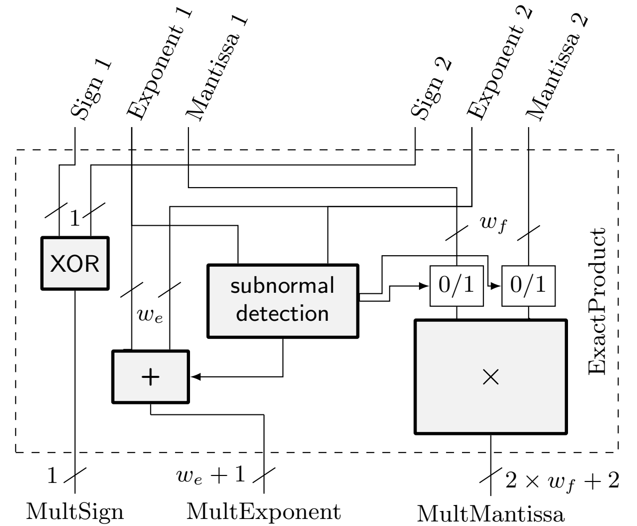

Figure 1: Exact floating-point multiplier

2.1 Exact floating-point product

The architecture used for the exact multiplier is depicted in Figure 1. Each floating-point number is input as sign, exponent and mantissa. The sign of the result is computed by XORing the input signs. The exponents are added and the mantissa multiplied.

Subnormal management adds very little overhead: for each input, if its exponent field is all zeroes, then the implicit bit in the mantissa is set to 0 (otherwise it is set to 1) and a correction of 1 has to be added to the exponent sum.

For input exponents of w_e bits, the exact product requires $w'_e = w_e + 1$ exponent bits. The implicit bit is added to input significands of w_f bits, leading to mantissas of $w_f + 1$ bits. The product of two such mantissas fits on $w'_f = 2 \times w_f + 2$ bits. This is the exact product that can be added, unnormalized (its leading bit may be 0 or 1), to the long accumulator.

A standard floating-point number can trivially be converted to this extended format. Therefore, in all the following, the terms "exponent" and "mantissa" refer to exact product exponent and mantissa.

2.2 Kulisch 1: Long adder and long shift

In this approach, the architecture contains a register of w_a bits and an adder of w_a bits. Every input (floating-point number or exact product) is shifted, according to its exponent, to the correct place, then added to the long accumulator.

This architecture is impractical but for small FP formats, for two reasons. First, it requires a very large shifter. Second, it requires the addition of w_a bits at each cycle, which will be either slow due to carry propagation, or expensive if a fast adder architecture is used. Still, it may be the best choice for very small formats, especially if the context enables small exponent ranges.

2.3 Splitting the accumulator into words

All the other architectures split the long accumulator in words of b bits. There are $N = \lceil w_a/b \rceil$ such words. This decomposes the shift operation in two: select the words to which send a mantissa, and shift within a word. If b is chosen as a power of two, $b = 2^k$, the intra-word shift distance is simply obtained as the k lower bits of the exponent, while the word address is obtained as the $w'_e - k$ leading bits. This is described in Figure 2.

The shifted mantissa will typically be spread across multiple words: at least two, but possibly more. Precisely, after a shift of maximum size $b - 1$, the shifted mantissa is of size $w'_f - 1 + b$, and is spread over $S = \left\lceil \frac{w'_f - 1 + b}{b} \right\rceil$ words.

Such a split has two advantages:

- The two steps of the shift can be executed in parallel;
- Accumulating an input only requires to add it to S words (Figure 2).

However, there are two issues to address. The first is *carry propagation* from one accumulator word to the next. This potentially actually requires to update all the words above the S target words. If done naively, this may either require a long delay, or a variable number of cycles , during which the accumulator is not available for further inputs.

The second is *sign management*: as input mantissa may be signed, they actually need to be either added or subtracted to the accumulator. In case of subtraction the first issue becomes an issue of *borrow propagation*.

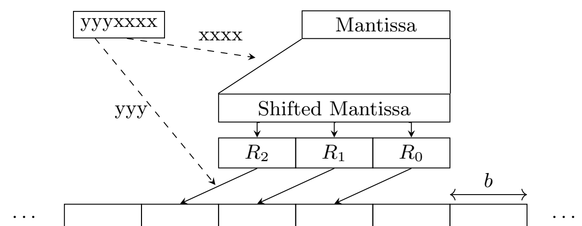

Figure 2: Shifting a mantissa into the sub-words of the long accumulator (here $S = 3$)

2.4 Kulisch 2: Split accumulator in RAM

In this variant, the long-accumulator words are stored in an on-chip RAM.

To address the carry propagation issue, Kulisch uses an array that stores the state of each accumulator word. The state can be one of those three: the word only contains ones (state 1), zeros (state 0) or both (state NA). When a mantissa is added some word number M, the architecture also loads from memory the word M' such that the state of M' is not 1 and $M' > M$ and $!\exists M'', (M'' < M') \wedge (M'' > M) \wedge \mathit{state}(M'' \neq 1)$. This word M' is the one that will receive a carry if produced, in which case the state of all the words between M and M' switch from 1 to 0. Subtractions are handled in a similar way, needing borrow bits in addition to the carry bits.

If the RAM word size is equal to $S \times b$, at most two RAM words are required for each accumulation, plus one RAM word for the carry resolution. This requires a three-port RAM, which is expensive. Note that the three writes always access three different cells, and can therefore be performed concurrently without contention.

The main limitation of this architecture is that it cannot be pipelined. An accumulation consists of RAM reads, additions, and RAM writes, and the reads of a new accumulation cannot start before the writes of the previous one are completed.

2.5 Kulisch 3: Sub-adders with carries and borrows

This architecture, depicted on Figure 3, instantiates one sub-adder for each accumulator word. These N sub-adders (SA) can work in parallel. Besides, two registers receive the carry and borrow values produced by each SA, and transmit it to the next SA, to be consumed at the next cycle. Thus, the carry propagation is delayed. Carries are partly propagated at each cycle, but it takes N cycles at the end of an accumulation (inputting zeroes) to complete the carry propagation. The N additional cycles are not a problem if the number of data to be added is much larger than N. A more expensive alternative would be to use the SA state array discussed in 2.4. It has not been investigated here.

In practice, each mantissa part, tagged with its destination address A, is sent on an accumulator bus (see Figure 3). This bus spans all the SAs and is composed of S lanes. Each SA listens to this bus, and the matching SAs perform their part of the summation in parallel.

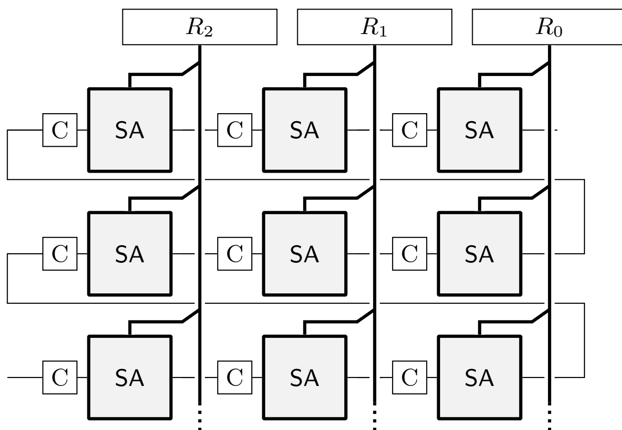

Figure 3: Kulisch 3 accumulator for $S = 3$

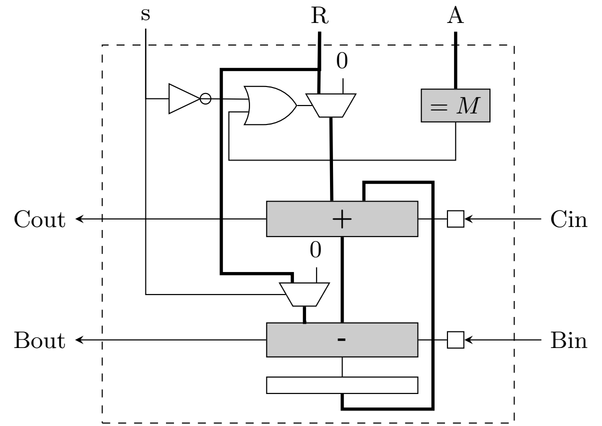

Figure 4: Architecture of a Kulisch SA

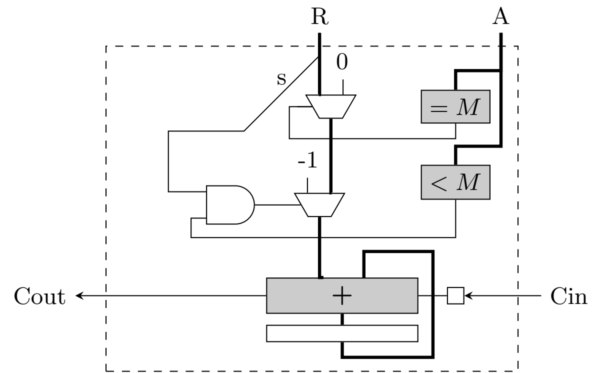

Figure 5: Architecture of a two's complement SA

Actually, the S shifted mantissa words are first rotated to the S lanes of the bus by a bit of modulo-S logic and multiplexing. This then simplifies each SA, which only has to listen to one of the S lanes.

This architecture enables accumulation with 1-cycle latency. The loop critical path that limits the frequency is the subword accumulation. In Kulisch version (depicted Figure 4) this accumulation adds one mantissa sub-word, one carry bit, and one (negative) borrow bit. Therefore, each SA has to perform both one addition and one subtraction in one cycle.

3. PROPOSED ACCUMULATOR 1: SUB-ADDERS IN TWO'S COMPLEMENT

We now introduce an improvement to the architecture described in Section 2.5. Instead of sending the mantissa in sign+magnitude format as in Kulisch proposal, the idea is to send it as a two's complement numbers. This replaces the borrow and carry bits with a single carry bit. It also simplifies the SA (Figure 5), which only requires one binary adder. Essentially, it transfers sign management from the SAs (where it was replicated N times) to a single unit described in Figure 6 directly after the exact multiplier.

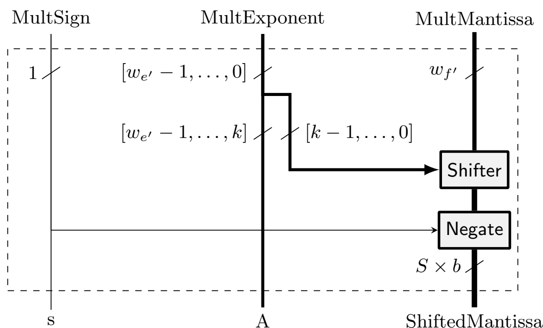

Figure 6: Two's complement conversion before accumulation

In two's complement, a negative number must have its sign extended all the way to the most significant bit of the accumulator. Therefore, when a negative mantissa is sent on the bus, the S SA's with the matching addresses add it to their accumulator sub-word, and all the SA's with higher addresses add -1 (a string of ones) to their sub-word. This performs the sign extension of the two's complement mantissa accross the entire floating-point range.

Before recovering the accumulator values, the carries have to be completely propagated in N cycles as previously. At that point, the combination of the SAs contains the exact result of the computation, but in two's complement.

Therefore, the leading zero counter of the Kulisch proposal is replaced with a leading sign-bit counter, and then the result mantissa has to be negated again.

4. PROPOSED ACCUMULATOR 2: TWO'S COMPLEMENT PIPELINED ARCHITECTURE

This architecture essentially pipelines Kulisch' bus. Again,the accumulator is split into multiple SAs, this time each associated to a pipeline stage. It uses two's complement data as per previous section. Figure 7 gives an overview of this architecture's principle, and Figure 8 describes one pipeline stage.

In Figure 7, the pipeline stages are separated by vertical lines. The horizontal line separates the terms to add and the result of the summation. The dotted arrows represents the data propagation between stages at each clock cycle.

The S words of the shifted mantissa are sent to the special stages to the right (numbered $-S+1$ to 0). The pipeline also transmits the mantissa sign, as well as the stage where the mantissa needs to be added.

The stages between stage 0 and stage $N-S$ are identical. They add the S mantissas in the current SA whose stage match, plus the carry from the previous stage. This carry does not consists of a single bit. Indeed, the result of the sum is computed over $b+\log_2(S+2)$ bits. The b lower bits of the results are kept in the current SA, and the $\log_2(S+2)$ higher bits are the carry sent to next stage.

On both ends of the pipelines, there are $S-1$ stages that are not identical. They performs the same computation but with fewer terms to sum.

Figure 8 gives the details of the central stages. On this figure, M is the address of the stage. It shows that the mantissa words are added if their destination stage match,

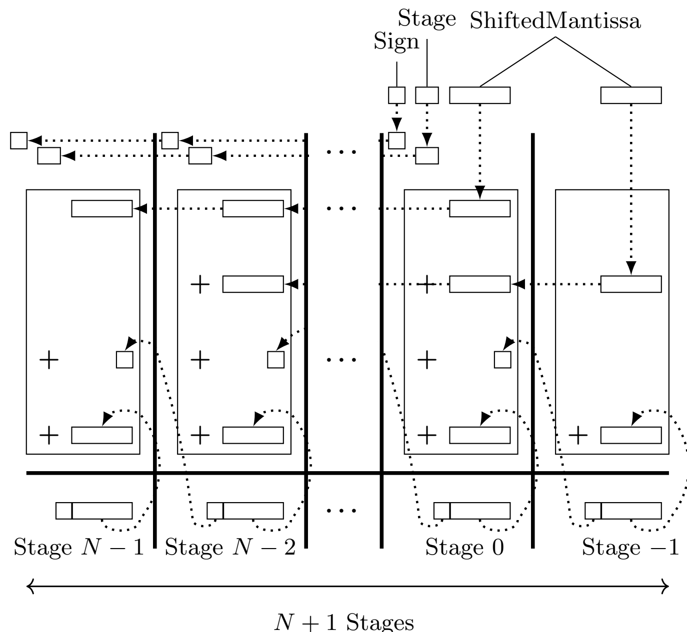

Figure 7: Pipelined accumulator for $S=2$

otherwise the stage adds 0. It also shows how sign extension is managed. The stage matching the leading mantissa word, after adding this mantissa word, replaces it with -1 (all ones in two's complement) and transmits it to the remaining pipeline stages, which will add them to their accumulator sub-words, thus perform the sign extension.

In order to recover the final result, the same method as in previous section is used.

5. EVALUATION OF THE PRESENTED KULISCH ACCUMULATORS

The architectures presented in this paper have as parameters the input/output sizes and the SA size b. Therefore, we wrote VivadoHLS C++ compliant generators for these architectures. For comparison purpose, we also implemented the original Kulisch versions – synthesis results in the litterature target older VLSI technologies, which would have made a direct comparison difficult.

All the results are reported for a VivadoHLS program that computes the accumulation of 1000 numbers, then converts the final value back to floating point. All the numbers presented here were obtained with Vivado 2016.3 after place and route, on a Kintex 7 FPGAs.

For all the split accumulators, a first question was to determine the optimal SA size b. This study is summarized in Figure 9. The optimal size is found to be 64 bits in most cases. More detailed data, including the maximum achieved frequency, is presented in Table 3. The results of Kulisch 1 and Kulisch 2 accumulators are given in Table 2; As expected, the Kulisch 2 version as the lowest resource usage but suffers from a very long latency. The Kulisch 1 accumulator has a lower latency but a very high LUT usage.

One can see that the Proposed 2 accumulator performs the worst although it was designed to be the best. The reports shows that the multiple input sum of this architectures cre-

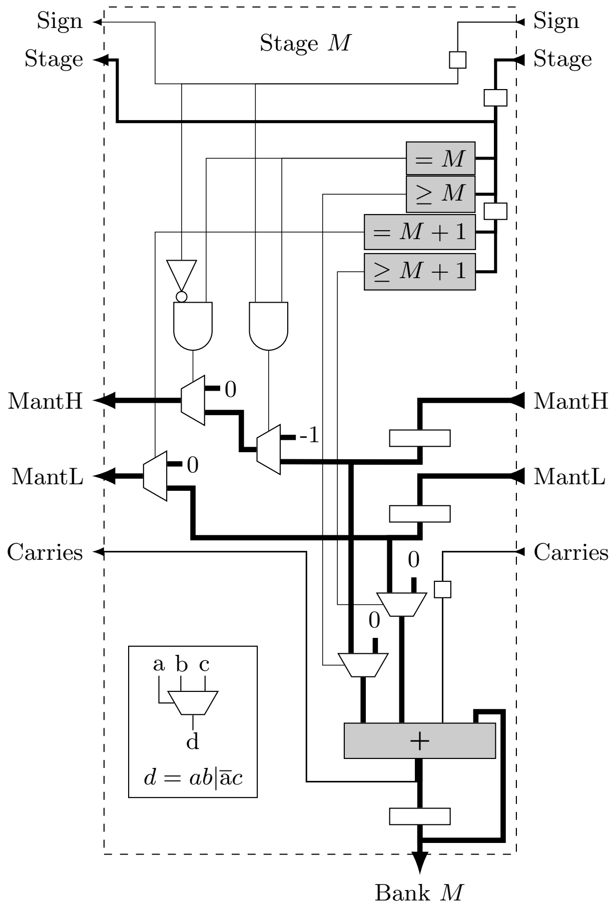

Figure 8: One pipeline stage for $S = 2$

ates a long data-path that slows down the entire architecture. Conversely, the proposed 1 accumulator performs the best, both in terms of frequency and in terms of resources.

There is also a small trade-off between frequency and resource consumption when varying the SA size.

Finally, it is interesting to compare the best accumulator for single precision with the naive alternative of performing the accumulation in double precision. The Proposed 1 accumulator requires 5x the resources, but brings in a 30x latency improvement. Techniques from the literature that attempt to reduce the latency [16] require much more than 5x resources. Besides, the Kulisch accumulator has guaranteed accuracy whatever the number of terms to add.

Similarly, computing with half-float becomes much safer with a long accumulator, actually safer than switching to

	Resources	DSPs	Latency
Kulisch 1	58129L + 59932R	9	9233
Kulisch 2 (SA=64)	2814L + 1919R	9	68173

Table 2: Synthesis results of the Kulisch accumulators from 2.2 and 2.4 for 1000 accumulations with double precision floating-points as inputs

single precision (as long as the small exponent range is not a problem), and for a comparable cost.

In the state of the art, only [3] also addresses both accuracy and performance. Its setup is quite different, but it also requires floating-point memory covering the whole exponent range, with many more shift and normalization steps than in the proposed accumulators. It is therefore even more expensive. It also requires a lot of control, and its actual timing is dependent on the inputs. However, it is able to process in parallel the equivalent of N floating-point numbers in $O(\log N)$ time. Comparing these two approaches in a realistic setup is quite challenging. We hope to do it in the future.

6. CONCLUSION

This paper presented several implementations of the Kulisch accumulator, an expensive but low-latency and highly accurate implementation of sums of products. These architectures allow to compute exact floating-point sums-of-products, whatever the size of the data.

The proposed two's complement implementation of the split architecture can run at higher frequencies for a lower resource usage than Kulisch' original proposal. It is very competitive in terms of performance/cost/accuracy compared to naive use of floating-point.

Further work include integrating this work in an easy-to-use HLS flow. Compressor trees [5] should also be investigated for the pipelined accumulator.

This work could also be evaluated in the context of VLSI circuits, for inclusion in general purpose processors.

7. REFERENCES

[1] *IEEE standard for binary floating-point arithmetic.* Institute of Electrical and Electronics Engineers. Note: Standard 754–1985.

[2] F. de Dinechin, B. Pasca, O. Creţ, and R Tudoran. An FPGA-specific approach to floating-point accumulation and sum-of-products. In *Field-Programmable Technologies*, pages 33–40. IEEE, 2008.

[3] E. Kadric, P. Gurniak, and A. DeHon. Accurate parallel floating-point accumulation. *IEEE Transactions on Computers*, pages 3224–3238, 2016.

[4] U. Kulisch. *Computer arithmetic and validity: Theory, implementation, and applications.* 2013.

[5] M. Kumm and P. Zipf. Pipelined compressor tree optimization using integer linear programming. In *Field Programmable Logic and Applications*, pages 1–8, 2014.

[6] D. U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides. Accuracy-guaranteed bit-width optimization. *Computer-Aided Design of Integrated Circuits and Systems*, pages 1990–2000, 2006.

[7] M. Lin, S. Cheng, and J. Wawrzynek. Cascading deep pipelines to achieve high throughput in numerical reduction operations. In *Reconfigurable Computing and FPGAs*, pages 103–108, 2010.

[8] Z. Luo and M. Martonosi. Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques. *IEEE Transactions on Computers*, pages 208–218, 2000.

[9] J-M. Muller, N. Brisebarre, F. de Dinechin, C-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol,

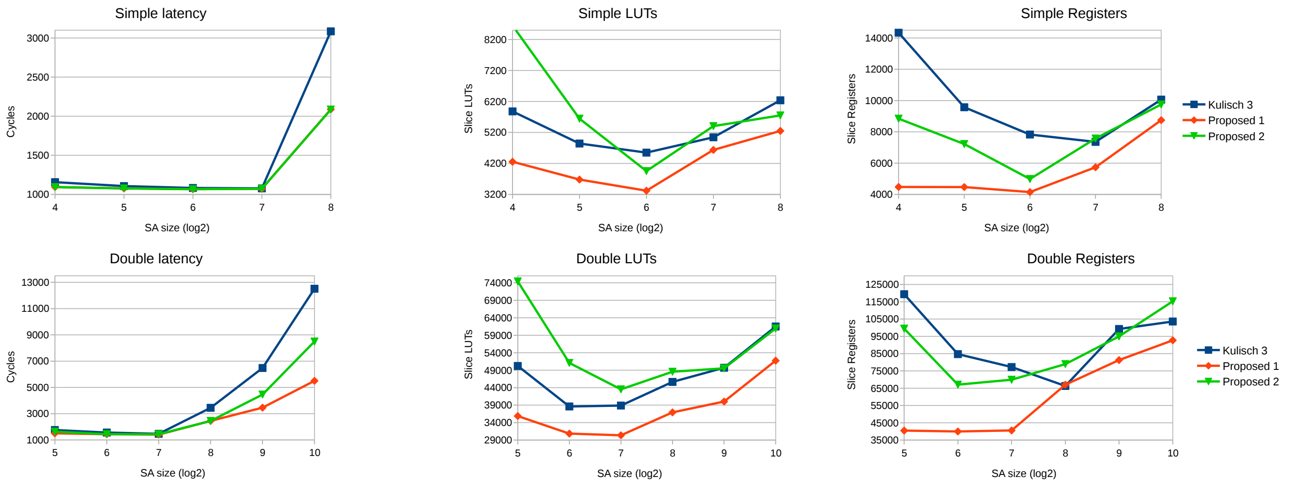

Figure 9: Synthesis results of the Kulisch 3, Proposed 1 and Proposed 2 accumulators for different SA sizes (in logarithmic scale in the figure), for simple and double precision floating-point inputs

Input	Implementation	SA Size	LUTs	Registers	DPSs	Latency	Max Freq
half-floats (16 bits)	Proposed 1	16	841	822	1	1021	363 MHz
single precision (32 bits)	Proposed 1	64	4751	6516	2	1111	344 MHz
	Kulisch 3	64	5162	9760	2	1103	209 MHz
	Proposed 2	32	7774	11960	2	1143	298 MHz
		64	5968	9716	2	1098	297 MHz
single precision (32 bits)	IEEE-754		305	673	9	24218	356 MHz
double precision (64 bits)	Proposed 1	64	42415	72621	9	1795	300 MHz
		128	39590	69826	9	1643	273 MHz
	Kulisch 3	64	36688	76171	9	1493	173 MHz
	Proposed 2	64	58175	82188	9	1707	201 MHz
double precision (64 bits)	IEEE-754		854	1873	14	35317	338 MHz

Table 3: Synthesis results of the presented accumulators for different input sizes for 1000 summands, compared with a dot product using IEEE-754 operators (in gray).

D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[10] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, pages 1955–1988, 2005.

[11] T. Ould Bachir and J. P. David. Performing floating-point accumulation on a modern FPGA in single and double precision. In *Field-Programmable Custom Computing Machines*, pages 105–108, 2010.

[12] A. Paidimarri, A. Cevrero, P. Brisk, and P. Ienne. FPGA implementation of a single-precision floating-point multiply-accumulator with single-cycle accumulation. In *Field Programmable Custom Computing Machines*, pages 267–270, 2009.

[13] O. Sentieys, D. Menard, D. Novo, and K. Parashar. Automatic Fixed-Point Conversion: a Gateway to High-Level Power Optimization. Design Automation and Test in Europe, 2014.

[14] S. Sun and J. Zambreno. A floating-point accumulator for FPGA-based high performance computing applications. In *Field-Programmable Technology*, pages 493–499, 2009.

[15] T. Teufel and M. Baesler. FPGA implementation of a decimal floating-point accurate scalar product unit with a parallel fixed-point multiplier. *Reconfigurable Computing and FPGAs*, pages 6–11, 2009.

[16] D. Wilson and G. Stitt. The unified accumulator architecture: A configurable, portable, and extensible floating-point accumulator. *Reconfigurable Technology and Systems*, pages 21:1–21:23, 2016.