An FPGA-specific Approach to Floating-Point Accumulation and Sum-of-Products

Florent De Dinechin, Bogdan Pasca, Octavian Creţ, Radu Tudoran

AN FPGA-SPECIFIC APPROACH TO FLOATING-POINT ACCUMULATION AND SUM-OF-PRODUCTS

LIP RESEARCH REPORT RR2008-22

Florent de Dinechin, Bogdan Pasca *

LIP (CNRS/INRIA/ENS-Lyon/UCBL)
École Normale Supérieure de Lyon
Université de Lyon
Florent.de.Dinechin@ens-lyon.fr

Octavian Creţ, Radu Tudoran

Computer Science Department
Technical University of Cluj-Napoca
Octavian.Cret@cs.utcluj.ro

ABSTRACT

This article studies two common situations where the flexibility of FPGAs allows one to design application-specific floating-point operators which are more efficient and more accurate than those offered by processors and GPUs. First, for applications involving the addition of a large number of floating-point values, an ad-hoc accumulator is proposed. By tailoring its parameters to the numerical requirements of the application, it can be made arbitrarily accurate, at an area cost comparable for most applications to that of a standard floating-point adder, and at a higher frequency. The second example is the sum-of-product operation, which is the building block of matrix computations. A novel architecture is proposed that feeds the previous accumulator out of a floating-point multiplier without its rounding logic, again improving both area and accuracy. These architectures are implemented within the FloPoCo generator, freely available under the GPL.

1. INTRODUCTION

Most general-purpose processors have included floating-point (FP) units since the late 80s, following the IEEE-754 standard. The feasibility of FP on FPGA was studied long before it became a practical possibility [16, 10, 12]. As soon as the sizes of FPGAs made it possible, many libraries of floating-point operators were published (see [1, 9, 11, 15] among other). FPGAs could soon provide more FP computing power than a processor in single precision [11, 15], then in double-precision [17, 5, 4]. Here single precision (SP) is the standard 32-bit format consisting of a sign bit, 8 bits of exponent and 23 bits of significand (or mantissa), while double-precision (DP) is the standard 64-bit format with 11 bits of exponent and 52 significand bits. Since then, FPGAs have increasingly been used to accelerate scientific, financial and other FP-based computations. This acceleration is essentially due to massive parallelism: Basic FP operators in an FPGA are typically slower than their processor counterparts by one order of magnitude.

Most of the aforementionned applications are very close, from the arithmetic point of view, to their software implementations. They use the same basic operators, although the internal architecture of the operators may be highly optimised for FPGAs [13]. Most published FP libraries are fully parameterisable in significand length and exponent length, but applications that exploit this flexibility are rare [15, 17].

The FloPoCo project[1] studies how the flexibility of the FPGA target can be better exploited in the floating-point realm. In particular, it looks for operators which are radically different from those present in microprocessors. In the present article, such operators are presented for the ubiquitous operation of floating-point accumulation, and applied to sums of products.

All the results in this article are obtained for Virtex4, speedgrade -12, using ISE9.1, and should be reproducible using Xilinx WebPack and FloPoCo, both available at no cost. Very similar results have been obtained for Altera Stratix II. FloPoCo produces portable VHDL, and is designed with the goal of automatic fine-tuning the architectural parameters to the target hardware and frequency.

2. FLOATING-POINT ACCUMULATION

Summing many independent terms is a very common operation. Scalar product, matrix-vector and matrix-matrix products are defined as sums of products. Numerical integration usually consists in adding many elementary contributions.

*This work was partly supported by the XtremeData university programme, the ANR EVAFlo project and the Egide Brâncuşi programme 14914RL.

[1] `www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/`

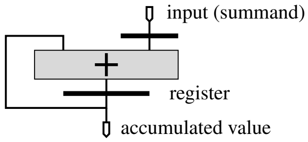

Fig. 1. Iterative accumulator

Monte-Carlo simulations also involve sums of many independent terms. Many other applications involve accumulations of floating-point numbers, and some related work will be surveyed in section 6.

If the number of summands is small and constant, one may build trees of adders, but to accomodate the general case, it is necessary to design an iterative accumulator, illustrated by Figure 1.

It is a common situation that the error due to the computation of one summand is independent of the other summands and of the sum, while the error due to the summation grows with the number of terms to sum. This happens in integration and sum of products, for instance. In this case, it makes sense to have more accuracy in the accumulation than in the summands.

A first idea is to use a standard FP adder, possibly with a larger significand than the summands. The problem is that FP adders have long latencies: typically $l = 3$ cycles in a processor, up to tens of cycles in an FPGA (see Table 1). This is explained by the complexity of their architecture, illustrated on Figure 2.

This long latency means that an accumulator based on an FP adder will either add one number every l cycle, or compute l independent sub-sums which then have to be added together somehow. This will add to the complexity and cost of the application, unless at least l accumulation can be interleaved, which is the case of large matrix operations [18, 2].

In addition, an accumulator built out of a floating-point adder is inefficient, because the significand of the accumulator has to be shifted, sometimes twice (first to align both operands and then to normalise the result, see Figure 2). These shifts are in the critical path of the loop of Figure 1.

In this paper, we suggest to build an accumulator of floating-point numbers which is tailored to the numerics of each application in order to ensure that 1/ its significand never needs to be shifted, 2/ it never overflows and 3/ it eventually provides a result that is as accurate as the application requires. We also show that it can be clocked to any frequency that the FPGA supports. We show that, for many application, the determination of operator parameters ensuring the required accuracy is easy, and that the area can be much smaller for a better overall accuracy. Finally, we combine the proposed accumulator with a modified, errorless FP multiplier to obtain an accurate application-specific dot-product operator.

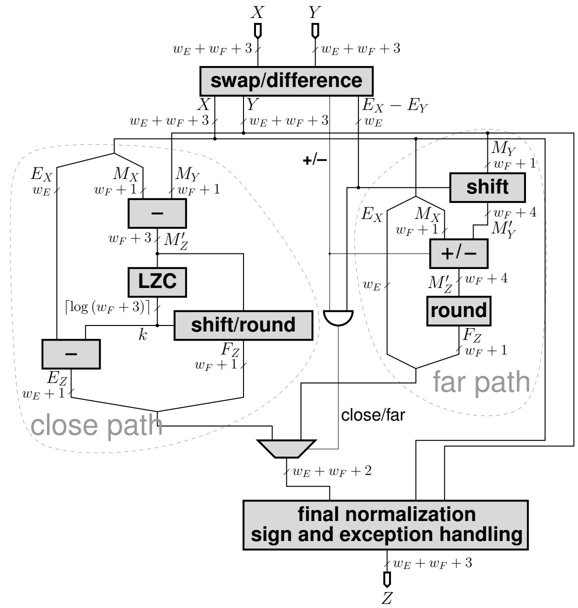

Fig. 2. A typical floating-point adder (w_E and w_F are respectively the exponent and significand sizes)

3. A FAST AND ACCURATE ACCUMULATOR

This section presents the architecture of the proposed accumulator. Section 3.2 will discuss the determination of its many parameters in an application-specific way.

3.1. Overall architecture

The proposed accumulator architecture, depicted on Figure 3, removes all the shifts from the critical path of the loop by keeping the current sum as a large fixed-point number. Figure 4 illustrates the accumulation of several floating-point numbers (represented by their significands shifted by their exponent) into such an accumulator.

There is still a loop, but it is now a fixed-point addition for which current FPGAs are highly efficient. Specifically, the loop involves only the most local routing, and the dedicated carry logic of current FPGAs provides good performance up to 64-bits. For instance, a Virtex4 with speed grade -12 runs such an accumulator at more than 220MHz, while consuming only 64 CLBs. Section 3.3 will show how to reach even larger frequencies and/or accumulator sizes.

The shifters now only concern the summand (see Figure 3), and, being combinatorial, can be pipelined as deep as required by the target frequency.

As seen on Figure 3, the accumulator stores a two's complement number while the summands use a sign/magnitude representation, and thus need to be converted to two's complement. This can be performed without carry propagation:

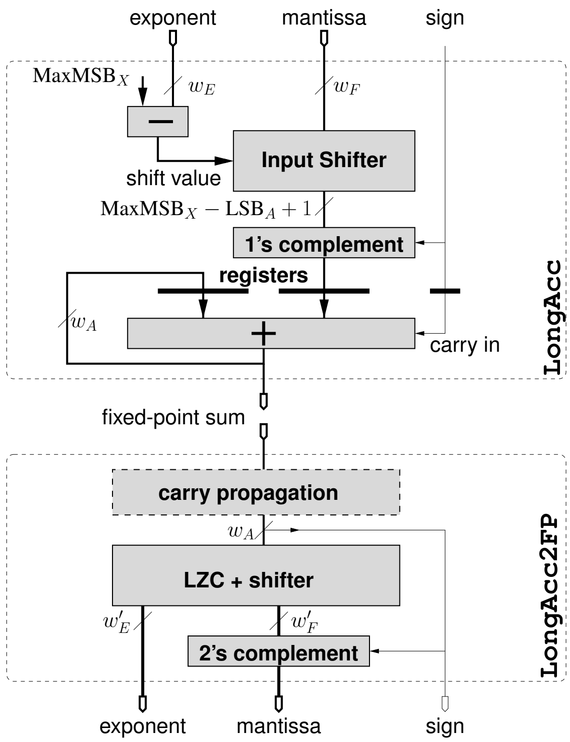

Fig. 3. The proposed accumulator (top) and post-normalisation unit (bottom). Only the registers on the accumulator itself are shown. The rest of the design is combinatorial and can be pipelined arbitrarily.

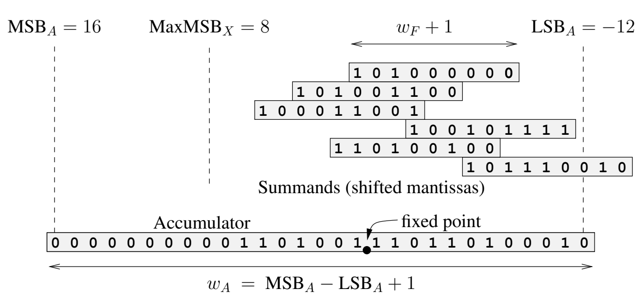

Fig. 4. Accumulation of floating-point numbers into a large fixed-point accumulator

if the input is negative, it is first complemented (fully in parallel), then a 1 is added as carry in to the accumulator. All this is out of the loop's critical path, too.

3.2. Parameterisation of the accumulator

Let us now introduce, with the help of Figure 4, the parameters of this architecture.

- w_E and w_F are the exponent and significand size of the summands
- The weight MSB_A of the most-significant bit (MSB) of the accumulator has to be larger than the maximal expected result, which will ensure that no overflow ever occurs.
- The weight LSB_A of its least-significant bit can be arbitrarily low, and will determine the final accuracy as Section 4 will show.
- For simplicity we note $w_A = \mathrm{MSB}_A - \mathrm{LSB}_A$ the width of the accumulator.
- MaxMSB_X is the maximum expected weight of the MSB of a summand. MaxMSB_X may be equal to MSB_A, but very often one is able to tell that each summand is much smaller in magnitude than the final sum. In this case, providing $\mathrm{MaxMSB}_X < \mathrm{MSB}_A$ will save hardware in the input shifter.

The main claim of the present article is the following: For most applications accelerated using an FPGA, values of MaxMSB_X, MSB_A and LSB_A can be determined a priori, using a rough error analysis or software profiling, that will lead to an accumulator smaller and more accurate than the one based on an FP adder. This claim will be justified in section 4.

This claim sums up the essence of the advantage of FPGAs over the fixed FP units available in processors, GPUs or dedicated floating-point accelerators: we advocate an accumulator specifically tailored for the application to be accelerated, something that would not be possible or economical in an FPU.

3.3. Fast accumulator design using partial carry-save

If the dedicated carry logic of the FPGA is not enough to reach the target frequency, a partial carry-save representation allows to reach any arbitrary frequency supported by the FPGA. As illustrated by Figure 5, the idea is to cut the large carry propagation into smaller chunks of k bits ($k = 4$ on the figure), simply by inserting $\lfloor(\mathrm{MSB}_A - \mathrm{LSB}_A)/k\rfloor$ registers. The critical path is now that of a k-bit addition, and the value of k can therefore be chosen to match the target frequency. This is a classical technique which was in particular

summand (w_E, w_F)	CoreGen FP adder (w_E, w_F)	$2w_F$ accumulator, MaxMSB$_X = 1$	$2w_F$ accumulator, MaxMSB$_X =$ MSB$_A$
(7,16)	304 slices + 1 DSP, 12 cycles @ 359 MHz	129 slices, 8 cycles @ 472 MHz	176 slices, 9 cycles @ 484 MHz
(8,23) SP	317 slices + 4 DSP, 16 cycles @ 450 MHz	165 slices, 8 cycles @ 434 MHz	229 slices, 9 cycles @ 434 MHz
(10,37)	631 slices + 1 DSP, 14 cycles @ 457 MHz	295 slices, 10 cycles @ 428 MHz	399 slices, 11 cycles @ 428 MHz
(11,52) DP	771 slices + 3 DSP, 15 cycles @ 366 MHz	375 slices, 11 cycles @ 414 MHz	516 slices, 12 cycles @ 416 MHz

Table 1. Compared synthesis results for an accumulator based on FP adder, versus proposed accumulator with MSB$_A = w_E$, LSB$_A = -w_E$, all targetted for 400MHz on a VirtexIV.

suggested by Hossam, Fahmy and Flynn [6] for use as an internal representation in processor FPUs. For $k = 1$ one obtains a standard carry-save representation, but larger values of k are prefered as they take advantage of dedicated carry logic while reducing the register overhead. The FloPoCo implementation computes k out of the target frequency. For illustration, $k = 30$ allows to reach 400MHz on VirtexIV. The additional hardware cost is just the few additional registers – $1/4$ more in our figure, and $1/30$ more for 400MHz accumulation.

Of course a drawback of the partial carry-save accumulator is that it holds its value in a non-standard redundant format. To convert to standard notation, there are two options. One is to dedicate $\lfloor(\text{MSB}_A - \text{LSB}_A)/k\rfloor$ cycles at the end of the accumulation to add enough zeroes into the accumulator to allow for carry propagation to terminate. This comes at no hardware cost. The other option, if the running value of the accumulator is needed, is to perform this carry propagation in a pipelined way before the normalisation – this is the dashed box on Figure 3. The important fact is again that this carry propagation is outside of the critical loop.

3.4. Post-normalisation unit, or not

Figure 3 also shows the FloPoCo `LongAcc2FP` component, that performs the conversion of the long accumulator result to floating-point. However, let us first remark that this component is much less useful than the accumulator itself. Indeed, some applications need the running sum at each cycle, but most applications need only the final sum and have no need for all these intermediate sums. Let us take a few examples.

In [3], the FPGA computes a very large integration – several hours– and only the final result is relevant. In such cases, of course, it makes no sense to dedicate hardware to the conversion of the accumulator back to floating-point. FPGA resources will be better exploited at speeding up the computation as much as possible. FloPoCo provides a small helper program to perform this conversion in software.

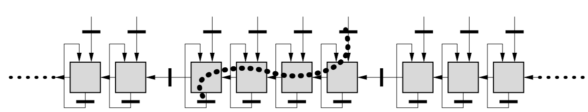

Fig. 5. Accumulator with 4-bit partial carry-save. The boxes are full adders, bold dashes are 1-bit registers, and the dots show the critical path.

Another common case is that a single post-normalisation unit may be shared by several accumulators. For instance, a dot product of vectors of size N accumulates N numbers before needing to convert the result back to floating-point. Therefore, in matrix operations, one pipelined `LongAcc2FP` may be shared between N dot product operators [18].

For these reasons, it makes sense to provide `LongAcc2FP` as a separate component, as on Figure 3. Note that the same holds for an accumulator based on an FP adder of latency l (that actually computes l intermediate subsums). If only the final sum is needed, it may be computed in software at no extra hardware cost. However, if the running sum is needed at each cycle, it will take $l - 1$ additional FP adders (and some registers) to add together the l sub-sums. And finally, such a post-normalisation unit can be shared as in the long accumulator case for matrix operations [18, 2].

To sidestep this discussion, in the following, we choose to compare the performance only for the accumulators themselves. We nevertheless provide for completeness some synthesis results for the post-normalisation unit in Table 2, but the reader should have in mind that comparable extra costs will plague an accumulator based on an FP adder.

Back to `LongAcc2FP`, it mostly consists in leading-zero counting and shifting, followed by conversion from 2's complement to sign/magnitude, and rounding. If the accumulator holds a partial carry-save value, the carries need to be propagated – this simply requires $\lceil w_A/k\rceil$ pipeline levels, each consisting of one k-bit adder and $\lceil w_A/k\rceil - 1$ registers of k bits. Again, all this may be performed at each cycle and pipelined arbitrarily. For an IEEE-754-like correctly-rounded conversion, one must in addition compute a sticky bit out of the discarded bits, but it is yet unclear if users are ready to pay this cost.

3.5. Performance results

Table 1 illustrates the performance of the proposed accumulator compared to one built using a floating-point adder from the Xilinx CoreGen tool.

For each summand size, we build accumulators of twice the size of the input significand ($\text{MSB}_A = w_E$, $\text{LSB}_A = -w_E$) for two configurations: a small one where $\text{MaxMSB}_X = 1$, and a larger one where $\text{MaxMSB}_X = \text{MSB}_A = w_E$. As section 4 will show, a typical accumulator will be between these two configurations. Again, these results are for illustration only: An accumulator should be built in an application-specific way.

Table 2 provides preliminary results for the current implementation of the `LongAcc2FP` post-normalisation unit.

4. APPLICATION-SPECIFIC TUNING OF ACCUMULATOR PARAMETERS

Let us now justify the claim, made in 3.2, that the many parameters of our accumulator are easy to determine on a per-application basis.

4.1. A tale of performance, and also accuracy

First note that a designer has to provide a value for MSB_A and MaxMSB_X, but these values do not have to be accurate. For instance, adding 10 bits of safety margin to MSB_A has no impact on the latency and very little impact on area. Now from the application point of view, 10 bits mean 3 orders of magnitude: it is huge. A designer in charge of implementing a given computation on FPGA is expected to understand it well enough to bound the expected result with a margin of 3 orders of magnitude. An actual example is detailed below in 4.2. As another example, in Monte-Carlo simulations of financial markets, each accumulation computes an estimate of the value of a share. No share will go beyond, say, $100,000 before something happens that makes the simulation invalid anyway.

It may be more difficult to evaluate MaxMSB_X. In doubt, $\text{MaxMSB}_X = \text{MSB}_A$ will do, but the accumulator will be larger than needed, and application knowledge should help reduce it. For instance, implementers of Monte-Carlo simulation will exploit the fact that probabilities are smaller than 1. Another option is profiling: a typical instance of the problem may be run in software, instrumented to output the max and min of the absolute values of summands. Again, the trust in such an approach comes from the possibility of adding 20 bits of margin for safety.

(w_E, w_F)	LongAcc2FP, $2w_F \rightarrow w_F$
(7,16)	142 slices, 6 cycles @ 386 MHz
(8,23) SP	304 slices, 7 cycles @ 366 MHz
(10,37)	546 slices, 10 cycles @ 325 MHz
(11,52) DP	863 slices, 12 cycles @ 306 MHz

Table 2. Synthesis results for a LongAcc2FP compatible with Table 1, rounding an accumulator of size $2w_F$ to an FP number of size w_F.

In some cases, the application will dictate MaxMSB_X but not MSB_A. In this case, one has to consider the number n of terms to add. Again, one will usually be able to provide an upper bound, be it the extreme case of 1 year running at 500MHz, or 2^{53} cycles. In a worst-case scenario on such simulation times, this suggests the relationship $\text{MSB}_A = \text{MaxMSB}_X + 53$ to avoid overflows. For comparison, 53 is the precision of a DP number, so the cost of this worst case scenario is simply a doubling of the accumulator itself, *but not of the input shifter* which shifts up to MaxMSB_X only. It will cost just slightly more than 53 LUTs in the accumulator (although much more in the post-normalisation unit if one is needed).

The last parameter, LSB_A, allows a designer to manage the tradeoff between precision and performance. First, remark that if a summand has its LSB higher than LSB_A (case of the 5 topmost summands on Figure 4), it is added exactly, entailing no rounding error. Therefore, the proposed accumulator will be exact (entailing no roundoff error whatsoever) if the accumulator size is large enough so that its LSB is smaller than those of all the inputs. Conversely, if a summand has an LSB smaller than LSB_A (case of the bottommost summand on Figure 4), adding it to the accumulator entails a rounding error of at most $2^{\text{LSB}_A - 1}$. In the worst case, when adding n numbers, this error will be multiplied by n and invalidate the $\log_2 n$ lower bits of the accumulator – note that this worst-case situation is antagonist to the one leading to the worst-case overflow. A designer may lower LSB_A to absorb such errors, an example is given in next section. Again, a practical maximum is an increase of 53 bits for 1 year of computation at 500MHz. However, in a global sense, it doesn't make much sense to consider this worst case situation, because when a summand is added exactly, it is usually the result of some rounding, so it carries an error of the order of its LSB, which it adds to the accumulator (by not adding bits lower than its LSB). These errors, which are not due to the accumulator, will typically dwarf the rounding errors due to the accumulator. They can be reduced by increasing w_F, but this is of course outside of the scope of this article.

All considered, it is expected that no accumulator will need to be designed larger than 100 bits.

Finally, the proposed accumulator has sticky output bits for overflows in the summands and in the accumulator, so the validity of the result can be checked a posteriori.

	accuracy	area	latency
SP FP adder acc	$1.2 \cdot 10^{-3}$	317 sl, 4 DSP	16 @ 450 MHz
DP FP adder acc	$2.8 \cdot 10^{-15}$	771 sl, 3 DSP	15 @ 366 MHz
proposed acc	$2.0 \cdot 10^{-16}$	247 sl	10 @ 454 MHz

Table 3. Compared performance and accuracy of different accumulators for SP summands from [3].

4.2. A case study

In the inductance computation of [3], physical expertise tells that the sum will be less than 10^5 (using arbitrary units due to factoring out some physical constants), while profiling showed that the absolute value of a summand was always between 10^{-2} and 2.

Converting to bit weight, and adding two orders of magnitude (or 7 bits) for safety in all directions, this defines $\mathrm{MSB}_A = \lceil \log_2(10^2 \times 10^5) \rceil = 24$, $\mathrm{MaxMSB}_X = 8$ and $\mathrm{LSB}_A = -w_F - 15$ where w_F is the significand width of the summands. For $w_F = 23$ (SP), we conclude that an accumulator stretching from $\mathrm{LSB}_A = -23 - 15 = -38$ (least significant bit) to $\mathrm{MSB}_A = 24$ (most significant bit) will be able to absorb all the additions without any rounding error: no summand will add bits lower than 2^{-38}, and the accumulator is large enough to ensure it never overflows. The accumulator size is therefore $w_A = 24 + 38 + 1 = 63$ bits.

Remark that only LSB_A depends on w_F, since the other parameters (MSB_A and MaxMSB_X) are related to physical quantities, regardless of the precision used to simulate them. This illustrates that LSB_A is the parameter that allows one to manage the accuracy/area tradeoff for an accumulator.

4.3. Accuracy measurements

Table 3 compares for accuracy and performance the proposed accumulator to one built using Xilinx CoreGen. To evaluate the accuracies, we computed the exact sum using multiple-precision software on a small run (20,000,000 summands), and the accuracy of the different accumulators was computed with respect to this exact sum. The proposed accumulator is both smaller, faster and more accurate than the ones based on FP adders. This table also shows that for production runs, which are 1000 times larger, a single-precision FP accumulator will not offer sufficient accuracy.

Table 4 provides other examples of the final relative accuracy, with respect to the exact sum, obtained by using an FP adder, and using the proposed accumulator with twice as large a significand. In the first column, we are adding n numbers uniformly distributed in [0,1]. The sum is expected to be roughly equal to $n/2$, which explains that the result becomes very inaccurate for $n = 1,000,000$: as soon as the sum gets larger than 2^{17}, any new summand in [0,1] is simply shifted out and counted for zero. This problem can be anticipated by using a larger significand, or a larger MSBA

sum size	rel. error for unif[0, 1]		rel. error for unif[-1, 1]	
	FP adder	long acc.	FP adder	long acc.
1000	-5.76e-05	1.05e-07	-1.59e-05	1.40e-04
10,000	-2.74e-04	1.07e-08	-3.04e-04	2.36e-04
100,000	-4.31e-04	1.07e-09	2.54e-03	-2.73e-04
1,000,000	-0.738	-3.57e-09	3.18e-03	-4.47e-05

Table 4. Accuracy of accumulation of FP(7,16) numbers, using an FP(7,16) adder, compared to using the proposed accumulator with 32 bits ($MSBA = 20$, $LSBA = -11$).

in the accumulator as we do. In the second column, numbers are uniformly distributed in [-1,1]. The sum grows as well (it is a random walk) but much more slowly. As we have taken a fairly small accumulator ($\mathrm{LSB}_A = -11$), for the first sums floating-point addition is more accurate: while the sum is smaller than 1, its LSB is smaller than -16. However, as more numbers are added, the sum grows. More and more of the bits of a summand are shifted out in the FP adder, but kept in the long accumulator, which becomes more accurate. Note that by adding only 5 bits to it ($\mathrm{LSB}_A = -16$ instead of -11), the relative error becomes smaller than 10^{-10} in all cases depicted in Table 4: Again, LSB_A is the parameter allowing to manage the accuracy/area tradeoff.

We have discussed in this section only the error of the long fixed-point accumulator itself (the upper part of Fig. 3). If its result is to be rounded to an FP(7,16) number using the post-normalisation unit of Figure 3, there will be a relative rounding error of at most $2^{-17} \approx 0.76 \cdot 10^{-5}$. Comparing this value with the relative errors given in Table 4, one concludes that the proposed accumulator, with the given parameters, always leads to a result accurate to the two last bits of an FP(7,16) number.

5. ACCURATE SUM-OF-PRODUCTS

We now extend the previous accumulator to a highly accurate sum-of-product operator. The idea is simply to accumulate the exact results of all the multiplications. To this purpose, instead of standard multipliers, we use *exact* multipliers which return all the bits of the exact product: for $1 + w_F$-bit input significand, they return an FP number with a $2 + 2w_F$-bit significand. Such multipliers incur no rounding error, and are actually *cheaper* to build than the standard (w_E, w_F) ones. Indeed, the latter also have to compute $2w_F + 2$ bits of the result, and in addition have to round it. In the exact FP multiplier, results do not need to be rounded, and do not even need to be normalised, as they will be immediately sent to the fixed-point accumulator. There is an additional cost, however, in the accumulator, whose input shifter is twice as large.

This idea was advocated by Kulisch [8] for inclusion in microprocessors, but a generic DP version requires a 4288 bits accumulator, which manufacturers always considered too costly to implement. On an FPGA, one may design an application-specific version with an accumulator of 100-200 bits only. This is being implemented in FloPoCo, and Table 5 provides preliminary synthesis results for single and double precision input numbers, with the same $2w_F$ accumulators as in Table 1. Some work is still needed to bring the FloPoCo multiplier generator to CoreGen level.

6. COMPARISON WITH RELATED WORK

Luo and Martonosi [14] have described an architecture for the accumulation of SP numbers that uses two 64-bit fixed-point adders. It first shifts the input data according to the 5 lower bits of the exponent, then sends it to one of the fixed-point accumulators depending on the higher exponent bits. If these differ too much, either the incoming data or the current accumulator is discarded completely, just as in an FP adder. This design is much more complex than ours, and the critical path of the accumulator loop includes one 64-bit adder and a 3-2 compressor. The main problem with this approach, however is that it is a fixed design that will not scale beyond single-precision. Another one is that the detection of accumulator overflow may stall the operator, leading to a variable-latency design. The authors suggest a workaround that imposes a limit on the number of summands to add.

He et al [7] have suggested group-alignment based floating-point accumulation. The idea is to first buffer the inputs into groups of size m (with $m = 16$ in the paper), while keeping trace of the largest exponent of a group. Then all the numbers in a group are aligned with this largest-magnitude number, and added in a fixed-point accumulator larger than the significands (30 bits for SP numbers). All this can be run at very high frequency. Then, these partial sums are fed to a final stage of FP accumulation that may run at $1/m$ the frequency of the first stage, and may therefore use a standard unpipelined FP adder. The authors also show that this accumulator is more accurate than one based on a standard FP adder. Again, this is a very complex design (for SP, 443 slices without the last stage, 716 with it). Besides, the frequency of the group accumulator will not scale well to higher precisions without resorting to techniques similar to our partial carry save.

Zhuo and Prasanna [18], then Bodnar et all [2] have described high-throughput matrix operations using standard FP adders scheduled optimally thanks to additional buffers or memories. Performance-wise, this approach should be comparable to ours. The advantage of our approach is its better genericity and its finer control of the accuracy-performance tradeoff.

CoreGen, SP $\times$, SP $+$	484 sl + 8 DSP, 26 cycles @ 366 MHz
ours, SP $\times$, DP acc	*319 sl + 4 DSP, 13 cycles @ 363 MHz*
CoreGen, SP $\times$, DP $+$	973 sl + 7 DSP, 26 cycles @ 366 MHz
CoreGen, DP $\times$, DP $+$	1241 sl + 19 DSP, 37 cycles @ 366 MHz
ours, DP $\times$, 105-bit acc	*1441 sl + 9 DSP, 23 cycles @279 MHz*

Table 5. Preliminary results for Sum-Of-Product operators

7. CONCLUSION AND FUTURE WORK

The accumulator design presented in this article perfectly illustrates the philosophy of the FloPoCo project: Floating-point on FPGA should make the best use of the flexibility of the FPGA target, not re-implement operators available in processors. By doing so, one comes up with a fairly simple design which can be tailored to be arbitrarily faster and arbitrarily more accurate than a naive floating-point approach, without requiring more resources.

This is done in an application-specific manner, and requires the designer to provide bounds on the orders of magnitudes of the values accumulated. We have shown that these bounds can be taken lazily. In return, the designer gets not only improved performance, but also a provably accurate accumulation process. We believe that this return is worth the effort, especially considering the overall time needed to implement a full floating-point application on an FPGA.

The challenge is now to integrate such advanced operators in the emerging C-to-FPGA compilers. A profiling-based approach might be enough to set up the proposed accumulator. In parallel, FloPoCo should be extended with tools that help a designer set up a complete floating-point datapath, managing synchronisation issues but also accuracy ones. Such tools are still at the drawing board.

8. REFERENCES

[1] P. Belanović and M. Leeser. A library of parameterized floating-point modules and their use. In *Field Programmable Logic and Applications*, volume 2438 of *LNCS*, pages 657–666. Springer, 2002.

[2] M. R. Bodnar, J. R. Humphrey, P. F. Curt, J. P. Durbano, and D. W. Prather. Floating-point accumulation circuit for matrix applications. In *FCCM*, pages 303–304. IEEE Computer Society, 2006.

[3] O. Creţ, F. de Dinechin, I. Trestian, R. Tudoran, L. Creţ, and L. Văcariu. FPGA-based acceleration of the computations involved in transcranial magnetic stimulation. In *Southern Programmable Logic Conference*, pages 43–48. IEEE, 2008.

[4] M. deLorimier and A. DeHon. Floating-point sparse matrix-vector multiply for FPGAs. In *Field-Programmable Gate Arrays*, pages 75–85. ACM, 2005.

[5] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. In *Field-Programmable Gate Arrays*, pages 86–95. ACM, 2005.

[6] H. A. H. Fahmy and M. J. Flynn. The case for a redundant format in floating point arithmetic. In *16th Symposium on Computer Arithmetic*, pages 95–102. IEEE Computer Society, 2003.

[7] C. He, G. Qin, M. Lu, and W. Zhao. Group-alignment based accurate floating-point summation on FPGAs. In *ERSA*, pages 136–142, 2006.

[8] U. W. Kulisch. *Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units*. Springer-Verlag, 2002.

[9] B. Lee and N. Burgess. Parameterisable floating-point operators on FPGAs. In *36th Asilomar Conference on Signals, Systems, and Computers*, pages 1064–1068, 2002.

[10] Y. Li and W. Chu. Implementation of single precision floating point square root on FPGAs. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 56–65, 1997.

[11] G. Lienhart, A. Kugel, and R. Männer. Using floating-point arithmetic on FPGAs to accelerate scientific N-body simulations. In *FPGAs for Custom Computing Machines*. IEEE, 2002.

[12] W. Ligon, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. Underwood. A re-evaluation of the practicality of floating-point operations on FPGAs. In *IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.

[13] J. Liu, M. Chang, and C.-K. Cheng. An iterative division algorithm for FPGAs. In *ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 83–89. ACM, 2006.

[14] Z. Luo and M. Martonosi. Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques. *IEEE Transactions on Computers*, 49(3):208–218, 2000.

[15] E. Roesler and B. Nelson. Novel optimizations for hardware floating-point units in a modern FPGA architecture. In *Field Programmable Logic and Applications*, volume 2438 of *LNCS*, pages 637–646. Springer, 2002.

[16] N. Shirazi, A. Walters, and P. Athanas. Quantitative analysis of floating point arithmetic on FPGA based custom computing machine. In *FPGAs for Custom Computing Machines*, pages 155–162. IEEE, 1995.

[17] L. Zhuo and V. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on FPGAs. In *Reconfigurable Architecture Workshop, Intl. Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2004.

[18] L. Zhuo and V. K. Prasanna. High performance linear algebra operations on reconfigurable systems. In *ACM/IEEE conference on Supercomputing*. IEEE, 2005.