

When FPGAs are better at floating-point than microprocessors

Florent de Dinechin, Jérémie Detrey, Octavian Creţ, Radu Tudoran

▶ **To cite this version:**

Florent de Dinechin, Jérémie Detrey, Octavian Creţ, Radu Tudoran. When FPGAs are better at floating-point than microprocessors. 2007. ensl-00174627

HAL Id: ensl-00174627

https://ens-lyon.hal.science/ensl-00174627

Preprint submitted on 24 Sep 2007

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.

When FPGAs are better at floating-point than microprocessors

Florent de Dinechin, Jérémie Detrey
LIP, École Normale Supérieure de Lyon
(CNRS/INRIA/ENS-Lyon/Univ. Lyon 1)
Florent.de.Dinechin@ens-lyon.fr

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

LIP research report RR2007-40

Abstract

It has been shown that FPGAs could outperform high-end microprocessors on floating-point computations thanks to massive parallelism. However, most previous studies re-implement in the FPGA the operators present in a processor. This is a safe and relatively straightforward approach, but it doesn't exploit the greater flexibility of the FPGA. This article is a survey of the many ways in which the FPGA implementation of a given floating-point computation can be not only faster, but also more accurate than its microprocessor counterpart. Techniques studied here include custom precision, specific accumulator design, dedicated architectures for coarser operators which have to be implemented in software in processors, and others. A real-world biomedical application illustrates these claims. This study also points to how current FPGA fabrics could be enhanced for better floating-point support.

1 Introduction

Floating-point (FP) is mostly a commodity: In theory, any application processing real numbers, after a careful analysis of its input, output and internal data, could be implemented using only fixed-point arithmetic. For most applications (not all), such a fixed-point implementation would even be more efficient than its floating-point counterpart, for the floating-point operators are much more complex and costly than the fixed-point equivalents. Unfortunately, there is currently no hope to fully automate the transformation of a computation on real numbers into a fixed-point program. This requires specific expertise and may be very tedious when done by hand. The floating-point representation solves this problem by dynamically adapting the number representation to the order of magnitude of the data. It may not be economical in terms of hardware resources or latency, but it becomes so as soon as design effort is taken into account. This explains why most general-purpose processors have included floating-point units since the late 80s.

Feasibility of FP on FPGA was studied long before it became a practical possibility [37, 27, 29]. The turning point was the beginning of the century: Many libraries of floating-point operators were published almost simultaneously (see [32, 24, 28, 36] among other). The increase of capacity of FPGAs soon meant that they could provide more FP computing power than a processor in single precision [32, 28, 36], then in double-precision [41, 16, 8]. Here single precision (SP) is the standard 32-bit format consisting of a sign bit, 8 bits of exponent and 23 mantissa bits, while double-precision (DP) is the standard 64-bit format with 11 bits of exponent and 52 mantissa bits. Since then, FPGAs have been increasingly used to accelerate scientific, financial or other FP-based computations. This performance is essentially due to massive parallelism, as basic FP operators in an FPGA are typically slower than their processor counterparts by one order of magnitude. This is the intrinsic performance gap between reconfigurable logic and the full-custom VLSI technology used to build the processor's FP unit.

Most of the aforementionned applications are very close, from the arithmetic point of view, to their

microprocessor counterpart. They use the same basic operators, although the internal architecture of the operators may be highly optimized for FPGAs [30]. Besides, although most published FP libraries are fully parameterizable in mantissa length and exponent length, applications rarely exploit this flexibility: with a few exceptions [36, 41], all of them use either the SP or the DP formats. None, to our knowledge, uses original FP operators designed specifically for FPGA computing.

This article is a survey on how the flexibility of the FPGA target can be better employed in the floating-point realm. Section 2 describes a complete application used here as a running example. Section 3 discusses mixing and matching fixed- and floating-point of various precisions. Section 4 discusses the issue of floating-point accumulation, showing how to obtain faster and more accurate results thanks to operators that are very different from those available in processor FPUs. Section 5 surveys several coarser operators (euclidean norm, elementary functions, etc) that are ubiquitous enough to justify that specific architectures are designed for them. The resulting operators may be smaller, faster and more accurate than a combination of basic library operators. The last section concludes and discusses how current FPGA hardware could be enhanced for better FP support.

This article is mostly a prospective survey, however it introduces and details several novel operators in Section 4 and Section 5. Another transversal contribution is to show how to make the best use of existing operators thanks to back-of-the-enveloppe error analysis.

2 Running example: inductivity computation

Let us now briefly describle the application (described in more details in [40]) that motivated this work. It consists in computing the inductance of a set of coils used for the magnetic stimulation of the nervous system. Each coil consists of several layers of spires, each spire is divided in segments (see Figure 1), and the computation consists in numerically integrating the mutual inductance of each pair of elementary wire segments. This computation has to be fast enough to enable trial-and-error design of the set of coils (a recent PC takes several hours even for simple coil configurations). However, the final result has to be accurate to a few digits only.

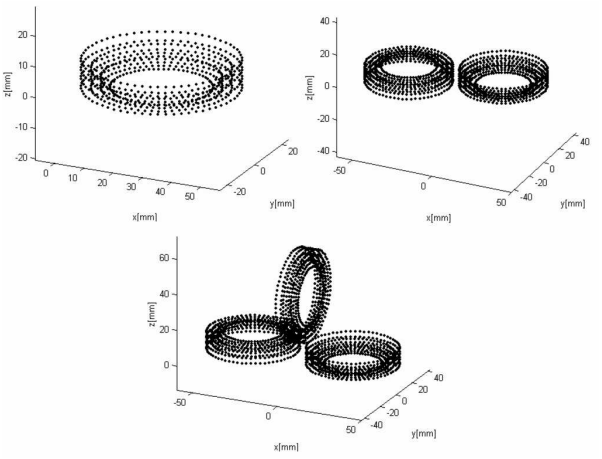


Figure 1: Three small coil configurations

This integration process consists of 8 nested loops, the core of which is an accumulation given below:

$$\text{Acc} = \text{Acc} + \frac{v_1}{v_2} \times \log \frac{v_3 + v_2 - \frac{v_5}{v_2}}{v_4 - \frac{v_5}{v_2}} \quad . \qquad (1)$$

Here the v_i are intermediate variables computed as combinatorial functions of the parameters of the coil segments (mostly 3D cartesian coordinates read from a RAM) and an integration variable t, using basic arithmetic operations:

$$\begin{aligned} v_1 &= (x_1 - x_0)(x_3 - x_2) + (y_1 - y_0)(y_3 - y_2) \\ &\quad + (z_1 - z_0)(z_3 - z_2) \qquad (2) \\ v_2 &= \sqrt{(x_3 - x_2)^2 + (y_3 - y_2)^2 + (z_3 - z_2)^2} \qquad (3) \\ \cdots &\quad \text{(the definitions of the other variables are similar)} \end{aligned}$$

It is important to note that the only loop dependency, in this computation, is in the accumulator: All the values to be summed can be computed in parallel. This makes this application an easy one, from the point of view of parallelism but also of error analysis, as the sequel will show.

3 Using flexible formats

This section does not introduce anything new, however it defines the core of our approach: when using floating-point, one should dedicate some effort to make the best use of the flexibility of available

building blocks. We show that the required analysis is not necessarily difficult, and we demonstrate its benefits.

3.1 Inputs and outputs

Most of the interface of a computing system is intrinsically in fixed-point. To our knowledge, no sensor or analog-to-digital converter provides a floating-point output (some use a logarithmic scale, though), not to mention the 64-bit resolution of a DP number. Similarly, very few engineers are interested in the result's digits after the 5th. However, as soon as one wants to compute hassle-free on such data with a commodity processor, the straightforward approach is to cast the input to DP, do all the computation using the native DP operators of the processor, and then print out only the few digits you're interested in. This is absolutely valid on a processor, where the DP operators are available anyway and highly optimized.

3.2 Computing just right

With the fine-granularity of the FPGA, however, you pay for every bit of precision, so it would make sense to compute with "just the right precision". In some cases, reducing the precision means reducing the hardware consumption, hence having a design that fits in a given device, or removing the need to partition a design. Reducing the precision also means computing faster, all the more as hardware reduction may also allow for more parallelism. Finally, iterative convergence algorithms may expose a trade-off between a larger number of iterations due to reduced accuracy, and a smaller iteration delay [23].

However, it is striking how few [36, 41] of the applications published on FPGAs actually diverge from the processor-based approach, and use a precision different from the standard single (32 bit) and double (64 bit) precisions. One reason, of course, is that a designer doesn't want to add a parameter – the precision – to each operator of a design that is already complex to set up.

3.3 What do you want to compute?

Another, deeper reason of using standard formats is often the concern to ensure a strict compatibility with a trusted reference software, if only for debugging purpose. Athough it is not the main object of this article, we believe that this conservative, structural approach should ultimately be replaced with a more behavioral one: *the behaviour of a floating-point module should be specified as a desired mathematical relationship between its inputs and outputs*.

As an example of the benefits of such approach, consider that it is already used for very small modules with a clean mathematical specification, such as elementary function (exp, log, sine, etc) or the the euclidean norm $\sqrt{x^2+y^2+z^2}$. We will see in section 5 the benefits of such a clean specification: using internal algorithms designed for the FPGA, we are able to desing operators for such modules that are numerically compatible with the software ones [13], or provably always more accurate.

The main problem with such behavioral specification, however, is that it is not supported by current mainstream programming languages, probably because a compiler for such a language is currently out of reach.

However, an ad-hoc study of the precision requirements of an application is often possible, if not for the whole of an application, at least for parts of it. Although this requires an expertise which is rarely associated with circuit design, we now illustrate on our example application that a back-of-the-enveloppe precision analysis may already provide valuable information and lead to improvements in both performance and accuracy. This approach is very general. For instance, the interested reader will find similar analysis for the implementation of elementary functions in [11].

3.4 Mixing and matching fixed- and floating-point in the coils application

We know that the coils will be built with a precision of at best 10^{-3}. Their geometries are expressed in terms of 3D cartesian coordinates, which are intrinsically fixed-point numbers.

This suggests that the coordinate inputs, instead of SP, could be 11-bit fixed-point numbers, whose resolution $(1/2048)$ is enough for the problem at hand. Now consider the implementations of equations (2) and (3): we first substract coordinates, then multiply the results, then sum then up, then possibly take the square root. If the inputs are fixed-point numbers, it becomes natural to do the subtractions in fixed-point (it will be exact, without any rounding error). Similarly, we may perform exact multiplications in fixed point, keeping all the 22 bits of

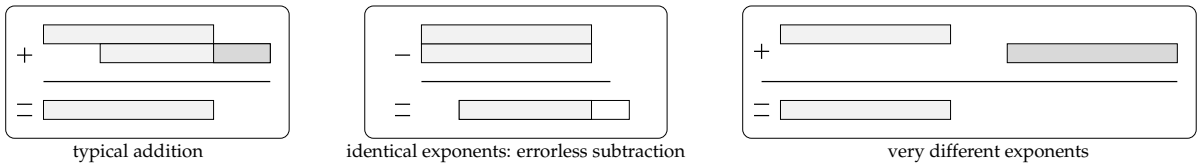


Figure 2: Special cases of floating-point addition

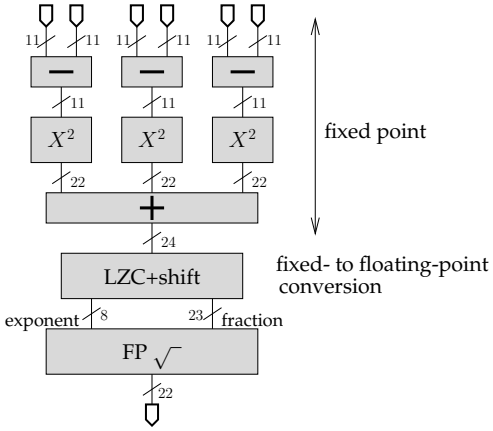


Figure 3: An ad-hoc datapath for Eq. (3)

the results. The sum of three such terms fits in the 24 bits of a SP mantissa.

It is therefore natural to switch to floating-point only at that time, when square root and divisions come into play – the easy back-of-the-enveloppe evaluation of the required precision indeed stops with these operators. The conversion to FP will be errorless, as the numbers fit on 24 bits. However it requires a dedicated operator which will mostly have to count the leading zeroes of the fixed-point numbers to determine their exponent, and shift the mantissa by this amount to bring the leading one in the first position.

To sum up (see Figure 3), we have replaced 3 SP subtractions, 3 SP multiplications and 2 SP additions with as many fixed-point operators, at the cost of one final fixed-to-floating-point conversion (in the initial, all-SP implementation, these conversions were free, since they were done in software when filling the RAMs with the coordinates). We have proven that all the replacement operations were errorless. Indeed, they were already in the full SP version, which also shows that all the rounding hardware present in this version went unused.

Considering the implementation of an FP adder given at Figure 4, the saving of resources is tremendous. Remark that the hardware for the final conversion is actually smaller than the LZC (leading zero counter)+shift part of the close path in this figure. Concerning the multiplications, a $11 \times 11 \rightarrow 22$-bit fixed-point multiplication will consume only one of the DSP blocks of a Virtex device, where a SP multiplication consumes four of them. Also, the cycle count of these fixed-point operations is much lower than that of their floating-point counterpart, which will save a tremendous number of registers: we are able to remove more than 20 32-bit registers from the implementation of (1) depicted in [40].

Note that if, for some reason, we need more than 11 bits of resolution for the coordinates, the flexibility of the FPGA will allow us to design a similar errorless datapath, with a single final rounding when converting to SP, or with the possibility of converting to a format larger than SP.

This analysis could also be used to reduce the size of the exponent of the FP format to $\lceil \log_2(24) \rceil = 5$ bits. However, the cost of an overestimated exponent size is very small, so we may quietly keep the 8 bits of the SP format.

3.5 Fast approximate comparisons

There are many other ways in which fixed- and floating-point can be mixed easily. Here is another example. There are many situations where an approximate floating-point comparison is enough, and this comparison is on the critical path. A typical example is a `while(epsilon>threshold)` condition in a numerical iterative algorithm, where `threshold` is somehow arbitrary. In this case, consider the following: the features of the IEEE-754 format (normalized mantissas, implicit leading 1, and biased exponent), have been designed in such a way that positive floating-point numbers are sorted by lexicographic order of their binary representation [18] – this even extends to subnormal numbers. As

a consequence, comparison of positive numbers can be implemented by an *integer* subtractor or comparator inputting the binary representations considered as integers. Indeed, the latency of FP compare operations is always much smaller than that of FP add/subtract. Furthermore, approximate comparison can be obtained by integer comparison of a chosen number of most significant bits. For example, if `epsilon` is computed as a DP number (64 bits) but the comparison to `threshold` is acceptable with a $2^{-10} \approx 10^{-3}$ relative error, then a valid implementation of this comparison is a 21-bit integer comparator inputting the 21 most significant bits of `epsilon` (its 11 exponent bits, and 10 bits of mantissa).

An extreme instance of the previous is the following: A common recipe to increase the accuracy of an FP accumulation is to sort the input numbers before summing them [19]. In the general case (sum of arbitrary numbers) or in the case when all the summands have the same sign, they should be sorted by increasing order of magnitude. However, if the result is expected to be small, but some summands are large and shall cancel each other in the accumulation, a sum by decreasing orders of magnitude shall be preferred. In both cases, the motivation is to add numbers whose mantissas are aligned (see Figure 2). Otherwise, the lower bits of the smaller number are rounded off, meaning that the information they hold will be discarded. This shows that it will be enough to sort the summands *according to their exponents only*.

Another point of view on the same issue is that the exponents can be used to predict the behaviour of following operations. For instance, if the exponents are equal, a subtraction will be exact (no rounding error). If they are very different, an addition/subtraction will return the larger operand unmodified (see Figure 2). Such information could be used to build efficient speculative algorithms [7].

3.6 ... to be continued

This section did not pretend to be exhaustive, as other applications will lead to other optimization ideas. Next section, for example, will use non-standard FP multipliers whose output precision is higher than input precision.

We hope to have shown that the effort it takes to optimize an application's precision is not necessarily huge, especially considering the current complexity of programming FPGAs. We do not (yet) believe

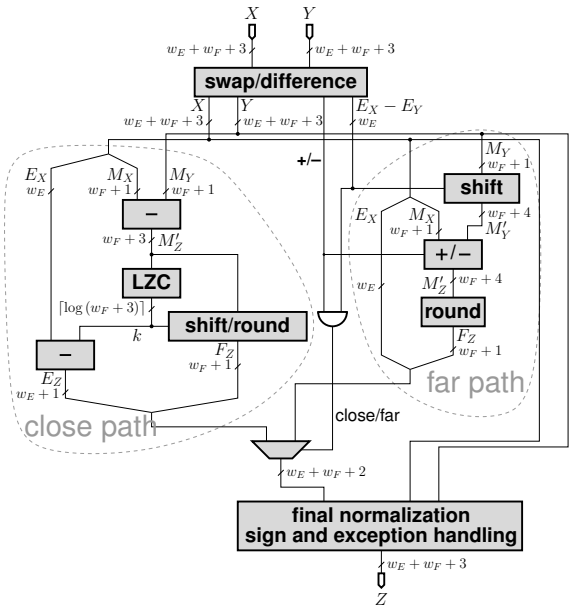


Figure 4: A typical floating-point adder (w_E and w_F are respectively the exponent and mantissa sizes)

in fully automatic approaches, however such work does not require too specialized an expertise. We will survey in conclusion the tools available to assist a designer in this task.

The remainder of this paper now uses the same philosophy of "computing just right" to design completely new operators which are not available in a processor.

4 Accumulation

Summing many independent terms is a very common operation. Scalar product, matrix-vector and matrix-matrix products are defined as sums of products. Another common pattern is integration, as in our example application: when a value is defined by some integral, the computation of this value will consist in adding many elementary contributions. Many Monte-Carlo simulations also involve sums of many independent terms.

For few simple terms, one may build trees of adders, but when one has to add many terms, iterative accumulation is necessary. In this case, due to the long latency of FP addition (6 cycles in [40] for SP, up to 12 cycles for Xilinx cores), one needs to design specific architectures involving multiple buffers

to accumulate many numbers without stalling the pipeline. This long latency is explained by the architecture of a floating-point adder given on Figure 4.

In the previous examples, it is a common situation that the error due to the computation of one summand is more or less constant and independent of the other summands, while the error due to the summation grows with the number of terms to sum. This happens in our example, and also in most matrix operations. In this case, it makes sense to have more accuracy in the accumulation than in the summands.

A first idea, to accumulate more accurately, is to use a standard floating-point adder with a larger mantissa. However, this leads to several inefficiencies. In particular, this large mantissa will have to be shifted, sometimes twice (first to align both operands and then to normalize the result). These shifts are in the critical path loop of the sum (see Figure 4).

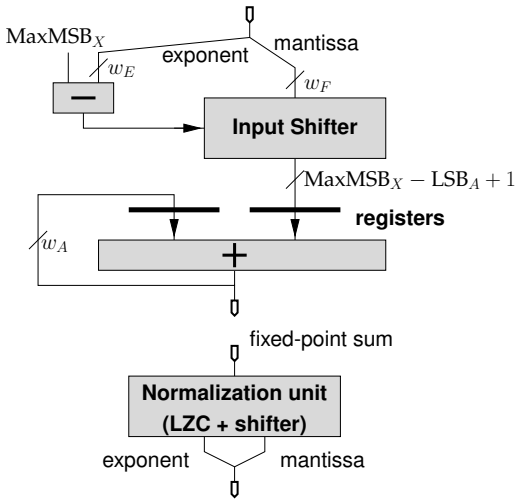


Figure 5: The proposed accumulator. Only the registers on the accumulator itself are shown, the rest of the design is combinatorial and can be pipelined arbitrarily.

4.1 The large accumulator concept

The accumulator architecture we propose here (see Figure 5) removes all the shifts on the critical path of the loop by keeping the current sum as a large fixed-point number (typically much larger than a mantissa, see Figure 6). There is still a loop, but it is now a fixed-point accumulation for which current FPGAs are highly efficient. Specifically, the loops uses fast-carry logic and involves only the most local routing. For illustration, for 64-bits (11 bits more than the DP mantissa), a Virtex4 with the slowest speed grade (-10) runs such an accumulator at more than 200MHz, while consuming only 64 CLBs. Section 4.4 will show how to reach even larger frequencies or sizes.

The shifters now only concern the summand (see Figure 5), and can be pipelined as deep as required by the target frequency.

The normalization of the result may be performed at each cycle, also in a pipelined manner. However, most applications won't need all the intermediate sums: they will output the fixed-point accumulator (or only some of its most significant bits), and the final normalization may be performed offline in software, once the sum is complete, or in a single normalizer shared by several accumulators (case of matrix operations). Therefore, it makes sense to provide this final normalizer as a separate component, as shown by Figure 5.

For clarity, implementation details are missing from these figures. For example, the accumulator stores a two's complement number, so the shifted summand has to be sign-extended. The normalization unit also has to convert back from two's complement to sign/magnitude. All this isn't on the critical path of the loop either.

In addition to being simpler, the proposed accumulator has another decisive advantage over the one using the standard FP adder: it may also be designed to be much more accurate. Indeed, it will even be exact (entailing no roundoff error whatsoever) if the accumulator size is large enough so that its LSB is smaller than that of all the inputs, and its MSB is large enough to ensure that no overflow may occur. Figure 6 illustrates this idea, showing the mantissas of the summands, and the accumulator itself.

This idea was advocated by Kulisch [21, 22] for implementation in microprocessors. He made the point that an accumulator of 640 bits for SP (and 4288 bits for DP) would allow for arbitrary dot-products to be computed exactly, except when the final result is out of the FP range. Processor manufacturers always considered this idea too costly to implement. When targetting a specific application to an FPGA, things are different: instead of a huge generic accumulator, one may chose the size that matches the requirements of the application. There are 5 parameters on Figure 5: w_E and w_F are the ex-

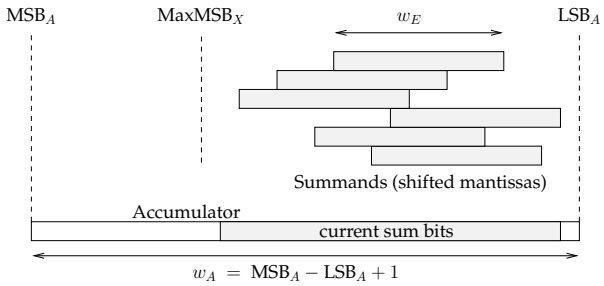


Figure 6: Accumulation of floating-point numbers into a large fixed-point accumulator

ponent and mantissa size of the summands; MSB$_A$ and LSB$_A$ are the respective weights of the most and less significant bit of the accumulator (the size in bits of the accumulator is $w_A = \text{MSB}_A - \text{LSB}_A + 1$), and MaxMSB$_X$ is the maximum expected weight of the MSB of a summand. By default MaxMSB$_X$ will be equal to MSB$_A$, but sometimes the designer is able to tell that each summand is much smaller in magnitude than the final sum. For example, when integrating a function that is known positive, the size of a summand could be bounded by the product of the integration step and the max of the function. In this case, providing MaxMSB$_X$ $<$ MSB$_A$ will save hardware in the input shifter. Defining these parameters requires some trial-and-error, or (better) again some back-of-the-enveloppe error analysis. We now demonstrate that on our example application.

4.2 Our application

In our inductance computation, physical expertise backed by preliminary software simulations gave us the order of magnitude of the expected result: the sum will be less than 10^5 (using arbitrary units due to factoring out some physical constants). Adding two orders of magnitude for safety, and converting to bit weight, this defines $\text{MSB}_A = \lceil \log_2(10^2 \times 10^5) \rceil = 24$.

Besides, in software simulation, the absolute value of a summand never went over 2 and below 10^{-2}. Again adding two orders of magnitude (or 7 bits) for safety in all directions, this provides $\text{MaxMSB}_X = 2^8$ and $\text{LSB}_A = 2^{-w_F-15}$ where w_F is the mantissa width of the summands. For $w_F = 23$ (SP), we conclude that an accumulator stretching from $\text{LSB}_A = 2^{-23-15} = 2^{-38}$ (least significant bit) to $\text{MSB}_A = 2^{24}$ (most significant bit) will be able to absorb all the additions without any rounding error: no summand will add bits lower than 2^{-38}, carries propagate from right to left, and the accumulator is large enough to ensure it never overflows. The accumulator size is therefore $w_A = 24 + 38 + 1 = 63$ bits. Again, this will not only be more accurate than using an accumulator based on a DP FP adder, it will also be smaller and faster.

Note that the value of LSB$_A$ should be considered as a tradeoff between precision and performance: we have discussed above a perfect, errorless accumulator, but one may also be contented with a smaller, still more-accurate-than-FP accumulator.

Remark that defining specifications this way is both safe and easy, thanks to the freedom to add several orders of magnitude of margin in all directions. This, of course, has a hardware cost: the accumulator has 14 bits out of 63 that are probably never used, and the same holds for the input shifter. In our application, this adds roughly 20% to the cost of the accumulator, but this overhead is negligible in the total cost of the application.

Also 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 make it easy to experiment with various w_F. Besides, it shows that the dependency of the cost of the long accumulator to w_F will be almost linear (actually it is in $w_F \log w_F$ due to the input shifter). This is confirmed by synthesis: for SP, a 63-bit accumulator occupies 125 Virtex-2 slices. Should we upgrade the summand pipeline to DP, we would need a 92-bit accumulator requiring 219 slices.

Finally, note that this accumulator design has only removed the rounding error due to accumulation per se. All the summands are still computed using w_F bits of accuracy only, and therefore potentially hold an error equivalent to their least-significant bit. In a hypothetical worst-case situation, when adding 2^N numbers, these errors will accumulate and destroy the signification of all the bits lower than 2^{8-w_F+N} in our example. To be totally safe, w_F could be chosen accordingly as a function of N and of the least significant digit the end user is interested in. In our application we have less than 2^{40} numbers to accumulate, we have taken some margin, errors will compensate each other, so computing the summands in SP is largely enough.

4.3 Exact dot-products and matrix operations

Kulisch's designs address the previous problem in the simpler case of dot product [22]. The idea is simply to accumulate the exact results of all the multiplications. To this purpose, instead of standard multipliers, we use *exact* multipliers that return all the bits of the exact product: for $1 + w_F$-bit input mantissa, they return an FP number with a $2 + 2w_F$-bit mantissa. The exponent range is also doubled, which means adding one bit to w_E. 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, we save all the rounding logic altogether: Results do not even need to be normalized, as they will be immediately sent to the fixed-point accumulator. The only additional cost is in the accumulator, which requires a larger input shifter (see Figure 5).

With these exact multipliers, if we are able to bound the exponents of the inputs so as to obtain a reasonably small accumulator, it becomes easy to prove that the whole dot-product process is exact, independently of its size. Otherwise it is just as easy to provide accuracy bounds which are better than standard, and arbitrarily small. As matrix-vector and matrix-matrix products are parallel instances of dot-products, these advantages extend to such operations. We welcome suggestions of matrix-based actual applications that could benefit from these ideas.

4.4 Very large accumulator design

We have mentionned that 64-bit accumulation was performed at the typical frequency of current FPGAs. This should be enough for SP computations, but it is only 11 bits more than DP. Up to 120 bits, we have obtained the same frequency at twice the hardware cost using a classical one-level carry-select adder (see Figure 7).

Other designs will be considered for even larger precision if needed [22], but we believe that 120 bits should be enough for most DP computations. Note that this carry-select design can also be used to implement a smaller accumulator with even larger frequency. For example, it improves the frequency of a 64-bit accumulator from 202 to 295 MHz on a Virtex4, speed grade -10.

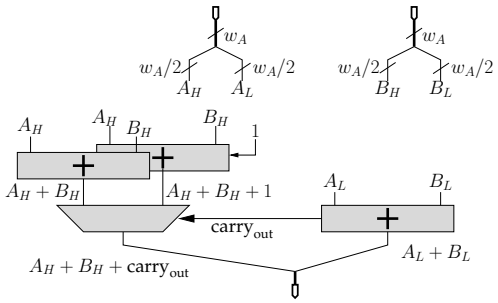


Figure 7: Carry-select adder

5 Coarser-grain operators

If a sequence of floating-point operations is central to a given computation, it is always possible to design a specific operator for this sequence. Some examples that have been studied (to various extents) are multiplication by a constant, combinations of multiply and add such as complex number multiplication, variations around the euclidean norm $\sqrt{x^2 + y^2 + z^2}$, elementary functions, and other compound functions.

5.1 Smaller, and more accurate

The successful recipe for such designs will typically be to keep the interface to the operator (again, these blocks are small enough to have a clean mathematical specification) but perform as much as possible of the computation in fixed point. If the compound operator is proven always to compute more accurately than the succession of elementary operators, it is likely to be accepted. Another requirement may be to provide results guaranteed to be faithful (last-bit accurate) with respect to the exact result.

The real question is, when do we stop? Which of these optimized operators are sufficiently general and offer sufficient optimization potential to justify that they are included in a library? There is a very pragmatical answer to this question: As soon as an operator is designed for a given application, it may be placed in a library. From there on, other applications will be able to use it. Again, this approach is very specific to the FPGA paradigm. In the CPU world, adding a hardware operator to an existing processor line must be backed by a lot of benchmarking showing that the cost does not outweight the benefit. Simply take the example of division: Is a hardware divider wasted silicon in a CPU? Roughly

at the same time, Flynn *et al.* did such benchmarking to advocate hardware dividers [33], while the Itanium processor family was designed without a hardware divider, but with two fused multiply-and-add units designed to accelerate software division algorithms [31].

We now present some of these compound operators in more detail, without pretending to exhaustiveness: It is our belief that each new application will bring in some compound operation worth investigating.

5.2 Multiplication by a constant

By definition, a constant has a fixed exponent, therefore the floating point is of little significance here: all the research that has been done on integer constant multiplication [4, 26, 6, 14] can be used straightforwardly. To sum up this research, a constant multiplier will always be at least twice as small (and usually much smaller) than using a standard multiplier implemented using only CLBs. A full performance comparison with operators using DSP blocks remains to be done, but when DSP blocks are a scarce resource, it is good to know that the multiplications to be implemented in logic should be the constant ones.

We may also define constant multipliers that are much more accurate than what can be obtained with a standard FP multiplier. For instance, consider the irrational constant π. It can not be stored on a finite number of bits, but it is nevertheless possible to design an operator that provably always returns the correctly rounded result of the (infinitely accurate) product πx [3]. This is useful for trigonometric argument reduction, for example [10]. The number of bits of the constant that is needed depends on the constant, and may be computed using continued fraction arguments [3]. Although one usually needs to use more than w_F bits of the constant to ensure correct rounding, the resulting constant multiplier may still be smaller than a generic one. Optimizing such correctly-rounded constant multipliers is the subject of current investigation.

5.3 Exact sum and product

Much recent work has been dedicated to improving the accuracy of floating-point software using *compensated summation*, see [34] for a review. The approach presented Section 4 will often make more sense in an FPGA, however compensated summation techniques have the advantage of requiring less error analysis and (equivalently) of scaling better to problem sizes unknown a-priori, and also to be software-compatible.

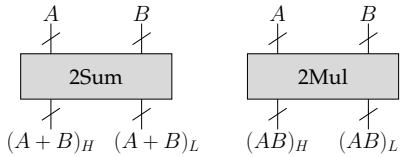


Figure 8: The 2Sum and 2Mul operators

These approaches rely on two basic blocks called 2Sum and 2Mul that respectively compute the exact sum and the exact product of two FP numbers. In both cases, the result fits in the unevaluated sum of two FP numbers (see Figure 8). For example (in decimal for clarity), the product of the 4-digit numbers 6.789 and 5.678 is $3.8547942 \cdot 10^1$ which may be represented by the unevaluated sum of two 4-digit numbers $3.854 \cdot 10^1 + 7.942 \cdot 10^{-3}$. Actually, to ensure unicity of the representation, compensated algorithms require that the most significant FP number is the correct rounding of the exact result, so in our example the result will be written as $3.855 \cdot 10^1 - 2.058 \cdot 10^{-3}$ (the reader may check that the sum is the same).

In a processor, these 2Sum and 2Mul operators require long sequences [20] of standard FP additions and multiplications: 3 to 8 FP additions for an exact sum, depending on the context, and between 7 and 17 operations for the exact multiplication, depending on the availability of a fused multiply-and-add [31]. Besides, most of these operations are data-dependent, preventing an efficient use of the pipeline.

The overhead of 2Sum and 2Mul over the standard FP adder and multiplier will be much smaller in an FPGA, as all the additional information to output (the least significant bits of the sum or product) is already available inside the FP operator, and usually just discarded by the rounding. Specifically, an FP multiplier always has to compute the full mantissa product to determine rounding. An FP adder does not compute the full addition, but the lower bits are untouched by the addition, so no additional computation is needed to recover them. The only additional required work consists in 1/ propagating the rounding information of the most significant part down to the least significant part, sometimes

changing its sign as in our example, and 2/ normalizing the lower part using a LZC+shifter.

Summing it up, the area and delay overhead of 2Mul over an FP multiplier is a few percents, while the area and delay overhead of 2Sum over the FP adder is less than a factor 2 (for delay also, as the LZC/shift of the lower part has to wait for the end of the LZC/Shift of the higher part). In terms of raw performance, comparing with the cost of software implementations of 2Mul and 2Sum on a processor, we conclude that the FPGA will recover its intrinsic performance gap on algorithms based on these operators [34].

Similar improvements could probably be brought to the basic blocks used in the architecture of DeHon and Kapre [7] that computes a parallel sum of FP numbers which is bit-compatible with the sum obtained in a sequential order.

5.4 Variations around the euclidean norm

Let us now consider the case of the 3D norm $\sqrt{x^2+y^2+z^2}$. In our application, this operator is no longer needed after Section 3.4, however it is felt to be ubiquitous enough to justify some optimization effort.

The first option is to combine library $+$, $\times$ and $\sqrt{}$ operators. When we did so, using FPLibrary (`www.ens-lyon.fr/LIP/Arenaire/`), we had the surprise to observe that the area was much smaller than expected [12]. It turned out that large useless portions of the floating-point adders were discarded by the synthesis tools. More specifically, these adders are dual-path designs [17] (see Figure 4), where one of the path is only useful in case of subtraction of the mantissas. In $\sqrt{x^2+y^2+z^2}$, of course, there is no subtraction, and the synthesis tool (Xilinx ISE) was able to detect that the sign of the addends was always positive and that one of the path was therefore never needed. Interestingly, using pre-compiled, pre-placed cores would not offer the same benefit. Note that in principle, a squarer implemented in logic is also in theory almost twice smaller than a full multiplier, but this doesn't seem to be optimized by current tools.

A second option is to compute the squares in fixed-point, align them then add them, still in fixed point, then use a fixed-point square root (the one used within the floating-point operator) along with ad-hoc renormalization step. As the architecture is only slightly more complex than the norm part of Figure 3, the design effort is not very high, but so is the benefit (mostly the saving of a few intermediate normalizations). As an illustration, for SP on a VirtexII device, such an optimized operator requires 72% of the area of the unoptimized one, and 78% of the delay.

A third option is to explore specific algorithms such as those published by Takagi [39, 38]. Compared to the previous simpler approach, Takagi's architecture seems to reduce the latency but not the hardware cost, however this is a promising new direction. Besides, it may be used to compute other useful compound operators such as $\frac{1}{\sqrt{x^2+y^2+z^2}}$ or $\frac{x}{\sqrt{x^2+y^2}}$. In such cases, the hope is to combine the digit recurrence of the square root with that of the division [38]. This is currently future work.

5.5 Elementary and compound functions

Much recent work has been dedicated to FP elementary function computation for FPGAs (exp and log in SP [15, 11] and DP [13], trigonometric functions in SP [35, 10]. The goal of such works is to design specific, combinatorial architectures based on fixed-point computations for such functions. Such architectures can then be pipelined arbitrarily to run at the typical frequency of the target FPGA, with latencies that can be quite long (up to several tens of cycles for DP), but with a throughput of one elementary function per cycle. The area of the state of the art is comparable to that of a few multipliers. As these pipelined architecture compute one result per cycle, the performance is is one order of magnitude better than a processor. Our application uses such a logarithm function for Eq. (1).

The holy grail of this kind of research would be to be able to automatically generate architectures for arbitrary functions. Several groups have demonstrated generic tools for this purpose in fixed-point [9, 25], but these do not yet seem to be ready for mass-consumption. Besides, these approaches are not directly transposable to floating-point, due to the large range of floating-point numbers.

6 Conclusion and future work

There are two aspects in the present paper. One is to show that original floating-point operators can be designed on FPGAs. All these operators will at

some point be available in a tool called $(FPG)^2A$ – a Floating-Point Generator for FPGAs – currently under development as the successor to FPLibrary [12]. It will include basic operators, whose only originality will be to allow for different input and output precisions, and also more exotic ones such as elementary functions and large accumulators.

6.1 The missing tools

The second aspect of this paper is to advocate using custom precisions and exotic operators. We acknowledge that the expertise required is not widespread among hardware engineers, and tools will have to be built to assist them. Still, some of our solutions are easy to use: elementary function or euclidean norm operators may be used transparently, just like any other library component. Some other techniques are not too difficult to use as soon as one is aware of the possibility: We hope to have shown that the large accumulator concept is simple, and that its benefits will often justify the additional design effort. Now, to resize a full datapath so that it computes just right with a proven bound on the final accuracy, this is something that requires skills in hardware engineering as well as in error analysis, and even then, the difficulty of the task may be arbitrarily large. Again, we acknowledge that our example application, although not a toy one, is relatively easy to study.

We have already mentionned that the main problem is one of programming languages and compilers: as compiling (subsets of) C to FPGA is still the subject of active research, automating our back-of-the-enveloppe calculations is far beyond the horizon. Current research focusses on isolating use cases on which existing tools can be used: interval arithmetic for error analysis, polynomial approximation for univariate functions, etc. For instance, there is some hope to be able to infer the required size of a large accumulator automatically, or at least with machine assistance. The Gappa proof assistant has already been used to help validate complex datapaths for elementary function implementations [5]. The goal of $(FPG)^2A$ is to build upon such tools. We will also have to collaborate with the C-to-FPGA compilation community.

6.2 Area, delay, and also accuracy

A transversal conclusion of this survey is that the quality of some floating-point computation has to be evaluated on a three-dimensional scale: *area* and *performance* are important, but so is *accuracy*. It was already known that FPGAs were able to outperform processor in the area/performance domain thanks to massive parallelism. Our thesis is that their flexibility in the *accuracy* domain may bring even higher gain: many computations are just too accurate on a processor, and an FPGA implementation will recover its intrinsic performance gap by computing "just right". Some applications are not accurate enough on a processor, and an FPGA implementation may be designed to be more accurate for a marginal overhead.

6.3 The missing bits

As a conclusion, should floating-point units be embedded [2] in FPGAs? We believe that this would be a mistake, as it would sacrifice the flexibility that currently gives the advantage to the FPGA. On the other hand, the FPGA fabric could be improved for better FP support. Looking back at several years of FP operator design, it is obvious that they all, at some point, use LZC/shifters to renormalize floating-point results or to align operands before addition. The introduction of embedded shifters has already been suggested [1]. We suggest that such shifters should be easily cascadable for arbitrary precision floating-point, and that they should be able to combine leading-zero/one counting and shifting.

Another issue is that of the granularity of the embedded multipliers: One common feature of many of FPGA-specific FP architectures is rectangular multiplication [9]. The DP logarithm of [13], for instance, involves several such multiplications, from 53×5 to 80×5 bits. These are currently implemented in CLBs. Small multipliers are easily combined to form larger ones, so replacing current DSP blocks with many more, but smaller (say, 8-bit) ones would be welcome. There is probably an issue of additional routing cost, so it would be interesting to investigate this issue on large floating-point benchmarks involving elementary functions. Even a Pentium, with MMX, has finer multiplication granularity than an FPGA: isn't something wrong there?

Acknowledgement

Support of the Egide Brancusi programme of the French government is gratefully acknowledged.

References

[1] M. Beauchamp, S. Hauck, K. Underwood, and K. Hemmert. Architectural modifications to improve floating-point unit efficiency in FPGAs. In *Field Programmable Logic and Applications*, pages 1–6, Aug. 2005.

[2] M. J. Beauchamp, S. Hauck, K. D. Underwood, and K. S. Hemmert. Embedded floating-point units in FPGAs. In *ACM/SIGDA international symposium on Field programmable gate arrays*, pages 12–20. ACM, 2006.

[3] N. Brisebarre and J.-M. Muller. Correctly rounded multiplication by arbitrary precision constants. In *Proc. 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*. IEEE Computer Society Press, June 2005.

[4] K. Chapman. Fast integer multipliers fit in FPGAs (EDN 1993 design idea winner). *EDN magazine*, May 1994.

[5] F. de Dinechin, C. Lauter, and G. Melquiond. Assisted verification of elementary functions using Gappa. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1318–1322, 2006.

[6] F. de Dinechin and V. Lefèvre. Constant multipliers for FPGAs. In *Parallel and Distributed Processing Techniques and Applications*, pages 167–173, 2000.

[7] A. DeHon and N. Kapre. Optimistic parallelization of floating-point accumulation. In *18th Symposium on Computer Arithmetic*, pages 205–213. IEEE, June 2007.

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

[9] J. Detrey and F. de Dinechin. Table-based polynomials for fast hardware function evaluation. In *Application-specific Systems, Architectures and Processors*, pages 328–333. IEEE, 2005.

[10] J. Detrey and F. de Dinechin. Floating-point trigonometric functions for FPGAs. In *Intl Conference on Field-Programmable Logic and Applications*, pages 29–34. IEEE, Aug. 2007.

[11] J. Detrey and F. de Dinechin. Parameterized floating-point logarithm and exponential functions for FPGAs. In *Microprocessors and Microsystems*. Elsevier, 2007. To appear.

[12] J. Detrey and F. de Dinechin. A tool for unbiased comparison between logarithmic and floating-point arithmetic. *Journal of VLSI Signal Processing*, 2007. To appear.

[13] J. Detrey, F. de Dinechin, and X. Pujol. Return of the hardware floating-point elementary function. In *18th Symposium on Computer Arithmetic*, pages 161–168. IEEE, June 2007.

[14] V. Dimitrov, L. Imbert, and A. Zakaluzny. Multiplication by a constant is sublinear. In *18th Symposium on Computer Arithmetic*. IEEE, June 2007.

[15] C. Doss and R. L. Riley, Jr. FPGA-based implementation of a robust IEEE-754 exponential unit. In *Field-Programmable Custom Computing Machines*, pages 229–238. IEEE, 2004.

[16] 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.

[17] M. J. Flynn and S. F. Oberman. *Advanced Computer Arithmetic Design*. Wiley-Interscience, 2001.

[18] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, Mar. 1991.

[19] N. J. Higham. *Accuracy and stability of numerical algorithms*. SIAM, 1996.

[20] D. Knuth. *The Art of Computer Programming, vol.2: Seminumerical Algorithms*. Addison Wesley, 3rd edition, 1997.

[21] U. Kulisch. Circuitry for generating scalar products and sums of floating point numbers with maximum accuracy. United States Patent 4622650, 1986.

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

[23] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra. Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). In *ACM/IEEE conference on Supercomputing*. ACM Press, 2006.

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

[25] D. Lee, A. Gaffar, O. Mencer, and W. Luk. Optimizing hardware function evaluation. *IEEE Transactions on Computers*, 54(12):1520–1531, Dec. 2005.

[26] V. Lefèvre. Multiplication by an integer constant. Technical Report RR1999-06, Laboratoire de l'Informatique du Parallélisme, Lyon, France, 1999.

[27] 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, Apr. 1997.

[28] 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.

[29] 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.

[30] 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.

[31] P. Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000.

[32] K. R. Nichols, M. A. Moussa, and S. M. Areibi. Feasibility of floating-point arithmetic in FPGA based artificial neural networks. In *CAINE*, pages 8–13, 2002.

[33] S. F. Oberman and M. J. Flynn. Design issues in division and other floating-point operations. *IEEE Transactions on Computers*, 46(2):154–161, 1997.

[34] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

[35] F. Ortiz, J. Humphrey, J. Durbano, and D. Prather. A study on the design of floating-point functions in FPGAs. In *Field Programmable Logic and Applications*, volume 2778 of *LNCS*, pages 1131–1135. Springer, Sept. 2003.

[36] 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, Sept. 2002.

[37] 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.

[38] N. Takagi. A hardware algorithm for computing the reciprocal square root. In *15th Symposium on Computer Arithmetic*, pages 94–100, Vail, Colorado, June 2001. IEEE.

[39] N. Takagi and S. Kuwahara. A VLSI algorithm for computing the euclidean norm of a 3D vector. *IEEE Transactions on Computers*, 49(10):1074–1082, 2000.

[40] I. Trestian, O. Creţ, L. Creţ, L. Văcariu, R. Tudoran, and F. de Dinechin. FPGA-based computation of the inductance of coils used for the magnetic stimulation of the nervous system. Technical Report ensl-00169909, École Normale Supérieure de Lyon, 2007. http://prunel.ccsd.cnrs.fr/ensl-00169909/fr.

[41] 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.