Mixed-precision Fused Multiply and Add

Nicolas Brunie, Florent de Dinechin, Benoît de Dinechin

Mixed-precision Fused Multiply and Add

Nicolas Brunie, Florent de Dinechin, and Benoit de Dinechin
florent.de.dinechin@ens-lyon.fr, {nicolas.brunie,benoit.dinechin}@kalray.eu

***Abstract*—The standard floating-point fused multiply and add (FMA) computes R=AB+C with a single rounding. This article investigates a variant of this operator where the addend C and the result R are of a larger format, for instance binary64 (double precision), while the multiplier inputs A and B are of a smaller format, for instance binary32 (single precision). With minor modifications, this operator is also able to perform the standard FMA in the smaller format, and the standard addition in the larger format.**

For sum-of-product applications, the proposed mixed-precision FMA provides the accumulation accuracy of the larger format, at a cost that is close to that of a classical FMA in the smaller format. Besides, it is fully compatible with existing arithmetic and language standards.

The architectural cost of this operator is analysed in detail. An implementation of a mixed binary32/binary64 operator fully supporting subnormal numbers, binary64 addition and binary32 FMA is demonstrated and evaluated: its area overhead is one third over the classical binary32 FMA. Similarly, in high-end processors, a mixed binary64/binary128 FMA could provide an adequate solution to the binary128 requirements of very large scale computing applications.

I. Introduction

The fused multiply-add operator (FMA) is now an IEEE-754-2008 standard operator. It combines improvements in performance (two operations in a single instruction) and improvements in accuracy (one single rounding). The latter allows for many algorithmic improvements [16], for instance efficient implementations of division and square root. As the FMA can also be used as an adder or as a multiplier, most recent instruction sets (including IBM Power/PowerPC and Intel/HP IA64, but also recent graphical processing units [15]) build their floating-point unit (FPU) around the FMA. This operator will come to the legacy IA32 instruction set with the SSE5 and AVX extensions from AMD and Intel respectively.

Moore's law, bringing more and more transistors per chip, reduces the relative area of a floating-point unit. For instance, one 64-bit FPU of the 8-core POWER7 processor consumes only a quarter of a mm^2 [2], a tiny fraction of the area of a core. The mainstream way to exploit these transistors for floating-point is to provide more parallel FPUs per core, exploited through new vector (or SIMD) instructions. For instance the POWER7 has 4 FPUs per cores. Recent graphical processing units (GPUs) also include large numbers of FMAs [15].

In this article, we consider a complimentary approach, which is to design more functionality in a coarser FPUs. We introduce a floating-point mixed-precision FMA, or MPFMA. For $k \in \{16, 32, 64\}$, the MPFMAk computes $R = \circ(A \times B + C)$ where A and B are binary$k$ numbers, C and R are binary$2k$ numbers, and $\circ$ is one of the rounding modes to the binary$2k$ format as defined by the IEEE-754-2008 standard (and summarized in Table I).

Name	binary16	binary32	binary64	binary128
p	11	24	53	113
$e_{\max}$	$+15$	$+127$	$+1023$	$+16383$
$e_{\min}$	-14	-126	-1022	-16382

Table I
MAIN PARAMETERS OF THE BINARY INTERCHANGE FORMATS SPECIFIED BY THE 754-2008 STANDARD [10].

We show that an FPU based on such an operator may also take care of two operations that are somehow simpler: the standard binary$k$ FMA, and the standard binary$2k$ addition.

The MPFMA32 operator constitutes the basis of the Floating-Point Unit (FPU) of the Kalray K1 processor, a many-core architecture designed for the embedded, low-consumption and DSP market. There, the primary focus is on binary32 support, with binary64 support being a secondary concern, and the MPFMA32 is a versatile and cost-effective answer to these needs. In a different context, we also see the MPFMA64 as a cost-effective way to extend existing binary64 FPUs to provide binary128 precision where large-scale applications need it.

This article is organized as follows. Section II motivates this operator from an applicative point of view, considering the pervasive sum-of-product kernels. This section also compares this solution to previous hardware or software approaches.

The following sections study the construction of an MPFMAk, complete with support of subnormals, of binary$k$ FMA, and of binary$2k$ addition. First, Section III explicits the data alignment requirements for this operator, pointing out where the datapath has to be extended with respect to a standard FMA. This analysis intends to be fairly independent of the constraints of an actual implementation.

Then, Section IV presents an actual implementation in the context of the Kalray K1 processor. The MPFMA32 is compared against classical FMA32, FMA64, and binary64 addition, all designed with comparable optimization effort. It is found that the MPFMA32 area is only 1.3 the area than the classical FMA32, whereas an FMA64 would consume 2.5 times this area and require more bandwidth from the register file.

II. Context and motivations

The MPFMAk is relevant for computing kernels based on sums of products of binary$k$ numbers. Such kernels are pervasive, from linear algebra to signal processing transforms. They may return inaccurate results for at least two reasons. The first is the accumulation of the rounding errors, whose

impact is proportional to the number of products to add. The second, more dependent on the input data, is the occurence of cancellations along the sum. The motivation of the MPFMAk is to provide extra accuracy (and pay its price) for the accumulation, and only for the accumulation.

A. Related work

This accuracy issue in sums and sums of products is pervasive enough to have motivated a lot of techniques that provide extended precision for the accumulation process.

On the hardware side, Digital Signal Processors (DSP) have long offered *mixed-precision* operators for fixed-point: A typical DSP operator is a multiply-accumulate that adds the product of two 16-bit number to a 40-bit accumulator. One initial motivation of the MPFMA operator, in the Kalray processor context, was that is fitted neatly in a DSP-oriented datapath already offering a fixed-point multiply-accumulate with 32-bit multiplier operands and 64-bit accumulators.

For floating-point, Kulisch advocated augmenting the processors with a long accumulator that would enable *exact* accumulation and dot product [11]. So far, processor vendors have not considered the benefits of this extension to be worth its cost. The MPFMA approach is an intermediate trade-off between accumulation using standard operators, and Kulisch's exact accumulation.

On the software side, many techniques have been suggested to double (or more) the precision of accumulation and sums of products, notably by Babuška [1], Pichat [18], Neumaier [17], Priest [19], and Rump, Ogita, and Oishi [23]. They are reviewed in [16, ch. 6]. These techniques cost at least 5 binary$k$ additions per accumulated term.

It has been suggested that these techniques should be assisted by hardware [5], [7] for better performance. This is what our MPFMAk does. Compared to these propositions, it has the additional advantage of an extended exponent range, not only extended precision. This reduces the risk of returning ∞ due to an intermediate overflow when the result should be representable.

Operators managing several precisions have been proposed, for instance an operator able to compute either one binary64, or two binary32 FMA operations in parallel [9]. However, in both cases, the operation uses the same format for all inputs and output. The accumulation of binary32 product in a binary64 register is possible by casting all the operands to binary64 registers, but this solution uses 53-bit significand multiplications to multiply 24-bit data. This is all the more wasteful as the hardware is there to compute two 24-bit multiplications in parallel instead.

Closer to our design is a recent FMA design based on a floating-point multiplier with an extended output and a floating-point adder with an extended operand [13]. This choice of a split implementation is motivated only by performance (we will review it in V-D) and, contrary to our proposal, does not offer access to the extended precision in the instruction set. Therefore, it cannot be used for accurate accumulation or sum-of-product. The extended operand would not be in a standard floating-point format anyway.

Indeed, a strong point of the MPFMA is that it is fully standard-compliant, as we now detail.

B. Standard compliance

Consider the following C code, archetypal of many computing kernels, including matrix operations, finite impulse response (FIR) filters, fast Fourier transforms (FFT), etc.

```
float A[], B[];    /* binary32 */
double C;          /* binary64 */

C=0;
for(i=0; i< N; i++)
   C = C + A[i]*B[i];
```

We observe the following:

Using the MPFMA32 for computing the line `C = C + A[i]*B[i]` *is both C99-compliant and IEEE-754 compliant.*

Proof: Assume we only have the standard addition and multiplication operators. As we have a mix of precisions in this code, there are two ways of implementing it in practice. Either cast `A[i]` and `B[i]` to double, then perform a double-precision operation, or perform a single-precision multiplication, then cast the product to a double. The C99 standard encourages implementation to use wider precisions for intermediate computations if it is not slower. On a processor only offering double-precision hardware, the first approach, which is more accurate and no slower, would therefore be preferred.

Now let us detail what happens in this first option. The cast of a float/binary32 to a double/binary64 is errorless. The product is also errorless, since its significant size is at most 48 bits, which fits in the 53 bits of a binary64 number. In addition, no overflow nor underflow are possible: For underflow, the smallest binary32 subnormal (of value 2^{-149}) is converted to a binary64 normal number, the square of which (2^{-298}) is well within the normal range of binary64. Similarly, the square of the largest, non infinite binary32 values $\left(2-2^{-23}\right)\cdot 2^{127}$ is well within the normal range of binary64. To sum up, `A[i]×B[i]` is computed exactly and without over- or underflow before being added to the binary64 number `C`. This floating point binary64 addition performs one rounding, so there is a single final rounding in the computation of `A[i]*B[i]+C`. This is exactly the behaviour of the proposed MPFMA.

In other words, in a processor offering an MPFMA, we obtain a result that is bit-identical to a result compliant with C99/IEEE-754.

This property holds for MPFMA16 and MPFMA64 as well, as one can check from Table I. For each column from binary32 to binary128, the precision p in this column is larger than twice the precision in the column to the left, which guarantees errorless multiplication, and the same holds for $e_{\min}$ and $e_{\max}$ values, which guarantees absence of underflow and overflow.

C. Hardware support of additions in the wider format

As we will see, an MPFMA also naturally supports addition in the wider format, which requires the same amount of inputs and outputs, and only marginal modifications to the datapath.

This operation is useful in its own right, but it is also useful in the context of the extra accurate accumulation: to fill the operator pipeline of depth l, it is desirable to split a large accumulation into l smaller ones which can be computed in parallel. Doing so eventually provides l partial sums in the larger format, which can be summed thanks to the addition operation in the larger format.

D. Hardware support of FormatOf operations

The IEEE 754-2008 standard mandates the implementation (in hardware or software) of a large number of *FormatOf* operations. These are operations that mix precisions, like binary32 + binary32 → binary64 [10, Section 5.4]. Full compliance with the standard requires to offer them all for the supported formats, but a software implementation is both tricky and costly [14]. The proposed operator provides hardware implementations for most of the FormatOf operations mixing binary$k$ and binary$2k$.

Let us now study the construction of this operator.

III. Operand alignment

A. Notations

In an MPFMAk, what matters most in terms of delay and silicon area is not k but the precision of the significands, which we note p for the binary$k$ multiplier operand, and q for the binary$2k$ addend and result.

We note d the exponent difference between the addend and the product.

B. Alignment cases

Figures 1, 2, and 3 describe the various cases of product and addend alignment. In these figures, we use $p = 5$ and $q = 12$ for illustration, and we represent the significand product of AB on $2p$ bits, and the significand of C on q bits. The purpose of each figure is to determine the size in bits of the intermediate sum required for each case. On each figure, we represent the extreme cases of alignment. For instance, the third alignment of Figure 1 illustrates that if $d \geq q+3$, all the bits of AB will only participate to the sticky bit, not to the sum.

These figures hold for any formats such that $q \geq 2p + 2$, which is the case for the standard precisions defined in Table I.

C. Subnormal support

As already mentioned, if either A, or B, or both are subnormals, the product AB nevertheless belongs in the normal range of the result format, binary$2k$. Managing these cases therefore resumes to normalizing this product, i.e. bringing its leading one in the leftmost position. This corresponds to a shift of up to $2p$ bits.

The shift distance is the sum of the leading zero counts (LZC) on the significands of A and B. These LZCs can be performed in parallel to the multiplication [13], which is why we prefer to normalize the product, and not the inputs A and B themselves.

Managing subnormal values of C has no overhead at all: If C is subnormal, then either $AB = 0$ and the result is C,

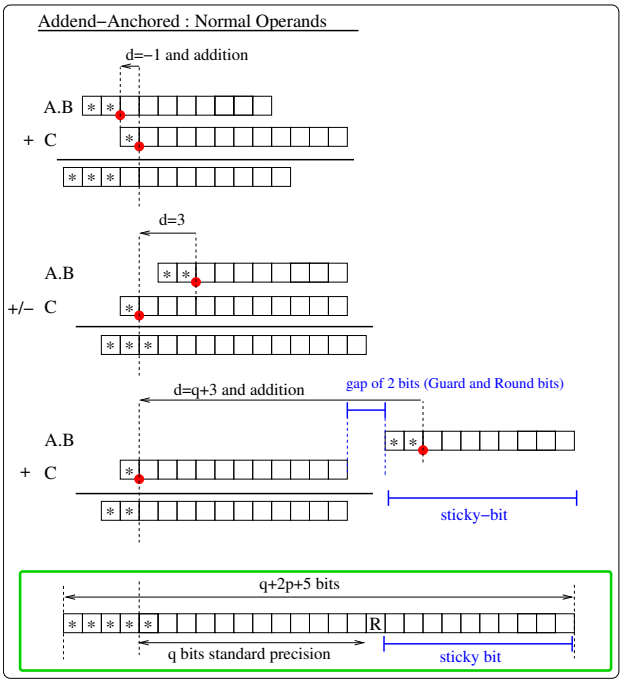

Figure 1. Operand alignment for the addend-anchored case of a mixed-precision FMA. The stars denote the possible positions of the leading bit.

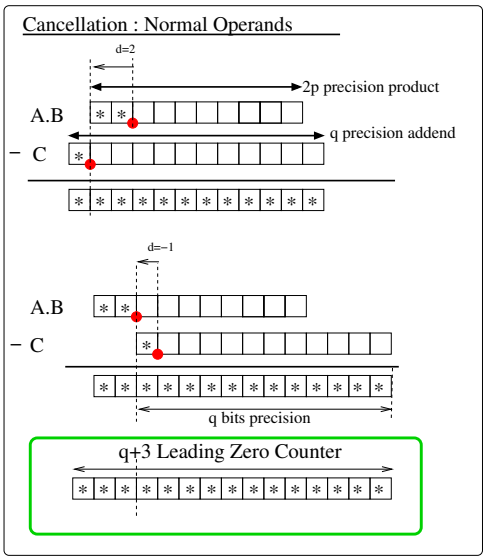

Figure 2. Operand alignment for the cancellation case of a mixed-precision FMA.

or $AB \neq 0$ and it is very far from the subnormal range, so the whole of C should only be taken into account as a sticky (second case of figure 3).

D. Support of binary$2k$ addition

One may remark in Figures 1, 2, and 3 that the product AB may be replaced with a binary$2k$ input D with very little impact on the datapath width. Specifically, only Figure 1 would need to be modified, with the $2p$ replaced with a q in the final sum size. Note that the maximum shift distance

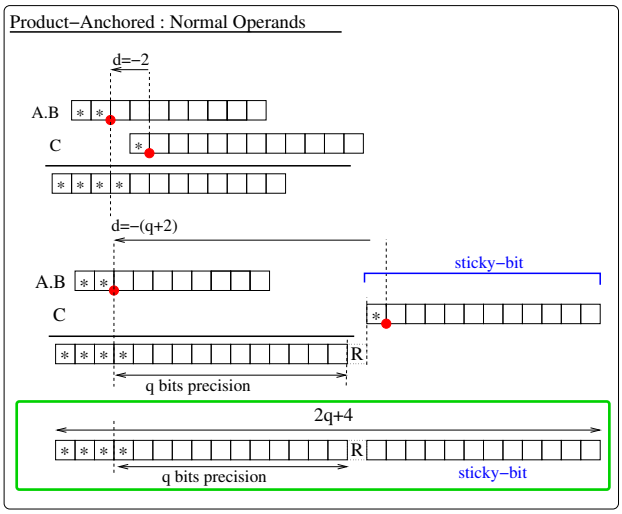

Figure 3. Operand alignment for the product-anchored case of a mixed-precision FMA.

is not modified (third case of Figure 1), only the amount of bits to be shifted. We claim that this entails a minor overhead, since q is only slightly larger than $2p$: 53 versus 48 for the MPFMA32, and 113 versus 106 for the MPFMA64.

However, we have to take care of the case when this second binary$2k$ input D is subnormal, since it replaces AB which could never be so. However it turns out this case adds very little logic. Specifically, the only new problem is the apparition of a subnormal as the result of a cancellation. The additional logic required only concerns the exponent datapath, to saturate the shift value to the minimal binary$2k$ exponent. This has a very small overhead.

E. binary$k$ FMA support

To support a classical FMAk operation, there are again a few multiplexers to add and constants to change on the exponent datapath, and again this represents a minor overhead. A more important modification is the addition of a rounding module to the binary$k$ format at the end of the datapath in addition to the module rounding to binary$2k$. The two formats have different exponent bias and mantissa precisions. The global latency is only slightly increased by the output muxing between the two formats.

IV. Architecture

A. Discussion on the alignment architecture

Figures 1, 2, and 3 defining the extremal alignment cases, there are two main approaches to building an architecture able to manage these cases:

1) distinguishing beween product-anchored and addend-anchored cases, or
2) anchoring the datapath on one operand , and aligning the second operand on it. In the standard FMA, the anchored operand is typically the product, since it will be available after the addend and it is larger than it. This allows the alignment shift to be processed concurrently to the significand multiplication.

The first solution implies that before entering the datapath we swap the operands based on their exponent. It is also possible to distinguish more cases (close/cancellation, far addend anchored and far product anchored, corresponding roughly to our three figures) in order to optimize a different datapath for each case [21], [24].

The greater operand is always statically driven at the left of the datapath, and the lower is shifted right for the alignment. In this case, the alignment shift itself is about q at most, leading to roughly $2q$ bits for the operands of the effective addition (Figures 1 and 3). However, the normalisation of a subnormal product may add $2p$ to this shift distance, as explained in III-C. To sum up, the critical path of this solution, before the effective addition, consists of a multiplier, a multiplexer to swap operands according to their order, and a shifter with (roughly) $2q + 2p$ output bits.

The second solution, also called statically product-anchored, is motivated as follows [12]. The multiplier is the largest and slowest unit, its latency is longer than that of the alignment shifter. Therefore, once computed, the product should not be shifted: instead, it is statically extended and driven to the middle q bits of a (roughly) $3q$ register. In parallel to the product computation, the addend operand is placed at the left of a (roughly) $3q$ register, then right-shifted for alignment.

This is the solution we chose to explore further here for the MPFMA, and it is illustrated on Figure 4. In this solution the critical path consists of the multiplier alone, hiding the latency of the (large) alignment shifter.

The datapath widths in this approach are obtained by superimposing figures 1 to 3.

To deal correctly with subnormal A and B, we have once again at least two solutions. The first one is to consider a $3q + 5$ bits datapath for the effective addition with at least a $2q$ LZA or LZC on its output. The second solution is to introduce a $2p$ shifter after the multiplier output, renormalizing the multiplication result. This second solution was prefered since it reduces the adder size to $2q+6$ bits and the LZC/LZA to $q + 3$ bits.

Finally, in this MPFMA architecture, the effective addition size is $2q + 6$ bits (112 bits for the MPFMA32 with $p = 24$ and $q = 53$). This is larger than in the FMAk ($3p+4$, 76 bits for $p = 24$) but much smaller than in the FMA$2k$ ($3q+4$ bits, 163 bits for $q = 53$).

We remark again that this $2q + 6$ addition is more than enough to manage the binary$2k$ addition. In practice, the overhead of managing this operation is essentially in multiplexers and exponent management.

B. Summary of overhead with respect to FMAk

This subsection focuses on comparing the overhead of a MPFMA over a standard FMA, in terms of area and latency. Energy could be a valued addition to this list and should be addressed in a future work.

We first compare statically product-anchored implementations. The main differences are

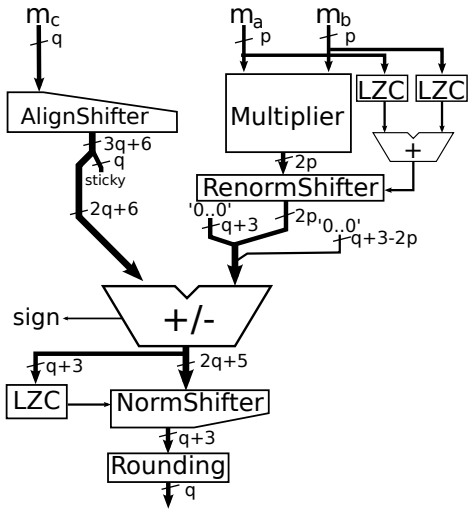

Figure 4. Baseline architecture for significand processing in a mixed-precision FMA

- the normalisation shifter of a standard product-anchored FMA is about $4p$ bits large, with a $3p$-bit maximal shift. For the MPFMA this shifter need to be $3q$-bit large with a $2q$-bit maximal shift. Comparing an MPFMA32 to a standard FMA32, this is a 159-bit datapath for a 106-bit maximal shift, versus a 96-bit datapath with a 72-bit shifter.
- Rather than a $3p$-bit adder datapath, we will need a $2q$-bit , this allow direct support of large precision addition. For FMA32 and MPFMA32 that is 72 bits against 106 bits.
- Concerning the needed leading zero count, if we suppose that both operators use an early normalization and that the product is normalized before entering the adder datapath, then a standard FMA needs a $2p$-bit LZC, while an MPFMA will need a q-bitLZC. These numbers are also applicable if the operator use a Leading Zero Anticipator on the adder entry, rather than an LZC on the adder output.
- In the MPFMA, we have to add a second rounding module for a q-bit result to the p-bit module of the standard FMA.

In the case of multipath architecture like those used by [21] or [24], the datapath widening would be repercuted on each of the path. For the close path, this implies a q-bit path rather than a $2p$-bit path (adder and Leading Zero Count prediction, plus new rounding module). For the far paths (both product-anchored and addend-anchored), that would mean a $2q$-bit shifter with a q bit adder, rather than a $3p$-bit shifter with a $2p$-bit adder.

V. Implementation and evaluation

A. The Kalray processor context

The Kalray processor is a high-performance embedded processor with primary support of binary32 and secondary support of binary64.

It has a unified register file: the same 32-bit registers may hold integer or FP data. Two consecutive 32-bit registers may be paired and accessed as a single 64-bit register. For instance, the same two registers may be used as two binary32 multiplicands for an MPFMA, or as a single binary64 register for a binary64 addition.

It is important to notice that the MPFMA register footprint is not very different from that of a standard FMA : they both need 3 register read ports and 1 register write port. However the total input length is 128 bits for the MPFMA versus 96 bits for the FMA, and the total output length is 64 versus 32. Energy consumption varies as the cube of the number of ports [22], and is only linear in the width of the registers, so the impact of the increase in input/output bandwidth is moderate.

B. Development and testing

The MPFMA32 has been designed using a derivative of the FloPoco framework [4]. This framework strongly assists the pipelining work, guaranteeing a correct-by-construction pipeline out of a functional combinatorial operator.

This operator was submitted to extensive testing, using FloPoCo facilities for test-bench generation. FloPoCo may generate test vectors using a mixture of truly random inputs, random inputs biased towards rare specific situations (for instance cancellations and subnormals in the case of the MPFMA), and specific corner cases and regression tests. The generated test benches check the actual output of the operator against its expected behaviour. This expected behaviour is programmed at a very high level, in terms of exact operations using MPFR [6] and rounding: this is as close as possible to the specification of the operation in the IEEE-754 standard. Our implementation also tests correct IEEE-754 exceptions (with pre-rounding version of the IEEE underflow exception). The standard operations provided by the MPFMA have also been successfully submitted to IEEE 754 Compliance Checker and TestFloat [8].

C. Synthesis results and comparisons

Here we compare an MPFMA32 to an FMA32, an Add64, and an FMA64, all designed with the same design effort, and in the same processor context and with the same constraints.

Synthesis results are provided in Table V-C. For each operator, we performed iterative synthesis to approximate the best reachable latency, but with a 28nm component library optimized for area.

The MPFMA32 operator in this table is also capable of standard binary32 FMA operation and binary64 addition. Removing support for one of these options saves only a few hundred μm^2.

As this table shows, the MPFMA32 adds only one third to the area of an binary32 FMA for the same frequency. The additional area represents less than half the size of a binary64 addder. All this is consistent with the estimations of previous section.

D. Related work with respect to FMA optimizations

We acknowledge that our implementations are not as optimized as they could be.

Operator	best latency (ns)	area (μm^2)
FMA32	3.5	10566
FMA64	3.5	24500
Add64	3.5	8800
MPFMA32	3.5	14000

Table II
SYNTHESIS RESULTS FOR 28NM TECHNOLOGY (HIGH DENSITY). ALL OPERATORS ARE DIVIDED INTO 3 PIPELINE STAGES

We should point out that we didn't even use one of the most standard technique, the use of carry-save representation. This is due to the context in which this work took place. We were extending a fixed-point processor, and had the constraint of using, for significand multiplication, the existing fixed-point multiplier, for which it was not possible to obtain a carry save result. In future revisions of this processor we hope to relax this constraint.

Many architectural optimizations have been used for the classical FMA that could be relevant for the FPFMA:

- It is possible to rearrange and fuses the addition, normalisation and rounding steps [12].
- We have already mentionned the triple path approaches [20], [21].
- It may be better to split the implementation of the FMA between multiplier and adder [13]. This leads to a larger FMA latency than a monolithic FMA (in this article, 8 cycles, 4 by unit), but to a better overal performance: as the author points out, when iteratively computing dot product, the data dependency is only on the addition, so we shouldn't suffer any latency due to the multiplier, which we do in a monolithic FMA.
- Similarly, it is possible to reduce the latency of the FMA used as a floating-point adder [3].
- The floorplan of the FMA is critical for high-performance implementations [2].

The relevance of such optimizations, which often trade off area for latency, depends on a given processor context, and studying them is beyond the scope of this article.

VI. CONCLUSION AND FUTURE WORK

In low-power, DSP-oriented embedded processors, an MPFMA32 turns out to be a cost-effective alternative to a full binary64 floating-point unit. In high-end processors, an MPFMA64 could enable a low-cost transition towards the quadruple precision (binary128) demanded by some large-scale physics simulations.

Future work will include a thorough study of further possible optimizations and their relevance with respect to area, speed, and power consumption.

The availability of the classical FMA has lead to a number of clever algorithms to implement efficiently all sorts of low-level operations, from the initial division and square root to constant multiplication, complex operations, polynomial evaluation, range reductions for elementary functions, multiple-precision operations, and others [16]. We could expect the same with the proposed operators, and future work will be to explore such algorithms.

REFERENCES

[1] I. Babuška. Numerical stability in mathematical analysis. In *Proceedings of the 1968 IFIP Congress*, volume 1, pages 11–23, 1969.

[2] M. Boersma, M. Kröner, Ch. Layer, P. Leber, S. M. Müller, and K. Schelm. The POWER7 binary floating-point unit. In *Proceedings of the 20th Symposium on Computer Arithmetic*. IEEE, July 2011.

[3] Javier D. Bruguera and Lang Tomas. Floating-point fused multiply-add: Reduced latency for floating-point addition. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, ARITH '05, pages 42–51. IEEE Computer Society, 2005.

[4] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, August 2011.

[5] W. R. Dieter, A. Kaveti, and H. G. Dietz. Low-cost microarchitectural support for improved floating-point accuracy. *IEEE Computer Architecture Letters*, 6(1):13–16, January 2007.

[6] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):13:1–13:15, June 2007.

[7] Guenter Gerwig, Eric M. Schwarz, and Ronald M. Smith. Fused multiply add split for multiple precision arithmetic. US Patent 0061392 A1, september 2005.

[8] John R. Hauser. TestFloat. http://www.jhauser.us/arithmetic/TestFloat.html.

[9] Libo Huang, Li Shen, Kui Dai, and Zhiying Wang. A new architecture for multiple-precision floating-point multiply-add fused unit design. In *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 69–76, Washington, DC, USA, 2007. IEEE Computer Society.

[10] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, August 2008.

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

[12] T. Lang and J-D. Bruguera. Floating-point fused multiply-add with reduced latency. *Proceedings of the 2002 IEEE International Conference on Computer Design : VLSI in Computers and Processors*, 2002.

[13] David R. Lutz. Fused multiply-add microarchitecture comprising separate early-normalizing multiply and add pipelines. In *IEEE Symposium on Computer Arithmetic*, pages 123–128, 2011.

[14] David R. Lutz and Neil Burgess. Overcoming double-rounding errors under IEEE 754-2008 using software. In *Asilomar Conference on Signals, Systems, and Computers*, pages 1399–1401, November 2010.

[15] Michael J. Mantor, Jeffrey T. Brady, Daniel B. Clifton, and Christopher Spencer. Method and system for multi-precision computation. US Patent 2011/0055308, March 2011.

[16] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhauser Boston, 2009.

[17] A. Neumaier. Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen. *ZAMM*, 54:39–51, 1974. In German.

[18] M. Pichat. Correction d'une somme en arithmétique à virgule flottante. *Numerische Mathematik*, 19:400–406, 1972. In French.

[19] D. M. Priest. Algorithms for arbitrary precision floating point arithmetic. In *10th Symposium on Computer Arithmetic*, pages 132–144. IEEE, 1991.

[20] Eric Charles Quinell. *Floating-Point Fused Multiply-Add Architectures*. PhD thesis, The University of Texas at Austin, May 2007.

[21] Eric Quinnell, Earl E. Swartzlander, and Carl Lemonds. Three-path fused multiply-adder circuit. US Patent 2008/0256150 A1, april 2008.

[22] Scott Rixner, William J. Dally, Brucek Khailany, Peter Mattson, Ujval J. Kapasi, and John D. Owens. Register organization for media processing. In *HPCA6*, pages 375–386, 2000.

[23] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part I: Faithful rounding. *SIAM Journal on Scientific Computing*, 31(1):189–224, 2008.

[24] Peter-Michael Seidel. Multiple path ieee floating-point fused multiply-add. In *46th Midwset Symposium on Circuits and Systems*, december 2003.