# A Mixed-Precision Fused Multiply and Add

Nicolas Brunie
Kalray
Email: nicolas.brunie@kalray.eu

Florent de Dinechin
LIP (CNRS/INRIA/ENS-Lyon/UCBL)
Université de Lyon
Email: florent.de.dinechin@ens-lyon.fr

Benoit de Dinechin
Kalray
Email:benoit.dinechin@kalray.eu

***Abstract*—The floating-point fused multiply and add, computing R=AB+C with a single rounding, is now an IEEE-754 standard operator. This article investigates variants in which 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). Like the standard FMA operator, the proposed mixed-precision operator computes AB+C with a single rounding, and fully support subnormals. With minor modifications, it 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 shown to be only one third more than that of a classical FMA in the smaller format. Besides, we show that such a mixed-precision FMA, although not mentioned in existing standard (IEEE 754, C and Fortran), is perfectly compliant to these standards.**

**For DSP and embedded applications, a mixed binary32/binary64 FMA will enable binary64 computing where it is most needed, at a small cost overhead with respect to current binary32 FMAs, and with fewer data transfers, hence lower power than a pure binary64 approach. In high-end processors, a mixed binary64/binary128 FMA could provide an adequate solution to the binary128 requirements of very large scale computing applications.**

**Keywords Floating-point; fused multiply-add; dot product; mixed precision.**

## I. Introduction

The fused multiply-add operator (FMA) is now an IEEE-754-2008 standard. 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 [8], 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) build their floating-point unit around the FMA. This operator will come to the legacy IA32 instruction set with the SSE5 and AVX extensions from AMD and Intel respectively.

A multiple-precision FMA has been proposed in [4]. It is able to compute either one binary64 (double precision), or two binary32 (single precision) FMA operations in parallel. However, in both cases, each operation uses the same format for all inputs and output. 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.

In this work, we consider a *floating-point* mixed-precision FMA, or MPFMA. For $k \in \{16, 32, 64\}$, the MPFMA$k$ 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 below). We show that such an operator may also take cares of two operations that are somehow simpler: the standard binary$k$ FMA, and the standard binary$2k$ addition.

This article is organized as follows. Section II motivates this operator from an applicative point of view. It shows in particular that the proposed MPFMA, although not mentioned in existing arithmetic or language standard, is perfectly compliant with these standards, which do allow precision mixing.

The following sections study the construction of an MPFMA$k$ with subnormal support, and support of binary$k$ FMA and binary$2k$ addition. Section III explicits the data alignment requirements, enabling in Section IV a fully parametric description of a baseline FPFMA architecture. The purpose of this work is to assess the cost of the proposed operator compared to a standard FMA. In Section V, we present an actual implementation of the MPFMA32 in the context of a high-performance embedded processor with primary support of binary32 and secondary support of binary64. The MPFMA32 is compared against classical FMA32, FMA64, and binary64 addition, all designed with comparable optimization effort.

Many architectural optimizations have been used for the classical FMA. Among them, [7] rearranges and fuses the addition, normalization and rounding steps. The multipath floating point adder optimization [10] can be applied to the FMA [13, chapter 5]. All these optimizations are typically trading off area for latency, and many of them could be applied to the MPFMA. However, the relevance of such optimizations depends on a given processor context, and studying them is beyond the scope of this article.

## II. Motivations

The MPFMA$k$ is relevant for computing kernels based on sums of products of binary$k$ numbers. In such cases, the MPFMA$k$ will provide an accuracy for the result that is close to binary$2k$, at a cost that is close to a binary$k$ FMA (this claim will be substantiated below). However, the idea of using an extended precision acumulator in such cases has been the subject of much previous work.

## A. Alternative approaches

On the hardware side, Kulisch advocated augmenting the processors with a long accumulator that would enable exact accumulation and dot product [6]. 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 accumulation using Kulisch's proposition.

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 [11], Neumaier [9], Priest [12], and Rump, Ogita, and Oishi [15]. They are reviewed in [8, 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 [2], [3] for better performance. An FPFMA will provide this better performance, and has the additional advantage of an extended exponent range, not only extended precision. This reduces the risk of returning $\infty$ due to an intermediate overflow when the result should be representable.

## B. Standard compliance

One initial motivation of the FPFMA operator was that is fitted neatly in the datapath of a DSP-oriented processor already offering a fixed-point multiply-accumulate with 32-bit multiplier operands and 64-bit accumulators.

However, it turned out that this operator can be used in a standard-compliant way for a large class of code. 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]`$\times$`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]`. This is exactly the behaviour of the proposed MPFMA.

| 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 [5].

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.

Let us now study the construction of this operator.

## III. OPERAND ALIGNMENT

### A. Notations

In an FPFMA$k$, what matter 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.

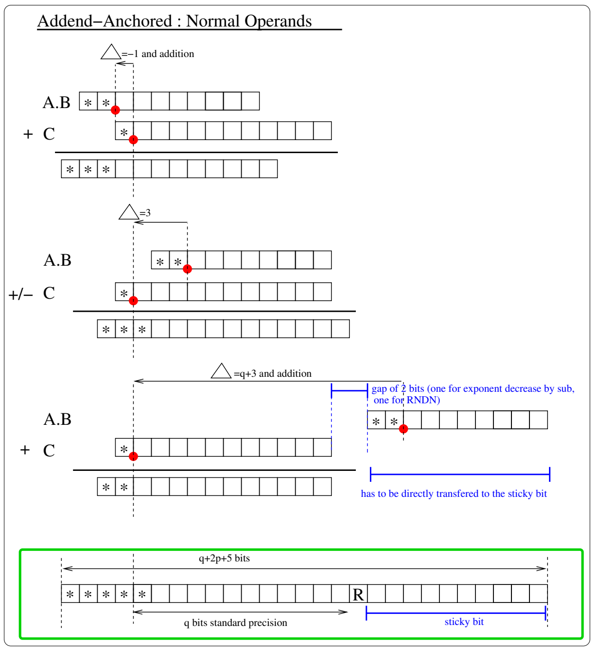
### B. Alignment cases

Figures 1, 2, and 3 describe the various cases of product and addend alignment. They cover the extreme cases as long as $q \geq 2p+2$, which is the case for the standard precisions defined in Table I. These diagrams will help us define the sizes of the datapaths of the architecture presented below on Figure 4.

### 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, which is why we



Addend−Anchored : Normal Operands
△=−1 and addition
A.B
+ C
△=3
A.B
+/− C
△=q+3 and addition
gap of 2 bits (one for exponent decrease by sub, one for RNDN)
A.B
+ C
has to be directly transfered to the sticky bit
q+2p+5 bits
R
q bits standard precision
sticky bit

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

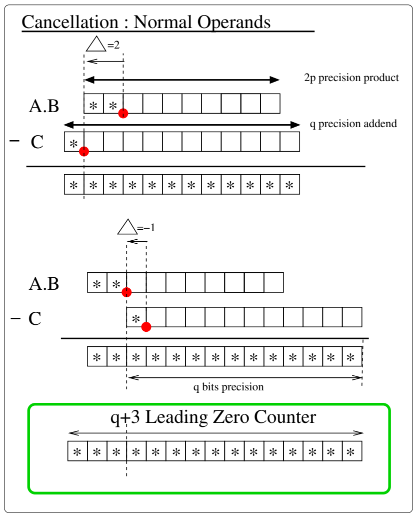


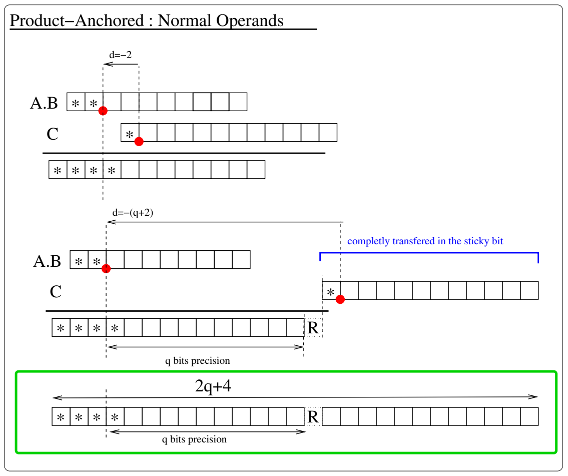
Figure 2. Operand alignement for the cancellation case of a mixed-precision FMA.

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 $C$, 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. binary2$k$ addition support

On may remark in Figures 1, 2, and 3 that the product $AB$ may be replaced with a binary2$k$ input $D$ with very little



Product−Anchored : Normal Operands
d=−2
A.B
C
d=−(q+2)
completly transfered in the sticky bit
A.B
C
R
q bits precision
2q+4
R
q bits precision

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

impact on the datapath. Specifically, only Figure 1 would need to be modified, with a $2p$ replaced with a $q$.

We have to take care of the case when this second binary2$k$ 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. With the LZC and NormShifter already in place for the normal case, the additional logic required only concerns the exponent datapath, to saturate the shift value to the minimal binary2$k$ exponent. This has a very small overhead.

### E. binary$k$ FMA support

To support a classical FMA$k$ 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 binary2$k$. 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 (typically the product, which is larger in the classical FMA), and aligning the second operand on it.

The first solution implies that before entering the datapath we swap the operands based on their exponent. 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 operand according to their order, and a shifter with (roughly) $2q + 2p$ output bits.

If latency is a concern [7], the second solution is often prefered. Indeed, the multiplier is the largest and slowest unit, its latency is longer than that of the alignment shifter. In a product-anchored solution, the product, once computed, is statically extendend 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 have chosen for the MPFMA, due to the critical latency constraints in the embedded processor for which it was designed. In this solution the critical path consists of the multiplier alone, hiding the latency of the (large) alignment shifter.

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 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 FMA$k$ ($3p + 4$, 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.

## V. Evaluation

This operator has been designed in VHDL and carefully tested, using constructed special-case tests and millions of random test vectors. We acknowledge that our implementations are not as optimized as they could be. However they allow us to compare an MPFMA32 to an FMA32, an Add64, and an FMA64, all designed with the same design effort, and in the same processor context with the same constraints. Synthesis results are provided in Table V. For each operator, we performed iterative synthesis to approximate the best reachable latency, but with a 28nm component library optimized for area.

For the MPFMA32, the operator synthesized is also capable of standard binary32 FMA operation, and binary64 addition. Removing support for one of these options saves only a few hundred $\mu m^2$. As we can see, we add only one third to the area of an binary32 FMA. The additional area represents less than half the size of a binary64 adddder.

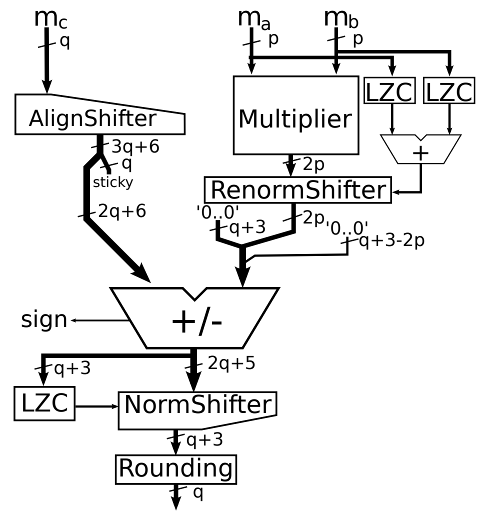


Figure 4. Baseline architecture for significand processing in a mixed-precision FMA

| Operator | best latency (ns) | area ($\mu m^2$) |
|---|---|---|
| FMA32 | 3.5 | 10566 |
| FMA64 | 3.5 | 24500 |
| Add64 | 3.5 | 8800 |
| MPFMA32 | 3.5 | 14000 |

Table II
Synthesis results for 28nm technology. All operators are divided into 3 pipeline stages

All sorts of optimizations like the ones described in [7], [13] or [14] could be used for the FPFMA. 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 may try and relax this constraint.

The three paths optimization would certainly lead to a large increase in silicium area for a latency reduction that was not necessary to reach the targeted frequency in our case. However this is also worth of future investigations.

## VI. Conclusion and future work

In low-power, DSP-oriented embedded processors, an MPFMA32 could 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, range reductions, multiple-precision operations, and others [8]. 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] 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.

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

[4] 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.

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

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

[7] 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.

[8] 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.

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

[10] Stuart F. Oberman and Michael J. Flynn. Reducing the mean latency of floating-point addition. *Theoretical Computer Science*, 1998.

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

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

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

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

[15] 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.