Floating–Point Fused Multiply–Add: Reduced Latency for Floating-Point Addition [*]

Javier D. Bruguera
Dept. Electronic and Computer Eng.
University of Santiago de Compostela
15706 Santiago de Compostela, Spain
bruguera@dec.usc.es

Tomás Lang
Dept. of Electrical Eng. and Computer Science
University of California at Irvine
Irvine CA 92697, USA
tlang@uci.edu

Abstract

In this paper we propose an architecture for the computation of the double–precision floating–point multiply–add fused (MAF) operation $A + (B \times C)$ that permits to compute the floating–point addition with lower latency than floating–point multiplication and MAF. While previous MAF architectures compute the three operations with the same latency, the proposed architecture permits to skip the first pipeline stages, those related with the multiplication $B \times C$, in case of an addition. For instance, for a MAF unit pipelined into three or five stages, the latency of the floating–point addition is reduced to two or three cycles, respectively.

To achieve the latency reduction for floating-point addition, the alignment shifter, which in previous organizations is in parallel with the multiplication, is moved so that the multiplication can be bypassed. To avoid that this modification increases the critical path, a double-datapath organization is used, in which the alignment and normalization are in separate paths. Moreover, we use the techniques developed previously of combining the addition and the rounding and of performing the normalization before the addition.

1 INTRODUCTION

The floating–point unit of several recent commercial general–purpose processors include as a key feature a unified floating–point multiply–add fused (MAF) unit [5, 6, 12, 14]. This unit executes the single or double–precision multiply–add, $A+(B\times C)$, as a single instruction, with no intermediate rounding. The standard operations floating-point add and floating-point multiply are performed using this unit by making $C = 1$ or $A = 0$. Moreover, floating-point division might also be implemented using the MAF unit.

The implementation of multiply-add-fused has two advantages [7]: (1) the operation $A + (B \times C)$ is performed with only one rounding instead of two and (2) by sharing several components, there is reduction in the delay and hardware required for multiply–add operations, with respect to the traditional implementation performing first a multiplication and then an addition. On the other hand, the latency of the floating–point addition and multiplication is larger when implemented in a MAF unit than in a floating–point adder and a floating–point multiplier. Moreover, if only one MAF unit is provided, it is not possible to perform concurrently additions and multiplications which is possible for the case in which there is one floating-point adder and one floating-point multiplier.

The main characteristic of the previous implementations of MAF units [2, 5, 7, 13, 15] is that the alignment is done in parallel with the multiplier, by placing the significand of A all the way to the left of the binary point of $B \times C$. In this way, the alignment is implemented always by right shifting A. After that, the carry-save representation of $B \times C$ and the aligned A are added, normalized and rounded. Several modifications have been proposed to reduce the delay. In [8], the delay is reduced by anticipating the normalization before the final addition and by combining the final addition and the rounding using a dual adder. In [15], the delay is reduced by considering multiple exclusive parallel computation paths in the implementation, depending on the alignment shift required, and by using an integrated addition and rounding implementation with variable position rounding. Additional reduction can be achieved by considering a variable latency implementation. Note that, in all these implementations,

[*]Work supported in part by Ministry of Science and Technology of Spain and FEDER under contract TIC2001-3694-C02

the latencies of all the three operations, floating–point addition, multiplication and MAF, are the same.

In this paper we propose a MAF architecture that reduces the latency of the floating-point addition with respect to that of the floating–point multiplication and floating–point MAF. The key point to reduce the latency of the addition is not to perform the alignment in parallel with the multiplier, but later; in this way, the multiplication step is bypassed in addition.

Since the alignment is delayed, to avoid having two shifters in the critical path, the alignment and normalization shifters, the architecture uses a double–datapath organization with combined addition and rounding. This organization, which defines two parallel paths in the unit according to the effective operation being performed and the exponent difference, has been previously employed in floating–point adders [11], to take advantage of the fact that the full–length alignment shift and the full–length normalization shift are mutually exclusive, and only one of such shifts needs ever appear on the critical path. Moreover, combining the addition and rounding, using a dual adder that computes simultaneously the sum and the sum plus 1, permits to reduce the latency, by eliminating the addition due to the rounding. In floating–point addition this has been achieved by interchanging the order of normalization and rounding, in such a way that the result of the addition is first rounded and then it is normalized.

However, in the MAF unit the interchange of the rounding and the normalization, and consequently the utilization of the combined addition and rounding, is not straightforward, since the rounding position is not known before the normalization. Therefore, we perform the normalization before the addition[1]. This has been used previously in [8], and requires a careful design of the LZA and the normalization shifter to avoid additional delays. In the MAF unit we propose this approach is implemented in both datapaths.

In summary, the proposed MAF architecture uses some techniques previously used in floating–point adders and floating–point MAF units; but there are important differences with respect to both units. In floating-point addition and in the multiple path MAF, the normalization is not performed before the addition. Moreover, in all the previous MAF implementations, the alignment of A is done in parallel with the multiplier; whereas in the proposed MAF, it is done after it. On the other hand, one of the two paths of the double datapath MAF, the path we call CLOSE datapath, is quite similar to the single datapath MAF, with some significant differences: the 2–bit alignment shifter and the calculation of the alignment shift amount. The other path, which we call FAR datapath, has been completely designed for this organization. The main differences with respect to the FAR datapath of a floating–point adder are that there are two alignment shifters to align the carry–save representation of the product, the normalization is performed before the addition, and the calculation of the alignment and normalization shift amounts is overlapped with the shifts themselves. Moreover, some modules in the architecture have been redesigned to be tailored to the requirements of the proposed MAF organization.

[1]In [15], a unit is used that determines the position of rounding and performs a combined addition with rounding in a variable position; however, this special implementation is not described and no reference is provided. Consequently, we are not able to estimate the impact of this unit on the overall delay.

In the resulting architecture, A is used later than B and C, which permits to bypass the first stages of the MAF unit when evaluating a floating–point addition and to reduce its latency. Moreover, it maintains the same performance as previous implementations when executing multiplications or MAF operations. For instance, for a MAF unit pipelined into three stages [7, 12], the latency for the floating–point addition is reduced from 3 to 2 cycles. If the MAF unit is pipelined into a larger number of stages, say five stages [5, 14], the latency is reduced from 5 to 3 cycles. Note that, the reduced latency for the addition will affect the scheduling of the operations in the floating-point unit, since an addition can collide in the pipeline with a MAF operation or multiplication issued earlier. This restriction has to be taken into account in the instructions issue policy to avoid a degradation of the performance.

2 FLOATING–POINT MAF

Before describing the MAF we propose, we outline the main features of the MAF architecture previously proposed in [8], which we use as a starting point to explain the modifications introduced to reduce the latency of the floating–point addition. To be more specific in the descriptions, we consider the IEEE double-precision format, but we do not discuss neither special nor denormalized numbers. The necessary steps in the traditional implementation of the MAF unit [7], used in some recent floating–point units of general–purpose processors [5, 6, 12], are:

1. Multiplication of B and C, to produce a carry–save product, in parallel with inversion and alignment of A. The alignment is implemented as a right shift by placing A to the left of the MSB of $B \times C$. The shift amount is $56 - d$, being $d = exp(A) - (exp(B) + exp(C))$.

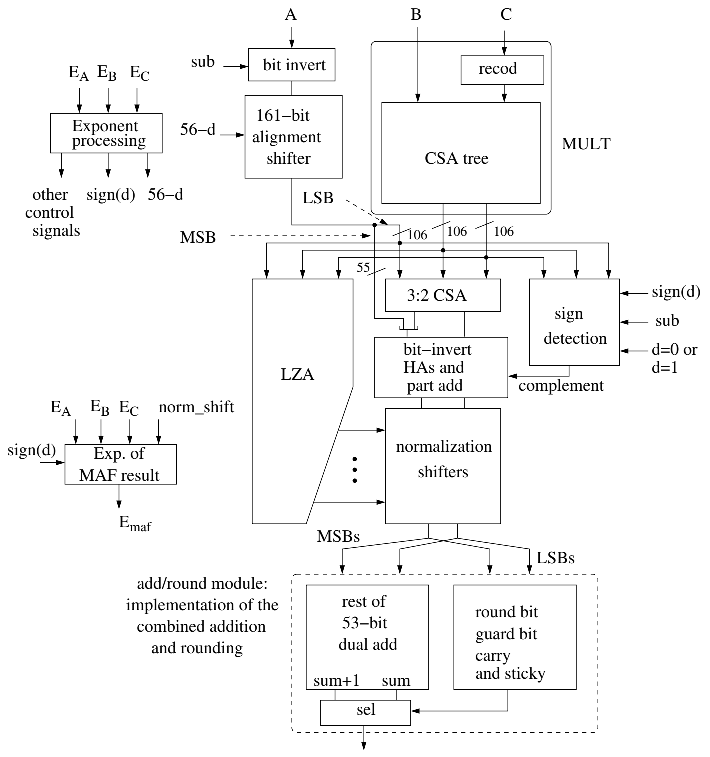

Figure 1. MAF with normalization before addition

2. Addition of the 106–bit $B \times C$ and the 161–bit aligned A. This requires a 106–bit 3:2 CSA plus a 161–bit adder. In parallel, the normalization shift amount is determined.

3. Normalization and rounding[2]. A massive normalization is not only needed during subtraction with cancellation, but also because of the way the alignment is performed.

An alternative MAF architecture, shown in Figure 1, has been recently proposed [8]. The goal of this architecture is to reduce the overall delay and the latency of the MAF operation. As done in floating–point addition and multiplication, the basic idea is to combine addition and rounding and, in consequence, to interchange the order of normalization and rounding[4, 11, 16]. Since in the MAF the rounding position is not known before the normalization[3], we perform the normalization before the addition. This requires to overlap the operation of the LZA and the normalization shift, in such a way that the shift amount is obtained starting from the most–significant bit. The main differences with respect to the traditional implementation are:

- The sum and carry vectors of the 3:2 CSA are normalized before the addition. In this way, the result (after the addition) is always normalized and the rounding is simpler.

- The add/round module consists of a 53-bits dual adder which computes the *sum* and the *sum+1*. The least–significant part at the output of the normalization shifter is used to select the correct output of the dual adder.

- To avoid a negative result, the sign of the output of the 3:2 CSA is detected and, if negative, the sum and carry words at the output of the CSA are complemented. In parallel, a 3–input Leading Zero Anticipator (LZA)[4] determines the normalization shift amount from the carry–save product $B \times C$ and the aligned A. This LZA has been modified to obtain the shift amount starting from the most–significant digit, to overlap with the operation of the shifter.

- Since the sign detection as well as the part of the LZA that cannot be overlapped have a significant delay, it is convenient to place the HA and some parts of the dual adder before the normalization.

The delay reduction achieved with this organization has been estimated in about 15% to 20% [8]. Both architectures can be divided into three steps,

1. Alignment shift and multiplication.

2. In the basic architecture, 3–2 CSA, 161–bits adder and LZA, and complementer. In the architecture with normalization before addition, the three parallel paths (sign detection, 3:2 CSA plus HA plus first levels of the dual adder, and LZA) and the normalization, which overlaps with the LZA.

3. Normalization and rounding for the basic architecture, and rest of the dual adder for the architecture with normalization before addition.

Two alternatives have been used for the pipelining of the basic architecture: (1) a pipelining into three stages [7, 12], where each steps corresponds to one pipeline stage, and (2) a pipelining into five stages [5], where steps 1 and 3 are divided into two pipeline stages each. Similarly, the architecture with normalization before

[2]In [9], a variation is proposed in which the rounding is delayed to the next dependent operation.

[3]Although this is also true for addition, in that case the bits introduced by the left shift normalization are all zeros, so no round up is required when massive normalization is performed

[4]In [3] the implementation of a 5–input LZA is presented

addition can be also pipelined into three or five stages. However, because of the delay reduction achieved with respect to the basic architecture, it could be pipelined into four stages while maintaining the same cycle time. In any case, the latency is the same for the floating–point addition, multiplication and MAF: three, four or five cycles depending on the particular implementation.

3 GENERAL STRUCTURE OF THE PROPOSED MAF

The objective of the proposed MAF architecture is to reduce the latency of the floating–point addition while maintaining the critical path delay of the single–datapath MAF. The latency reduction is achieved by performing the alignment of A after the multiplier instead of in parallel with it, as in previous implementations. In this way, the multiplier can be bypassed in case of an addition. However, placing the alignment shifter after the multiplier would have the alignment and the normalization shifters in the critical path and, consequently, a larger delay. To avoid this, we use a double–datapath organization with combined addition and rounding, previously used to reduce the latency of the floating–point addition [11], that permits to take advantage of the fact that the full–length alignment shift and the full–length normalization shift are mutually exclusive, and only one shift need ever appear in the critical path[5].

In floating–point addition, the datapath is split into two paths: the *CLOSE* datapath that computes the effective subtractions for an exponent difference $|d| \leq 1$, and the *FAR* datapath that computes all the effective additions and the effective subtractions for an exponent difference $|d| > 1$. In this way, the FAR datapath requires a full alignment shift and the normalization shift is, at most, of 1 bit; whereas the CLOSE datapath requires a full normalization shift and only a 1–bit alignment shift. Although slightly different partitioning criteria have been recently proposed to avoid the rounding step in the CLOSE datapath [10, 16], because of the type of alignment performed in the MAF operation, rounding is always required in effective subtraction. Then, the criterion we used for partitioning the MAF datapath into CLOSE and FAR has been adapted from the scheme in [11].

Consequently, considering that the exponent difference for the MAF operation is $d = exp(A) - (exp(B) + exp(C))$[6] and taking into account that the multiplication can produce an overflow, the CLOSE datapath is used for effective multiply–subtractions with (1) an exponent difference $d = 0, 1$, (2) an exponent difference $d = 2$ and $OVF(B \times C) = 1$, and (3) an exponent difference $d = -1$ and $OVF(B \times C) = 0$. The FAR datapath is used for the remaining cases. Note that this partitioning is also valid for the floating–point addition, since in this case $d = exp(A) - exp(C)$ and $OVF(B \times C) = 0$, and the CLOSE datapath is used only for effective subtractions with $|d| \leq 1$, as in [11].

Moreover, as said in section 2, the combined addition and rounding permits to reduce the latency by eliminating the addition due to the rounding and performing the normalization before the rounding. However, in a MAF operation, as the rounding position is unknown until the normalization has been carried out, the normalization is performed before the addition [8].

Then, the proposed MAF makes use of several techniques already used in floating–point adders and MAF units (double–datapath organization with combined addition and rounding and anticipation of the normalization before the addition), which together allow to reduce the latency of the floating–point addition when implemented in a MAF unit. Although these techniques have been previously used, the proposed MAF architecture presents significant differences with respect to the double–datapath adders and the single–datapath and the multiple–datapath MAF units. The resulting architecture is shown in Figure 2 and the differences can be summarized in the following points,

1. With respect to the double—datapath adder and the multiple–datapath MAF: (1) the normalization is performed before the addition to make possible the use of the combined addition and rounding and, to avoid large delays, the calculation of the normalization shift amount is overlapped with the normalization shift. (2) There are two full alignment shifters operating in parallel in the FAR datapath, needed for the alignment of the carry-save output of the multiplier. (3) The floating–point adder and the multiple–datapath MAF use two dual adders, whereas because of the anticipation of the normalization before the addition, the proposed MAF requires only one dual adder. Moreover, whereas the dual adder in our proposal is 53-bit width, adders in the multiple–datapath MAF are 53–bit and 160-bit width.

2. With respect to the single–datapath MAF: (1) the architecture of the CLOSE datapath is similar to

[5]Depending on the relative alignment of the product $B \times C$ with significand A, the alignment shift is not necessary and can be skipped. This reduces the latency for these cases, but at the expense of having a variable latency implementation [15]

[6]Note that the *actual* exponent difference might be $d_{act} = d - 1$, since the multiplication can produce an overflow

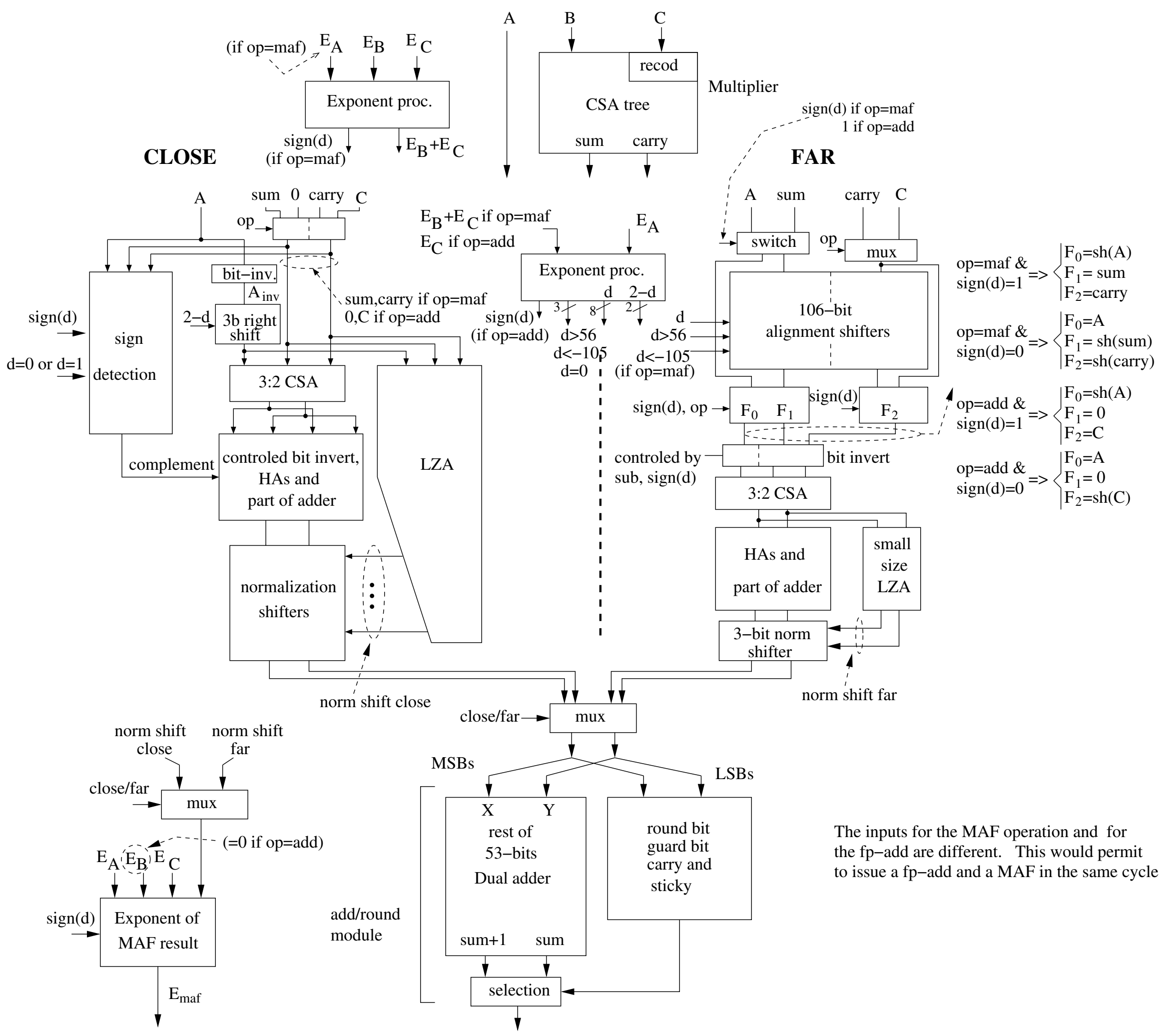
CLOSE
FAR
Exponent proc.
CSA tree
recod
Multiplier
sign detection
3:2 CSA
LZA
controled bit invert, HAs and part of adder
normalization shifters
106–bit alignment shifters
HAs and part of adder
small size LZA
3–bit norm shifter
mux
rest of 53–bits Dual adder
round bit guard bit carry and sticky
selection
Exponent of MAF result
op=maf & sign(d)=1 => F0=sh(A), F1= sum, F2=carry
op=maf & sign(d)=0 => F0=A, F1= sh(sum), F2=sh(carry)
op=add & sign(d)=1 => F0=sh(A), F1= 0, F2=C
op=add & sign(d)=0 => F0=A, F1= 0, F2=sh(C)
The inputs for the MAF operation and for the fp–add are different. This would permit to issue a fp–add and a MAF in the same cycle

Figure 2. General Scheme of the proposed MAF.

the single–datapath MAF, but with a 2–bit alignment shifter and with differences in the calculation of the alignment shift amount. Moreover, the normalization shift is of 108 bits instead of the 162–bit normalization shifter of the single–datapath MAF. (2) The FAR datapath does not exist in the previous MAFs, it has been been completely designed for this organization.

Moreover, the calculation of the alignment shifts amounts has been tailored to the requirements of delay imposed by the proposed MAF architecture, overlapping these calculations with the operation of the shifters.

In case of a floating–point addition, the multiplication stage is bypassed, operand A inputs directly to the alignment stage, C inputs the alignment shifter instead of the carry word of the output of the multiplier, and the sum word of the output of the multiplier is forced to 0. This is equivalent to making $B = 1$, since the result of the multiplication is C. However, because of the hardware organization required for the MAF operation, the floating–point addition is implemented differently than in a standard double–datapath floating–point adder [11, 16]. Note that, as there are two alignment shifters in the FAR datapath and A_{inv} is always rigth–shifted in the CLOSE datapath, the swap of the operands before the alignment, typical in standard floating–point adders, is not needed now and, therefore, the calculation of the exponent difference is not in the critical path. But, on the other hand, the alignment shifters are larger, one more level than in an adder, and there is an additional 3:2 CSA in the critical path.

The description of the architecture can be split into several parts: multiplication, alignment (including exponent processing), normalization, and combined addition and rounding. Some of these parts and components, multiplication, normalization in the CLOSE path, LZA, and combined addition and rounding, are already implemented in the single–datapath MAF with normalization before rounding, and described in detail in [8]. Then, they are not described here. Therefore, in the next subsections, the main characteristics of the remaining parts of the architecture are summarized. However, we do not describe these parts in large detail. A detailed description of the components of the architecture can be found in [1].

3.1 Alignment and exponent processing

The alignment is performed as follows:

- As done for the single datapath MAF architectures [7, 8], the alignment in the CLOSE datapath is implemented as a right shift of A_{inv}; in this case, and since the maximum alignment shift is 2, A_{inv} is placed two bits to the left of the most significant bits of $B \times C$ or of C in case of floating–point addition (Figure 3(a)). Then, the resulting alignment shift is $sh_{close} = 2 - d$, with $0 \leq sh_{close} \leq 3$; that is, a 3-bits right shifter is required. Note that in floating–point addition, as $OVF(B \times C) = 0$, $1 \leq sh_{close} \leq 3$. Since the maximum shift of A_{inv} is 3 bits, only the 54 most–significant bits of the $B \times C$ input the 3:2 CSA; the 52 least–significant bits input directly into the HAs.

- To use this approach in the FAR path requires a full normalization shift, since the number of leading zeros after the alignment is unknown. Then A or $B \times C$ (or C in case of floating–point addition) is aligned depending on the sign of the exponent difference and the shift amount is given by d. An example of the alignment $B \times C$ (or C) is shown in Figure 3(b). During the alignment, the bits of $B \times C$, C or A shifted out can propagate 1 or 2 carries[7] to the upper part. These carries are calculated during the alignment/normalization and are added at the output of the 3:2 CSA and in the add/round module [1].

  Only two shifters are needed for the alignment. This means that one shifter is shared for the alignment of A and the sum word of $B \times C$. In case of a floating–point addition, and to avoid the use of $sign(d)$ at the beginning, these two shifters are used, in such a way that A and C are right shifted and then, depending on $sign(d)$, a shifted operand and a non-shifted operand are selected.

  The maximum alignment shift is 53 bits for $B \times C$ or 106 bits for A (only for maf operations), since shifts larger than these maxima place the shifted operand to the right of the least–significant bit of the non–shifted operand, affecting only the calculation of the sticky bit.

In both paths, the three resulting vectors after the alignment are added in a 3:2 CSA. To avoid a negative result in the CLOSE datapath, which would complicate the add/round stage, the sign of the 3:2 CSA output is detected and, if negative, the outputs are inverted. In both paths, an additional row of HAs is required to correctly perform the rounding [4, 8].

Exponent processing

The exponents of the three operands are processed to obtain the control signals and the shift amounts for the alignment and normalization stage. However, in order to reduce the latency of the floating–point addition, the exponent processing logic is split into two parts in such a way that, in case of a MAF operation, part of the processing is done in parallel with the multiplier. Special attention deserves the calculation of the sign of the exponent difference, $sign(d)$. This sign is needed before the alignment shifters of the FAR datapath in case of a multiplication or a multiply–add, and after the shifters in case of an addition. Therefore, it is computed in parallel with the multiplier for floating–point MAF, and in parallel with the alignment shifters in case of floating–point addition. This requires the replication of the logic for obtaining $sign(d)$.

Moreover, the calculation of the shift amounts for the alignment shifters in the CLOSE and FAR datapath, $2 - d$ and d respectively, are overlapped with the operation of the shifters; in this way, no additional delay is introduced.

Complementing in effective subtraction

A two's complement representation is used in both the CLOSE and FAR datapaths, to avoid the end around carry adjustment needed in a one's complement adder and to simplify the design of the combined addition and rounding. However, there are some differences on how the bit–inversion and addition of the least–significant one are carried out in both paths:

- In the CLOSE path, to avoid a slow comparison when the exponent difference is 0 or 1, A is always inverted, and the 1 required to complete the two'complement is added using an empty slot in the 3:2 CSA.

[7]Two carries can propagate to the upper part in case of effective multiply–subtraction and $d \geq 0$, due to the 2'complement of the aligned $B \times C$

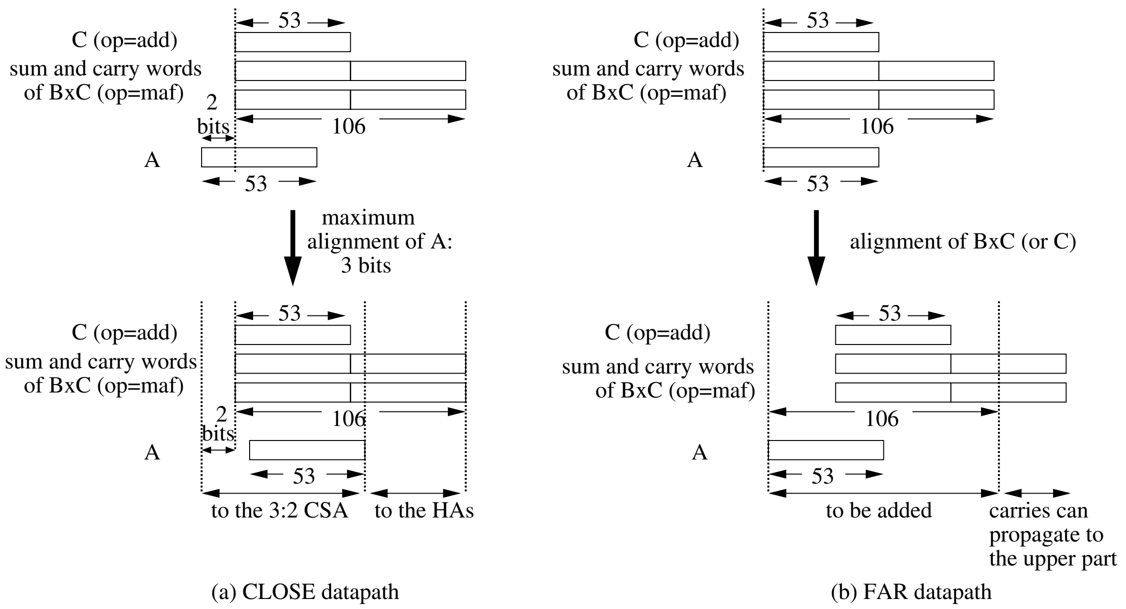

Figure 3. Alignment in (a) the CLOSE datapath and (b) the FAR datapath

Then, the result of an effective subtraction is negative when $sign(d) = 0$; but if $d = 0$ or $d = 1$ the result can be also positive and the magnitudes have to be compared to determine the sign (module *sign detection*). To avoid additional delays, the calculation of the 2's complement of the output of the 3:2 CSA is performed after the row of HAs and the bit-inversion of the carry and sum words is carried out conditionally. The HAs are duplicated, with and without inverted inputs, and the correct output is selected according to the *complement* signal. To complete the 2's complement, the two 1's required are introduced as the least–significant bit of the carry word at the output of the HAs and as an additional 1 in the add/round module.

- In the FAR path A or $B \times C$ is inverted depending on the exponent difference, and the result is always positive. Then, one or two 1s are required to complete 2's complement of A or $B \times C$, respectively. These additional 1s are added using an empty slot in the 3:2 CSA and in the add/round module.

3.2 Normalization

The normalization is performed before the addition [8]. This requires two full–length shifters in the CLOSE path (one for the sum word and another for the carry word) and two 3–bit shifters in the FAR path. The 3–bit shifter in the FAR datapath is necessary since, in an effective multipliy–addition and due to a possible overflow in the multiplication, the result could have two overflow bits, requiring a 2–bit right shift; whereas in an effective multiply–subtraction the result can be unnormalized, requiring a 1-bit left shift.

To reduce the delay in the CLOSE path, the operation of the LZA is overlapped with the normalization shift. To achieve this overlap, the shift count is obtained starting from the most–significant bit and the normalization begins once the first bit (MSB) is obtained. On the other hand, since the part of the LZA that cannot be overlapped with the shifter has a significant delay, the HA and part of the adder are placed before the normalization [8].

4 DELAY COMPARISON AND LATENCY REDUCTION

We evaluate the proposed MAF architecture by comparing it with the single–datapath architecture with normalization before addition, which has been estimated to be faster than the basic MAF [8]. We show that the latency of the floating–point addition is reduced with a small effect on the cycle time.

On the other hand, the architecture in [15] shares some of the characteristics of the MAF proposed in this paper: multiple–datapaths depending on the alignment shift and combined addition and rounding. However, whereas in our architecture we anticipate the normalization before the addition to overcome the fact that in the MAF operation the rounding position is not known before the normalization, the architecture in [15] uses an integrated unit that (1) determines the position of the rounding and (2) performs the addition and the variable position rounding; however, this unit is not described and no reference is provided. Therefore, we cannot compare with this MAF while the structure of this unit is unknown. In any case, the objective of our architecture is to reduce the latency of the floating–point addition with respect to MAF operation, and in the MAF in [15], the latency of the addition is not reduced, since it is the same as the latency of the MAF operation.

Then, to compare with the MAF with normalization before addition, we consider three different steps,

- *Single–datapath MAF*: (1) alignment and multiplication, (2) normalization, and (3) combined addition and rounding.
- *Proposed MAF*: (1) multiplication, (2) alignment and normalization, and (3) combined addition and rounding.

The delays of steps (1) and (3) are the same for both architectures [8]; then, we concentrate on step 2.

4.1 Delay of the normalization step in the single–datapath MAF

The delay of the normalization (Figure 1) is [8], $t_{norm}^{single} = t_{paths} + t_{162sh}$ being t_{162sh} the delay of the 162–bit normalization shift, and t_{paths} the largest delay of the three parallel paths: (1) sign detection, (2) 3:2 CSA, HAs and part of the adder and (3) LZA. We have estimated that the larger delay corresponds to the 3:2 CSA, HAs and part of the adder[8]. Then,

$$t_{norm}^{single} = t_{csa} + t_{ha} + t_{part\ add} + 2\ t_{mux} + t_{162sh} \quad (1)$$

4.2 Delay of the alignment/normalization step in the proposed MAF

The estimates shown has been obtained according to the detailed description of the architecture given in [1]. In both paths, the selection between addition or MAF operations in the exponent processing introduces some additional delay [1]: one and–or gate and one xor gate ($t_{and-or} + t_{xor}$) in the CLOSE path and one multiplexer in the FAR path (t_{mux}).

- **CLOSE**. Three parallel paths:(1) sign detection, (2) bit–invert, 3–b shifter, 3:2 CSA, HA and part of the adder, and (3) bit–invert, 3–b shifter and LZA. The slowest is the second path [1, 8]. Then

$$\begin{aligned} t_{al/nr}^{close} \;&=\; (t_{and-or} + t_{xor}) + (t_{3sh} + 2\ t_{ha} + \\ &\quad t_{part\ add} + t_{mux}) + t_{108sh} \qquad (2) \end{aligned}$$

Note that part of the 3:2 CSA, one HA, is in parallel with the alignment shifter. Moreover, the bit-invert is in parallel with the xor gate in the exponent processing.

- **FAR**. The delay of this step is,

$$t_{al/nr}^{far} = 3\ t_{mux} + t_{106sh} + t_{and} + 2\ t_{ha} + t_{paths}^{far} + t_{1sh} \quad (3)$$

We have made the following assumptions: (1) $t_{106sh} = t_{55sh} + t_{mux}$, (2) $t_{bit_inv} = t_{ha}$, and (3) the cancelation of the aligned operands when the maximum alignment shift has been exceeded is t_{and}

Moreover, t_{paths}^{far} is the delay of the parallel paths: (1) HA, part of the dual adder and 2–b right/1–b left shifter, (2) 3–bit adder and shift amount coding and (3) MSC detector. The larger delay corresponds to the first path. Then,

$$\begin{aligned} t_{al/nr}^{far} \;&=\; 3\ t_{mux} + t_{106sh} + t_{and} + 3\ t_{ha} + \\ &\quad t_{part\ add} + t_{3sh} + t_{1sh} \qquad (4) \end{aligned}$$

By comparing equations (2) and (4), we obtain

$$t_{al/nr}^{double} = t_{al/nr}^{far} \quad (5)$$

4.3 Comparison

We compare now equations (1) and (5). We use as delay unit the delay of an inverter with load of 4, t_{inv4}. Although this model is used to be independent of the technology, there are several modules which have alternative implementations; consequently, we provide only an estimation of the critical path delay. We consider: (1) $t_{csa} = 2\ t_{ha}$, (2) $t_{162sh} = t_{106sh} + t_{mux}$ and (3) $t_{and} = 2\ t_{inv4}$[9], $t_{mux} = 4\ t_{inv4}$ and $t_{3sh} = 8\ t_{inv4}$ (two levels of muxes). Then,

$$t_{al/nr}^{double} - t_{norm}^{single} = t_{and} + t_{mux} + t_{3sh} = 14\ t_{inv4} \quad (6)$$

To put in perspective this increment, we have to take into account that the estimated delay of the basic

[8]Note that the delay of part of the adder depends on the number of levels anticipated before the shifter; that is, the part of the adder that is anticipated is such that the delay of this path matches the delay of the longest of the other two paths.

[9]The set of AND gates in the far datapath can be replaced by NAND gates if the control signals of the controlled bit inverters after the AND gates are negated.

	Basic	norm.bef.add.	Proposed	
	maf,mul,add	*maf,mul,add*	*maf,mul*	*add*
Delay (t_{inv4})	1	0.8	0.9	0.6

Delays normalized to the delay of the basic architecture

Table 1. Delays of the MAF architectures

single–datapath MAF and the single–datapath MAF with normalization before addition are 175 t_{inv4} and 145 t_{inv4}, respectively [8].

The delay reduction with respect to the basic MAF are shown in Table 1. Consequently, the delay increment for the computation of the MAF operation is small, around 10%, with respect to the single–datapath MAF with normalization before the addition; but, even with this increment, the global delay is smaller than the delay of the basic MAF [7]. Moreover, the delay of the floating–point addition has been reduced significantly both with respect to the basic MAF, around 40%, and with respect to the single–datapath MAF with normalization before the addition, around 30%.

4.4 Latency reduction in a pipelined unit

We consider three steps in the MAF architecture: (1) Multiplication $B \times C$ and part of the exponent processing, (2) Alignment and normalization shifts and part of the dual addition, and (3) Rest of the dual adder, in parallel with carry and sticky bit calculation.

This architecture can be pipelined in a way similar to previous implementations of the single–datapath MAF, into three or five stages, by assigning each step to a stage (three stages) or splitting steps 1 and 3 into two pipeline stages (five stages), or into four stages, because of the delay reduction with respect to the basic MAF while maintaining the same cycle time.

In this way, if the architecture is pipelined into three stages, the number of cycles required is 3 cycles for fp–mult and fp–MAF (stages 1 to 3), and 2 cycles for fp–add (stages 2 and 3). On the other hand, if it is pipelined into five stages, the fp–mult and MAF require 5 cycles, and the fp–add only 3 cycles. In case of a pipelining into four stages, the latency of the fp–add could be reduced to only 2 cycles.

5 CONCLUSIONS

An architecture for a floating-point Multiply–Add–Fused (MAF) unit that reduces the latency of the floating–point addition with respect to previous MAF implementations is proposed. This architecture is based on a previous MAF architecture, with combination of the final addition and the rounding, by using a dual adder, and anticipation of the normalization before the addition. The proposed MAF architecture incorporates a double–datapath organization, previously used in floating–point adders and MAF units, to take advantage of the fact that the full–length alignment shift and the full-length normalization shift are mutually exclusive, and only one of such shifts appears on the critical path.

The proposed organization allows to carry out the alignment of the addend after the multiplication, instead of in parallel with it as done in previous architectures. This permits to bypass the first stages when a floating–point addition is executed, resulting in a lower latency for this operation. This reduction from three to two cycles, when a pipelining into three stages is implemented, or from five to three stages if the unit is pipelined into five stages. Consequently, the floating–point addition has been integrated into the floating–point MAF unit with major benefits for the addition at the expense of a minor delay increment for the MAF operation.

Additionally, the delay of the proposed architecture has been estimated and compared with the single–datapath MAF architectures. The conclusion is that there is a significant delay reduction in the computation of a floating–point addition, about 40% with respect to the basic MAF, and around 30% with respect to the single–datapath MAF with normalization before the addition.

References

[1] J.D. Bruguera, T. Lang. *Floating–Point Multiply–Add Fused: Latency Reduction for Floating–Point Addition.* Internal Report. University of Santiago de Compostela (Available at www.ac.usc.es). (2004).

[2] C. Chen, L-A. Chen, J-R. Cheng. *Architectural Design of a Fast Floating–Point Multiplication–Add Fused Unit Using Signed–Digit Addition.* Proc. Symp. on Digital System Design (DSD'2001). pp. 346–353. (2001).

[3] M. DiBrino, F. Karim. *Floating-point pipeline with leading zeros anticipator circuit.* US-Patent 6542915. April 2003.

[4] G. Even, P.M. Seidel. *A Comparison of Three Rounding Algorithms for IEEE Floating–Point Multiplication.* IEEE. Trans. Comput. Vol. 49. No. 7. pp. 638–650. (2000).

[5] G. Gerwig, H. Wetter, E.M. Schwarz, J. Haess, C.A. Krygowski, B.M. Fleischer, M. Kroener. *The IBM eServer z990 Floating–Point Unit.* IBM J. Research and Development. Vol. 48, No. 3/4. pp. 311–322. (2004).

[6] C. Heikes, G. Colon–Bonet. *A Dual Floating Point Coprocessor with an FMAC Architecture.* Proc. ISSCC96. pp. 354–355. (1996).

[7] R.M. Jessani, M. Putrino. *Comparison of Single– and Dual–Pass Multiply–Add Fused Floating–Point Units.* IEEE Trans. Computers. Vol. 47, No. 9. pp. 927–937. (1998).

[8] T. Lang, J.D. Bruguera. *Floating–Point Fused Multiply–Add with Reduced Latency.* IEEE Trans. Computers. Vol. 53, No. 8, pp. 988–1003.(2004)

[9] R. K. Montoye, E. Hokenek, S. L. Runyon. *Design of the IBM RISC System/6000 floating-point execution unit.* IBM Journal of Research and Development. Vol. 34, No. 1. pp. 59–70. (1990)

[10] A. Naini, A. Dhablania, W. James, D. DasSarma. *1–GHz HAL SPARC64 Dual Floating Point Unit with RAS Features.* Proc. IEEE 15th Symp. Computer Arithmetic (ARITH15). pp. 173–183. (2001).

[11] S.F. Oberman, H. Al–Twaijry, M.J. Flynn. *The SNAP Project: Design of Floating–Point Arithmetic Units.* Proc. IEEE 13th Symp. Computer Arithmetic (ARITH13). pp. 156–165. (1997).

[12] F.P. O'Connell, S.W. White. *POWER3: The Next Generation of PowerPC Processors.* IBM J. Research and Development. Vol. 44, No. 6. pp. 873–884. (2000).

[13] R.V. Pillai, D. Al-Khalili, A.J. Al-Khalili. *Low Power Architecture for Floating–Point MAC Fusion.* IEE Proc. Comput. Digit. Tech. Vol. 147. pp. 288-296. (2000).

[14] H. Sharangpani, K. Arora. *Itanium Processor Microarchitecture.* IEEE Micro Mag. Vol. 20, No. 5. pp. 24–43. (2000).

[15] P.M. Seidel. *Multiple Path IEEE Floating–Point Fused Multiply–Add.* Proc. 46th Int. IEEE Mid-West Symposium on Circuits and Systems (MWSCAS). (2003)

[16] P.M. Seidel, G. Even. *On the Design of Fast IEEE Floating–Point Adders.* Proc. IEEE 15th Symp. Computer Arithmetic (ARITH15). pp. 184–194. (2001).