Floating-Point Single-Precision Fused Multiplier-adder Unit on FPGA

Wilson José*, Ana Rita Silva*, Horácio Neto†, Mário Véstias‡
*INESC-ID, †INESC-ID/IST/UTL, ‡INESC-ID/ISEL/IPL
wilsonmaltez@inesc-id.pt, anaritasilva@inesc-id.pt, hcn@inesc-id.pt, mvestias@deetc.isel.pt

Abstract

The fused multiply-add operation improves many calculations and therefore is already available in some general-purpose processors, like the Itanium. The optimization of units dedicated to execute the multiply-add operation is therefore crucial to achieve optimal performance when running the overlying applications. In this paper, we present a single-precision floating-point fused multiply-add optimized unit implemented in FPGA and prepared to integrate a data flow processor for high-performance computing. The unit presents a numerical accuracy according to the IEEE 754-2008 standard and a performance and resource usage comparable with a state-of-the-art non-fused single-precision unit. The fused multiplier-adder was implemented targeting a Virtex-7 speed-grade -1 device and occupies 754 LUTs, 4 DSPs and achieves a maximum frequency of 361 MHz with 18 pipeline stages. A lighter low latency design of the same unit was also implemented in the same device presenting a resource usage of 845 LUTs, 2 DSPs and achieving a maximum frequency of 285 MHz.

1. Introduction

The floating-point multiplication-add operation is very important in several digital signal processing and control engineering applications, specifically applications which rely on the dot product. For that reason, the optimization of the hardware which performs the floating-point multiplication-add operation can result in significant gains in the execution of these applications. IBM recognized the importance of the multiplication-add operation and in 1990 unveiled the implementation of a floating-point fused multiply-add arithmetic execution unit on the RISC System 6000 (IBM RS/6000) chip [1], [2]. This floating-point arithmetic unit executes the equation $(A \times B) + C$ in a single instruction.

The advantages of a fused multiplier-adder (FMA) is that it not only improves the performance of an application that recursively executes multiplication followed by addition, but as well can execute a floating-point multiplication or addition by inserting fixed constants into its data path. However, this emulation of a floating-point multiplier or floating-point adder do not come without a cost, as the block's additional hardware imposes extra latency on the stand-alone instructions as compared to their original units [3]. Also, commonly the bit-widths and interconnectivities of internal blocks of the FMA more than double compared to those of floating-point adders and floating-point multipliers which complicates the process of routing the design and achieving the timing goals. Finally, the fused multiply-add unit is characterized by a heavy power consumption. Nevertheless, with the continued demand for 3D graphics, multimedia applications, and new advanced processing algorithms, the performance benefits of the fused multiply-add unit out-weights its drawbacks.

The work presented in this paper focus on another advantage of the fused multiply-add floating-point units which is increasing the numerical accuracy. By joining the multiply and add operation we can maintain the maximum precision through the intermediate results and only perform rounding after the add operation. In this context, we present the accuracy gain of our single-precision fused floating-point multiply-add design and respective area/speed trade-offs in relation to a non-fused unit. The unit presents a numerical accuracy according to the IEEE standard [4] and a performance and resource usage comparable with a state-of-the-art non-fused single-precision unit based on the Xilinx Core Generator floating-point multiplier and adder [5]. Also, the FMA unit is intended to be integrated with our reconfigurable processor which is part of multiprocessor system dedicated to accelerate a set of High-Performance Computing (HPC) applications [6]. This particular aspect was an important one to take into account when we designed the FMA unit, more on that later.

Next, we present the paper structure. In section 2, we discuss the most relevant works exploring fused floating-point multiply-add architectures with emphasis on the FPGA approaches. Section 3 describes the floating-point IEEE standard for single-precision numbers representation. In section 4 we present our FMA architecture and in section 5 we discuss the numerical accuracy results. Also, we compare our fused unit with non-fused single-precision multiply-add unit based on the Xilinx floating-point units in terms of the accuracy provided against the resources usage and maximum frequency achieved. Finally, we present our conclusions and discuss the future work in section 6.

2. Related Work

The multiply-add fused unit, was first proposed in 1990 [2]. After that, there were several works which tried to improve FMA architectures, but often target stand-alone Application-Specific Integrated Circuits (ASICs) or units integrated into general-purpose processor pipelines [7], [8],

[9]. In [3], the author gives a survey of the wide spectrum of FMA architectures developed from 1990 to 2007. The principle of fused operators has also been applied to other computations, such as fused dot products [10] and FFT [11].

The application-specific use of non-standard formats for improved numerical accuracy has been proposed for FPGAs in [12]. The use of non-standard formats to improve performance is presented in [13] for the use of a multiply-accumulate (MAC) unit. The design uses a PCS (Partial Carry Save) representation to achieve low latency at the addition stage but relies on application-specific knowledge of the input and output value ranges. Also, the authors only provide implementation results for the accumulator. Implementations of Radix 4 and 16 exponents showed improved addition speed but slower multiplication [14]. Also, it is stated that contrary to established belief, higher radix representations are useful for FPGA applications requiring IEEE 754 compliance, since they can deliver superior numerical performance while still using less FPGA resources.

The paper in [15] discusses the fundamentals of floating-point operations on FPGAs. The authors present a 270 MHz IEEE compliant double precision floating-point performance with a 9-stage adder pipeline and 14-stage multiplier pipeline mapped to a Xilinx Virtex4 FPGA (-11 speed grade). Their area requirement is approximately 500 slices for the adder and under 750 slices for the multiplier.

The paper in [16] examines existing solutions and proposes two new architectures for floating-point fused multiply-adds having heterogeneous input formats and considering the impact of different in-fabric features of recent FPGA architectures. The units are evaluated at the application level by modifying an existing high-level synthesis system to automatically insert the new units for computations on the critical path of three different convex solvers. The unit improves performance by up to 2.5× over the closest state-of-the-art competitor and also achieves a better numerical accuracy. Although, this results seem to come at the expense of a brutal increase in the resources needed (between 4x to 5x more than the Xilinx cores).

3. IEEE Floating-Point Representation

The FMA unit design follows the ANSI/IEEE-754 standard for binary floating-point numbers representation. The single-precision format is 32-bits wide and its structure is represented in figure 1.

Together, the three components form the number $x = (-1)^{sign} \times s \times 2^{(e-bias)}$. The use of a biased exponent format has virtually no effect on the speed or cost of exponent arithmetic (addition/subtraction), given the small number of bits involved. It does, however, facilitate zero detection (zero can be represented with the smallest biased exponent of 0 and an all zero significand) and magnitude comparison (we can compare normalized floating-point numbers as if they were integers) [17].

The notation "$23+1$" for the width of the significand is meant to explain the role of the hidden bit, which contributes to the precision of the number without requiring an extra bit in the representation. Denormals, or denormalized values, are defined as numbers without a hidden 1 and with the smallest possible exponent. They are provided to make the effect of underflow less abrupt. However, the standard do not require the support for denormals.

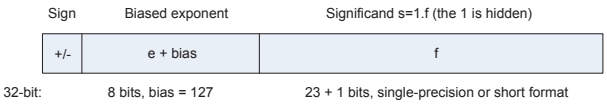

Figure 1. The ANSI/IEEE floating-point number format for single-precision.

In 2008, the IEEE defined a standard for fused floating-point operations [4]. Basically, it states that the fused operation multiply-add, should compute $(A \times B) + C$ as if with unbounded range and precision, rounding only once to the destination format. The main objective of this work is to achieve the maximum accuracy on a fused multiply-add operation, following the stated IEEE guidelines.

4. Design Considerations

The FMA unit was designed with the intent to be part of the data path of the reconfigurable processor which integrates our multiprocessor architecture in [6]. This multiprocessor has a linear array based architecture in that each processor forwards data to the next processor further on the processor array. Each individual processor has a very simple instruction set and pipeline structure without the data hazards associated with general purpose processors. In the context of a matrix multiplication application and other similar applications, since none of the processors are dependent on each other results and produce independent chunks of data, as long there are enough bandwidth they can produce one result per cycle. This basically means that increasing the processor data path number of pipeline stages has a negligible impact on the application total execution time. Therefore, during the design we make a bigger effort in increasing the FMA maximum frequency and are not so concerned with the latency of the unit.

Other consideration is that the FPGAs slices have one register per each LUT, which means, pipelining in FPGA designs can come at little or no cost contrary to ASIC designs.

Since we have less restrictions in the number of pipeline stages, we focus less in parallelizing the hardware (which can be troublesome as this kind of optimization can end up in describing hardware at a very low level as is the case with implementing a leading zero counter predictor for example). Contrary to other approaches, [3], we do not try to parallelize the alignment of the adder operands with the mantissas multiplication as this can also turn more difficult using just the multiplier or the adder in a design.

In our design, like in the Xilinx floating-point units [5], we do not support denormals, as these only marginally extend the representable number range and lead to cost and speed overhead in hardware [17].

To achieve the increased precision we have proposed to,

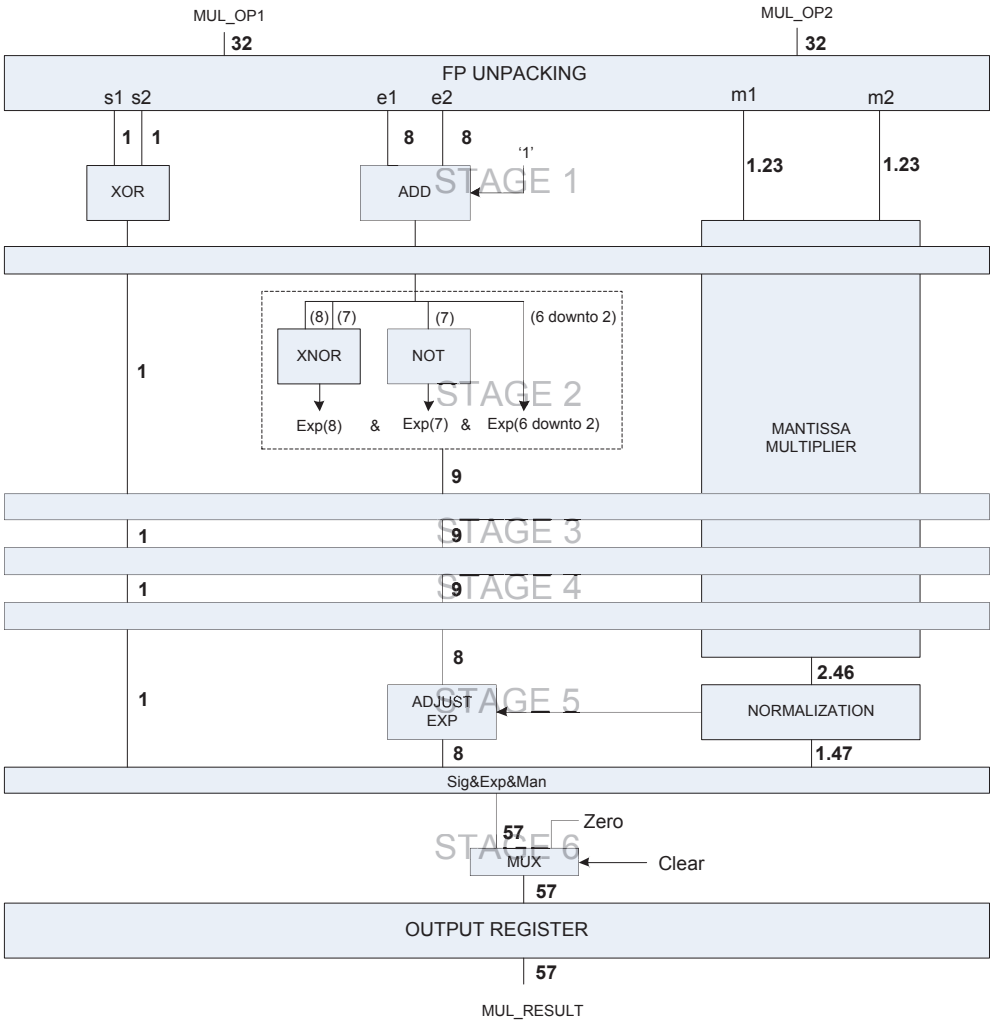

Figure 2. Floating-Point Multiplier Architecture.

we maintain the whole 48 bits from the multiplier result to the adder, contrary to a non-fused implementation. In the adder we maintain full precision in all operations till the rounding stage.

In the next section we will begin by describing the multiplier part of our design followed by the adder architecture.

5. FMA Architecture

The floating-point multiplier is divided in six stages as figure 2 shows.

When two valid 32-bits operands enter the first stage they are codified in the IEEE floating-point single-precision format, therefore these operands are unpacked into their respective components, signal, exponent and mantissa (the hidden one included). The resulting signal is immediately determined by the XOR of the two operand signals. The two exponents are also added in this stage (result exponent has 9 bits because the carry out is concatenated with the adder output). The bias subtraction is equivalent to adding -128+1 which can be done by having the carry in set to one and flipping the most significant bit of the exponent result. This last part is not that simple as we have to consider the carry out signal from the exponent adder to detect overflow/underflow situations. The correct two most significant bits of the result exponent are determined in the second pipeline stage of the multiplier as represented in the figure.

The mantissa multiplier spreads across 4 pipeline stages to achieve the maximum frequency possible. In the fifth pipeline stage the mantissa result may have to suffer a shift right to be normalized if the result is equal or bigger than 2. In this stage the exponent can be also adjusted depending if the mantissa needed to be normalized or not. The sign, exponent and mantissa are then concatenated and travel to the next stage. The last pipeline stage addresses the situation in which the result is equal to zero. When any of the operands is zero the clear signal is set to 1 and chooses a result in which both the sign, exponent and mantissa are zero. Finally, the multiplication result travels to the adder.

The floating-point adder architecture 12 pipeline stages is depicted in figure 3. The 12 pipeline architecture was achieved after an effort to maximize the maximum frequency and then reducing the area of the design. As it was referenced in beginning of the paper, we also designed a low latency version of the FMA architecture. A lighter unit may enable us to have more processors in our multiprocessor system which can be more advantageous than having less processors working at a higher frequency. This version is not represented in a figure as the modifications were not enough to justify it. A small description of those modifications is given in the end of this section.

The first step of the floating-point addition consists in the alignment of the operands. The operands need to be aligned to enable the mantissas addition. In our case the alignment of the operands is preceded by a swap step, necessary to avoid having two shifters. In the first pipeline stage we determine the operand to be right shifted, that is, the operand with smallest exponent. We subtract both exponents to calculate the shift amount and the sign of this operation lets us know which operand has the smallest exponent. Also, the mantissa from the second operand is concatenated with zeros to the right in order to have the same width of the mantissa from the multiplier result before entering the multiplexer. Finally, in this stage we verify if the two operands are equal and have different signs, being that the case we set the result exponent to zero.

The second and third stage of the floating-point adder correspond to the mantissas alignment. The shifter implemented accounts for the maximum shift necessary to achieve high precision. Figure 4 allows to develop a better understanding of the maximum shift we have to support to achieve the precision we desire. Basically, there are two different situations we have to consider.

When the second operand from the addition has a smaller exponent we must allow a maximum shift of 47 bits because of the ”rollback effect”, that is, we have to take into consideration all the bits which can ripple during the mantissas addition and change the final rounded result. The 48 most significant bits from the shifter output form the aligned mantissa of the second operand. The 23 less significant bits are used to calculate the sticky bit in the next stage. In fact, for shifts bigger than 47, we have to always take into consideration all bits shifted past the 48th bit because the discarded bits affect the sticky bit, which may be needed to determine if the result is rounded when the round bit (mantissa result's 25th bit) is one. We do this by fixing the maximum shift in 48 and always calculating the sticky bit with the 24 less significant bits of the shifter output.

On the other hand, when the multiplier result has a smaller exponent, we must allow a maximum shift of 24 bits like is represented in the right part of the figure 4. Shifting past this value is not worth doing because in that case the round bit is always zero and therefore the final result is simply truncated and not rounded. As we only use one shifter, we need a shifter capable of shifting a maximum value of 48 bits and thus the multiplier result can also be shifted up to 48 bits. This means the shifter must have a 96 bit output. The 48 most significant bits correspond to the mantissa and the 48 less significant bits to the discarded bits used to calculate the sticky bit. The shifter module is a base 4 logarithmic shifter composed by three 4:1 multiplexer stages which can shift the respective mantissa up to 63 bits. Also, the shifter is structured in pipeline, having a set of registers between the second and the third multiplexers stages.

In the fourth and fifth stage we add the mantissas. The mantissa adder is mapped on a DSP which spreads across two pipeline stages being that we only use one stage of registers in the DSP and the result is stored in a register outside the DSP. In parallel with the mantissa adding in the fourth stage, we determine the sticky bit.

The normalization step begins in the sixth stage. The block *adjust mantissa* makes the two's complement of the mantissa result in case the result from the mantissa adder is negative or right shift the mantissa by one in case the two operands have the same signal and the result has overflown. The dropped bit is accounted for the sticky bit.

When two operands are subtracted the result mantissa may be less than one and therefore need a left shift to be normalized. The next four stages involve the left shift normalization process. Across the seventh and eighth stages we have the leading zero detector block is responsible for determining the amount the mantissa must be left shifted. The left shifter is a base 4 logarithmic shifter composed by three 4:1 multiplexer stages which can shift the respective mantissa by 63 bits (since the input is 48 bits, the shift will never be bigger than 48 bits). The shifter is spread across the stage 9 and 10, having a set of registers between the second and the third multiplexers stages

The exponents have also to be updated during the normalization process. In stage 9 the exponent is updated accordingly to the normalization needed (increments 1 in the case of a right shift or is subtracted by the amount shifted in the case the mantissa was left shifted).

The tenth stage is also the beginning of the rounding process. The rounding process is defined by the round to nearest even method specified by IEEE. To determine if the result has to be rounded we consider the least significant bit of the mantissa (24th bit), the round bit (25th bit) and the sticky bit. The final sticky bit is the result of the OR of the previous sticky bit with the OR of all bits past the round bit from the normalized mantissa. The result is rounded if the round bit is 1 and the least significant bit or the sticky bit are 1.

We add the round bit in a DSP which spreads across the eleventh and twelfth stage. The first input of the adder is the exponent as the 8 significant bits and the mantissa (already without the hidden 1) 23 least significant bits. The second input is zero and the carry in is the round bit. In the case an overflow happens an exception is set to high. The definitive value of the exponent correspond to the 8 most significant bits of the adder result and the definitive value of the mantissa to the 23 least significant bits of the adder result. Note that the addition of the round bit with the mantissa can lead to an overflow and consequentially is needed the exponent and the mantissa must be normalized

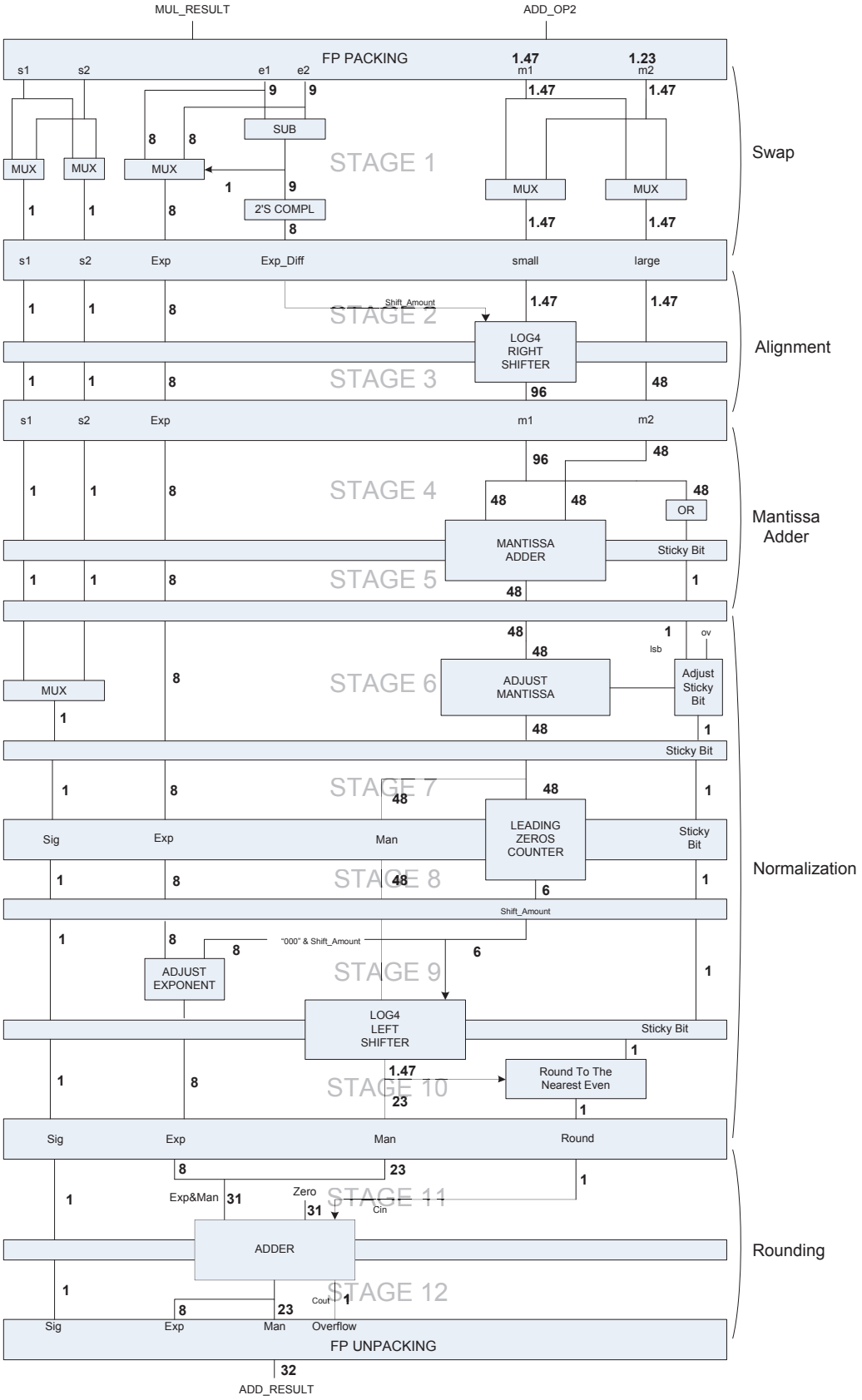

Figure 3. Floating-Point Adder Architecture.

as explained next. If an overflow occurs after adding the round bit with the mantissa, the exponent is automatically updated because it is the most significant part of the adder result. On the other hand the mantissa does not need to be

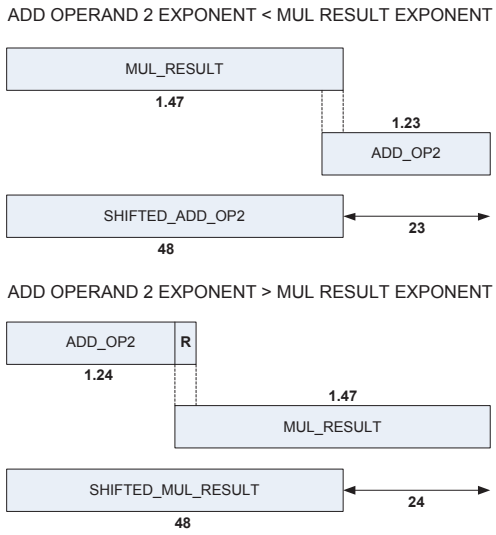

Figure 4. The alignment boundaries considering the two floating-point adder operands.

shifted. Since the mantissa entering the adder does not have the hidden bit, it means that an overflow only occurs when the mantissa is all ones, and this results in the mantissa being all zeros after the round bit addition.

Hereinafter we describe the modifications done in order to have low latency version of our floating-point adder. We designed this version with the objective of having a lighter unit which can be useful in multiprocessor systems where the FMA unit is not the performance bottleneck. Moreover, we implement the low latency unit only with LUTs which enable the possibility of the adder being mapped in previous FPGAs generations where the DSPs could not implement a 48-bit adder. The first simplification was to fuse the second and third pipeline stages, that is, the alignment is done in one clock cycle. The fifth and sixth stages were fused too and the leading zeros counter hardware was moved from the previous seventh stage to the now fifth stage. Finally the rounding adder is part of only one pipeline stage in the low latency version.

6. Results

The unit was tested with valid random generated data which follows a normal distribution. Valid data means that we apply some boundaries in the test data to better appreciate the accuracy results. Situations as overflows/underflows which rise exceptions are avoided. Also, we limit the multiplication result and the second addition operand exponents to avoid having a big difference between the two exponents because we ended adding zero to another operand except for the case in which the multiplication result exponent is bigger than the exponent of the other operand and the sticky bit can contribute to the decision of rounding the result.

We generated 10000 samples for each one of the three operands involved in the multiply-add operation which are fed to our hardware unit during the simulation. A high level C implementation was created to compute the result of each multiply-add operation emulating a non-fused single-precision unit and a double-precision unit. The results from the double-precision unit serve as a standard to which our unit and the non-fused single-precision unit compare to. Although, before comparing, the results from the double-precision unit are rounded following the IEEE round to nearest even rules. The process of calculating the 10000 samples was repeated 20 times to confirm the consistence of the results.

The results given by our FMA unit were compared to the non-fused single-precision and double-precision results. We verified that our hardware always achieves the maximum precision possible, whereas that does not always happen for the non-fused single-precision. In fact, over the 20 sets of tests made, our hardware presents an average of 1127 results in 10000 with more accuracy than the non-fused single-precision multiply-add unit. The single-precision error is in average 0.0000001192092896, approximately 2^{-23} in base 2.

Next, we discuss the implementation results of our FMA unit against a non-fused multiply-add unit composed by a state-of-the-art floating-point multiplier and floating-point adder from Xilinx[5]. Presented in table 1 are the post place and route results obtained with the Xilinx ISE tool version 14.5 targeting the device Virtex-7 (XC7V585T-1).

Table 1. Implementation results for our FMA unit and a non-fused multiply-add unit based on the Xilinx floating-foint units

XC7V585T-1				
	Ours (Low Lat.)	Xilinx (Low Lat.)	Ours (High Lat.)	Xilinx (High Lat.)
FFs	773	820	990	683
LUTs	845	1018	754	751
DSPs	2	2	4	4
Freq(MHz)	285	350	361	332
Latency	14	14	18	18

In table 1 we present results for two different versions of our FMA unit and the Xilinx based non-fused equivalents. The low latency versions uses only LUTs to implement the floating-point adder as this can possibly lead to a lower routing delay overhead and consequently to achieving higher frequencies. When comparing the two low latency units we can see that our version uses less resources than the non-fused equivalent unit. This advantage, is however, balanced by the fact that our unit has a lower maximum frequency. The frequency could be improved by putting more effort in parallelizing the hardware in some stages in order to achieve better frequencies with the same latency, even if it means spending more resources.

In relation to the high latency units, we verify that our FMA resource usage is very close to the non-fused unit resource usage, with the exception of the number of registers used. This is difficult to avoid because with more pipeline stages and having bigger buses (due the increase width of

the signals necessary to maintain more precision in the calculations), we end up spending more registers. Nevertheless, our high latency unit has the advantage of not only enabling higher accuracy in the calculations, but also can work at a higher frequency.

7. Conclusions and Future Work

We have proposed a hardware design for an FMA unit in complete compliance with the IEEE guidelines for the fused multiply-add operation (section 2.1.28 in [4]).

Comparing the resource and performance results of our unit with a state-of-the art non-fused multiply-add design, we verified that we can achieve significant better numerical accuracy results with only mild degradation of the performance or area usage. Also, the low latency design can be useful when implementing multiprocessor systems in target devices where the DSPs are the critical resource. More so, if the system is synchronous and the FMA is not the limiting factor in the system maximum frequency achieved.

In the future, we pretend to integrate our FMA unit in an application driven processor and test the impact that the increased accuracy of a fused architecture can have in continuous multiply-add operations like the dot product (in which the error can accumulate).

Finally, we pretend to extent this study to double-precision operations and see how well the hardware can scale.

Acknowledgment

This work was supported by national funds through FCT, Fundação para a Ciência e Tecnologia, under projects PEst-OE/EEI/LA0021/2013 and PTDC/EEA-ELC/122098/2010.

References

[1] E. Hokenek R.K. Montoye and S.L. Runyon. Design of the ibm risc system/6000 floating-point execution unit. *IBM Journal of Research & Development*, 34:59–70, 1990.

[2] R. Montoye E. Hokenek and P. Cook. Second-generation risc floating point with multiply-add fused. *IEEE Journal of Solid-State Circuits*, 25(5):1207–1213, 1990.

[3] E. Quinnell. *Floating-Point Fused Multiply-Add Architectures*. PhD thesis, The University of Texas at Austin, May 2007.

[4] IEEEStandard754-2008. Ieee standard for floating-point arithmetic. Technical report, IEEE Computer Society, 2008.

[5] Xilinx. Xilinx logicore ip floating-point operator v6.1 product specification. Technical report, Xilinx, 2012.

[6] H. Neto W. Maltez, A. R. Silva and M. Véstias. Analysis of matrix multiplication on high density virtex-7 fpga. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, Sep 2013.

[7] E. S. Jr E. Quinnell and C. Lemonds. *Three-path fused multiply-adder circuit*, 2011.

[8] V. Erraguntla S. Jain and S. R. Vangal. A 90mw/gflop 3.4ghz reconfigurable fused/continuous multiply-accumulator for floating-point and integer operands in 65nm. In *2010 23rd Int. Conf. on VLSI Design*, pages 252–257, Jan 2010.

[9] J. Brooks and C. Olson. *Processor Pipeline which Implements Fused and Unfused Multiply-Add Instructions*, 2012.

[10] H. H. Saleh and E. E. Swartzlander. A floating-point fused dot-product unit. In *in 2008 IEEE Int. Conf. on Computer Design*, pages 427–431. IEEE, Oct 2008.

[11] E. Swartzlander and H. Saleh. Fft implementation with fused floating-point operations. *Computers, IEEE Transactions on*, 61(2):284–288, 2012.

[12] B. Pasca S. Banescu, F. de Dinechin and R. Tudoran. Multipliers for floating-point double precision and beyond on fpgas. *ACM SIGARCH Computer Architecture News*, 38(4):73–79, 2010.

[13] F. D. Dinechin and B. Pasca. An fpga-specific approach to floating-point accumulation and sum-of-products. In *ICECE Technology, 2008. FPT 2008. International Conference on*, pages 33–40, Dec 2008.

[14] B. Catanzaro and B. Nelson. Higher radix floating-point representations for fpga-based arithmetic. In *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCMÓ5)*, pages 161–170, Apr 2005.

[15] K. S. Hemmert and K. D. Underwood. Fast, efficient floating-point adders and multipliers for fpgas. *ACM Transactions on Reconfigurable Technology and Systems*, 3(3):1–30, 2010.

[16] J. Huthmann B. Liebig and A. Koch. Architecture exploration of high-performance floating-point fused multiply-add units and their automatic use in high-level synthesis. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 134–143, May 2013.

[17] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 2000.

[18] J. Detrey and F. Dinechin. A tool for unbiased comparison between logarithmic and floating-point arithmetic. *The Journal of VLSI Signal Processing Systems for Signal*, 49(1):161–175, 2007.