Single-Precision Floating-Point Multiply-Add-Fused for Field Programmable Gate Arrays

Ahmed Youssef, *Student Member, IEEE*

***Abstract*—This report outlines the architectural design and the VHDL and C implementation of a single-precision floating-point multiply-add-fused unit. We start by discussing the motivation behind FLP MAF units and their advantages over separate floating-point adders and multipliers. We then discuss the architecture of the implemented MAF Unit and describe every stage in the design. We then detail the C and VHDL implementation specifics of each component in our design. Finally we describe the test cases that were conducted to verify and validate correct functionality of our software and hardware implementation.**

***Index Terms*—Computer Arithmetic, Floating-Point Functional Units, Multiply-Add-Fused, VLSI Design.**

I. Introduction

COMPUTATIONS involving multiplication followed by addition represent one of the most common arithmetic operations in numerous applications. These applications include graphic processing, digital Signal processing, image and video processing, and scientific computing [1] as well as other various other applications where matrix/vecror multiplications or convolutions are required. The multiply-add operation, is commonly used for other applications that also require matrix/vector multiplications or convolutions.

The popularity of multiply-add operations, $A + B \times C$, in various applications has led to the adoption of multiply-add-fused (MAF) units in the instruction set of numerous Fixed-Point and Floating-Point (FLP) processors [2]–[4] and in virtually all recent GPU, DSP and general-purpose processors [5]. Having a FLP MAF unit entails three main advantages over separate floating-point adder and multiplier units [6]: 1) Reduction in the overall delay 2) Reduction in errors due to rounding 3) Reduction in hardware costs for the case where a MAF unit is substituted for a FLP multiplier and adder.

The first advantage occurs due to exploitation of the inherent parallelism between the hardware blocks that are required for addition and multiplication allowing an overlap in their operation. Another source of delay reduction comes from performing some of the common blocks once, such as normalization and rounding, that would otherwise have been performed twice if the operation was to be executed in two different instruction sets.

The second advantage is attributed to the fact that with MAFs, rounding is only done once at the end of execution allowing a higher precision to be adopted in the multiply-add operation. The third advantage is attributed to the fact that MAFs can be used to perform either operation (multiplication or addition) individually. This can be done be setting a multiplicand to one ($B = 1$ or $C = 1$) when addition is to be performed or setting A to zero when multiplication is to be performed. The MAF unit has a reduced hardware footprint compared to separate FLP addition and multiplication units due to the sharing of common blocks between operations. However, this may not be advantageous if performance is required since 1) The delay is now increased due to execution of the other redundant operation 2) Multiplication and Addition of separate operands cannot be performed concurrently as is the case if separate FLP addition and multiplication units are available.

The rest of this paper is organised as follows: Section II explains the architecture of our designed FLP MAF unit; Section III describes the design considerations and details of our software model; Section IV describes in detail the VHDL and C implementation of all the blocks in the architecture; Section V describes the test cases that we used to verify their correct functionality of our VHDL and software implementation.

II. Architecture of Implemented MAF Unit

In this section, we describe the overall architecture of our designed MAF unit. We designed the MAF Unit for single-precision floating-point operands based on the IEEE 754 standard. Our implementation can be broken down into three stages; 1) The multiplication and alignment stage 2) The Addition and Exponent Calculation Stage 3) The Normalization and Rounding Stage. Exception handling is not supported in our current implementation since the focus is on the architecture and the computation of the result. This was also the case with all the papers that were surveyed in the literature.

Figure 1 shows the block diagram of the single-precision FLP MAF unit that was implemented. Our design is based on the architecture of conventional FLP MAFs with minor modifications. The first modification is the utilization of conventional multipliers to perform the operand multiplication. Conventional FLP MAFs use an array multiplier followed by a Carry-Save Adder to accomplish the same task. Moreover, a Leading Zero Detector (LZD) is used and is placed in the third stage. Conventional FLP MAFs use Leading Zero Anticipator to accomplish the same task and is performed in the second stage. We also target single-precision FLP format instead of double-precision FLP format. This only affects the bit widths of the hardware blocks and internal operands. Finally, we do not perform pipelining of the three stages as is done by conventional MAFs. However, our implementation lends itself well to pipelining due to its modular design. Note that the green and blue blocks in Figure 1 represent the

A. Youssef is with the Department of Electrical and Computer Engineering, McGill University, Montreal, QC, Canada. E-mail: Ahmed.Youssef@mail.mcgill.ca.

Manuscript received December 3, 2014.

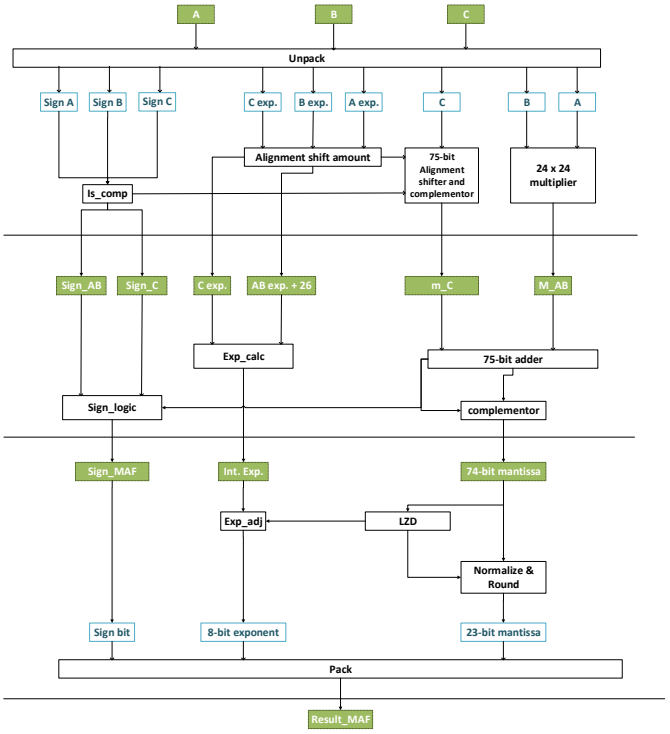

Fig. 1. The block Diagram of the designed FLP MAF Unit.

STD_LOGIC_VECTORS between the intermediate stages and not pipelined registers. We will elaborate on the details of the implementation of each block in Section IV.

A. Multiplication and Alignment Stage

In this stage, the shift amount d, is determined as follows: $d = e_C - (e_A + e_B - bias)$ where e_A, e_B, e_C are the biased exponents of operands A, B, and C, respectively. $bias = 127$ for single-precision FLP representation. At the same time, the mantissa of C, C_m, is passed to the alignment shifter to align the addend for the subsequent addition. To accomplish that, 75-bit internal STD_LOGIC_VECTORs are used. C_m is placed at the 25 Most Significant Bits(MSBs) of the internal registers. Altough the mantissa is 24 bits after appending the hidden '1', we append another '0' after that for the two's complement sign bit incase C_m is to be complemented. The 48-bit multiplication result A_mB_m is placed at the 48 Least Significant Bits (LSBs) of the multiplier output STD_LOGIC_VECTOR. This effectively simulates a 27-bit right shift of m_{AB} as shown in Figure 2. 27 is chosen as the default shift amount since the addends are 24 bits wide which means that any shift of 27 bits or more will not affect the adder output. The amount by which to shift C_m is $27-d$. We will explain how the alignment works using three cases:

- $d > 27$. This implies that m_{AB} needs to be right shifted by more than 27 bits. Since the significand of single-precision FLP is 24 bits, this implies that m_{AB} will have no effect on the addition result. In this case, C_m will not be right shifted. The two extra bits (guard and round bits) shown in Figure 2(a) ensures that m_{AB} bits do not affect the unshifted C_m bits. No leading zeros will be detected and hence no left shifts occur in the normalization stage and thus the 24 bits of C_m will be used as the mantissa result.
- $0 \leq d \leq 27$.Since m_{AB} is already right-shifted by 27 bits relative to C_m, C_m will be right shifted by $27 - d$ as shown in Figure 2(b).
- $d < 0$. This implies that $e_C < e_{AB}$. C_m will be shifted by $27 + |d|$. This can cause some of the least significant bits or all of C_m's bits to be discarded in the case that $27 + |d| > 47$. If $27 - d > 75$, shift amount is set to 75.

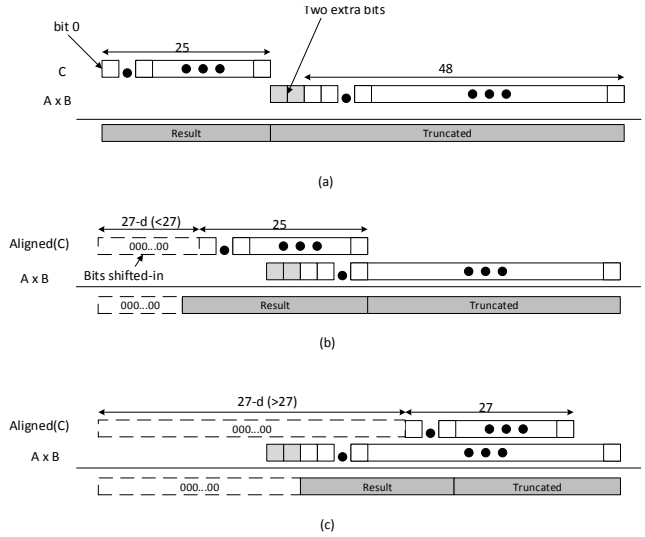

Fig. 2. Alignment of C_m relative to A_mB_m. (a) Alignment with $d > 27$. (b) Alignment with $0 \leq d \leq 27$. (c) Alignment with $d < 0$.

B. Addition and Exponent Calculation Stage

In the second stage, addition of m_C (the aligned mantissa of C) with m_{AB}(the multiplication result) is performed in parallel with the calculation of the intermediate exponent of the MAF result. A complementer is used at the output of the adder incase the adder result is negative. A signal is sent to the sign logic block if the adder output is complemented to correctly compute of the output sign. The MAF output sign bit is adjusted accordingly. Table I shows the truth table of the sign logic block. As can be seen from the table, if $sign_{AB} = sign_C$, the sign of the MAF output $sign_{MAF}$ will have the same sign. However, if the signs differ, then the complementer signal is used to determine $sign_{MAF}$. Keep in mind that C is always the operand that is complemented if the signs differ. Hence, in the case that AB is positive, and C is negative, the sign of the adder output will be the sign of the MAF result since the adder computes the correct operation i.e. $m_{AB} - C_m$. However, In case, AB is positive and C is negative, the complement of the sign of the adder result will be the sign of the MAF output. Also in this stage is the exponent calculation block which compares the the two input exponents and assigns the larger one as the intermediate exponent of the result.

TABLE I
THE TRUTH TABLE OF THE SIGN LOGIC BLOCK.

$sign_{AB}$	$sign_C$	$comp$	$sign_{MAF}$
0	0	X	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	X	1

C. Normalization and Rounding Stage

The third stage converts the adder output from the internal format to the IEEE 754 standard format. This is done by normalization of the adder output and subsequent conversion to the required mantissa width (23 bits for single-precision). The left shift amount is determined by the LZD. Normalization places the first '1' of the result at the right most bit. Note that no right shift is needed since the placement of the significands in the 75-bit internal registers cannot cause addition overflow. The LZD shift amount is subtracted from the intermediate exponent to produce the exponent of the MAF output. Finally, the result is rounded using truncation by taking the 24 MSBs of the normalized result and truncating the rest. Postnormalization is not needed since truncation cannot cause overflow.

III. SOFTWARE MODEL DETAILS

We created a Fixed-Point software model for our implementation using the C programming language. Each block was modeled as a C function that preserves the semantics of the operation of each hardware block. To achieve this, bit manipulation operations (such as C's right/left shifting operations, addition etc.) were used to model the function of each block. We start by initializing 3 floats that will be used for the multiply-add operation. We then perform the appropriate bit-wise manipulations to extract the sign, mantissa, and exponent of each float. Namely, to extract the mantissa, since we know it resides in the 23 LSBs, we mask these bits by ANDing with the bit-mask 0x7FFFFF. We then add the hidden '1' at bit 24 by ORing with a '1' left shifted by 24 ($1 << 24$). To extract the exponent, we mask the 8-bit position of the exponent (bits 30 to 23) by ANDing the float with the bit-mask 0x7F800000 and then right shifting by 23. To extract the sign, we mask bit-31 and right shift by 30. Once we have extracted all three components, we save them in a struct (struct float_i). The struct consists of three elements; mantissa, exponent, and sign. Section IV explains how the functions that represent the each hardware block is coded in C. The software model was successfully compiled and executed on an x86 64-bit Linux machine running Ubuntu 14.04 using the GCC compiler.

IV. SOFTWARE AND VHDL IMPLEMENTATION OF EACH BLOCK

In this section, we provide details about how each block in our designed architecture shown in Figure 1 is coded in C and VHDL to perform their intended functionality.

A. C_m Complement Indicator (Is_Comp) Block

The Is_Comp block is used to indicate to the alignment shifter and complementer block whether C_m should be complemented or not. It takes as input the sign-bit of each of the three floating point operands. The signal $comp$ is then computed by XORing the three signs $comp = sign_A \oplus sign_B \oplus sign_C$. If $comp = 1$ then the alignment shifter complements C_m otherwise, C_m is not complemented. The native XOR operation of VHDL and C are used to perform the XOR operation.

B. Alignment Shift Amount Block

The alignment shift amount block's main function is to compute the amount by which C_m should be left shifted by so that it is aligned with the multiplication result. To accomplish this, $exp_{AB} + 27$ is computed by the following equation $exp_{AB} = exp_A + exp_B - 127 + 27$ where 127 is the exponent bias and 27 is the preshift as explained in Section II-A. $shift\_amount$ is then calculated by the following equation: $shift\_amount = max(0, exp_{AB} + 27 - exp_C)$ since as mentioned earlier, if $exp_C > exp_{AB} + 27$, then C_m is not left shifted. This equation is implemented using the native arithmetic operations, conditional statements('if' in C and 'when' in VHDL) and compare operations ($>$ and $<$) in C and VHDL. The same procedure is done in C except using 64-bit integers. Hence, m_C is preshifted by 17 bits (instead of 24) relative to m_{AB} and $exp_{AB} = exp_A + exp_B - 127 + 17$. To account for this, m_C is right shifted by $shift\_amount$ in case $shift\_amount > 0$. If $shift\_amount < 0$, m_{AB} is right shifted by $max(10, -shift\_amount)$. This adjustment in the C code is used since the compiler used does not support shifting operations for integers larger than 64-bits wide. Note that this does not change the semantics of the operation since the right shifted bits of m_{AB} would have been truncated anyways in the rounding stage. Although, some external libraries have templates that can support larger integers, they have critical disadvantages for our software model implementation. These disadvantages arise due to their complexity and inability to easily implement the various arithmetic and bit manipulation operations, such as shifting and masking, which are extensively used in the design. Moreover, even if implemented, they unnecessarily complicate the software model which can be avoided by performing this simple semantic preserving technique.

C. Alignment Shifter and Complementer Block

The alignment shifter and complementer block takes as input the $shift\_amount$ and $comp$ signals and computes m_C (the aligned mantissa of C) as follows: $m_C = (C_m >> shift\_amount)$ when $comp =' 0'$ else $m_C = -(C_m >> shift\_amount)$. The native srl (shift right logical) VHDL operation is employed to perform the shifting in the VHDL implementation. In case $comp$ is asserted, m_C is complemented by performing the bit-wise NOT operation on m_C and adding 1 $\overline{m_C} = NOT(m_C) + 1$. Having the MSB of C_m placed at the second MSB of the internal register insures that the MSB reflects the sign of the result.

D. 24 x 24 Multiplier Block

The multiplier block's main function is to reinstate the hidden '1' and perform multiplication on A_m and B_m to get the 48 bit result i.e. $m_{AB} = A_m * B_m$. The native multiply operation in VHDL and C are used to perform the multiplication.

E. Sign Logic Block

The Sign Logic block takes as input $sign_{AB}$, $sign_C$ and the $comp_{add}$ signal (which indicates whether or not the adder result has been complemented). The truth table for the sign logic block is shown in Table I. Using this truth table, we find that $sign_{MAF} = sign_{AB} \cdot \overline{comp_{add}} + sign_C \cdot comp_{add}$. The native AND, OR, and NOT operators in VHDL and C are used for this computation.

F. Intermediate Exponent Calculator (Exp_calc) Block

The main functionality of this block is to determine the intermediate exponent of the MAF result. It takes as input $exp_{AB} + 27$ and exp_C and assigns the intermediate exponent the greater of the two input exponents i.e. $exp_{int} = max(exp_{AB} + 27, exp_C)$. This is done using the native comparator operations and conditionals in C and VHDL. For example, the VHDL code for that implements this equation is

```
exp_MAF_int <= exp_C_int
   when (signed(exp_C_int) > signed(exp_AB))
   else exp_AB;
```

G. 75-bit Adder and Complementer Block

The Adder and Complementer block is responsible for adding m_C and m_{AB}. For the VHDL implementation, the adder block takes the 48-bit multiplier output m_{AB} and places it at the 48 LSBs of a 75-bit STD_LOGIC_VECTOR. It then performs the addition of the two operands. If the result is negative (which is determined by checking the MSB of the result), then result is complemented and $comp_{add}$ is asserted. The sign-bit is then discarded and the 74-bit result is passed to the next stage. The addition operation is performed directly in C since the operands are already supplied in their correct alignment and bit width at the input.

H. Leading Zero Detector Block

The main function of the Leading Zero Detector (LZD) block is to determine the amount by which the adder result needs to be left shifted by for normalization. The shift amount is determined by starting from the right and counting the number of leading zeros before the first '1' is seen. Since the adder result is complemented incase it is negative, this insures that only a LZD counter is needed since the result is always going to be positive (i.e. does not have leading 1s instead of 0s). Below is a VHDL code snippet for this block

```
LZN <=  "0000000" when add_result(73) = '1' else
        "0000001" when add_result(72) = '1' else
        "0000010" when add_result(71) = '1' else
        "0000011" when add_result(70) = '1' else
        ...
        "1000111" when add_result(2) = '1' else
        "1001000" when add_result(1) = '1' else
        "1001001" when add_result(0) = '1' else
        "1001010";
```

where LZN is the left shift amount. The LZD function is implemented in C using a while loop, MSB mask, and a counter. The while loop condition checks if the MSB is '1'. If not, then adder result is shifted to the left and the counter is incremented. Note that adder result is passed by value and hence left shifting does not affect the actual adder result that was passed to the function. The loop exits if a '1' is detected at the MSB. Part of the C code for the LZD function is

```
MSB = (uint64_t)1 << (uint64_t)63; // MSB mask

LZN = 0;
while(!(adder_result & MSB))
{
   adder_result = adder_result << (uint64_t)1;
   LZN++;
}

return LZN;
```

I. Exponent Adjustment (Exp_adj) Block

The exponent adjustment block calculates the final exponent of the MAF result by subtracting the output of the LZD LZN from the intermediate exponent. In other words $exp_{MAF} = exp_{int} - LZN$. To see why this works, consider the three cases discussed in Section II-A. If $d > 27$, then $exp_C > exp_{AB}+27$ and hence the intermediate exponent is exp_C as determined by the Exp_Calc block discussed previously in Section IV-F. Since in this case m_C is not left shifted, no leading '0's will be detected (since the sign bit is discarded by the adder before passing to the next stage). Therefore, $LZN = 0$ and $exp_{MAF} = exp_C$ as required. For the case where $d \leq 27$, this implies that $exp_{AB}+27 \geq exp_C$ and $exp_{int} = exp_{AB}+27$. In that case, the radix point is implied to be between the first and second MSBs of the internal register. Hence, by left shifting the result, we can visualize this as moving the radix point to the right so that it is after the actual MSB of the result (i.e. the first '1') and therfore we need to decrease the exponent accordingly.

J. Normalize and Round Block

The function of the normalization block is to shift the 74-bit adder result by LZN i.e $MAF_m = adder\_result << LZN$. To perform this left shifting in VHDL, the sll (shift left logical) operator was used. Then, bits 72 down to 50 were used as the MAF mantissa output thereby effectively truncating the result and removing the hidden one. In C, the native left shift operator was used followed by right shifting the result by 39 to perform the truncation and place the manitssa at the 23 LSBs of the integer of the function output.

V. Testing and Verification

In this section, we describe the test cases that were performed to verify correct functionality of the our VHDL implementation and software model. To test the software

$sign_{AB}$	$sign_C$	$exp_{AB} > exp_C$			$exp_{AB} < exp_C$		
		A	B	C	A	B	C
+	+	100.5	20.22	51.8	5.25	1.22	21.5
-	-	-90.25	20.22	-31.8	-5.28	1.22	-21.5
+	-	13.28	15.22	-31.8	0.28	0.22	1021.5
-	+	-15.28	11.22	21.5	0.5	-0.5	1000000.06

Fig. 3. Values of A, B, and C for all the test cases of the software and VHDL implementation.

TABLE II
TARGET FAMILY AND TOTAL LOGIC ELEMENTS OF THE FLP MAF UNIT.

Family	Total Logic Elements
Cyclone IV GX	1,822

model, we start by performing the floating point computation $F = A * B + C$ where F, A, B, and C are floating-point numbers initialized in C. We then convert A, B, and C to our Fixed-point C struct as described in Section III. We then perform the computation using our fixed-point software implementation described in Section IV. We then convert F to our fixed-point struct and compare it to the result from our software model. Table 3 shows the test cases that were used to compare F with the result from our software model. As can be seen from the figure, we cover all possible combinations for the sign of AB and C. For each case, we test the scenario when $exp_{AB} > exp_C$ and the case where $exp_{AB} < exp_C$.

The rounding scheme of the native C implementation is round-to-the-nearest-even while our rounding scheme is through truncation. Accordingly, the result of our software model matched exactly the result of the native C floating-point operation (F) except in the cases where rounding caused the mantissa of F to be incremented by 1. In that case, the mantissa of our software model result was equal to the mantissa of F subtracted by 1 as expected.

After verifying correct functionality of our software model, we used the same test cases to verify correct functionality of our VHDL implementation. FLP_MAF_test.vhd was the test file used. The file contains all the test cases used for our VHDL implementation shown in Figure 3. The equivalent bit pattern of the floating-point representation of each operand was provided as input. The output bit-pattern was compared to the output bit-pattern of our software model. The output of our FLP MAF unit exactly matched the expected output provided by the software model. Table II shows the number of logic elements of our design. Our design was successfully synthesized using the Quartus II software targeting Cyclone IV GX. ModelSim-Altera was used for simulation.

VI. CONCLUSION

We successfully implemented a single-precision floating-point MAF unit that is based on the conventional MAF unit found in the literature and integrated in commercial processors [2], [4], [7]. We also implemented a bit accurate software model for the design. Possible extensions to the current design include pipelining the implementation to increase performance, using a LZA instead of LZD to decrease the critical path delay, and reducing the overall delay and increasing performance by employing some of the techniques found in recent publications such as utilizing DSP blocks in FPGAs to perform the multiplication and addition as done in [5].

REFERENCES

[1] C. Chen, L.-A. Chen, and J.-R. Cheng, "Architectural design of a fast floating-point multiplication-add fused unit using signed-digit addition," *Computers and Digital Techniques, IEE Proceedings* -, vol. 149, no. 4, pp. 113–120, Jul 2002.

[2] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with flopoco," *Design Test of Computers, IEEE*, vol. 28, no. 4, pp. 18–27, July 2011.

[3] E. Hokenek, R. Montoye, and P. Cook, "Second-generation risc floating point with multiply-add fused," *Solid-State Circuits, IEEE Journal of*, vol. 25, no. 5, pp. 1207–1213, Oct 1990.

[4] R. Montoye, E. Hokenek, and S. 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, Jan 1990.

[5] A. Amaricai, O. Boncalo, and C.-E. Gavriliu, "Low-precision dsp-based floating-point multiply-add fused for field programmable gate arrays," *Computers Digital Techniques, IET*, vol. 8, no. 4, pp. 187–197, July 2014.

[6] T. Lang and J. Bruguera, "Floating-point multiply-add-fused with reduced latency," *Computers, IEEE Transactions on*, vol. 53, no. 8, pp. 988–1003, Aug 2004.

[7] R. Jessani and C. Olson, "The floating-point unit of the powerpc 603e microprocessor," *IBM Journal of Research and Development*, vol. 40, no. 5, pp. 559–566, Sept 1996.