Design and Analysis of a Floating Point Fused Multiply Add Unit using VHDL

Farouq Aliyu
Department of Computer Engineering
King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia 31261

***Abstract*—Ever since the growth of processing speed began to slow down due to immense heat generated by the processing units, engineers started looking for other speed up alternatives. One of such is the "Fuse Multiply Add" (FMA) unit. This unit combines multiplication of two operands and their summation with third operand as a single instruction. As a result, a floating point primitive is created out of the two arithmetic operations.**

This paper describes the design and development of a FMA. It also points out the limitations of the FMA using VHDL programming language. Furthermore, the paper points the mechanics of the different parts of the FMA in order to shade more light on those parts of the FMA that prove costly in terms of speed, power and area.

Keywords*—Computer Arithmetic, Fused Multiply-Add, FMA, IEEE Floating-point, Leading Zero Detector, Leading Zero Anticipator

I. Introduction

Floating point multiplications and additions are a norm in Digital signal Processing and multimedia application [1]. With the optimization of floating-point operations reaching diminishing return [2], it is apparent that innovative solutions are necessary. Fused Multiply-Add (FMA) operation was included in IBM RS/6000 in 1990 [3][4]. FMA combines multiplication and addition into one instruction as described by Equation 1. Two operands can be added by making B=1 and two operands can be multiplied by making C=0. This flexibility means there is no need for additional multiplier (that does only multiplication) or adder (that will perform addition only) as long as there is a FMA. In general, the unit increases throughput due to fusing of the two operations and also increases accuracy since rounding is carried out once instead of twice.

$$R = ((A \times B) + C) \tag{1}$$

The FMA can be broken down into six basic operations that work together to carry out the fused multiply-addition. The following are the FMA's basic operations;

1) Multiplication
2) Alignment
3) Addition
4) Leading Zero Anticipation (LZA)
5) Normalization, and
6) Rounding.

First, the two operands (A and B) are multiplied. The product is shifted left or right in order to decrease or increase its exponent respectively. This process is known as alignment. It allows the exponent of the product and that of the third operand (C) to have the same value, which is necessary for accurate addition. The alignment could be done in parallel with the multiplication, such that C is aligned while A and B are multiplied. This helps save some clock cycles. After alignment, comes the addition of the product of A and B to C. However, in order to maintain IEEE (normalized) format for the final result (i.e. R), it is necessary to locate the leading one bit of the final sum. This could be done with the help of LZA, which counts the number of leading zeros before the first 1 is found. LZA is also known as Leading One Predictor (LOP) [5]. The process can be implemented in parallel with the addition. The counts are then collected by a shifter which shifts the result accordingly, thereby normalizing the sum to IEEE 754 format. Finally, the sum is rounded with the round-to-nearest even (RNE) [1].

This project is aimed at developing and evaluating the performance of a Fused Multiply-Add unit in an FPGA using Xilinx. The module is equipped with control registers that can be used to select the rounding method. In addition, the module is also equipped with flag registers that help notify the user of the state of the module. The FMA has been designed with embedded systems in mind, in an effort to provide engineers with energy efficient FMA solution.

The remaining section of this paper is as follows: Section II reports a review of advances in FMA designs available in the literature. Section III presents detailed explanation of the proposed system. Section IV provides information about how testing took place and Section V presents the corresponding results obtained from the experiments carried out. Finally, Section VI concludes the paper and provides a summary of future work.

II. Literature Review

Increase in throughput and decrease in latency and reduced hardware overhead are the key features of FMA [6]. Several works have been carried out in the effort to improve the design of FMA unit. Quach *et al.* [7], reported three techniques for developing the FMA unit. These are;

1) non-overlapped
2) Fully overlapped (greedy)
3) Partially overlapped

The non-overlapped method is a mere concatenation of the multiplier and the adder of a floating-point unit. Therefore, there no speedup is gained. This method was used by earlier floating point units.

Fully overlapped (also known as the greedy) approach is the process where the product calculation, the true exponential calculation and the (right) shifting of the addend overlap. Therefore the speed up is a factor of two compared to the non-overlapped FMA. That is to say, shifting of addend is done before the partial products arrive. The addition between the product and the addend takes place in a carry save adder. The partial sums are then added using a full adder and normalization is carried out afterwards. Hokenek *et al.* [6] proposed that in order to get the best performance, products should first be calculated in parallel with true exponent calculations and aligning the addend in the two possible directions (i.e. either left or right). Afterwards, post normalization and rounding may take place. The authors further suggested the use of modulo shifter for fast shifting. However, the uses of modulo shifters incur high latency and due to the fact that they are combinatorial devices make them consumes substantial amount of power. In an effort to solve this problem, [8] suggested the use of a multilevel shifter. However, the register for the addend C must be expanded by one hundred and six bits in order to be able to shift it in both directions. This has a negative implication on the area of the multiplier, because the size of the addend has to be tripled. Louca *et al.* [9] proposed shifting right only. In this technique, only fifty three bits are added to one of the addend registers. The arguments are swapped depending on which is to be shifted right. Although area is saved, speed is reduced. Finally, the addition section of the fully overlapped method has simpler but larger data path than that of a single floating point adder, hence it has more wires when implemented [7].

Partially-overlapped FMA, as the name implies, is a technique whereby the floating point multiply sub-module and the floating point add sub-module partially overlapped. The true exponent calculation and the product, right-shift and addition and left-shift and a final addition are overlapped. Unfortunately the wiring and area overhead of this method is greater than that of the greedy approach.

Seidel *et al.* [10], optimized the adder. The authors believe that a lower average latency can be achieved, if the addition section is selected out of several other addition sections base on the complexity of the product of A and B (i.e. fprod) and the addend (i.e. fC). Figure 1 shows the cases considered by the authors.

$$\delta = ea + eb - ec \tag{2}$$

Where: ea, eb and ec are the true exponents of the operands A, B, and C respectively.

The authors categorize this complexity using the identifier δ. The identifier is described by Equation 2. The possible values obtainable were grouped into five classes as listed by Table I. These groups were based on the possible result of the addition: group one is for cases where one of the arguments has no significance in the operation and can be approximated to zero. Example of such cases is Case 1 and 5. Group two, these are operations that require no pre-normalization after the multiplication. In such cases no LZA in needed. Case 2 and 4 fall within this category. Finally, the third group where all FMA processes are needed and this is where Case 3 falls.

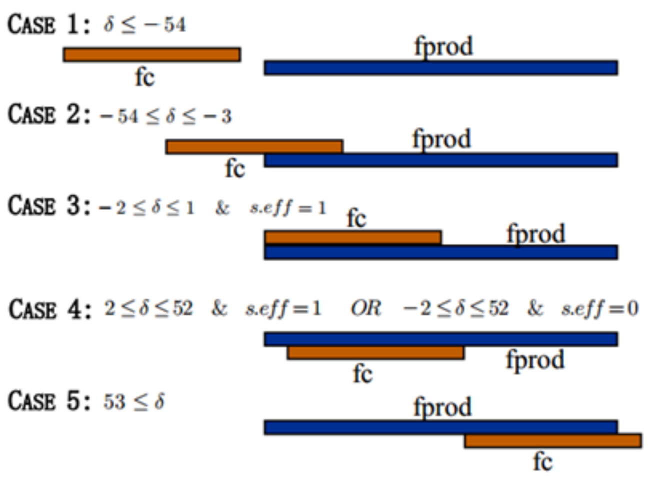

CASE 1: $\delta \leq -54$
fc
fprod
CASE 2: $-54 \leq \delta \leq -3$
fc
fprod
CASE 3: $-2 \leq \delta \leq 1$ & $s.eff = 1$
fc
fprod
CASE 4: $2 \leq \delta \leq 52$ & $s.eff = 1$ OR $-2 \leq \delta \leq 52$ & $s.eff = 0$
fprod
fc
CASE 5: $53 \leq \delta$
fprod
fc

Fig. 1. Criterion for multiple path selection [10].

The technique is able to gain 26% more speed that the IBM system proposed in [4]. However, it is obvious that there is huge overhead in terms of area and power, given in to account: 1) the system has two adders and two rounding modules and 2) the several stages of comparators that are needed before decision is made. Furthermore, some of the cases may not be used over a long period of time, since smaller numbers are more likely to be involved in a computation compared to their larger counterparts [8]. For this reasons [11], selected only three cases in their implementation of a Quadruple Floating-point Fused Multiply-Add (QPFMA). The chosen cases are the case are; 1) when fprod is negligible (which is similar to Case 1), 2) when fc is negligible (which is similar to Case 5) and 3) everything in between is considered a third case.

TABLE I. SELECTION CONDITIONS AND THEIR IMPLICATIONS

Case	Condition	Conclusions
1	$\delta \leq -54$	If mode is round to infinity then result equal post_norm($f_c + 2^{-53}$), else result equal f_c.
2	$-54 \leq \delta \leq 3$	Complementation of fprod (when subtraction operation is been executed) and pre-normatlization (before addition) can be ignored.
3	$-2 \leq \delta \leq 1$	Parallel computation of significand summation, rounding and leading zero count. Integrated computation of pre-normalization before addition and Post-normalization shifts
4	$2 \leq \delta \leq 54$	Complementation of fprod (when subtraction operation is been executed) and pre-normatlization (before addition) can be ignored.
5	$54 \leq \delta$	Contribution of fc is only to the rounding digits

Our aim in this paper is to design a simple implementation of FMA that will allow us investigate in details the limitations and avenues for possible optimization in each of a FMA. We believe that this research will paint a vivid picture of how a FMA works and will also help scientists figure out ways to improve its performance.

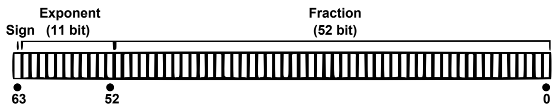


Fig. 3. IEEE 754 floating point format

III. Fused-Multiply Adder

Figure 2 shows the block diagram of the FMA developed. The proposed system consists of an Input stage where data is collected, an unpacking stage where the significand and the exponent are extracted, a multiplication stage, an addition stage and a rounding stage. It should be noted that a simple module that finds the leading zero was developed instead of an elaborate LZA.

A. Inputs

Even though the floating point FMA is parameterized such that it can work with any range of values for the sign, exponent and mantissa that follow the IEEE 754 floating point format rules, this paper focuses on the double precision IEEE floating point numbers. Double precision is chosen because it is half way between single and extended IEEE floating point format as can be seen in Table II. IEEE floating point numbers are composed up of three basic components:

1) Sign bit,
2) Exponent, and
3) Mantissa (i.e. Significand).

The sign bit "S" is the first bit, when it is one it represents negative number, else the number is positive. The exponent is represented by "E". The exponent field needs to represent both positive and negative exponents. This is done when a bias is added to the true exponent. For double precision, the exponent field is 11 bits, and has a bias of 1023. The mantissa is denoted by "f". The mantissa ranges [1, 2) and it represents the fraction part of the number represented. An implicit leading one (1) is also found in IEEE 754 format. This leaves us with the format shown in Equation 3. Table II shows the layout of the three widely used IEEE floating-point formats. The number of bits for each field are shown (bit ranges are in square brackets):

$$(-1)^S \times (E) \times (1.f) \tag{3}$$

The input sub-module is responsible for collecting the arguments from the user. A user can be any device that is using the FMA. Therefore, the input is a set of ports that collect data from the outside world and pass it to the proposed system without any modifications.

TABLE II. IEEE 754 DOUBLE PRECISION FLOATING POINT FORMAT

	Sign	Exponent	Fraction	Bias
Single Precision	1 [31]	8 [30-23]	23 [22-00]	127
Double Precision	1 [63]	11 [62-52]	52 [51-00]	1023
Extended Precision	1 [79]	15 [78-64]	64 [63-00]	16383

B. Unpack

The values are provided in an 8 byte format with no demarcation whatsoever. The function of unpack is to break the inputs into their respective components. For example sA, eA and fA shown in Figure 4, represent sign bit of the first operand (i.e. A), its exponent and its fraction part respectively. It is worth noting that the mantissa has an implicit bit that is one (1). Moreover, the implicit digit automatically turns to zero when the exponent is equal to zero (0). Furthermore, if $E = 0$ and $f = 0$ then the floating point number is zero (0). However, when $f \neq 0$ then the floating point is a subnormal (also known as a Denormal) which is very difficult for mathematical operations. Therefore, in such cases where subnormal number is formed, our proposed system halts and a flag is set to inform the user that an error has occurred.

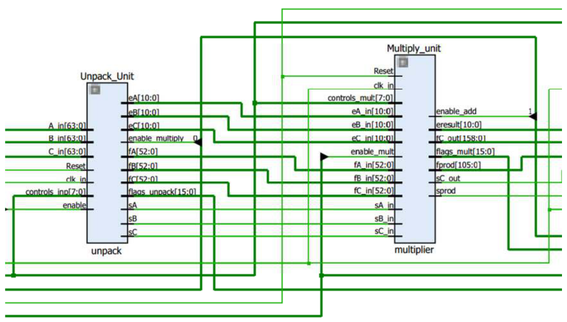


Fig. 4. Schematic diagram of Unpacking module connected to Multiply_unit.

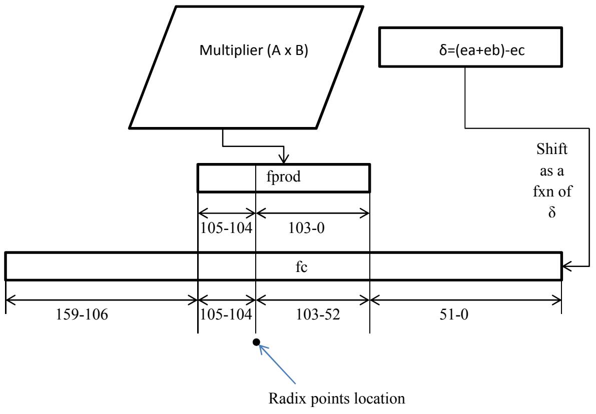


Fig. 5. Multiplier structure and how fC is placed with respect to the radix point.

For cases where $E = 0 \times 7FF$ meaning all bits of exponents are ones (1). Two possible values are formed depending on f; when $f = 0$ means the value is infinity (positive or negative depending on the sign bit), when $f \neq 0$ then it is "Not-a-Number" (NaN). This module is designed to fish out these special cases and raise the corresponding flag. Raising flags is necessary in other to notify the user what halted the calculation process and to distinguish final answer from error.

C. Multiplier

The main function of the multiplier stage is to add the exponents eA and eB and multiply the significands $1.fA$ and $1.fB$. The identifier Δ is calculated at the same time the multiplication is taking place. The identifier is used to shift fC either left or right depending on whether Δ is positive or negative respectively.

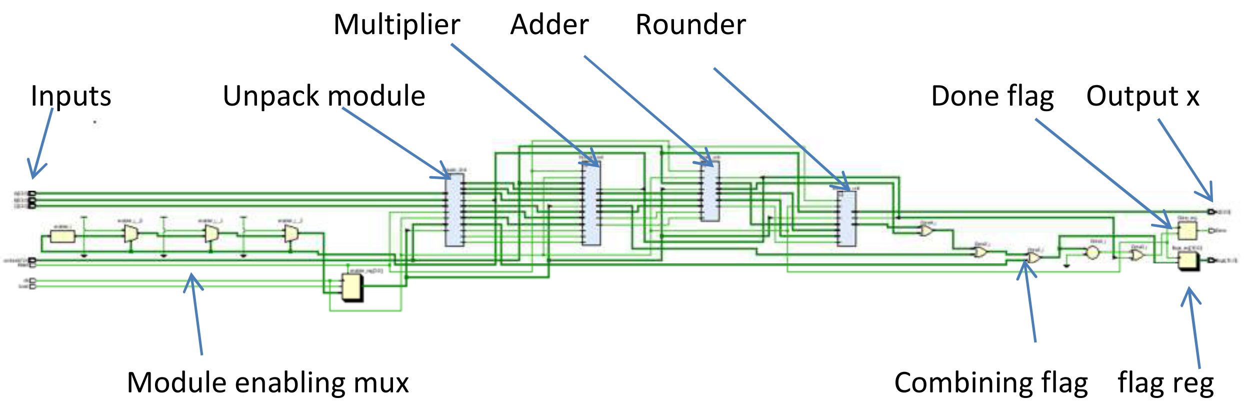


Fig. 2. Schematic diagram of the Fused Multiply Add (FMA).

TABLE III. FLAGS USED BY THE FMA

NaN_A	NaN_B	NaN_C	Inf_A	Inf_B	Inf_C	NaN_X	Inf_X	Inv	Ovfl	Ufl	Denum_A	Denum_B	Denum_C	Denum_X	-

NaN : Not a Number
Inf : Infinity
Denum : Denormal
Inv : Invalid
Ovfl : over flow
Ufl : Under flow

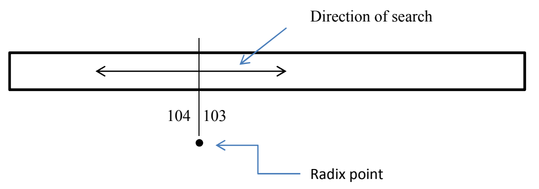


Fig. 7. Searching for leading one detector.

In order to achieve addition in the next stage after multiplication, fC is loaded in a register of size $3f$ (where f is the size of a significand – that is 53-bits). The significand fC is loaded to the register such that it align with the (virtual) radix point between bit 104 and 103 as shown in Figure 5. Afterwards, the following steps take place;

1) If $ea \neq 0$ and $eb \neq 0$ then $eresult \leq (ea+eb)-bias$, else $eresult \leq ec$.
2) $\Delta = ((ea+eb)-ec)-bias$
3) if Δ is ve then shift f_c Δ times to the left, else shift it Δ times to the right

Once multiplication is completed, "fprod" which is the product of the operation is loaded in a register $2f$ wide with the same alignment rule that applied to fC point. That is to say, its radix point must be between bit 104 and 103. The two results are then passed to the adder, the connection between adder and multiplier is shown in Figure 6.

D. Adder

The adder adds or subtracts fprod from fC depending on their sign. The result is "fr". The value fr is then saved on a 3f wide register and it is passed to a leading one detector LOD, which is designed to scan the register in both directions from the radix point.

Algorithm 1 describes how the LOD works. The LOD scans the register containing fr in order to find the first bit whose value is one as the register is observed from left to right. The scanning is started from the center of the register and it spans out towards both ends. This helps reduce the scanning time by one-half and ensures that all bit are not missed.

Algorithm 1 Finding the leading zero

```
1:  while not end of register do
2:      if $(fsum(My\_radix + count) =' 1')$ then
3:          $east\_marker \leftarrow count$
4:      end if
5:      if $(fsum(My\_radix - count - 1) =' 1')$ then
6:          if $(west\_one = False)$ then
7:              $west\_marker \leftarrow count + 1$
8:          end if
9:      end if
10:     $count \leftarrow count + 1;$
11: end while
```

The register fr is shown in Figure 7. For every while loop, a pointer called "count" shifts one bit away from the radix point. Both sides of the radix point are checked in each loop by the two if-statements at step two (2) and five (5) of Algorithm 1. After the scanning is over, shifting right has high priority that shifting left. Therefore, if "east_marker" is greater than zero then a shift right east_maker times is carried out and west_marker is neglected. Clearly, this technique is slow, because it searches for the missing bit in an iterative manner. Hence it has a complexity of O($\frac{n}{2}$), where n is the size of the register. For this reason, LZAs were developed. They search for the leading one using a non-iterative technique, hence it has a complexity of O(1). The operation is carried out at the same time the addition of fprod and fC is carried out.

The new exponent becomes the sum of the old exponent and the east_marker or the west_marker depending on whether,

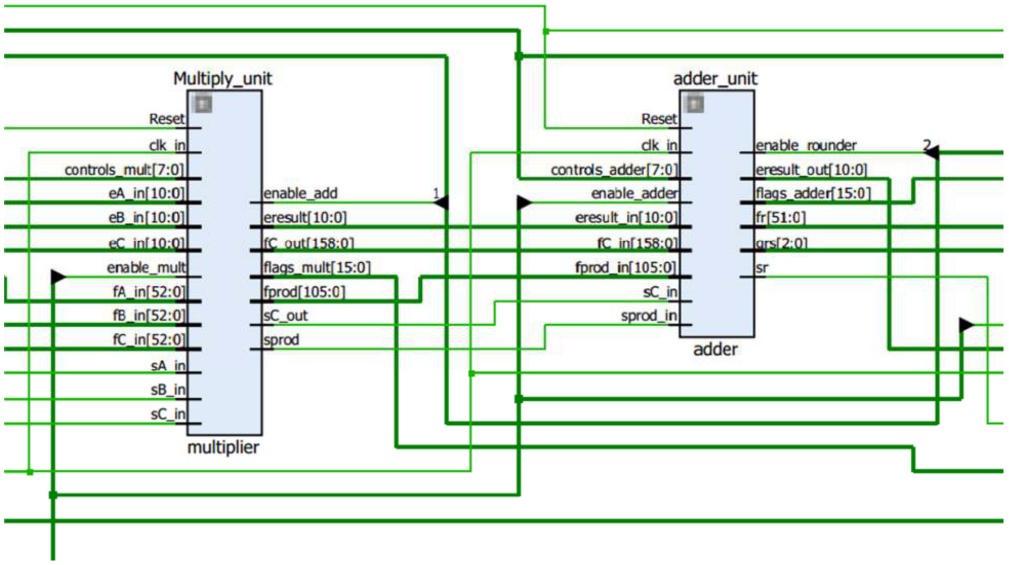


Fig. 6. Schematic diagram of the connection between the multiplier and adder.

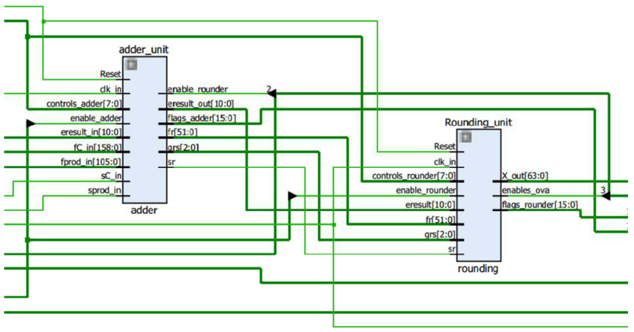


Fig. 8. Schematic diagram of connection between the Adder and Rounding unit.

right- or left shifting took place. The exponent is then checked for overflow or underflow and the corresponding flag (as shown in Table III) is set to notify the user. The operation is exited when either overflow or underflow is found. An underflow occurs when $eresult \leq 0$, while an overflow occurs when $(eresult \geq 2^{number\_of\_exponential_bits}1)$. The variable $eresult$ is the resulting exponential after normalization. Finally, when neither overflow nor underflow has occurred the result of is forwarded to the rounding unit. The registers connected to the rounding unit are;

1) eresult
2) fr truncated to 52 bits
3) guard bit
4) round bit and
5) sticky bit.

Gaurd bit and round bit are the 53rd and 54th from the radix point of fr. The remaining bits are Ored and the result is the sticky bit. These bits help improve the rounding accuracy.

E. *Rounding unit*

The rounding unit does the rounding and packing of data. The rounding unit first checks the control register to determine which rounding mode is selected by the user. Table IV shows the control register. Two most significant bits of the control register (i.e. Controls_rounder[7:6]) where used for this purpose to select between different rounding modes;

00	Round to nearest
01	round to nearest even,
10	round to +ve infinity and
11	Truncate.

The $\overline{silent}$ control bit helps the user suppress the error warnings. This control bit comes in handy during system testing.

TABLE IV. CONTROL REGISTERS

Round1	Round0	$\overline{Silent}$	-	-	-	-	-

Where;
Round1: MSB of the rounding control
Round0: LSB of the rounding control
Silent: (Active low) disable invalid flags

IV. EXPERIMENT

The latency of the FMA was tested using a test bench. A test bench is a VHDL code that is developed in order to test and verify the designed circuit [12]. The test bench is connected to the circuit, an Input file and an output file. The input file provides the test bench with data with which it tests the performance of the proposed FMA. Finally, we used the Xilinx synthesis tools to evaluate the power and area of the system. The resources for the proposed system and the test bed can be found in [13].

A. *Input file*

The system was tested using a test bench developed using VHDL. The test bench collects input from a text file called "testing_doc/input.txt" — testing_doc is the name of the folder where this file lies. Furthermore, this folder is stored in the working folder for the project. This arrangement allows the test bench to access the input file. The input file is in the following format [A B C X]. It gives the test bench the operands A, B and C and the last column contains the expected solution "X" as expressed in Equation 1. The test bench feeds the unit under test (UUT) with the operands A, B and C. The solution X is returned from the UUT to the test bench. The test bench compares the returned result with the solution from the input file. A variable in the test bench called "err" is used to control the percentage error allowed. The percentage error is represented in terms of number of LSB in error allowed by the UUT.

B. Input file generator

The file input.txt contains 1500 sets of inputs for testing the UUT. These inputs were generated by a c++ code. Figure 9 shows the flowchart for the software.

The software generates numbers randomly, user has the option of selecting number of decimal places and whether signed (i.e mixture of positive and negative numbers) or positive numbers only. The software has default settings with one digit whole number and one digit fraction, but it produces only positive numbers. For the sake of convenience the software produces two files and save them in the folder, testing_doc automatically. These two files are;

1) Input.txt: this file contains only hexadecimal numbers places in three columns. The columns represent A, B, C and X.
2) Bloody_humans.txt: this text file contains a corresponding decimal interpretation of the numbers in the input.txt file for easy testing and debugging.

C. Output

The output from the test is saved as output.txt. It consists of three types of columns; the answer, the flags and comment. The answer column prints the answer produced by the UUT in hexadecimal form for easy comparison with the actual answer. The second column prints an abbreviation of the flags set during a calculation. The third column shows whether the answer is equal to the expected answer or not. The word "right" or "wrong" is printed depending on whether the two match or they do not.

V. Results

The test bench is used to test the latency and accuracy of the system, while the Xilinx synthesis tool was used to investigate the power and area of the system. The results of the implementation are as follows;

A. Speed/Latency

In order to measure the speed of the proposed system, the clock period was continuously decreased from ten and the total negative slack is checked for negative values. This process was stopped after a negative value is obtained. A negative value is obtained at 2.5 ns, hence 3 ns is accepted as the minimum clock period. Therefore, the system has a speed of 333.33 Mhz.

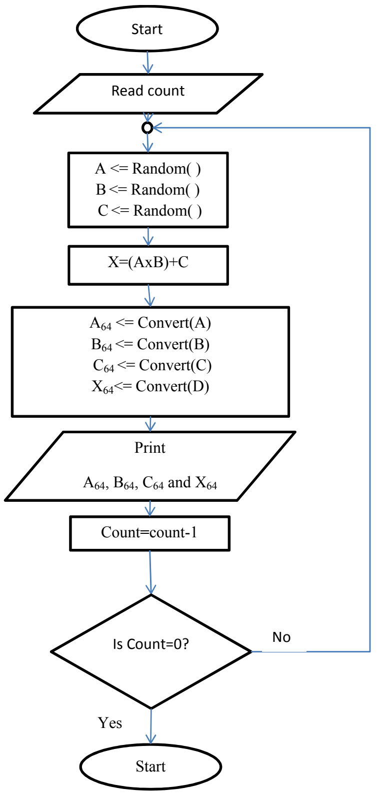


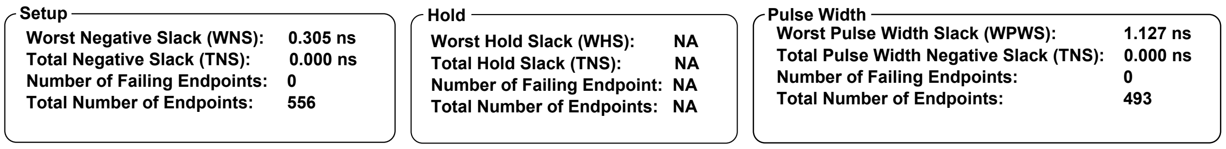
Fig. 9. Flowchart for file generator.

Figure 10 shows summary of the timing characteristics of the system. The speed of the system is limited by the shifters and the Leading One Detector. Finally, Xilinx Modelsim was used to investigate the number of clock cycles needed for the one FMA operation. Figure 11 shows a snapshot of the simulation. Equation 6 shows that it takes the unit 2400 clock cycles for a single FMA operation to be completed, which is $3 \times 2400 \approx 7.2msec$.

$$T_{clock} = 100ns \quad (4)$$
$$T_{FMA} = 239950ns \quad (5)$$
$$\begin{aligned} \therefore Clock\_cycle_{FMA} &= \frac{T_{FMA}}{T_{clock}} \quad (6) \\ &= \frac{239950}{100} \\ &\approx 2400cycles \end{aligned}$$

B. Power

Figure 12 shows how the dynamic and static power varies with frequency. It can be seen that the dynamic power increases exponentially with the logarithm of the frequency ($\log_{10}^{frequency}$). The maximum power that can be consumed by the unit is 112 mW static power and 104 mW dynamic power, making a total of 216 mW. Furthermore, the system at its highest speed can be powered with 3.3v power supply and it will consume only 65.45mA. This power which can easily be provided by battery from hand-held devices.



Setup

Worst Negative Slack (WNS):	0.305 ns
Total Negative Slack (TNS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	556

Hold

Worst Hold Slack (WHS):	NA
Total Hold Slack (TNS):	NA
Number of Failing Endpoint:	NA
Total Number of Endpoints:	NA

Pulse Width

Worst Pulse Width Slack (WPWS):	1.127 ns
Total Pulse Width Negative Slack (TNS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	493

Fig. 10. Summary of timing characteristics of the system.

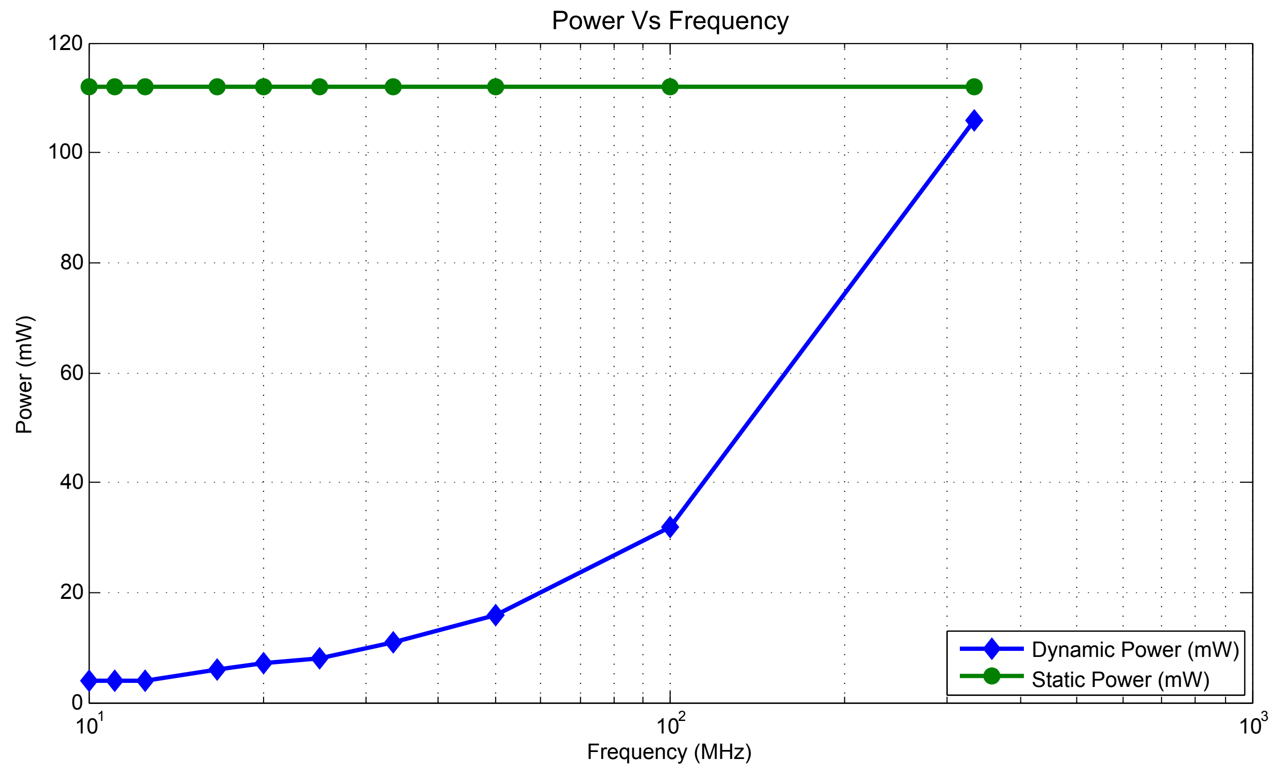


Fig. 12. Power consumption report.

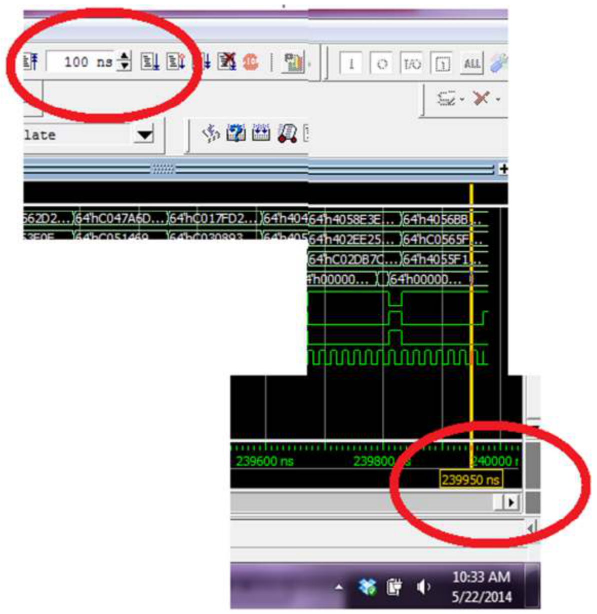


Fig. 11. Simulation run using modelsim.

C. Accuracy

In order to compare all computations from the c++ complier (i.e. the input file generator) and the FMA unit, a variable "err" is used to tell the test bench the number of LSBs to ignore. Table V shows the outcomes of the experiment at different numbers of err. It is worth noting that this does not mean the FMA unit in not accurate, because;

1) It is not known which rounding mode is used by the c++ complier.
2) FMA could be more accurate since rounding is done after all operations that are carried out.

TABLE V. ACCURACY OF THE SYSTEM

Simulation Run	Total set of inputs	No. of Results in error	Accuracy (%)	LSBs ignored (nibbles)
1	1500	180	88	0
2	1500	3	99.8	1
3	1500	1	99.9333	2
4	1500	0	0	3

D. Area

The area of an FPGA is fixed, therefore we can only report the percentage of the area utilized in by our implementation on a given FPGA. Table VI shows the percentage area of FPGA that is been utilized. The FPGA, ”xc7k160t fbg676-2” was used [14] for the FMA implementation.

TABLE VI. AREA UTILIZED BY THE PROPOSED FMA

Resource	Utilization	Available	Utilization %
Slice Logic	5086	380250	1
Slice Logic Distribution	10201	263148	4
DSP	9	600	2
IO and GTX Specific	279	2342	12
Clocking	9	248	4

VI. CONCLUSION AND FUTURE WORK

The proposed multiplier is test by random combination of A, B and C values. The values of X were calculated from the operands. One thousand five hundred set of A, B, C and X values were generated and they were fed to the proposed system. It is observed that the system performs faster than expected. This is because the system checks the values before any computation takes place. As such those values that are infinite or not a number are ignored. This shows us that the system will perform better than expected in real life situations where cluster of computations are carried out at a given time.

Our future work will focus on optimizing the multiplier and the adder. Furthermore multilevel shifter and a leading zero anticipator will be used in order to speed up the system.

REFERENCES

[1] K.-Y. Wu, C.-Y. Liang, K.-K. Yu, and S.-R. Kuang, “Multiple-mode floating-point multiply-add fused unit for trading accuracy with power consumption,” in *Computer and Information Science (ICIS), 2013 IEEE/ACIS 12th International Conference on*. IEEE, 2013, pp. 429–435.

[2] N. T. Quach and M. J. Flynn, *Suggestions for implementing a fast IEEE multiply-add-fused instruction*. Computer Systems Laboratory, Stanford University, 1991.

[3] E. Quinnell, E. Swartzlander, and C. Lemonds, “Floating-point fused multiply-add architectures,” in *Signals, Systems and Computers, 2007. ACSSC 2007. Conference Record of the Forty-First Asilomar Conference on*, Nov 2007, pp. 331–337.

[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] C. N. Hinds and D. R. Lutz, “A small and fast leading one predictor corrector circuit,” in *Proc. 39 Asilomar Conference on Signals, Systems and Computers*, 2005, pp. 1181–1185.

[6] 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.

[7] N. T. Quach and M. J. Flynn, *Leading one prediction–Implementation, generalization, and application*. Computer Systems Laboratory, Stanford University, 1991.

[8] I. Koren, *Computer arithmetic algorithms*. Universities Press, 2002.

[9] L. Louca, T. Cook, and W. Johnson, “Implementation of ieee single precision floating point addition and multiplication on fpgas,” in *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, Apr 1996, pp. 107–116.

[10] P. Seidel, “Multiple path ieee floating-point fused multiply-add,” in *Proceedings of The IEEE Midwest Symposium On Circuits And Systems*, vol. 46, no. 3. LIDA RAY TECHNOLOGIES INC.,, 2003, p. 1359.

[11] J. He and Y. Zhu, “Design and implementation of a quadruple floating-point fused multiply-add unit,” in *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering*. Atlantis Press, 2013.

[12] P. Ashenden, *The Designer’s Guide to VHDL*, ser. Systems on Silicon. Elsevier Science, 2010. [Online]. Available: https://books.google.com.sa/books?id=XbZr8DurZYEC

[13] F. M. Aliyu, “Fma source code,” 2015. [Online]. Available: https://www.researchgate.net/profile/Farouq_Aliyu/contributions

[14] Xilinx, “7 series fpgas overview,” 2015. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf