

Optimization Effects on Modeling and Synthesis of a Conventional Floating Point Fused Multiply-Add Arithmetic Unit Using CAD Tools.

Jaafar M. Alghazo, Assistant Professor in Computer Engineering, Prince Mohammad Bin Fahd University, jghazo@pmu.edu.sa

Abstract

In this paper, a high speed Arithmetic synthesizable Fused Multiply Add Unit (FMA) is modeled capable of implementing the following operations: Addition/subtraction and multiplication. With area speed tradeoff limitation, concentration is on modeling high speed arithmetic units with moderate area increase. Thus, the concentration is on developing units that share the same hardware. A model of a high speed arithmetic fused multiply add unit ($A * B + C$) Capable of addition/subtraction and multiplication is implemented. The focus is on reducing the delay in critical path by identifying the most time consuming operations in the critical path of a basic multiplication/addition fused unit. CAD tools are used to model this system. Once modeled and synthesized the system is downloaded onto a FPGAs chip. The chip became a stand alone FMA unit capable of implementing the operations mentioned. Synthesis tools are used to evaluate these designs and reports showed that the estimated minimum delay of the designed unit was 112.917ns. After implementing optimization techniques and modeling the FMA unit using Verilog synthesis tools, the estimated the minimum delay of the design unit was 9.236 ns. The efficiency of the system is therefore increased by over a factor of 10.

1. Introduction

In this paper a single precision floating point unit that can perform multiplication and addition/subtraction is modeled. This arithmetic unit can be used for extensive arithmetic application such as Digital Signal Processing (DSP), graphic applications, and scientific computations. It is also a practical arithmetic unit where area and speed are within practical ranges of application Specific Integrated Circuits (ASIC) designs. The operations chosen in this design were important operations in many applications. Most DSP algorithms are extensively inhabited with the multiplication operation, executing half of their instruction in the multiplier. The capability of a multiplication/addition operation is an essential part of modern day DSP processor.

The unit is configured for FPGAs design and has the flexibility, like other FPGAs based design algorithm, of being reconfigurable, making this unit more practical for arithmetic intensive research. The unit is optimized for synthesis. Many applications depend on arithmetic operations. FPGAs based robust arithmetic coprocessor are an alternative solution to the costly and time consuming ASIC's designs.

The most widely used instructions in microprocessors, digital signal processors and graphics accelerators are the Arithmetic operations. The demand for fast computation goes beyond fast addition and multiplication, to include high performance fast divide units and the other elementary functions. In many numerically intensive applications, addition is the most frequent arithmetic operation, followed by multiplication. Division and the other elementary function are the least frequent operations. This is why division and other elementary function did not receive much attention for a long time. But as more

numerically intense applications are being used, the need for fast dividers and other elementary function units become immanent. The time as well as area of such arithmetic operation circuits is very crucial. The top Level Design Conventional FMA unit is shown in figure 1.

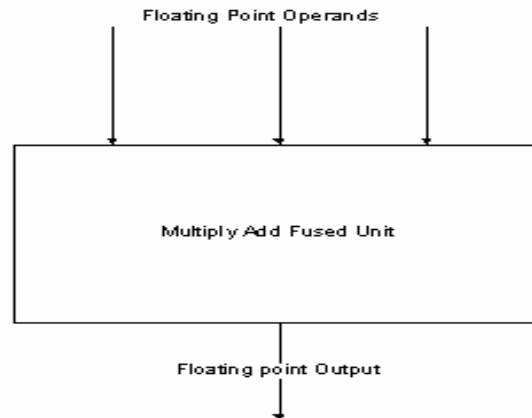


Figure 1: Top Level Design of Fused Multiplication/Addition Unit.

The time of arithmetic operations in numerically intensive applications can be improved with an Application Specific Integrated Circuit (ASIC), designed specifically for a particular application. The development cost and time length of fabrication make such ASIC circuits impractical. An alternative can be Field Programmable Gate Arrays (FPGAs) based arithmetic units. According to the applications, a designer or researcher can select from among various algorithms for addition, multiplication, division and elementary functions to form an arithmetic unit suited for that particular application. FPGAs programmability delays make it slower than ASIC designs, yet faster than software implementations.

With FPGAs based arithmetic units, a designer can mix radices for the different algorithms. The designer may even choose different number systems, such as the Logarithmic number system or even mix pipelined with non-pipelined operation [1-5].

In comparison with software simulation, the hardware realization proposed in this paper has the following advantages: a) the chip, once is realized, is stand-alone electronic circuit that does not need a host computer for operation, b) the chip is compact (1"x1"x.1" without socket and 1"x1"x.7" with socket), c) the chip has an average cost of less than \$30, d) the chip can be easily interfaced to: external signals or stimuli, external devices, or another chip through the input/output pins, e) the downloaded design can be easily modified by just changing the HDL-program and re-downloading, f) the chip, because it is dedicated hardware without a host computer, can operate in real-time applications where a higher speed of operation is needed [6].

Through the realization of arithmetic operations on FPGAs, it is the ambition of this research that researchers could someday be able to pick from off the shelf arithmetic operations and realize them on FPGAs chips. The FPGAs chip can then be arithmetic co-processors for researchers according to their specific needs. Many arithmetic operations have been realized on FPGAs chips. One such research is the hardware realization of krawtchouk Transform using VHDL modeling and FPGAs [7].

2. Floating Point Multiplication

Floating point multiplication does not require significands to be aligned. Assume two operands x and y single precision floating point where $x = S \times 2^E$ and $y = T \times 2^F$, then the multiplication will be:

$$x \times y = (S \times T) \times 2^{E+F} \quad (1)$$

From (1) it is seen that three steps are needed to accomplish floating point multiplication:

- 1- Multiply the significands.
- 2- Add the exponents
- 3- Normalize and correctly round the result.

Two basic operations are involved in multiplication: one is the generation of partial products and second is their accumulation. This means that there are two ways to speed up multiplication either by reducing the number of partial products or accelerating the accumulation. Multipliers are classified into three main classes: Parallel Multipliers, Sequential Multipliers and Array Multipliers. Parallel multipliers generate all partial remainders in parallel and they use a fast multi operand adder for the accumulation. Sequential multipliers sequentially generate the partial products and add each new partial product to the previous accumulated partial product. Array multipliers are made up of an array of identical cells that generate new partial products and accumulate them simultaneously [1-5].

The reduction of the number of partial products which also implies complexity reduction can be done examining two or more bits of the multiplier at a time. This would require generating the first three multiples of the multiplicand. Even though this reduced the number of partial products to $\frac{n}{2}$, yet each step becomes more complex than the previous. There are many algorithms proposed for reducing the number of partial products without increasing the complexity. One such scheme is Booth's algorithm, which along with all schemes proposed for partial product reduction have the same basic idea of generating fewer partial products for groups of 0's and 1's. For a group of consecutive 0's there is no need to generate any new partial products, only shifting the previous partial product one bit to the right. The booth algorithm involves recoding the multiplier in signed digit code. For the n -bit multiplier $x_{n-1}x_{n-2}\dots\dots\dots x_1x_0$ each two adjacent bits are examined to produce the output. The recoding scheme is shown in table 1. Where each bit of the multiplier $x^{(k+1)}$ along with the previous $x^{(k)}$ bit are examined to produce the $(k+1)^{th}$ bit $y^{(k+1)}$ of the recoded multiplier $y_{n-1}y_{n-2}\dots\dots y_1y_0$.

Table 1- Booth's Algorithm

$x^{(k+1)}$	$x^{(k)}$	Operation	Interpretation	$y^{(k+1)}$
0	0	Shift	String of 0's	0
1	1	Shift	String of 1's	0
1	0	Subtract and shift	Start of string of 0'1	$\bar{1}$
0	1	Add and shift	End of string of 1's	1

Notice that $y^{(k+1)} = x^{(k)} - x^{(k+1)}$. Booth's Algorithms can also handle 2's complement multipliers properly. This algorithm which is also called radix 2 booth's algorithm has the disadvantage of having variable add/subtract and shift operation and also is inefficient for isolated 1's [4].

An alternative would be radix 4 booth's algorithm in which three bits of X are examined. The bits $x^{(k+1)}$ and $x^{(k)}$ are recoded into $y^{(k+1)}$ and $y^{(k)}$ while $x^{(k-1)}$ is used as a reference bit. The recoding rules for radix 4 booth are shown in table 2. The algorithm can be modified even further to be of radix 2^k but becomes more complex [4-5].

Another way to implement multipliers is to use smaller multipliers. Once partial products are obtained they should be accumulated. A fast multi operand adder should be employed when high speed accumulation of the partial products is desired [4-5]. The reduction of partial product is sometimes done by the use of an internal redundant representation of the numbers. Even when the original operands are not redundant, they can be converted to redundant form. For example by summing three partial products to give a sum vector and carry vector, this becomes a redundant representation of the sum. This process reduces three product vectors to two vectors; carry and sum. This is done using a (3, 2) counter which is simply a full adder. Alternatively, a [4:2] compressor may be used. The compressor has a reduction ratio of two to one, since it adds four input bits plus a carry bit to produce two outputs and carry [4-5].

Table 2 - Radix 4 modified booth's algorithm.

$x^{(k+1)}$	$x^{(k)}$	$x^{(k-1)}$	$y^{(k+1)}$	$y^{(k)}$	Operation
0	0	0	0	0	+0
0	1	0	0	1	+A
1	0	0	$\bar{1}$	0	-2A
1	1	0	0	$\bar{1}$	-A
0	0	1	0	1	+A
0	1	1	1	0	+2A
1	0	1	0	$\bar{1}$	-A
1	1	1	0	0	+0

Daumas and Matula proposed a booth multiplier accepting both a redundant and non redundant input with no additional delay [8]. Horima et. al proposed an improved design for parallel multiplier based on phase mode logic [9]. In this study, the use of booth encoder as a substitute to an AND array was proposed. Mokrian et al, provide an in depth look at the [4:2] compressor for partial production reduction [10]. Wei, Chen, and Shimizu proposed a fast modular multiplication using Booth recoding based on signed digit number arithmetic [11]. In their algorithm, they use the booth recoding to reduce the number of partial products and give pipeline architecture with two signed digit adders to realize a fast modular multiplication.

3. Floating Point Addition/Subtraction

The addition/subtraction of two operands is the most frequent operation in almost any arithmetic unit. Assume two operands x and y single precision floating point where $x = S \times 2^E$ and $y = T \times 2^F$, then the addition will be:

$$(S + T) \times 2^E \quad \text{if } E = F \tag{2}$$

The result must be normalized if $(S + T) \geq 2$ or $(S + T) < 1$. Where as if $E \neq F$, (e.g., $E > F$), the significands must be aligned, shifting T right $E - F$ positions so that second number is no longer normalized and both number have the same exponent E . The significands are then added as before.

Fahmy and Flynn proposed a case for a redundant format in floating point arithmetic [12]. In this algorithm the proposed redundant number system enables carry free arithmetic operations to improve performance. Beaumont-Smith, Burgess, Lefrere, and Lim proposed reduced latency IEEE floating point standard adder architectures [13]. Their architecture uses flagged prefix addition to merge rounding with the significand addition. Kornerup provides a review of 4-to-2 adders for multi-operand additions [14]. In this research, the author analyzes the use of various 3- and 4-element redundant digit set for radix 2, and compare their adder implementations using various encoding of the digits and carries.

4. Modeling of the FMA Unit

The main object of this paper is to model and realize the FMA unit onto FPGAs Virtex 2 Pro chip. VHDL modeling was implemented to simulate the mechanism. Initially developed for the U.S Department of Defense (DOD) in the 1980's, VHDL has since been standardized and widely used in industry, research, and academia. VHDL is a very powerful tool that can be used to model digital systems at any level of abstraction.

Field Programmable Gate Arrays (FPGAs) are a new technology that provides users programmability in the field. They contain arrays of Configurable Logic Blocks (CLBs) that can be programmed to realize different designs. FPGA families differ by their chip level and intra- and interchip wiring organizations. Because of the short turnaround time and low cost, there is increased interest in system prototyping using FPGAs [6].

4.1 Fused Multiply Add Unit (FMA).

A key and fundamental operation in any DSP processor and many other engineering and scientific applications is the fused Multiply Add Unit. Due to the parallel operations that maybe performed during multiplication and addition in the FMA unit, the overall latency of the operation is reduced. In the past, only fixed point implementations of this unit were included in DSP processors. The reason is the parallel operations in floating point are much more complex. But recently, Floating point FMA's have been integrated into many commercial processors and have become an interesting area of research. The Fused Multiply Add unit is a unit that performs $A \pm B * C$. It can actually perform multiplication by setting $A = 0$, and it can perform addition either between A and $B * C$, or between A and B or C . The subtraction is actually $(-A + B * C)$. There are advantages in doing these operations, one being that there will be only one rounding instead of two, and the second is the decreased hardware and delay because of resource sharing. The one disadvantage of such a unit is that both multiplication and addition can not be performed concurrently.

A Conventional FMA similar to the one mentioned in [15-16] is shown and modeled here. A major reason for implementing multiply-add is that in a typical array multiplier carry propagation exists only for the last addition, {the rest are carry save}. If the result is to be consequently added to another number, this adder can receive a redundant form of the multipliers result {two numbers one from the carry's and one from the sum of the last addition} and a carry propagation is saved, of course an additional carry save operation which is much faster is still needed.

4.1.1 Conventional FMA Unit

The Fused Multiply Add unit is a unit that performs $A \pm B * C$ as one operation. On the other hand, in multiply add non-fused units, two rounding operations are needed, one for multiplication and the other for accumulation. With fused units this rounding is done in one step near the end of the operation. The unit is designed for input operands in IEEE single-precision floating point representation shown in figure 2:

$$(-1)^{sign} \times 2^{e+127} \times (1.f)$$

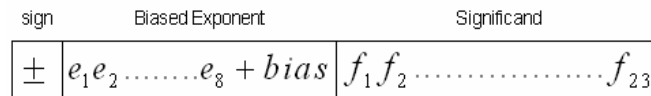


Figure 2: IEEE single formats.

Where the explicit 1 is always f_1 , $significand + 1 = Sps$, and $1 \leq Sps < 2$, significand is the fractional part of the normalized number. Since a top down design strategy is followed, any FMA unit is identified to have the following phases as shown in Figure 3. Note that these phases apply to any FMA unit, with differences in implementation details to enhance the latency of these units.

Figure 4 shows a Conventional FMA unit. The phases are shown with the components that implement each operation. First, start with phase 1, which consists of an array multiplier, an inverter, alignment shifter, and 3:2 Carry Save adders (CSA) unit. Since the design is done for IEEE single precision, the input operands are 24 bits with 8 bit exponents. The mantissas of operand (B) and (C) are multiplied. The multiplier utilizes two basic techniques, the booth encoding, and Wallace tree. The output of the multiplier will be in Carry Save Adder format and consisting of 48 bits for sum and 48 bits for carry. In parallel with multiplication, bit inversion and alignment of mantissa (A) are executed. The alignment is implemented as a right shift by correctly placing (A) to the left of the most significant bit (MSB) of (B*C). Two extra bits are placed between (A) and (B*C) to permit correct rounding when (A) is not shifted. The number of shifts to be done is calculated as follows:

$$27 - d \quad \text{where } d = \exp_A - \exp_{(B*C)} \tag{3}$$

$$\exp_{(B*C)} = \exp_B + \exp_C - 127 \tag{4}$$

In order to reduce latency, in parallel with the multiplication, the operand (A) is inverted in case of a subtraction. It is then aligned according to the number of shifts obtained in (3). Figure 5 shows the operands prior and after shifts.

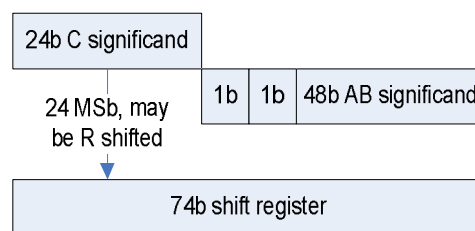


Figure 5: alignment shifter

The aligned (A) is added together with the intermediate product (B*C). Only the least significant 48 bits of aligned A will be added to the sum and carry outputs of the multiplier. The most significant 26 bits of aligned (A) will be either concatenated at the output of the Carry Save Adder or before the Carry Propagate Adder (CPA) in phase two. The shifted-in bits will be either a 0 if number is not inverted in previous step or a 1 if number is inverted. This is done to preserve sign extension. The Carry Save Adder is just a simple thing to understand, it is just a 3:2 adder with three inputs and two outputs (sum and carry). For adding 48 bit operands, 48 CSA adders are needed.

In phase two of the Conventional FMA unit, a 74 bit adder that adds the outputs of the CSA along with the 24 MSB of the aligned (A) is used. This is usually a Carry Propagate Adder (CPA). In parallel with this addition, the Leading Zero Anticipator (LZA) unit will be working to anticipate the number of zero's in the final result before normalization. The LZA unit used is equivalent to a NOR of sum bits and carry bits, which checks for the leading zeros, Then equivalently, XOR the Bits to check if a "0,1" or "1,0" "Sum, Carry" is encountered. Finally, we AND the bits to see if we encounter "1,1". Lets assume the NOR output is (P), the XOR output (Ne), and the AND output is called N. So, if our LZA encounters a (P) at the beginning it will start counting and incrementing the LZA counter each time it finds a (P). Then, if it finds a (Ne,) it will stop and if it finds a (N) it will stop and decrement 1. This LZA unit determines how many left shifts are needed to normalize the result to a floating point representation.

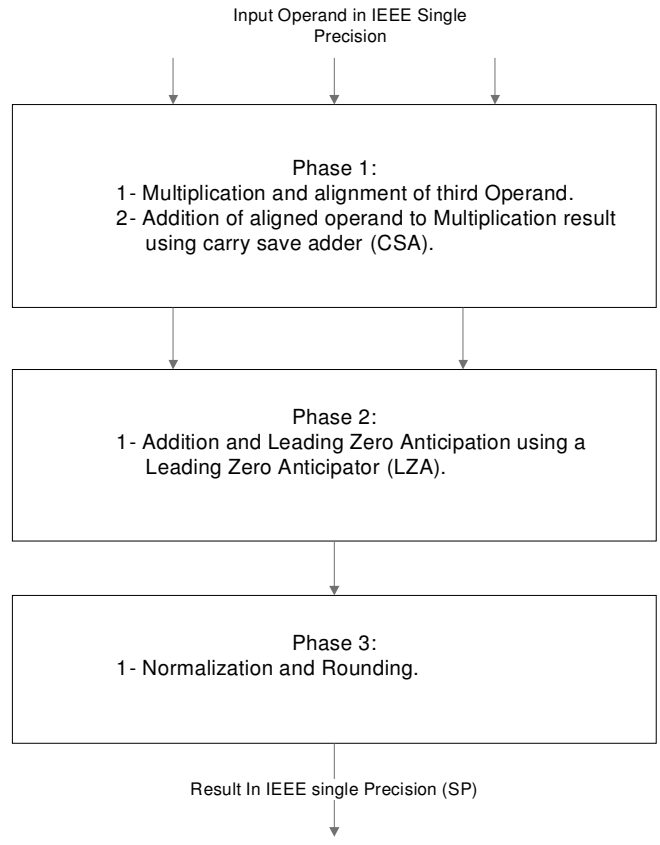


Figure 3: General FMA Implementation Steps

In the third Phase, a complimenter consisting of inverters as an input complements the result obtained from CPA. It will invert the result only if result is negative. The normalization unit takes the number of shifts from the LZA and shift the result by the number provided by the LZA. The sticky bit is calculated, and basically the sticky bit will be "1" only if one of the LSB to the right of the LSB of the result is "1", and is implemented using OR gates. The sticky along with the normalization are used for the rounding step, which usually consists of adding 1ulp and updates exponent if a one is propagated to the MSB of the normalized result.

It has been identified that the most time consuming operation in the FMA unit is the slow 74 CPA. The CPA adder exceeds three times the word length of each operand. The complimenter that follows the

CPA is of equal word length to the CPA. To reduce the latency of the conventional FMA unit, concentrate on the delay of the 74 bit CPA and Complimenter.

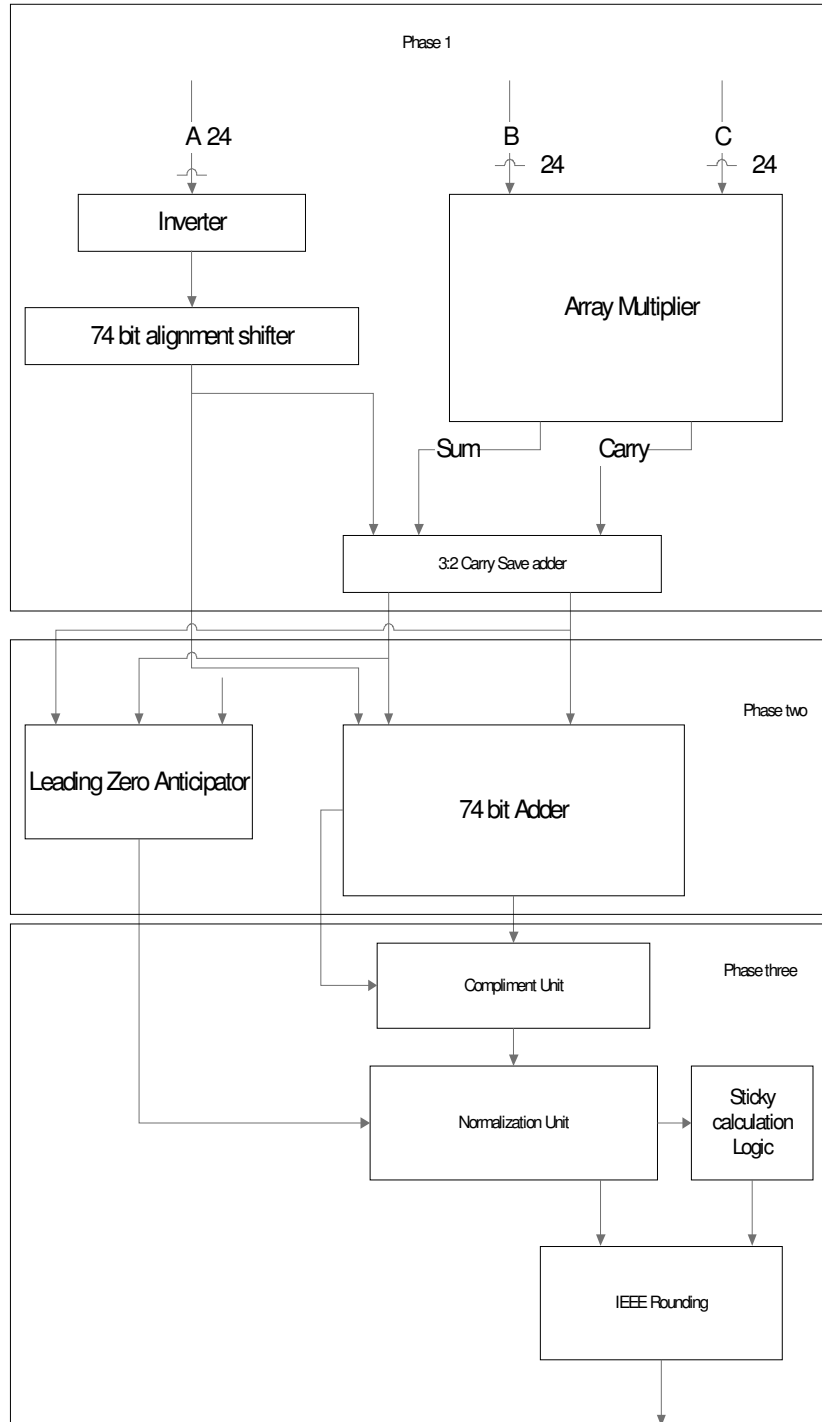


Figure 4: Conventional FMA unit.

5. Conventional FMA Results

The Conventional FMA unit is implemented using Hardware Descriptive Language (HDL). It is implemented in both behavioral and structural modeling. The model is created on Xilinx ISE 6.2i software package. The model is tested thoroughly and proven operational. The following diagrams show the simulation results of the Conventional FMA for various inputs. The operation performed is $A + B * C$. The figures are named according to their inputs and results are shown in figures.

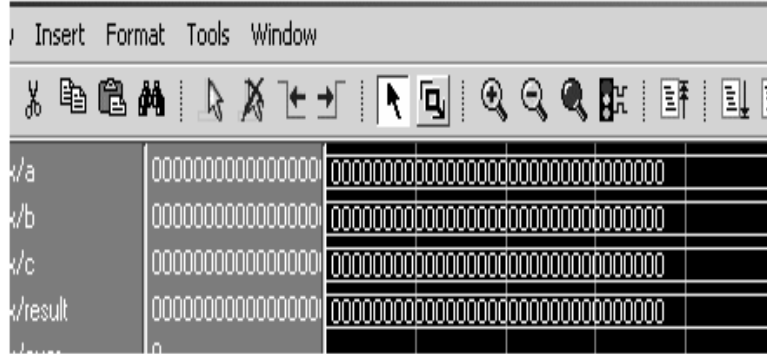


Figure 6: FMA (A=0, B=0, C=0)

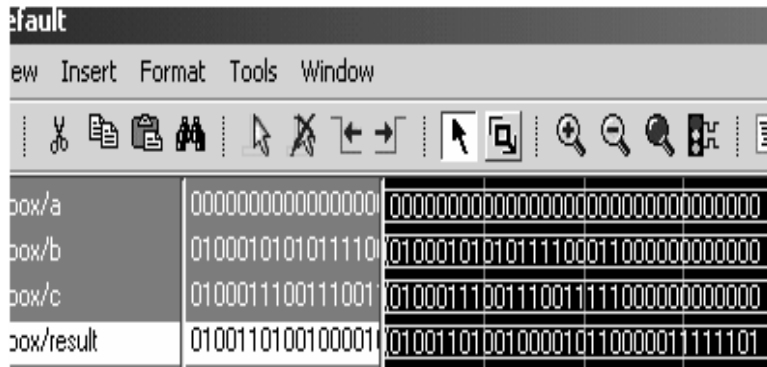


Figure 7: FMA (A=0, B=3555, C=47600)

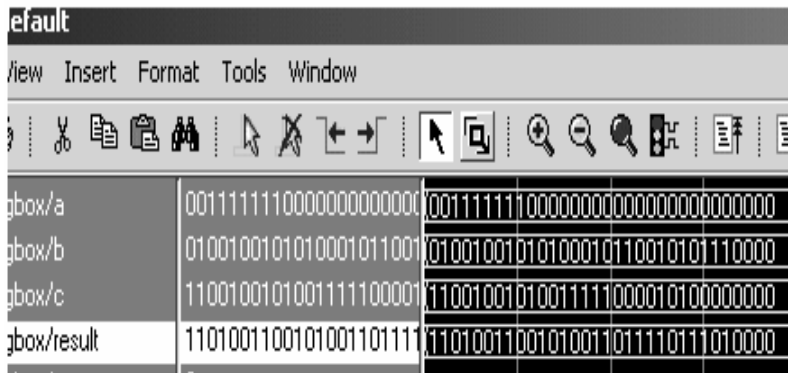


Figure 8: FMA (A=1, B=857687, C=-850000)

default		
View Insert Format Tools Window		
gbox/a	0010111011011011	001011101101101111001101111111
gbox/b	0100100101010001	01001001010100010110010101110000
gbox/c	1100100101001111	11001001010011110000101000000000
gbox/result	1101001100101001	11010011001010011011110111010000
gbox/over	0	

Figure 9: FMA (A=1E (-10), B=857687, C=-850000)

default		
View Insert Format Tools Window		
gbox/a	0100011101010010	01000111010100101110000000000000
gbox/b	0000000000000000	00000000000000000000000000000000
gbox/c	0100011100111001	01000111001110011110000000000000
gbox/result	0100011101010010	01000111010100101110000000000000

Figure 10: FMA (A=54000, B=0, C=47600)

default		
View Insert Format Tools Window		
gbox/a	0100011101010010	01000111010100101110000000000000
gbox/b	0100010101011110	01000101010111100011000000000000
gbox/c	1100011100111001	11000111001110011110000000000000
gbox/result	1100110100100001	11001101001000010101001111001110
gbox/over	0	

Figure 11: FMA (A=54000, B=3555, C=-47600)

default			
View Insert Format Tools Window			
gbox/a	1100011010111000	11000110101110000001111000000000	
gbox/b	1100010110001111	11000101100011111011100000000000	
gbox/c	1011101100011011	10111011000110110101001000000000	
gbox/result	1100011010111000	1100011010111000000100000110100	

Figure 12: FMA (A= -23567, B=-4599, C= -0.00237)

Figures 6 through 12 show that our Conventional FMA is a working unit that can compute $A+B*C$ for floating point single precision numbers. The FMA Conventional unit is synthesized using Xilinx ISE 6.2i. Some of the generated schematic for the Conventional FMA unit are shown below. After synthesis, the device utilization summary is extracted from the synthesis report and is as follows:

- Number of Slices: 16177 out of 46592 34%
- Number of Slice Flip Flops: 10 out of 93184 0%
- Number of 4 input LUTs: 31099 out of 93184 33%
- Number of bonded IOBs: 131 out of 824 15%

Where LUTs are look-up tables and IOBs are input and output buffers. These are technologies specific to the ASIC design on FPGAs.

The critical path of the Conventional FMA unit is:

$$t_{basic} = t_{muxmult} + t_{csa} + t_{106-qa} + t_{compl} + t_{normalizer} + t_{round} \dots\dots\dots(5)$$

The timing report indicates that the minimum period for the Conventional FMA is 112.917ns with a maximum frequency of 8.856MHZ.

Parts of the synthesized FMA unit are shown in fig.13 and fig. 14. Fig. 13 shows part of the synthesized LZA. Fig. 14 shows part of the synthesized Multiplier.



Figure 13: Part of Synthesized LZA.

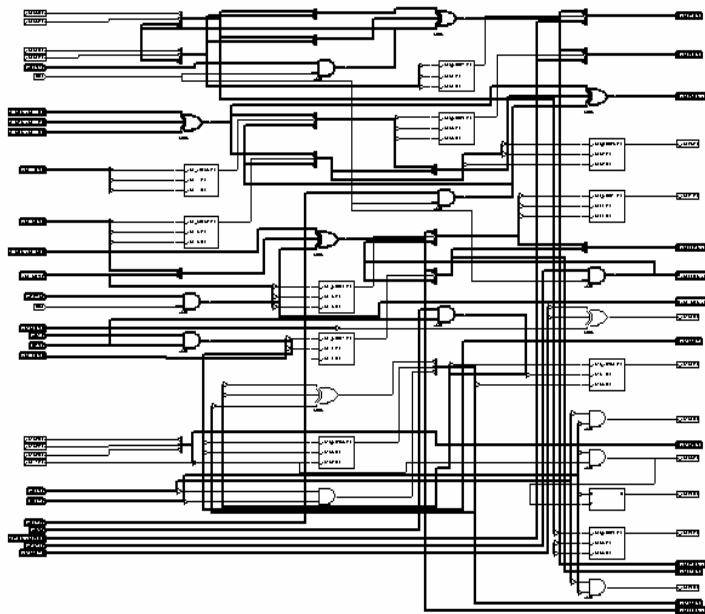


Figure 14: Part of Synthesized Multiplier.

6. The FMA Unit, Optimization and Modification

The Verilog Hardware Description Language (HDL) is used to model and synthesize the FMA design using Xilinx design tools and targeting a Xilinx Virtex 2 Pro FPGA device with a speed grade of -6. While a full-custom ASIC implementation would result in higher performance, the process is faster and the resulting design more adaptable when done using Verilog and programmable hardware devices such as FPGAs. Modern FPGAs now boast acceptable performance metrics for many of today's applications and are cheaper to manufacture for low production volume designs when compared with ASICs. [17] [22]

Some specific implementation methods used during the Verilog modeling of the FMA unit will now be covered. This unit has no instruction input. Despite the fact that it performs both addition and subtraction along with multiplication; the op-code is assumed to be implicitly included in the 'C sign' input. That is, lets assume that the sign of C is adjusted according to its original sign and the operation specified (add or subtract) before it reaches the input of the FMA, and the FMA circuit itself performs only $AB+C$. All other operations are supported by the sign and inversion logic; the 'alignment shift & invert' and 'sign logic' blocks of stage one as well as the 'invert' and 'sign logic' blocks in stage two.

In addition to sign information, the 'alignment shift & invert' block also receives input from the 'alignment shift calculation' block in stage one. This block performs the addition of the A and B exponents, compares the resulting AB exponent to C, and generates the required shift amount of the C significand, if any. The formula used to generate the required shift amount is: $27 - (C_{exp} - AB_{exp})$. [17] The 'shift & invert' block itself is also responsible for aligning the C significand for addition; the 24-bit significand (the implied leading 1 is concatenated) is inserted into the most significant (MS) bits of a 74-bit signal and right-shifted according to the shift signal explained above. The 74-bit width of the signal is required for proper alignment and insertion of the guard and round bits, as shown in Fig. 15, and is equal to the 24 bits of the C significand, plus the 2 extra bits, plus the 48 bits of the AB significand. The stage one multiplier and CSA were designed using 4:2 and 3:2 compressors respectively, the designs of which can be found in any digital arithmetic book and were used to provide fast, propagation free addition and multiplication in order to reduce the overall latency [5]. Booth recoding is not used, so

the 4:2 compressors of the multiplier were arranged in a tree design to implement the partial products and their accumulation into the sum and carry signals.

The standard 75-bit adder in stage two operates in parallel with the Leading Zero Anticipation and Detection (LZA & LZD) units to complete the addition, adding the sum and carry signals from the 3:2 CSA with the remaining 26 MSBs of the aligned C significand. The LZA is designed using the following algorithms presented by Lee and Nowka. It receives the same three signals as the adder and generates a symbolic bit pattern (one for the positive and one for the negative) indicating what bit position contains the arithmetically significant leading digit in the result of the addition. The bit strings are computed for each bit using the logic shown in figure 15 by examining the bit position in question and its less significant neighboring bit (to the right) [18] [20].

$$\begin{array}{l} \text{POS_L}_i = p_i \text{ <xnor> } z_{i+1} \quad \text{NEG_L}_i = p_i \text{ <xnor> } g_{i+1} \\ p_i = A_i \text{ <xor> } B_i \quad z_i = \sim A_i \text{ <and> } \sim B_i \quad g_i = A_i \text{ <and> } B_i \end{array}$$

Figure 15: LZA logic

The LZA and LZD blocks consider bit 0 to be the MS bit (the reverse of conventional numbering where 0 is the LS bit); one reason is this allows the output of the LZD to represent both the bit position and the shift amount. In the above equations L_i is the i th bit of the symbolic leading digit string, currently being generated; the $i+1$ bit is then the bit to the right, or below the current bit in significance (due to the backwards numbering of the units). A_i and B_i are the corresponding bits in the two inputs (the addends); pairs of bits from these inputs are used to generate the propagate (p_i), to generate (g_i), and to kill (z_i) signals of the addition. These signals are then used to generate the POS (positive case) and NEG (negative case) bit strings. The LZA is inexact and may require one additional shift during post-normalization. [18] [20]

The POS and NEG bit patterns pass to the LZD for detection of the leading bit position and generation of the shift amount signals (one for each case). The LZD is designed following an algorithmic and hierarchical method presented by Oklobdzija. This method generates a position (P) and valid (V) bit for each adjacent pair of bits in the LZA string and then combines these P and V bits in a tree structure to produce a final position and valid signal for the entire bit-width of the input (Figure 16). [19] [21]

Two LZDs are implemented in the design to run in parallel on the POS and NEG LZA strings. The correct shift amount is then selected by a 2:1 multiplexer using the sign bit resulting from the completed addition, with minimal delay. This shift amount is passed to stage three 'normalization shift' block, which performs the left-shifting of the $AB+C$ significand. The shifted amount is also input to the 'exponent adjust' block; every left-shift increases the value of the significand by a factor of 2 and the exponent must also be reduced by 1 in order to maintain the value of the FMA result.

Rounding is accomplished using the IEEE-754 standard round to the nearest even algorithm. After rounding the final adjustments to the significand and exponent are made before they are sent to the output; any remaining unnecessary leading digits are shifted out and the concurrent exponent adjustment is made.

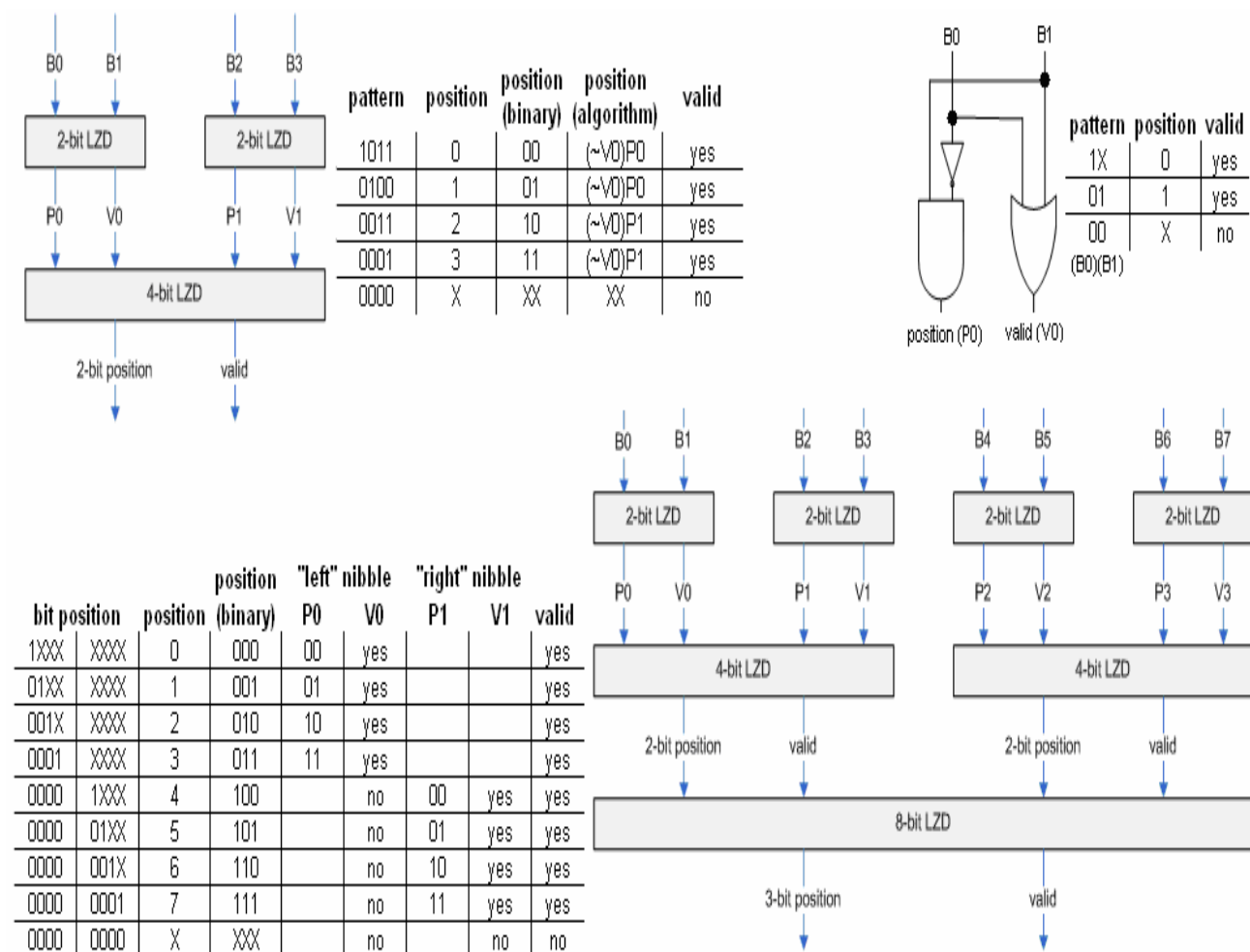


Figure 16: LZD algorithm; 2-, 4-, and 8-bit implementations

6.1 Modifications

Stage three is the stage with the most latency during simulation, so it is targeted for modification in an attempt to clock the FMA at 100 MHz or more. The normalization – rounding – post-normalization operation chain is clearly the largest delay path in the stage. The post-normalization operation is removed by duplicating the normalization and rounding logic for each post-normalization case (LZD shift output is accurate, or one additional shift required), reducing a possible shift operation at the end of the chain to a simple 2:1 multiplexer. The increase in circuit area is rewarded by a ~14 MHz speed increase.

6.2 Results of the Optimized FMA unit

After making the optimizations described in the modifications section, the optimized circuit synthesis results are shown in Fig. 17. Overall utilization of the FPGA remains low.

<u>logic utilization</u>	<u>used</u>	<u>available</u>	<u>utilization</u>
slices	1380	13696	10%
slice flip flops	530	27392	1%
4-input LUTs	2511	27392	9%
bonded IOBs	129	556	23%
GCLKs	1	16	6%
minimum period		9.236 ns	
maximum frequency		108.267 MHz	

Figure 17: optimized FMA synthesis results

7. Conclusion

In this paper, the effect of optimization on the modeling and realization of arithmetic operations specifically FMA unit using VHDL, Verilog and FPGAs is presented. Using this method, developing optimized arithmetic algorithms from already existing methods on FPGAs is a main objective. This will give researchers, whose research applications are arithmetically extensive, the freedom to use off the shelf algorithms that meet their needs. These algorithms can then be downloaded onto FPGA chips that may act as mathematical Co-processors. Although the system here is just the conventional FMA, this will be the base for developing and modeling new and existing algorithms onto FPGAs.

8. FUTURE WORK

Future work in this field will include modeling and realization; and optimization of modified FMA units, Division, Square Root, and Reciprocal Square Roots units. The author is already in the process of developing and optimizing such units.

9. REFERENCES

- [1] J. L. Hennessy and D.A Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, 2 ed., 1996.
- [2] J. L. Hennessy and D.A Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, 3 ed., 2003.
- [3] M. J. Flynn and S. F. Oberman, *Advanced Computer Arithmetic Design*, John Wiley & Sons, Inc, 2001.

- [4] I. Koren, *Computer Arithmetic Algorithms*, A K Peters, 2 ed, 2002.
- [5] B. Parhami, *Computer Arithmetic Algorithms and Hardware Designs*, Oxford University Press, 2000.
- [6] N. Botros, A. Akaaboune, and J. Alghazo, "Modeling and Synthesis of Human Growth Hormone Secretion Mechanism Using CAD tools", *Proc. 29th annual conf. of the IEEE Industrial Electronics Society*, Vol. 3, Nov. 2003, pp. 2429-2434.
- [7] N. Botros, J. Yang, P. Feinsilver, and R. Schott, "Hardware Realization of Krawtchouk Transform Using VHDL Modeling and FPGAs", *IEEE Trans. Ind. Electron.*, vol. 49, no. 6, pp. 1306-1312, 2002.
- [8] M. Daumas and D. W. Matula, "A Booth multiplier accepting both a redundant or a non redundant input with no additional delay," *IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pp. 205-214, 10-12 July 2000.
- [9] Y. Horima, T Onomi, M. Kobori, I. Shimizu, and K. Nakajima, "Improved Design for Parallel Multiplier Based on Phase-Mode Logic," *IEEE Transaction on Applied Superconductivity*, vol. 13, no. 2, 2003.
- [10] P. Mokrian, G. M. Howard, G. Jullien, M. Ahmadi, "On the use of 4:2 Compressors for partial Product Reduction," *IEEE Canadian Conference on Electrical and Computer Engineering*, vol. 1, pp. 121-124, May 4-7, 2003.
- [11] S. Wei, S. Chen, and K. Shimizu, "Fast modular multiplication using Booth recoding based on signed-digit number arithmetic," *Asia-Pacific Conference on Circuits and Systems*, Vol. 2, pp. 31-36, 28-31 Oct. 2002.
- [12] H. Fahmy and M. Flynn, "The Case for a Redundant Format in Floating Point Arithmetic," *Proceedings of the 16th IEEE Symposium on Computer Arithmetic* Santiago de Compostela, Spain, June 2003.
- [13] A. Beaumont-Smith, N. Burgess, S. Lefrere, and C.C. Lim, "Reduced Latency IEEE Floating Point Standard Adder Architectures," *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, 1999.
- [14] P. Kornerup, "Reviewing 4-to-2 Adders for Multi-Operand Addition," *Proceedings of the IEEE International Conference on application-Specific systems, architecture, and Processors*, 2002.
- [15] T. Lang and J. Bruguera, "Floating-Point Fused Multiply-Add with Reduced Latency," *Proceedings of the 2002 IEEE International Conference on Computer Design*, 2002.
- [16] C. Chen, L. A. Chen, and J. R. Cheng, "Architectural design of a fast Floating-point Multiplication-Add fused unit using signed-digit addition," *IEE Proc. -Comput. Digit. Tech.*, vol.149, no. 4, 2002.
- [17] J. Alghazo, "Modeling and Synthesis of a Conventional Multiply-Add Fused Floating Point Arithmetic Using CAD Tools," *14th IWLS*, June 2005.
- [18] K.T. Lee, K.J. Nowka, "1 GHz Leading Zero Anticipator Using Independent Sign-Bit Determination Logic," *IEEE 2000 Symp. on VLSI Circuits, Digest of Tech. Papers*, June 2000, Pg.7-12
- [19] V.G. Oklobdzija, "An Implementation Algorithm and Design of a Novel Leading Zero Detector Circuit," *Record of the 26th Asilomar Conf. on Signals, Systems and Comp.*, Vol.1, Oct.1992, Pg.391-395

[20] M.S. Schmookler, K.J. Nowka, "Leading Zero Anticipation and Detection – A Comparison of Methods," Proc. of the 15th IEEE Symp. on Comp. Arithmetic, June 2001, Pg.7-12

[21] V.G. Oklobdzija, "An Algorithmic and Novel Design of a Leading Zero Detector Circuit: Comparison with Logic Synthesis," IEEE Trans. on VLSI Systems, Vol.2, Issue 1, March 1994, Pg.124-128

[22] J. Alghazo, N. Botros, "Modeling and Synthesis of a Modified Floating Point Fused Multiply-Add (FMA) Arithmetic Unit Using VHDL and FPGAs," Proc. of the 2005 Intl. Conf. on Comp. Design, June 2005, Pg.136-142