RESEARCH ARTICLE OPEN ACCESS

A Survey on Floating Point Adders

Deepak Mishra [1], Vipul Agrawal [2]
Department of Electronics and Communication Engineering [1] & [2]
Trinity Institute of Technology & Research, Bhopal
India

ABSTRACT
Addition is the most complex operation in a floating-point unit and can cause major delay while requiring a significant area. Over the years, the VLSI community has developed many floating-point adder algorithms aimed primarily at reducing the overall latency. An efficient design of the floating-point adder offers major area and performance improvements for FPGAs. This paper studies the implementation of standard; leading-one predictor (LOP); and far and close data path (2-path) floating-point addition algorithms in FPGAs. Each algorithm has complex sub-operations which contribute significantly to the overall latency of the design. Each of the sub-operations is researched for different implementations According to the results, the standard algorithm is the best implementation with respect to area, but has a large overall latency of 27.059 ns while occupying 541 slices. The LOP algorithm reduces latency by 6.5% at the cost of a 38% increase in area compared to the standard algorithm. The 2-path implementation shows a 19% reduction in latency with an added expense of 88% in area compared to the standard algorithm. The five-stage standard pipeline implementation shows a 6.4% improvement in clock speed compared to the Xilinx IP with a 23% smaller area requirement. The five-stage pipelined LOP implementation shows a 22% improvement in clock speed compared to the Xilinx IP at a cost of 15% more area.

Keywords:- Floating Point Adder, FPGA, Delay, Area Overhead

I. INTRODUCTION

By the early 1980s, custom integrated circuits (ICS) were often designed to replace the large amounts of glue logic in electronic device and thus reduce manufacturing cost and system complexity. However, because custom ICS are expensive to develop, they are viable only for very high-volume products. To address this limitation, xilinx produced the field-programmable gate array (FPGA) technology in 1984 as an alternative to custom ICS. An FPGA is a silicon chip with unconnected logic blocks. These logic blocks can be defined and redefined by the user at any time. FPGAs are increasingly being used for applications which require high numerical stability and accuracy. Given their shorter time to market and low cost, FPGAs are becoming a more attractive solution for low-volume applications compared to application specific ICS (asICS). Modern FPGAs provide on-chip prefabricated arithmetic units. For example, carry-look ahead adders are common. While this development confirms that the need for arithmetic units is real, it also demonstrates the limitation of having fixed resources. Often one needs a different type of arithmetic circuit or perhaps more arithmetic circuits than are provided. In other words, efficient implementations of various arithmetic functions using reconfigurable arrays are always important.

The use of FPGAs in compute-intensive applications has been growing dramatically. Examples of such applications include digital signal processing (dsp), molecular analysis, and image processing. The majority of such applications use fixed-point arithmetic because of its smaller size requirement. However, the dramatic increase in application size has allowed FPGAs to be considered for several scientific applications that require floating-point arithmetic. The advantage of floating-point arithmetic over fixed-point arithmetic is the range of numbers that can be represented with the same number of bits. The results in [1] showed that FPGAs are capable of achieving up to a sixfold improvement in terms of the performance-per-unit power metric over general-purpose processors. The results in [2] also showed that an FPGA-based face detector which takes 1.7 ms to process one frame is 38 times faster than a personal computer. Floating-point addition and subtraction are the most common floating-point operations. Both use a floating-point adder. According to real application data in [3], signal processing algorithms require, on average, 40% multiplication and 60% addition operations. Therefore, floating-point addition is a fundamental component of math coprocessors, dsp processors, embedded arithmetic processors, and data processing units. Floating-point addition is the most complex operation in a floating-point unit and consists of many variable latency- and area dependent sub-operations. In floating-point addition implementations, latency is the primary performance

bottleneck. Much work has been done to improve the overall latency of floating-point adders. Various algorithms and design approaches have been developed by the vlsi community [4]–[7] in the last two decades. For the most part, digital design companies around the globe have focused on FPGA design instead of asICS because of their effective time to market, adaptability, and, most importantly, low cost. The floating-point unit is one of the most important custom applications needed in most hardware designs, as it adds accuracy, robustness to quantization errors, and ease of use. There are many commercial products for floating-point addition [8]–[10] that can be used in custom designs in FPGAs but cannot be modified for specific design qualities like throughput, latency, and area. Much work has also been done to design custom floating-point adders in FPGAs. Most of this work aims to increase the throughput by means of deep pipelining [1], [11]–[15].

II. STANDARD FLOATING POINT ADDER ALGORITHM

The standard architecture is the prototype algorithm for floating-point addition in any kind of hardware and software design [17]. Micro-architecture of the floating-point adder is shown in Fig. 1. The first step, not shown in Fig. 1, is to check whether the inputs are deformalized, infinity, or zero. These numbers are defined by special formats and standards, and VHDL comparators are used to identify them. The results are used to identify exceptions and are common to all algorithms. Next, the exponents are subtracted from one another to compute the absolute difference and identify the larger exponent. The mantissa of the number with the smaller exponent is right-shifted by the exponent difference and extended by 3 bits, to be used later for rounding; then the two mantissas are added using a two-complement adder. The next step is to detect the leading number of zeros before the first 1 in the result; this step is done by the module known as the leading-one detector (LOD). Using this value, the result is left-shifted by means of a left-shifter. When the result is negative and the operation is subtraction, the result is right-shifted by 1. The last 5 bits are used to detect whether rounding is needed, and another adder is used to add a 1, yielding the resulting mantissa. The resulting exponent is computed by subtracting the leading-zeros amount from the larger exponent and adding a 1 when there is a right shift.
The standard floating-point adder consists of five variable-size integer adders and one right-shifter which can extend the result by 3 bits, named the guard (g), round (r), and sticky (s) bits. For post normalization, we need an LOD and a left-shifter. All these modules add significant delay to the overall latency of the adder.

A. Adder

In order to compute the exponent difference, twos-complement addition, rounding, and exponent-result variable-width integer adders are needed. These requirements prompt a major increase in the overall latency of the floating-point adder. Over the years, a tremendous amount of work in VLSI has been done to make the integer adder as fast as possible [18]. A 16-bit carry-lookahead adder, a carry-save adder, and a ripple-carry adder have been designed and synthesized for the Vertex-II Pro FPGA by [24]. Combinational delay and slice information are compared with the Xilinx built-in adder function. Table 1 shows the synthesis results obtained using the Xilinx ISE [24]. Combinational delay is independent of the clock and thus is defined as the total propagation and routing delays of all the gates included in the critical path of the circuit. Each configurable logic block (CLB) consists of four slices in the Vertex-II Pro architecture, and the CLB is used as the basic unit for measuring area in Xilinx FPGAs. Both these design parameters are reported by the Xilinx ISE after the circuit is synthesized, routed, and placed onto an FPGA device. [24]

B. Right-shifter

In order to pre-normalize o pre-normalize the mantissa of the number with the smaller exponent, a right-shifter is used to right-shift the mantissa by the absolute-exponent difference. This is done so that the two numbers will have the same exponent and normal integer addition can be carried out. The right-shifter is one of the most important modules to consider when designing for latency, as it adds significant delay. In order not to lose the precision of the number, three extra bits are added. The sticky bit is obtained by ORing all the bits shifted out. Three custom shifters are designed for this purpose. For a single-precision floating point adder, the 24-bit mantissa acts as input, and the result is 27 bits. A typical barrel shifter was implemented with a 2:1 multiplexer as its fundamental module and was used to shift the number on five different levels. The sticky bit is the OR of all the bits discarded during the alignment shift.

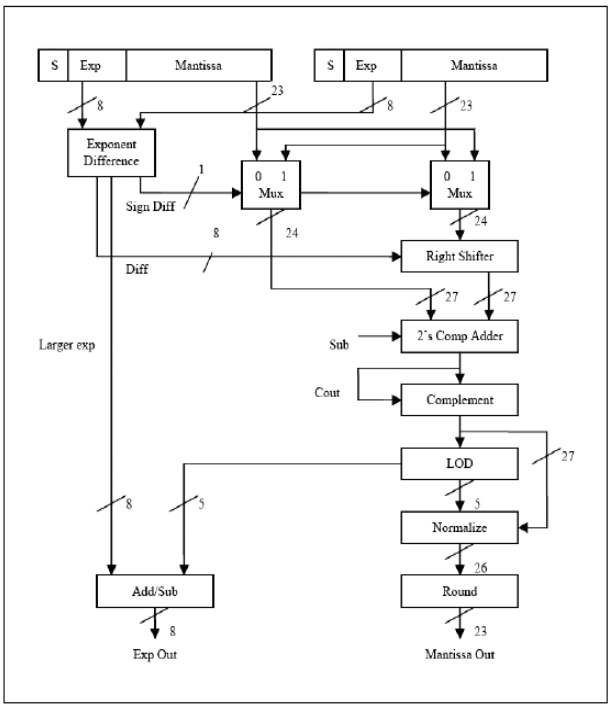

Figure 1: Architecture of Standard floating point adder [24]

Table 1
Adder implementation analysis

Adder type	Combinational delay (ns)	Area (slices)
Ripple-carry	15.910	18
Carry-save	11.951	41
Carry-lookahead	9.720	39
Xilinx	6.018	8

Table 2
Right-shifter implementation analysis

Shifter type	Combinational delay (ns)	Area (slices)
Align	10.482	71
Barrel	9.857	71
Behavioural	9.357	201

Other designs for the right-shifter, named the aligned shifter and the behavioral shifter, were also implemented. In each case, the bits shifted out are ORed separately. Table 2 shows the synthesis results obtained with the Xilinx ISE. [24]
The behavioural implementation is best in terms of latency, but leads to a huge increase in the area requirement because the Xilinx synthesizer is not able to optimize the "case" statements, thus yielding a cumbersome design. The barrel and align shifters require the same area, but the barrel implementation provides a smaller combinational delay and therefore was chosen for our design. The structural and more defined implementation of a barrel-type shifter uses a 2:1 multiplexer as its basic unit, which is easily synthesizable by the function generators present in the slices. The aligned shifter relies on the synthesizer to implement the behaviourally coded large multiplexers and thus offers more propagation delay due to added routing.

Table 3
LOD implementation analysis

LOD type	Combinational delay (ns)	Area (slices)
Behavioural	9.05	20
Oklobdzija	8.32	18

Table 4
Left-shifter implementation analysis

Shifter type	Combinational delay (ns)	Area (slices)
Behavioural	8.467	80
VHDL	8.565	90

C. Leading-one detector

For post-normalization, the leading number of zeros must be detected in the result of the adder. This amount is then used to left-shift the result and normalized to obtain the mantissa before rounding. There are a number of ways to design a complex circuit such as an LOD. A combinational approach is complex because each bit of the result is dependent on all the inputs. This approach leads to large fan-in dependencies, and the resulting design is slow and complicated. Another approach is to use Boolean minimization and Karnaugh maps, but the design is again cumbersome and unorganized. The circuit can also be easily described behaviorally using VHDL, and the rest can be left to Xilinx ISE or any synthesis tool. An LOD can also be designed by identifying common modules; this approach imposes a hierarchy on the design. In comparison to other options, such a design has low fan-in and fan-out, which leads to an area- and delay-efficient design as first presented by Oklobdzija [19]. Behavioral and Oklobdzija-type LODs were implemented, and Table 3 shows the synthesis results obtained with the Xilinx ISE.
Oklobdzija implementation has a better latency-to-area ratio and is chosen over the behavioural model because of its performance and simple hierarchy. The implementation of the behavioural LOD is done entirely by the Xilinx synthesizer, which results in a cumbersome design and adds routing delays. On the other hand, the basic module for implementation described by Oklobdzija is a 2:1 multiplexer, which is implemented by the built-in function generators of the slices in the CLBs of the Vertex-II Pro FPGA. Each connection is defined. Thus minimum routing delay is expected, which results in better propagation delay and area usage compared to the behavioral implementation.

D. Left-shifter

Using the results from the LOD, we left-shift the result from the adder to normalize it. This means that the first bit is now 1. The shifter can be implemented using "sll" in VHDL, or it can be described behaviorally using case statements. Table 4 presents the synthesis results obtained from the Xilinx ISE implemented for the Vertex-II Pro FPGA device. The behavioral model had a slightly smaller combinational delay and smaller area and is therefore used in our implementation. For a single-precision floating-point adder, the maximum required amount of left shift is 27. Therefore hardware for the behavioral left-shifter is designed to accommodate only the maximum shift amount. As we have no control over the hardware implementation in the VHDL shifter, it implements hardware for shift amounts greater than 27, thus yielding bigger area requirements and delay compared to the behavioral shifter. When the carry out from the adder is 1 and the operation is addition, the result is right-shifted by one position.

E. Time and area analysis

Using the above modules, we synthesized a standard floating-point adder for the Vertex-II Pro FPGA. As the design was implemented for only one pipeline stage, the minimum clock period reported by the synthesis tool after placing and routing was 27.059 ns, and the levels of logic reported were 46. This

means that the maximum achievable clock speed for this implementation is 36.95 MHz the number of slices reported by the synthesis tool was 541. All this information is used as a base to analyze improvements in the floating-point adder. [24]

F. Pipelining

Pipelining is used in order to decrease clock period, run operations at a higher clock rate, and boost speedup by increasing the throughput. Pipelining is achieved by distributing hardware into smaller operations, such that the overall operation takes more clock cycles to complete but permits new inputs to be added with every clock cycle to increase the throughput. Pipelining of floating-point adders has been discussed in a number of previous research papers [13]–[14]. Minimum, maximum, and optimum numbers of pipeline stages for a 32-bit floating-point adder have been given, based on frequency per area (MHz/slice). According to these studies, the optimum number of pipeline stages for a single-precision adder implementation is 16. In order to achieve this number, all hardware modules must be sub pipelined within them. In order to analyse the effects of pipelining on floating-point adder implementations in FPGAs, we compare our implementation results with those for the Xilinx IP core by Digital Core Design [8].

Fig. 2 shows the micro-architecture of a five-stage pipelined implementation of the standard floating-point adder algorithm. The number of pipeline levels chosen is based purely on comparison with the Xilinx IP core and is based entirely on design needs. Five is a good choice because more stages would require sub-pipelining of the modules. The placement of the registers is indicated by the dotted lines in Fig. 2. The main reason for pipelining is to decrease the clock period, thus increasing the overall clock speed at which the application can be run. Adding pipeline stages exploits the D flip-flops in the slices already being used for other logic, and thus does not increase the area. Pipelining also helps increase throughput since a result is produced every clock cycle after the first five clock cycles. In the first stage of the implementation, the two operands are compared to identify demoralization and infinity. Then the two exponents are subtracted to obtain the exponent difference and identify whether the operands need to be swapped using the exponent difference sign. In the second stage, the right-shifter is used to pre-normalize the smaller mantissa. In the third stage, addition is done along with the leading-one detection. In the fourth stage, a left-shifter is used to post-normalize the result. In the last stage, the exponent out is calculated, and rounding is done. The results are then compared to set overflow or underflow flags. Table 5 compares results for the five-stage standard-algorithm pipelined implementation with data provided for the Xilinx IP core.

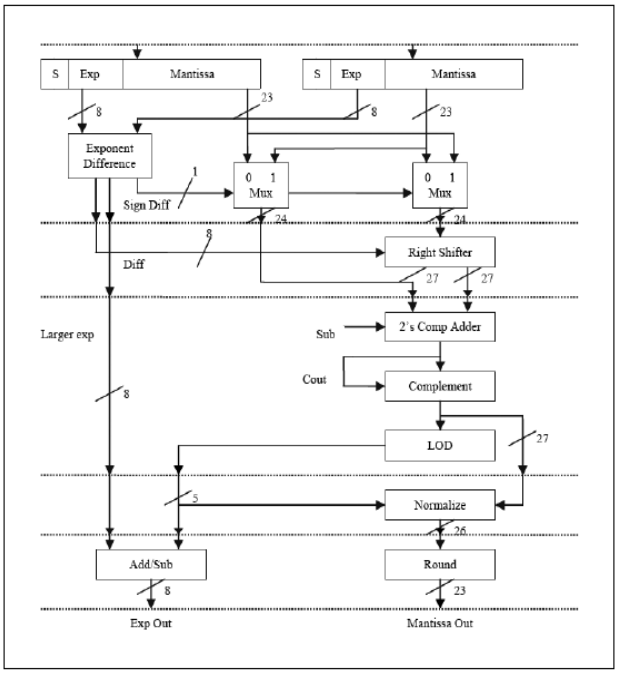

Figure 2: Pipelined architecture of floating point adder

Table 5

Five-stage standard pipeline and Xilinx IP analysis

Module	Clock speed (MHz)	Area (slices)
Xilinx IP [12]	120	510
Five-stage standard pipeline	127	394
Percent improvement	+6.4	+23

The clock speed of the five-stage pipelined standard floating-point adder implementation is 6.4% better than that reported for the Xilinx IP, and the area requirement reported for our implementation is 23% better than that reported for the Xilinx IP. As a result of better slice packing, the area occupied by the five-stage pipelined version of the standard adder implementation is approximately 27% (147 slices) smaller than the area of its non-pipelined version (541 slices). [24]

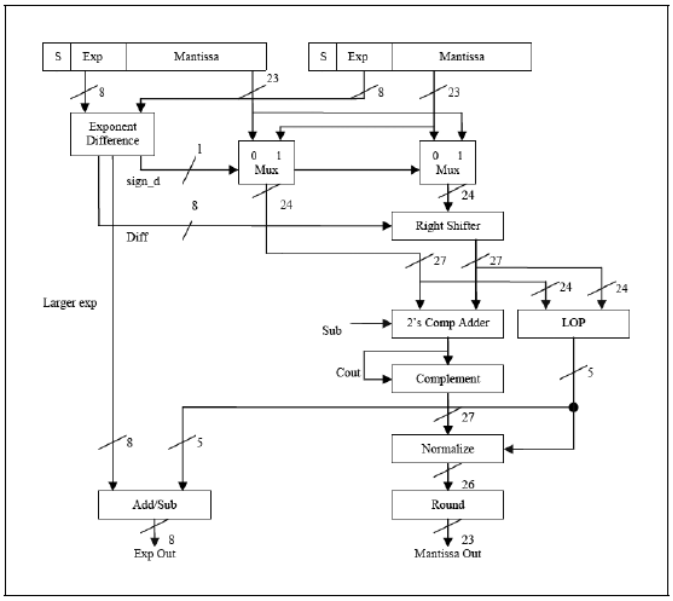

Figure 3: Architecture of LOP algorithm

Table 6

LOP vs. LOD implementation analysis

Module	Combinational delay (ns)	Area (slices)
LOP	13.600	240
LOD	8.320	18
Adder	6.893	14

III. LOP ALGORITHM

In Section II, the different subcomponents used to design a floating point adder were analysed for different architectural approaches to target the best possible implementation of a standard floating-point adder for a Vertex-II Pro FPGA. Over the years, floating-point algorithms have been researched to obtain better overall latency. One of the improvements is the LOP algorithm. Fig. 3 shows the micro-architecture of this algorithm. In this implementation, an LOP is used instead of an LOD. The main function of the module is to predict the leading number of zeros in the addition result, working in parallel with the twos-complement adder. This concept was first introduced by Flynn [20] in 1991. Over the years there have been a number of improvements in its design and application [6]–[7], [21]–[22]. An LOP has three major modules: the pre-encoder, an LOD, and an error-detection tree. The error detection is an important step which detects prediction errors in certain cases. In the VLSI design, the main objective has been to reduce the latency prompted by error detection. The most feasible design, given in [6], detects the error concurrently with the leading-one detection. The design requires a larger area because of added pre-encoding, but offers the best latency because of parallelism and concurrency. The design was implemented on a Vertex-II Pro FPGA, and the results obtained are given in Table 6. In the standard algorithm, an LOD and adder working in parallel have a combinational delay of 15.213 ns, while the LOP offers a delay of 13.6 ns. The addition is done in parallel, but, as seen in Table 6, at the cost of a larger area. Pre-encoding consists of equations based on AND, OR, and NOT gates. It uses the sum-of-product chains in the slices and requires 60% (146 slices) of the overall area of the LOP in FPGAs. [24]

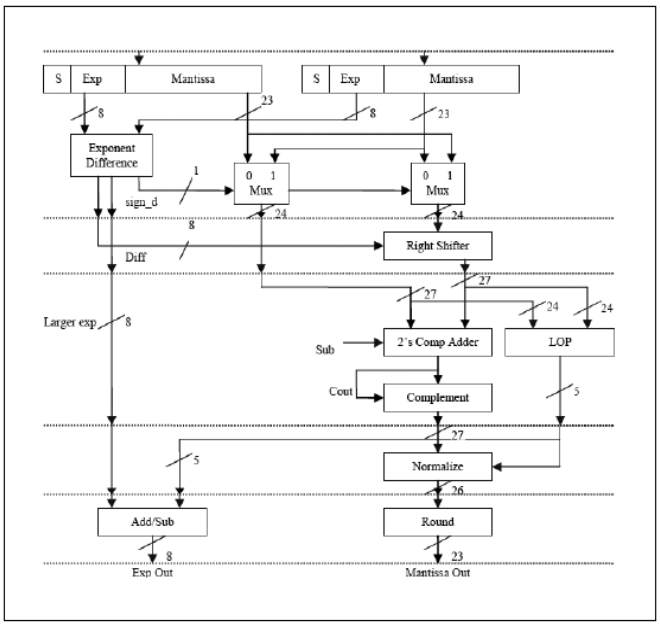

Figure 4: Pipelined Floating point adder

Table 7

Standard and LOP algorithm analysis

Module	Clock period (ns)	Clock speed (MHz)	Area (slices)	Levels of logic
Standard	27.059	36.95	541	46
LOP	25.325	39.48	748	35
Percent improvement	+6.5	+6.5	−38	+23

Table 8

Five-stage LOP pipeline and Xilinx IP analysis

Module	Clock speed (MHz)	Area (slices)
Xilinx IP [12]	120	510
Five-stage LOP pipeline	152	591
Percent improvement	+22	−15

A. Timing and area analysis

Using all the same modules described in Section II, we implemented the LOP algorithm with one cycle latency. The results are summarized and compared with the standard implementation in Table 7. The main improvement seen in the LOP design is a reduction of 23% in the levels of logic at an added expense of 38% more area. The minimum clock period shows only a small improvement. Having addition in parallel with the leading-one detector helps reduce the levels of logic, but adds significant routing delay between the look-up tables (LUTs), as most of the added logic is in the form of

logic gates. This is the main reason for the lack of significant improvement in latency. In a VLSI design, the levels of logic affect the latency, and thus the LOP algorithm is a good option. However, for an FPGA design, the added area cost of 38% for an improvement of just 6.5% in latency suggests that this is not a feasible design option.

B. Pipelining

The LOP algorithm was also pipelined into five stages. Fig. 4 shows the micro-architecture of the five-stage pipelined implementation of the LOP floating-point adder algorithm. The dotted lines represent the registers used to separate five stages between logic. In the first stage of the implementation, the two operands are compared to identify denormalization and infinity. Then the two exponents are subtracted to obtain the exponent difference and determine whether the operands need to be swapped using the exponent difference sign. In the second stage, the right-shifter is used to pre-normalize the smaller mantissa. In the third stage, the addition is done along with leading-one prediction. In the fourth stage, a left-shifter is used to post-normalize the result. In the last stage, the exponent out is calculated, and rounding is done. The results are then compared to set overflow or underflow flags. Table 8 shows the comparison between a five-stage pipelined implementation of the LOP algorithm and data provided for the Xilinx IP core.

The pipelined LOP adder implementation [24] shows great improvement in clock speed compared to both a pipelined standard adder and the Xilinx IP core, but at the cost of added area. The five-stage pipelined standard adder implementation is a better choice in terms of area, occupying only 394 slices. If throughput is the criterion for performance, we note that the five-stage pipelined LOP adder implementation provides 22% better clock speed than the Xilinx IP and 19% better clock speed than the five-stage pipelined standard adder implementation and thus is clearly a better design choice. [24]

IV. FAR AND CLOSE DATA PATH ALGORITHM

According to studies, 43% of floating-point instructions have an exponent difference of either 0 or 1 [21]. A leading-one detector or predictor is needed to count the leading number of zeros only when the effective operation is subtraction and the exponent difference is 0 or 1; for all other cases, no leading-zero count is needed. Another opportunity for improvement is based on a compound adder which, with the help of additional logic, does addition and rounding in one step. The main contributions regarding this algorithm are presented in [7] and [21]–[23].

This algorithm is potentially larger than the previously implemented algorithms, but yields significant improvement in latency and thus is used in almost all current commercial microprocessors, including AMD, Intel, and PowerPC [5]. The micro-architecture of a far and close data path algorithm is shown in Fig. 5. Compared to the standard algorithm, one shifter delay and the rounding delay have been removed in critical paths of the data paths at the cost of almost double the hardware and a compound adder implementation. Most of the components were selected in Section II to implement the far and close data path algorithm and were synthesized with the Xilinx ISE. Table 9 shows a comparison between the standard algorithm and the far and close data path algorithm implementation on a Vertex-II Pro FPGA device.

The minimum clock period reported by the synthesis tool after placing and routing was 21.821 ns, which is 19% better than that of the standard floating-point adder implementation. Thirty levels of logic were reported, representing a 34% improvement. Both these improvements were realized because one shifter and one adder were removed in the critical paths. The maximum achievable clock speed for this implementation is 45.82 MHz. The number of slices reported by the synthesis tool was 1018, which is a significant increase compared to the standard algorithm. The far and close data path algorithm was not chosen for the five-stage pipelined implementation because the optimum number of stages is either three without sub-pipelining of the internal modules [22] or, according to our experiment, six with the sub-staging of the modules.

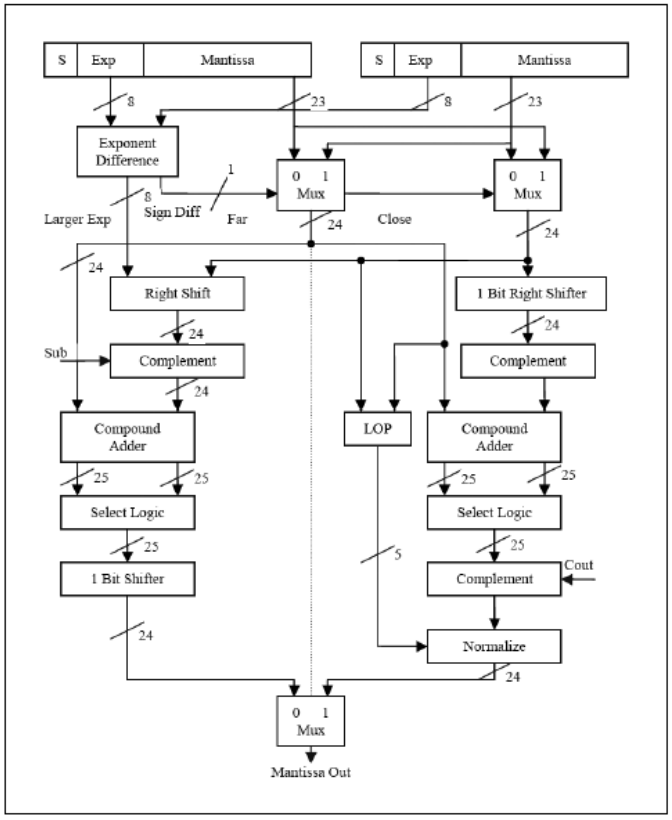

Figure 5: Far and close data path algorithm

Table 9

Standard and far and close datapath algorithm analysis

Module	Clock period (ns)	Clock speed (MHz)	Area (slices)	Levels of logic
Standard	27.059	36.95	541	46
Far and close datapath	21.821	45.82	1018	30
Percent improvement	+19	+19	−88	+34

V. CONCLUSION

This paper presented historical perspectives of computer arithmetic and FPGAs. The use of FPGAs in computationally intensive applications has been growing dramatically, and the floating-point adder is the most complex arithmetic circuit in such cases. We have given a detailed design tradeoff analysis of floating-point adders with respect to basic subcomponents of the standard adder algorithm along with implementation of the overall architectural improvements in a Vertex-II Pro FPGA.

The standard algorithm implementation was analyzed and compared with LOP and far and close data path algorithm implementations. The standard algorithm is area-efficient, but has more levels of logic and greater overall latency.

An LOP algorithm adds parallelism to the design and thus reduces levels of logic significantly, but because of added hardware and significant routing delays it does not significantly improve overall latency in FPGAs. It is not an effective design for FPGAs because the added slices required for an LOP use sum-of-product chains present in the CLBs of the Vertex- II Pro FPGA device and add significant routing and gate delay.

A far and close data path algorithm reduces overall latency significantly by distributing the operations into two separate paths, thus reducing critical paths and affecting latency significantly, but at the cost of added hardware.

REFERENCES

[1] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna, "Analysis of high-performance floating-point arithmetic on FPGAs," in Proc. Int. Symp. Parallel and Distributed Processing, Santa Fe, N.M., Apr. 2004, p. 149.

[2] Y. Lee and S. Ko, "FPGA implementation of a face detector using neural networks," in Proc. IEEE Can. Conf. Elect. Comput. Eng., May 2006, pp. 1883–1886.

[3] F. Pappalardo, G. Visalli, and M. Scarana, "An application-oriented analysis of power/precision tradeoff in fixed and floating-point arithmetic units for VLSI processors," in Proc. IASTED Conf. Circuits, Signals, and Systems, Dec. 2004, pp. 416–421.

[4] M. Farmland, On the Design of High Performance Digital Arithmetic Units, Ph.D. dissertation, Department of Electrical Engineering, Stanford University, Stanford, Calif., Aug. 1981.

[5] P.M. Seidel and G. Even, "Delay-optimization implementation of IEEE floatingpoint addition," IEEE Trans. Comput., vol. 53, no. 2, Feb. 2004, pp. 97–113.

[6] J.D. Bruguera and T. Lang, "Leading-one prediction with concurrent position correction," IEEE Trans. Comput., vol. 48, no. 10, 1999, pp. 1083–1097.

[7] S.F. Oberman, H. Al-Twaijry, and M.J. Flynn, "The SNAP Project: Design of floating-point arithmetic units," in Proc. 13th IEEE Symp. Computer Arithmetic, 1997, pp. 156–165.

[8] "DFPAU: Floating point arithmetic coprocessor," San Jose, Calif.: Digital Core Design, Dec. 14, 2007, http://www.dcd.pl/acore.php?idcore=13.

[9] "Quixilica floating point FPGA cores," Seoul, South Korea: Quixilica, Dec. 2002, http://www.eonic.co.kr/data/datasheet/transtech/FPGA/qxdsp001 fp.pdf .

[10] "IEEE 754 compatible floating point cores for Virtex-II FPGAs," Eldersburg, Md.: Nallatech, 2002, http://www.nallatech.com/mediaLibrary/images/english/2432.pdf.

[11] L. Louca, T.A. Cook, andW.H. Johnson, "Implementation of IEEE single-precision floating point addition and multiplication on FPGAs," in Proc. IEEE Symp. Field- Programmable Custom Computing Machines, 1996, pp. 107–116.

[12] W.B. Ligon, S. McMillan, G. Monn, F. Stivers, and K.D. Underwood, "A reevaluation of the practicality of floating-point operations on FPGAs," in IEEE Symp. FPGAs for Custom Computing Machines, Apr. 1998, pp. 206–215.

[13] E. Roesler and B.E. Nelson, "Novel optimizations for hardware floating-point units in a modern FPGA architecture," in Proc. IEEE Int. Conf. Field-Programmable Logic and Applications, Sept. 2002, pp. 637–646.

[14] J. Liang, R. Tessier, and O. Mencer, "Floating point unit generation and evaluation for FPGAs," in Proc. IEEE Symp. Field-Programmable Custom Computing Machines, Apr. 2003, pp. 185–194.

[15] A. Malik and S. Ko, "Efficient implementation of floating point adder using pipelined LOP in FPGAs," in Proc. IEEE Can. Conf. Elect. Comput. Eng., May 2005, pp. 688–691.

[16] Xilinx, http://www.xilinx.com/.

[17] J. Hennessy and D.A. Patterson, Computer Architecture: A Quantitative Approach, 2nd ed., San Francisco, Calif.: Morgan Kaufmann Publishers, 1996.

[18] I. Koren, Computer Arithmetic Algorithms, Natick, Mass.: A.K. Peters, 2002.

[19] V.G. Oklobdzija, "An algorithmic and novel design of a leading zero detector circuit: Comparison with logic synthesis," IEEE Trans. VLSI Syst., vol. 2, no. 1, 1994, pp. 124–128.

[20] M. Flynn, "Leading one prediction: Implementation, generalization, and application," Computer Systems Laboratory, Stanford University, Stanford, Calif., Tech. Rep. No. CSL-TR-91-463, Mar. 1991.

[21] M.J. Flynn and S.F. Oberman, Advanced Computer Arithmetic Design, New York: John Wiley & Sons, Inc. 2001.

[22] S.F. Oberman, "Design issues in high performance floating-point arithmetic units," Computer Systems Laboratory, Stanford University, Stanford, Calif., Tech. Rep. No. CSL-TR-96-711, Dec. 1996.

[23] J.D. Bruguera and T. Lang, "Rounding in floating-point addition using a compound adder," University of Santiago de Compostela, Santiago de Compostela, Spain, Internal Report, July 2000.

[24] Ali Malik, Dongdong Chen, Younhee Choi, Moon Ho Lee, and Seok-Bum Ko "Design tradeoff analysis of floating-pointadders in FPGAs" Can. J. Elect. Comput. Eng., Vol. 33, No. 3/4, Summer/Fall 2008.