Efficient Implementation of Pipelined Double Precision Floating Point Unit on FPGA

Riya Saini, Galani Tina G. and R.D. Daruwala

Abstract: The arithmetic circuits are the most fundamental blocks which are used to build DSP [9] hardware. Traditionally, digital signal processing (DSP) is performed using fixed-point or integer arithmetic. Use of floating-point arithmetic provides a large dynamic range. Floating-point representation is able to retain its resolution and accuracy compared to fixed-point representations. Unfortunately, floating point operators require excessive area (or time) for conventional implementations. Here we introduce a pipeline floating point arithmetic logic unit (ALU).Pipeline is used to give high performance and throughput to arithmetic operation. In this paper an arithmetic unit based on IEEE standard for floating point numbers has been implemented on FPGA Board. The arithmetic unit implemented has a 64- bit pipeline processing unit which allows various arithmetic operations such as, Addition, Subtraction, Multiplication and Division on floating point numbers. All the modules in the ALU design are realized using VHDL. Design functionalities are validated through simulation and compilation. The throughput is increased by pipelining the designed unit. This design unit is mapped onto Vertex 4 XC4VLX40 FPGA in order to achieve higher data rates. Comparative analysis is done between pipeline and sequential approach in terms of speed, power, throughput, latency and chip area.

Keywords: **Floating Point Unit, FPGA, VHDL, Xilinx ISE 12.1.**

I. INTRODUCTION

The reconfigurability and programmability of Field Programmable Gate Arrays (FPGAs) make them attractive tools for implementing digital signal processing (DSP) applications. However, DSP applications are arithmetic intensive tasks, requiring in most cases floating point operations to be performed as part of the application itself.

As demand rises for electronic devices to be smaller, faster and more efficient, increasing importance is placed on well designed pipelined architecture. Pipelined architecture that uses concurrent processing tends to use faster clock period, less combinational delay and hence faster speed but also consumes more chip area compared with standard architecture(without pipeline) that uses sequential processing. Pipelining is directly proportional to chip area. As stages of pipelining increases there is increase in throughput but with an adverse effect of increase in chip area.

Riya Saini, Galani Tina G. are M.Tech Student, dept. of Electrical Engineering, VJTI, Matunga, Mumbai-19 and R. D. Daruwala is working as Professor, dept. of Electrical Engineering, VJTI, Matunga, Mumbai-19, Emails: sainiriya993@gmail.com

The IEEE standard for binary floating-point arithmetic [2] provides a detailed description of the floating-point representation and the specifications for the various floating-point operations. It also specifies the method to handle special cases and exceptions.

In this paper an arithmetic unit based on IEEE standard for floating point numbers has been implemented on FPGA. The arithmetic unit implemented has a 64- bit pipeline processing unit. All the modules in the ALU design are realized using VHDL. We implemented our design using Xilinx ISE 12.1, with Xilinx Virtex-4 XC4VLX40 FPGA as our target device. In the top-down design approach, four arithmetic modules: addition, subtraction, multiplication, and division: are combined to form the floating-point ALU. The pipeline modules are independent of each other.

The organization of this paper is as follows: Section 2 describes background on floating point representation. Section 3 describes algorithm used by each module in design of floating point unit. Section 4 describes our approach in design of FPU. The simulation environment and results are summarized in Section 5 and concluding remarks are discussed in Section 6.

II. BACKGROUND

A. Floating Point Representation

Standard floating point numbers are represented using an exponent and a mantissa in the following format:

$$(sign\ bit)\ mantissa \times base^{exponent} + bias$$

The *mantissa* is a binary, positive fixed-point value. Generally, the fixed point is located after the first bit, *m0*, so that *mantissa = {m*0.*m*1*m*2*...mn}*, where *mi* is the *ith* bit of the mantissa. The floating point number is "normalized" when *m*0 is one. The *exponent*, combined with a *bias*, sets the range of representable values. A common value for the bias is $-2k-1$, where k is the bit-width of the exponent [4].The double precision floating point format has an 11 bit exponent and 52 bit mantissa plus a sign bit. This provides a wide dynamic range.

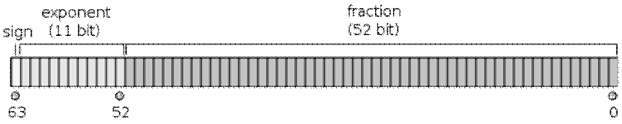


Fig 1.64-bit Floating Point Format

The sign bit occupies bit 63. '1' signifies a negative number, and '0' is a positive number. The exponent field is 11 bits long, occupying bits 62-52. The value in this 11-bit

field is offset by 1023, so the actual exponent used to calculate the value of the number is $2^{(E-1023)}$. The mantissa is 52 bits long and occupies bits 51-0. There is a leading '1' that is not included in the mantissa, but it is part of the value of the number for all double precision floating point numbers with a value in the exponent field greater than 0.

B. Reconfigurable Computing Systems

Field Programmable Gate Arrays (FPGAs) provide a hardware fabric upon which applications can be programmed. FPGAs are based on look-up tables, flip-flops, and multiplexers. An FPGA device consists of tens of thousands of logic blocks (a cluster of *slices*) whose functionality is determined by programmable configuration bits. These logic blocks are connected using a set of routing resources that are also programmable. Thus, mapping a design to an FPGA consists of determining the functions to be computed by the logic blocks, and using the configurable routing resources to connect the blocks.

Recently, the computing power of FPGAs has increased rapidly. Besides more configurable slices, current FPGA fabrics now contain large numbers of hardware primitives, such as fixed-point multipliers, Block RAMs (BRAMs), etc [8]. Therefore, FPGAs have become an attractive option for implementations of floating-point applications because multiple floating-point units can be configured on one device.

III. ALGORITHMS FOR ARITHMETIC OPERATIONS

A. Addition and Subtraction

The conventional floating-point addition algorithm consists of five stages – exponent difference, pre-alignment, addition, normalization and rounding.

Figure 2 shows a flow chart of adder/subtractor algorithm. In order to perform floating-point addition/subtraction, a simple algorithm is realized:

1. Compare the exponents and take difference.
2. Shift the smaller significand by difference.
3. Add/Subtract the mantissas according to sign bit.
4. Normalize the resulting value, if necessary.

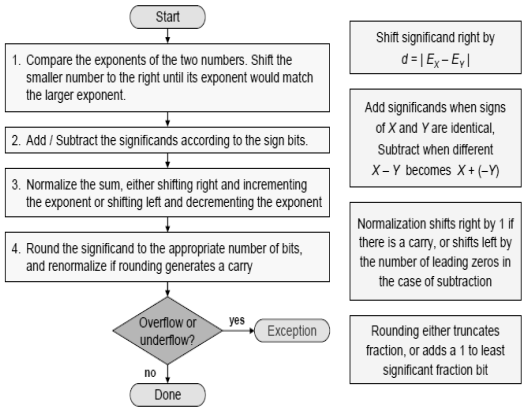


Fig 2.Floating Point Addition/Subtraction Algorithm

B. Multiplication

The multiplier structure is organized as a three-stage pipeline. Figure 3 shows a flow chart of multiplier algorithm.

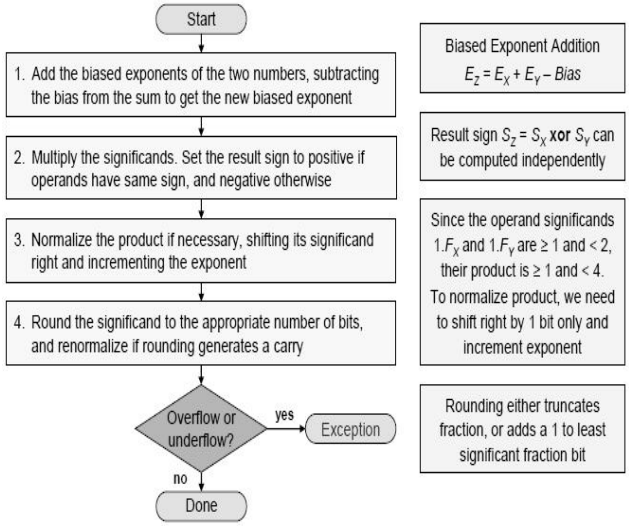


Fig 3.Floating Point Multiplier Algorithm

The two mantissas are to be multiplied, and the two exponents are to be added. In order to perform floating-point multiplication, a simple algorithm is realized:

1. Add the exponents and subtract 1023 (bias).
2. Multiply the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.

C. Division

Figure 4 shows flow chart of division algorithm.

1. For division the two mantissa are compared and aligned.
2. Take the difference of the exponents and add bias to them.
3. Xoring the sign bits.
4. Normalize the result.

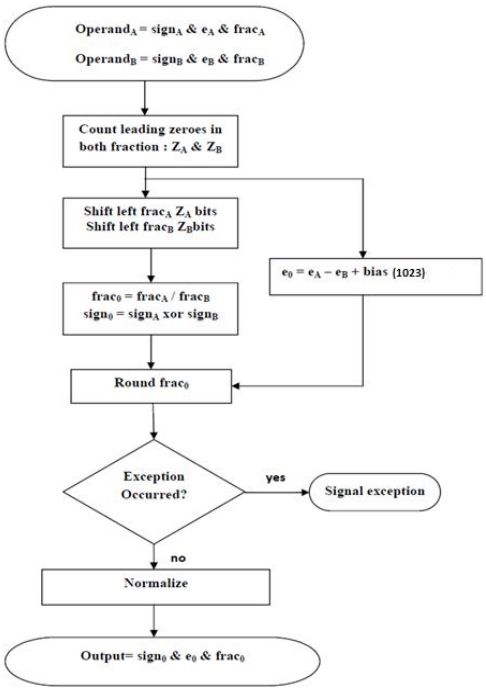


Fig 4.Floating Point Division Algorithm

Mantissa of operand A is dividend and mantissa of operand B is divisor. Division is performed by comparison between dividend and divisor. If the dividend is greater than the divisor quotient bit is set to '1' and then the divisor is subtracted from the dividend, this difference is shifted one bit to the left and it becomes dividend for the next clock cycle. If the dividend is less than the divisor, the dividend is shifted one bit to the left and then this shifted value becomes the dividend for the next clock cycle.

IV. PIPELINED FLOATING POINT UNIT

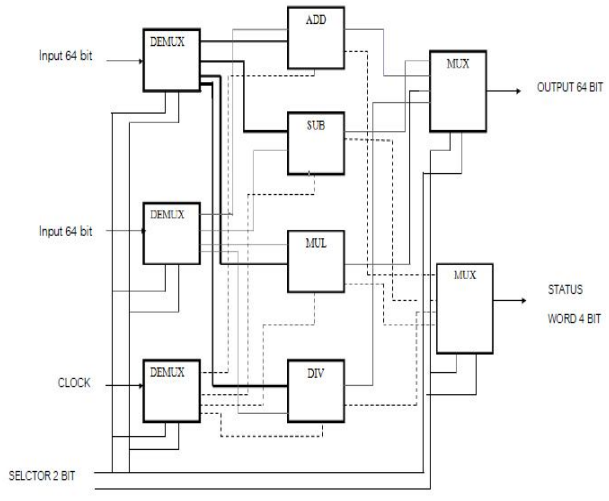


Fig 5.Floating Point Unit

ALU is separated into smaller modules: addition, subtraction, multiplication, division, demux and mux (selector). Each arithmetic module is further divided into smaller modules that are coded individually. Fig. 5 shows the top level view of the ALU. It consists of four functional arithmetic modules, three demultiplexers and two multiplexers. The demultiplexers and multiplexers are used to route input operands and the clock signal to the correct functional modules. They also route outputs and status signals based on the selector pins.

After a module completes its task, outputs and status signals are sent to the muxes where they multiplexed with other outputs from corresponding modules to produce output result. Selector pins are routed to these muxes such that only the output from the currently operating functional module is sent to the output port.

i. 'Selection' a 2-bit input signal that selects ALU operation and operate as shown below:

Selections
00 - Addition
01 - Subtraction
10 - Multiplication
11 - Division

ii. 'Status' a 4-bit output signals. It notifies the state of the arithmetic results as follows:

Status
"0000" – result zero
"0001" – overflow
"0010" – underflow
"1000" – normal operation
"1000" – divide by zero

iii. Only the module that is selected using a demux apply the clock cycle and saves it.

iv. Different processes are run in parallel, hence pipelining.

A. Pipeline Floating Point Addition/Subtraction

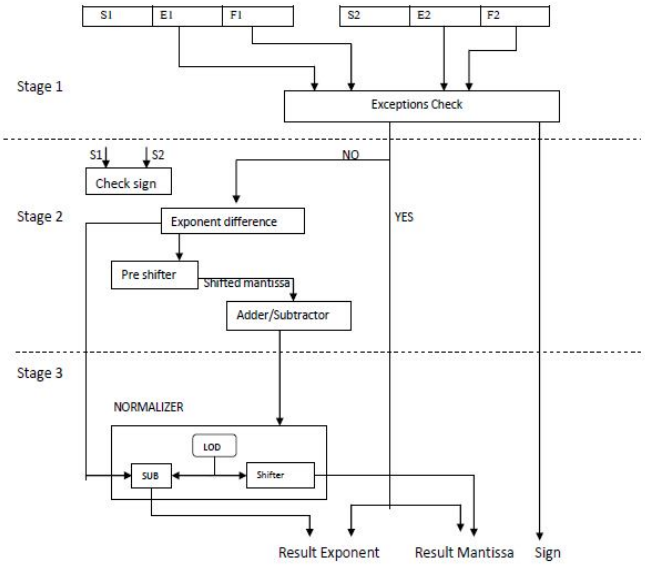


Fig 6.Floating Point Adder/Subtractor

The two operands are read in and compared for denormalization and infinity. If numbers are denormalized, set the implicit bit to 0 otherwise it is set to 1. The fractional part is extended to 52 bits. The two exponents e1 and e2 are

compared. Now smaller is right shifted by the difference between the two exponents. Now both the numbers have the same exponent. The signs are used to see whether the operation is addition or subtraction. If the operation is subtraction, perform 2's complement addition. If the result is negative take the 2's complement of the result to obtain actual result. The result is normalized by left shifting. Then the result is checked for overflow and underflow and the sign of the result is also computed.

B. Pipeline Floating Point Multiplier

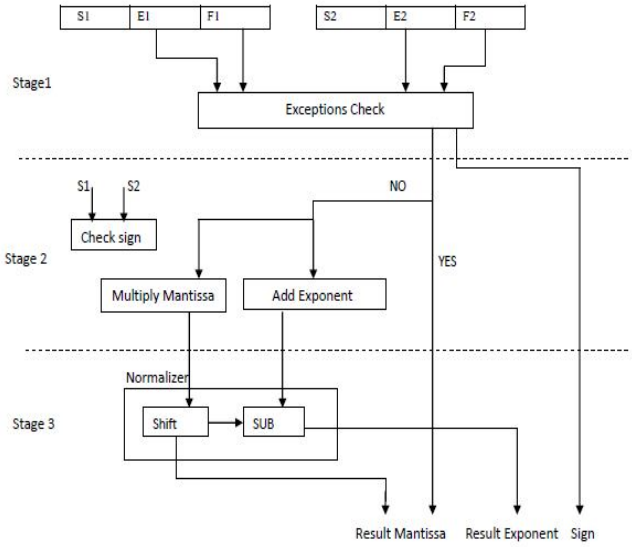


Fig 7.Floating Point Multiplier

Multiplication module is divided into five modules: exceptions check, add exponent, multiply mantissa, check sign, and normalize, which are executed concurrently. Status signal indicates special result cases such as overflow, underflow and result zero. Pipelined floating point multiplication is divided into three stages.

Stage 1.check whether the operand is 0 and report the result accordingly.

Stage 2.determines the product sign add exponents and multiply fractions.

Stage3.normalize and concatenate the product.

During the normalization operation, the mantissa's MSB is 1. Hence, no normalization is needed. The hidden bit is dropped and the remaining bit is packed and assigned to the output port. Normalization module set the mantissa's MSB to 1. The current mantissa is shifted left until a 1 is encountered. For each shift the exponent is decreased by 1. Therefore, if the mantissa's MSB is 1, normalization is completed and first bit is the implied bit and dropped. The remaining bits are packed and assigned to the output port.

C. Pipeline Floating Point Divider

Division module is divided into five modules: exceptions check, check sign, subtract exponent, divide mantissa and normalize concatenate modules. Each module is executed concurrently. Status indication is used to indicate special result cases such as overflow, underflow, result zero and divided by zero. Fig.8 shows the pipeline structure of the division module.

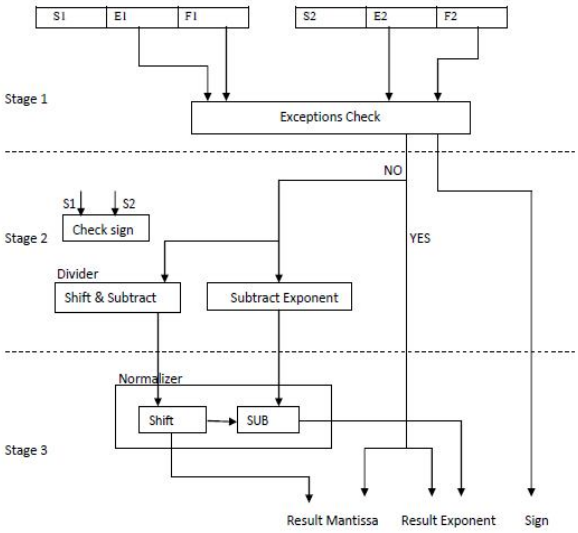


Fig 8.Floating Point Divider

V. SIMULATION AND RESULTS

The functional and timing analysis is carried out using Xilinx ISE 12.1 software. Programming is done on VHDL. Selection lines are used to select the corresponding modules. For each module the result is obtained after 4 clock cycles for a clock of 10 ns. The special cases and exception cases are signaled by a flag. Figure below show the Synthesis and simulation results of floating point unit.

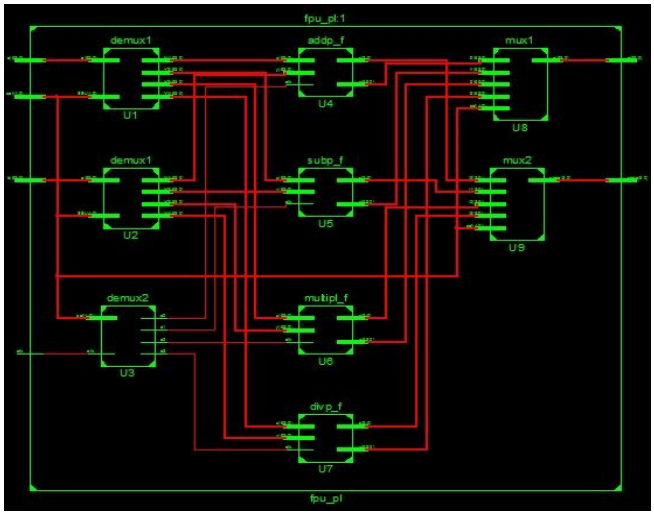


Fig 9.Synthesize result of Floating Point Unit

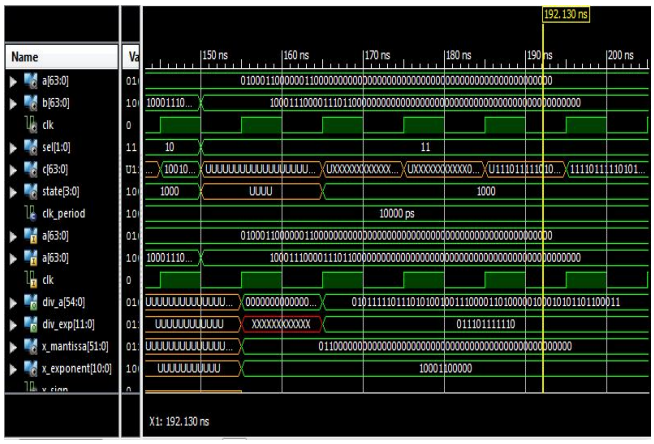


Fig 10.Simulation result of Floating Point Unit

All four arithmetic operations available in the design are tested with the same inputs. Finally we compare our module with sequential processing module. For the processing module various parameters i.e. speed, clock period, chip area (no. of slices used), power dissipation and combinational delay are compared. Comparing various parameters provide with the information that which type of processing in a floating point unit will take more clock period and less chip area. Table1 represents the maximum power dissipation, area occupied and time taken by each module to compute the results.

Table I

Module	Delay (ns)	Slice Utilized (Area)	Max. Frequency (MHz)	Total Power (W)
Adder	8.252	644	121.189	0.35
Subtracter	8.344	841	119.847	0.36
Multiplier	14.683	374	68.105	0.35
Divider	219.79	3104	4.55	0.38

VI. CONCLUSION

By simulating with various test vectors the proposed approach of 64bit pipeline floating point ALU design using VHDL is successfully designed, implemented, and tested on Vertex 4 FPGA. The design achieves higher performance, speed and lower power dissipation as compared to sequential approach.

Table II

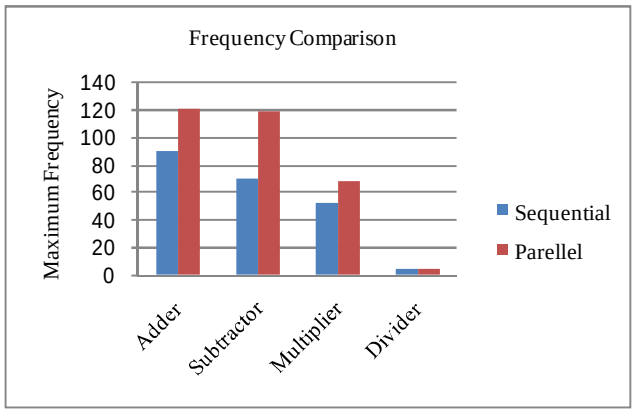
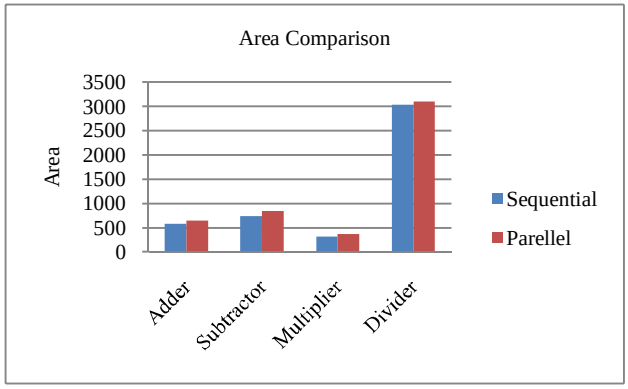


Table III



The simulation results show that maximum frequency of pipeline processor is increased which gives speed to our floating point computation. Pipelined architecture provide a faster response in floating point arithmetic but also consumes more area i.e. number of slices used on reconfigurable hardware are more as compared with standard architecture. Pipelining is used to decrease clock period. Using sequential processing there is larger latency but less number of slices are used on FPGAs as compared with pipelined architecture. We are conducting further research that considers the further reductions in the area and time required by divider.

REFERENCES

[1] W. Kahan "IEEE Standard 754 for Binary Floating-Point Arithmetic," 1996.

[2] Taek-Jun Kwon, Jeff Sondeen and Jeff Draper ,"Floating-Point Division and Square Root Implementation using a Taylor-Series Expansion Algorithm", IEEE 2008.

[3] Michael Parker, Altera Corporation," High-Performance Floating-Point Implementation Using Fpgas".

[4] K. Underwood. FPGAs vs. CPUs: Trends in peak floating point performance. In Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays, pages 171–180. ACM New York, NY, USA, 2004.

[5] B. Parhami. Computer arithmetic: algorithms and hardware designs. Oxford University Press Oxford, UK, 1999.

[6] F. De Dinechin et al. When FPGAs are better at floating-point microprocessors. ´E Normale Superieure de Lyon, Tech. Rep. ensl-00174627, 2007.

[7] B. Fagin and C. Renard. Field programmable gate arrays and floating point arithmetic. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2(3):365–367, 1994.

[8] The Institute of Electrical and Electronic Engineers, Inc. IEEE Standard for Binary Floating-point Arithmetic. ANSI/IEEE Std 754-1985.

[9] P. Belanovi´*c* and M. Leeser. A Library of Parameterized Floating Point Modules and Their Use. In *Proceedings, International Conference on Field Programmable Logic and Applications*, Montpelier, France, Aug. 2002.

[10] A. A. Gaffar, W. Luk, P. Y. Cheung, N. Shirazi, and J. Hwang. Automating Customization of Floating-point Designs. In *Proceedings, International Conference on Field-Programmable Logic and Applications*, Montpelier, France, Aug. 2002.

[11] W. B. Ligon, S. McMillan, G. Monn, F. Stivers, and K. D. Underwood. A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs. In *Proceedings, IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 206–215, Napa, CA, Apr. 1998.

[12] L. Louca, W. H. Johnson, and T. A. Cook. Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs. In *Proceedings, IEEE Workshop on FPGAs for Custom Computing Machines*, pages 107–116, Napa, CA, Apr. 1996.

[13] E. M. Schwarz, M. Schmookler, and S. D. Trong. FPU implementations with denormalized numbers. IEEE Transactions on Computers, 54(7), July 2005.

[14] Xilinx, Inc, "Virtex - 4 Platform FPGAs: Complete Data Sheet," 2010.