Vendor Agnostic, High Performance, Double Precision Floating Point Division for FPGAs

Xin Fang and Miriam Leeser

Dept of Electrical and Computer Eng
Northeastern University
Boston, Massachusetts 02115
Email: fang.xi@husky.neu.edu, mel@coe.neu.edu

***Abstract*—Double precision Floating Point (FP) arithmetic operations are widely used in many applications such as image and signal processing and scientific computing. Field Programmable Gate Arrays (FPGAs) are a popular platform for accelerating such applications due to their relative high performance, flexibility and low power consumption compared to general purpose processors and GPUs. Increasingly scientists are interested in double precision FP operations implemented on FPGAs. FP division and square root are much more difficult to implement than addition and multiplication. In this paper we focus on a fast divider design for double precision floating point that makes efficient use of FPGA resources including embedded multipliers. The design is table based; we compare it to iterative and digit recurrence implementations. Our division implementation targets performance with balanced latency and high clock frequency. Our design has been implemented on both Xilinx and Altera FPGAs. The table based double precision floating point divider provides a good tradeoff between area and performance and produces good results when targeting both Xilinx and Altera FPGAs.**

I. Introduction

The IEEE 754 standard [7] specifies the common formats for representing floating point numbers, including single and double precision. Double precision is widely used in scientific applications since it can represent a wider range of values with greater precision than single precision. There are many computations where a list of input values are summed and then normalized. Even if the input values are single precision, the accumulation and division are done in double precision to retain accuracy. In hardware, double precision division operates more slowly and requires more hardware resources than single precision. Speed is measured in number of clock cycles to produce a result (latency), clock cycle time, and total time for a single operation. Many of these resource issues compete with one another. For example, a subtractive divider, similar to the long division taught in grade school, uses few resources and can be implemented with a small clock cycle, but has a long latency. In contrast, a multiplicative divider uses more resources, a longer clock cycle, and has a shorter latency. The ideal divider optimizes all three of these constraints.

In this paper we describe a double precision, IEEE compliant floating point divider, adopted from [3], which uses a combination of Look Up Tables (LUTs) and Taylor series expansion, and thus tries to balance the competing design goals of fast operation with a small amount of hardware. Due to its use of LUTs and multipliers, this design is particularly well suited for FPGA implementation with their embedded multipliers and RAM. The divider that we implement consists of a reciprocal unit followed by a multiplication unit, so the reciprocal unit can be separated into an independent component in the library.

Note that our implementation is designed to handle any width of exponent and mantissa, and not just those formats specified by the IEEE 754 floating point standard. To support this, we make use of parameterized modules, including and gates, or gates, adders and subtractors. In this paper we have focused on the results for a double precision divider; hence other formats are not covered here.

The rest of this paper is organized as follows. Background, including the IEEE double precision format representation as well as three famous methods for computing floating point division are presented in Section II. Section III describes the algorithm in detail and Section IV introduces how to implement the divider. Section V presents the results of our project and a comparison with other division implementations on both Altera and Xilinx hardware. The paper ends with conclusions and future work.

II. Background

A. Double Precision Format Representation

The IEEE 754 double precision standard format is represented using one sign bit, 11 exponent bits, and 52 mantissa bits, for a total of 64 bits. The exponent bias is 1023, and for normalized numbers, the leading 1 is not explicitly stored. The value represented is therefore:

$$(-1)^{sign} * 2^{exponent-1023} * 1.mantissa$$

To implement the division of two floating point numbers, $Z = X/Y$; the sign bit, exponent and mantissa of Z must be calculated and the result normalized. The sign is the XOR of the sign bits of X and Y, while the resulting exponent (before normalization) is the difference between the two input exponents with the bias suitably handled. The bulk of the computation is the division operation, which is the same as integer division. We will focus on this part for the remainder of the paper.

The IEEE standard also specifies other values, including subnormal numbers, NaN, and $\pm\infty$. We do not consider these values further.

978-1-4799-1365-7/13/$31.00 ©2013 IEEE

B. Methods For Computing Floating Point Division

There are three main methods used to implement floating point division. They are digit recurrence or subtractive, iterative or multiplicative and table-based. Each method is discussed in more detail below.

1) Digit Recurrence: Digit recurrence algorithms compute one digit of the result per clock cycle and are based on the long division commonly taught in grade school for performing division by hand. The most widely used digit recurrence method is known as SRT after the three people, Sweeney, Robertson, and Tocher who invented it [11]. The textbook Digital Arithmetic [2] gives more details. Freiman [4] provides a good example of the SRT method based on non-restoring division; the radix of the example is 2. For binary arithmetic, the approach used to obtain more than one bit per cycle employs a radix higher than 2.

2) Iterative Method: Iterative methods are based on multiplications that gives you intermediate results that converge to the highest number of bits of precision required. Two widely used iterative methods for floating point division are Newton Raphson and Goldschmidt [13], [14], [12], [5].

With Newton-Raphson, the inverse of the divisor is computed and then multiplied with the dividend using a double precision multiplier. In order to find $1/Y$, the iteration is seeded with S_0, an estimate to $1/Y$; the iteration is: $S_{i+1} = S_i * (2 - Y * S_i)$. The algorithm iterates until S approximates the required precision.

Goldschmidt's algorithm also uses an iterative loop to calculate the quotient. The difference between this and Newton-Raphson is that Goldschmidt's algorithm multiplies both the dividend and the divisor. The iteration completes when the divisor is approximately equal to one and the required precision is reached. The value by which the dividend and divisor are multiplied in each iteration is 2 - current value of the divisor.

3) Table based Method: Two table based methods are discussed here. The first is currently used to implement variable precision dividers in the VFLOAT library [9], [16], [15]. The second is an improved algorithm whose table size grows more slowly as the bit width of the inputs grows. These methods use a stored value for the initial guess to a result. They use Taylor series expansion to obtain the required number of bits of precision.

The first approach, from [6], is the divider currently used in VFLOAT. It makes use of two multipliers and one look up table to implement division. In the single precision floating point divider implementation, the size of the first multiplier is 24×24 bits of input with 48 bits of output 48; the second multiplier has input 28×28 bits with an output of 56 bits. The look up table has 12 bits for input, 28 for output, for a total size of approximately 16K bytes. However, the disadvantage of this algorithm is that the size of the look up table increases exponentially with the bit width of the input. It is therefore impractical to implement a double precision divider with this approach since the LUT would be prohibitively large. For this reason we have changed to a different divider algorithm that scales better as the size of the division operation grows [3] . This algorithm requires a smaller look up table and several small multipliers. The remainder of this paper covers details of this algorithm and its implementation. The next section explains the algorithm that we use. Section IV covers the implementation of the divider using VHDL. Experimental results obtained from this implementation and a comparison to previous methods are presented in Section V. Finally we conclude and present future work.

III. Algorithm Description

The algorithm we use is from [3]. The discussion below is taken from that paper and more details can be found there. This approach finds X/Y by first finding the reciprocal of Y and then multiplying by X. The approach is based on a Taylor series expansion and makes use of LUTs and multipliers, resources readily available in FPGAs. To ensure growth of the look up table is slow, a reduction step is used.

A. Reciprocal Operation

To find the reciprocal $1/Y$ the algorithm uses three steps: reduction, evaluation, and post-processing. The top level is shown in Figure 1.

a) The reduction step: From the IEEE standard, we know that for Y normalized, $1 \leq Y < 2$. Assume Y has an m bit significand and k is $\lfloor (m+2)/4 \rfloor + 1$; $Y^{(k)}$ represents the truncation of Y to k bits. In the reduction step, define $\hat{R}$ as the reciprocal of $Y^{(k)}$. $\hat{R}$ can be determined using a look up table with a 14 bit address for double precision. This is due to the fact that the mantissa is 53 bits with the leading one represented, so $k = 14$. The number of bits used for $\hat{R}$ is 16.

B is defined as the Taylor series expansion of

$$f(A) = 1/(1+A)$$

where A is defined as $(Y \times \hat{R}) - 1$. Note that $-2^{-k} < A < 2^k$. For $Z = 2^{-k}$, A can be represented as:

$$A = A_2 z^2 + A_3 z^3 + A_4 z^4 + ...$$

where $|A_i| \leq 2^k - 1$. We ignore the smaller terms that do not contribute to the double precision result.

b) In the evaluation step: using the Taylor series expansion,

$$\begin{aligned} f(A) &= C_0 + C_1 A + C_2 A^2 + C_3 A^3 + C_4 A^4 + \cdots \\ &\approx C_0 + C_1(A_2 z^2 + A_3 z^3 + A_4 z^4) \\ &\quad + C_2(A_2 z^2 + A_3 z^3 + A_4 z^4) \\ &\quad + C_3(A_2 z^2 + A_3 z^3 + A_4 z^4)^3 \\ &\quad + C_4(A_2 z^2 + A_3 z^3 + A_4 z^4)^4 \\ &\approx C_0 + C_1 A + C_2 A_2^2 z^4 + 2C_2 A_2 A_3 z^5 + C_3 A_2^3 z^6 \end{aligned}$$

Here, $C_i = 1$ when i is even, $C_i = -1$ when i is odd. Simplifying we get:

$$\begin{aligned} B &\approx 1 - A_2 z^2 - A_3 z^3 + (-A_4 + A_2^2) z^4 + \\ &\quad 2A_2 S_3 z^5 - A_2^3 z^6 \\ &\approx (1 - A) + A_2^2 z^4 + 2A_2 A_3 z^5 - A_2^3 z^6 \end{aligned}$$

The above equation is the one used in the implementation.

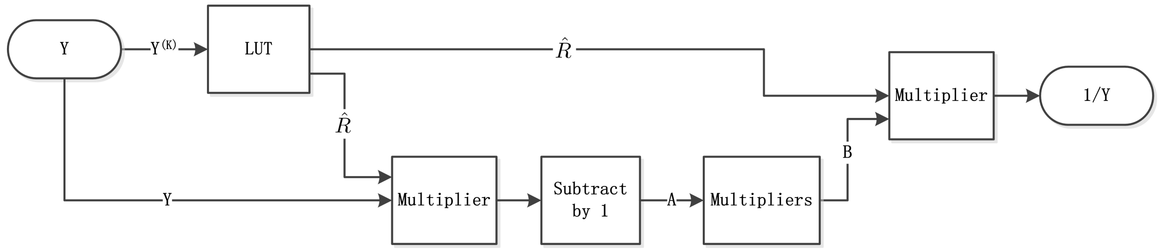

Fig. 1. Reciprocal block diagram

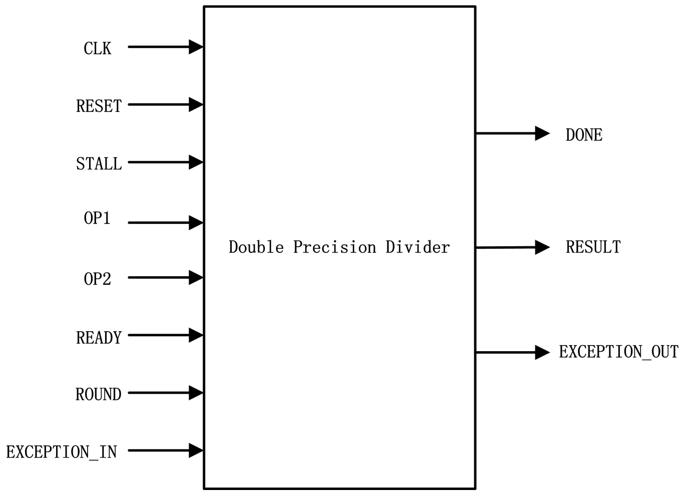

Fig. 2. Black box of design

c) In the post-processing step: , the result of the reciprocal of Y is given by the product of $\hat{R}$ and B: $1/Y = \hat{R}B$.

B. Multiplying Reciprocal by Dividend

After obtaining the reciprocal of the divisor, the final step is to multiply the reciprocal with X which is the dividend. In our implementation, this is done before the number is normalized. Another multiplier is generated for this operation. The final result of division is obtained by combining the result of X/Y with the sign and exponent; after X/Y is normalized and rounded the resulting exponent is appropriately adjusted.

IV. Divider Implementation

The double precision divider is designed as part of the VFLOAT library available for download from Northeastern University [9], [16]. Components in the VFLOAT library are designed to be joined together as a pipeline. Thus they have inputs READY, STALL, ROUND and EXCEPTION_IN and outputs DONE and EXCEPTION_OUT specifically to handle pipelining. The READY and DONE signals are used for knowing when the inputs are ready and when the results are available for use. STALL allows a bubble to be inserted into the pipeline if needed. Round has two modes: round to zero or truncate, and round to nearest. The exception signals propagate an exception flag with the value that may be incorrect through the pipeline. For division, an exception is identified if the divisor is zero. Otherwise, the exception input is propagated to the exception output. The complete list of input ports is: CLK, RESET, STALL, OP1, OP2, READY, ROUND and EXCEPTION_IN. The output ports are DONE, RESULT, and EXCEPTION_OUT, as shown in Figure 2.

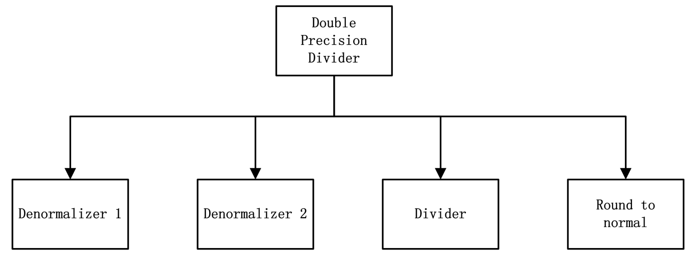

Fig. 3. Top level components of divider

Our implementation of the division algorithm for double precision floating point is written in VHDL and targets the two most popular families of commercial FPGAs: Altera and Xilinx. For tools we use Altera Quartus II 13.0 and the Xilinx ISE Design Suite 13.4. The designs make use of embedded multipliers and RAMs, which require using the intellectual property components provided with each set of tools. For Altera these are called Megacores; for Xilinx they are called IP Cores. The two manufacturers also use different simulators; Altera ships a version of Modelsim called ModelSim-Altera 10.1d while Xilinx has its own simulator, ISim, that is integrated with its tool suite. Both simulators were used to validate these designs.

Figure 3 shows the top level components of the divider. The top level divider consists of computation to denormalize the floating point input (explicitly represent the hidden one), divider computation which includes a reciprocal and a multiplication of the dividend with the reciprocal, and round to normal for adjusting the result to standard floating point format.

A. Denormalizer

A normalized floating point number in IEEE 754 standard format has a sign bit, exponent bits and mantissa bits. As described in Section II, the stored part of the mantissa does not include the leading '1'; however, this bit is necessary for computation. All normalized floating point inputs to a component are sent through a denormalizer which adds this '1' to the head of the mantissa. For the divider, two such steps are needed, one for the dividend and one for the divisor.

B. Divider

After obtaining the denormalized number, we need to calculate the sign bit, exponent and quotient of the result. The

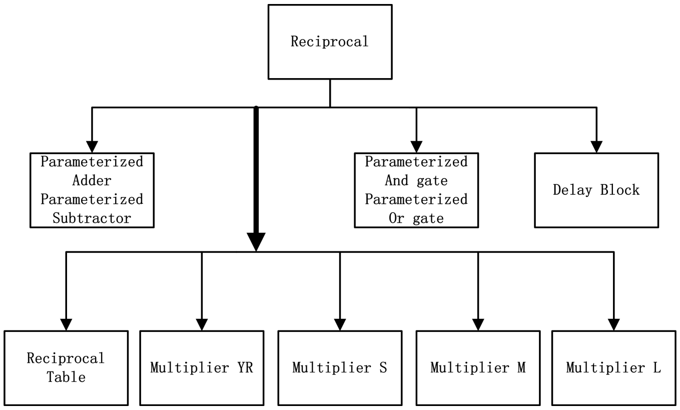

Fig. 4. Reciprocal components

sign bit is obtained by XORing the sign bits of the inputs. The exponent of the result is the exponent of the dividend minus the exponent of the divisor with the bias suitably handled. Note that the output of this subtraction might not be the exponent of the final result, because we still need to normalize the result after the division operation. The following subsections describe the computation of the reciprocal and the multiplication with the dividend to obtain the final mantissa.

1) Reciprocal Operation: Figure 4 shows the hierarchy of structural components used to implement the reciprocal algorithm described in Section III-A. The design uses the reciprocal table to calculate $\hat{R}$ and 4 multipliers. The lookup table has 14 address bits and 16 output bits for a total of 32K bytes. The four multipliers are the following sizes. Multiplier YR has 17 bit and 58 bit inputs and a 71 bit output; its purpose is to multiply Y and $\hat{R}$ in order to get A. Multiplier S has two 14 bit inputs and a 28 bit output; its purpose is to compute $A_2 * A_2$ and $A_2 * A_3$. Multiplier M has 28 bit and 14 bit inputs and a 42 bit output; it computes the cube of A_2. Multiplier L has one 16 bit and one 58 bit input and a 74 bit output; it computes $\hat{R} * B$. In addition, parameterized adders and logic gates from the VFLOAT library are used.

2) Multiplier: After determining the reciprocal of the divisor, the next step is to multiply this reciprocal with the mantissa of the dividend. This multiplier is called the mantissa multiplier. The inputs are 53 bits and 57 bits, and the output is 110 bits. The number of pipeline stages for this and the other multipliers in the implementation can be adjusted using the parameters available in Xilinx Core Generator or Altera Megacores. After this multiplication, the last component will round an renormalize the output.

C. Round to Normal

The IEEE standard specifies four rounding modes: Round to zero, round to nearest, round to negative infinity and round to positive infinity. In the VFLOAT library, two rounding options are supported: round to zero or truncation and round to nearest. Each component has an input signal, round. If Round is 0, truncation is implemented, else round to nearest. This operation may change the exponent as well as the mantissa of the result. The round component also removes the leading '1' from the mantissa of the result for storage in IEEE floating point format. After this step, the normalized, double precision floating point result represented in IEEE standard notation will be on the output port.

V. Results and Comparison

We have implemented our double precision floating point divider and compared it to other, popular designs. Results are shown in Table I for designs mapped to Xilinx and Table II for designs mapped to Altera. As mentioned above, we used Altera Quartus II 13.0 and the Xilinx ISE Design Suite 13.4 to implement our design. CC is the number of clock cycles latency. All designs can be pipelined with a throughput of one clock cycle. Our design has the advantage of working on both Altera and Xilinx FPGAs. In contrast, the IP cores in Xilinx and Megacores from Altera are vendor specific. Our designs allow the user to adjust the number of clock cycles of latency by adjusting the level of pipelining for each multiplier. Xilinx allows the designer to choose the latency of the divider as a parameter in Core Generator. We chose to compare to latencies of 8, 14 and 20 clock cycles. For Altera Megacores, the designer is offered a handful of latencies from which to choose.

The latency of our first implementation is 14 clock cycles on both vendor's hardware. The maximum frequency on a Stratix V device(5SGXB6) from Altera is 121MHz. Implemented on a Xilinx Virtex 6 device (XC6VLX75T), the maximum frequency is 148MHz. Note that a faster clock frequency does not necessarily make a better design. Since our goal is to fit the divider into larger pipelines, the goal is to balance the clock speeds across all components. In addition, there is often a tradeoff between high clock frequency and number of clock cycles latency, as can be seen in these results. Computing the result in the same total time with a lower clock frequency dissipates less energy, another design goal. However, too slow of a latency can slow down the entire pipeline.

We compared our design to several popular methods for implementing division, including the floating point cores provided from each manufacturer. Table I shows the synthesis results of several methods using the Xilinx IDE. Note that given the same latency, our design provides a better maximum frequency. This faster frequency comes at the cost of more hardware resources being used. The last design is reported from a paper that presents a multiplicative divider [8]. This design can only support double precision with an error of 2 units in the last place (ulps) while our error is one ulp. Also, this design has long latency although a fast maximum frequency. These numbers are reported by the authors; we did not implement their design.

Table II shows the synthesis results of several methods using the Altera IDE. The Megacore from Altera has only three fixed latencies that you can choose from: 10, 24 and 61 clock cycles. In this case, the Altera Megacores provide the best overall latency. Note that these designs use more DSP blocks than our design. Also note that we can change the latency, as our second implementation shows, by adjusting the pipeline parameter of the MegaCore multipliers. Thus our divider design is more flexible compared to the one available from Altera. The last two designs repeat results previously published in [10]. The Radix 4 digit recurrence implemented by FloPoCo has larger maximum frequency but also longer

Method	Device	Latency	Max Frequency	Total Latency	Resource
Our Implementation	Virtex 6	14 Clk Cycles	148 MHz	95 ns	934 Registers, 6957 Slice LUTs
IP Core from Xilinx	Virtex 6	8 Clk Cycles	74 MHz	108 ns	722 Registers, 3210 Slice LUTs
IP Core from Xilinx	Virtex 6	14 Clk Cycles	117 MHz	120 ns	1188 Registers, 3234 Slice LUTs
IP Core from Xilinx	Virtex 6	20 Clk Cycles	192 MHz	104 ns	2035 Registers, 3216 Slice LUTs
Multiplicative	Virtex 6	36 Clk Cycles	275 MHz	131 ns	2097 Slices, 118K BRAM, 28 18*18

TABLE I. RESULTS AND COMPARISON WITH XILINX

Method	Device	Latency	Max Frequency	Total Latency	Resource
Our Implementation 1	Stratix V	14 Clk Cycles	121 MHz	116 ns	818 ALMs, 931 Logic Register, 11 DSP block
Our Implementation 2	Stratix V	16 Clk Cycles	145 MHz	110 ns	1004 ALMs, 1105 Logic Register, 13 DSP block
MegaCore from Altera	Stratix V	10 Clk Cycles	176 MHz	57 ns	525 ALMs, 1247 Logic Register, 14 DSP block
MegaCore from Altera	Stratix V	24 Clk Cycles	237 MHz	101 ns	849 ALMs, 1809 Logic Register, 14 DSP block
MegaCore from Altera	Stratix V	61 Clk Cycles	332 MHz	184 ns	9379 ALMs, 13493 Logic Register
Radix 4 Digit Recurrence	Stratix V	36 Clk Cycles	219 MHz	164 ns	2605 ALMs, 5473 Logic Register
Polynomial Approx d =2 + Newton Raphson	Stratix V	18 Clk Cycles	268 MHz	67 ns	444 ALMs, 823 Logic Register, 2 M20K, 9 DSP block

TABLE II. RESULTS AND COMPARISON WITH ALTERA

latency [1]. The method with Polynomial Approximation (d=2) plus Newton Raphson algorithm [10] has the fastest overall latency. It also uses significant memory resources compared to the other designs.

VI. CONCLUSIONS AND FUTURE WORK

We have presented a double precision floating point divider implementation that produces hardware with a good tradeoff of frequency, latency and resource usage and that can be implemented on both Altera and Xilinx FPGAs. Our design makes use of the embedded multipliers and embedded RAMs commonly found in modern FPGA fabric. It is flexible, with the designer able to adjust the maximum number of clock cycles.

In the future, we plan to investigate our divider further. In particular, we will focus on improving the frequency by focusing on optimizing the critical path and trying different levels of pipelining. In addition we plan to adapt our implementation to support variable precision division on both Altera and Xilinx FPGAs, and to make it available as part of the VFLOAT library [9], [16].

REFERENCES

[1] J. Detrey and F. de Dinechin. A tool for unbiased comparison between logarithmic and floating-point arithmetic. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 49(1):161–175, 2007.

[2] M. D. Ercegovac and T. Lăng. *Digital arithmetic*. Morgan Kaufmann, 2003.

[3] M. D. Ercegovac, T. Lang, J.-M. Muller, and A. Tisserand. Reciprocation, square root, inverse square root, and some elementary functions using small multipliers. *Computers, IEEE Transactions on*, 49(7):628–637, 2000.

[4] C. Freiman. Statistical analysis of certain binary division algorithms. *Proceedings of the IRE*, 49(1):91–103, 1961.

[5] R. Goldberg, G. Even, and P.-M. Seidel. An fpga implementation of pipelined multiplicative division with ieee rounding. In *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pages 185–196. IEEE, 2007.

[6] P. Hung, H. Fahmy, O. Mencer, and M. J. Flynn. Fast division algorithm with a small lookup table. In *Signals, Systems, and Computers, 1999. Conference Record of the Thirty-Third Asilomar Conference on*, volume 2, pages 1465–1468. IEEE, 1999.

[7] Institute of Electrical and Electronics Engineers (IEEE). 754-2008 — IEEE standard for floating-point arithmetic. *IEEE*, pages 1–58, 2008.

[8] M. K. Jaiswal and R. C. Cheung. High performance reconfigurable architecture for double precision floating point division. In *Reconfigurable Computing: Architectures, Tools and Applications*, pages 302–313. Springer, 2012.

[9] Northeastern University Reconfigurable Computing Laboratory. Vfloat: The northeastern variable precision floating point library. http://www.coe.neu.edu/Research/rcl/projects/floatingpoint/index.html.

[10] B. Pasca. Correctly rounded floating-point division for dsp-enabled fpgas. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 249–254. IEEE, 2012.

[11] J. Robertson. A new class of digital division methods. *Electronic Computers, IRE Transactions on*, EC-7(3):218–222, 1958.

[12] E. Roesler and B. Nelson. Novel optimizations for hardware floating-point units in a modern fpga architecture. In *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 637–646. Springer, 2002.

[13] P. Soderquist and M. Leeser. Area and performance tradeoffs in floating-point divide and square-root implementations. *ACM Computing Surveys (CSUR)*, 28(3):518–564, 1996.

[14] P. Soderquist and M. Leeser. Division and square root: choosing the right implementation. *Micro, IEEE*, 17(4):56–66, 1997.

[15] X. Wang, S. Braganza, and M. Leeser. Advanced components in the variable precision floating-point library. In *Field-Programmable Custom Computing Machines, 2006. FCCM'06. 14th Annual IEEE Symposium on*, pages 249–258. IEEE, 2006.

[16] X. Wang and M. Leeser. Vfloat: A variable precision fixed-and floating-point library for reconfigurable hardware. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 3(3):16, 2010.