Open-Source Variable-Precision Floating-Point Library for Major Commercial FPGAs

XIN FANG and MIRIAM LEESER, Northeastern University

There is increased interest in implementing floating-point designs for different precisions that take advantage of the flexibility offered by Field-Programmable Gate Arrays (FPGAs). In this article, we present updates to the Variable-precision FLOATing Point Library (VFLOAT) developed at Northeastern University and highlight recent improvements in implementations for implementing reciprocal, division, and square root components that scale to double precision for FPGAs from the two major vendors: Altera and Xilinx. Our library is open source and flexible and provides the user with many options. A designer has many tradeoffs to consider including clock frequency, total latency, and resource usage as well as target architecture. We compare the generated cores to those produced by each vendor and to another popular open-source tool: FloPoCo. VFLOAT has the advantage of not tying the user's design to a specific target architecture and of providing the maximum flexibility for all options including clock frequency and latency compared to other alternatives. Our results show that variable-precision as well as double-precision designs can easily be accommodated and the resulting components are competitive and in many cases superior to the alternatives.

CCS Concepts: • **Hardware** → **Arithmetic and datapath circuits; Reconfigurable logic and FPGAs; Hardware accelerators**; • **Mathematics of computing** → *Arbitrary-precision arithmetic*

Additional Key Words and Phrases: FPGA, floating point, variable precision, cross-platform

ACM Reference Format:

Xin Fang and Miriam Leeser. 2016. Open-source variable-precision floating-point library for major commercial FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 9, 3, Article 20 (July 2016), 17 pages.

DOI: <http://dx.doi.org/10.1145/2851507>

1. INTRODUCTION

Support for floating-point operations in Field-Programmable Gate Arrays (FPGAs) has become increasingly popular over the years. While many users are only interested in IEEE single- or double-precision floating-point formats, others use the flexibility available on FPGA fabric to customize their operations to specific data widths and intermediate sizes of floating-point numbers [Paschalakis and Lee 2003; Underwood 2004; Zhuo and Prasanna 2007; Wang and Leeser 2007, 2009]. FPGAs provide a flexibility not available on other platforms and allow designers to make tradeoff decisions between resources used and precision of the output, among others. Nonstandard precisions can potentially increase the stability of some numerical algorithms by computing with wider mantissa values. In applications where the data has a large dynamic range but precision is less important, large exponent ranges and narrower mantissa fields can be used.

This work was supported in part by the National Science Foundation grants CCF-1218075 and CNS-1337854, and by gifts from Altera and Xilinx.

Authors' addresses: X. Fang and M. Leeser, Department of Electrical and Computer Engineering, Northeastern University, 360 Huntington Ave, Boston, MA 02115; emails: fang.xi@husky.neu.edu, mel@coe.neu.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

2016 Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 1936-7406/2016/07-ART20 \$15.00

DOI: <http://dx.doi.org/10.1145/2851507>

At Northeastern University, we have developed the Variable Precision FLOATing Point Library (VFLOAT) [Wang and Leeser 2010; NEU Reconfigurable Computing Lab 2014]. VFLOAT supports basic floating-point operations such as addition and multiplication, more advanced components including accumulation and multiply accumulate, and format conversion operators to convert from fixed to floating point and back again. In this article, we present the library; highlight new implementations of division, reciprocal, and square root components [Fang and Leeser 2013; Fang 2013]; and present new results targeting both variable-precision and double-precision IEEE formats. Components in the VFLOAT library are extremely flexible. The number of bits of mantissa and exponent can be specified by the user on an individual component basis. Pipeline components can be combined in such a way that normalization and rounding are not implemented after every step. Thus, a user can potentially save area in applications that contain large adder trees, where extra bits can be added to the datapath instead of normalizing and rounding after each step, trading off one use of FPGA resources for another.

VFLOAT works with the development environments for the two major FPGA commercial vendors: Altera and Xilinx. All components are deeply pipelined, and the VFLOAT library gives the user many options for developing floating-point implementations for their designs. The level of pipelining is determined by the underlying components used and set by the user. A large range of latencies can be implemented for each of the components in the VFLOAT library.

The VFLOAT library is based on the binary IEEE 754-2008 standard floating-point format [Institute of Electrical and Electronics Engineers (IEEE) 2008]. Since VFLOAT is variable precision, many more mantissa and exponent sizes than those specified in the standard are supported. However, the basic format of sign bit, biased exponent, and mantissa are followed. VFLOAT does not comply with every detail of the standard. For example, subnormal numbers are not supported, and two rounding modes are supported: `roundTowardZero` and the `roundTiesToEven` directional mode for round to nearest. This latter mode is the default IEEE rounding mode and the most commonly used. Note that these simplifications are common to many embedded designs and to FPGA floating-point libraries. For example, both Xilinx and Altera only support the `roundTiesToEven` rounding mode, and neither supports subnormals.

Altera has recently introduced hard-core floating-point implementations for single-precision addition and multiplication [Altera.com 2014b]. This new hard-core implementation is testament to the fact that users are increasingly interested in floating-point applications being realized on FPGAs. Note that these cores are for single precision only, and it is difficult to implement single-precision division and square root with these components, as more bits of precision may be needed before rounding.

The contributions of this article are as follows:

- Open-source implementations of reciprocal, division, and square root based on Taylor series that scale well with a large number of mantissa bits and are available in the VFLOAT library
- Results for VFLOAT components targeting both Altera and Xilinx hardware, and a comparison with the vendors' built-in IP cores
- A comparison of VFLOAT components to the FloPoCo Floating Point Compiler [De Dinechin and Pasca 2011] and a demonstration that VFLOAT offers more flexibility and in some cases higher performance

The rest of the article is organized as follows. In the next section, we provide more details on the IEEE floating-point format and present related work. In Section 3, we give an overview of the VFLOAT library, followed by more details of the division, reciprocal, and square root implementations in Section 4. Then we compare our components to other FPGA floating-point libraries in Section 5 and present conclusions and future

Fig. 1. IEEE floating-point format.

work. All designs described in this article are available open source [NEU Reconfigurable Computing Lab 2014] and build on the OpenFabric platform [Chiou et al. 2014].

2. BACKGROUND

2.1. IEEE Floating-Point Representation

The VFLOAT library is based on the binary IEEE 754-2008 standard floating-point format [Institute of Electrical and Electronics Engineers (IEEE) 2008]. Since VFLOAT is variable precision, many more mantissa and exponent sizes than those specified in the standard are supported. However, the basic format of sign bit, biased exponent, and mantissa are followed, as shown in Figure 1. In VFLOAT, any number of exponent or mantissa bits can be specified, while in the IEEE format these are defined to be certain values for different fixed representations, including single and double precision.

The floating-point format has three parts to represent the numeric value: the sign bit, the exponent, and the mantissa. For base b , sign bit s , biased exponent e , and fractional part of the mantissa c , the value of a floating-point number is $(-1)^s * 1.c * b^{e-bias}$. The IEEE specification describes base 2 and base 10 implementations; we support base 2. Since a normalized mantissa always has the value 1.c, the 1 is not explicitly represented; however, it must be included in any computation. The bias for an exponent represented with w bits is $2^{(w-1)} - 1$. The IEEE standard specifies specific formats with number of exponent and mantissa bits equal to (8,23) for single precision and (11,52) for double. VFLOAT supports these as a subset, as well as any number of exponent and mantissa bits starting from 2 bits. We have tested up to double-precision values, meaning exponents up to 11 bits and mantissas up to 52 bits.

The IEEE standard specifies two directional attributes for rounding to nearest (roundTiesToEven, roundTiesToAway) as well as three directed rounding attributes (roundTowardPositive, roundTowardNegative, and roundTowardZero). An implementation that complies with the spec will implement all three directed rounding attributes as well as roundTiesToEven. VFLOAT supports roundTiesToEven and RoundTowardZero. Note that the Altera and Xilinx floating-point libraries only support RoundTiesToEven, which is the default rounding mode.

The IEEE standard also defines several exceptions and special values, including NaN (Not a Number) and positive and negative infinity. The VFLOAT implementation supports these special values by propagating them through the pipeline and generating new values where appropriate. For example, divide by zero and taking the square root of a negative number will generate the appropriate values to be propagated in the pipeline.

2.2. Related Work

There has been an increased interest in floating-point implementations in FPGAs, due to the increased amount of logic available on reconfigurable hardware including logic blocks and embedded multipliers and adders, as well as the relatively low power

consumption of FPGAs compared to other alternatives such as Graphics Processing Units (GPUs).

Both major FPGA vendors provide their own floating-point IP cores. These are called MegaCores for Altera [Altera.com 2014a] and IP Cores for Xilinx [Xilinx.com 2015]. Note that these are tied to the vendor's chips and lock a designer in to that design flow. Both support addition/subtraction, multiplication, division, and conversion between fixed- and floating-point representation. Altera has the richest set of operators, which include transcendental and trigonometric functions. Some of these components have restrictions. For example, Xilinx's reciprocal only supports single and double precision. For Altera MegaCores, only a few latencies are available for the designs we are interested in for this article. We compare VFLOAT components to these vendor cores in the results sections.

The work most similar to VFLOAT is FloPoCo [De Dinechin and Pasca 2011], which is the more recent version of the FPLibrary [Detrey and De Dinechin 2006]. FloPoCo is a generator of arithmetic cores for FPGAs, as opposed to a library of operators. It allows the user to target both Altera and Xilinx hardware. It is written in C++ and inputs operator specifications and outputs synthesizable VHDL code. Note that, while FloPoCo's representation of floating-point numbers is inspired by the IEEE standard, it differs in some key aspects. The main difference is that two leading bits are added to each word to signal special cases such as exceptions and NaNs. This requires that the user convert each floating-point value before using this tool. With the FSF AGPL license, FloPoCo is provided open source to the public. We compare our results to the results obtained by using the FloPoCo compiler in Section 5.

There have been several papers in the past describing floating-point libraries and implementations of floating-point components [Govindu et al. 2005]. A group from the University of Politecnia in Madrid [Echeverría and López-Vallejo 2011] describes a library of components including addition, multiplication, division, and square root as well as exponential and logarithm. They do not fully implement the IEEE specification; they treat subnormals as zero and implement truncation but not the round-to-nearest rounding mode. They compare their results to the Xilinx cores and show their components can support a high clock frequency with low hardware usage. Since they implement digit recurrence algorithms for division and square root, they require a large number of clock cycles to produce a result. Their library does not appear to be open source.

A group from Brazil has described a parameterizable floating-point library [Sánchez et al. 2009; Muñoz et al. 2010]. They support addition, multiplication, division, and square root and use Goldschmidt's algorithm for division and square root. Their library is parameterizable by bit width and by number of clock cycles of latency. Their multiplier has a one-cycle latency and, as a result, their clock frequency is low. Their library does not appear to be open source.

Several methods are available for implementing floating-point reciprocal and division [Ercegovac and Lang 2003]. The most common methods are digit recurrence [Ercegovac and Lang 1994; Lee and Burgess 2002; Wang and Nelson 2003], iterative or multiplicative methods, and table-based methods.

The digit recurrence method for division has two steps. First, a number is chosen based on the quotient digit selection function, which is decided before operation. The second step is to update the quotient remainder based on the selected number. Steps one and two are repeated until the required precision has been reached. For the most common digit selection method, 1 bit of result is generated each time these two steps are performed. This division method is similar to the paper-and-pencil shift and subtract algorithm and is the most common hardware implementation for division. More sophisticated digit selection functions can generate more than 1 bit of the solution at

Fig. 2. VFLOAT library components.

a time. One of the most popular approaches is the SRT algorithm [Robertson 1958; Freiman 1961].

Iterative or multiplicative methods [Roesler and Nelson 2002; Goldberg et al. 2007; Soderquist and Leiser 1996, 1997] are another popular way to implement division. These are based on multiplications that generate intermediate results iteratively and converge to the number of bits required after a fixed number of iterations. Newton Raphson is one of the most popular iterative methods. While the digit recurrence method generates 1 bit per recurrence, iterative methods generate multiple bits per iteration and the output converges to the required result. The number of times needed to iterate is determined by the desired precision. A drawback of iterative methods is that they are not easy to pipeline unless the iterations are unrolled. A recent paper [Pasca 2012a] presents division algorithms based on Newton Raphson and piecewise polynomial approximations and compares them to digit recurrence for both single- and double-precision floating-point division. The division algorithms implemented are correctly rounded according to the IEEE FP specification. An alternative to all these is table-based algorithms; this is the approach used in the VFLOAT library.

Square root implementations use the same methods: digit recurrence, iterative or multiplicative, or table based. Most libraries implement the digit recurrence method, which, like division, takes the smallest area but requires the largest number of clock cycles. A recent article presents multiplicative implementations of square root for FPGAs [De Dinechin et al. 2010] based on piecewise polynomial approximations and compares their performance to digit recurrence algorithms. We compare our implementation to this approach in the results section.

3. VFLOAT

In this section, we go into more detail about the structure and organization of the VFLOAT library. In this article, we focus on new algorithms for division, reciprocal, and square root implemented since our previous publication on the VFLOAT library [Wang and Leiser 2010]. Figure 2 lists all the components currently supported in VFLOAT.

3.1. Library Component Features

All of our components are designed to be combined in a pipelined manner. Figure 3 shows the signals provided in each component to support this. The input port(s) OP(s) are the input operand(s). These are provided from the previous component(s) in the pipeline, along with a ready signal to show that the inputs are available. When the

Fig. 3. Generic component.

Table I. Parameterized Floating-Point Components

Component	Latency
Absolute Value	0
Priority Encoder	0
Shifter	0
Adder	0
Comparator	0
Denormalizer	0
Multiplier	Adjustable
Round and Normalize	2

component has completed operations and the result is available on the RESULT output, DONE is set to true. The ROUND input specifies the rounding mode. If ROUND is 1, the rounding mode used is roundTiesToEven. If ROUND is 0, the rounding mode is roundTowardZero.

VFLOAT provides support for exceptions. If an exception is detected (divide by zero, square root of a negative number), EXCEPTION_OUT is set to true. Exceptions are then propagated along with the value for the rest of the floating-point pipeline, through the EXCEPTION_IN and EXCEPTION_OUT signals.

The definition of each component contains a width in number of bits for the exponent and mantissa, and these can be set in the top-level component by changing the parameters.

VFLOAT components are built from a set of general parameterized components, some of which are listed in Table I. The goal here is to reuse components and to make the library as flexible as possible. The library is designed to make use of the best performance of the underlying target hardware, including fast carry chains and hardware multipliers if they are available. The latencies are also listed in Table I. Most of the building blocks are combinational and therefore have a latency of zero. The multipliers used are based on the IP cores provided by each vendor; their latency can be adjusted when the IP core is chosen.

Divide, reciprocal, and square root components use table-based algorithms, and thus require table entries. These are provided in a folder named TABLE as part of VFLOAT. This folder contains memory initialization files for both Altera (.mif) and Xilinx (.coe) formats. There are three folders, r_table, sqrt_m_table, and sqrt_m_mul2_table. These tables have been tested for precisions from 2 bits up to double-precision range.

Fig. 4. Variable-precision reciprocal and division.

4. VARIABLE-PRECISION DIVISION, RECIPROCAL, AND SQUARE ROOT

In this section, we describe our implementation of the reciprocal, division, and square root components, and compare them to other approaches. Experimental results are provided in Section 5.

4.1. VFLOAT Table-Based Reciprocal and Division

In the VFLOAT library, we make use of table-based methods for reciprocal and division. These have the advantage of delivering faster solutions than digit recurrence algorithms and being easier to pipeline than iterative methods. The main idea is to store a value as an initial approximation to the result and to use variations on Taylor series expansion to obtain the required bits of precision.

Previous versions of the VFLOAT library [Wang and Leiser 2010; Wang et al. 2006] made use of an algorithm that required two multipliers and one lookup table [Hung et al. 1999]. For a single-precision floating-point division implementation, the size of the first multiplier is 24 by 24 bits of input with 48 bits of output; the second multiplier has inputs of 28 by 28 bits with an output of 56 bits. The lookup table has 12 bits for input and 28 for output, for a total size of approximately 16K bytes. However, the disadvantage of this algorithm is that the size of the lookup table increases exponentially with the bit width of the inputs. It is therefore impractical to implement a double-precision division with this approach since the number of lookup tables required would be prohibitively large. For this reason, we have abandoned this approach for a table-based approach that scales better with the bit width of the operands.

In the current version of VFLOAT, we adopt an improved algorithm [Ercegovac et al. 2000] that requires a smaller lookup table and several small multipliers and scales well with the size of the divisor. Figure 4 shows the hierarchy of components for the variable-precision division and reciprocal implementations. In the next paragraphs are descriptions of both the reciprocal and the division implementations, which are from Ercegovac et al. [2000]. To find the reciprocal $1/Y$ or the quotient X/Y , the algorithm requires three steps: reduction, evaluation, and postprocessing. In general, the reduction and evaluation steps are similar for both reciprocal and division. The difference is within the postprocessing step, where for division the dividend X is multiplied to the result of the reciprocal. In Figure 4, the portion of computation that corresponds to the reciprocal is indicated by a dashed-line frame.

Reduction step. After representing the hidden 1 bit, the fractional part of the floating-point divisor Y lies in the range $1 \leq Y < 2$. 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. For the purpose of faithful rounding, the m value is larger than that of the reciprocal by one. Define R as the reciprocal of $Y^{(k)}$. R can be determined by checking the lookup table, $M = R$ for computing the reciprocal and divider. For example, in double precision based on the previous equation, $m = 53$ and $k = 14$. So R can be determined using one lookup table with a 14-bit address. The number of bits for the lookup table output for R is the number of input bits plus 2 or, in this example, 16. To implement this lookup table, the user needs to create a single-port ROM IP core, whose input width is k and output is $k + 2$. The memory initialization file is generated ahead of time.

Evaluation step. B is defined as the Taylor series expansion of

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

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

$$A = A_2z^2 + A_3z^3 + A_4z^4 + \dots,$$

where $|A_i| \leq 2^k - 1$. The smaller terms of this expansion will not contribute to the final result and can be safely ignored.

Using the Taylor series expansion,

$$\begin{aligned} B = f(A) &= C_0 + C_1A + C_2A^2 + C_3A^3 + C_4A^4 + \dots \\ &\approx C_0 + C_1(A_2z^2 + A_3z^3 + A_4z^4) \\ &\quad + C_2(A_2z^2 + A_3z^3 + A_4z^4)^2 \\ &\quad + C_3(A_2z^2 + A_3z^3 + A_4z^4)^3 \\ &\quad + C_4(A_2z^2 + A_3z^3 + A_4z^4)^4 \\ &\approx C_0 + C_1A + C_2A_2^2z^4 + 2C_2A_2A_3z^5 + C_3A_2^3z^6. \end{aligned}$$

Here, $C_i = 1$ when i is even, and $C_i = -1$ when i is odd. After simplifying:

$$\begin{aligned} B = f(A) &\approx 1 - A_2z^2 - A_3z^3 + (-A_4 + A_2^2)z^4 \\ &\quad + 2A_2A_3z^5 - A_2^3z^6 \\ &\approx (1 - A) + A_2^2z^4 + 2A_2A_3z^5 - A_2^3z^6. \end{aligned}$$

In Figure 4, four multipliers must be instantiated: multiplier YR for multiplication of Y and R , which determines A ; two multipliers S for computing both A_2^2 and $A_3 \cdot A_2$; and multiplier M for computing A_2^3 . Their input bit widths are $4 \cdot k + 2$ and $k + 3$ for multiplier YR ; two k bit inputs for multiplier S ; and $2 \cdot k$ and k bits for multiplier M .

Postprocessing step. The final result is given by multiplication. For reciprocal, the result of the reciprocal of Y is given by the product of M and B : $1/Y = M \times B$. For division, the result of division X/Y is given by the product of M , B , and X : $X/Y = M \times B \times X$.

In Figure 4, multiplier L executes $M \cdot B$ with input widths $k + 2$ and $4 \cdot k + 2$. For division, an additional multiplier XY is required to perform the final step and its input bit widths are $m + 1$ and $m + 5$.

The total latency for each of the reciprocal and division modules is variable and can be changed by adjusting the latency of the multiplier IP cores used. The only difference between the steps of creating the reciprocal and division is that one more multiplier IP core is added in the postprocessing step for division; when considering the total latency, this multiplier's delay should also be included. As shown in Figure 4, for reciprocal, there are four multipliers used, and for division, five. Each multiplier can have their

Fig. 5. Variable-precision square root.

latency adjusted. Both Xilinx and Altera allow users to choose the number of pipeline stages when the multiplier core is instantiated. There are many combinations of these latencies and different combinations will lead to different resource utilization, latency, and maximum clock frequency for the floating-point modules. For the reciprocal, the total latency is the sum of the latency of multiplier YR, 2*multiplier S, multiplier M, and multiplier L, plus an extra two clock cycles. The divider latency adds an additional delay of multiplier XY.

4.2. Variable-Precision Square Root

The VFLOAT square root implementation also uses a table-based method [Hung et al. 1999]. For computing square root, there are three steps, similar to the reciprocal computation: reduction, evaluation, and postprocessing. Figure 5 shows the hierarchy of components in the variable-precision square root implementation.

Reduction step. The difference between the reduction step of computing the square root and reciprocal is that after getting R, M is assigned to $1/\sqrt{R}$. So a different lookup table to compute the inverse square root of R is needed. If the exponent is odd, there will be $\sqrt{2}$ as a coefficient to multiply by the result in the last step; that is, we check the last bit of the exponent. If it is zero, the exponent is even, so we assign $1/\sqrt{R}$ to M. If not, M is assigned to $\sqrt{2}/\sqrt{R}$. To compute $\sqrt{2}/\sqrt{R}$, we create a separate lookup table.

Evaluation step. For the divider, $f(A)$ was defined as $1/(1 + A)$. For square root, let

$$f(A) = \sqrt{1 + A}.$$

The Taylor series expansion for $f(A)$ is the same as that for the reciprocal:

$$B = f(A) \approx C_0 + C_1A + C_2A_2^2z^4 + 2C_2A_2A_3z^5 + C_3A_2^3z^6.$$

The coefficient values are $C_0 = 1$, $C_1 = 1/2$, $C_2 = -1/8$, $C_3 = 1/16$. Thus,

$$B = f(A) = 1 + \frac{A}{2} - \frac{1}{8}A_2^2z^4 - \frac{1}{4}A_2A_3z^5 + \frac{1}{16}A_2^3z^6.$$

Here, to compute the polynomial expression, multipliers should be generated as shown in Figure 5. The three multipliers multiplier YR, multiplier S, and multiplier M have the same bit widths as for the reciprocal and division modules.

Postprocessing step. The final result of the square root is given by the product of M and B: $\sqrt{Y} = M \times B$, which is done by multiplier L.

Latency in square root. There are four multipliers that make up the square root component, as shown in Figure 5. Similar to reciprocal and division, the total latency of the square root is the sum of the latency of multiplier YR, $2 \times$ multiplier S, multiplier M, and multiplier L, plus an extra two clock cycles. As with division and square root, there are many combinations of these latencies, and different combinations will provide different resource, latency, and maximum clock frequency for the square root component. The minimum number of clock cycles occurs when the multiplier latencies are set to one and is seven clock cycles for square root.

4.3. Error Evaluation

We evaluated the error in the floating-point operations presented. In the evaluation step, ignoring some terms in the polynomials may introduce errors to the equations and the final results of the operation. Based on the error evaluation in Ercegovac et al. [2000], the upper bound of the absolute error for the reciprocal and square root are both smaller than 1 ulp (unit in the last place), resulting in a faithfully rounded output. However, the division module includes an extra multiplication, which will introduce an additional ulp of error, resulting in a total of 2 ulps. In order to guarantee an error smaller than 1 ulp and therefore faithful rounding, we increase the datapath for computing the reciprocal of the divisor by 1 bit compared to a reciprocal unit with the same input bit width for Y. This additional bit in computation of the divisor ensures faithful rounding for division modules, but only introduces a 1% increase in the corresponding FPGA resources.

5. RESULTS

Our implementations of variable-precision floating-point components are written in VHDL and target both popular commercial FPGA vendors: Altera and Xilinx. For Altera, we synthesize with the IDE tool Quartus 14.0 and target the Stratix V device in this article. For Xilinx, we use Xilinx IDE 14.6 and target a Virtex 6 FPGA. The designs make use of embedded multipliers and RAMs, which are the intellectual property components provided with each set of tools. For Altera, these are called MegaCores; for Xilinx, IP Cores.

Both Altera and Xilinx FPGAs have similar resources, including configurable logic elements, DSP blocks, and embedded memories. For Xilinx, we target the Virtex 6 FPGA family, introduced in 2009, specifically XC6VLX75T-3ff484. We quote results in terms of registers, LUTs, and DSPs used. Each Virtex-6 FPGA slice contains four 6-input LUTs and eight flip-flops. Note that some of the LUTs might be used for registers or shift registers. Each DSP48E1 slice contains a 25×18 multiplier, an adder, and an accumulator. Block RAMs are 36Kbits in size. For Altera, we target the Stratix V 5SGXB6, introduced in 2010, specifically 5SGXEB6R2F40C2ES. Altera calls their configurable logic elements Adaptable Logic Modules (ALMs). Each ALM has eight inputs with a fracturable LUT, two embedded adders, and four dedicated registers. DSP blocks can be configured as a 27×27 bit multiply accumulate or dual 18×18 multiply accumulates. Embedded RAMs are 20Kbits. Results will vary depending on the particular device targeted.

Our experimental results compare reciprocal, division, and square root implementations targeting each of the major vendors. Previously we have published results for double-precision floating-point implementations [Fang and Leeser 2013]; here, we focus on variable precision as well as presenting updated double-precision results to illustrate that our library components scale well. For each target chip, we compare results generated by the vendor's own IP tools, FloPoCo and VFLOAT. All the results shown are based on the output of the design tools and are postsynthesis results. Thus, quantities such as maximum frequency are only an estimate. The actual maximum

clock frequency will change as a result of running placement and routing, and can depend on user settings.

We use different floating-point formats for the components targeting Altera and those targeting Xilinx. This is partly to show the variable precision nature of our library and partly because Altera provides restrictions on the range of values it allows. For Altera, we use 1 sign bit, 11 exponent bits, and 31 mantissa bits. For mantissa bit widths, Altera floating-point division and square root MegaCores support single (23), double (52), and single extended precision (31 to 51). Xilinx allows for a larger range of values. For the results targeting Xilinx, we use 1 sign bit, 9 exponent bits, and 30 mantissa bits. For all experiments, we present number of clock cycles latency, maximum frequency, and total latency in nanoseconds. Note that this is a multidimensional optimization space, and in most cases it is difficult to identify the *best* design. For example, the component with the fastest clock frequency may have a long total latency, consume more energy than a slower clock frequency, and be difficult to combine into a larger pipeline. We point out advantages and disadvantages of different approaches when there is not a clear winner.

VFLOAT allows users to set the number of cycles of delay in their designs by the way the underlying multipliers are implemented. After creating multipliers, the user should update the `float_pkg.vhd` file based on the multiplier latency parameters. This mechanism gives the user complete control over the depth of the pipeline, but is manual. In contrast, FloPoCo automatically determines the pipelining based on the frequency setting [De Dinechin and Pasca 2011], but the achievable latency values are not continuous. Altera's floating-point core also has limited choice for pipelining options. As a result, our components can exhibit a range of latencies, while the number of pipeline stages is restricted in other cases.

We have tested all VFLOAT components for correctness. Our components are guaranteed to be faithfully rounded [Pasca 2012b] and return results with an error guaranteed to be 1 ulp. Although not guaranteed, in most cases our results are *correctly rounded* according to the IEEE standard, which is defined as the floating-point number closest to the infinitely accurate result that meets the rounding mode requirements. As our components are variable precision, testing is more complicated than if only IEEE standard formats are used. For IEEE standard formats, we compare our results with the online IEEE-754 analysis calculator [Lubow and Vickery 2014] as well as verifying them using the SoftFloat package [Hauser 2015]. For variable-precision components, we check VFLOAT results after transforming the variable-precision to double-precision floating-point representation. In addition, we translate the binary values obtained for variable precision to decimal and use MATLAB's variable-precision arithmetic package to check the results. The testbench vectors are randomly generated by MATLAB code and include positive and negative values. In addition, we generate test cases manually to ensure that test cases contain exceptions such as divide by 0 and square root of a negative number. All inputs and outputs have been tested.

5.1. Division Results

Variable-precision division results. Variable-precision floating-point division results targeting the Altera Stratix V are shown in Table II. Note that we can adjust the number of pipeline stages for the VFLOAT divider based on the underlying multipliers. We instantiate two versions of the divider with the FloPoCo library. FloPoCo does not allow the user to arbitrarily set the number of clock cycles; we chose to show two of the generated designs. Altera MegaCores are from the Altera IP library. It can generate dividers with eight, 18, or 41 stages, but no values in between. We choose eight and generate a VFLOAT divider with an eight-cycle latency for comparison purposes. We are unable to generate an eight-cycle divider with FloPoCo. Note that the two 8-cycle

Table II. Variable-Precision Division Results and Comparison for Altera (11 Exp Bits, 31 Mantissa Bits)

Method	Clock Cycles	Max Freq.	Total	Resources
VFLOAT	8	210MHz	38ns	405 ALMs, 396 Reg, 9 DSP, 5,788 Block memory bits
FloPoCo 1st	11	162MHz	68ns	1,250 ALMs, 1,435 Reg, 520 Block memory bits
FloPoCo 2nd	20	285MHz	70ns	1,391 ALMs, 2,218 Reg
Altera MegaCore	8	237MHz	34ns	223 ALMs, 491 Reg, 8 DSP

Table III. Variable-Precision Division Results and Comparison for Xilinx (9 Exp Bits, 30 Mantissa Bits)

Method	Clock Cycles	Max Freq.	Total	Resources
VFLOAT	8	176MHz	45ns	523 Reg, 1,820 Slice LUTs, 1 BRAM, 7 DSP
FloPoCo	11	204MHz	54ns	979 Reg, 1,746 Slice LUTs
Xilinx IP Core 1st	8	146MHz	55ns	455 Reg, 1,222 Slice LUTs
Xilinx IP Core 2nd	10	181MHz	55ns	577 Reg, 1,224 Slice LUTs

Table IV. Double-Precision Division Results and Comparison for Altera

Method	Clock Cycles	Max Freq.	Total	Resources
VFLOAT 1st	8	113MHz	71ns	785 ALMs, 867 Reg, 13 DSP, 2,058,308 Block memory bits
VFLOAT 2nd	10	150MHz	67ns	791 ALMs, 946 Reg, 13 DSP, 2,031,722 Block memory bits
FloPoCo 1st	17	139MHz	122ns	3,326 ALMs, 3,147 Reg, 631 Block memory bits
FloPoCo 2nd	31	273MHz	114ns	3,504 ALMs, 5,676 Reg, 631 Block memory bits
Altera MegaCore	10	197 MHz	51 ns	512 ALMs, 1,244 Reg, 14 DSP, 4,608 Block memory bits

implementations are very competitive, with similar total latency, clock frequency, and hardware resources used. The advantage of VFLOAT compared with MegaCores is that it does not only target Altera hardware. The FloPoCo divider does not use any DSP blocks and achieves a higher clock frequency at the cost of longer total latency.

Table III shows the variable-precision division results targeting the Xilinx Virtex 6. Xilinx IP cores support continuous latency for division. We generate two different designs with a differing number of clock cycles. Comparing the four designs, FloPoCo once again generates the fastest clock cycle but not the shortest total latency. Our eight-cycle design has the shortest total latency and an intermediate clock frequency. It uses more resources than the Xilinx cores, including block RAM and DSP blocks not used by the other implementations.

Double-precision division results. Double-precision floating-point division results targeting the Stratix V are shown in Table IV. Note that the VFLOAT components scale well. Comparing to FloPoCo, VFLOAT uses many fewer ALMs at the expense of DSP blocks and block memory bits. FloPoCo latencies are also longer. The VFLOAT and Altera results are competitive; VFLOAT uses more ALMs and block memory bits, while Altera uses more registers. The Altera divider MegaCore generates a smaller total latency with a 10-clock-cycle delay. However, Altera only supports three latency options (10, 24, and 61), while VFLOAT is more flexible.

Table V shows the double-precision results targeting the Xilinx Virtex 6. The Xilinx synthesis tool has an option for using DSP blocks, which we set to “auto.” In this case, the synthesis tools chose not to use DSP blocks for any of the target architectures. The FloPoCo-generated component has a long total latency and a high usage of registers and slices. The Xilinx IP core requires a slow clock and a long total latency, thus making it more difficult to include in a larger design. The VFLOAT library component

Table V. Double-Precision Division Results and Comparison for Xilinx

Method	Clock Cycles	Max Freq.	Total	Resources
VFLOAT 1st	8	140MHz	57ns	830 Reg, 3,487 Slice LUTs, 8 BRAM
VFLOAT 2nd	10	177MHz	56ns	1,833 Reg, 3,540 Slice LUTs, 8 BRAM
FloPoCo	31	180MHz	172ns	2,509 Reg, 4,368 Slice LUTs
Xilinx IP Core	7	55MHz	127ns	761 Reg, 3,226 Slice LUTs

Table VI. Variable-Precision Reciprocal Results and Comparison for Altera and Xilinx (11 Exp Bits, 31 Mantissa Bits)

Method	Device	Latency	Max Freq.	Total	Resources
VFLOAT	Stratix V	7	210MHz	33ns	283 ALMs, 430 Reg, 7 DSP, 45,204 Block memory bits
Altera MegaCore	Stratix V	20	385MHz	52ns	407 ALMs, 1,306 Reg, 6 DSP
VFLOAT 1st	Virtex 6	7	159MHz	44ns	567 Reg, 1,693 Slice LUTs, 1 BRAM
VFLOAT 2nd	Virtex 6	9	192MHz	47ns	818 Reg, 1,690 Slice LUTs, 1 BRAM

Table VII. DoublePrecision Reciprocal Results and Comparison for Xilinx

Method	Clock Cycles	Max Freq.	Total	Resources
VFLOAT 1st	7	141MHz	50ns	832 Reg, 3,449 Slice LUTs, 8 BRAM
VFLOAT 2nd	10	179MHz	56ns	1,752 Reg, 3,483 Slice LUTs, 8 BRAM
Xilinx IP Core	7	112MHz	63ns	344 Reg, 298 Slice LUTs, 14 DSP

outperforms the other alternatives, with low total latency, good clock frequency, and low resource utilization.

5.2. Reciprocal Results

Table VI shows combined results targeting both Altera and Xilinx devices for variable-precision reciprocal components. Here the format used for all target designs is 11 bits of exponent and 31 bits of mantissa. There is no reciprocal component in FloPoCo, so those results are the same as for division. The reciprocal MegaCore in Altera has an unchangeable delay of 20 clock cycles independent of the size of the value. As a result, the total latency is higher than the other alternatives, with similar resource utilization to the others.

The Xilinx reciprocal IP core only supports single- and double-precision formats. Table VII compares the double-precision reciprocal results targeting the Virtex 6 from VFLOAT and Xilinx. VFLOAT obtains the lowest total latency with a higher clock frequency than the Xilinx core. No DSP blocks are used, which is decided by Xilinx synthesis tools, so the VFLOAT slice count is higher as a tradeoff.

5.3. Square Root Results

Variable-precision square root results. Variable-precision square root results targeting the Altera Stratix V are shown in Table VIII. We compare VFLOAT with two kinds of alternative implementations from FloPoCo, FPSqrt, and FPSqrtPoly, as well as the Altera MegaCore for square root. FPSqrt claims to guarantee correctly rounded results and FPSqrtPoly faithfully rounded. FPSqrtPoly, floating-point square root using polynomial approximation [De Dinechin et al. 2010], allows the user to choose the polynomial degree when generating the module. Once again, with VFLOAT, there is a continuous choice in the number of clock cycles. For Altera MegaCores, the only options are 20 and 36 clock cycles for latency. In this case, the fastest clock cycle was achieved with Altera MegaCores, but the best total latency was FPSqrtPoly with degree 4 in FloPoCo. VFLOAT obtains an intermediate clock frequency, a competitive

Table VIII. Variable-Precision Square Root Results and Comparison for Altera (11 Exp Bits, 31 Mantissa Bits)

Method	Clock Cycles	Max Freq.	Total	Resources
VFLOAT	10	174MHz	57ns	525 ALMs, 971 Reg, 9 DSP, 43,008 Block memory bits
FPSqrt 1st	9	195MHz	46ns	588 ALMs, 536 Reg
FPSqrt 2nd	7	136MHz	51ns	627 ALMs, 438 Reg
FPSqrtPoly(degree=4)	11	274MHz	40ns	285 ALMs, 944 Reg, 5 DSP, 54,272 Block memory bits
Altera MegaCore	20	300MHz	67ns	403 ALMs, 852 Reg, 238 Block memory bits

Table IX. Variable-Precision Square Root Results and Comparison for Xilinx (9 Exp Bits, 30 Mantissa Bits)

Method	Clock Cycles	Max Freq.	Total	Resources
VFLOAT	11	191MHz	57ns	528 Reg, 776 Slice LUTs, 8 DSP, 3 BRAM
Xilinx IP Core	11	178MHz	62ns	367 Reg, 785 Slice LUTs
FPSqrt	8	172MHz	47ns	335 Reg, 684 Slice LUTs
FPSqrtPoly(degree=4)	11	210MHz	52ns	348 Reg, 382 Slice LUTs, 5 DSP, 3 BRAM
FPSqrtPoly(degree=8)	51	230MHz	222ns	1,456 Reg, 1519 Slice LUTs, 15 DSP, 7 BRAM

Table X. Double-Precision Square Root Results and Comparison for Altera

Method	Clock Cycles	Max Freq.	Total	Resources
VFLOAT	7	164MHz	43ns	530 ALMs, 850 Reg, 12 DSP, 2,001,828 Block memory bits
FPSqrt	7	87MHz	80ns	1740 ALMs, 629 Reg, 266 Block memory bits
FPSqrtPoly(degree=4)	14	240MHz	58ns	582 ALMs, 993 Reg, 13 DSP, 200,704 Block memory bits
Altera MegaCore	30	223MHz	135ns	955 ALMs, 1,986 Reg, 426 Block memory bits

total latency, and uses the fewest ALMs and fewer block memory bits compared with FPSqrtPoly, but uses more DSP blocks.

Table IX shows the variable-precision square root component comparison targeting the Xilinx Virtex 6 devices. FPSqrtPoly can achieve the highest frequency with large resource utilization as a tradeoff. Other results on the table are quite close in terms of maximum frequency, total latency, and FPGA resource.

Double-precision square root results. Double-precision floating-point square root results targeting the Stratix V are shown in Table X. Once again, VFLOAT is competitive. The Altera square root MegaCore supports only two latency options, 30 or 57, while VFLOAT is more flexible. The VFLOAT component has the shortest total latency and matches the number of clock cycles generated by FloPoCo at a higher clock frequency. Once again, this is at the cost of DSP blocks, which FloPoCo does not use, and a large number of block memory bits.

Table XI shows the double-precision square root results targeting Virtex 6. Here the VFLOAT library generates a square root component with the lowest total latency at the cost of larger FPGA resource usage compared with the Xilinx IP core and FloPoCo. It also achieves the fastest clock cycle, making it easier to include this design in larger pipelines.

6. CONCLUSIONS AND FUTURE WORK

We have presented new implementations for variable-precision division, reciprocal, and square root that target both Xilinx and Altera FPGAs. The new algorithm scales

Table XI. Double-Precision Square Root Results and Comparison for Xilinx

Method	Clock Cycles	Max Freq.	Total	Resources
VFLOAT 1st	7	145MHz	48ns	1,546 Reg, 6,243 Slice LUTs, 58 BRAM
VFLOAT 2nd	10	179MHz	56ns	4,025 Reg, 6,285 Slice LUTs, 58 BRAM
Xilinx IP Core	10	89MHz	112ns	695 Reg, 1,855 Slice LUTs
FPSqrt	7	91MHz	77ns	489 Reg, 1,741 Slice LUTs
FPSqrtPoly	14	173MHz	81ns	840 Reg, 1,093 Slice LUTs, 13 DSP, 6 BRAM

better for larger bit widths; we show results for 30- and 31-bit as well as 52-bit (double-precision) mantissas. We compare our results to the Altera and Xilinx tools and show that our implementations are competitive with both while being able to target either hardware platform. We also compare our results to FloPoCo. Our library is written in VHDL and is easy for the user to modify. It gives the user the ability to change the number of clock cycles to implement these components and is more flexible than other alternatives. FloPoCo is an open-source module generator, which operates at a higher level but gives the user less control over the design, and the floating-point format is not the IEEE-754 standard representation. VFLOAT maintains a good balance between number of clock cycles latency, maximum clock frequency, throughput, and FPGA resource utilization. All the control input and output signal ports provide a seamless interconnection for any component within or beyond the VFLOAT library. VFLOAT implementations are competitive with competing approaches and are open source.

We plan to continue to support the VFLOAT library and focus on improving clock frequency and resource utilization. Using the same methodology, we can implement a reciprocal square root component in the future. This is a common arithmetic operation in many machine-learning applications. Other modules that could benefit from a similar approach using Taylor series include trigonometric and elementary functions; we plan to investigate this in the future. We also plan to investigate higher-level tools for composing VFLOAT components.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their suggested improvements to this article.

REFERENCES

- Altera.com. 2014a. Floating-Point IP Cores User Guide. Retrieved May 26, 2015, from http://www.altera.com/literature/ug/ug_altpf_mfug.pdf.
- Altera.com. 2014b. The Industry's First Floating-Point FPGA. Retrieved May 26, 2015, from <http://www.altera.com/literature/po/bg-floating-point-fpga.pdf>.
- Derek Chiou, Arvind, Joel Emer, James Hoe, Shih-Lien Lu, and John Wawrzynek. 2014. The OpenFabric Platform. Retrieved May 26, 2015, from <http://openfabric.org/>.
- Florent De Dinechin, Mioara Joldes, Bogdan Pasca, and Guillaume Revy. 2010. Multiplicative square root algorithms for FPGAs. In *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications (FPL'10)*. IEEE, 574–577.
- Florent De Dinechin and Bogdan Pasca. 2011. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers* 28, 4 (2011), 18–27.
- Jérémie Detrey and Florent De Dinechin. 2006. A VHDL library of parametrisable floating-point and LNS operators for FPGA. (2006). Retrieved May 26, 2015, from <http://www.ens-lyon.fr/LIP/Arenaire/Ware/FPLibrary/>.
- Pedro Echeverría and Marisa López-Vallejo. 2011. Customizing floating-point units for FPGAs: Area-performance-standard trade-offs. *Microprocessors and Microsystems* 35, 6 (2011), 535–546.
- Miloš D. Ercegovac and Tomas Lang. 1994. *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers.
- Miloš D. Ercegovac and Tomas Lang. 2003. *Digital Arithmetic*. Elsevier.

- Miloš D. Ercegovac, Tomás Lang, J.-M. Muller, and Arnaud Tisserand. 2000. Reciprocatation, square root, inverse square root, and some elementary functions using small multipliers. *IEEE Transactions on Computers* 49, 7 (2000), 628–637.
- Xin Fang. 2013. *Variable Precision Floating Point Reciprocal, Division and Square Root for Major FPGA Vendors*. Master's thesis. Northeastern University Dept. of ECE, Boston, MA. Retrieved May 26, 2015, from <http://www.coe.neu.edu/Research/rcf/publications.php#theses>.
- Xin Fang and Miriam Leeser. 2013. Vendor agnostic, high performance, double precision floating point division for FPGAs. In *Proceedings of the 2013 IEEE High Performance Extreme Computing Conference (HPEC'13)*. IEEE, 1–5.
- C. V. Freiman. 1961. Statistical analysis of certain binary division algorithms. *Proceedings of the IRE* 49, 1 (1961), 91–103.
- Ronen Goldberg, Guy Even, and Peter M. Seidel. 2007. An FPGA implementation of pipelined multiplicative division with IEEE Rounding. In *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2007 (FCCM'07)*. IEEE, 185–196.
- Gokul Govindu, Ronald Scrofano, and Viktor K. Prasanna. 2005. A library of parameterizable floating-point cores for FPGAs and their application to scientific computing. In *Proceedings of the International Conference on Engineering Reconfigurable Systems and Algorithms, 2005*. Citeseer, 137–148.
- John Hauser. 2015. SoftFloat. (2015). Retrieved May 26, 2015, from <http://www.jhauser.us/arithmetic/SoftFloat.html>.
- Patrick Hung, Hossam Fahmy, Oskar Mencer, and Michael J. Flynn. 1999. Fast division algorithm with a small lookup table. In *Conference Record of the 33rd Asilomar Conference on Signals, Systems, and Computers, 1999*. Vol. 2. IEEE, 1465–1468.
- Institute of Electrical and Electronics Engineers (IEEE). 2008. 754-2008 — IEEE Standard for Floating-Point Arithmetic. (2008).
- Barry Lee and Neil Burgess. 2002. Parameterisable floating-point operations on FPGA. In *Proceedings of the Conference Record of the 36th Asilomar Conference on Signals, Systems and Computers, 2002*. Vol. 2. IEEE, 1064–1068.
- M. Lubow and C. Vickery. 2014. IEEE-754 Analysis. Retrieved from <http://babbage.cs.qc.cuny.edu/IEEE-754/FAQ/>.
- Daniel M. Muñoz, Diego F. Sanchez, Carlos H. Llanos, and Mauricio Ayala-Rincón. 2010. Tradeoff of FPGA design of a floating-point library for arithmetic operators. *Journal of Integrated Circuits and Systems* 5, 1 (2010), 42–52.
- NEU Reconfigurable Computing Lab. 2014. VFLOAT: The Northeastern Variable Precision FLOATing Point Library. (2014). Retrieved May 26, 2015, from <http://www.coe.neu.edu/Research/rcf/projects/floatingpoint/index.html>.
- Bogdan Pasca. 2012a. Correctly rounded floating-point division for DSP-enabled FPGAs. In *Proceedings of the 2012 22nd International Conference on Field Programmable Logic and Applications (FPL'12)*. IEEE, 249–254.
- Bogdan Pasca. 2012b. Correctly rounded floating-point division for DSP-enabled FPGAs. In *Proceedings of the 2012 22nd International Conference on Field Programmable Logic and Applications (FPL'12)*. IEEE, 249–254.
- Stavros Paschalakis and Peter Lee. 2003. Double precision floating-point arithmetic on FPGAs. In *Proceedings of the 2003 IEEE International Conference on Field-Programmable Technology, 2003 (FPT'03)*. IEEE, 352–358.
- James E. Robertson. 1958. A new class of digital division methods. *IRE Transactions on Electronic Computers* EC-7, 3 (1958), 218–222. DOI: <http://dx.doi.org/10.1109/TEC.1958.5222579>
- Eric Roesler and Brent Nelson. 2002. Novel optimizations for hardware floating-point units in a modern FPGA architecture. In *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*. Springer, 637–646.
- Diego F. Sánchez, Daniel M. Muñoz, Carlos H. Llanos, and Mauricio Ayala-Rincón. 2009. Parameterizable floating-point library for arithmetic operations in FPGAs. In *Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design: Chip on the Dunes*. ACM, 40.
- Peter Soderquist and Miriam Leeser. 1996. Area and performance tradeoffs in floating-point divide and square-root implementations. *ACM Computing Surveys (CSUR)* 28, 3 (1996), 518–564.
- Peter Soderquist and Miriam Leeser. 1997. Division and square root: Choosing the right implementation. *IEEE Micro* 17, 4 (1997), 56–66.
- Keith Underwood. 2004. FPGAs vs. CPUs: Trends in peak floating-point performance. In *Proceedings of the 2004 ACM / SIGDA 12th International Symposium on Field Programmable Gate Arrays*. ACM, 171–180.

- Xiaojun Wang, Sherman Braganza, and Miriam Leeser. 2006. Advanced components in the variable precision floating-point library. In *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2006. (FCCM'06)*. IEEE, 249–258.
- Xiaojun Wang and Miriam Leeser. 2007. K-means clustering for multispectral images using floating-point divide. In *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2007 (FCCM'07)*. IEEE, 151–162.
- Xiaojun Wang and Miriam Leeser. 2009. A truly two-dimensional systolic array FPGA implementation of QR decomposition. *ACM Transactions on Embedded Computing Systems (TECS)* 9, 1 (2009), 3.
- Xiaojun Wang and Miriam Leeser. 2010. VFloat: A variable precision fixed-and floating-point library for reconfigurable hardware. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 3, 3 (2010), 16.
- Xiaojun Wang and Brent E. Nelson. 2003. Tradeoffs of designing floating-point division and square root on virtex FPGAs. In *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003 (FCCM'03)*. IEEE, 195–203.
- Xilinx.com. 2015. Floating-Point Operator. (2015). Retrieved May 26, 2015, from http://www.xilinx.com/products/intellectual-property/FLOATING_PT.htm.
- Ling Zhuo and Viktor K. Prasanna. 2007. Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems. *IEEE Transactions on Parallel and Distributed Systems* 18, 4 (2007), 433–448.

Received December 2014; revised May 2015; accepted November 2015