A Library of Parameterizable Floating-Point Cores for FPGAs and Their Application to Scientific Computing[1]

Gokul Govindu*, Ronald Scrofano†, and Viktor K. Prasanna*
*Department of Electrical Engineering
†Department of Computer Science
University of Southern California
Los Angeles, CA
{govindu, rscrofan, prasanna}@usc.edu

Abstract— **Advances in field programmable gate arrays (FPGAs), which are the platform of choice for reconfigurable computing, have made it possible to use FPGAs in increasingly many areas of computing, including complex scientific applications. These applications demand high performance and high-precision, floating-point arithmetic. Until now, most of the research has not focussed on compliance with IEEE standard 754, focusing instead upon custom formats and bitwidths. In this paper, we present double-precision floating-point cores that are parameterized by their degree of pipelining and the features of IEEE standard 754 that they implement. We then analyze the effects of supporting the standard when these cores are used in an FPGA-based accelerator for Lennard-Jones force and potential calculations that are part of molecular dynamics (MD) simulations.**

I. Introduction

The state-of-the-art devices for reconfigurable computing are FPGAs. The density of FPGAs has been steadily increasing. In addition to increased density, FPGA vendors are increasing the amount of embedded features within the FPGAs. These are dedicated hardware features that increase the capability of the FPGAs. One example is the embedded 18-bit $\times$ 18-bit multipliers in Xilinx Virtex-II Pro FPGAs [1]. The increased logic and the features embedded into the FPGA fabric have made FPGAs a promising technology to be exploited by the computer science and computer engineering communities. Already, FPGAs have provided fantastic performance increases in areas such as signal processing, cryptography, and embedded computing.

FPGAs' past successes in application acceleration coupled with their increasing densities have led to the development of several FPGA-based reconfigurable computers, including the SRC 6e [2] and the Cray XD1 [3]. These machines are comprised of both general purpose processors (GPPs) and FPGAs, as well as the interconnect between them. The FPGAs act as programmable application accelerators for the GPPs. Importantly, the target applications for these machines are in scientific, rather than embedded, computing. Scientific computing applications require high-precision, floating-point arithmetic. There have been several efforts into developing floating-point cores for FPGAs, all of which have shortcomings. For instance, none of the existing floating-point cores are compliant with IEEE standard 754 [4]. Consequently, in the reconfigurable computing community, there is no general understanding of the resource and performance overheads that arise from adhering to the standard. Also, in commercial reconfigurable computing platforms such as those from SRC and Cray, FPGA-based accelerators interface with fully IEEE-compliant GPPs. Users would find it inconvenient to have to consider the IEEE compliance, or lack thereof, of the FPGA-based designs and the corresponding effect on the results. On the other hand, if the users know that they do not need full compliance, they should have the option of only implementing in hardware the features that they need.

In this paper, we present a library of floating-point cores developed for FPGAs to support all types of numbers representable, all types of exceptions that can be generated, all rounding modes, and the NaN diagnostics that are described in IEEE standard 754, and to do so in double precision (64-bit). We have developed floating-point cores for addition/subtraction, multiplication, division, and square root. These cores are parameterized by degree of pipelining and by the features of IEEE standard 754 that are implemented. We present the performance data for three configurations of the cores. Then, we show how implementing various features of IEEE standard 754 affects the performance of a kernel in molecular dynamics simulations. Note that we do not assess the effects on accuracy of implementing only selected features of IEEE standard 754. Rather we assess the effects on throughput; we leave it up to users of the cores to determine what features they need to implement in order to obtain the accuracy desired. The library (VHDL source code and compliance testbenches) is available to the community through the World Wide Web at `http://indus.usc.edu/fpus`.

In the next section, we describe most of the existing FPGA floating-point cores and contrast them with the cores that we describe here. We also briefly introduce IEEE standard 754. Then, in Section III, we describe our floating-point cores

[1]This work is supported by Los Alamos National Laboratory under contract/award number 95976-001-04 3C.

and analyze their performance. In Section IV, we describe means of analysis to determine the best floating-point core configuration and degree of pipelining for a given situation. In Section V, we describe the use of our floating-point cores in a scientific computing application and analyze the effects of varying levels IEEE compliance. Finally, in Section VI, we summarize our work and present our plans for future work.

II. Related Work

Some early work in floating-point arithmetic for FPGAs was done in [5], [6], and [7], among others. These early works, however, were targeted to more primitive FPGAs than are available today. Specifically, these FPGAs lacked the large amount of logic resources and the embedded arithmetic units that today's state-of-the-art FPGAs have. Thus, in these works, nothing more than 32-bit precision is ever considered, the area available for the floating-point cores is severely limited, and the clock frequencies are rather low. Much of the work dealt with variations in the arithmetic algorithms and their resource implications. Further, only addition and multiplication are implemented; there is no implementation for division or square root. Consequently, we do not focus on these early attempts but instead look into more recent investigations of floating-point arithmetic on FPGAs.

[8] is one of the earliest works about the implementation of floating-point cores for modern FPGAs. In this work, a parameterized floating-point library is presented. The library contains addition and multiplication cores as well as cores for the conversion from (to) floating-point format to (from) fixed-point format. These cores are parameterized by the bit-width of the floating-point numbers. In [9], this work is extended to include a division core and a square root core. Like the other cores, these are parameterized by bit width.

Another early work implementing floating-point cores for FPGAs is [10]. This work describes a parameterized floating-point library and uses this library to implement the SPH force calculation algorithm on a Virtex-II FPGA. The library has addition, multiplication, division, and square root cores. The performance of each floating-point core for up to 24 bits of precision is listed. The FPGA implementation of SPH using the floating-point cores with 24-bit precision was able to achieve a performance of 3.9 GFLOPS.

[11] presents a floating-point unit generator that can generate a large space of floating-point adders, multipliers, and dividers based on a variety of parameters. Tradeoffs at the architectural level, the algorithm level, and the representation level are analyzed. The floating-point unit generator is implemented as part of a C++ to FPGA configuration bitstream compiler. In the code, users can specify whether cores optimized for area, throughput, or latency should be selected. Tradeoffs in area and latency are analyzed, but only for bit widths less than or equal to 32 bits.

[12] describes floating-point adders and multipliers that follow the IEEE-standard-754 single- and double-precision formats, as well as a 48-bit format. It is important to note that while these cores represented the floating-point numbers as specified in IEEE standard 754, they did not implement all the features of the standard. For example, denormal numbers were not supported. Nonetheless, this work was the first to develop FPGA floating-point cores specifically designed for such large bit widths. [13] uses these adders and multipliers in a high performance floating-point matrix multiplication design. This design is able to achieve a performance of 8.3 GFLOPS. Furthering the work of [12], a floating-point divider is developed in [14]. The divider, along with the previously developed adder and multiplier, are used to perform matrix factorization on an FPGA. [15] develops a 64-bit, floating-point square root core to go with the cores developed in [12] and [14]. All of these cores are used in a design for Lennard-Jones force and potential calculation. This design achieves a 41% or greater performance improvement over state-of-the-art GPPs performing the same calculations.

[16] presents a high-level analysis comparing the trends in peak floating-point performance of GPPs and FPGAs and concludes that the peak FPGA floating-point performance is growing significantly faster than that of GPPs. To do these projections, the author presents floating-point adder, multiplier, and divider cores for both single precision and double precision. These cores implement most features of IEEE standard 754, but leave out the exception generation and all rounding modes but round to nearest. The adder is parameterized by pipeline depth. The multiplier and divider are parameterized by pipeline depth and denormal support. The multiplier has an additional parameter to set whether or not embedded multipliers in the FPGA are utilized. [17] then uses these floating-point cores to study the performance of FPGA-based implementations for three linear algebra kernels from the BLAS library: vector dot product, matrix-vector multiplication, and matrix multiplication. This work concludes that FPGAs can significantly outperform GPPs in executing these kernels. Further, it predicts that the performance gap between FPGAs and GPPs will widen in the future.

The work that we present in this paper differs in important ways from the related work listed above. The floating-point cores that we have developed are the first developed for FPGAs to support all types of numbers representable, all types of exceptions that can be generated, all rounding modes, and the NaN diagnostics that are described in IEEE standard 754 and to do so in double precision (64-bit). Consequently, we are also the first to analyze the performance effects of supporting the standard when the floating-point cores are used in a scientific computing kernel.

A. IEEE Standard 754

Here, we give a brief overview of IEEE standard 754 (see [4] for details). IEEE standard 754 is a standard for binary floating-point arithmetic. It specifies four numerical formats: single precision, single extended precision, double precision, and double extended precision. Our floating-point cores implement double-precision arithmetic, so, from here out, we will only describe the parts of the standard that pertain to double-precision numbers.

Double-precision numbers are 64-bit values with three fields: a sign bit s, a biased exponent e (11 bits), and a fraction f (52 bits). There are five types of numbers: NaNs, $\pm\infty$, ± 0, normal numbers and denormal numbers. A number's type is determined by the values of its exponent and fraction.

IEEE standard 754 defines four rounding modes: round to nearest (the default), round toward 0, round toward $+\infty$, and round toward $-\infty$. All of the operations are to behave as if they produce an infinite precision result and then round that result using the specified mode.

IEEE standard 754 defines addition, subtraction, multiplication, division, remainder, and square root as operations, as well as several conversions between formats and comparisons. At this time, we have developed floating-point cores for addition/subtraction, multiplication, division, and square root. We have not developed cores for remainder or the conversion or comparison operations and thus do not describe those operations here. Arithmetic operations on infinities and NaNs are both defined in IEEE standard 754. Also, diagnostic information can be stored in the fractional part of NaNs. These "NaN diagnostics" should be propagated from a NaN that is input to an operation to the NaN that is the output.

Finally, IEEE standard 754 defines five types of exceptions: invalid operation, division by zero, overflow, underflow, and inexact. Note that IEEE standard 754 also defines trapping mechanisms which we do not implement.

III. Floating-Point Cores

In this section, we describe the implementation and analyze the performance of each of the floating-point cores. The cores have several parameters. One parameter is the degree of pipelining. The other parameters determine the level of compliance with IEEE standard 754—whether or not NaN diagnostics are supported, whether or not denormal numbers are supported, whether or not exceptions are generated, and which rounding modes are supported.

We present three configurations for analysis. The first configuration is the most compliant with IEEE standard 754. In it, NaN diagnostics, all four IEEE rounding modes, exception generation, and denormal numbers are supported. The "common features" configuration implements only the round to nearest rounding mode and support for exceptions but does not implement NaN diagnostics; denormal numbers are treated as zero. In the "lowest overhead" configuration, the only rounding mode implemented is round toward 0; denormal numbers are treated as zero, exceptions are not generated, and NaN diagnostics are not supported.

Each of the floating-point cores was coded in VHDL and synthesized with Synplicity Synplify Pro 7.2. The parameters determining the number of pipelining stages and the features supported are implemented as VHDL generics. The user sets the appropriate values before synthesis and only the necessary hardware is synthesized. In our experiments, the target device was the Xilinx Virtex-II Pro XC2VP7 FPGA. The place and route tool used was Xilinx PAR, part of Xilinx ISE 5.2 and ISE 6.2. Each of the floating-point cores can achieve a frequency of at least 140 MHz. We analyze the performance and area requirements of the floating-point cores when they are fully pipelined and, in Section IV, describe techniques for further analysis.

A. Testing

When implementing floating-point cores, especially ones that implement parts of IEEE standard 754, it is important to test the behavior of the cores for a variety of inputs. However, to the best of our knowledge, there is no existing, publicly available set of test vectors and expected outputs that tests compliance with parts of IEEE standard 754. Thus, we have developed one. Our philosophy in developing the test vectors for the floating-point cores is that if, for the same inputs, our floating-point cores give the same output as a floating-point unit (FPU) that complies with IEEE standard 754 does, a system comprised of our cores would also be compliant with the standard. So, in order to develop our test vectors, we needed an IEEE-standard-754-compliant FPU and inputs that test its compliance.

Intel claims that the FPU in the Itanium processor is IEEE-standard-754-compliant. In order to test this claim, we used the C version of the program "paranoia" [18]. This program tests the IEEE-standard-754 compliance of the FPU in a GPP. With the Intel C++ Compiler 8.1 for Linux, we compiled paranoia.c on a Hewlett-Packard rx2600 server with dual 900 MHz Intel Itanium 2 processors and 8.3 GB of memory, running the Red Hat Linux Advanced Server 2.1 operating system. When run on this system, paranoia reports

```
No failures, defects, nor flaws
have been discovered. Rounding
appears to conform to the proposed
IEEE standard P754. The arithmetic
diagnosed appears to be Excellent!
```

Thus, we used the Itanium's FPU as the compliant FPU against whose output we would check the output of our floating-point cores.

After testing the FPU, we edited paranoia to output the binary values for all the arguments to the arithmetic operations of interest to us. We wrote a C program that reads in these arguments and performs the desired computations. Our program outputs the arguments to the floating-point operators as stimulus for VHDL testbenches. It also outputs the VHDL code to check that that the results from our floating-point cores match those of the Itanium's FPU. Additionally, our program outputs VHDL testbench code to test if the exceptions generated by our floating-point cores are the same as those generated by the Itanium's floating-point unit.

To test the floating-point cores, we first simulate them behaviorally with ModelSim using the generated testbenches [19]. Once they are correct, we then place and route them and perform post-place-and-route simulations using the testbenches to ensure that the cores still perform correctly.

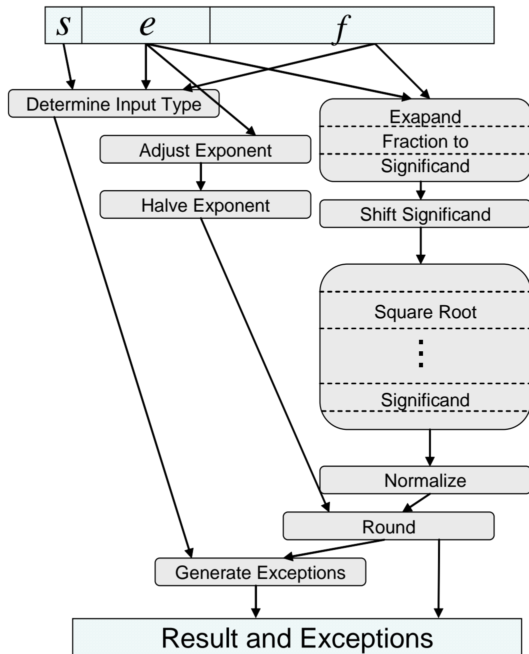

Fig. 1. Steps to perform floating-point square root. s, e, and f are the input sign bit, exponent, and fraction, respectively.

B. Square Root Core

The steps for performing the floating-point square root are shown in Figure 1 and detailed below.

Determine the type of input: The hardware examines the sign bit, exponent, and fraction of the input floating-point number to determine what type of number it is: positive normal number, positive denormal number, negative number, ± 0, $\pm\infty$, or quiet or signaling NaN. Determining the input type requires one clock cycle.

Expand the 52-bit fraction to the full 53-bit significand: If the number is a positive normal, the hidden 1 is prepended to f, making $1f$. If the number is denormal, a 0 is prepended to f, making $0f$. This step is done in parallel with the input determination. The input is then sent to a priority encoder that determines the number of places, if any, that the number must be left shifted. The shifting itself is done by a logarithmic shifter that takes two cycles.

Adjust the exponent, if necessary: The exponent must be adjusted if the input is denormal and will be left-shifted. The number of bits by which the input will be shifted must be subtracted from 1. This stage takes place after the priority encoder, in parallel with the first stage of the logarithmic shifter.

Halve the Exponent: The exponent of the result of the square root operation is half that of the original number. Halving the exponent takes one stage and can be done in parallel with the last stage of the logarithmic shifter.

Square root the significand: This step is the most compute- and resource-intensive of the floating-point square root algorithm. We first perform any necessary shifting to ensure that the value of the significand is in the range [1, 4). After doing so, this step is the same as taking a fixed-point square root. Many methods exist for taking a fixed-point square root. For example, in the square root core described in [9], the table lookup method for square root from [20] is used. In [20], it is determined that for double-precision floating-point square root, the size of the table would be 142 KB. While this is a fairly small size, it is still larger than the amount of embedded memory present in our target FPGA, the Xilinx Virtex-II Pro XC2VP7. Thus, in order to use this method, we would need to use logic resources for storage. Instead, we have chosen a different method.

In our square root core, the significand is square rooted using the non-restoring algorithm of Li and Chu [21]. This algorithm calculates the output one bit at a time, starting with the most significant bit. Each computation requires only an integer add or an integer subtract, depending on the bit generated in the prior stage. The bitwidth of the addition/subtraction grows by one bit for each new bit of the square root calculated. The output of the algorithm is not only the square root of the input, but also the remainder.

This algorithm has several features that make it attractive for our square root unit. It is easy to pipeline (one stage per bit generated) and requires only adder/subtractors, registers, and logical NOT gates. The exact remainder produced by the algorithm can be used in rounding toward $+\infty$ and in generating the inexact exception. The only drawbacks are that the adder/subtractors at the ending stages of the algorithm are quite large and there is a significant pipeline latency to produce the results. Square rooting the significand requires 54 clock cycles.

Normalize the result: Because in an earlier stage we ensured that the input significand was in the range $[1, 4)$, the output significand is in the range $[1, 2)$. A number in the range $[1, 2)$ is normalized, so no extra normalization step is needed.

Round the result The first step in rounding is to correct the remainder if it is negative. Correcting the remainder requires a 57-bit addition and takes one clock cycle. Round to the nearest is implemented as an integer addition of 1 to the full output of the square root algorithm. Rounding toward $+\infty$ is implemented similarly. If the (corrected) remainder is 0, the number is left as it is. If the (corrected) remainder is nonzero, an integer addition of 1 to the result of the square root algorithm except for the rounding bit is performed. In square root, all outputs that are not special numbers are positive. Consequently, round toward 0 and round toward $-\infty$ are equivalent. In both cases, all the bits generated by the square root algorithm, except for the rounding bit, are used as the result.

Output the final, correctly rounded result and raise any exceptions that have occurred: If the input type, determined in the first stage, was positive normal or positive denormal, the

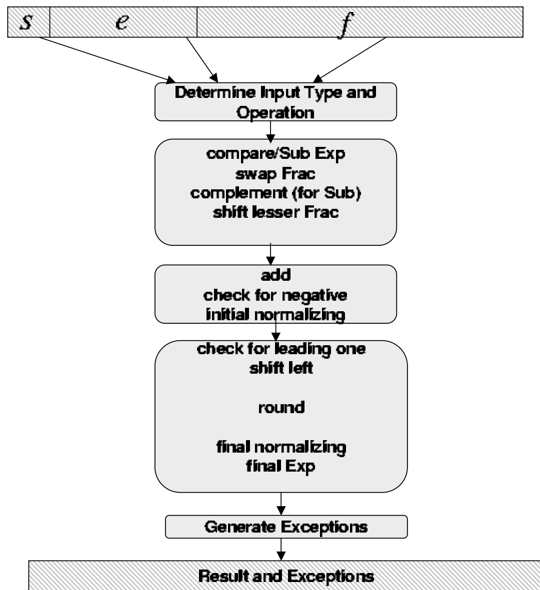

Fig. 2. Steps to perform floating-point addition-subtraction. s, e, and f are the input sign bit, exponent, and fraction, respectively.

correctly rounded result is output. Otherwise, the appropriate special number is output.

Of the five types of exceptions described in IEEE standard 754, it is only possible for square root to generate two of them: invalid operation and inexact result. Under the appropriate conditions, these exception flags are raised. The core also passes along any exceptions that were generated by other floating-point cores and given as input to it.

1) Performance Analysis: Table I shows the properties of the square root core in the three configurations described above when it is fully pipelined. From the table, we can see that the configurations vary little in maximum frequency. Reducing the features of IEEE standard 754 that are implemented gives a small improvement in pipeline latency. The common features configuration has three fewer pipeline stages than the most compliant case because the common features case lacks denormal support: the priority encoder and logarithmic shifter, and their three cycles of pipeline latency, are removed. The lowest overhead configuration has two fewer cycles than that because it additionally removes the cycle for correcting the remainder and the cycle for the addition that is part of round to nearest and round toward $+\infty$. Clearly, the greatest difference between the configurations is the amount of area that each requires. The most compliant configuration requires 40% more area than the lowest overhead configuration and 28% more area than the common features case. In analyzing all possible configurations of the square root core, we find that removing the support for denormals leads to the most area savings: 262 slices, on average.

C. Addition/Subtraction Core

The steps for performing the floating-point addition-subtraction are shown in Figure 2 and detailed below.

Similiar to the square root core, the addition-subtraction core initially has hardware to determine the the type of input. The unpacking of the hidden bit depends upon the input type: whether it is zero, denormal, or normal. The two numbers are compared and an arithmetic right shift must be performed on the lesser of the two numbers. If the operation is a subtract, the two's complement of the subtrahend is taken before the shift occurs. The number of bits of the shift is dependent upon the difference in the exponents. The shifting itself is done by a logarithmic shifter that takes three or two cycles depending upon the level of compliance. For example, in the least-overhead version of the cores, we need to obtain only the guard bit from the bits shifted out. However, for the fully compliant version we need to obtain guard, round and sticky bits.

A preliminary result is calculated by using an adder. The result is checked to see if there was a negative result and is replaced with two's complement if required. The result is shifted to the right by one place if the most significant bit is a zero and the exponent is adjusted accordingly. The preliminary normalization of the result is done by checking for a leading one using a priority encoder and the result is shifted left, if required. Here, the fully compliant core needs to shift until a denormal is obtained. For the other versions, the result can be flushed to zero. The rounding is carried out based upon the rounding mode, the sign, the round and sticky bits and the least significant bit of the result. The exponent is adjusted accordingly and result is checked for underflow or overflow. Finally the fraction, the exponent, and the sign are packed together as the output. The exceptions are checked at the final stage of rounding.

Table II shows the properties of the adder/subtractor core in the two configurations described above when it is fully pipelined. We see that compliance in the form of obtaining exceptions and supporting denornmals does require a lot more area. However, compared to the multiplier and the divider, in order to support denormal numbers, the adder/subtractor doesn't change much in terms of functionality.

D. Multiplication Core

The steps for performing the floating-point multiplication are shown in Figure 3 and detailed below.

The lowest overhead version of the multiplier is the simplest floating-point core to implement. However the most compliant multiplier has to handle denormals which increase the area overhead along with the handling of zeros, infinities and NaNs and supporting exceptions.

The numbers are initially unpacked after checking whether the number was a denormal or a normal. The exponents are added to obtain a tentative exponent. The fractions are multiplied together to obtain the result, the round, guard and sticky bits. An initial normalizing is done with a left shift, depending upon the most significant bit of the result. If the result is a denormal, it will have leading zeros. Supporting denormals is the functionality that requires the most area additional area when compared to the lowest overhead multiplier core.

TABLE I
PROPERTIES OF THE FULLY PIPELINED SQUARE ROOT CORE

Configuration	No. of Stages	Maximum Frequency (MHz)	Area (slices)	% of FPGA's Total Area	Frequency/Area (MHz/1000 slices)
Most Compliant	60	164	2332	47	70.3
Common Features	57	164	1826	37	89.8
Lowest Overhead	55	169	1666	34	101.4

TABLE II
PROPERTIES OF THE FULLY PIPELINED ADDER CORE

Configuration	No. of Stages	Maximum Frequency (MHz)	Area (slices)	% of FPGA's Total Area	Frequency/Area (MHz/1000 slices)
Most Compliant	17	170	1640	33	103
Common Features	17	170	1580	31	107
Lowest Overhead	14	170	1078	21	157

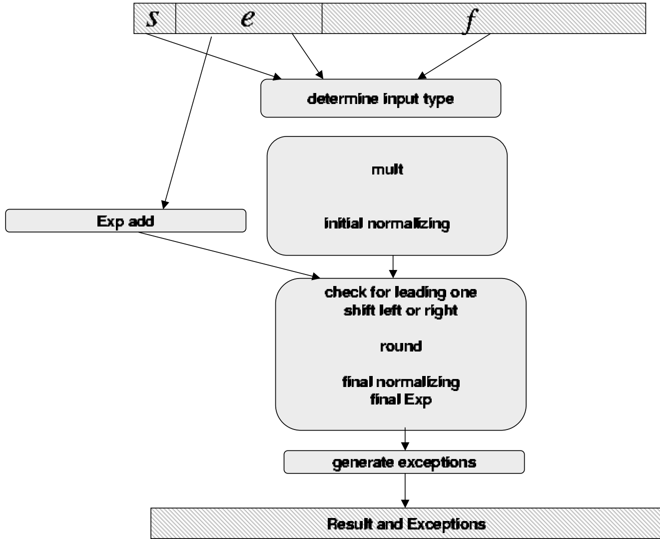

Fig. 3. Steps to perform floating-point multiplication. s, e, and f are the input sign bit, exponent, and fraction, respectively.

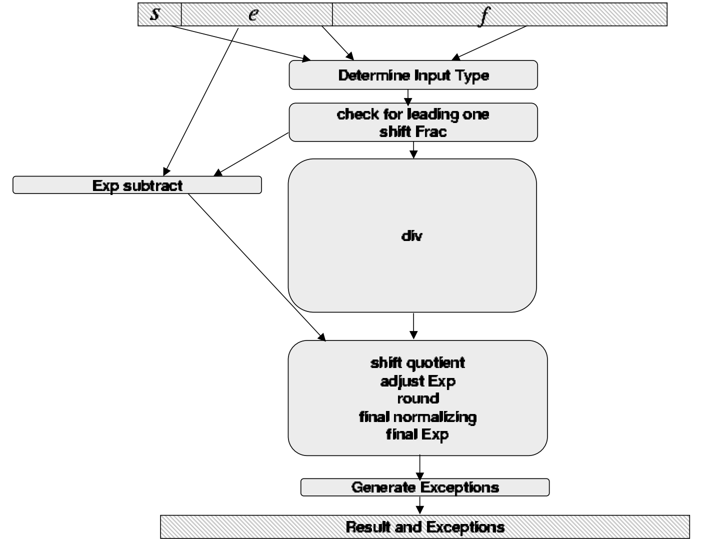

Fig. 4. Steps to perform floating-point division. s, e, and f are the input sign bit, exponent, and fraction, respectively.

Depending upon the leading one and the tentative exponent, a right or a left shift might be required. This also means that the round and sticky bits have to be computed after the shifting is done. The tentative exponent is adjusted after determining whether there is overflow or not in the result.

Table III shows the properties of the multiplier core in the two configurations described above when it is fully pipelined. We see that handling denormals requires a lot more area overhead than exception handling.

E. Division Core

The steps for performing the floating-point division are shown in Figure 4 and detailed below.

The numbers are initially unpacked. The most compliant core requires a leading zero checker to check for denormals and a logarithmic shifter to normalize the denormals. If a fraction is a denormal, then it is shifted left to normalize it before sending it into the divider. Similar to the square rooter, the floating-point divider has a large, fixed-point divider core. This divider core has been built using both the SRT4 and the NRD algorithms (see below). The exponents are subtracted in parallel to the division, to give the tentative exponent. After division, depending upon the tentative exponent's value, the quotient is shifted left. The exponents are adjusted accordingly and finally the exceptions are set.

Table IV shows the properties of the division core in the two configurations described above when it is fully pipelined. The version using the NRD algorithm runs faster than the version using the SRT4 algorithm but with a greater number of pipeline stages. The version using the SRT4 algorithm gives a lower latency but has a lower F/A ratio because it runs at a lower speed and has a greater area.

TABLE III
PROPERTIES OF THE FULLY PIPELINED MULTIPLIER CORE

Configuration	No. of Stages	Maximum Frequency (MHz)	Area (slices)	% of FPGA's Total Area	Frequency/Area (MHz/1000 slices)
Most Compliant	19	170	2085	42	81
Common Features	12	170	1165	23	150
Lowest Overhead	11	170	935	19	181

TABLE IV
PROPERTIES OF THE FULLY PIPELINED DIVIDER CORE

Configuration	No. of Stages	Maximum Frequency (MHz)	Area (slices)	% of FPGA's Total Area	Frequency/Area (MHz/1000 slices)
Most Compliant (NRD)	68	140	4243	86	33
Common Features (NRD)	60	140	3625	73	38
Lowest Overhead (NRD)	58	140	3213	66	44
Lowest Overhead (SRT4)	32	90	3713	75	24

IV. UTILIZING THE PARAMETERS OF THE FLOATING-POINT CORES

Our floating-point cores provide the user with two types of parameters. The first is the features of IEEE standard 754 that are supported. Whether or not the user can make use of these parameters is dependent upon the application, the data on which it operates, and the numerical accuracy desired. For example, flushing denormals to zero is a latency optimization provided by many compilers, such as the Intel C++ Compiler 8.1. If the user would use this optimization for the software version of his application, then he need not implement denormal support in the hardware version of his application: doing so would increase the area significantly while not providing any benefit. As another example, if the user would only use round to nearest rounding mode in the software version, then he need not implement the other rounding modes in the hardware version. However, it is important that these features of IEEE standard 754 be available if they are needed. For example, if the user wants the accuracy of denormals, he should be able to have that accuracy in his hardware version of the application. The ability to implement only those features necessary is a great benefit of using reconfigurable hardware as the computing platform.

The other parameter provided is the degree of pipelining of the floating-point cores. The degree of pipelining affects both the frequency and the area of the design: pipelining is necessary to achieve a high frequency but it also leads to a larger area. Figure 5(a) shows the frequency of the lowest overhead square root core as a function of the number of pipelining stages. We use the lowest overhead square root core as an example, but the analysis technique applies to any of the cores and configurations. The degree of pipelining ranges from 1 pipeline stage to 55 pipeline stages. The frequency increases close to linearly with the number of pipeline stages. The rate of increase is about 3 MHz per pipeline stage. Note that once 50 pipeline stages are used, the increase in frequency levels off. In fact, it even dips slightly until the full 55 pipeline stages are used, which yields the maximum frequency.

Figure 5(b) shows the area of the square root core as a function of the degree of pipelining. The increase in area is also close to linear. The rate of increase is about 15 slices per pipeline stage. Note that in contrast to the frequency curve, the area curve is monotonically increasing. Clearly, there is a tradeoff between frequency and area: setting the degree of pipelining to achieve a high frequency implies a large area and setting the degree of pipelining for a small area implies a low frequency.

To further analyze this tradeoff, we have developed a third metric, the *frequency-to-area ratio*. The degree of pipelining that leads to the highest frequency-to-area ratio is the one that gives the most throughput per unit area. Any further pipelining will increase the area at a higher rate than it increases frequency. The frequency-to-area ratio for the square root core is shown in Figure 5(c). The highest frequency-to-area ratio occurs when the number of pipelining stages in the core is 51. Even though there is a slightly higher frequency when the unit is fully pipelined, the increase in frequency is overshadowed by the increase in area.

V. APPLICATION KERNEL

We demonstrate the utility of our cores by applying them to a scientific computing application. Molecular dynamics (MD) simulation is a technique for simulating the movement of particles in a system over time [22]. There are many kernels in an MD simulation, but the one that requires the bulk of the processing time is the force and potential calculation. There are many different types of forces and potentials; the Lennard-Jones force and potential is one of the most widely used. In [15], we developed a simple, pipelined architecture to calculate the Lennard-Jones force and potential in hardware. The dataflow graph for the calculation is shown in Figure 6. We give each operation in the dataflow graph its own floating-point core. In the architecture, there are eight floating-point

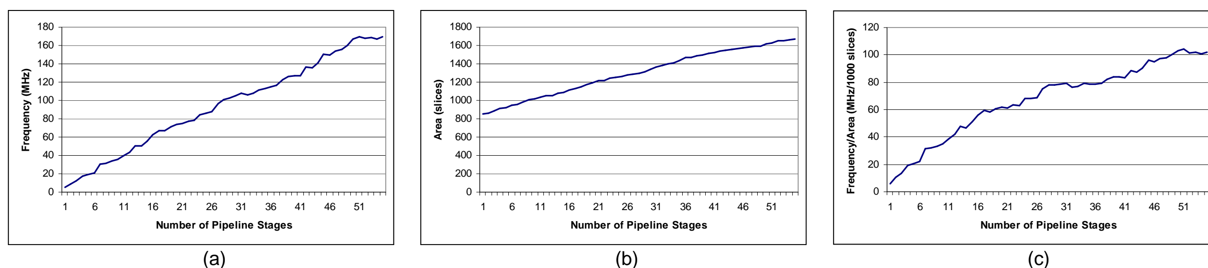

Fig. 5. Frequency (a), Area (b), and Frequency/Area ratio (c) as a function of the number of pipeline stages in the lowest overhead square root core

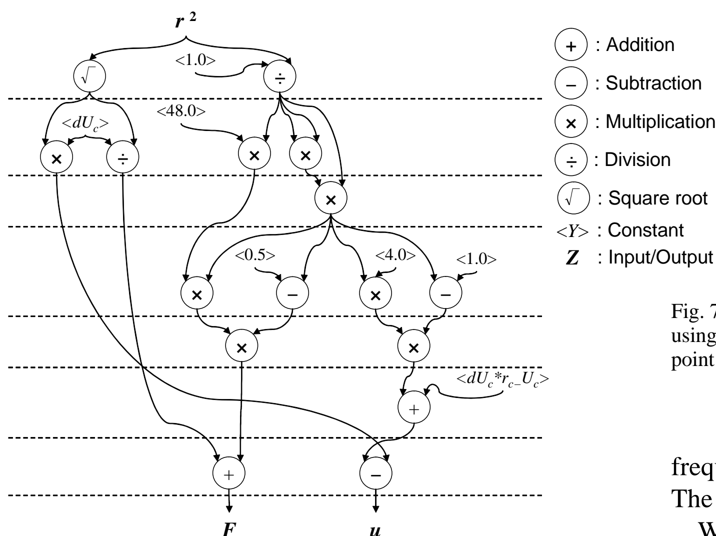

Fig. 6. Architecture for the Lennard-Jones force and potential calculation kernel

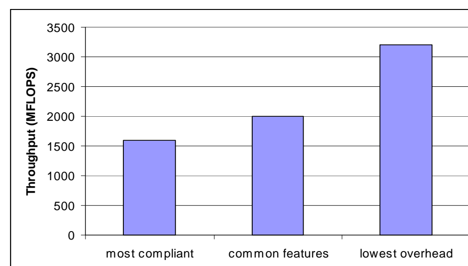

Fig. 7. Throughput of the Lennard-Jones force and potential calculation when using the most compliant, common features, and lowest overhead floating-point core configurations

multiplications, five floating-point addition/subtractions, two floating-point divisions, and one floating-point square root. The input is a double-precision number representing the square of the distance between two molecules and the outputs are one double-precision number representing a component of the Lennard-Jones force and another double-precision number representing the Lennard-Jones potential. The architecture is very deeply pipelined because it is comprised of the pipelined floating-point cores. However, once the pipeline is full, there are two outputs per clock cycle.

We coded this design in VHDL, utilizing our fully pipelined floating-point cores. We synthesized the design with Synplicity Synplify Pro 7.2 and placed-and-routed the design with Xilinx PAR, part of ISE 7.1. The target device was the Virtex-II Pro XC2VP100. Due to very long place-and-route times, the results for the implementation with fully compliant cores are only an estimate.

When using the lowest overhead configuration of the floating-point cores, it is possible to implement two of the force calculation pipelines in parallel on a single FPGA. The frequency achievable by such an implementation is 100 MHz. The total area of this implementation is 35315 slices.

When using the common features or the most compliant configurations of the floating-point cores, it is not possible to implement two of the force calculation pipelines in parallel on a single FPGA: the area required is too great. So, only a single force calculation pipeline is implemented. When using the common features configuration, the frequency achievable is 125 MHz and the area is 23864 slices. When using the most compliant configuration, the estimated frequency is 100 MHz and the estimated area is 35698 slices.

The graph in Figure 7 shows the throughput achievable by the implementations. Because of the parallelism, when the lowest overhead configuration of the floating-point cores is used, the throughput is 1.6 times that of when the common features configuration is used and is twice that of when most compliant configuration is used. This example demonstrates the dramatic difference in the performance with differently configured floating-point cores. Therefore, it is necessary to provide the user with these parameters so that he can utilize those configurations which best suit his performance and numerical accuracy needs.

VI. Conclusion

We have presented the first floating-point cores for FPGAs that support all types of numbers representable, all types of exceptions that can be generated, all rounding modes, and the NaN diagnostics that are described in IEEE standard 754 and do so in double precision (64-bit). We have also described our

methodology for testing that the floating-point cores function correctly. We detail the performance of the three configurations of our floating-point cores: the lowest overhead configuration, which supports the fewest features of IEEE standard 754; the common features configuration, which supports the features of the standard that are commonly used in applications; and the most compliant configuration, which supports most of the features of the standard. Our analysis shows that the main difference between the configurations is in the area required by their implementations. The most compliant configuration is much larger than the other two. On the other hand, the frequencies of the configurations are about the same.

We then described techniques for further analysis of the configurations, such as using the frequency-to-area ratio to determine the degree of pipelining to use for the cores. Finally, we compared the various floating-point core configurations when used in the force and potential calculation kernel for MD. We saw that implementation utilizing the lowest overhead configuration allowed for the exploitation of parallelism because of the comparatively small size of the floating-point cores.

In the future, our main interest is in applying these floating-point cores to other kernels. Already, work is under way in the area of linear algebra. We would also like to develop hardware designs for the calculations of other forces used in MD simulations. Doing so may require the development of floating-point cores for other operations such as trigonometric functions.

Acknowledgments

The authors would like to thank Maya Gokhale and Frans Trouw of Los Alamos National Laboratory for the direction they have provided.

References

[1] Xilinx Inc., http://www.xilinx.com.

[2] SRC Computers, Inc., http://www.srccomputers.com.

[3] Cray, Inc., http://www.cray.com.

[4] *IEEE Standard for Binary Floating-Point Arithmetic*, 1985, IEEE Std. 754-1985.

[5] B. Fagin and C. Renard, "Field programmable gate arrays and floating point arithmetic," *IEEE Transactions on VLSI Systems*, vol. 2, no. 3, pp. 365–367, September 1994.

[6] L. Louca, T. A. Cook, and W. H. Johnson, "Implementation of IEEE single precision floating point addition and multiplication on FPGAs," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 1996.

[7] W. B. Ligon, III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. D. Underwood, "A re-evaluation of the practicality of floating-point operations on FPGAs," in *Proceedings of the 6th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.

[8] P. Belanovic and M. Leeser, "A library of parameterized floating point modules and their use," in *Proceedings of the 12th International Conference on Field Programmable Logic and Applications*, M. Glesner, P. Zipf, and M. Renovell, Eds. Berlin: Springer-Verlag, September 2002, pp. 657–666.

[9] X. Wang, M. Leeser, and H. Yu, "A parameterized floating-point library applied to multispectral image clustering," in *Proceedings of the 7th annual MAPLD International Conference*, September 2004, http://klabs.org/mapld04/abstracts/wang_x_a.pdf (abstract).

[10] G. Lienhart, A. Kugel, and R. Manner, "Using floating-point arithmetic on FPGAs to accelerate scientific N-body simulations," in *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society Press, April 2002, pp. 182–191.

[11] J. Liang, R. Tessier, and O. Mencer, "Floating point unit generation and evaluation for FPGAs," in *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2003, pp. 185–194.

[12] G. Govindu and V. K. Prasanna, "Analysis of high performance floating point arithmetic on FPGAs," in *Proceedings of the 11th Reconfigurable Architectures Workshop (RAW 2004)*, April 2004.

[13] L. Zhuo and V. K. Prasanna, "Scalable modular algorithms for floating-point matrix multiplication on FPGAs," in *Proceedings of the 11th Reconfigurable Architectures Workshop (RAW 2004)*, April 2004.

[14] V. Daga, G. Govindu, V. K. Prasanna, S. Gangadharpalli, and V. Sridhar, "Floating-point based block LU decomposition on FPGAs," in *Proceedings of the 2004 International Conference on Engineering Reconfigurable Systems and Algorithms*, T. Plaks, Ed., June 2004, pp. 276–279.

[15] R. Scrofano and V. K. Prasanna, "Computing Lennard-Jones potentials and forces with reconfigurable hardware," in *Proceedings of the Interational Conference on Engineering Reconfigurable Systems and Algorithms*, T. Plaks, Ed., June 2004, pp. 284–290.

[16] K. Underwood, "FPGAs vs. CPUs: Trends in peak floating-point performance," in *Proceedings of the 2004 ACM/SIGDA Twelfth International Symposium on Field Programmable Gate Arrays*, February 2004.

[17] K. D. Underwood and K. S. Hemmert, "Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance," in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[18] D. Gay and T. Sumner, "paranoia (C version)," http://www.netlib.org/paranoia/.

[19] ModelSim, http://www.model.com/.

[20] M. D. Ercegovac, T. Lang, J.-M. Muller, and A. Tisserand, "Reciprocation, square root, inverse square root, and some elementary functions using small multipliers," *IEEE Transactions on Computers*, vol. 49, no. 7, pp. 628–637, July 2000.

[21] Y. Li and W. Chu, "A new non-restoring square root algorithm and its VLSI implementations," in *Proceedings of the 1996 International Conference on Computer Design*, October 1996.

[22] M. Allen and D. Tildeseley, *Computer Simulation of Liquids*. New York: Oxford University Press, 1987.

[23] N. Tredennick and B. Shimamoto, "Reconfigurable systems emerge," in *Proceedings of the 2004 International Conference on Field Programmalbe Logic and Its Applications*, ser. Lecture Notes in Computer Science, J. Becker, M. Platzner, and S. Vernalde, Eds., vol. 3203, September 2004, pp. 2–11.

[24] Nallatech, http://www.nallatech.com.

[25] Quixilica, http://www.quixilica.com.

[26] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. San Francisco: Morgan Kaufman, 2004.

[27] J. F. Wakerly, *Digital Design: Principles and Practices*, 3rd ed. Prentice Hall, 2001.