Implementation and Analysis of Numerical Components for Reconfigurable Computing

Walter B. Ligon III Greg Monn Dan Stanzione Fred Stivers Keith D. Underwood[*]

Planet Earth Research Lab[†]
Department of Electrical and Computer Engineering
Clemson University
105 Riggs Hall
Clemson, SC 29634-0915

{walt, fstiver, gmonn, dstanzi, keithu}@parl.clemson.edu
http://ece.clemson.edu/perl

Abstract—In the past, reconfigurable computing has not been an option for accelerating scientific algorithms (which require complex floating-point operations) and other similar applications due to limited Field Programmable Gate Array (FPGA) density. However, the rapid increase of FPGA densities over the past several years has altered this situation. The central goal of the Reconfigurable Computing Application Development Environment (RCADE) is to capitalize on these improvements. Through RCADE, an algorithm is translated into a data flow design, which is then implemented on a reconfigurable computing platform using a "toolbox" of components. This paper expands this library of components by implementing the following IEEE single precision floating-point functions: sine, cosine, arctangent, arcsine, arccosine, square root and natural logarithm. Each component is designed around the CORDIC shift-and-add algorithms. A discussion of how each operation is implemented is followed by an analysis of the space requirements of each component on current and future Xilinx FPGAs. Performance results are compared for each component individually and for two example equations against several current workstations.

TABLE OF CONTENTS

1. INTRODUCTION
2. BACKGROUND
3. COMPONENT IMPLEMENTATION
4. PERFORMANCE
5. CONCLUSIONS AND FUTURE WORK

[*]Keith Underwood is supported by an NSF Graduate Research Fellowship

[†]The Planet Earth Research Lab is supported by NASA GSFC under grant number NAG 5-3053. Donations were received from Xilinx under donation number 0000CU7-1978 and Synopsys under agreement number UPS-70004334

1. INTRODUCTION

Reconfigurable computing has proven to be extremely useful at enhancing the performance of many applications and algorithms over that of a workstation alone [1], [2], [3], [4] and [5]. This success has been difficult to copy for more complicated scientific algorithms due to limited FPGA density. These algorithms typically require a fractional representation (usually a standard IEEE floating-point format) and have many complex elementary operations (i.e. trigonometric and logarithmic functions). These floating-point operations have proven to be too large to be performed on a single FPGA. Attempts to overcome this problem have used fixed-point representations and reduced precision floating-point formats [6]. Unfortunately, these formats are inconvenient and do not have the dynamic range or precision required by many algorithms. However, with the rapid increase in FPGA density, reconfigurable computing can now overcome these problems. Our previous work [7] has demonstrated that FPGAs can now achieve high performance with simple floating-point operations, such as multiplication and addition. For reconfigurable computing to be a useful tool for scientific algorithms, the more complex operations must also be available. With a "toolbox" of these (and other) floating-point operations, many scientific algorithms can be directly implemented on current reconfigurable computing platforms.

This is the goal of the Reconfigurable Computing Application Development Environment (RCADE). RCADE provides a "user friendly" means to describe an algorithm as a data flow design. This design is then implemented on a reconfigurable computing platform using a library of components. Using RCADE, reconfigurable computing can provide non-hardware designers an alternative for accelerating algorithms. A complete description of RCADE can be found in [8], while this paper focuses on component implementation. This paper discusses implementing the following floating-point operations: sine, cosine, arctangent, arcsine, arccosine, square root, and natural logarithm. Each operation is designed as a single component capable of being interfaced with any other component in the RCADE library without the need for additional control logic. This allows equations to be easily constructed in a data flow design, by combining these operations with previously developed multiply, add/subtract, sum and division components.

Each of the components described in this paper are designed around the CORDIC shift-and-add algorithms. CORDIC is a set of hardware efficient algorithms that uses only shifts and add/subtracts to compute a number of trigonometric and logarithmic functions. The CORDIC algorithms provide the core for each component. Floating-point extensions are added before and after the CORDIC core to handle IEEE single-precision floating-point values. This simplifies interfacing components with the host program and typically provides sufficient dynamic range and precision for most algorithms. High performance is achieved through fully pipelining each component.

The next section gives a brief background on the CORDIC shift-and-add algorithms and how they can compute the above operations. Section 3 describes how the CORDIC algorithms are extended to handle floating-point values. Section 4 details the space requirements for each operation and compares the performance of each operation and two example equations on an Annapolis Microsystems Wildforce reconfigurable computing platform with that of several current workstations. Finally, our conclusions and the direction of future work are presented.

2. Background

CORDIC is a collection of iterative shift-and-add algorithms which form an extremely efficient means of computing trigonometric functions and other complex elementary functions, such as square root, natural log and exponential. They were originally introduced by Volder [9] and were later expanded by Walther [10]. A number of machines have implemented CORDIC. The base set of equations is as follows:

$$x_{n+1} = x_n - m d_n y_n 2^{-n} \tag{1}$$

$$y_{n+1} = y_n + d_n x_n 2^{-n} \tag{2}$$

$$z_{n+1} = z_n - d_n e_n \tag{3}$$

CORDIC operates by performing multiple iterations of the above equations. This has the effect of rotating a vector (x, y) through one of three coordinate systems. z is the angle generated by the vector described by (x, y). Rotation of the vector is controlled by d_n(either 1 or -1). d_n is based on the mode of operation selected (rotation or vectoring). When in vectoring mode, d_n = 1 if $y_n < 0$ and -1 otherwise. For rotation mode, d_n = -1 if $z_n < 0$ and 1 otherwise (except for sin^{-1} and cos^{-1}). The coordinate system is selected by m (circular, m = 1, linear, m = 0, and hyperbolic, m = -1). e_n is the angle of rotation at each iteration and is set by the coordinate system used. As each iteration of the CORDIC equations is performed, either z (rotation mode) or y (vectoring mode) is driven to 0 and the equations in Table 1 are generated. Each coordinate system is useful for computing different functions. For the functions of interest, the circular and hyperbolic systems are used.

Table 1 describes the equations generated after n iterations for each coordinate system and mode of operation. Table 1 also provides the equations for A_n. A_n is the gain produced during the vector rotation. The gain must be accounted for in order to produce accurate results. Typically, one bit of precision is developed per iteration. As indicated by Table 1, by selecting the proper initial values (while compensating for the gain), the functions of interest can be computed. The following sections will briefly describe how each function is generated.

2..1 *Sine/Cosine*

The sine and cosine of an input angle are computed using the rotational mode of operation and the circular coordinate system. The circular coordinate system is used for all of the trigonometric functions. From Table 1, by setting y_0 to zero the equations reduce to:

$$x_n = A_n \cdot x_0 cos z_0 \tag{4}$$

$$y_n = A_n \cdot x_0 sin z_0 \tag{5}$$

Therefore, with x_0 = 1/A_n, after n iterations of equations 1, 2 and 3, $x_n = cos z_0$ and $y_n = sin z_0$.

2..2 *Arctangent*

The arctangent, $\theta = tan^{-1}(y_0/x_0)$, can be computed using the vectoring mode of operation. From Table 1:

$$z_n = z_0 + tan^{-1}(y_0/x_0) \tag{6}$$

Therefore, by setting z_0 = 0 and properly selecting y_0 and x_0, the tan^{-1} of an input argument can be computed directly.

2..3 *Arcsine/Arccosine*

The arcsine and arccosine require a manipulation of the basic cordic equations. Starting with a unit vector either on the

Table 1: CORDIC equations[11]

Coordinate System	rotation mode	vectoring mode	scale factor, A_n	rotation angle, e_n
m = 1 (circular)	$x_n = A_n(x_0 cos z_0 - y_0 sin z_0)$ $y_n = A_n(y_0 cos x_0 + x_0 sin z_0)$ $z_n = 0$	$x_n = A_n\sqrt{(x_0^2 + y_0^2)}$ $y_n = 0$ $z_n = z_0 + tan^{-1} y_0/x_0$	$\prod_{n=0}^{\infty}\sqrt{1+2^{-2n}}$ $\simeq$ 1.64676	$tan^{-1}2^{-n}$ (n = 0 to ∞)
m = 0 (linear)	$x_n = x_0$ $y_n = y_0 + x_0 y_0$ $z_n = 0$	$x_n = x_0$ $y_n = 0$ $z_n = z_0 + y_0/x_0$	no scale factor	2^{-1}
m = -1 (hyperbolic)	$x_n = A_n(x_0 cosh z_0 + y_0 sinh z_0)$ $y_n = A_n(y_0 cosh x_0 + x_0 sin z_0)$ $z_n = 0$	$x_n = A_n\sqrt{(x_0^2 - y_0^2)}$ $y_n = 0$ $z_n = z_0 + tanh^{-1} y_0/x_0$	$\prod_{n=1}^{\infty}\sqrt{1+2^{-2n}}$ $\simeq$ 0.82816	$tanh^{-1}2^{-n}$ (n = 1 to ∞)

positive x axis (arcsine) or on the positive y axis (arccosine), the vector is rotated until its $y(x)$ component is equal to the input angle. Instead of using the sign of an intermediate result to control the direction of rotation, it is determined by whether $y(x)$ is less than (greater than) the input value, c. For the sin^{-1} using the vectoring mode, the equation for z_n from Table 1 becomes:

$$z_n = z_0 + sin^{-1}\{c/(A_n \cdot x_0)\} \qquad (7)$$

where d_n = 1 if $y_n < $ c and -1 otherwise.

By setting x_0 to 1/A_n, y_0 to 0 and z_0 to 0, the arcsine is directly computed. Unfortunately, the accuracy of the result diminishes with inputs of $\simeq \pm$ 0.98 or greater. As the y axis is approached, rotation decisions are made improperly because the gain causes the vector to be smaller than the input. Using a “double iteration algorithm” [12] can correct this problem, but it increases the complexity. For our purposes, this loss of accuracy is not critical. As larger FPGAs become available, this problem can be corrected with this algorithm.

For the arccosine, a few modifications to the above algorithm must be made. Instead of comparing the y component to the input, the x component is compared. This produces:

$$z_n = z_0 + cos^{-1}\{c/(A_n \cdot y_0)\} \qquad (8)$$

where d_n = 1 if $x_n > $ c and -1 otherwise.

Therefore, the $cos^{-1}c$ is calculated when x_0 = 0, y_0 = 1/A_n and z_0 = π/2, Again, the accuracy of the result decreases as $\pm$ 1 is approached and the same corrective measures can be introduced.

2..4 Square Root and Natural Logarithm

While the circular coordinate system is useful for computing trigonometric functions, the hyperbolic coordinate system is used for the other functions of interest (square root and natural log). For the hyperbolic system, m = -1 in the base equations. Additionally, instead of starting iterations at n = 0, for the hyperbolic system iterations must start at n = 1. Certain iterations must also be repeated (n = 4, 13,40,...,k,3k+1,...) or the vector will not converge. This generates the hyperbolic equations in Table 1 for each mode of operation. As can be seen, the same methods can be used to calculate $sinh$, $cosh$ and $tanh^{-1}$ as were used for sin, cos and tan^{-1}. From [10]:

$$\sqrt{\alpha} = \sqrt{(x^2 - y^2)} \qquad (9)$$

where x_0 = α + 1/4 and y_0 = α - 1/4

and,

$$ln\alpha = 2 \cdot tanh^{-1}[y_0/x_0] \qquad (10)$$

where x_0 = α + 1 and y_0 = α - 1

By using the vectoring mode of operation from Table 1, the square root and natural log can be directly computed. The square root of a value (α) is generated by setting $x_0 = \alpha + 1/[4A_n^2]$ and $y_0 = \alpha - 1/[4A_n^2]$. x_n is then equal to $\sqrt{\alpha}$ after n iterations. (As an added benefit, since the vectoring mode is used, and z_n is not needed, 1/3 of the adder/subtractors can be eliminated.)

The natural log is also computed using the vectoring mode of operation. With z_0 = 0 and $x_0 = \alpha$ + 1 and $y_0 = \alpha$ - 1, $z_n = \frac{1}{2}(ln\alpha)$. Therefore, a 1 bit left shift of z_n produces the ln α. (Also of note, e^{α} can also be computed. $e^{\alpha} = sinh\alpha + cosh\alpha$. However, extending this to floating-point was not feasible, at this time.)

Table 2: Range of Valid Inputs

Function	Domain of Convergence
sine/cosine	± 1.74
tan^{-1}	$\pm\infty$
sin^{-1}/cos^{-1}	± 0.98
$\sqrt{\alpha}$	[0.03,2.42]
lnα	[0.10,9.58]

2..5 *Range of Valid Inputs*

Table 2 describes the range of valid inputs for each operation. For sine and cosine, this range can be misleading. Both are valid for an entire period of each function (-$\pi/2$ to $\pi/2$). Therefore, it is only a matter of reducing the input into the valid range in order to compute the sine and cosine for any value. However, since each component is fully pipelined for performance, we were forced to limit the inputs to a set range (selected as 0 to 2π). For the arcsine and arccosine, the range of valid inputs is already limited to a [-1, +1] by the functions themselves. While the single iteration approach does not completely cover this range, it is sufficient for these tests.

For the arctangent, there is no specified valid range. The CORDIC algorithm will compute correct results provided the ratio y_0/x_0 = input value. However, hardware design constraints of bit lengths and data format do limit the arctangent component's range. The absolute value of the maximum input allowed is determined by the maximum y_0 and minimum x_0 possible from the data format. Any input values greater than this maximum must be considered as $\pm\infty$ and are calculated by setting x_0 to 0.

For natural log and square root, the range of valid inputs from Table 2 are limiting for fixed-point formats. However, for a floating-point format, this is not a problem. The mantissa is usually on the binary range of [1.0,2.0). This falls in the valid range for the CORDIC algorithms. For both cases, zero and negative inputs are treated as exceptions.

2..6 *Precision*

The precision is determined by 2 factors: the angular error and the truncation error [5]. Since CORDIC works through rotating a vector, the largest potential error occurs when the next to last iteration produces the correct result. The final rotation would then produce the worst case angular error equal to the last rotation angle and the correct result. Therefore, the number of iterations is the primary determinant of the accuracy. Sufficient iterations must be performed such that the maximum angular error is less than the required precision. One bit of precision is typically produced per iteration [5].

S	exp (+127)	mantissa

Figure 1: IEEE single precision floating-point format

To a smaller degree, the accuracy is affected by the truncation of intermediate results due to limited storage (i.e., from the right shifted terms at each iteration). Some error from truncation is unavoidable since it is impractical to maintain all significant bits throughout the computation. The truncation of intermediate results produces log(2L) of error for L iterations [10]. To eliminate this error, the data must be padded on the right by the same number of bits and rounded at the end.

For single precision floating-point values (with a 24 bit mantissa), 24 bits of precision are required. This produces approximately 24 iterations and 4.5 bits of padding. These guidelines were used for the initial design of each component. Experimentation was then used to optimize the number of iterations and padding bits to establish the required precision while minimizing the size of each component.

3. Component Implementation

The previous section describes how to use the CORDIC algorithms to compute each function from fixed-point data. For floating-point data, extensions must be added to this CORDIC core. Figure 1 shows the IEEE single precision floating-point format as described by ANSI/IEEE Standard 754-1985 [13]. The most significant bit is the sign bit, S. The next 8 bits are for the exponent, E, in excess-127 notation. The final 23 bits are the mantissa, M. The mantissa has an implicit 1 as its most significant bit (for most cases). Therefore, for most values, the number represented is $-1^S * 2^{E-127} *$ (1.M). Zero is represented by all zeros. There are other elements to this standard (i.e. NaN, Infinity, etc.). However, in order to keep the component designs straightforward (without numerous special conditions), these exceptional cases are not handled. The complete standard can be implemented in future revisions.

Figure 2 is a basic block diagram which is applicable for each component. There are 3 steps to each component. The first is to convert the floating-point input into a fixed-point format (i.e., to provide the proper inputs to the CORDIC core). The next step uses the CORDIC shift-and-add stages to compute the applicable function. Finally, the result from the CORDIC section is converted back into floating-point format. This includes two's complement conversion (as necessary), normalization, rounding and exception handling. The following sections describe each component's implementation in more detail.

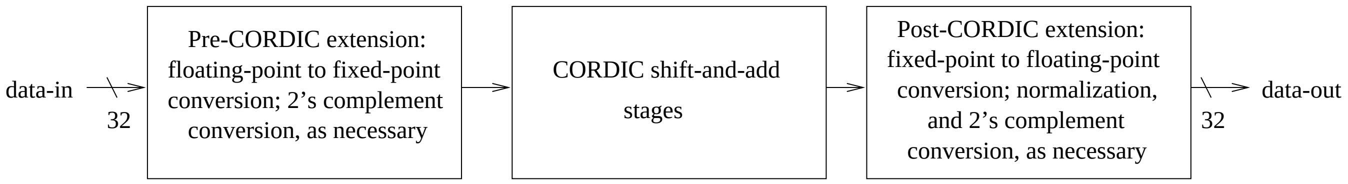
data-in
32
Pre-CORDIC extension: floating-point to fixed-point conversion; 2's complement conversion, as necessary
CORDIC shift-and-add stages
Post-CORDIC extension: fixed-point to floating-point conversion; normalization, and 2's complement conversion, as necessary
32
data-out

Figure 2: Block Diagram of floating-point CORDIC functions

Table 3: Extending Sine/Cosine for 0 to 2π

Quadrant, Q	Sine	Cosine
0	sin D	cos D
1	cos D	-sin D
2	-sin D	-cos D
3	-cos D	sin D

3..1 *Sine/Cosine*

For the sine and cosine components, the first step is to convert the floating-point input into fixed-point for the CORDIC shift-and-add stages. The range of valid inputs is 0 to 2π. The mantissa is directly latched into the the following format: xxx.xxxxx...x. A 3 stage barrel shifter is used to right shift the mantissa by 129 - E bits. This properly aligns the decimal point. However, the CORDIC stages require an input on the range of $-\pi/2$ to $\pi/2$. Therefore, the input angle is reduced to the form $(Q \cdot \pi/2 + D)$, such that D is between 0 and $\pi/2$ and Q is the input quadrant. Q is saved for later use and D is the input to the CORDIC stages.

Using the CORDIC algorithm, both the sine and cosine of D are computed. No extra logic is required because the CORDIC algorithm computes both functions simultaneously. The value of Q is used to latch the proper output. Table 3 indicates which output of the CORDIC stages should be used depending on the value of Q. Since the sine and cosine of all angles between 0 and $\pi/2$ are positive, the resultant sign and mantissa are latched directly.

Once the output is latched, it is then normalized to 1.xxxxx...x, if possible. The resultant exponent is computed based on how many bits the mantissa is left shifted during normalization. Normalization requires 4 stages. Finally, rounding and exception handling is performed and the result is output. The overall component requires 38 stages: 12 stages to convert to and from floating-point and 26 CORDIC stages.

3..2 *Arctangent*

As described in section 2..2, the CORDIC stages require 2 inputs to compute the arctangent, x_0 and y_0, where y_0/x_0 = input value. y_0 could be set equal to the input value with x_0 set equal to 1, but that would severely limit the range of inputs. In order to give a full range of inputs, x_0 and y_0 are set based on the input value. When the |input| < 2.0,

$$x = 1.0 \tag{11}$$

$$y = mantissa * 2^{-(127-E)}. \tag{12}$$

When the |input| $\geq$ 2.0,

$$x = 1.0 * 2^{-(E-127)} \tag{13}$$

$$y = mantissa. \tag{14}$$

This provides the maximum range of inputs.

The same 3 stage barrel shifter is used to perform the right shifting of either x_0 or y_0 by the value of |E-127|. The last stage prior to the CORDIC stages performs a two's complement conversion of y based on the input sign. x_0 and y_0 are then input to the CORDIC stages. The first stage after the CORDIC section performs a two's complement conversion of the result, as necessary. The same normalization, rounding and handling of exceptions converts the result to floating-point. The arctangent component requires 36 stages, including 24 CORDIC stages.

3..3 *Arcsine/Arccosine*

The floating-point extensions for the arcsine and arccosine components operate in much the same manner as the sine and cosine. The only differences are in the fixed-point format and the need to handle negative inputs and outputs for the CORDIC sections. For the pre-CORDIC section, the input value is latched into the following format: x.xx...x. The decimal point is aligned using the same 3 stage barrel shifter. The input to the barrel shifter is 127 - E. After the barrel shifter, the two's complement is taken for negative inputs. The first step in the post-CORDIC section is to take the two's complement of the result, if negative, and to latch the sign of the result. The rest of the post-CORDIC section is identical to the sign and cosine. The arcsine and arccosine components have 24 CORDIC stages and 36 overall.

3..4 *Square Root*

The square root component takes a slightly different path from the previous components. It is not necessary to align the mantissa's decimal point or to perform a long normalization process. An element of the floating-point representation is used to simplify the computation. Because floating-point values are represented in base 2, it is simple to compute the resultant exponent if the input exponent is even (resultant exponent prior to normalization of the mantissa = E/2). The pre-CORDIC stage needs only to shift the mantissa so that the exponent is made even while the mantissa remains in the valid range. (An input of zero is handled as an exception.) The mantissa is initially on the range [1.0,2.0). Therefore, if the exponent is odd, the mantissa is right shifted 1 bit and 1 is added to the exponent. The mantissa is now on the range [0.5,2.0), which is still in the valid range, and the exponent is even. The mantissa provides the input to the CORDIC stages. Simultaneously, the unnormalized resultant exponent is calculated by first subtracting 127 (to convert from excess-127 notation). The exponent is right shifted by 1 bit to divide by 2 and converted back into excess-127 notation by adding 127. This value is then passed through the pipeline to be reunited with its mantissa. The post-CORDIC section only requires one normalization stage since the resultant mantissa is on the range[$\sqrt{.5}$ = 0.707, $\sqrt{2.0}$ = 1.414). The final stages handle exceptions, rounding and output the result. Through experimentation, it was determined that only 20 CORDIC stages were required, with 29 total stages.

3..5 Natural Logarithm

Similar to the square root, an element of the floating-point representation is used to assist computation. It is known that:

$$ln(M * 2^E) = ln(M) + ln(2^E) \tag{15}$$

Using this equation, the natural log of a floating-point number can be computed using a combination of the CORDIC algorithm and an adder. Since the exponent is only 8 bits, it is feasible to store the ln(2^i) for i from -127 to 128 in memory. Exponents of -127 and 128 (0 and 255 in excess-127 notation) are reserved for special values (i.e. ∞ and zero) and can be dealt with by storing the appropriate adder inputs for those exceptions in their associated memory locations. Therefore, the CORDIC algorithm is used to compute the ln(M), the ln(2^E) is read from memory and a floating-point adder is used to compute the final result. Figure 3 is a block diagram of the natural log component.

The ln(M) is computed similarly to the previous components. The mantissa is directly latched from the input and no decimal alignment is required. This provides the input for the CORDIC stages. The CORDIC section requires 26 stages to produce the desired precision. Since the ln(M) = 2*$tan^{-1}(y/x)$, the multiplication is performed by a 1 bit left shift of z_n. This value is then normalized and converted back into a floating-point value in the same manner as the sine and cosine component. This output, ln(M), is used as one of the adder inputs.

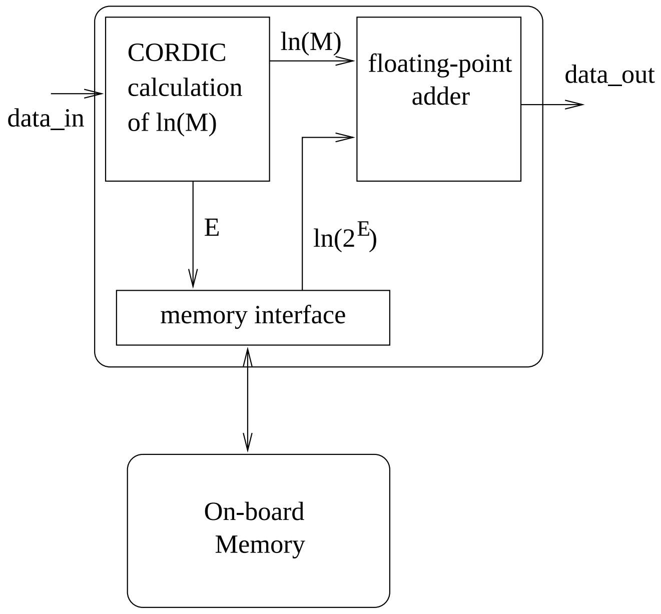

Figure 3: Block Diagram of Wildforce board

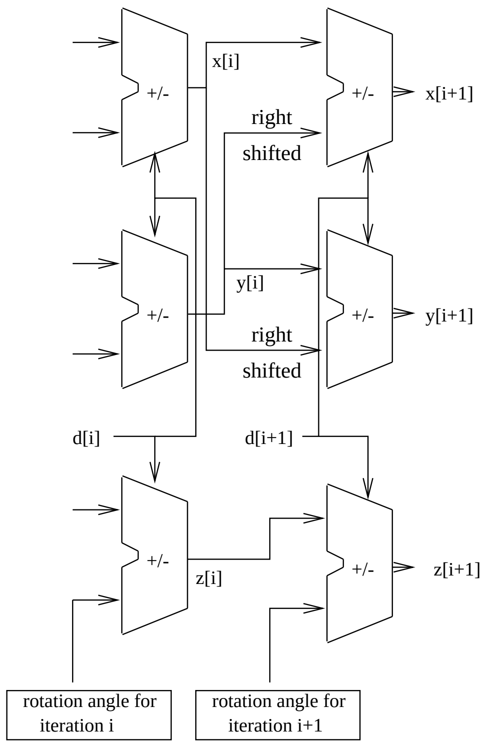
x[i]
+/-
x[i+1]
right shifted
y[i]
y[i+1]
d[i]
d[i+1]
z[i]
z[i+1]
rotation angle for iteration i
rotation angle for iteration i+1

Figure 4: Common CORDIC add/subtract stages

The other input to the adder (ln(2^E)) is from memory. The Xilinx 4000 series FPGAs have an on-chip memory capability. However, a 256x32 bit on-chip ROM would require an unacceptable number of CLBs and addressing would limit maximum clock rate. Instead, an off-chip SRAM provided by the implementation platform is used for the look-up table. The exponent provides the memory address for the value of the ln(2^E). The memory interface has 2 16x32 bit FIFOs to buffer the memory addresses and data so that the memory does not become a bottleneck. The memory data is then matched with the corresponding ln(M) result at the input to the adder. The result of the adder is then the ln(M*2^E).

3..6 Common Structures

Since the same basic overall design is used to compute each function, many of the same internal components are reused. The most obvious of these are the CORDIC shift-and-add stages. Figure 4 shows a diagram of 2 shift-and-add stages. Each stage requires 3 adder/subtractors (except for the square root component which does not need the z component). Since this is a pipelined design, separate shifting hardware is not required. For each stage, the number of bits to shift is constant. Therefore, the shifting is performed by hardwiring the shift at the input to the adjacent adder/subtractor. Another benefit to the pipelined design is that a ROM is not required to store the e_n data. Each stage always uses the same e_n data (based on the stage) and these values are hardwired to the appropriate input for the z adder/subtractors. These stages only differ between components by the adder/subtractor size, adder/subtractor control (d_n) and the number of bits shifted in each stage (circular vs. hyperbolic).

Another common internal component is the 3 stage barrel shifter for the pre-CORDIC section. This component uses 3 stages to right shift the mantissa by 0 to 24 bits. It performs this by utilizing 3 4-to-1 multiplexers. The total number of bits to be shifted provides the input for the multiplexers. For the sine and cosine components, this is 129-E. For the arctangent, it is the $\mid E - 127 \mid$ and for the arcsine and arccosine components it is 127-E. The first multiplexer shifts by either 0, 1, 2 or 3 bits based on the 2 low-order bits of the control input. The second multiplexer shifts by 0, 4, 8 or 12 bits based on bits 2 and 3 and the third multiplexer shifts by 0, 16 or shifts in all zeros based on bit 3 and a combination of the higher order bits. The barrel shifter is used by all 5 trigonometric components.

The other primary shared component is the 4 stage normalization unit. It is utilized by all of the components, except the square root. The first 3 stages use 4-to-1 multiplexers to normalize by 0, 1, 4 or 8 bits. The final stage normalizes by 0, 1, 2 or 3 bits. Therefore, any value (28 bits long or less) can be normalized in 4 stages. The first 3 stages operate by checking for a ‘1’ in the high order bit, in the 4 highest order bits and in the 8 highest order bits. Based on these conditions, the multiplexer shifts the value. After 3 stages, if the value contains a ‘1’, the highest order ‘1’ will be in the 4 highest order bits. The last stage then finishes the normalization. In conjunction, the resultant exponent is adjusted by the same amount as the mantissa is left shifted in each stage, producing the normalized resultant exponent. Additionally, standard components from the Xilinx design library (i.e. adder/subtractors, subtractors, multiplexers, etc.) are used to minimize component size (CLBs used) and maximize performance (maximum clock rate).

Table 4: Maximum Component Clock Rates

Function/ Equation	Max Clock Rate (MHz)	Board Clock Rate (MHz)
sin/cos	27.7	33
tan^{-1}	26.6	33
sin^{-1}/cos^{-1}	18.5	33
$\sqrt{\alpha}$	38.2	33
ln α	24.9	33

4. PERFORMANCE

4..1 Implementation Platform

The reconfigurable computing platform used to test each component was the Annapolis Microsystems Wildforce board. This board contains 5 Xilinx 4062 FPGAs with 1 MB of SRAM per FPGA. One of the FPGAs serves as the Control Processing Element while the other 4 are the Systolic Processing Elements. Each of the 4 SPEs are interconnected by a 36 bit wide by-directional connection. Each end of the systolic array is connected to a bi-directional 36 bit, 512 deep FIFO. The CPE is also connected to a 36 bit, 512 deep FIFO. The CPE and the SPEs are also interconnected through a crossbar. Figure 5 is a block diagram. The board communicates with the host computer through the PCI bus, providing high speed I/O. The host computer is a 233 MHz Pentium II.

4..2 Resource Utilization

The flip-flop and look-up table utilization for each individual component are provided in Table 5. Table 5 also includes information for a previously designed fully pipelined multiplier and adder/subtractor. At a minimum, at least one component will fit on a single FPGA. (Each component was successfully placed and routed for a Xilinx 4062xl FPGA). With 4 FPGAs on the Wildforce board, equations of at least 4 functions can be evaluated. With the next generation of FPGAs, 3 to 4 times as many components can be placed on each FPGA, as demonstrated by the utilization numbers for the 40250xv FPGAs. Therefore, large complicated equations (typical in scientific

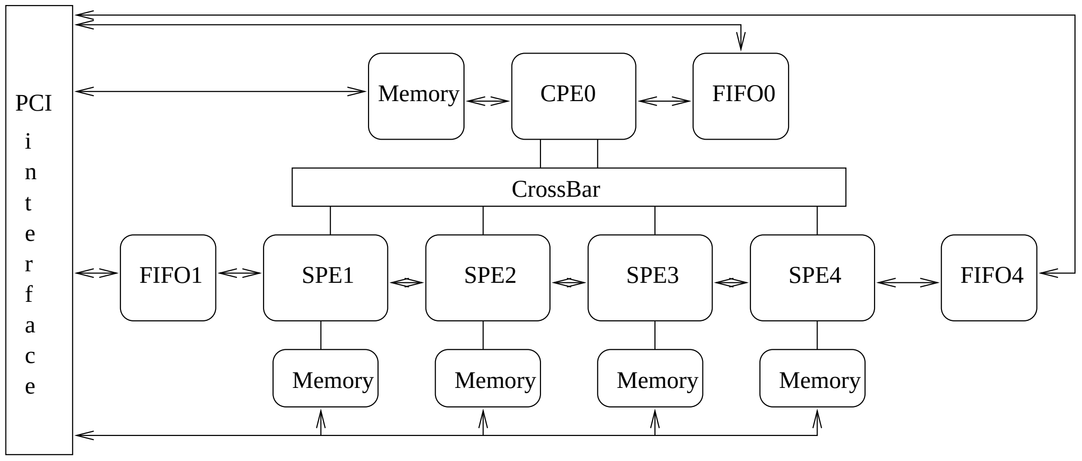

Figure 5: Block Diagram of Wildforce board

algorithms) can be evaluated on a reconfigurable computing platform.

Each component was designed in VHDL using a combination of behavioral and structural constructs. The Synopsys FPGA Compiler was used for component synthesis and the Xilinx M1.5 tools performed place and route. Table 4 lists the maximum clock rates as given by the place and route tools and the clock rate as tested on the Wildforce board. In order to consume data as fast as it can be input, the component clock rate must be at least 33 MHz. Little benefit is derived from any faster clock rate. While only the square root component meets this minimum clock rate as reported by the place and route tools, all components were successfully tested at 33 MHz on the board.

4..3 *Component Performance*

Component performance was evaluated by comparison against several current workstations, a 300 MHz Sun Ultra 10, a 233 MHz Pentium II and a 500 MHz Alpha 21164. Individual components were evaluated by comparing the time to perform 2^{20} computations for each function. For the Wildforce board implementation, each component was placed on SPE1. The floating-point values were input through FIFO1 and the results were output through SPE1's SRAM (except for natural log, which wrote its results to SPE2's SRAM to prevent the memory from being a bottleneck). The time recorded is the total time required to write the data from host memory to the board, perform the computation and read the results back into host memory. For each workstation, it is the time required to read the data from memory, perform the computation and write the results back to memory. This provides a realistic comparison of component performance. Table 6 shows the results.

Given a maximum PCI bus bandwidth of 132 MB/sec, complete overlap of computation and I/O, and no overlap of input and output, the theoretical maximum performance would be 60.6 msec. This performance cannot be achieved because I/O and computation cannot be completely overlapped and the maximum PCI bandwidth cannot be reached. However, the Wildforce board did achieve significant speedups for each component over each workstation. Against the host computer, speedups ranged from 3.9 for the square root to over 17.4 for the arcsine and arccosine. When compared to the Alpha, speedups averaged from 2.9 to 4.9, with even greater speedups against the Ultra 10.

4..4 *Application Performance*

For a more realistic performance comparison, two equations were implemented on the Wildforce board using these floating-point components. The first equation tested is the Pythagorean theorem.

$$c = \sqrt{a^2 + b^2} \tag{16}$$

The values for a and b were input through FIFO1 and SPE2's SRAM, respectively. The equation was partitioned across the board as follows:

SPE1: a^2
SPE2: b^2 and the addition
SPE3: $\sqrt{sum}$ and write the result to SRAM

Table 6 reports the results to compute 2^{20} values for c. (The theoretical performance for the Wildforce board, using the same assumptions as before, is 91 msec since twice as much data must be input to the board with the same amount of output.) Even for a simple equation such as the Pythagorean theorem, performance was enhanced with speedups of 1.87 (Alpha), 1.94 (Pentium II) and 3.2 (Ultra 10).

Table 5: Resource Utilization of fully pipelined floating-point functions

Function	4-LUTs/ Flip-Flops	Xilinx 4062XL	Xilinx 40250XV
sin/cos	2928 / 2629	63% / 57%	16% / 14%
tan^{-1}	2745 / 2399	59% / 52%	15% / 13%
sin^{-1}/cos^{-1}	2789 / 3010	60% / 65%	15% / 16%
$\sqrt{\alpha}$	1600 / 1663	34% / 36%	9% / 9%
$\ln \alpha$	3620 / 3276	78% / 71%	20% / 18%
add	629 / 671	12% / 8%	4% / 4%
multiply	759 / 1017	16% / 22%	4.5% / 6%

Table 6: Performance Results

Function/ Equation	Wildforce (msec)	Sun Ultra 10 (msec)	Pentium II (msec)	Alpha (msec)
sin/cos	70	750	480	220
tan^{-1}	70	560	770	265
sin^{-1}/cos^{-1}	70	680	1220	335
$\sqrt{\alpha}$	70	330	270	205
$\ln \alpha$	70	605	400	220
$\sqrt{x^2+y^2}$	160	515	310	300
$tan^{-1}(\sqrt{t}) + \theta_{old}$	160	520	970	500

The second equation evaluated is a simplified satellite navigation algorithm [14]. Navigation is the process of extrapolating the position of the satellite at a particular time given that the satellite was in a known position at a previous known time. Navigation is frequently used in satellite image processing, where the only data available about a given image is the time at which the satellite took the image. This may be done on a per image, per scan line, or per pixel basis. Navigation is typically done using the Brouwer-Lyddane orbit prediction package, which is too complex for implementation on the current testbed in a single configuration. In this example, the orbit of the satellite is assumed to follow a completely circular path, and the original position and new position are specified in polar coordinates. It is assumed that the satellite has a time-varying angular velocity which can be integrated analytically. The navigation process then simply becomes determining the new angle by adding the old angle to the integral of the angular velocity over the elapsed time. The resulting equation is shown below.

$$\theta_{new} = tan^{-1}(\sqrt{t}) + \theta_{old} \tag{17}$$

where,
θ_{new} is the satellite position after an elapsed time,
θ_{old} is the known initial satellite position, and
t is the elapsed time.

The equation was partitioned as follows:
SPE1: $\sqrt{t}$
SPE2: tan^{-1} and the addition
SPE3: write the result to SRAM

The data for t and θ_{old} were input in the same manner as a and b above. The equation was partitioned in this manner to maximize performance by preventing bottlenecks at the bus between SPE2 and SPE3 or at SPE2's SRAM. Again, the reconfigurable computing implementation outperformed each workstation with speedups ranging from 3.1 to 6.1. These results demonstrate that reconfigurable computing can now provide a performance enhancement for an area that it traditionally has not addressed. With denser FPGAs capable of handling more complicated equations, even greater speedups are expected.

4..5 I/O Considerations

It should be noted that we do not compare the performance of these components for single or small numbers of data items. Since the reconfigurable computing platform must communicate with the host through I/O channels (i.e., the PCI bus for the Wildforce board), it cannot operate as a true co-processor. This provides an algorithm and design limitation determined by the I/O bandwidth. For this approach to provide a performance enhancement, an algorithm or application must perform the same computation over large streams of independent data.

5. Conclusions and Future Work

This paper has described the implementation of several complex floating-point operations as numerical components to be utilized to enhance the performance of scientific algorithms on reconfigurable computing platforms. A hardware efficient algorithm was used as the core of each operation and expanded to handle floating-point values. Through deep pipelining we achieved significant performance improvements over several current workstations. When these components are combined into even more complex equations, even greater speedups can be achieved. These results demonstrate that reconfigurable computing can provide significant performance enhancements for many scientific algorithms and illustrates the feasibility of RCADE. In the future, we will continue to optimize these components for integration into RCADE. As necessary, more components will be developed.

References

[1] L. Abbott, P. Athanas, L. Chen, and R. Elliot, "Finding lines and building pyramids with splash 2," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 155–163, 1994.

[2] W. King, T. Drayer, R. Conners, and P. Araman, "Using morrph in an industrial machine vision system," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 18–26, 1996.

[3] J. Villasenor, B. Schoner, K.-N. Chia, C. Zapata, H. J. Kim, C. Jones, S. Lansing, and B. Mangione-Smith, "Configurable computing solutions for automatic target recognition," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 70–79, 1996.

[4] M. Dao, T. Cook, D. Silver, and P. D'Urbano, "Acceleration of template-based ray casting for volume visualization using fpgas," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 116–124, 1995.

[5] R. Andraka, "Building a high performance bit-serial processor in an fpga," in *Proceedings of Design SuperCon '96*, pp. 5.1–5.21, 1996.

[6] N. Shirazi, A. Walters, and P. Athanas, "Quantitative analysis of floating point arithmetic on fpga based custom computing machines," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 155–162, 1995.

[7] W. Ligon, K. Underwood, F. Stivers, S. McMillan, and G. Monn, "A re-evaluation of the practicality of floating-point operations on fpgas," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 118–125, 1998.

[8] W. Ligon, B. Boysen, N. DeBardeleben, K. Hazelwood, R. Sass, D. Stanzione, and K. Underwood, "A development environment for configurable computing," in *Proceedings of the SPIE International Symposium on Voice, Video, and Data Communications*, 1998.

[9] J. Volder, "The cordic trigonometric computing technique," *IRE Trans. Electronic Computing*, pp. 330–334, 1959.

[10] J. Walther, "A unified algorithm for elementary functions," in *Spring Joint Computer Conference proceedings*, pp. 379–385, 1971.

[11] R. Andraka, "A survey of cordic algorithms for fpga based computers," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 191–200, 1998.

[12] Y. H. Hu and S. Naganathan, "A novel implentation of chirp z-transformation using a cordic processor," *IEEE Transactions on ASSP*, vol. 38, pp. 352–354, 1990.

[13] I. S. Board, "IEEE standard for binary floating-point arithmetic," Tech. Rep. ANSI/IEEE Std 754-1985, The Institute of Electrical and Electronics Engineers, New York, 1985.

[14] National Oceanic and Atmospheric Administration, National Environmental satellite, Data and Information Service, National Climatic Data Center Satellite Data Services Division, Washington, D.C., *NOAA Polar Orbiter Data: Users Guide*, 1991.

***Walter Ligon** received his Ph. D. in Computer Science from the Georgia Institute of Technology in 1992. Since then he has been at Clemson University where he is an Assistant Professor in the Department of Electrical and Computer Engineering. His current research interests are in parallel and distributed systems, I/O for parallel systems, reconfigurable computing,and problem solving environments.*

***Greg Monn** received his M.S. in Computer Engineering from Clemson University in 1998. His interests include reconfigurable computing and FPGA design.*

***Fred Stivers** received his M.S. in Computer Engineering from Clemson University in 1998. His interests include reconfigurable computing and FPGA design.*

***Dan Stanzione, Jr.** is a Ph. D. candidate in Computer Engineering at Clemson University. He received his B.S. in Electrical Engineering in 1991 and his M.S. in Computer Engineering in 1993, both from Clemson. Dan's research interests include reconfigurable computing, remote sensing, and problem solving environments for high-performance computing systems.*

***Keith D. Underwood** received his B.S. in Computer Engineering from Clemson University in 1995. Currently, he is working on his Ph. D. in Computer Engineering from Clemson University. His interests include reconfigurable computing, parallel and distributed systems, and problem solving environments.*