XAPP552 (v1.0) June 1, 2012

Parameterizable CORDIC-Based Floating-Point Library Operations

Authors: Nikhil Dhume and Ramakrishnan Srinivasakannan

Summary

This application note presents a design methodology using the Xilinx System Generator for DSP tool to create a parameterizable floating-point library computation method for trigonometric, power, and logarithmic operations based on the coordinate rotational digital computer (CORDIC) algorithm [Ref 1]. The design methodology leverages the fixed-point CORDIC LogiCORE™ IP v5.0 block, along with floating-point building blocks such as adders, multipliers, comparators, ROM, and FIFOs to create a set of floating-point CORDIC functions that can be used as building blocks in applications. These functions are an essential requisite in a wide range of engineering applications such as image processing, manipulator kinematics, radar signal processing, robotics, and optimization processes in which a large number of trigonometric or power operations must be computed in an efficient manner. The library has been designed using System Generator for DSP, version 13.4, and supports single- and double-precision input as defined by the IEEE 754 floating-point standard [Ref 2].

Introduction

The arithmetic unit is one of the important components of CPU design. For computation of complex arithmetic functions on hardware, the CORDIC algorithm is an attractive fixed-point algorithm that uses a sequence of simple “shift and add” operations to compute a wide variety of arithmetic functions. However, many applications are required to work not only with high precision but also a large dynamic range. Floating-point arithmetic is a feasible solution for such high-performance systems providing a dynamic range for representing real numbers and capabilities to retain resolution and accuracy.

Floating-Point Solution for Xilinx FPGAs

Xilinx FPGAs have long been used to implement fixed-point DSP and video algorithms in hardware. The flexibility of programmable logic allows fixed-point arithmetic to use custom bit widths that are not bound to the 8-, 16-, or 32-bit boundaries of a fixed-point processor. Fixed-point bit widths can grow as needed to accommodate applications that require large dynamic range. However, as the dynamic range needs to grow, a fixed-point implementation becomes increasingly expensive. Although floating-point solutions on FPGAs are inherently slower than contemporary processors, the inherent massive parallelism allows these solutions to be competitive to the software equivalent. For this reason, FPGAs are increasingly being used as floating-point accelerators. To benefit from the parallelism, there is a requirement to use hardware-efficient algorithms for FPGAs. More complex floating-point systems on FPGAs require good implementations of elementary functions such as logarithmic, power, and trigonometric. The System Generator for DSP tool meets this demand by supporting design and implementation of floating-point algorithms from within the Simulink modeling environment. System Generator for DSP also has the flexibility of optimizing an implementation that is bit- and cycle-accurate to the original model. This library has been designed as an extension for customers familiar with the flow of the System Generator for DSP tool.

Fixed-Point CORDIC Algorithm

CORDIC algorithms are a class of iterative solutions for trigonometric and other transcendental functions that use only shifts and adds to perform. The trigonometric functions are implemented based on vector rotation. Incremental functions such as logarithm and power are performed with a simple extension to the hardware architecture and, while not CORDIC in the strictest sense, are often included because of close similarity. A detailed study of the algorithm is given in Fixed-Point CORDIC Algorithm.

Design Approach

A floating-point library for CORDIC trigonometric functions has been developed using the fixed-point CORDIC block and other basic blocks. The approach chosen has been to use underlying trigonometric relations to extend the range of the fixed-point CORDIC algorithm. The input floating-point number is passed through a range reduction step and then processed with a fixed-point CORDIC block. The range reduction step reduces the input range to the one allowed by the fixed-point CORDIC algorithm. A post-processing step performs the inverse of the range reduction step after fixed-point computation. This approach is detailed in Figure 1. The library has been made parameterizable to ensure maximum flexibility. The floating-point CORDIC library presented in this application note has been implemented using the Xilinx IP portfolio. System Generator for DSP, version 13.4, was used to implement the flow.

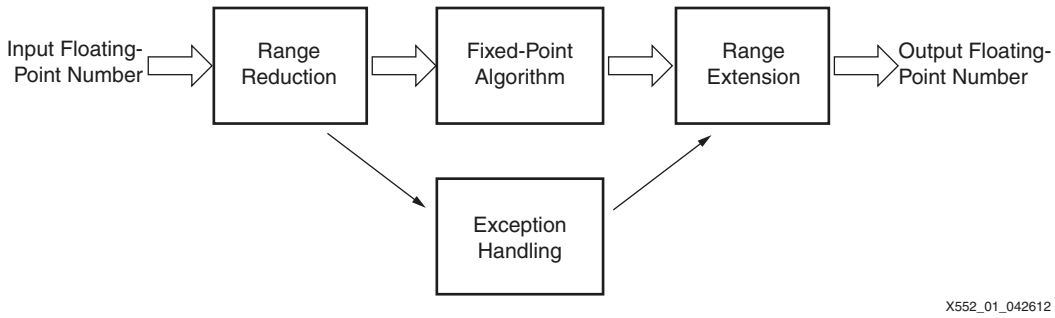

Figure 1: **Approach Chosen for Floating-Point CORDIC Library**

Fixed-Point CORDIC Algorithm

The CORDIC algorithm was initially designed to perform vector rotation, where the vector (X,Y) is rotated through an angle θ yielding a new vector (X',Y'). The vector rotation equations are:

$$X' = (\cos(\theta) \times X - \sin(\theta) \times Y) \quad \textit{Equation 1}$$

$$Y' = (\cos(\theta) \times Y + \sin(\theta) \times X) \quad \textit{Equation 2}$$

$$\theta' = 0 \quad \textit{Equation 3}$$

The CORDIC algorithm performs a vector rotation as a sequence of successively smaller rotations, each of angle atan(2^{-i}), known as micro rotations. Equation 4 through Equation 6 show the expression for the i^{th} iteration, where i is the iteration index from 0 to n. The expression for the i^{th} micro rotation is:

$$X_{i+1} = x_i - \alpha_i \times y_i \times 2^{-i} \quad \textit{Equation 4}$$

$$y_{i+1} = y_i + \alpha_i \times x_i \times 2^{-i} \quad \textit{Equation 5}$$

$$\theta_{i+1} = \theta_i - \alpha_i \times \text{atanh}(2^{-i}) \quad \textit{Equation 6}$$

Where α_i is the direction of rotation and can have a value of ±1.

A detailed description of the CORDIC algorithm is given in Floating-Point Algorithms and Library Interface Specifications, page 10.

Floating-Point Algorithms

Floating-Point CORDIC sin-cos

The rotational mode of fixed-point CORDIC can be used to simultaneously compute the sine (sin) and cosine (cos) of the input angle. Setting the *y* component of the input vector to zero reduces the rotation mode to:

$$X_n = A_n \times x_0 \times \cos(\theta) \quad \text{Equation 7}$$

$$Y_n = A_n \times x_0 \times \sin(\theta) \quad \text{Equation 8}$$

Where A_n is a gain factor corresponding to the i^{th} micro rotation.

Range Enhancement

The range reduction step for the fixed-point CORDIC algorithm can be achieved by performing an angle rotation on the input. The input is rotated to a value between $-\pi$ and $+\pi$, which can then be fed to the fixed-point CORDIC as input. This step is required because the fixed-point CORDIC only converges for the range between $-\pi$ and $+\pi$. The rotation of the angle can be given by:

$$\text{Rotated angle(x)} = \text{remainder(x, } 2 \times \pi) - \pi \quad \text{Equation 9}$$

Because *n* is subtracted from the remainder, there is an inherent reflection in this step which needs to be adjusted for post-processing.

Algorithm

The steps in the algorithm are:

1. Range reduction: The input is rotated between $-\pi$ and $+\pi$.
2. The fixed-point CORDIC block is used for computation of the sine and cosine of the number, as detailed above.
3. Post-processing: A reflection operation is performed on the output of the fixed-point CORDIC block.

Floating-Point CORDIC sinh-cosh

The close relationship between trigonometric and hyperbolic functions suggests that the same architecture can be used to compute hyperbolic functions. The CORDIC equations for hyperbolic rotations are derived by setting the α_i factor in Equation 4, Equation 5, and Equation 6 by the amount shown in Equation 10.

$$\alpha_i = -1 \text{ if } i < 0, +1 \text{ otherwise} \quad \text{Equation 10}$$

This reduces the CORDIC output in rotation mode to Equation 11 and Equation 12.

$$x_n = A_n \times [x_0 \times \cosh(\theta_0) + y_0 \times \sinh(\theta_0)] \quad \text{Equation 11}$$

$$y_n = A_n \times [x_0 \times \cosh(\theta_0) + y_0 \times \sinh(\theta_0)] \quad \text{Equation 12}$$

$$A_n = \Pi\sqrt{1 - 2^{-2i}} \cong 0.80 \quad \text{Equation 13}$$

The value of hyperbolic sine (sinh) and hyperbolic cosine (cosh) can be found by setting y_0 to 0.

Range Enhancement

The range reduction for the hyperbolic functions can be found by splitting the input into fractional and integer portions. The integer portion can be processed by means of a stored look-up table (LUT). The fractional portion can be processed separately using a stored LUT.

Algorithm

The steps in the algorithm are:

1. Range reduction: The input is reduced in range by first finding the absolute value of the number and then splitting the number into integer (*int*) and fractional (*frac*) portions.
2. The *int* portion is processed by means of a stored LUT while the *frac* portion is processed by means of a fixed-point CORDIC algorithm.
3. Post-processing: The outputs from the *int* and *frac* portions are combined using Equation 14 and Equation 15.

$$\cosh(int + frac) = \cosh(int) \times \cosh(frac) + \sinh(int) \times \sinh(frac) \quad \text{Equation 14}$$

$$\sinh(int + frac) = \cosh(int) \times \sinh(frac) + \cosh(frac) \times \sinh(int) \quad \text{Equation 15}$$

4. Final stage: A reflection operation is performed when the input is negative.

Floating-Point CORDIC Power

The algorithm for exponential (e^x) is computed by using the sinh and cosh computed from Floating-Point CORDIC sinh-cosh. In addition to exponential, powers of 10 and 2 are supported in the library.

Algorithm

1. The exponential value of a number can be calculated from the sinh and cosh from Equation 16.

$$e^x = \sinh(x) + \cosh(x) \quad \text{Equation 16}$$

2. The power value of 10 and 2 is computed by the core from Equation 17 and Equation 18.

$$10^x = e^{(x \cdot \ln(10))} \quad \text{Equation 17}$$

$$2^x = e^{(x \cdot \ln(2))} \quad \text{Equation 18}$$

Floating-Point CORDIC atan

The floating-point CORDIC computes arctangent (atan (y/x)) directly using the vectoring mode of the CORDIC rotator if the angle is initialized with 0. The argument must be presented as a ratio of x/y. The angle accumulator output is given by Equation 19.

$$\theta_n = \theta_0 + \tan^{-1}\left(\frac{x_0}{y_0}\right) \quad \text{Equation 19}$$

Range Enhancement

The range reduction step is performed by removing the sign portion of the input numbers and adjusting the input so that the imaginary part is always greater in magnitude than the real part. This adjustment is done to ensure that the output angle is always present in the first quadrant.

Algorithm

The steps in the algorithm are:

1. The absolute value of the input is found, and real and imaginary portions are adjusted so that the real portion is greater than the imaginary portion.
2. The fixed-point CORDIC atan is used to compute the output.
3. The output is rotated to the correct quadrant based on the input sign and whether the real portion of the number is greater than the imaginary portion.

Floating-Point CORDIC log

The CORDIC logarithm (ln) is implemented using the hyperbolic vectoring mode of CORDIC. The hyperbolic arctangent (atanh) can be used to compute log by using Equation 20. Logarithms to the base 10 and 2 are also supported.

$$\ln(w) = 2 \times \operatorname{atanh}\left(\frac{w-1}{w+1}\right) \quad \textit{Equation 20}$$

Range Enhancement

There is an inherent range reduction involved in the previous step. Using w–1 and w+1 ensures that the real portion is always less than the imaginary portion and that the real portion is never equal to the imaginary portion. If the real portion equals the imaginary portion, the output of atanh goes to infinity (which cannot be represented by the fixed-point CORDIC block). The other range reduction algorithm used is separation of input into the mantissa and exponent portion, which are processed separately. The mantissa is processed by means of the fixed-point CORDIC, and the exponent is processed by means of a multiplier.

Algorithm

The steps in the algorithm are:

1. Split the input into the exponent and mantissa portion. The exponent portion can be added back to the processed mantissa at the end.
2. Process the mantissa by using the fixed-point CORDIC algorithm as mentioned above.
3. The $\log_{10}$ and $\log_2$ is found in the CORDIC core using Equation 21.

$$\log_b(w) = \frac{\log(w)}{\log(b)} \quad \textit{Equation 21}$$

System Generator Implementation

Setting Up the Library

For usage of the library in the System Generator for DSP tool, a patch has been created in the TAR file contained in the reference design (see Reference Design, page 16). After the overlay is installed, the library should appear similar to Figure 2. The patch works with the System Generator for DSP tool, version 13.4 for nt, nt64, lin, and lin64 builds.

To set up the library:

1. Clear the MATLAB and System Generator for DSP tool caches using the using **`xlCache ('clear all')`** command.
2. Extract the patch on top of the IDS build using WinZip or the **`tar -xvf`** command.
3. Open the System Generator for DSP tool to the corresponding IDS build.
4. The floating-point blocks are now visible as a part of the reference blockset as a “Floating Point” library.
5. These blocks are present as part of the library: Absolute, Conditional Negate, Floor-ceil, Split, Merge, Remainder, Cordic Sin-cos, Cordic Sinh-cosh, Cordic Atan, Cordic Log, and Cordic Power.
6. The library works for single and double floating-point data types.

Library Usage

The library is available in the floating-point section of the reference blockset, as shown in Figure 2. To use this library, the user can drag and drop any of the blocks to a new model file.

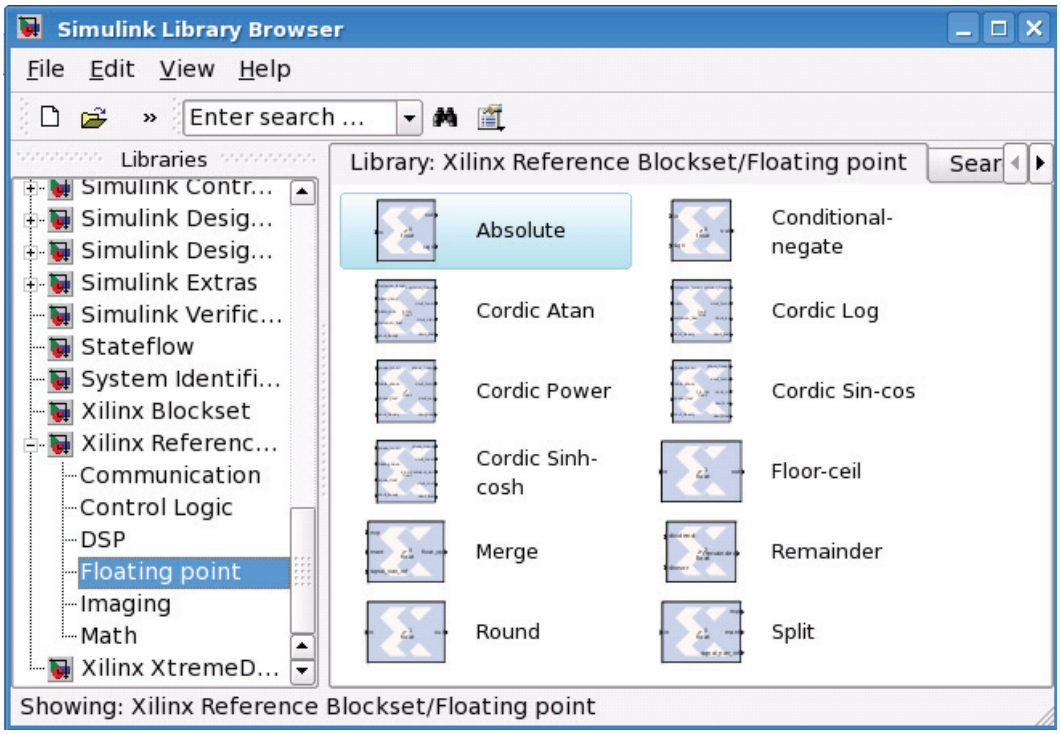

Figure 2: **Floating-Point Blocks of the Xilinx Reference Blockset**

Implementation Details

This section describes some of the System Generator for DSP tool-specific implementation details and application of additional blocks present in the library.

Parameterization of Blocks

The individual blocks in the library are created as subsystems. The GUI is customized using the parameters tab from the block mask. Additionally, user preferences (such as single or double, and size of the LUT) are handled by means of an initialization pane from the edit block mask window. The block mask window with parameters for sin-cos is shown in Figure 3. The initialization section also handles tuning latencies. Detailed instructions on how to create block masks can be found in Trade-offs in the Current Approach, page 13.

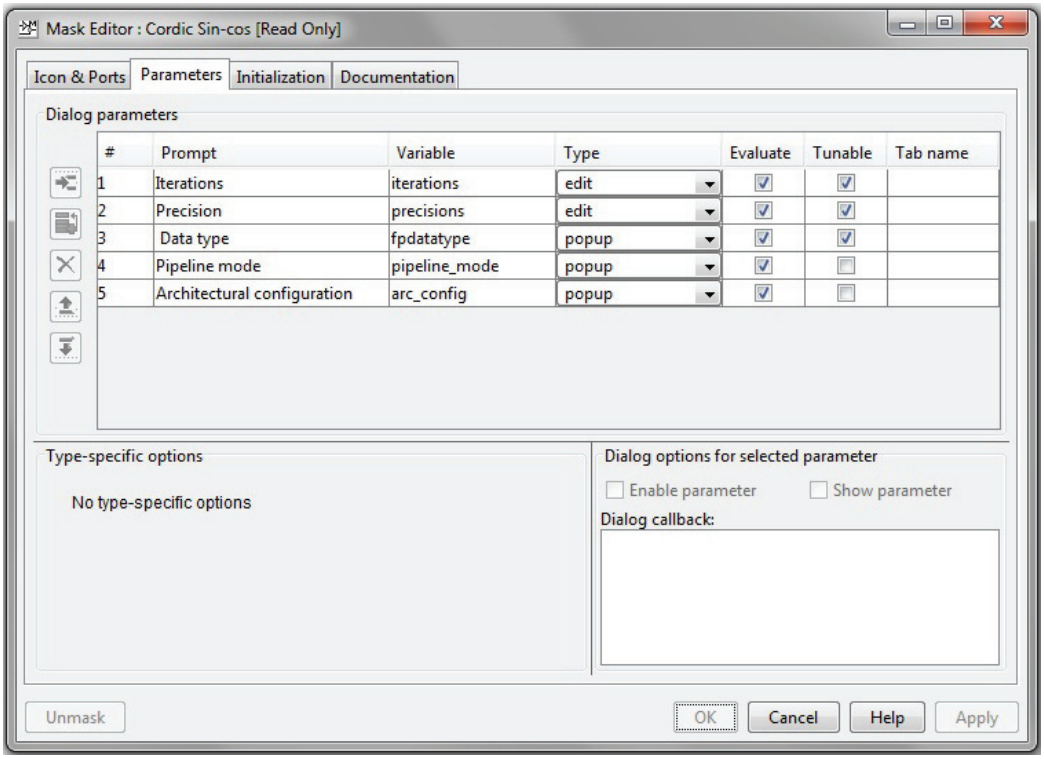

Figure 3: **Mask Editor: CORDIC Sin-cos Parameters**

Latency Handling for AXI Blocks

The CORDIC LogiCORE IP v5.0 block has an AXI stream interface, while the basic blocks do not. This necessitates some input parameters to be propagated to the output side to adjust for latencies. For this purpose, FIFOs are used in the design. An example of how FIFO blocks are used to adjust for latency of AXI blocks is shown in Figure 4. For example, the sin-cos block requires sign information to be made available at the output. The read enable of the FIFOs are driven by the output TVALID of the CORDIC block.

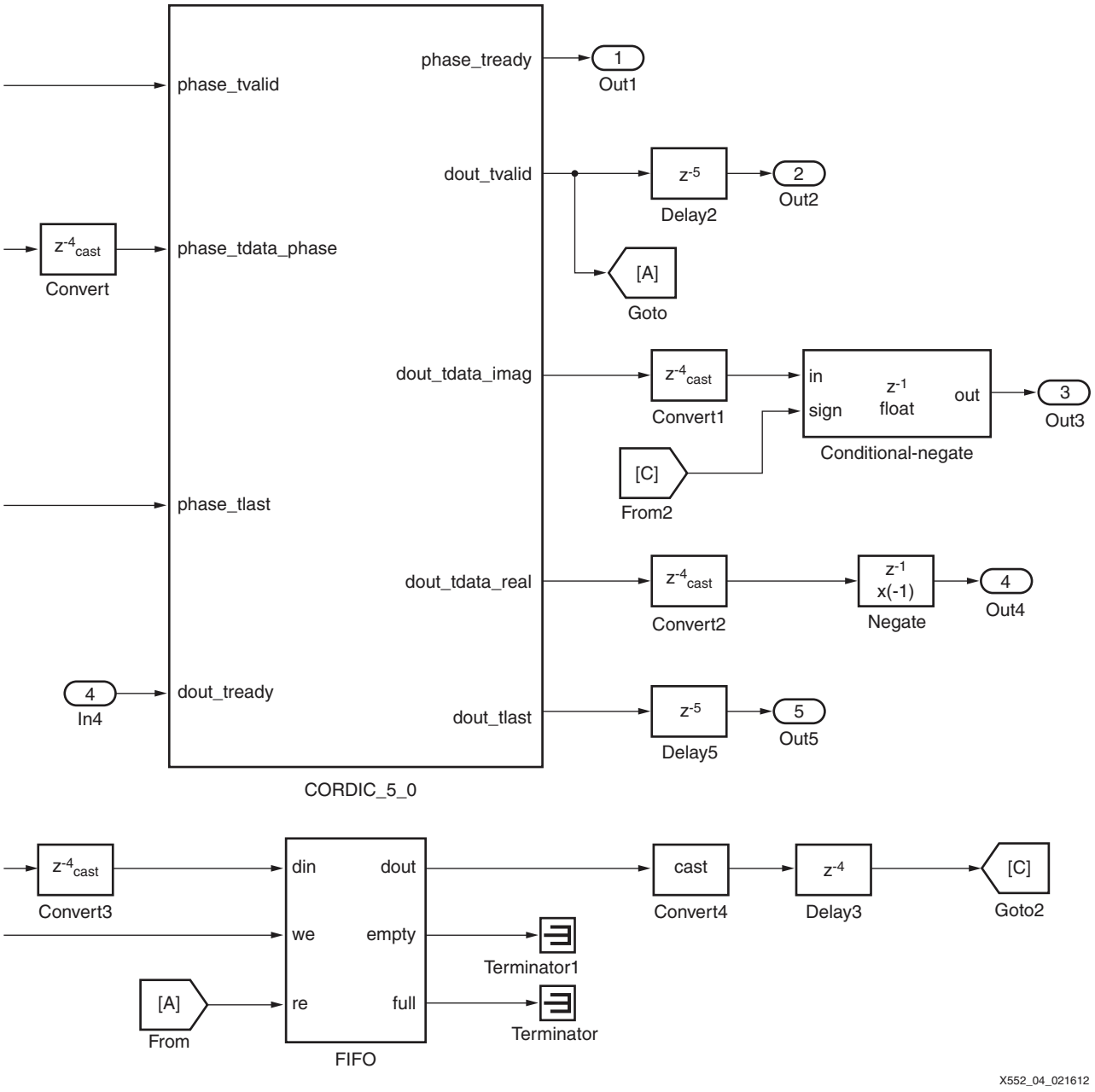

Figure 4: **Usage of FIFOs in Sin-cos Block to Adjust for Latencies in CORDIC LogiCORE IP v5.0 Block**

Split and Merge

For some blocks in the library, the exponent and mantissa at the input can be processed separately as fixed-point numbers (e.g., log, atan) and then recombined to form a meaningful floating-point output. The *split* and *merge* blocks are useful for this operation. The *split* block splits the input floating-point numbers into exponent and mantissa while providing information on whether the input was a not a number (NaN) signal. The *merge* block does the inverse operation, recombining normalized mantissa and exponent to form an output floating-point number.

Remainder

The range reduction step of some algorithms involves computing the modulo of a number (e.g., sin-cos and sinh-cosh). For this purpose, a remainder block is useful and is implemented by means of a floating-point divide and a floor operation. This block is also available to the user as part of the library.

LUTs

The sinh-cosh and power blocks use internal LUTs to implement some portions of the processing. The output range of the above blocks tends to infinity very quickly, and the whole input range might not be of interest to the user. Therefore, control has been provided to the user to select the LUT size based on the input range. For implementation of user-configurable LUTs, ROM blocks have been used, as shown in Figure 5. This option allows the user to optimize an instance of the block for resource utilization.

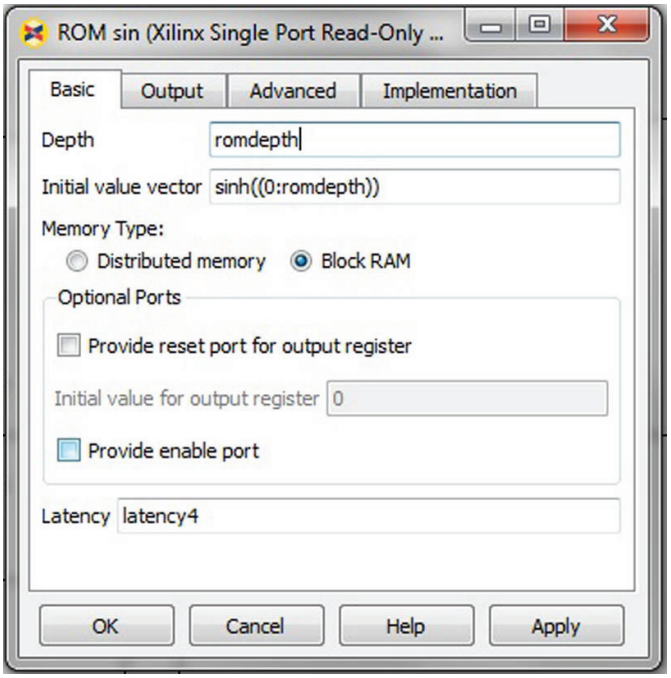

Figure 5: **ROM Block Configured to Support User-Defined Input Range**

Latency Tuning

The latencies of individual blocks in the subsystem are tuned to give approximately 280 MHz for a single precision floating-point data type and 240 MHz for a double precision floating-point data type.

Library Interface Specifications

This section describes the block interface terminology used for each of the different blocks. A sample user interface for sinh and cosh generated by MATLAB is shown Figure 6.

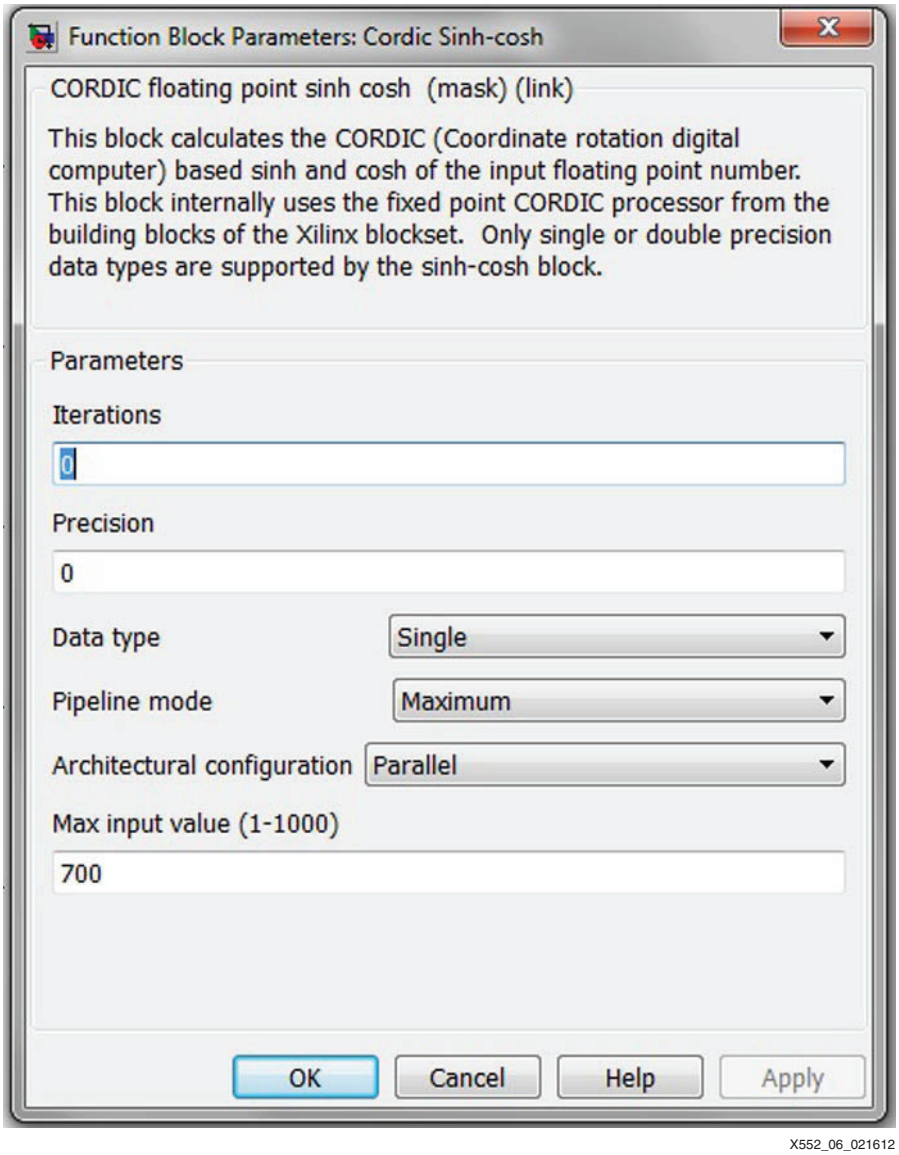

Figure 6: **Block Mask for CORDIC Sinh-cosh Block**

Table 1 describes parameter interfaces used across different blocks in the CORDIC library.

Table 1: **Parameter Interfaces in the CORDIC Library**

Parameter Name	Description
Iterations	Number of add-sub iterations. When set to 0, the number of iterations performed is determined by the required accuracy of the output. The default value of iterations is 0.
Precision	Configures the internal precision of the fixed-point CORDIC. When precision is set to 0, the internal precision is automatically determined.
Data type	The data types supported for this library are Single and Double, as defined by the IEEE 754 standard. Custom data types or parameter inference are not supported.

Table 1: **Parameter Interfaces in the CORDIC Library *(Cont'd)***

Parameter Name	Description
Pipeline mode	The supported pipeline modes are: • **Maximum**: The CORDIC core is implemented with a pipeline after every shift-add substage. • **Optimal**: The CORDIC core is implemented with as many stages of pipelining as possible without using additional LUTs.
Architectural configuration	The architectural configurations supported are: • **Parallel**: CORDIC core has single cycle data throughput and large silicon area. • **Word-serial**: CORDIC core is implemented with multi-cycle throughput and a smaller silicon area.
Maximum input value	This is present for sinh, cosh, and power to optimize the size of the LUT to conserve area. This is expected to be useful because power operations quickly tend toward infinity.

Table 2 describes port interfaces used across various blocks in the CORDIC library.

Table 2: **Port Interfaces in the CORDIC Library**

Port name	Direction	Description
Cartesian_tvalid	I	Input tvalid handshake signal for the AXI stream
Phase_tvalid	I	
Tdata_imag	I	Real portion of the input data
Tdata_real	I	Imaginary portion of the input data
Tdata_phase	I	Angle input used for sin-cos and sinh-cosh modes
Cartesian_tlast	I	Used to specify the last data in a stream
Dout_tready	I	Handshake signal for AXI stream
Cartesian_tready	O	Output tready signal used as handshake for AXI stream
Phase_tready	O	
Dout_tvalid	O	Signal valid data at the output
Dout_tlast	O	Signal last data at the output

Results and Discussion

To make comparisons at the architecture level, estimates have been made of the speed, area, latency, and throughput for the different blocks in the library. The results presented are based on the Virtex-7 devices. The speed information is presented for speed grades -1,-2, and -3. The device utilization figures are presented for -2 devices. The performance summary for the single data type is shown in Table 3.

Table 3: **Block Performance Summary for Single Data Type**

Block name	Mode	-3 Speed Grade	-2 Speed Grade	-1 Speed Grade	Latency	Throughput
		LUTs & Flip-Flops	Slice Registers	Slice LUTs	Block RAM	DSP48E1 Slices
Sin-cos	Word Serial	302	281	223	130	24
		6,514	4,725	4,898	1	9
	Parallel	302	281	223	130	1
		8,605	6,905	6,937	1	9
Sinh-cosh	Word Serial	302	281	223	138	28
		10,020	7,306	7,560	3	27
	Parallel	302	281	223	138	1
		12,453	9,804	9,940	3	27
Power	Word Serial	302	281	223	171	28
		11,049	8,091	8,198	3	40
	Parallel	302	281	223	171	1
		13,489	10,589	10,578	3	40
Log	Word Serial	302	279	223	64	28
		2,460	1,625	1,856	1	8
	Parallel	302	281	223	64	1
		4,944	4,132	4,290	1	8
Atan	Word Serial	302	279	223	67	26
		2,682	1,834	2,025	2	5
	Parallel	302	281	223	67	1
		4,935	4,152	4,257	2	5

The performance summary for the double data type is shown in Table 4.

Table 4: **Block Performance Summary for Double Data Type**

Block name	Mode	-3 Speed Grade	-2 Speed Grade	-1 Speed Grade	Latency	Throughput
		LUTs & Flip-Flops	Slice Registers	Slice LUTs	Block RAM	DSP48E1 Slices
Sin-cos	Word Serial	282	246	215	171	45
		16,598	12,504	12,060	1	20
	Parallel	297	258	222	171	1
		23,437	19,542	18,811	1	20
Sinh-cosh	Word Serial	285	251	217	179	49
		24,183	17,898	18,219	5	73
	Parallel	297	258	222	179	1
		31,607	25,498	25,553	5	73

Table 4: **Block Performance Summary for Double Data Type *(Cont'd)***

Block name	Mode	-3 Speed Grade	-2 Speed Grade	-1 Speed Grade	Latency	Throughput
		LUTs & Flip-Flops	Slice Registers	Slice LUTs	Block RAM	DSP48E1 Slices
Power	Word Serial	285	229	217	212	49
		26,257	19,438	19,663	5	101
	Parallel	297	229	222	212	1
		33,681	27,038	27,997	5	101
Log	Word Serial	283	248	216	85	49
		3,075	2,787	3,575	1	25
	Parallel	297	258	222	85	1
		12,045	10,463	11,020	1	25
Atan	Word Serial	283	247	216	88	47
		5,140	3,319	4,100	3	14
	Parallel	297	258	222	88	1
		12,399	10,636	11,271	3	14

Trade-offs in the Current Approach

In the current approach simple range-enhancement algorithms are used along with a fixed-point CORDIC algorithm. The resulting output is expected to be hardware-efficient and to deliver performance comparable to that of the fixed-point CORDIC block. For example, the underlying CORDIC block in 32-bit mode for fixed-point sin and cos is expected to run at 345 MHz for parallel mode and 222 MHz for word-serial architecture. This compares with 280 MHz achieved for the floating-point CORDIC algorithm in single mode. The percentage of error tends to increase at a lower input range, whereas absolute error tends to increase at a higher range.

Detailed error profiles for different blocks are shown in Figure 7 to Figure 9. The error profiles are presented in terms of units in last place (ULP). The error profiles for sin-cos and sinh-cosh in single mode are given in Figure 7. The error profiles were obtained by providing a ramp signal as the input of the floating-point reference block and comparing it with the Simulink output. It can be seen that the error is maximum when the input value is closest to 0.

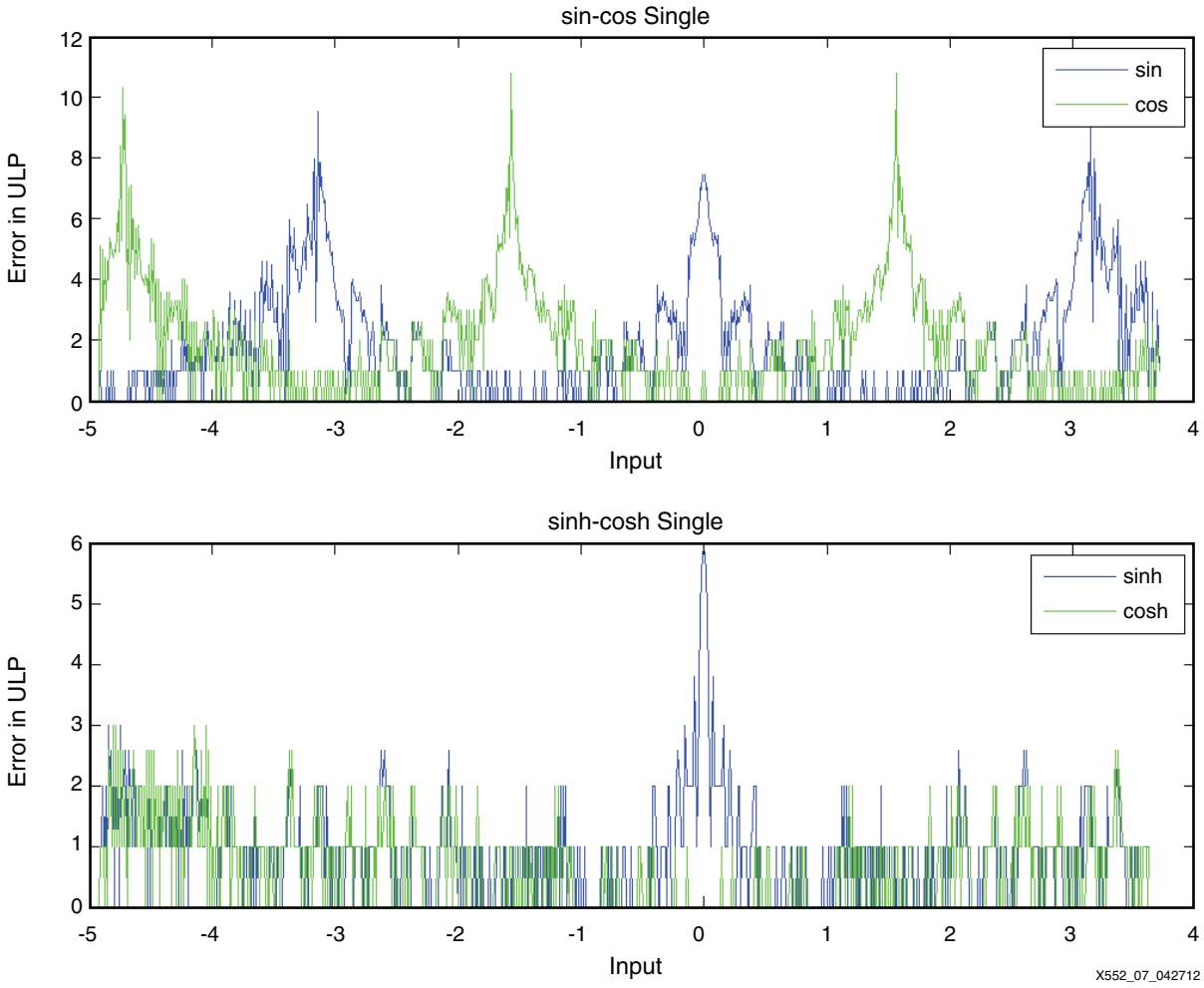

Figure 7: **Error Profile for sin-cos and sinh-cosh**

The error profiles for exponential and log are shown in Figure 8.

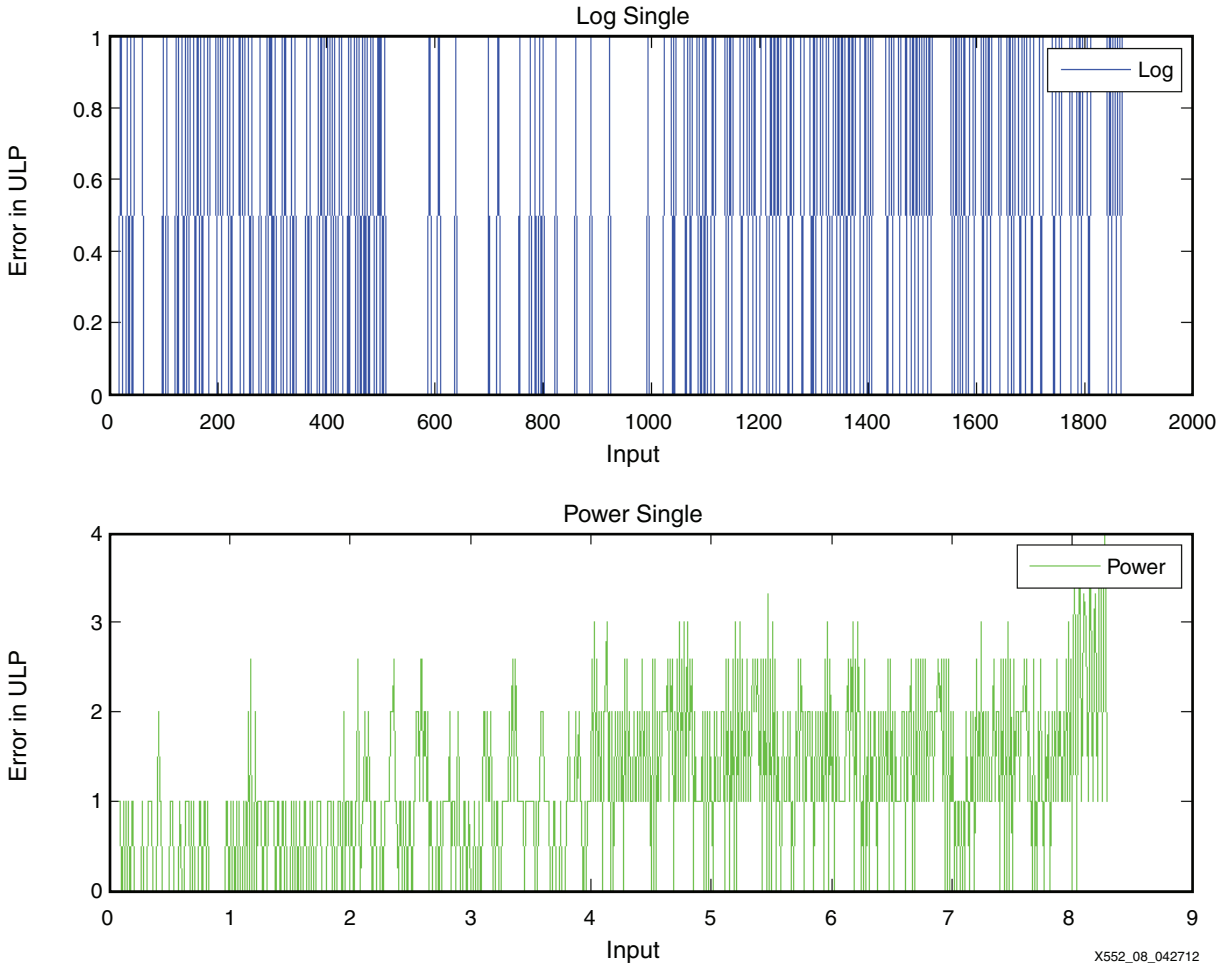

Figure 8: **Error Profile for Log and Power**

The error profile for atan produced by providing a ramp for input 1 and inverse ramp for input 2 is shown in Figure 9.

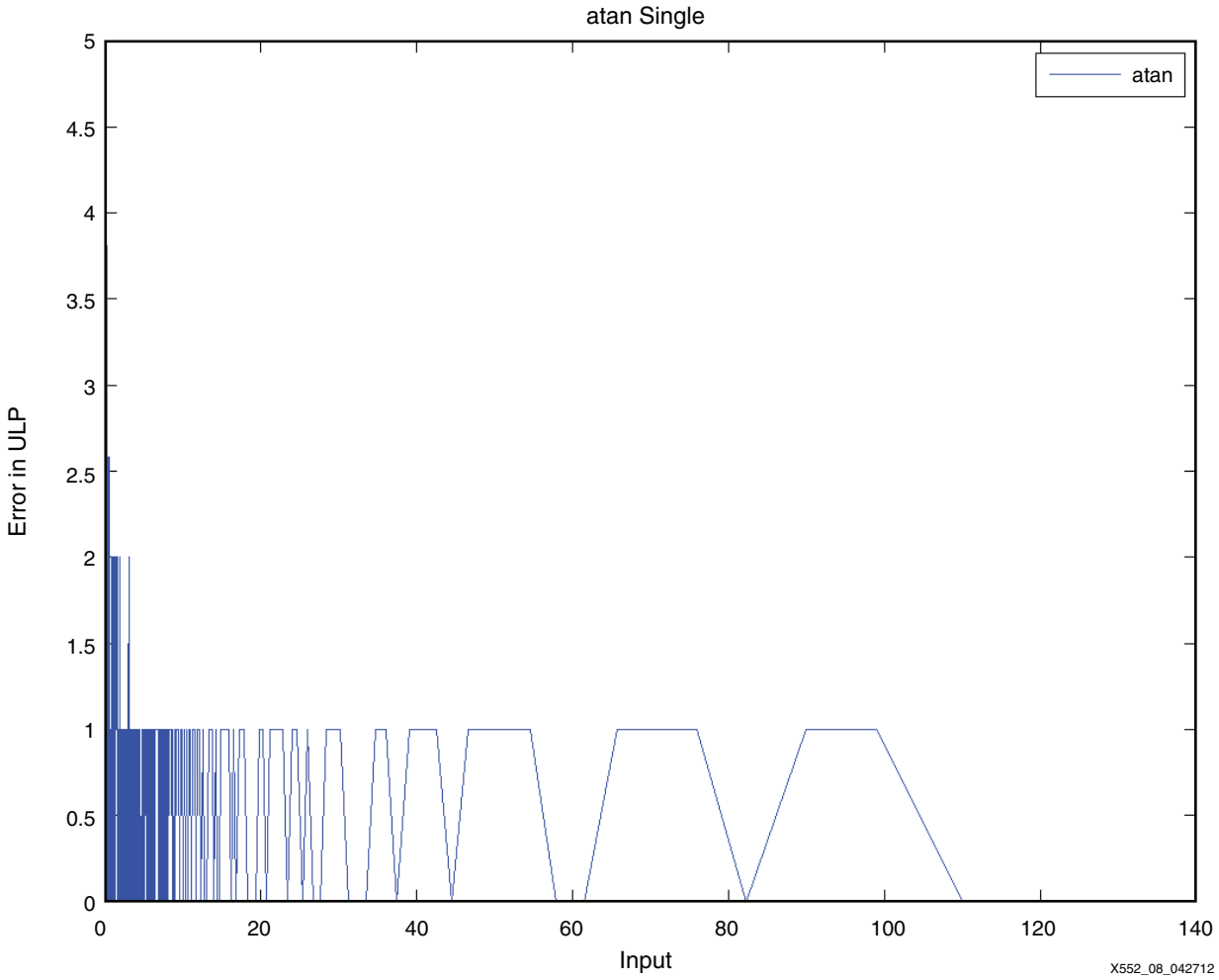

Figure 9: **Error Profile for atan**

Reference Design

The reference design for this application note can be downloaded from:

https://secure.xilinx.com/webreg/clickthrough.do?cid=185372

The reference design matrix is shown in Table 5.

Table 5: **Reference Design Matrix**

Parameter	Description
General	
Developer name	Nikhil Dhume and Ramakrishnan Srinivasakannan
Target devices (stepping level, ES, production, speed grades)	Spartan-6, Virtex-6, 7 Series, and Zynq-7000 devices
Source code provided	Yes
Source code format	System Generator MDL library

Table 5: **Reference Design Matrix *(Cont'd)***

Parameter	Description
Design uses code and IP from existing Xilinx application note and reference designs, CORE Generator software, or third party	Yes
Simulation	
Functional simulation performed	Yes
Timing simulation performed	Yes
Test bench used for functional and timing simulations	Yes
Test bench format	System Generator MDL file
Simulator software/version used	System Generator for DSP, version 13.4 ISE Design Suite 13.4
SPICE/IBIS simulations	No
Implementation	
Synthesis software tools/version used	Xilinx Synthesis Technology (XST) 13.4
Implementation software tools/versions used	System Generator for DSP, version 13.4 ISE Design Suite 13.4
Static timing analysis performed	Yes
Hardware Verification	
Hardware verified	No
Hardware platform used for verification	N/A

Conclusion

A floating-point library for computation of trigonometric, power, and logarithmic operations has been designed by applying range extension algorithms to underlying fixed-point blocks. This approach has been demonstrated to give comparable performance to the underlying fixed-point block. The device utilization, latency, and maximum operating frequency have been documented for all the blocks in the library.

References

This application note uses the following references:

1. Volder, Jack E., *The CORDIC trigonometric computing technique*. IRE Transactions on Electronic Computers, vol. EC-8, September 1959, pp. 330-334 http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5222693
2. IEEE Std. 754-2008, *IEEE Standard for Floating-Point Arithmetic,* August 2008. http://ieeexplore.ieee.org/servlet/opac?punumber=4610933
3. WP409, *High-Level Implementation of Bit- and Cycle-Accurate Floating-Point DSP Algorithms with Xilinx FPGAs*
4. Andraka, Ray. *A survey of CORDIC algorithms for FPGA based computers*, Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, Feb 22–24, 1998. pp. 191–200 http://www.andraka.com/files/crdcsrvy.pdf
5. Lang, T. and E. Antelo, *High-throughput CORDIC-based geometry operations for 3D computer graphics*, IEEE Transactions on Computers, vol. 54, no. 3, March 2005, pp. 347–361 http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1388199

6. DS858, *LogiCORE IP CORDIC Product Specification*
7. UG638, *System Generator for DSP Reference Guide*
8. *Creating a Block Mask* (in MATLAB) http://www.mathworks.com/help/toolbox/simulink/ug/brx7xj4.html

Revision History

The following table shows the revision history for this document.

Date	Version	Description of Revisions
06/01/2012	1.0	Initial Xilinx release.

Notice of Disclaimer