Implementation of a CORDIC based Double-Precision Exponential Core on an FPGA

Robin Pottathuparambil Ron Sass
rpottath@uncc.edu rsass@uncc.edu
Reconfigurable Computing Systems Lab
University of North Carolina at Charlotte

Abstract

Many natural processes exhibit exponential decay and, consequently, computational scientists make extensive use of e^{-x} in computer simulations. Many transcendental functions (sine, cosine, tangent, exponentiation, etc*) are readily and efficiently implemented in hardware using the well known CORDIC algorithm. However, many current FPGA implementations are either fixed point or reduced precision floating point (which suffers from a high average/mean error). These solutions are unacceptable for many computational scientist who require accurate double-precision values.*

This paper presents a direct implementation of an e^{-x} core in a CORDIC-style suitable for FPGA devices. The core generates IEEE 754 standard double-precision exponential values. The implementation described consumes fewer resources than the current approaches in the literature, enabling more e^{-x} cores per FPGA. Specifically, the results show that the implementation achieves correct double-precision e^{-x} values, consuming only 8% of slices on a Virtex 4 XC4VFX60 FPGA.

1 Introduction

Many processes in nature exhibit an exponential decay property. From gravitational forces to gradients of protein concentration, computational scientists make extensive use of e^{-x} in computer simulations of physical phenomena. However, most microprocessors do not provide an exponentiation unit in hardware. Instead the operation (and other transcendal functions) are implemented in software via a combination of look-up tables, floating-point multiplies, and additions. These algorithms are very slow when compared to a hardware implementation [1].

Field-Programmable Gate Arrays (FPGAs) are increasingly being included in commercial high-performance computing systems as "compute accelerators." For many computational scientists, a hardware implementation of e^{-x} would be an appropriate and advantageous use of these resources. Unfortunately, most hardware implementations of e^{-x} described in the literature are unsuitable for scientific applications. Either the implementation does not map well to FPGAs (consuming an unacceptable amount of resources) or preventing multiple instances or the design sacrifices double-precision IEEE 754 compatibility for speed (a trade-off that many computational scientists are unwilling to accept).

This paper describes an implementation of a double-precision, IEEE 754-compatible exponentiation unit. The design is an implementation of the CORDIC algorithm and does not require large look-up tables (making it possible to instantiate multiple parallel cores). The customary algorithm has been slightly modified to reduce the number of logical gates. In addition, the core produces both e^{x} and e^{-x} without the introduction of a floating point divide. The implementation can be readily extended to other transcendental functions (sine, cosine, etc.) and has the potential of being pipelined in the future.

Hardware methods for exponential functions include CORDIC based methods, table look-up methods and polynomial approximation. As the precision of the exponential function increases the above methods will suffer from an exponential increase in the look-up table size and in large word-size multiplications. However the CORDIC [11] method is an iterative algorithm for a two-dimensional vector in linear, circular and hyperbolic coordinate systems, using only add and shift operations. The add and shift operation of the CORDIC [8] algorithm makes it easy to implement the algorithm on the hardware. Increasing the precision (i.e single to

double precision), linearly increases the size of look-up table and the adders.

2 Design and Implementation

As defined by Hekstra [5], a floating-point CORDIC, is one in which a floating-point vector is rotated over a floating-point angle. The angle derived depends on whether the mode of operation is vectoring or rotation. The range of angle x is confined to be $x \in (0, \ln(e))$. The basic convergence range of the exponential function is between 0 and 1.1182 [6]. The convergence range is kept less than the maximum value so that precise results can be generated.

This paper concentrates on the rotation mode of CORDIC. The CORDIC method is governed by set of three equations, the angle z, x coordinate and the y coordinate. The x and y coordinate corresponds to $\cosh$ and $\sinh$ respectively. The difference of $\cosh(x)$ and $\sinh(x)$ for $x > 0$ is nothing but e^{-x}. The CORDIC equations for hyperbolic rotations are shown below.

$$x_{i+1} = x_i + y_i \cdot d_i \cdot 2^{-i} \quad (1)$$

$$y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i} \quad (2)$$

$$z_{i+1} = z_i - d_i \cdot \tanh^{-1}(2^{-i}) \quad (3)$$

where

$$d_i = \begin{cases} -1 & \text{if } z_i < 0 \\ +1 & \text{otherwise} \end{cases}$$

The value of d_i indicates the direction of rotation; x and y are the vector components. The CORDIC algorithm iterates to either zero y or zero z depending on the function required. Since our aim is e^{-x}, the set of equations (1 - 3) can be reduced by substituting w for $(x - y)$ as shown in equation (4) below.

$$w_{i+1} = w_i - w_i \cdot d_i \cdot 2^{-i} \quad (4)$$

Hence there are only two equations (3) and (4) that needs to be iterated to calculate the value of e^{-x}. The above two equations (3) and (4) has four major hardware components: A (fixed size) Hyperbolic tangent look-up table, double precision adder/subtractor, 2:1 and 61:1 multiplexer and a subtractor.

The value of $\tanh^{-1}(2^{-i})$ is pre-computed for each i and these values are used to create a look-up table. These hyperbolic arctangent values are stored in the look-up table as IEEE 754 double-precision format with little modification. The look-up table entries depend on the number of iterations. The number of iterations is 61 in our design. These look-up table values are referenced through a 61:1 multiplexer. The look-up table requires about 456 bytes (60×61 bits). The most significant bits for the look-up table entries are the same and hence are stored as a 60-bit values instead of 64 bit. The four most significant bits are concatenated in the design logic, there by reducing the size of the look-up table.

The double precision adder/subtractor is a three-stage pipelined core for addition or subtraction depending upon the operation flag. A 2:1 multiplexer is used to select the initial or the iterated value. A 61:1 multiplexer is used to select the hyperbolic tangent values from the look-up table. The values are selected using the iteration count. A normal subtractor is used to replace the division by two operation. The implementation of division by two operation in IEEE 754 double precision format is done by subtracting the exponent by one and concatenating with the mantissa and the sign bit.

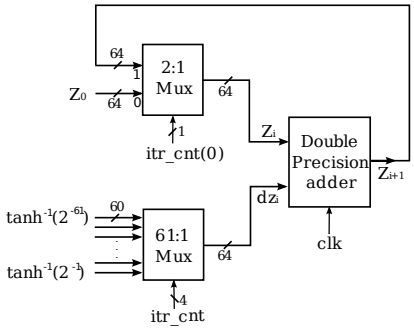

Figure 1: Angle Computation Hardware

The core consists of two main components: angle computation module and the main computation module all are written in VHDL and described in detail below. The angle hardware module shown in Figure 1 is designed in accordance with the equation (3). For calculating a double precision e^{-x}, the iterations needs to done on a 64-bit wide data. The equation requires hyperbolic arctangent values for performing iterations. These values from the look-up table are then fed into the 61:1 multiplexer. The current iteration value is used to select the value from the look-up table, using the multiplexer. The multiplexer value is then added or subtracted from the previous result of the iteration. The result is then fed back for the next iteration. A 2:1 mux is used for choosing between the input angle value (x) and the previous iteration result. When its the first it-

eration, the input angle value is chosen. The result of each iteration step decides whether the next iteration is addition or subtraction operation. If a negative result is obtained then an addition is performed in the next iteration. The last bit of the result, which is the sign bit decides the next iteration addition/subtraction operation. This value is also used for the main iteration hardware.

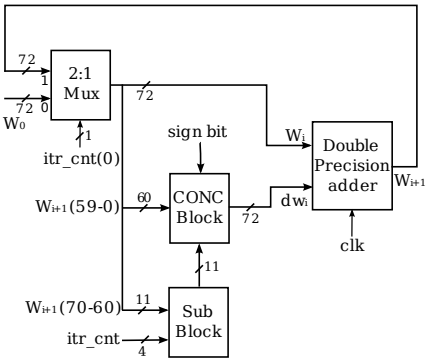

Figure 2: Main Computation Hardware

The main computation hardware module shown in Figure 2 is designed in accordance to the equation (4) and works on an extended floating point format (72 bits). Eleven bit for exponent, sixty bits for mantissa and one sign bit. The initial value is fed into the subtractor or the 'Sub' block which does a subtraction on the exponent part of the double precision value. This corresponds to dividing a double precision value by 2^i. The mantissa and the sign bit is then concatenated in the 'Conc' block with result from the 'Sub' block. The value is then given into the floating point adder/subtractor. The addition/subtraction operation is dependent on the value of previous angle iteration. If the angle iteration gives a negative value in the previous iteration a subtraction is performed or addition is performed. The result is then fed back for the next iteration. The initial value for the iteration is chosen by using a 2:1 mux. The elemental rotations in the hyperbolic coordinate system do not converge, However convergence can be achieved if iterations $4, 13, 40, 121.., k, 4k + 1$ are repeated [9]. Each hardware module is iterated n of times until a satisfiable accuracy is achieved. An $n+1$ number of iteration produces a n bit of accuracy [10]. A total of 61 iterations are required for a precision of 52 bits with eight guard bits. In order to increase the accuracy of the results the iterations are performed on a 72-bit data. These additional 8-bits, called as the guard bits increases the accuracy of the results.

The angle computation hardware and main computation hardware are a separate sub-processes written in VHDL and controlled by a Finite State Machine (FSM). The FSM has four states called as 'idle', 'initial', 'setup' and 'result'. The process waits in the 'idle' state until a valid new data (x) is received (`core_nd = 1`) for computation of $exp(-x)$. Once a value is received, the process transitions into the 'initial' state, where the data for the iteration process is computed. In the 'setup' state the floating point core is fed with the values. Once the values are fed into the core the process transitions into the 'result' state. In the 'result' state the process waits for the results from the floating point adder/subtractor core. Once the results are generated by core, these results are used as inputs for the next iteration. In case of the angle iteration process the hyperbolic arctangent look-up table value is read and is kept ready for the next state and in case of main iteration process the previous/initial value is divided by 2^i and is kept ready for the next state. The 'result' state also setups the signals for the floating point adder/subtractor. Once the data is ready for the next iteration, the iteration count is checked and if the iteration count has not reached a pre-set value, the FSM transitions back to the 'setup' state. If the iteration count has been reached the pre-set value, the FSM outputs the result and transitions into the 'idle' state. The FSM outputs a 72-bit result. The higher 64 bits are used as the result and the lower 8-bits are discarded.

3 Results

The error analysis of the core was done in a range of angle between 0 and $\ln(e)$ i.e $x \in (0, \ln(e)$. However this range is extended by doing a simple pre-computation. This pre-computation would take in a IEEE 754 double precision value and convert it into a input of our convergence range, multiplied by a value in the form of 2^p. The pre-computation is done using simple mathematical identities. The value of p is given as another input to the core. This input corresponds to p left shifts. An exhaustive set of test values ($\ln(2) \times 10^{11}$) were used to test the core. The generated test values were between the input range. The observed maximum error was 2^{-53}. However the average error was 0.047×2^{-53}, which is about 5% of the test cases. This shows that the results are very less prone to errors (only one in twenty on an average). As far as in the literature none of the designs have achieved such a high percentage of accurate

Details	Our values	Chen [2]	Jamro [7]	Doss [4]	Detrey [3]
Style	CORDIC	CORDIC + TD	Table-Driven	Table-Driven	Table Driven
Precision	Double	Single, Double	Double	Single	Single
Error	2^{-53} (max)	0.8735×2^{-53} (max)	0.4708 (Mean)	ND	ND
Guard bits	8 bits	ND	4 bits	ND	5 bits
FPGA	Virtex 4 XC4VFX60	ND	Virtex 4 LX200	Virtex II 4000	Virtex II XC2V1000
Core frequency	100 MHz	ND	166 MHz	85 MHz	100 MHz
Slices	2024	ND	5000	5564	948
DSP48/MUL18	3	ND	0	ND	ND
ROM table size	3.57 Kbits	81.875 Kbits	108 Kbits	ND	ND
Latency of core	258 clk cyc	86 gate delays	27 clk cyc	ND	85 ns

Table 1: Performance Results of the Core (ND means no data available in reference)

results. The design uses eight guard bits to achieve such a high number of the precise results. The core iterates 64 number of times to get a accurate results. As discussed earlier, since the core requires 53 iterations for IEEE 754 double precision standard and with 8 guard bits, the number of iterations are 61. Iterations 4, 13, 40 are repeated for convergence. The above iterations sums up to 64. Since 64 iterations are performed, the latency of the core is 64×4 clock cycles. Every iterations takes about four clock cycles. The total latency is about 258 clock cycles.

The proposed core has about 2024 slices and also uses three DSP48 cores for the entire design. The smaller size of the core was achieved by having a smaller look-up table with double-precision adder/subtractor. The small foot print of the core helps in easily adding the core to any application. It is also worth nothing that the small size of core helps in increasing the number of core units per FPGA and also aids in easily designing a pipelined version of the core. The implementation results are shown in table 1. The table also compares the existing implementation. Our design though has a higher latency, has very less average error, slices and ROM table size. The latency of the core can be easily reduced by having more number of cores computing e^{-x} values at the same time, there by reducing the average latency time.

4 Conclusion

This paper describes a modified CORDIC FPGA double-precision exponential core. The core is only 8% of the total slices on Xilinx Virtex 4 XC4VFX60 device and multiple, parallel cores can be instantiated. The maximum error of the designed core was 2^{-53}. The average error was 0.047×2^{-53}.

References

[1] H. T. Bui and S. Tahar. Design and synthesis of an IEEE-754 exponential function. In *1999 IEEE Canadian Conference on Electrical and Computer Engineering*, volume 1, pages 450–455, Edmonton, Alta., Canada, 1999.

[2] C. Chen, R.-L. Chen, and M.-H. Sheu. A fast additive normalization method for exponential computation. In *DSD '03: Proceedings of the Euromicro Symposium on Digital Systems Design*, page 286, Washington, DC, USA, 2003. IEEE Computer Society.

[3] J. Detrey and F. de Dinechin. Parameterized floating-point logarithm and exponential functions for fpgas. *Microprocess. Microsyst.*, 31(8):537–545, 2007.

[4] C. C. Doss and J. Robert L. Riley. Fpga-based implementation of a robust ieee-754 exponential unit. In *FCCM '04*, pages 229–238, Washington, DC, USA, 2004. IEEE Computer Society.

[5] G. Hekstra and E. F. A. Deprettere. Floating-point CORDIC. In E. E. Swartzlander, M. J. Irwin, and J. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 130–137, Windsor, Canada, 1993. IEEE Computer Society Press, Los Alamitos, CA.

[6] X. Hu, R. G. Harber, and S. C. Bass. Expanding the range of convergence of the cordic algorithm. *IEEE Trans. Comput.*, 40(1):13–21, 1991.

[7] E. Jamro, K. Wiatr, and M. Wielgosz. Fpga implementation of 64-bit exponential function for hpc. In *FPL 2007*, pages 718–721, 2007.

[8] U. Meyer-Bäse, A. Meyer-Bäse, and W. Hilberg. Coordinate rotation digital computer (cordic) synthesis for fpga. In *FPL '94*, pages 397–408, London, UK, 1994. Springer-Verlag.

[9] B. Parhami. *Computer arithmetic: algorithms and hardware designs*. Oxford University Press, Oxford, UK, 2000.

[10] J. Valls, M. Kuhlmann, and K.K.Parhi. Efficient mapping of cordic algorithms on fpga. In *SiPS 2000: IEEE Workshop on Signal Processing Systems*, pages 336–345, 2000.

[11] J. E. Volder. The cordic trigonometric computing technique. *IRE Transactions on Electronic Computers*, EC-8, no. 3:330–334, 1959.