Powering (x^y) architecture using hyperbolic CORDIC

Abdulraheem Aljarrah, Karam Abughalieh
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: aaljarrah@oakland.edu, abughalieh@oakland.edu

***Abstract*— This work presents FPGA architecture to calculate power operation based on hyperbolic CORDIC floating point format. The system is implemented on Zynq-7000 ARM/FPGA SoC Development Board. The system is implemented using IEEE-754 standard single precision computations. An IP is created and interfaced to Zybo board ARM processor via AXI-FULL interface. Simulation results and real implementation experiment are conducted with successful convergence to expected values.**

I. INTRODUCTION

A power is an exponent to which a given quantity is raised. It involves two numbers, the base x and the exponent y and its written as x^y. Powering is broadly used in wide rang of fields like economics, biology, chemistry, physics, and computer science, with applications such as compound interest, population growth, chemical reaction kinetics, wave behavior, and public-key cryptography.

The high cost of powering computation in addition to showing our deep understanding of problem solving and ability to design embedded systems design using FPGA were the motivations behind this project.

The rest of the papers is organized as follows, single precision floating point number system is introduced in sections 2, the expanded hyperbolic CORDIC is explained in section 3, Section 4 of this report explains the methodology adopted in order to implement and complete this project. In sections 4 and 5 the experimental setup and result are discussed and after that the paper is concluded.

II. SINGLE PRECISION FLOATING POINT NUMBER SYSTEM

Single-precision binary floating-point is used due to its wider range over fixed point compared to the same bit-width. It's an IEEE standard arithmetic that requires 32-bit word as in Figure 1 [1], with 8 bit reserved as the exponential and 23 for the fraction part and the last bit for the sign. It is represented as follows:

$$X = \pm 1.f\ x^e \tag{1}$$

Where $e = Exponent - 127$, and f is the Mantissa. Equation (1) applies for Ordinary numbers where the range of e is $[-e^{Exponent-1} + 2, e^{Exponent-1} - 1$.

Figure 1 IEEE 754 a 32 bit

When the Exponent field is a string of ones and the Mantissa is zero, the represented number is $\pm\infty$, the sign is determined based on the sign bit. Not a Number (NaN) case is when the exponent is a string of ones and the Mantissa is not equal to zero. Zero case is when the Exponent and Mantissa both equal to zero.

Equation (2) is used when the Exponent equal zero and mantissa is anything. This case is known as Denormalized Numbers.

$$X = \pm 0.f\ x^e \tag{2}$$

III. EXPANDED HYPERBLIC CORDIC

Hyperbolic CORDIC is used to compute hyperbolic functions in efficient and fast way. The problem is it has very limited range [2] which can be extended using negative iterations to produce what called Extended Hyperbolic CORDIC [3]. Equations (3) and (4) show the algorithm equations:

$$i \le 0 : \begin{cases} x_{i+1} = x_i + \delta_i y_i (1 - 2^{i-2}) \\ y_{i+1} = y_i + \delta_i x_i (1 - 2^{i-2}) \\ z_{i+1} = z_i - \delta_i \theta_i,\ \ \theta_i = Tanh^{-1}(1 - 2^{i-2}) \end{cases} \tag{3}$$

$$i > 0 : \begin{cases} x_{i+1} = x_i + \delta_i y_i 2^{-i} \\ y_{i+1} = y_i + \delta_i x_i 2^{-i} \\ z_{i+1} = z_i - \delta_i \theta_i,\ \ \theta_i = Tanh^{-1}(2^{-i}) \end{cases} \tag{4}$$

Equation (3) shows the negative iteration for $M+1$ iterations ($i=-M,\ldots,-1,0$). While 4 applied for N iterations with positive indices ($i=1,2,\ldots,N$), To ensure the convergence, iteration $4,13,40,\ldots,k,3k+1$ must be repeated. In this work negative iteration M = 5, and positive iteration N = 20 are chosen. The value of δ_i depends on the operation mode:

$$\begin{aligned} &Rotation: \delta_i = -1\ if\ z_i < 0;\ +1\ , otherwise \\ &Vectoring: \delta_i = -1\ if\ x_i y_i \ge 0;\ +1\ , otherwise \end{aligned} \tag{5}$$

The x_n, y_n, z_n converges to the following values depending on the operation mode:

$$Rotating: \begin{cases} x_n = A_n(x_0 coshz_0 + y_0 sinhz_0) \\ y_n = A_n(y_0 coshz_0 + x_0 sinhz_0) \\ z_n = 0 \end{cases} \quad (6)$$

$$Vectoring: \begin{cases} x_n = A_n\sqrt{x_{in}^2 + y_{in}^2} \\ y_n = 0 \\ z_n = z_{in} + Tanh^{-1}\dfrac{y_{in}}{x_{in}} \end{cases} \quad (7)$$

Where

$$A_n = (\prod_{-M}^{0}\sqrt{1-(1-2^{i-2})^2})\prod_{i}^{N}\sqrt{1-2^{-2i}} \quad (8)$$

For the chosen M and N A_n = 5.038156454149566 * e-4

By using well chosen values for the input hyperbolic the CORDIC could converge to coshx, sinhx and Tanh^{-1}. The architecture of the extended hyperbolic CORDIC structure is presented in Figure 2, where it consists of two stages , one for negative iterations and the second for positive iterations. Each stage requires different finite state machine to control counters, registers, multiplexers and the adders/subtractors units , two floating point shifters, a look-up table and multiplexers. The positive iteration stage requires five floating point adder/subtractor while the positive stage requires three.The used B in our project is 32 bits, 8 for the exponent and 23 for fraction part providing a range of [1.175×10^{-38} , 3.403×10^{38}]. Large dynamic range is required for the powering operation x^y since e^x, $\ln x$ could have huge grow or decrease between their inputs and the output due to the nature of these function.

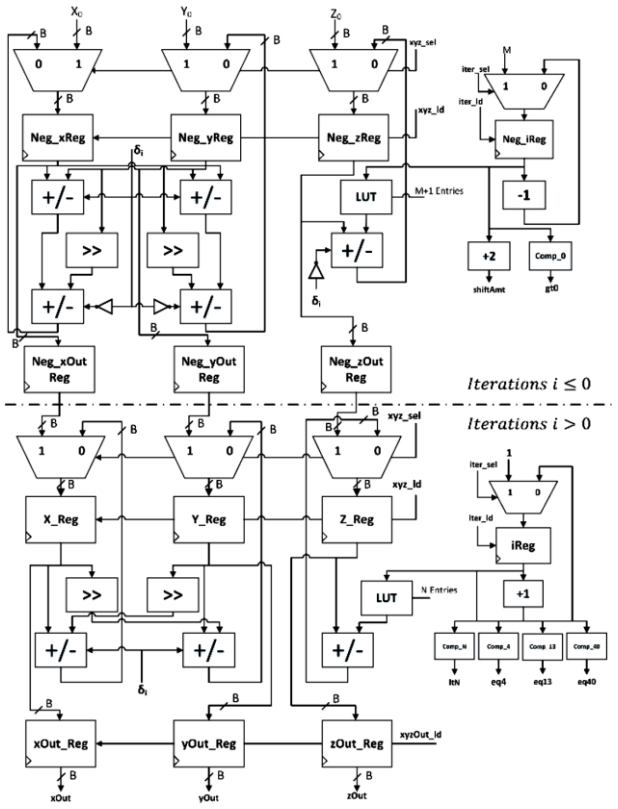

Figure 2 Extended Hyperbolic CORDIC Engine

IV. METHODOLOGY

A. Powering x^y Archticure

The design aims to compute x^y , which could be achieved by getting e^{ylnx} , here we have two functions the exponent and the natural logarithm and both can be obtained from hyperbolic functions as follows:

$$e^x = \cosh x + sinh\, x \quad (9)$$

Which can be obtained by setting the input to the CORDIC algorithm in the rotation mode to $x_0 = y_0 = \frac{1}{A_n}, z_0 = x$

To obtain

$$x_n = \cosh x + sinh\, x \quad (10)$$

Equation 9 is e^xdefinition using hyperbolic functions, in our case to get e^{lnx} we set $z_0 = ylnx$, but to dot that a prior stage of CORDIC in vectoring mode is required to get the natural logarithm using the math identity in (11)

$$\tanh^{-1}x = \frac{1}{2}\ln\frac{1+x}{1-x} \quad (11)$$

By setting $x_0 = x+1, y_0 = x-1, z_0 = 0$, the output z_n converges to $\frac{1}{2}lnx$, after that z_n is multiplied by $2y$ to get $z_n = ylnx$.

The implementation of x^yusing extended CORDIC is illustrated Figure 3, a finite stat machine utilizes the Expanded CORDIC twice as in the following steps:

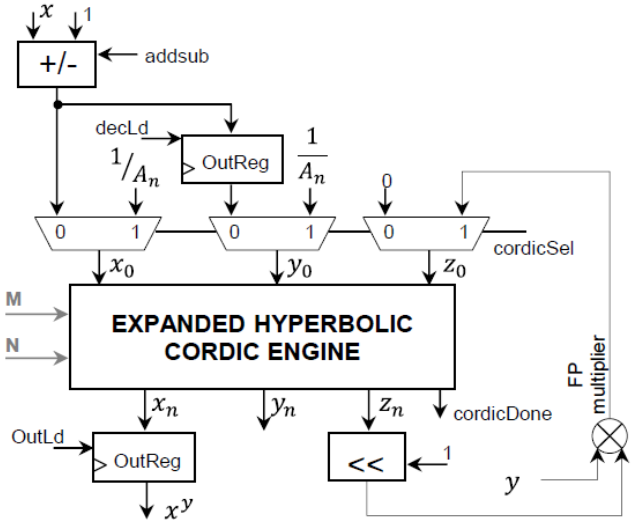

Figure 3 Powering xy implementation

1. Use vectoring mode to get $z_n = \ln x/2$ by loading $x_0 = x+1, y_0 = x-1, z_0 = 0$, this is done using the addsub unit on the top left corner and the multiplexers at the input stage.

2. To get $yln\,x$, first z_n is multiplied by 2 using the shifter , then the floating point multiplier is used to multiply $2z_n$ output by y to get $yln\,x$.
3. Use the current output of z_n as the next input to the CORDIC block in rotating mode. With $x_0 = y_0 = 1/A_n$, this is again done using the multiplexers at the input stages. The output $x_n = e^{ylnx} = x^y$is obtained in OutReg.

B. AXI4 Interface

In order to interface our hyperbolic CORDIC IP to AXI the 32 bits iFIFO, two registers are implemented to buffer the required 64 bits input to the CORDIC engine, as in Figure 4, the out is directly buffered to the oFIFO, the whole process is controlled using the red FSM illustrated in Figure 5.

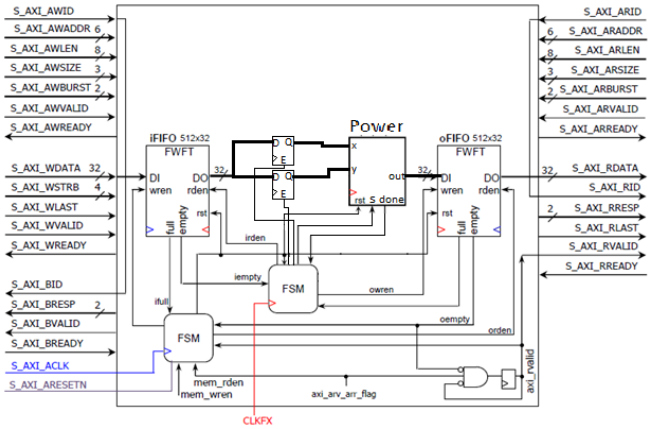

Figure 4 AXI4,FIFO and Power Interface

Red is designed to insure proper reading for the inputs x,y and the output of the powering block, after reset state, the FSM will check for data availability in iFIFO twice to load x and y, after that powering block is activated, the output ready signal and empty oFIFO required to to pass the data out to oFIFO.

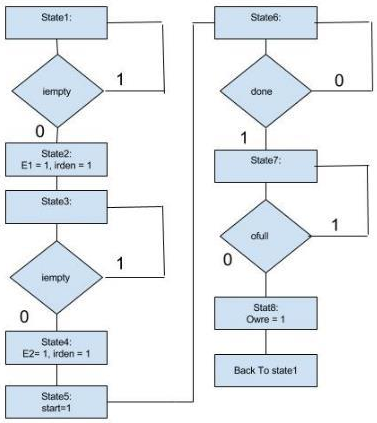

Figure 5 Red FSM Design

V. Experimental Setup

Each sub-system was tested and verified at every stage of the development of powering system using Vivado. MATLAB was used to verify the test examples which were hardcoded into the SDK code which tested and verified with combination of our expanded CORDIC implementation using the Digilent ZYBO board. The board hosts a Xilinx Zynq Z-7010 SoC, 512 MB of DDR3 RAM, and several IO interfaces, i.a., an SD card slot. The Zynq-7000 SoC combines an Xilinx 7-series field programmable gate array (FPGA) and a state-of-the-art hard macro comprising a 650-MHz dual-core ARM Cortex-A9 processor, IO modules, and memory controllers

After the system was verified larger set of inputs is used and fed to the system using text file on SD card, the results also were written to text file and compared with MATLAB results.

VI. Results

The powering operation was simulated validated. The output was ready in 60 cycles, Figure 6 shows 1376.76269523125^0.1 example, the result was 2.056071, while in MATLAB it is 2.0600886.

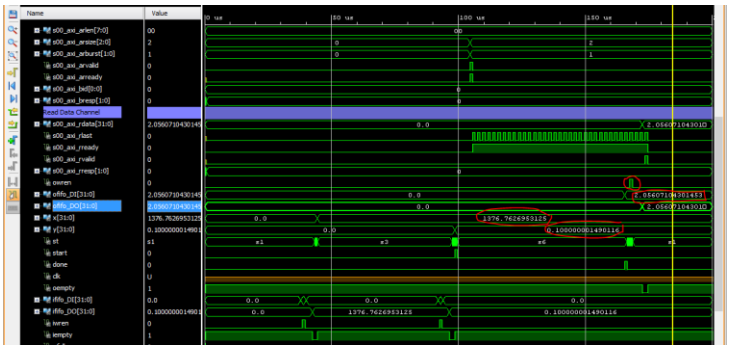

Figure 7 1376.76269523125^0.1 example

The error could be referred to the nature of CORDIC itself where it gives approximate value and to the number of bits used in the design. Figure 7 shows the error defined as in (12) for input range of $0 \le x < 1500$ and $0 \le y \le 1$

The whole used range of output x^y is plotted in figure 8.

$$relative\ error = \frac{Matlab\ Result - CORDIC\ Result}{Matlab\ Result} \qquad (12)$$

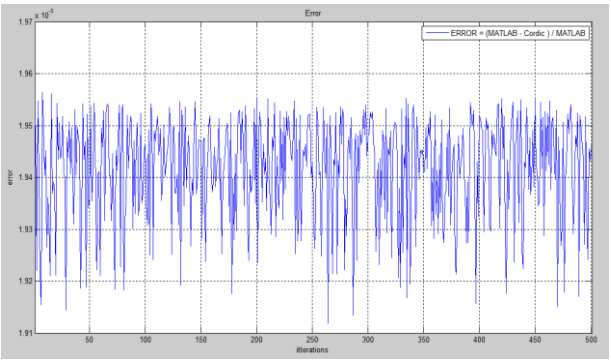

Figure 7 Relative error based on MATLAB results

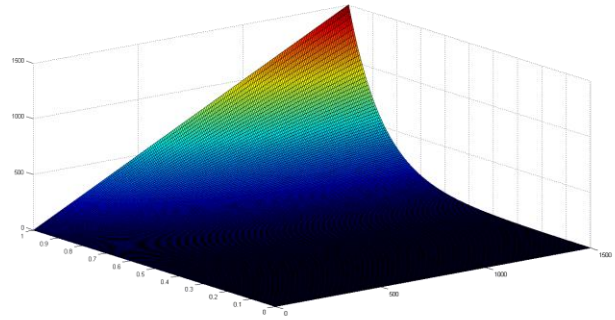

Figure 8 3-D plot for x^y

CONCLUSIONS

In this project, a powering architecture for single precision for x^y was presented and implemented on Zybo 7000 board. The use of floating point arithmetic in addition to expanded CORDIC approach provides higher accuracy and larger dynamics range. The extra phase of negative iterations in the expanded hyperbolic CORDIC increases the accuracy in a noticeable way.

Overall, this project was a satisfying experience where we implemented our understanding of what we learned in the course.

REFERENCES

[1]. IEEE Standard for Floating-Point Arithmetic, ANSI/IEEE Standard 754-2008, Aug. 2008.

[2]. X. Hu, R.G. Harber, S.C. Bass, "Expanding the range of convergence of the CORDIC algorithm," IEEE Transactions on Computers, vol. 40, no. 1, pp. 13-21, Jan. 1991.

[3]. D. Muñoz, D. Sanchez, C. Llanos, M. Ayala, "FPGA-based floating point library for CORDIC algorithms," in Proceedings of the 2010 Southern Programmable Logic Conference (SPL' 2010), March 2010, pp. 55-6