Floating Point CORDIC Based Power Operation

Kazumi Malhan, Padmaja AVL
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: kmalhan@oakland.edu, langaluruvenkat@oakland.edu

***Abstract*—This project presents an architecture to calculate floating point power operation based on hyperbolic cordic. The coding is performed in parameterized method to support numbers of different floating point format. In this paper, the system is implemented for IEEE-754 standard Single precision and Double precision floating point format. The IP is created using AXI4-Full bus interface for board implementation. SD card interface also developed to support reading inputs from and writing results to SD card. The paper discusses methodology, implementation, accuracy of hardware, resource usage, and possible improvements.**

I. INTRODUCTION

The goal of this project is to create power operation digital circuit that is based on hyperbolic cordic. Since we have used fixed point for most of the implementation in class, floating point format is chosen for this project.

The project was developed inside out, meaning developing the core center piece (extended hyperbolic cordic) first, then wrapping with top level file of power operation, and so on. The hardware was created in parameterized fashion to support different floating point format. Until FIFO interface, all development was done using Vivado. After creating power operation IP with AXI4-full interface, development shifts to SDK for software side. SD card interface was developed to pass large number of input combinations to IP, which is topic outside of the class.

The motivation behind this particular project was to show deep understanding of design and implementation of data path, to show the effectiveness of parameterized vhdl coding, and to demonstrate the powerful combination of software and hardware programing. The design decisions were based to maximize the usefulness of digital circuit.

This report explains the design process of each component, how the hyperbolic cordic and power algorithms are implemented, experimental setup, analysis of results and accuracy, resource usage, and possible improvements.

II. METHODOLOGY

A. Floating Point Number System

There are three standard types in IEEE floating point arithmetic: single precision, double precision and extended precision. Single precision numbers require a 32-bit word with 8 exponential bits and 23 fractional bits.

The +/- refers to the sign of the number, a zero bit being used to represent a positive sign. The representation for zero requires a special zero bit string for the exponent field as well as a zero bit string for the fraction, i.e.
0 00000000 00000000000000000000000

$\pm$	$a_1 a_2 a_3 \dots a_8$	$b_1 b_2 b_3 \dots b_{23}$

If exponent bitstring $a_1 \dots a_8$ is	Then numerical value represented is
$(00000000)_2 = (0)_{10}$	$\pm(0.b_1b_2b_3\dots b_{23})_2 \times 2^{-126}$
$(00000001)_2 = (1)_{10}$	$\pm(1.b_1b_2b_3\dots b_{23})_2 \times 2^{-126}$
$(00000010)_2 = (2)_{10}$	$\pm(1.b_1b_2b_3\dots b_{23})_2 \times 2^{-125}$
$(00000011)_2 = (3)_{10}$	$\pm(1.b_1b_2b_3\dots b_{23})_2 \times 2^{-124}$
$\downarrow$	$\downarrow$
$(11111100)_2 = (252)_{10}$	$\pm(1.b_1b_2b_3\dots b_{23})_2 \times 2^{125}$
$(11111101)_2 = (253)_{10}$	$\pm(1.b_1b_2b_3\dots b_{23})_2 \times 2^{126}$
$(11111110)_2 = (254)_{10}$	$\pm(1.b_1b_2b_3\dots b_{23})_2 \times 2^{127}$
$(11111111)_2 = (255)_{10}$	$\pm\infty$ if $b_1 = \dots = b_{23} = 0$, NaN otherwise

All the lines of Table above except the first and the last refer to the normalized numbers, i.e. all the floating point numbers which are not special in some way. Note especially the relationship between the exponent bit string a1a2a3…a8 and the actual exponent E, i.e. the power of 2 which the bit string is intended to represent. We see that the exponent representation does not use any of sign-and-modulus, 2's complement or 1's complement, but rather something called biased representation: the bit string which is stored is simply the binary representation of E + 127. In this case, the number 127 which is added to the desired exponent E is called the exponent bias. For example, the number
$1 = (1.000\dots 0)_2 * 2^0$ is stored as
0 01111111 00000000000000000000000

Here the exponent bit string is the binary representation for 0 + 127 and the fraction bit string is the binary representation for 0 (the fractional part of 1.0).

The range of exponent field bit strings for normalized numbers is 00000001 to 11111110 (the decimal numbers 1 through 254), representing actual exponents from $E_{min} = 126$ to $E_{max} = 127$. The smallest normalized number which can be stored is represented as
0 00000001 00000000000000000000000

Meaning $(1.000\dots0)^2 * 2^{-126}$, i.e. 2^{-126}, which is approximately $1.2 * 10^{-38}$, while the largest normalized number is represented as
0 11111110 11111111111111111111111
Meaning $(1.111\dots1)2 * 2^{127}$, i.e. $(2- 2^{-23}) * 2^{127}$, which is approximately $3.4 * 10^{38}$.

For many applications, single precision numbers are quite adequate. However, double precision is a commonly used alternative. In this case each floating point number is stored in a 64-bit double word with 11 exponential bits and 52 fractional bits.

The ideas are all the same; only the field widths and exponent bias are different. Clearly, a number like 1/10 with an infinite binary expansion is stored more accurately in double precision than in single, since b1,….., b52 can be stored instead of just b1,….,b23.

There is a third IEEE floating point format called extended precision. Although the standard does not require a particular format for this, the standard implementation used on PC's is an 80-bit word, with 1 bit used for the sign, 15 bits for the exponent and 64 bits for the significand. The leading bit of a normalized number is not generally hidden as it is in single and double precision, but is explicitly stored. Otherwise, the format is much the same as single and double precision.

The comparison between single precision and double precision floating point representation is as follows:

	32 bit (Single)	**64 bit (Double)**
Ordinary Number	-	-
Min	2^{-126}	2^{-1022}
Max	$(2\text{-}2^{-23})\times 2^{127}$	$(2\text{-}2^{-52})\times 2^{1023}$
Exponent bits E	8	11
Range of e	[-126, 127]	[-1022, 1023]
Bias	127	1023
Dynamic Range (dB)	759 dB	6153 dB
Significand range	$[1, 2\text{-}2^{-23}]$	$[1, 2\text{-}2^{-52}]$
Significand bits (p)	23	52

B. Extended Hyperblic Cordic

Hyperbolic cordic is the core of power calculation. The basic hyperbolic cordic has very limited range of convergence, so negative iteration is implemented to increase the range. The equation 1 and 2 shows the hyperbolic cordic algorithms for negative and positive iterations. To ensure the convergence, iteration i = 3k + 1 (4, 13, 40 …) must to be repeated. For this paper, negative iteration M = 5, and positive iteration N = 16 is chosen to have reasonable range for power operation. The convergence bound for basic cordic and expended cordic with M = 5 is shown in table 1.

M	e^x	ln(x)
Basic	[-1.11820,1.11820]	(0,9.35958]
5	[-12.42644, 12.42644]	$(0,6.21539_{EE}10]$

Table 1: Convergence bound for the domain

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

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

Delta in the equation is calculated using following two equation depends on the operation mode (Equation 3). An for this extended hyperbolic cordic is calculated using equation 4. With M = 5 and N = 16, An = $5.0382*10^{-4}$. After sufficient number of iterations, hyperbolic cordic converges to certain values (Equation 5, 6).

$$Rotation: \delta_i = -1 \; if \; z_i < 0; \; +1 \; otherwise$$
$$Vectoring: \delta_i = -1 \; if \; x_i y_i \ge 0; \; +1 \; otherwise \quad (3)$$

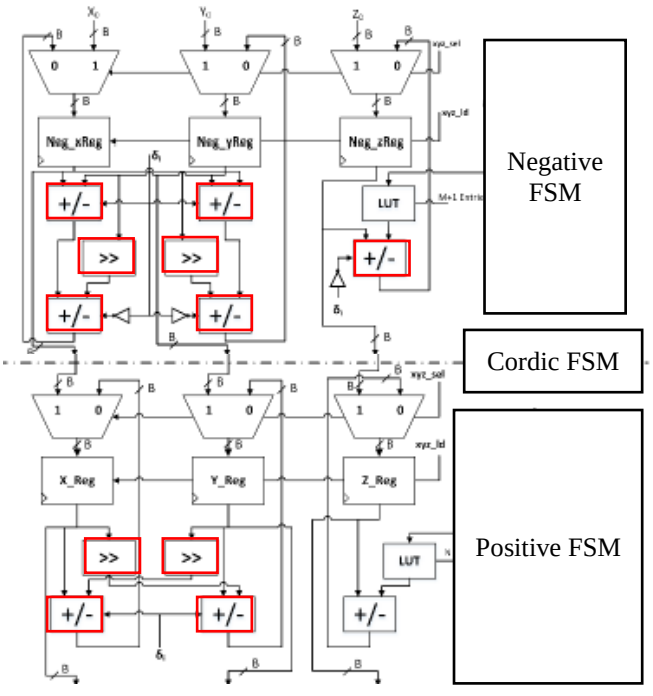
$$A_n = (\prod_{i=-5}^{0} \sqrt{1 - (1 - 2^{i-2})^2})(\prod_{i=1}^{16} \sqrt{1 - 2^{-2i}}) \quad (4)$$

$$Rotation: \begin{cases} x_n = A_n(x_0 coshz_0 + y_0 sinhz_0) \\ y_n = A_n(x_0 coshz_0 + y_0 sinhz_0) \\ z_n = 0 \end{cases} \quad (5)$$

$$Vectoring: \begin{cases} x_n = A_n(x_0 coshz_0 + y_0 sinhz_0) \\ y_n = 0 \\ z_n = z_0 + Tanh^{-i}(\frac{y_0}{x_0}) \end{cases} \quad (6)$$

By using special input combination, we are able to obtain exponential operation and natural logarithm. Exponential is calculated using rotation mode with $x_0 = y_0 = 1/An$, $z_0 = \alpha$, $x_n = \cosh \alpha + \sinh \alpha = e^{\alpha}$. Natural logarithm is calculated using vectoring mode with $x_0 = \beta+1$, $y_0 = \beta-1$, $z_0 = 0$, $z_n = \tanh^{-1}(\beta-1/\beta+1) = \ln(\beta)/2$.

Figure 1: Architecture of extended hyperbolic cordic

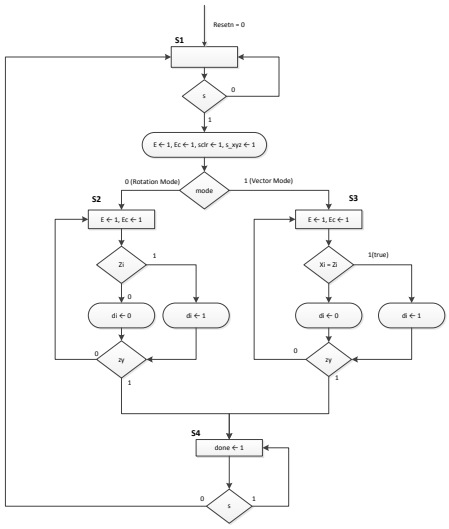


The cordic designed for this project is coded in parameterized fashion to support any format of floating point. It takes total number of bits (N), number of exponent

bits (EXP), and number of fractional bits (FR) as a parameter. It is notable that LUT must contain the pre-calculated values for tanh that will be used for operation.

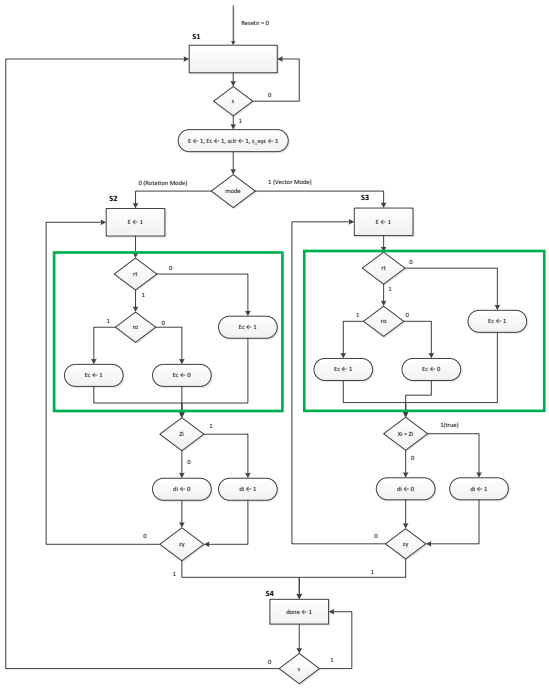
Figure 1 shows the basic architecture of extended hyperbolic cordic. Top half of the circuit is for negative iteration. The first multiplexers select initial inputs at the beginning, and then take previous output from following iteration. Components shown in red box (adder/subtractor, shifter) are floating point supported components which Professor Llamocca has provided. Other components are same as if the system is designed for fixed point format. LUT is hard coded inside the source file. The system supports 64, 32, 24, 16 bit floating point number. Appropriate LUT values are selected by using "if (N = XX) generate" statement. The negative finite state machine (FSM) (figure 2) controls the iteration, shift amount, enable signals, add/sub, and multiplexer select time. Negative FSM counts iteration from -7 to -2 to avoid i-2 operation for different location. After the negative iteration, positive iteration circuit begins to work. Again, multiplexer is located to select output from negative iteration for the first time, then takes previous outputs. Rest of the circuit works similar to negative iteration.

Figure 2: ASM chart of negative FSM



The extended hyperbolic cordic is controlled by three separate FSMs. Positive and negative ASM are show in figure 2 and 3. Both positive and negative FSM keeps track of iteration, and decides the shift amount, sign of add/sub, and enable signals. The challenge we faced is to develop the method to repeat i = 4 and 13. This is achieved by adding code to a counter inside positive FSM that generates flag during iteration 4 and 13, then implement a register to keep track of previous iteration. Also, we had hard time realizing that positive iteration starts from 1, not 0. The green boxed area in positive FSM performs this task. The cordic FSM is a simple FSM that first enables negative iterations. When it receives done from negative, FSM starts the positive.

Figure 3: ASM chart of positive FSM



C. *Power Operation*

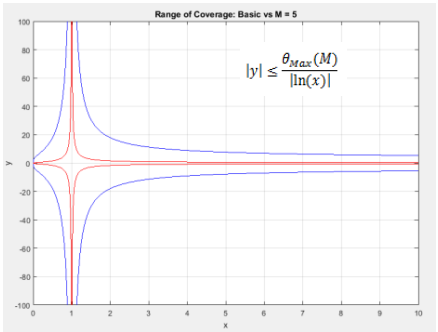
The power operation can be performed by combining exponential and natural logarithm operation. Since hyperbolic cordic performs above two operation, running it twice will generate the power operation. The following list outlines the brief steps to perform cordic based power operation.

Steps to obtain x^y

1. Vectoring mode, provide $x_0 = x+1$, $y_0 = x-1$, $z_0 = 0$
2. You get $z_n = \ln(x)/2$
3. Multiply $\ln(x)/2$ and 2 (by shifting)
4. Multiply $\ln(x)$ and y
5. Rotation mode, provide $x_0 = y_0 = 1/An$, $z_0 = y$.
6. You get $x_n = e^{y\ln x} = x^y$

With implementation of negative iteration, we are able to expand the supporting range of (x,y) input to the power block as shown in figure 4.

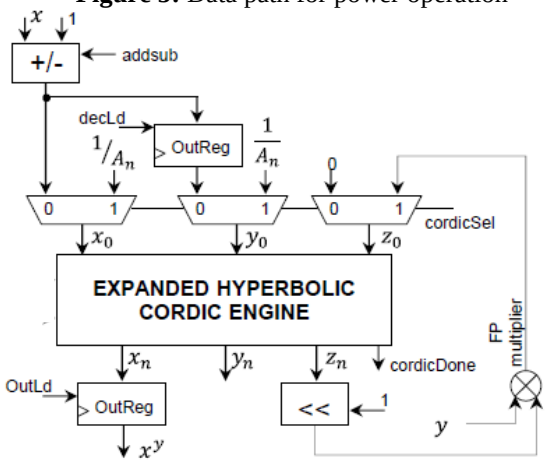
Figure 4: Range of convergence compared between basic cordic and M=5



For power operation, 1/An and other few constants need to be hard coded inside the top level of power vhdl file. As representaiton of number depends on floating point format, "if generate" statement is used to choose among possible formats.

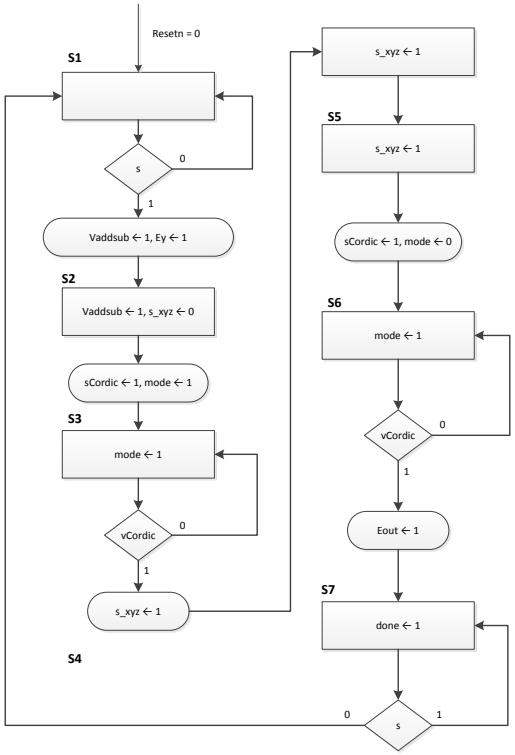
Data path and control unit for power operation is relatively simple once expanded hyperbolic cordic is done. Zn output is connected to shifter which performs the multiply by 2. The multiplier is floating point supported.

Figure 5: Data path for power operation



FSM for power operation is shown in figure 6. It implements the logic outlines as "Steps to obtain x^y" in this section.

Figure 6: ASM chart for Power FSM

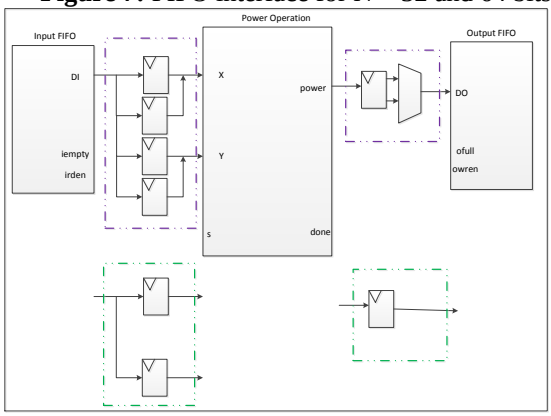


D. *FIFO Interface*

To implement the power operation circuit to Zed board and control from processor, AXI4-bus interface with FIFO was used. Therefore, input and output interface were required to pre and post process the data for power operation.

The interface was developed for both 32 bits and 64 bits. Since AXI4-full interface transmits 32 bit at a time, different circuit was required for two different floating point format. When N = 32, green box circuit is generated, and 2 FSM controls the inputs and outputs. For N = 64, purple box circuit was generated. Input creates 64 bit number by combining two 32 bits number. The output is divided to two 32 bits number before entering FIFO. Again, separate two FSMs were used to control operation.

Figure 7: FIFO interface for N = 32 and 64 bits

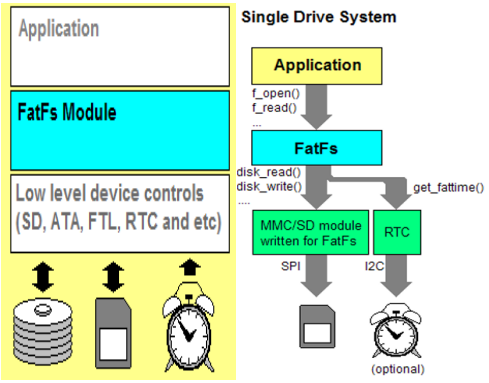


E. *SD Card Interface*

The Secure Digital Memory Card (shortly SD card) is the de facto standard memory card for mobile equipment. The SDC has a microcontroller in it. The flash memory controls (block size conversion, error correction and wear leveling - known as FTL) are completed inside of the memory card. The data is transferred between the memory card and the host controller as data blocks in unit of 512 bytes, so that it can be seen as a block device like a generic hard disk drive from view point of upper level layers.

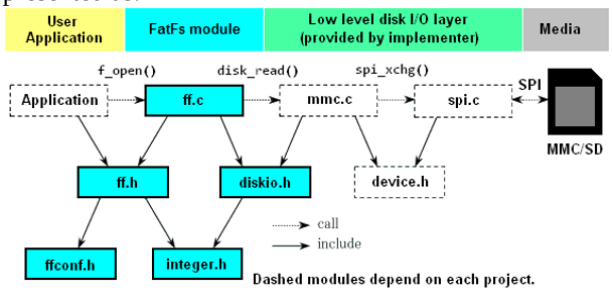
SD card interface uses FATFs file system. Fatfs is a generic FAT file system module for small embedded systems. The Fatfs module is written in compliance with ANSI C (C89) and completely separated from the disk I/O layer. Therefore it is independent of the platform. It can be incorporated into small microcontrollers with limited resource.

System architecture of different layers of hardware and software is SD card IP is created in Xilinx SDK and interfaced with different libraries to support SD card data transfer. Our project uses the XSDPS libraries at driver level. This driver is used to initialize read from and write to the SD card.



Data transfer: The SD card is put in transfer state to read from or write to it and works in polled mode using ADMA2. The default block size is 512 bytes.

File system: The xilffs library is used to read/write files to SD. Application file and functions are developed independently and it supports read from a file in SD card repeatedly until end of the file and after manipulating the data, write back into SD card file in another format. The application level software is written by us. This is pictorially represented as:



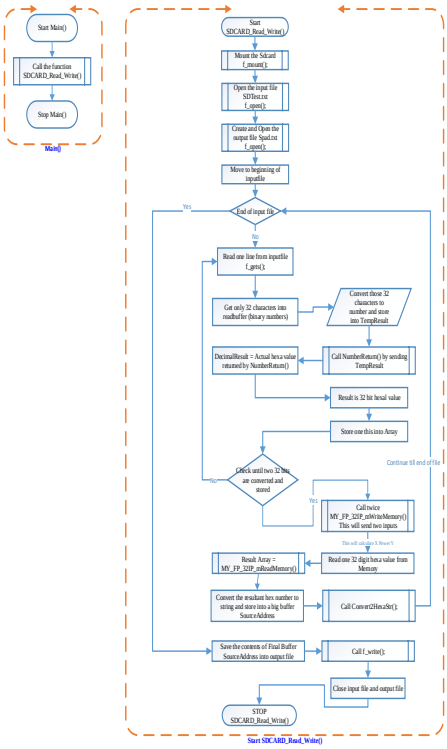
III. EXPERIMENTAL SETUP

For hardware component, test bench was created at every stage of development (hyperbolic cordic, power, FIFO interface). Initially, few random values are tested to confirm the each step of signals. After the confirmation of basic operation, large number of inputs are created using MATLAB and fed to the hardware by reading inputs from text files. Results are compared with MATLAB values and plotted.

For system level simulation, the 32bit and 64bit SD card IPs are created using Vivado and launched into Xilinx SDK for SD card application development and testing. To access the SD driver libraries there are some BSP settings to be modified. To ensure whether this library inclusion happened correctly, browse to project explorer and check xilffs libraries are included and ff.c and corresponding FATFs files are created and linked automatically.

During the implementation, we have encountered "timing violation" error. The Frequency of AXI bus was originally 100 MHz which is then reduced to 10 MHz (50MHz didn't work). The root cause of the problem is identified as the long combinational logic and the negative iteration of CORDIC used two floating point adders during one clock cycle and hence the propagation delay exceeded the clock frequency.

The control logic implemented in the application layer using C language is depicted using a flow chart below for 32bit floating point numbers. The same logic shall be used for 64bit floating point numbers except for specific Hardware interface functions:
MY_FP_32IP_mWriteMemory() and
MY_FP_32IP_mReadMemory().



IV. RESULTS

A. Hardware Component Simulation

To check the result of hardware architecture, generated results were compared with MATLAB result. For extended hyperbolic cordic, relative error of exponential operation and natural logarithm operation were plotted.

Figure 7: Relative Error of e^x

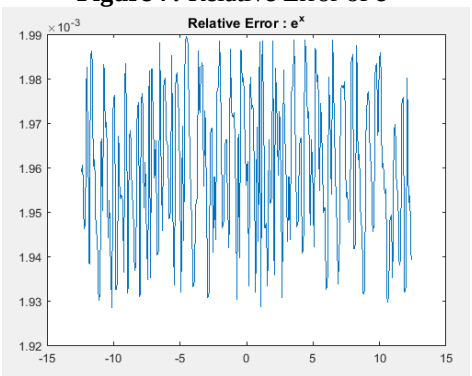
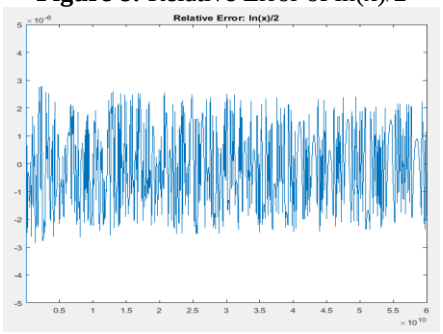


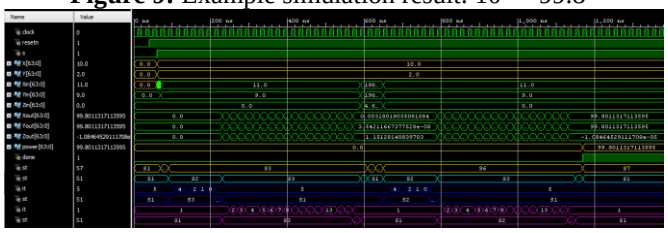
Figure 8: Relative Error of ln(x)/2



The error for both operations was very small. For exponential, error was within $2*10^{-3}$ range, while error for natural logarithm was within $2*10^{-6}$ range.

For power operation circuit, simulation was performed large amount of times to validate the results. As this circuit contains multiplier and shifters, small offset at the beginning becomes relatively visible offset. The example simulation result is shown in figure 9. The relative error for this case was about 2%.

Figure 9: Example simulation result: $10^2 = 99.8$



B. System Level Simulation

The resource usages of generated power IP (32 bits and 64 bits) are shown below.

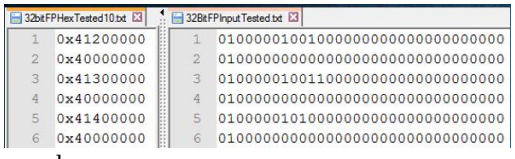
N	LUT	Flip Flops
32 bit	4378 (8%)	502
64 bit	9464 (18%)	895

To test the power IP using SD card IP, we have used different test files each containing series of numbers. The application is written in such a way that all numbers in each file are read, processed and results are stored into a separate file in the SD card.

Some of them are explained below:

Expected function: X power Y (i.e. X^Y)

32 Bit input: (X: 10, 11, 12 till 19 and Y: 2) in single precision

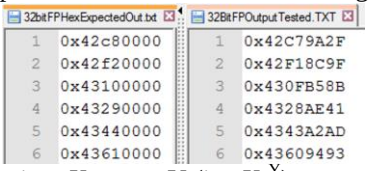


One example:

1st value 10 in single precision is 0x41200000

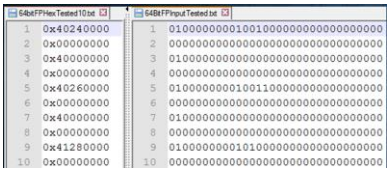
2nd value 2 in single precision is 0x40000000

Expected output = 10^2, 11^2, 12^2 etc. in single precision



Expected function: X power Y (i.e. X^Y)

64 Bit input: (X: 10, 11, 12 till 19 and Y: 2) in double precision



One example:

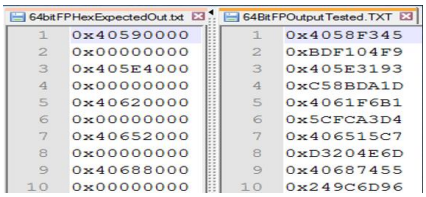
1st & 2nd value 10 in double precision is

0x 40240000 00000000

3rd & 4th value 2 in double precision is

0x 40000000 00000000

Expected output = 10^2, 11^2, 12^2 etc. in double precision



CONCLUSIONS

We have successfully completed the project and made operational power calculation circuit within the time frame. The project showed how resource intensive to implement arithmetic operation in floating point format. Also, it demonstrated importance of inserting register inside large combination circuit to improve clock frequency. The effectiveness of parameterized vhdl coding made it possible to try different floating point format with minimum modification.

The accuracy of power operation can be improved by modifying number of positive and negative iterations. Next step to this project will be to parameterize both M and N.

Overall, it was satisfying to work on this challenging project and get deep understanding of both software and hardware side of embedded system.

REFERENCES

[1] Mack, J., Bellestri, S., and Llamocca, D., "Floating Point CORDIC-based Architecture for Powering Computation", to appear in *Proceedings of the 10th International Conference on ReConFigurable Computing and FPGAs (ReConFig'2015)*, Mayan Riviera, Mexico, December 2015.

[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.