Dual Fixed Point Calculator

Furzana AbdulRahim, Jing Wu, Zhongda Gan
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
E-mails: fabdulrahim@oakland.edu, jingwu@oakland.edu, zhongdagan@oakland.edu

***Abstract*—during the calculation, the fixed point and floating point has their disadvantages, here the dual fixed point can meet the calculation precision and take less resource. In this project, the pipeline was used to make the calculation faster, and better for calculating a series of data.**

I. INTRODUCTION

In data analysis, there are lots of data around us every day, the calculator is needed to handle it more precise and fast, the hardware computation is a good choice, in hardware computation, there are fixed point and floating point formats, the former one is precise but limited range, the latter one is less precise but with wide range, so the dual fixed point can show its advantages now, it can satisfy the precise and the widen range at the same time. If a bunch of data was needed to be processed at the same time, the pipelined calculator can solve it easily. Which explained the reason why this project is demonstrated here.

In this calculator, three kinds of calculation are shown: division, multiplication, expanded hyperbolic. And also you can change the dual fixed point format as you need, here the format 32 14 5 and 32 18 5 is shown.

II. METHODOLOGY

A. Divison

The algorithm to compute the division for DFX is based on the concept of fixed-point number division learnt in class [1]. The inputs for this module are the dividend and divisor represented in [N p0 p1]. Also, a clock, reset and start signals. The outputs of this core are the DFX overflow signal, the done signal and the quotient represented in [N p0 p1]. The core first gets the positive numbers for the dividend and for the divisor. If we need to change the sign of the quotient we keep the sign signal. After both operands are positive, it could be the case that the dividend is num0 and the divisor is num1 or the other way around. The alignment is performed and precision bit are added. Both the numbers are of the same format when fed into the pipelined divider. Figure 1 shows the pipelined architecture of the divider. Result is available after few cycles. Since we add alignment bits and precision bits, the actual number of cycles after which the result is available is much larger than N cycles in DFX divider.

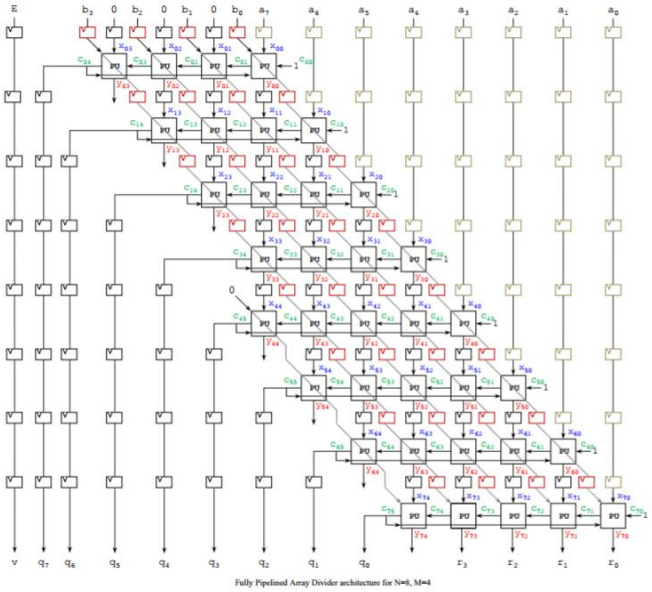

Figure 1 Fully Pipelined Array Divider Architecture

Output of the pipelined FX divider is fed into range detector to check the result is a num0 or num1. In some cases, overflow may result when converting the FX to DFX format as the bits used in FX divider is larger than in DFX [N p0 p1]. Sign change information is passed through the delay circuit to match to the time when the result is ready.

B. Multiplication

The DFX multiplier basically includes pre-scaler, pipelined unsigned multiplier and range detector. The two inputs will be converted into unsigned numbers at the beginning by pre-scaler, then got multiplied. The number coming out of the multiplier will be double than the length of the input, the range detector is then used to select whether the result is num0 or num1 and set the exponent bit accordingly. Finally, the conversion from unsigned number to signed number is necessary.

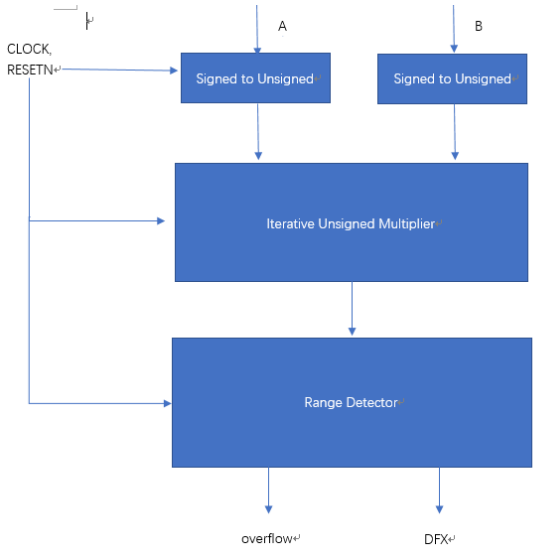

Figure 2 Arithmetic of the pipelined multiplier

The range detection module is responsible of determining if there is DFX overflow, slicing the product according to the possible outcomes due to the inputs number representation, determine if the num1 candidate can be represented as a the n0 candidate and changing the sign of the candidates if the sign signal is asserted.

C. *Expanded hyperbolic cordic*

The expanded hyperbolic cordic add the negative iteration to extend the range.

For $i \le 0, i = -M, -m+1, \dots, 0$. Here we choose M=4

$$\begin{cases} x_{i+1} = x_i - \delta y_i (1 - 2^{i-2}) \\ y_{i+1} = y_i - \delta x_i (1 - 2^{i-2}) \\ z_{i+1} = z_i + \delta tanh^{-1}(1 - 2^{i-2}) \end{cases}$$

For $i > 0, i = 0,1,2, \dots, N$

$$\begin{cases} x_{i+1} = x_i - \delta y_i 2^{-i} \\ y_{i+1} = y_i - \delta x_i 2^{-i} \\ z_{i+1} = z_i + \delta tanh^{-1} 2^{-i} \end{cases}$$

In rotation mode:

$$\delta = \begin{cases} +1, if\ z_i \le 0 \\ -1, if\ z_i > 0 \end{cases}$$

$$\begin{cases} x_{n+1} = A_n(x_{in}\cosh(z_{in}) + y_{in}\sinh(z_{in})) \\ y_{n+1} = A_n(x_{in}\sinh(z_{in}) + y_{in}\cosh(z_{in})) \\ z_{n+1} = 0 \end{cases}$$

Then convergence range is: $|z_{in}| \le 9.66581$

In vector mode:

$$\delta = \begin{cases} +1, if\ x_i y_i \ge 0 \\ -1, if\ x_i y_i < 0 \end{cases}$$

$$\begin{cases} x_{n+1} = A_n\sqrt{x_{in}^2 - y_{in}^2} \\ y_{n+1} = 0 \\ z_{n+1} = Z_{in} + tanh^{-1}(y_{in}/x_{in}) \end{cases}$$

Then convergence range is: $|tanh^{-1}(y_{in}/x_{in})| \le 9.66581$.

In figure 3 and figure 4 shows how to latch every internal signal.

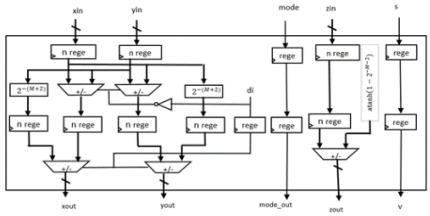

Figure 3 Negative iteration, -4 to 0

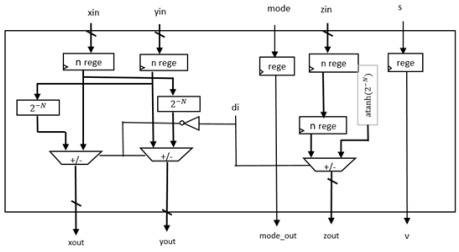

Figure 4 Positive iteration, 1 to 16

D. *Input and Output Interface*

Difference in the number of inputs for cordic and other two operations results in wasting few cycles in acquiring the input and loading the output into the FIFO. Four cycles are used in both cases. Figure 5 show the input and output interface structure and state machine used.

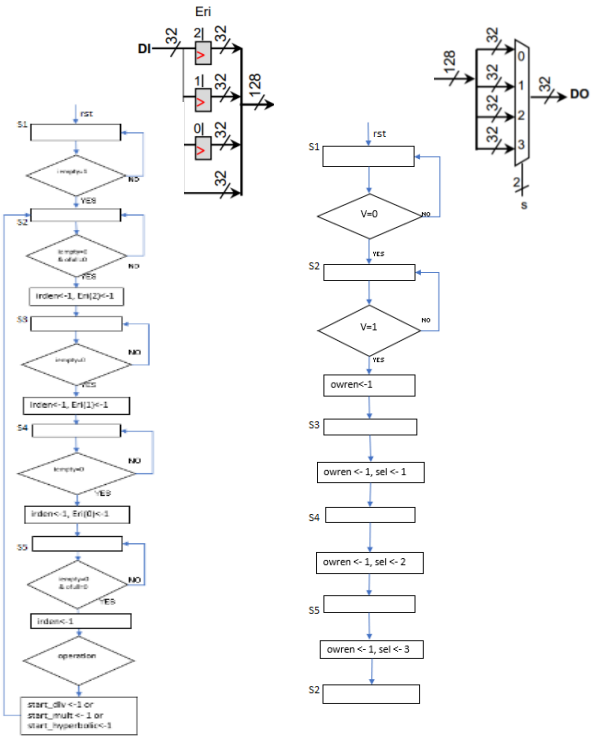

Figure 5 Input and Output Interface, State machine

III. Experimental Setup

Vivado simulator is used for functional simulation. Testing is done for each number combinations (n0-n0, n0-n1,n1-n0,n1-n1) from the numbers obtained from MATLAB for [32 14 5]format. Initially, individual blocks (divider, multiplier, cordic) are tested for basic functionality. Then random DFX numbers generated by MATLAB is used as input to the test bench in a text file. Results are collected in a text file and plotted to see the accuracy. After few calculations, we decided [32 18 5] had better accuracy than [32 14 5] format. Final results are plotted using [32 18 5] format. After this, blocks are combined and tested. Final verification is done with SDK initiating memory writes and reads from the testcase.

Except VHDL test bench and MATLAB Verification, we also set up a hardware implementation using the ZYBO board connected to the AXI bus. We programmed the FPGA with the bitstream of the project including the configuration of the Zynq. Afterwards we programmed the compiled software using the UART interface. Figure 6 shows the architecture of DFX calculator connected to the AXI full bus.

In expanded hyperbolic cordic, it need to push the y or z near to 0, in this process, these small number will make a huge difference for the operation for x and y, so the format 32 18 5 is chosen here. The precision is$3.8 * 10^{-6}$, which cause the convergence to be 9.65. Which is a little smaller than the theory one.

Using hyperbolic there are 4 common used function can be directly computed, sinh, cosh, exponent, atanh. In rotation mode, set the input like this,$xin = \frac{1}{A_n}$, $yin = \frac{1}{A_n}, zin = x$ then the function exponent is got, $xout = yout = e^x$, and the range of x is$|x| \leq 9.65$. Also in rotation mode, set the input as below: $xin = \frac{1}{A_n}$, $yin = 0\ zin = x$, the outputs xout is $\cosh(x)$, yout is $\sinh(x)$, the range of input x is $|x| \leq 9.65$. In vector mode, choosing the input: $xin = 1$, $yin = x, zin = 0$, the zout is $\operatorname{atanh}(x)$, the range of input x is $|x| < 1$.

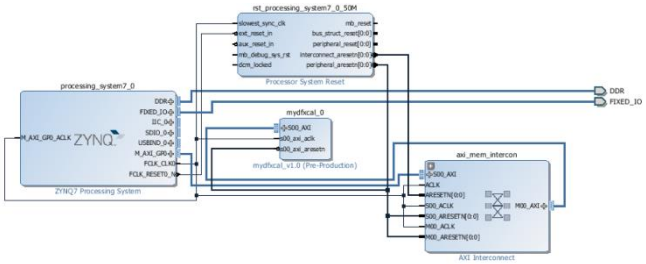

Figure 6 Cores with AXI Bus

IV. Results

Outputs (100 data sets for each case) from the DFX calculator is compared with the MATLAB function output. The result presented below which perfectly match the MATLAB output. Figure 7 shows the results of the divider with all possible combination of the operands n0-n0, n0-n1, n1-n0, n1-n1. In case of n0/n1, we see a little mismatch as the denominator is a large number and the precision of MATLAB is much higher than DFX calculator.

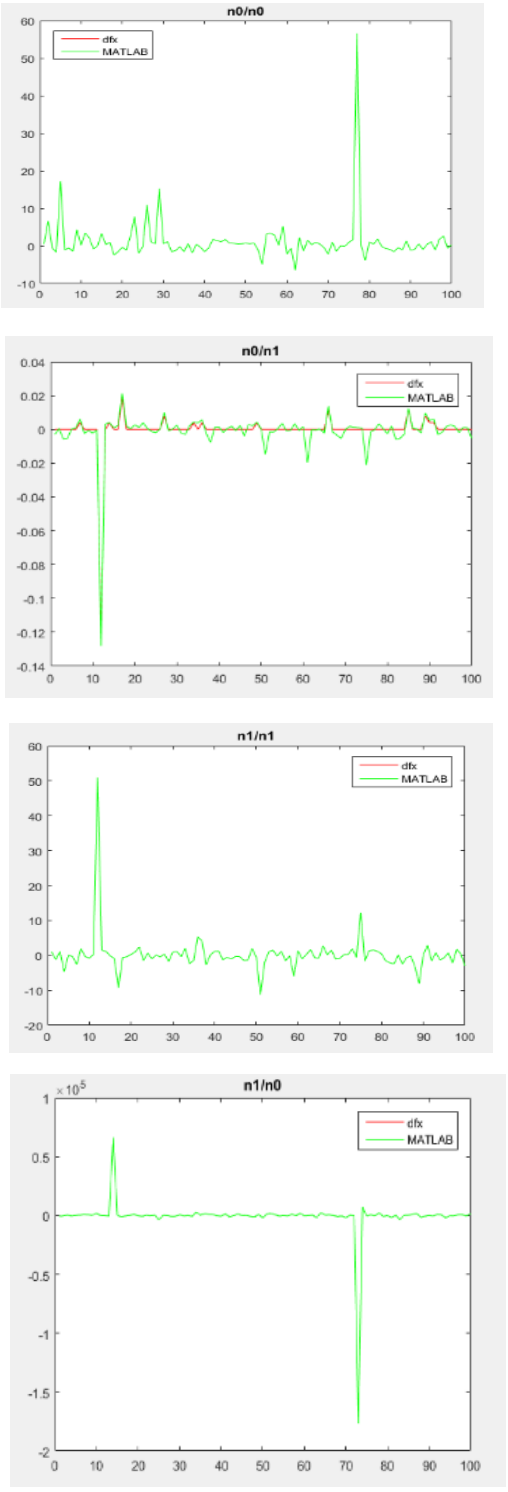

Figure 7 MATLAB output for divider with all cases

Figure 8 shows the DFX multiplication result for [32 14 5] between two n0 numbers. Figure 9 shows the relative difference between VHDL and MATLAB is very small.

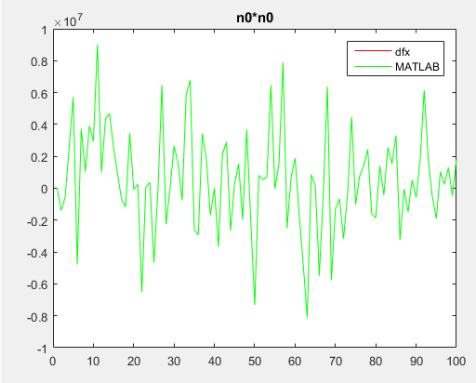

Figure 8 MATLAB output between two n0 numbers

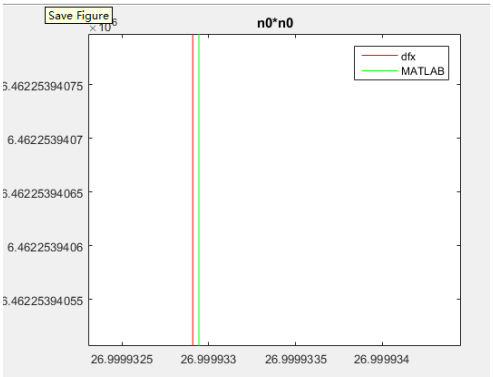

Figure 9 Comparison between MATLAB and VHDL

As explained before, 100 sets of data are being used to test each case. Figure 10 shows the function of e^x. It shows the two lines almost the same, and the difference is below 0.2, in

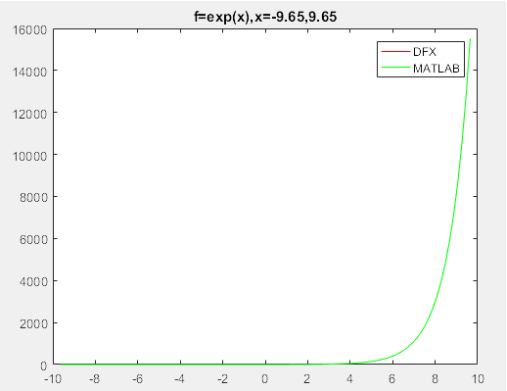

Figure 10 Exponential function

most of cases, the relative difference is below 2%, but at some points it is 12%, in that points, during the convergence, the demanded precision is higher than is $3.8 * 10^{-6}$, the difference between the cordic results and the functional results is smaller than at this area is smaller than $3.8 * 10^{-6}$, which is affordable.

Figure 11 and 12 shows the function of $\cosh(x)$ and $\sinh(x)$ respectly. The relative differences are below 0.014%, which looks very good.

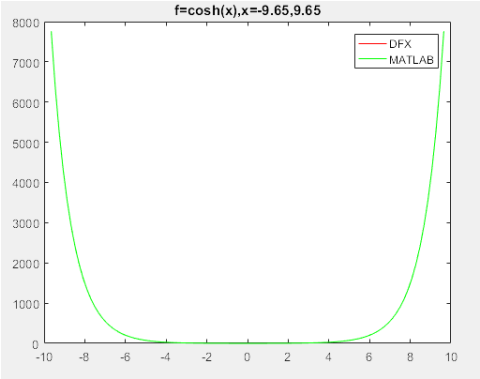

Figure 11 Function of cosh(x)

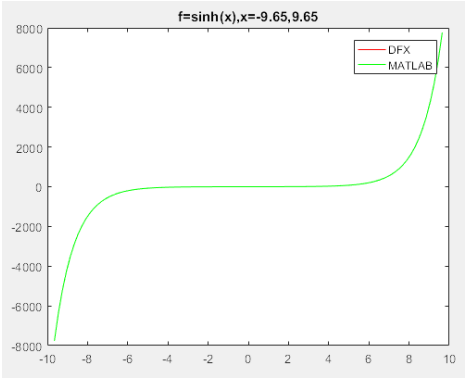

Figure 12 Function of sinh(x)

Figure 13 shows the function of $\operatorname{atanh}(x)$, it is difficult to distinguish the two lines, they are almost as one. in most cases, the relative difference is below 2%, and the difference between the cordic results and the functional results is smaller than $6 * 10^{-3}$.

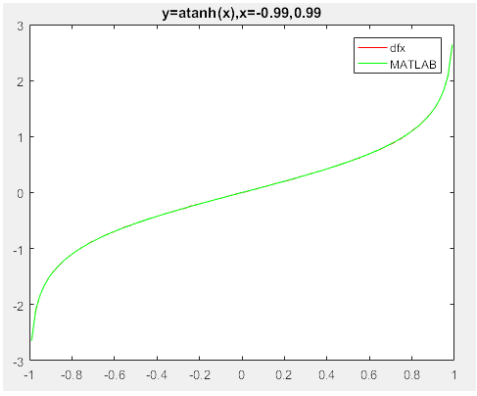

Figure 13 Function of atanh(x)

CONCLUSIONS

In this project, we used large number of inputs to verify our DFX calculator in [32 18 5] and [32 14 5] two different configurations. The result shows that the design is perfectly done based on pipelined multiplication, division and expanded hyperbolic cordic. The program logic clock in our design is 60MHz which means the calculator can compute 60M numbers in 1 second theoretically. Based on this character, our pipelined calculator can process a mess amount of input numbers and do the operation in a very short time. For the two configurations we used in the calculator, [32 18 5] provides higher precision, but [32 14 5] can contain bigger range of input and output numbers.

Future work will focus on adding more operations and more effective input and output interfaces. DMA can be used to write and read the data from the IP. Higher the precision, smaller the range will be. We would like to find out the equilibrium point between the precision and the range of numbers.

REFERENCES

[1] Daniel Llamocca, “Notes – Unit 1” in ECE-5900: Specail Topics – Reconfigurable Computing. ECE Department, Oakland University.

[2] Chun Te Ewe, "Dual fixed-point: an efficient alternative to floatingpoint computation for DSP applications," in Field Programmable Logic and Applications, 2005. International Conference on , vol., no., pp.715- 716, 24-26 Aug. 2005

[3] http://www.secs.oakland.edu/~llamocca/arithcores.html