Dual Fixed Point Calculator

Gaurav Agalave, Andres Jacoby
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: gsagalave@oakland.edu, ajacobykrateil@oakland.edu

***Abstract*—This work presents a hardware design, verification and an implementation in a FPGA of a Dual Fixed-Point (DFX) calculator. The basic arithmetic operations: addition, subtraction, multiplication and division were perform by three modules that were generically designed to work with DFX numerical representation to archive a higher dynamic range than fixed point with much less resources requirement than floating point arithmetic. In this work, we show how different format of DFX can lead to higher precision and the tradeoff with resources and timing. We strongly believed that DFX has enormous potential to be used in dynamic partial reconfiguration (DPR) for runtime power consumption, precision, accuracy and resources optimization.**

I. INTRODUCTION

The two most common ways to represent real numbers in computer arithmetic are floating point and fixed point. Floating point offers a better dynamic range but requires more hardware resources to implement the basic: addition, subtraction, multiplication and division. On the other hand, fixed point representation requires significant less hardware resources due to the fact that numbers are treated as integer in the computing of the basic operations. However, fixed point's disadvantage is that is it has a smaller dynamic range and it's not flexible in representing big numbers and small numbers once a certain format is picked. Dual fixed point (DFX) representation is a novel idea which tries to overcome the limited dynamic range of fixed point while at the same time not requiring as many hardware resources as a floating point [1]. DFX representation defines two different fixed point numbers within each DFX number [2]. The most significand bit (MSB) is called exponent which is cero for num0's numbers and one for num1's numbers. Both, num0 and num1, have p0 and p1 bits respectively, for the fractional part of the number. Conventionally num0 represents small numbers with a higher precision for the fractional part. Numbers that are big enough that cannot be represented with as a num0 are represented with less bits in their fractional. In this project we designed, verified and implemented the four basic operations in three hardware modules: an adder/subtractor, multiplier and divider. Moreover, the modules designed were generic which allows the implementation of them for different numeric representation therefore it is possible to optimize for hardware resources and precision as desired. This document is layout in the following structure: Section II presents the methodology followed in the project and explains in detail how the three different modules work. Section III explains how the modules were tested and verified. Section IV shows the results obtained from the experimental setup as well as other metrics that were used. Finally, conclusion are drawn based on the aforementioned results.

II. METHODOLOGY

Each DFX number is represented by a total of N bits, the first being the exponent which indicates if the number is represented as num0 or a num1. The N-1 bits following the exponent conform the number in complement's 2. In the case of num0 the number, N-1 bits, has p0 bits for representing the fractional part. On the other hand, num1's numbers have p1 for representing the fractional part. The methodology approach that was used for the three modules development was designed them as generic cores, were N, p0 and p1 are the parameters used to customized each of the modules. In the case of the divider the number of precision bits, x, it's also a parameter. This feature allows different dynamic range, numeric range and resources usage according to every application that could be developed in the future. Moreover, enabling dynamic partial reconfiguration (DPR) for power consumption, precision and resources usage optimization as demonstrated in [3].

.

A. Generic DFX Adder/Subtractor

The architecture for the generic DFX adder/subtractor was based on the class notes [4]. Figure 1, depicts the internal architecture. It is based on three main blocks, the pre-scaler which aligns the two numbers that can be the four combinations of num0 and num1. The pre-scaler module also keep the p0-p1 bits in case the number came out to be a numo0. Thus not losing precision when it's possible. The adder/subtractor module is basically an integer adder which also was a input signal (addsub) that determines if the operation that is going to be computed is an addition, in the case of addsub is cero or subtraction in case it is one. The adder also calculates the 2's complement when subtracting in that case the operand B is negated. This is important when using inserting the input values because it the order will affect the result. The adder also calculates the xor of the two most significant bits for the $overflow_{n-1}$ signal. The third module is called the post-scaler and integrates the range detector circuit, a control circuit, shifters and multiplexers. This module, compares p0-p1+1 bits from the

MSB-1 of the addition/subtraction result. If all those bits are either ceros or ones then exponent signal, E, gets the right exponent. Nevertheless, the result could be overflow, the control circuit deals with those cases implementing a simple true table. A mux implement the three possible situation if the number is no a DFX overflow. If the number is the same as N bits from the adder/subtractor or if it has to be shifter from a num0 representation to a num1 representation or the other way around in which case precision bits, if there are from the pre-scaler, are added in case it is addition.

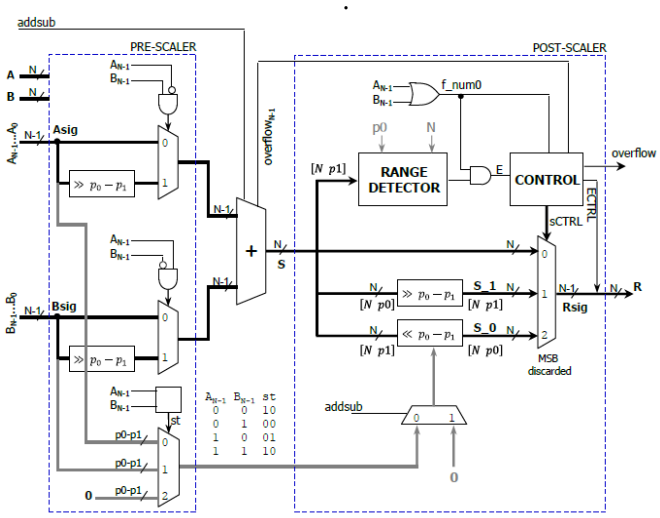

Figure 1. Architecture of DFX adder/subtractor

As it can be seen from the archicture the circuit is a pure combinational circuit. However, in an implementation we should add registers at the input and at the output for all the signals.

B. *Generic DFX Multiplier*

The design for the generic DFX multiplier was based on the unsigned integer multiplication cover in class [2]. Figure 2, shows the architecture of the DFX multiplier. This module has as inputs operand A and operand B with N bits, clock, reset and a start signal to start the multiplication. The outputs are the DFX product with N bits, a DFX overflow signal and the done signal. Due to the fact the multiplier can only deal with positive numbers, the two operands are first converted into positive 2's complement representation in case they are negative. At the same time, if there is going to be a sign change to the product because only one operand number was negative then the sign signal keeps asserted. The iterative unsigned multiplier replicates the traditional way of multiplying. By adding one of the operands every time the LSB of the other operand is one and then shifting this last operand until it becomes zero we can compute the integer unsigned multiplication. As explain by the algorithm we need at least the bit number of the operand being shifted plus one. In the case of the DFX multiplier, the two operands and the output are represented in the same [N p0 p1] format thus we need N+1 clock cycles in the worst case scenario. 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 num0 candidate and changing the sign of the candidates if the sign signal is asserted.

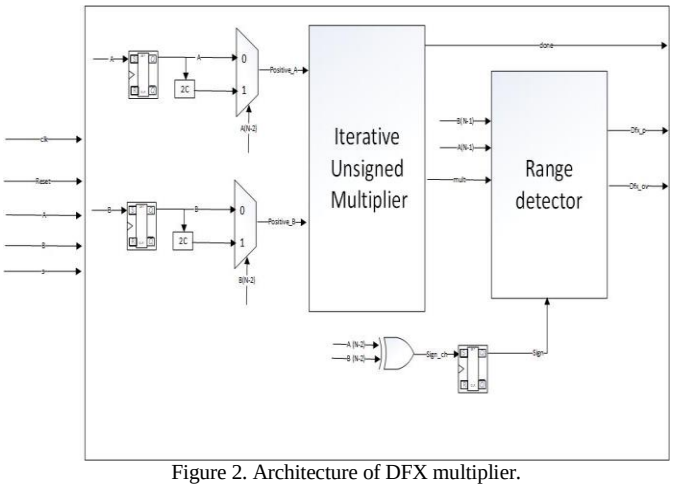

Figure 2. Architecture of DFX multiplier.

C. *Generic DFX Divider*

The design architecture for the generic DFX divider is shown in figure 3. The algorithm to compute the division for DFX as based on the concept of fixed-point number division learnt in class [2]. The inputs for this module are the dividend and divisor represented in [N p0 p1]. Additionally there is 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 get the positive numbers for the dividend and for the divisor. Similarly as the case of the DFX multiplier, 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 to the dividend. In order to keep the signals the same size we do the same to the divisor but we add the bit in the most significand part of the number thus no affecting the number. It's important to notices that alignments and precision bit are always ceros because we are working with positive numbers. Inside the iterative divisor a finite state machine will perform the subtractions and shifts based on comparisons similarly than manual division in which we increase the digits we take in case the number is smaller than the divisor or we perform the subtraction of a part of the dividend and the divisor. After that, a range detection circuit will slice the output quotient in difference in a num0 and a num1 candidate. The point of reference is the number of precision bits that we inserted previously. Then the range detection circuit compares the p0-p1+1 bits of the num1 and if they are all the same the num0 is taken. Concurrently a overflow detection circuit perform the DFX overflow check.

Depending on the precision bits, p1 and p0 the circuit checks of bits that are asserted but should not be.

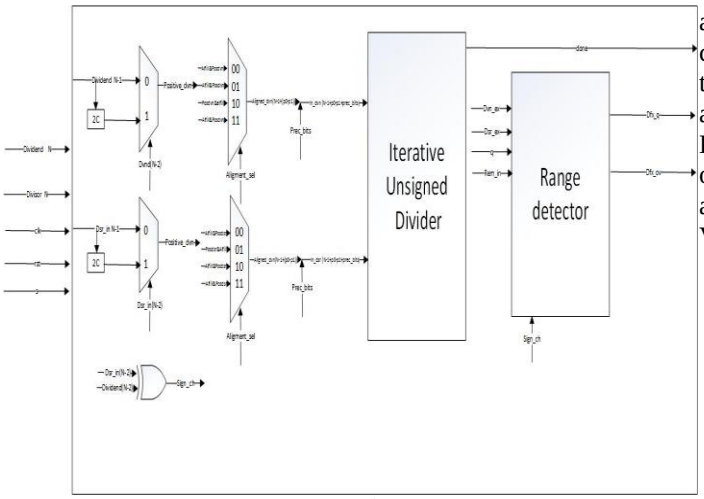

Figure 3. Architecture of DFX divider

III. EXPERIMENTAL SETUP

In the first stage of the development of the modules the homework solutions were taken as inputs to verify the simulations using Vivados' simulator. However, due to the fact we wanted to test a large amount of numbers to verify the modules properly and measure the relative error due to precision lost by performing the operation in a given DFX format in contrast to a general purpose computer with double precision floating point representation, we use Matlab to generate random operands for the VHDL test bench and improve the robustness of the testing. Figure 4 shows the algorithm to generate random floating point numbers ranging between maximum number that can be represented by num1 number to minimum number that can be represented by num1.

```
x = 2n * rand([1,100]) - n; % x random number generator
y = 2n * rand([1,100]) - n; % y random number generator
Where n = number of bits. (100 different random numbers are generated)
```

Once the numbers are generated, they are sorted out as num0 or num1 using range formula for num1 and num0. At the same time they are converted into their num1 or num0 representation using quantizers of fixed data type and n_P1 and n_P0 configuration.

```
% num0 boundary
b1 = -2^((n-1)-1) / 2^P0;
b2 = (2^((n-1)-1) - 1) / 2^ P0;
% num1 boundary
a1 = -2^(n- P1-2);
a2 = (2^(n- P1-2)) - 2^(- P0);
r = quantizer ('fixed',[n-1, P0]); % num0 fx form
q = quantizer ('fixed',[n-1 P1]); % num1 fx form
```

This step is important to lose some precision in floating point numbers and format the numbers in binary format to feed VHDL test bench. The numbers are formatted in binary fixed point, 1 bit less, making room for exponent bit. After that the numbers are assigned 1 or 0 as MSB for exponent to represent Dual fixed point number. The binary fixed point numbers are then again converted into floating point numbers for mathematical operations in Matlab. The addition, subtraction, multiplication and division are then carried out on the fixed point formatted floating points and the result is again converted into fixed pointy binary number along with the exponent bit. The operands and answers (in DFX format) for all operations are saved in a text file. The overflow occurred in all operations for all operations are also registered. The operands are used for test bench in VHDL.

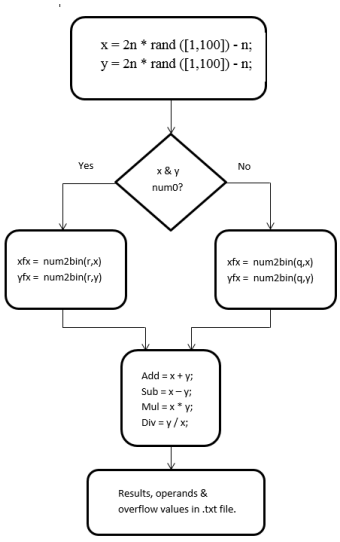

Figure 4. Flow chart for operand generation in Matlab.

A test bench for each different module and operation was wrote in VHDL using Xilinx Vivados' software. The test bench open and read the text file generated by Matlab for each operand and every output was saved in a text file. In each operation (addition, subtraction, multiplication, division) two files were saved, an overflow file with the DFX overflow signal and the actual result in DFX format. After the Vivados' simulator created the output files we used Matlab again. Figure 5 shows the script to compare and plot the VHDL and the Matlab results. First the outputs (in DFX format) from Matlab and VHDL are read from text files. The overflow flags generated during operations in both Matlab and VHDL which were also stored in text files are extracted. The binary DFX formatted numbers are converted into fixed point numbers by first chopping off exponent bits and converting binary FX numbers to numerical floating points using quantizers. The exponent bit is used for determining the number is num0 or num1 configuration. According to that, numbers are converted. Using Overflow information, the numbers which have overflow registered are forced to zero as there is no relation between those values in Matlab output and VHDL output. The generated floating points with zeroed out overflow values are later on plotted for picturing the performance of both models and test the output of VHDL.

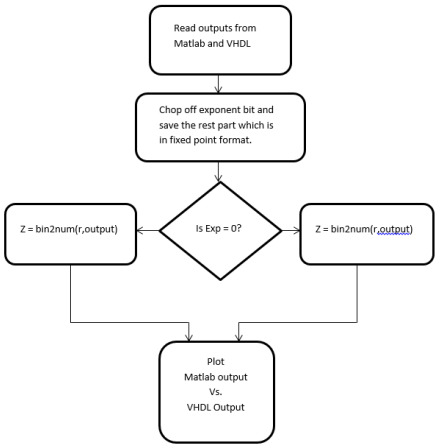

Figure 5. Flow chart of the comparison program in Matlab.

We expected that the results from VHDL and Matlab to be the same or very similar. In addition, the number of overflows was expected to be the same and in for the same operations in both the VHDL and Matlab results.The two set of parameters that we used to test the three cores were: [8 5 3] and [16 10 7].

A part from the Matlab/VHDL test bench we also set up a hardware implementation using the Digilent's ZYBO board which integrated an ARM A9 microcontroller connected to the AXI bus. We created a calculator which was composed by the two sets of adder/subtractor, multiplier and divisor. The first was in the [8 5 3] configuration and the second was in [16 10 7]. The input and outputs were map to 15 32-bits registers which later were connected to the AXI Lite as a slave core. After that, we wrote a small code to test the first set of adder/subtractor, multiplier and divisor. 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 high level architecture of DFX calculator using *my_dfx_alu* core connected to the AXI Lite bus.

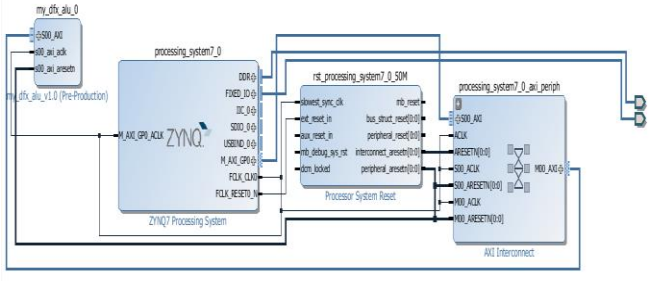

Figure 6. SoC with the DFX ALU connected to the AXI Lite bus.

IV. RESULTS

As stated before, when comparing the output files for the addition, subtraction, multiplication and division of the VHDL output and the Matlab output the first analysis was done in the overflow vectors. In all the cases presented in this document that overflow from VHDLresult and Matlab match. The core design doesn't take in consideration the underflow this cause some mismatch but was adjusted to only show numbers bigger in absolute value than the biggest num1. Once the number of overflow match and in the same points we plotted the values for the 100 pair of operands minus the overflow operands, due to the fact that we were interested just in the numbers that could be represented innum0 or num1 for each set of parameters. Figure 7, shows the DFX analysis. In red color the result from Matlab is shown and the black dots are the outputs from the VHDL using the format [8 5 3]. As can be seen they are the same or very similar there is some difference in num1 for sample 9 and 20.

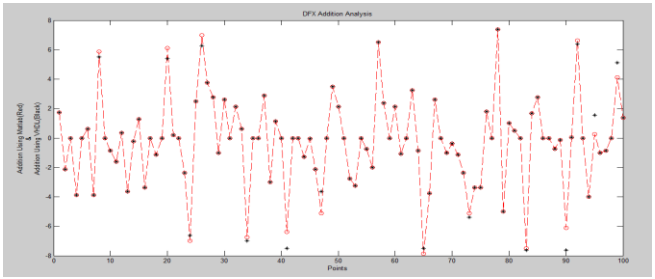

Figure 7. DFX addition analysis for [8 5 3].

Figure 8 shows the DFX subtraction analysis for the [8 5 3]. Similarly as the addition there are some points that are not in the same spot this is because the in num1 we lose precision while in Matlab the result is more accurate.

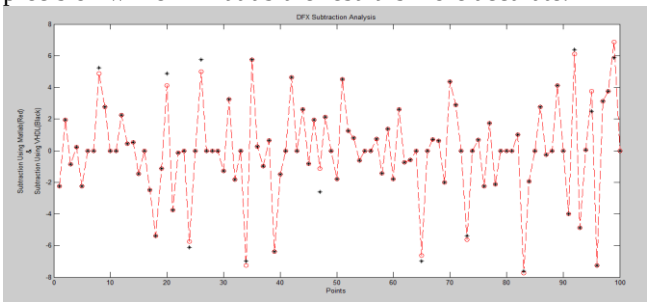

Figure 8. DFX subtraction analysis for [8 5 3].

Figure 9 shows DFX multiplication analysis for [8 5 3]. The results match and in some cases there is some difference mostly in the num1 results. We have less results due to the fact that is very easy to generate overflow.

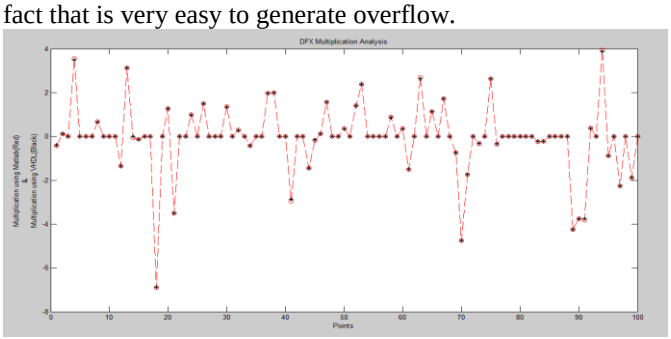

Figure 9. DFX multiplication analysis for [8 5 3]

Figure 10 shows DFX division analysis for [8 5 3]. From the plot we can see that all the outputs are very close to the right value. However, there is error in the results compare with Matlab.

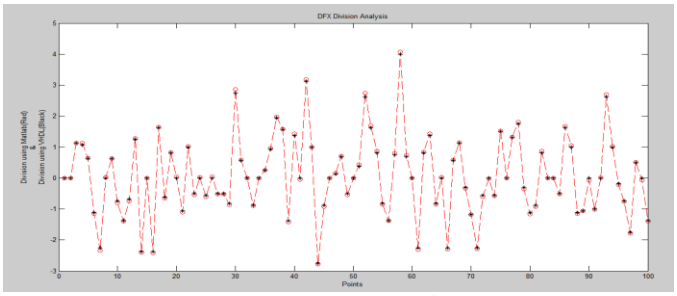

Figure 10. DFX division analysis for [8 5 3]

Figure 11 shows DFX addition analysis for [16 10 7]. From the plot we can see that most of the values match very close to the correct, Matlab values. Comparing with figure 6, we reduce the error in num1 due to the usage of more bits.

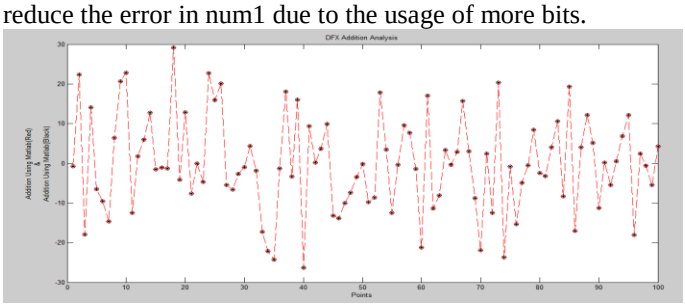

Figure 11. DFX addition analysis for [16 10 7]

Figure 12 shows DFX subtraction analysis for [16 10 7]. The black dots show a value very similar than the Matlab output. Comparing this result with the result from figure 8 we can appreciate that there a better improvemnet in accuracy for large values while at the same time num0 are accurately plotted.

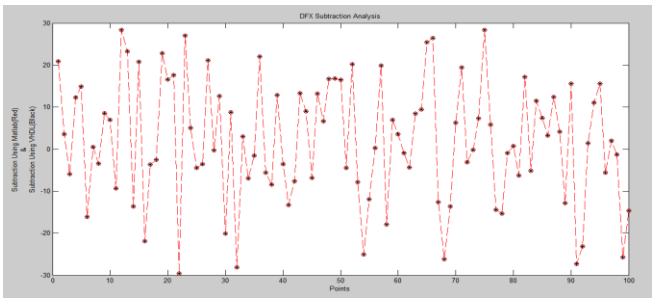

Figure 12. DFX subtraction analysis for [16 10 7]

Figure 13 shows the DFX multiplication analysis for [16 10 7]. Most of the values match the Matlab output with high precision. We can see that out num1 range is much bigger too.

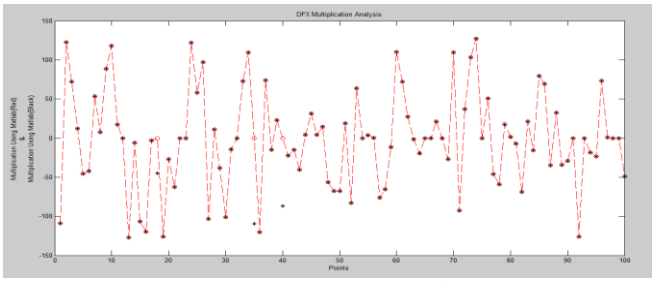

Figure 13. DFX multiplication analysis for [16 10 7].

Figure 14 shows the DFX division analysis for [16 10 7]. We can see that all the values match correctly the Matlab ouput. Most of the results are num0 but there are some num1 as well.

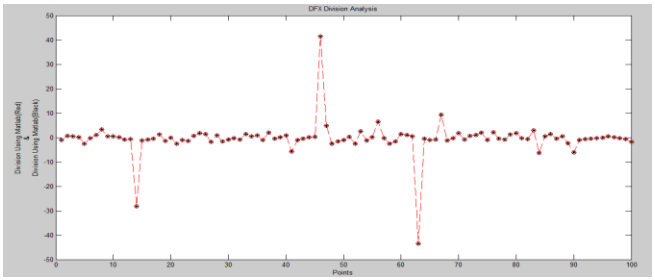

Figure 14 DFX division analysis for [16 10 7].

From the results of [8 5 3] previously shown we calculate the relative error. For addition the error was -0.0529, for subtraction the error was 1.303. In the case of multiplication the error was 0.0198 and for division the error was $-2.91*10^{-4}$.

Table 1 shows the hardware resources that would be necessary to implement each of the modules in each of the four configurations. As it was expected the DFX multiplier and the DFX divider use much more resources than the adder/subtractor and with an increase in the number of bits more hardware resources are needed.

Table 1. Resources utilization for different configuration (in LUT)

	[8 5 3]	[16 10 7]	[24 15 10]	[32 20 15]
DFX Adder/Sub	31	64	76	101
DFX Multiplier	73	187	302	353
DFX Divider	84	190	346	443

We also calculate the timing to get the correct answer for each of the modules in terms on clock cycles. As previously mention the adder/subtractor is a combinational circuit but it's a good practice to register the signal at the input and at the output for this reason the number of clock cycles required to get an answer from the input is 2 clock cycles. This result is not affected if we synthetized the circuit with a different format for the DFX numbers. In the case of the [8 5 3] multiplier the number of clocks required was 8 and for the divisor in the same format the number of clock is 16. Figure 15 shows the implementation result of the DFX

calculator in the ZYBO board. As it can be seen the output that we got in SDK terminal as the output expected of each of the operands pairs in each of the four operations.

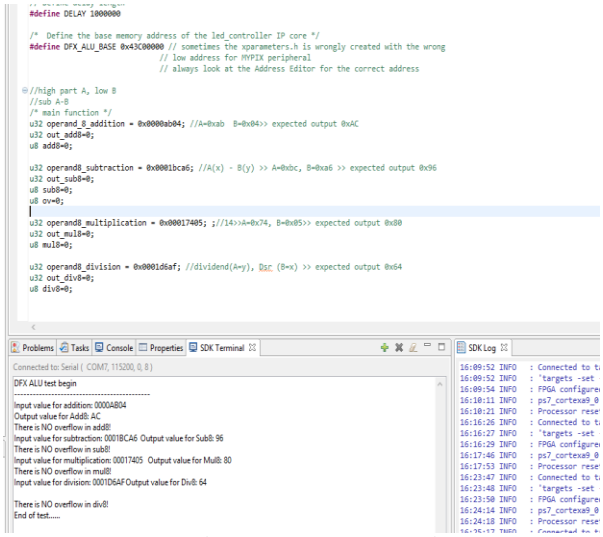

Figure 15. Result from the implementation of the DFX calculator

CONCLUSIONS

In this work we successfully designed, verified and implemented three hardware cores to perfume the four basic arithmetic operations: addition, subtraction, multiplication and division in DFX arithmetic. We used more than 200 pair of operands to properly verified our designed in two different configuration [8 5 3] and [16 10 7].

We showed how DFX can archived high precision and a bigger dynamic range with a small number of bits. We also demonstrated that our implementation of those three DFX cores used a small number of hardware resources. Moreover, in this work we show how a generic architecture can be used to represent different ranges of numbers and also how taking a given format for the DFX can yield to a highly accurate calculation but at the same time taking a another format can allow us to represent bigger numbers with less precision. In addition, we show how an increase in bit for different format utilized more resources in each of the three cores. In the case of the multiplier and division the underflow as not taken in consideration. I would be a good idea to assert the overflow signal or use a separate underflow signal. In the case of the divider it would be important to analyze the influence of the precision bits. As what's the point in which there is no more precision gain and the tradeoff it can leads in terms of timing and hardware resources. Further testing is also highly advised.

REFERENCES

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

[2] Daniel Llamocca, "Notes – Unit 1" in ECE-494/595: Specail Topics – Reconfigurable Computing. ECE Department, Oakland University.

[3] G. Alonzo Vera, Marios Pattichis, and James Lyke, "A Dynamic Dual Fixed-Pointed Arithmetic Architecture for FPGAs", Volume 2011, Article ID 518602, 19 pag on International Journal of Reconfigurable Computing

[4] Daniel Llamocca, "Notes – Unit 2" in ECE-494/595: Specail Topics – Reconfigurable Computing. ECE Department, Oakland University.