Dual Fixed-Point Calculator

Gabriel Ramirez, Austin Nolen
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
anolen@oakland.edu, gabrielramirez@oakland.edu

***Abstract* - This project demonstrates the design and test of a Dual-Fixed Point calculator. The calculator functions implemented include addition, subtraction, multiplication, and division. The calculator architecture design was built and tested on an FPGA. An AXI interface was used to interconnect software and hardware to input operands and receive results.**

I. INTRODUCTION

In computer arithmetic, there are several ways to represent numbers. Two of the most common are fixed-point and floating point. Fixed-point numbers are represented as an n-bit number with p fractional bits, i.e. [n p]. One advantage of using fixed-point is the low resource usage on a system. They are treated as integers in any arithmetic operation once the fractional bits are aligned. The disadvantage is the dynamic range. A number can be represented in any given form of fixed-point but once you have chosen an *n* and *p* you are limited in your dynamic range.

Floating point, on the other hand, has a much higher dynamic range than fixed-point. However, this comes with a compromise on resource usage. In applications where system resources are very lean, floating point would not be a good choice.

Dual fixed-point numbers are an attempt at solving the disadvantages of both fixed-point and floating point. This method of number representation involves an exponent E bit, followed by a signed n-1 bit significand. The exponent can be either 0 to 1 indicating two possible scaling. The formal representation is written as [n p0 p1]. An exponent with value 0 indicates p0 and 1 indicates p1. This allows the one to represent a very large integer with less fractional bits, or very small numbers with many fractional bits. This method effectively solves the problem of fixed-point numbers while maintaining a low resource usage.

This project will demonstrate the build and test of each component of the dual fixed-point calculator. We will show the design process that we followed utilizing topics learned in class along with testing and validation. The details of the AXI peripheral and software implementation will also be discussed. Dual fixed-point numbers hold a lot of potential to be useful in many different electrical engineering applications in today's world. This project hopes to show just one of those purposes.

II. METHODOLOGY

A. Adder/Subtractor

The Adder/subtractor was designed based on the architecture presented in class on Dual fixed-point addition and subtraction [2]. To add or subtract two dual fixed point numbers, one must first convert the numbers to fixed-point form. This is done in the pre-scaler of the adder/subtractor architecture. The pre-scaler also aligns the fractional points of the numbers so that they can be added or subtracted properly. This may cause some bits to be truncated. However, we can save these truncated bits and add them on in the post-scaler.

Once the operands are aligned it is time to perform addition or subtraction. We signed extend the numbers to n bits so the result will have n bits with p1 or p0 fractional bits. This addition will not result in overflow, however we must take the result as an n-1 bit number which could result in overflow.

The result of the addition/subtraction then feeds into the post-scaler component. In this part we determine which form num1 or num0 we will represent the fixed-point number. The ranged detector takes the fixed-point output in the same form as the input and determines what scaling it should be represented as. IN the case where left shifting is needed, we can use the truncated bits that were saved in the pre-scaler and shift them into the result. This will help maintain the accuracy of the adder.

Another important piece of the adder/subtractor architecture is the control block. This block controls a 3 to 1 multiplexor that chooses which output will be the final result. This block takes the overflow bit, the output of the range detector, and the scaling of the output number to determine the final result. Here, there are three possible options, keep the output in the same scaling after addition/subtraction, convert from p0 to p1, or convert from p1 to p0.

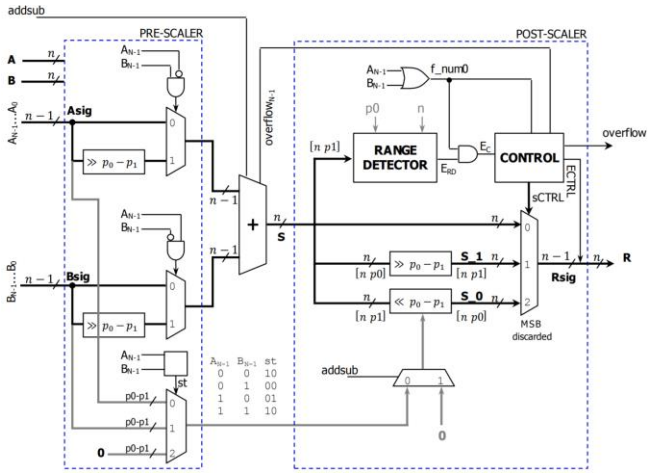

Figure 1: Adder/Subtractor Architecture

B. *Multiplier*

The multiplier architecture design was modified from Professor Llamocca's Unit 1 lesson notes [1]. The process is as follows. Remove the exponent bit from the 16 bits of data forming each operand and convert the operands to their absolute values. Using the bit values of Operand2 create vectors with Operand1 bit shifted to the left corresponding to the bit position of Operand2. Add all of the vectors formed from the previous operation. Checking the sign of the original operands, decide if the resulting value should remain positive or needs to be converted to its negative representation. This result has no fractional bits. Using the exponent values from the original data, determine how many fractional bits the resulting product contains. After determining the fractional bits check to see if the integer bits can be correctly represented by 7 or 11 bits. Specifically this was done by looking at the seventh integer bit and seeing if all leading bits were the same value. If this is true then it can be determined that the integer can be represented by 7 bits. Extend the number using as many fractional bits as necessary to have 15 bits total in the end result. Finally add the exponent bit to the result as the MSB.

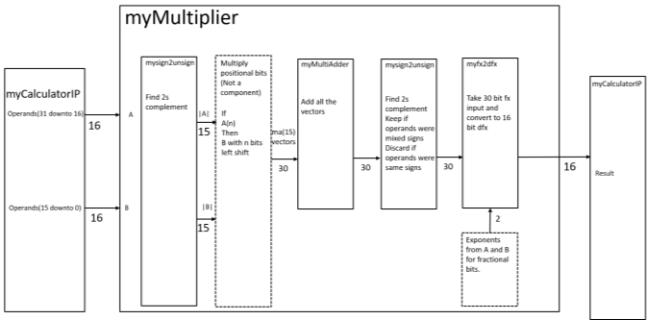

Figure 2: Multiplier Block Diagram

C. *Divider*

The divider architecture design was based around the method of unsigned iterative division [1]. To perform division on a dual fixed-point number, one must first convert the operands into fixed-point form. This is done by simply removing the exponent bit from the nth bit, leaving an n-1 bit signed significand. If the significand is a negative number then a 2's complement operation must be performed in order to get two unsigned numbers. Then, we must align the fractional bits so that we can perform integer division. To do this we look at the nth-bit exponent of both operands. If the exponents are the same they are already aligned. If one of the operands is in the *p1* scaling we must convert the other operand into the same scaling. This operation does involve truncating bits from the number being converted so we lose some accuracy. This conversion from dual fixed-point, signed to unsigned, and fractional bit alignment is completed in the pre-scaler section of the block diagram shown below.

The final output of the pre-scaler should be two 15-bit unsigned fixed point aligned numbers, dividend and divisor. Now, we have the option to add precision bits to the dividend to improve the precision on the output of the divider. The dividend in this project was tested using both 0 bits of precision to validate basic function, as well as 2 bits of precision. The two bits are added onto the end of the 15-bit dividend making it 17-bits. Adding precision does add cycles to the processing time, so in fast applications it should only be done if necessary.

The unsigned integer divider was based on a design presented in class notes [2]. Shown in figure 3 below is the architecture.

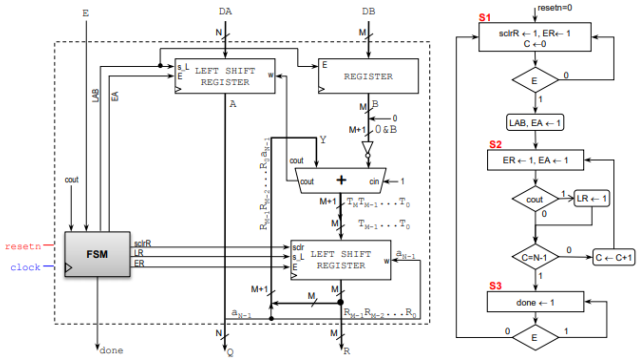

Figure 3: Iterative Unsigned Divider

The design includes an M-bit register, with an M-bit and N-bit left shift registers, a subtractor, and a finite-state machine. With operands N-bit A and M-bit B the divider will shift in the next bit of A or shift in A and subtract B at every clock cycle. The finite-state machine controls the inputs of all the registers to work in sequence with each other. After N cycles the output Q holds the result.

The results Q contains the integer result and precision bits if they were added. To get the final result in fixed-point we slice the output based on where the fractional point is and align it back to the proper scaling. In this

design, the output of the divider is two results for the same number, one in p0 form and one in p1.

The output of the unsigned integer divider feeds into a post-scaler circuit. This circuit first takes the unsigned output and inputs it into a range detector block. This block determines which scaling the output should fit in based on the size of the number. The post-scaler also performs a 2's complement operation based on the signs of the input operands.

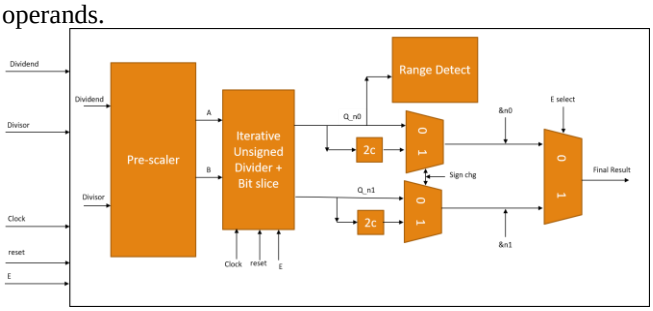

Figure 4: Divider Block Diagram

D. AXI Interface

An AXI-Lite Interface was designed around the calculator IP. We chose the Lite interface because we did not need the capability to FIFO data into the calculator. Also, a calculator is traditionally used in a 1 input per 1 output fashion which the Lite interface allowed us to do. The interface was designed using 3 slave registers, two on the input side and one on the output. Each slave register of the AXI can hold a maximum of 32 bits of data. Since our input operands are both 16 bits each, we were able to use 1 slave register on the input to hold both operands.

The second slave register was used to select between which of the calculator functions was active. We did this using a 4 to 1 multiplexor on the output with two selection bits to decide between calculator functions. We also used the AXI reset and clock signals primarily for the iterative unsigned divider which contains a finite-state machine that relies on these signals to function properly. Shown in the diagram below is the AXI peripheral architecture.

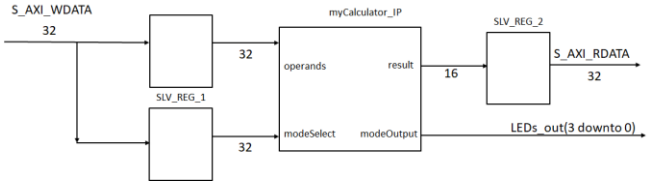

Figure 5: AXI Lite Block Diagram

E. Software

The software component of the project was designed using Vivado SDK in C programming language. It was designed to simply write to the slave registers of the AXI peripheral and then read the results of the calculator of the output slave register. This effectively demonstrated the interface between hardware and software using the Zybo board.

The figure below shows the full block diagram of the Zynq processing system as it is connected to the calculator utilizing the AXI-Lite interface. This design step is crucial to allowing us to use the software program to interface with the calculator. During this design step we setup a base address for writing and reading data to the Zybo board memory.

This base address is used in the c program to write the operands and the calculator function selection bits.

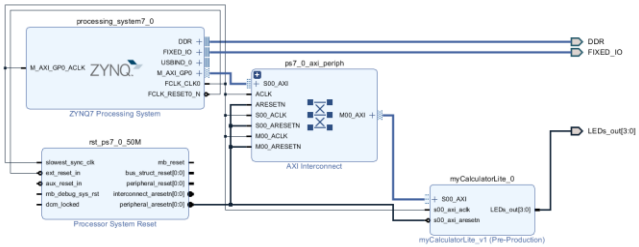

Figure 6: Design Block Diagram

III. Experimental Setup

During the design process, each component of the calculator was individually tested before it was integrated into the final product. This was done primarily using Vivado. A test bench was written to simulate input signals and read outputs on each component. The purpose of this was to validate that each component worked properly as a stand-alone device and to understand how each one operated.

We also tested the AXI-Lite interface including the full calculator design before it was interfaced with the software. This was done only after we validated that each component worked on its own.

Without these crucial steps, it would have been much more difficult to troubleshoot later down the design process when we began combining the components into the fully finished product. Lastly, once the entire design was connected, we used Vivado SDK to write inputs and read outputs from the calculator.

IV. Results

A. Adder/Subtractor

The adder/subtractor as a complete component was taken from Unit 7: Dynamic Dual Fixed-Point Adder with interrupt-based reconfiguration (ZYBO Z7-10). Various additions and subtractions were used to test the functionality of this component. Special attention was given to mixed exponent values in order to test the robustness of the adder and subtractor in handling DFX numbers. Two numbers of particular interest are x3200 (Dec 50) and x8260 (Dec 100). 3200 is a num0 in the [16 8 4] DFX format while 8260 is num1.

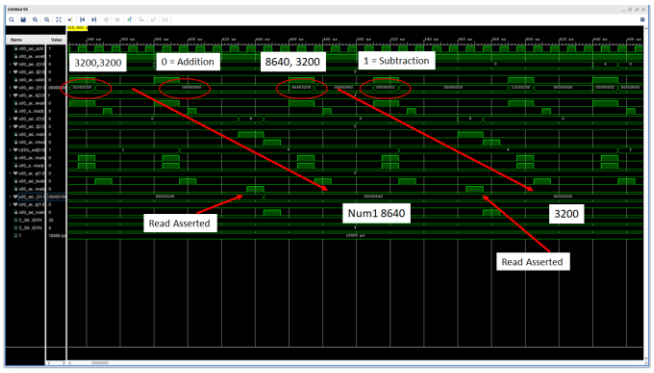

Figure 7: Adder/Subtractor results

B. Multiplier

The multiplier was tested in a similar fashion to the adder/subtractor. Multiple operands were used to check the full functionality of the component. Num0 and Num1 DFX numbers were considered as well as mixed sign operands. The latter is of special interest in order to determine that the sign2unsign component conversions were being performed correctly.

The multiplier test benching did take some significant debugging. This was a bit of a surprise because the project was functioning at the SDK level even before the test bench was fully developed. The failure was that when reading the result data from the calculator the data was showing up in the test bench as undefined. After careful inspection, it was found that the component, myMultiAdder, contained variables that were not initialized. This undefined data resulted in an undefined result being shown at the test bench level.

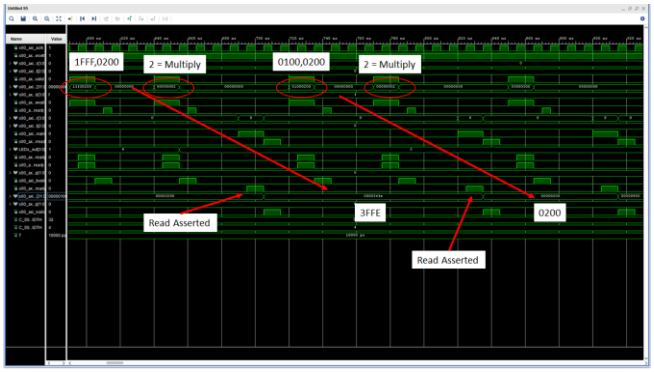

Figure 8: Multiplier Simple results

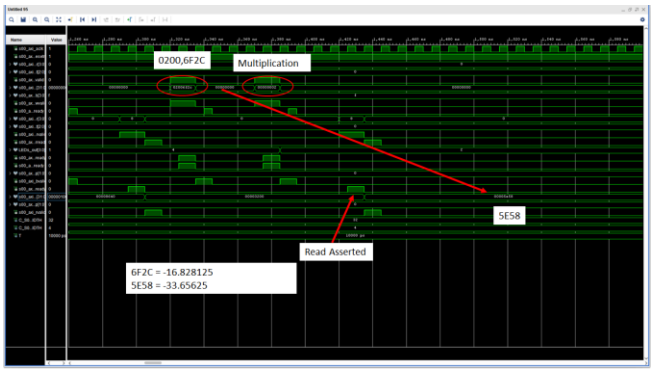

Figure 9: Multiplier Negative results

C. Divider

Shown in the figures below are the testbench results of the divider IP. The divider was tested using different combinations of operands in terms of scaling value and sign. This was done to test that not only the division was working properly but that the pre-scaler, range detection, and 2's complement circuits were also functioning. The divider takes N+1 cycles to complete a division operation. So in the testbench, after two operands were written the E signal would go high which triggers the divider circuitry. After N+1 cycles the result Q is available.

Figure 10 shows the dividend 4 and divisor 1 in n0 and n1 scaling, respectively. We see the correct result 2. This demonstrates that the pre-scaler circuit converted the divisor 2 into n1 form before completing the division.

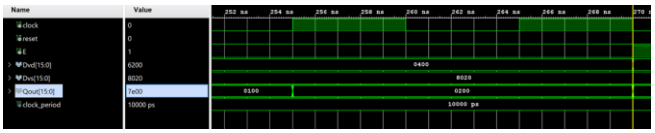

Figure 10: Dividend 4 (n0) Divisor 2 (n1)

Figure 11 shows the dividend -30 and divisor 2 in n0 and n1 scaling, respectively. We see the correct answer output, -15. This demonstrates that the 2's complement operation worked as it should by detecting a negative operand and complementing the output.

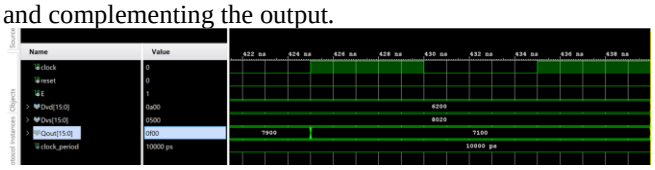

Figure 11: Dividend -30 (n0) Divisor 2 (n1)

Figure 12 shows the dividend -9 and divisor -3 both in n0 scaling. We see the correct answer 3. This demonstrates the pre-scaler circuit correctly identifying the two operands having the same scaling and also performing the 2's complement operation before division due to both operands being negative.

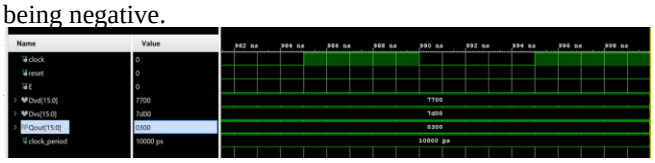

Figure 12: Dividend -9 (n0) Divisor -3 (n0)

Figure 13 below shows the dividend 7.5 and divisor 2.5 both in n0 scaling. This test demonstrates two operands that have fractional bits.

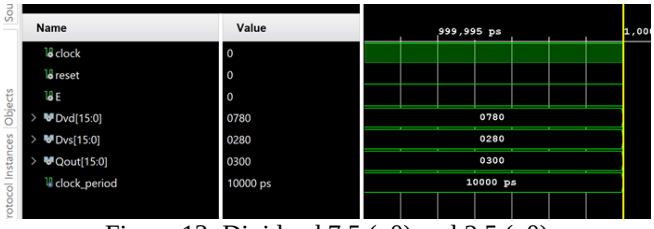

Figure 13: Dividend 7.5 (n0) and 2.5 (n0)

D. Full Design

The Adder/Subtractor, Multiplier, and Divider were integrated into a larger Calculator IP. The Adder/Subtractor and Multiplier performed as expected on the test bench. Unfortunately the Divider results, once fully integrated into the calculator, were not correct. The Adder/Subtractor and Multiplier required no additional inputs besides standard AXI inputs, namely the AXI clock and slave registers. The Divider also required an enable bit for an embedded state machine. This enable bit was never fully included into the calculator resulting in incorrect results.

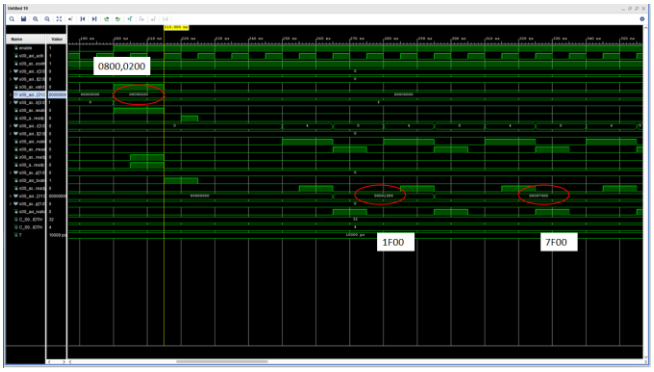

Figure 14: Implemented Division results

The end goal for this project was to input data from an external source and read the result. For this the Vivado SDK was used. A basic C program was used that wrote data to the slave registers and read the result. In testing, the SDK performed the same as the test bench. It was discovered in the SDK tests; however, that if the divider component was run twice in a row, the second set of results would be as expected.

V. Conclusions

The main take away from this project was to show just one purpose of using Dual Fixed-Point numbers in an electrical engineering application. We demonstrated a calculator capable of four different basic arithmetic functions. We used the Dual Fixed-Point format of [16 8 4], however since we designed the project components as generic, this could be implemented with any format.

We were also able to demonstrate the advantages that dual fixed-point has over floating point and fixed-point. Being more resource efficient and having comparable dynamic range to floating point. There are, however, some potential opportunities for improvement to this project. One example is the circuit doesn't validate the inputs of the user. Meaning, it doesn't detect that the user actually inputted a number in [16 8 4] dual fixed-point format. This would then cause the result to be incorrect.

Another potential improvement would be handling overflow. This circuit doesn't detect and appropriately flag an overflow so if two operands were inputted that resulted in an overflow there would be incorrect results. The circuit also only accepts two inputs at a time since it is an AXI Lite interface. If there was a need to pipeline data and results, then the AXI interface would need to be redesigned to a FIFO full interface.

There were a variety of implementation issues involved in this project. The first design had mode select decided by switch inputs directly to the calculator IP. This was a particularly pleasing implementation because it showed how changes to the physical hardware would result in different results when the software was run. Unfortunately while integrating the adder/subtractor into the calculator it forced one of the switch signals to become a generic clock. Much time was spent trying to resolve this failure but in the end the design was changed to take two additional data bits to decide on the mode selection and scrap the switch inputs. All was not lost; however, because this incorporated a second slave register for the data read and overall this achieved the same level of complexity.

One area of improvement would be in the overall design of the multiplier. The multiplier was designed with an approach best described as a C-programming approach. The VHDL is written in a very linear fashion using components as if they were functions. Processes are used to control the flow of logic and maintain this systematically linear approach. This is very different from the intention of VHDL. Incorporating better component logic into the design and eliminating some of the pseudo C implementation would likely make the project much more efficient from a resource perspective.

Overall, this project effectively demonstrates a hardware and software implementation on an FPGA device with hardware design based around using Dual Fixed-point numbers and showing their potential use in everyday engineering applications.

VI. References

[1] *Daniel Llamocca, "Unit 1 - Computer Arithmetic" in ECE-5736 Reconfigurable Computing*

[2] *Daniel Llamocca, "Unit 3 - Special Purpose Arithmetic Circuits and Techniques" in ECE-5736 Reconfigurable Computing*