Fixed-Point Calculator

Kushagra Gupta, Roman Hryntsiv
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: kgupta@oakland.edu, rhryntsiv@oakland.edu

***Abstract*— Calculators are widely used in the real world. Everyone uses it at least once in their lives. The fixed point calculator is designed to perform calculations with fixed point numbers. This project implements a fixed point calculator by using the AXI-Full bus for communication between PS and PL of the Xilinx Zybo board.**

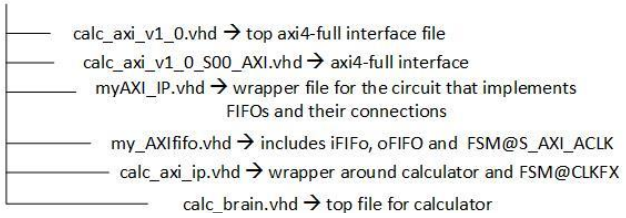
I. Introduction

Fixed point representation is an efficient method to represent fractional numbers in fixed number of bits. This is highly required in embedded systems where the resources available for computations are limited. Fixed point numbers are widely used due to the benefits of them. The benefits of fixed point arithmetic are that they are as straight-forward and efficient as integers arithmetic in computers. We can reuse all the hardware built for integer arithmetic to perform real numbers arithmetic using fixed point representation. In other words, fixed point arithmetic comes for free on computers. It is a simple yet very powerful way to represent fractional numbers.. By reusing all integer arithmetic circuits of a computer, fixed point arithmetic is orders of magnitude faster than floating point arithmetic. This is the reason why it is being used in many game and DSP applications. Even though fixed point numbers are easier to implement and understand then floating point numbers, they still require a certain order and general understanding. They are represented and interpreted in a specific manner; thus, we need a special calculator to perform arithmetic operations on fixed point numbers. In this project, we implemented addition, subtraction, and multiplication operations of the fixed point calculator.

This project is implemented utilizing the AXI4-Full protocol. The AXI protocol is used for the communication between PS and peripheral. Advantage of the AXI4-Full protocol is that it allows to read and write data in bursts, therefore allowing to exchange more data at faster speed.

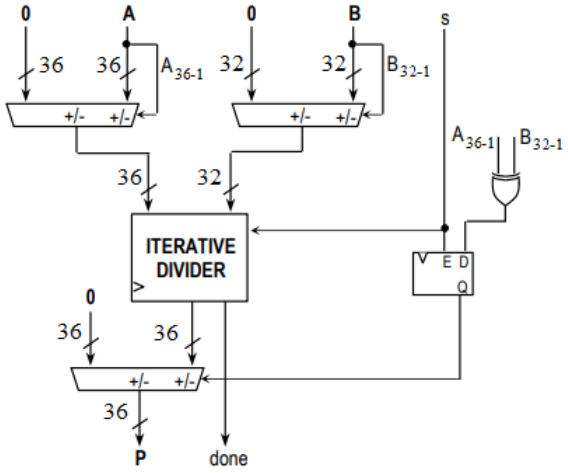
II. Methodology

The complete calculator project can be broken down into 3 main parts: calculator body, AXI4- Full interface and software design. Following diagram is the high level file structure which provides the details about what part of logic is implemented in which file for our HDL part.

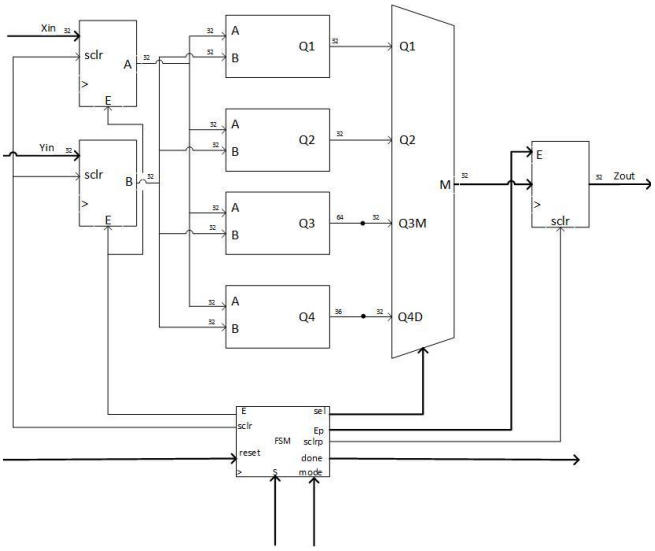


A. Calculator body

Calculator body is the component responsible for performing the fixed-point calculation. This is the component that was fully designed by our team. It has 4 sub- components - Adder, Subtractor, Multiplier and Divider. All the 4 components are designed to perform operations on 32-bit numbers. As for the adder, subtractor and multiplicator for our circuit we used the generic signed adder, subtractor and multiplier. As for the divider we used a generic unsigned divider that was transferred by us to the signed divider [32 4]. The diagram of the transferred divider is provided below.



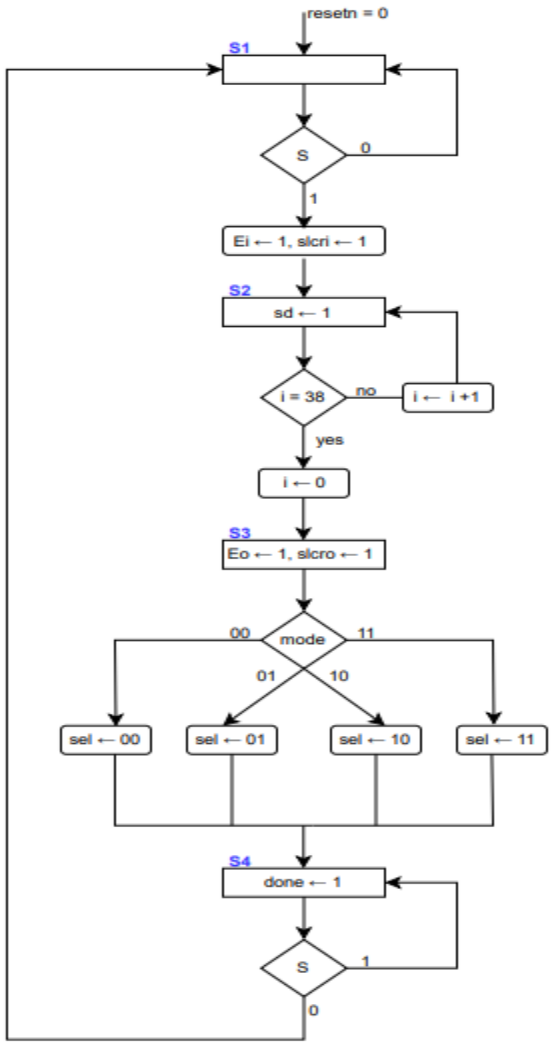
And here is the interconnection of our arithmetic units and the rest of the datapath.



It works the following way. The values Xin and Yin, and mode for our circuit are provided from the outside. The Xin and Yin are 32 bit words and the mode is a 2 bit word. The Xin and Yin are stored into our input registers and then they go to our arithmetic units while mode goes directly into the FSM. After the performance of the arithmetic units, the output values go into the multiplexor. The outputs of the adder and the subtractor are the 32 bit words; however, the output of the multiplier is a 64 bits word. Here we perform truncation. By doing the truncation we lose some of the precision which is very little and has no effect on our final project. The output of the divider needs to be truncated as well because it consists of 36 bits. The reason we have 36 bits and not 32 is because we appended 4 zeros to it, to have additional 4 zeroes after the period. At the multiplexor, only one set of data from our arithmetic units is picked and written into the output register. The output of the register is the output of our calculator body.

The "brain" of this body is the FSM. The FSM controls most of the signals in the circuitry. It is the most important component in our calculator body, so we will cover it in more details below. In a nutshell, the FSM is designed to control the input data to the arithmetic units and to retrieve the data from them. The diagram of the FSM is provided on the right side of the page.

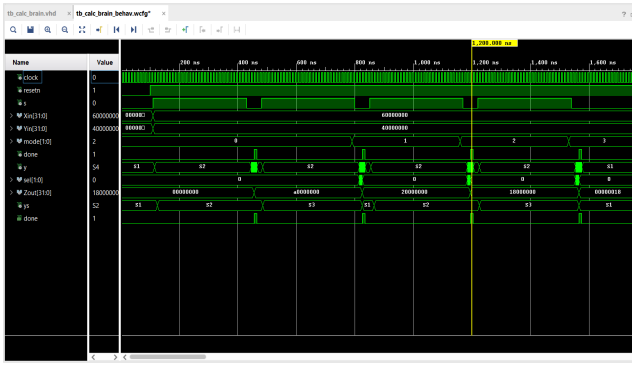
At the beginning, the reset value is given to make sure that we have no random data on our registers. After the reset is given we go to the state 1. During the state 1 we make sure that the signal s is given, which means that our data is ready to be read. After we receive s =1, we declare Ei and sclri equal to 1. It allows the input register to get and store the input data from inputs Xin and Yin. After that we go to the state 2 where we produce a signal sd to our divider, letting it know that the data is ready to be received.



During the state 2 we also verify that the circuit has enough time to process data. Since the arithmetic units are mainly combinational circuits, we will have a roughly 20 clock cycle delay for our adder, subtractor and multiplier. However, our divider requires at least 32 clock cycles delay to perform the operation, so in the state 2 we embed a counter into our FSM and make sure that it counts until 38. Before the counter reaches 38, it stays in the state 2 and adds one to its value every clock cycle. After the counter reaches its maximum value it zeroes itself and we go to the state 3 (the initial value of the counter is set to 0 to make sure we start counting from the beginning.) After we wait for 38 clock cycles, it is time to retrieve the data from our arithmetic units. First, we have to make sure that our output register is ready to receive the data. For that, before giving it any value, we set the values of the signals Eo and sclro to 1. After, we read the value of the signal mode which is given from the iFIFO. The signal mode tells us what operation must be performed. The FSM provides a signal sel which is hooked up to the selector of the multiplexor. The inputs of

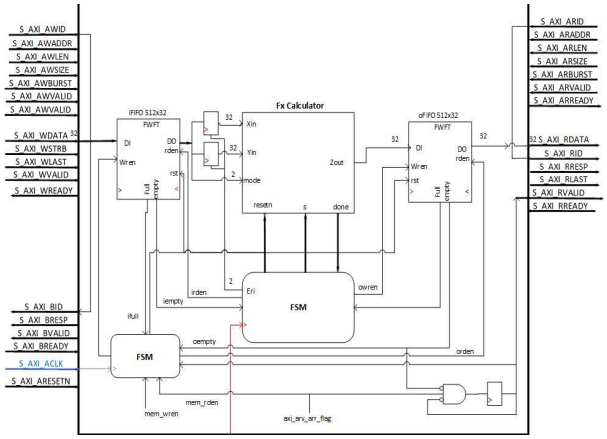
the multiplexor are the outputs of the arithmetic units. Accordingly, the value of the signal mode creates a value of the sel signal, and the correct data is loaded into the output register. The last state is the state 4. In this state we produce a signal done and verify that the value of s is 0. While the value of s remains 1, we stay in state 5. After the value of s is 0 we go to the state 1 and the cycle starts again.

The important part of a successful design is running a behavioral simulation of a circuit to make sure everything works properly. In our case we have a testbench for the calculator body by itself, the body with the peripherals, and the software testbench. The first testbench is a purely VHDL testbench of the calculator body. The signals Xin, Yin and mode are given certain values. Signals s is also included in the testbench since it begins the cycle. At the beginning we assign Xin and Yin as 1 and 3 correspondingly and wait for a clock cycle. After, we correspond s as 1 and wait for 38 clock cycles to make sure that the values are ready. After that, we have signal mode as 00 and signal s as 0. We have to add an additional few cycles of wait, and after we could compare our data. After that we change values of mode and run additional sets of data to verify that all of the arithmetic units work properly. This is the simple testbench regarding verifying that our circuitry works.



B. AXI4- Full

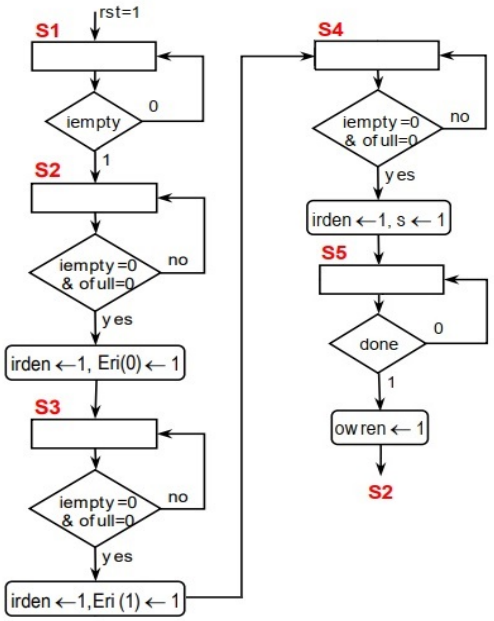
To interconnect the PL and the PS part of the Zynq 7000, we use AXI4-Full in this project. The advantage of using AXI4-Full is that we can read/write data in bursts, so we load the data into our circuit in the burst. To interface the calculator peripheral with AXI-Full, we added some additional circuitry described below the diagram. Following diagram depicts the circuit designed for it.



The provided above circuit contains two FIFOs, two FSM machines, the calculator body described in the part A, some additional registers, and "glue logic." The memory functions of our AXI peripherals perform by two FIFOs, (the original memory was replaced by them) one for the inputs and the other for the output signals. A massive advantage of the FIFOs is that their inputs and outputs can run on different frequencies, allowing users to implement even circuits with different frequencies inside of the PSoC. However, to implement that advantage we need to have separate FSMs that would control the inputs and the outputs of our FIFOs respectively. The first FSM is implemented by the template, (modified to replace the original memory by FIFOs) and it regulates the input of the iFIFO and the output of the oFIFO. Both of them are run by the inside clock of the chip, and the FSM can take care of both of them. The FSM provides the FIFOs with signals wren for the iFIFO and rden for the oFIFO that allow data to be written to the iFIFO and to be read from the oFIFO. The iFIFO also has a flag full, informing the FSM that the FIFO is full. The oFIFO has a flag empty that informs that it's empty. Also, one of the main functions of the first FSM is that it provides the active high rst signal not only to the FIFOs, but also to the rest of the components in the circuit. The RP outputs tend to toggle erratically and the reset signal ensures that no spurious data is transferred into oFIFO. The second FSM controls the output of the iFIFO and the input of the oFIFO. It also controls the calculator's body and the "glue logic." This is the FSM that was fully developed by our team. The FSM controls the output of the iFIFO and input of the oFIFO similarly to the way the first FSM controls the input of the iFIFO and the output of the oFIFO. It produces a signal irden that allows data to be read from the outside input of the iFIFO and a signal owren that allows data to be written onto the oFIFO. Also, it checks for a signal iempty from the iFIFO to check whether the iFIFO is not empty and ofull from the oFIFO in order to verify that the oFIFO is not full.

Additionally, the FSM controls the "glue logic," specifically the input registers of the "glue logic." It gives the values Eri to the registers that allow writing data onto them. In terms of the calculator body, it provides it with the signal s that signals the component inside of it to start performing the calculation. When the calculator body finishes the

calculation, it inserts a signal done to our FSM. The FSM diagram is provided below and the performance of it follows it.

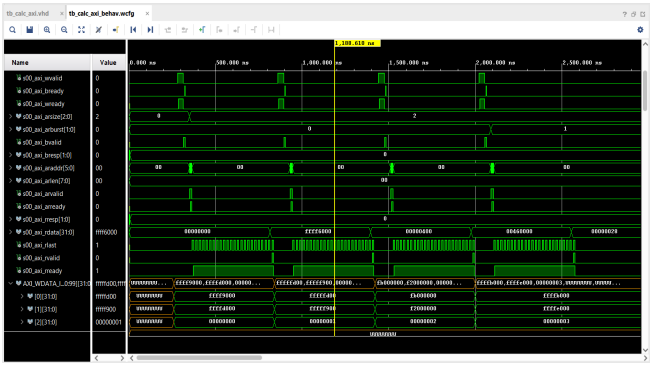


FSM @ CLKFX

After the signal reset is given, we go to the state 1 where we wait for signal iempty to be 1. If the signal is 0 we stay in the state 1. After the signal iempty is 1 we go to the state 2 where we wait for signals iempty and ofull to be 0. If any of them is 1 we stay in the state 2, but if they both are 0, we give the signals irden and Eri (0) value of 1 and go to the state 3. In the state 3 we wait for iempty and ofull to be 0 again, and stay in the state 3 if they're not; however, this time after they are 0 we give irden and Eri (1) value of 1, instead of the signals irden and Eri (0) as it was in the state 2. After giving the signals irden and Eri (1) value of 1 we go to state 4 where we again wait for the signals iempty and ofull to be 0. We stay in the state 4 if any of them is 1, but if both of them are 0 we give the signals irden and s value of 1. This is where the calculation process starts. In the state 5 we wait for the signal done to be inserted which means that the calculation is finished. If done is 0, we stay at the state 5, but if the done is 1 we give owren the value of 1 and go to the state 2. After that, the cycle starts again.

One of the last and most important parts of the AXI peripheral circuit is reading the correct data from S_AXI_RDATA. The signal S_AXI_VALID is inserted when the right data is ready to be read. The signal comes from the register that is connected to the AND gate. The inputs of the AND gate are the signals oempty, mem_rden, and the output of the register from the previous sentence. That mini circuit guarantees that we get the right data at the right time.

To verify the proper operations of our circuit we must test it again by using another testbench. This time we used more complex negative numbers, to see the full performance of our circuit from those two testbenches.



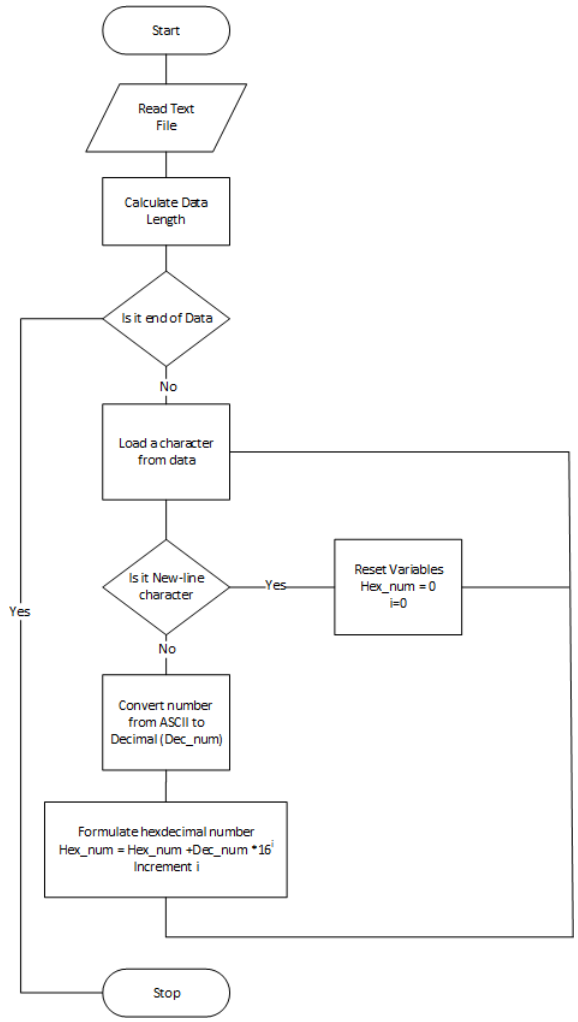
C. *Software Design*

Our system includes the FPGA part (PL) and the ARM microprocessor (PS) part of the Zynq 7000 chip. To make the PL part fully embedded with the PS part we have a C code for our system that would communicate directly with the microprocessor. We use the C code for loading and retrieving the data in and out of the circuit. Since we have a lot of combinations of data for our circuit, we use an external SD card for loading the data. The C code provides us with the connection between the microprocessor and the SD card. Also, it specifies the additional settings for loading the data correctly into our calculator inputs. Finally, it allows data to be kept and written as a text file into our SD card from the outputs of our circuit. Since the SD card is a FAT32 system, we have to configure the board support package to use xilffs library. This library provides all the functions to work with writing and reading data to/from the SD card. To read data from the SD card, we mount the SD card and then open the text file to read data from it. Since it reads data from a text file, all the data would be read in ASCII format. Therefore, we have added logic to decode this data and convert it into useful hexadecimal numbers. Since we read 32 bit hexadecimal numbers from the file, we use the following logic to read data. We read all characters in a loop and separate out separate numbers whenever a new line character is read. We also convert ASCII values to corresponding hexadecimal numbers. For example, A is read as 65 which needs to be converted to 10(decimal equivalent of hex number). After doing this conversion, we construct a 32 bit hex number by multiplying a byte by $16^{\text{position of byte}}$. After we add all these values, we get a equivalent hexadecimal number which can be written to memory address.

The input data and the final results are also displayed as hexadecimal characters on the SDK terminal via the UART port.

To test our final project, we reuse the AXI4-FULL peripheral we designed for cordic in Lab 3. We write the input data into a test file which is loaded on the SD card. Then this SDK software design was used to read data from

text files and write to memory registers. We displayed the input data on the SDK terminal and its expected output data that confirms the software design works as expected.



We read multiple text files which contain data for different operations. This allows us to test the calculator on hundreds of numbers on each operation. Displaying the data on the SDK terminal is also very beneficial because we can verify the output immediately instead of connecting the SD card to the laptop and then reading the output file to verify the results.

III. Experimental Setup

We tested each component individually in the project before integrating all the components together and then the final testing was done on the integrated project.

On the hardware side, we used Vivado test benches to test the calculator for individual operations. Once all the operations were tested, the axi-full peripheral was tested using the Vivado test bench. During this testing, an issue was identified where the calculator was reading only one setup of inputs. It was identified and fixed with the help of the professor.

For the software design, we started by reading and writing data to text files on the SD card. The SDK terminal was primarily used for debugging. We would display the output of different processes implemented in the software on SDK terminal to verify that each process worked as expected. This helped us to easily pinpoint the problem and fix it.

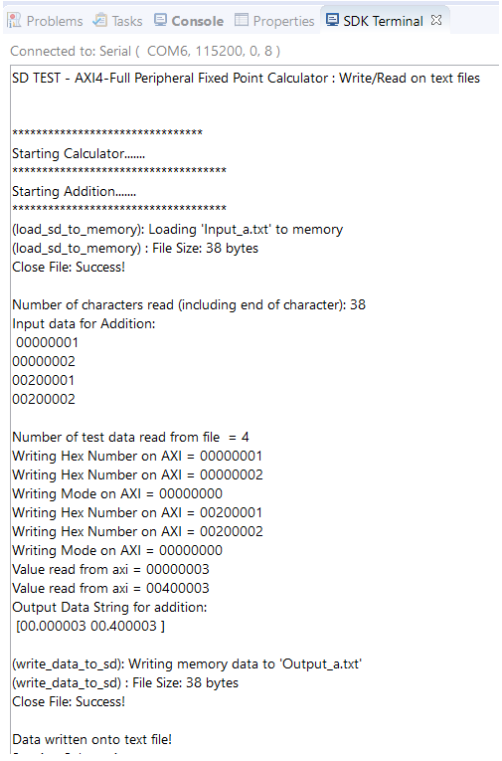
After this functionality was tested, we used lab3 project, to test reading data from SD card, writing it to axi full bus and reading the output which was further written to SD card. We printed all the data which was written and read back from the AXI bus which confirmed that data is not correctly processed in hardware. Eventually, all the data to be written to text files was also printed on SDK and compared with text files to verify write operation is performed without any issues.

After successful testing of hardware and software design individually, we tested the integrated functionality. Initially, we started with reading a few test data from text files and writing corresponding output to text files. After verifying the complete functionality, we tested it for a huge set of input data and multiple input and output files.

IV. Results

Following are the test results which were obtained on the SDK terminal.

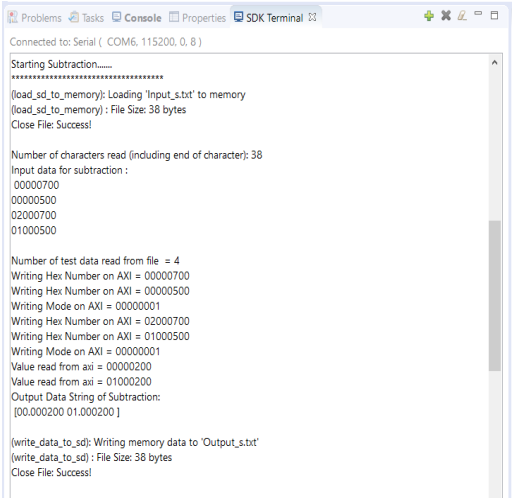
Addition: Following is the snapshot of all the data displayed on SDK terminal for addition operation.



In this test, 4 numbers are read from the input file named “Input_a.txt”. These 4 input numbers are written on axi bus in a set of 2 inputs along with the operation to be performed

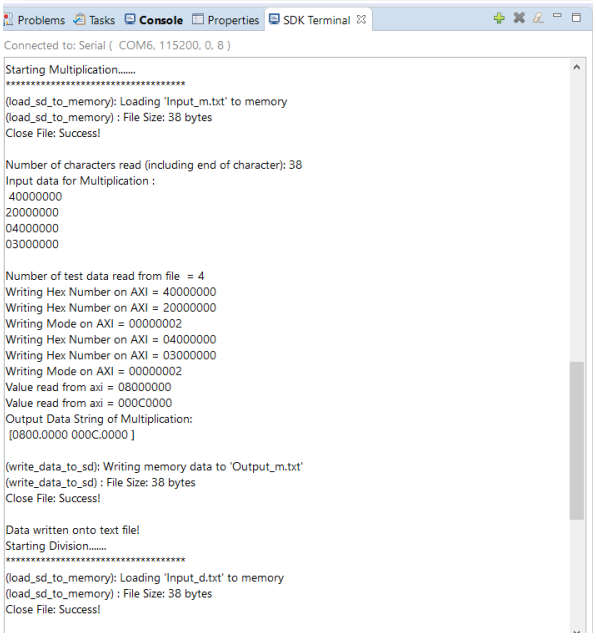
(0 represents addition). After writing all the inputs to axi the bus, output of the addition operation is read from axi bus. 2 numbers in the format [32 24] are read from axi bus and are written on the output file “Output_a.txt”.

Subtraction: Similar to addition, following is the snapshot of data written on SDK terminal for subtraction operation.



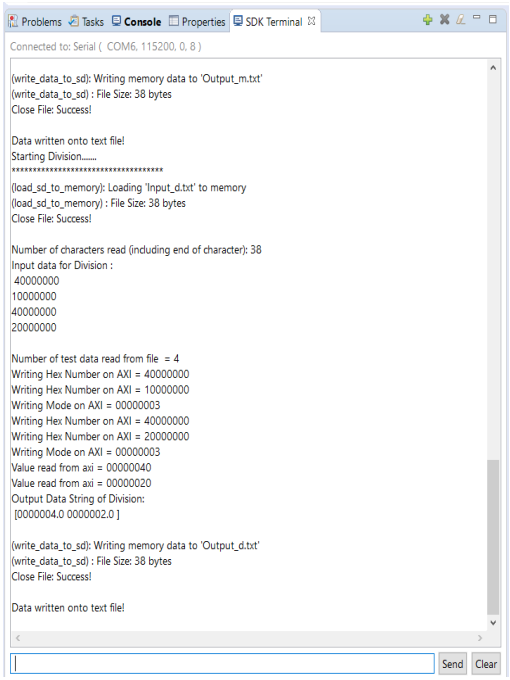
For subtraction, we also tried reading 4 input numbers and 1 is used to select the subtraction operation. Output numbers are generated in format [32 24] and are written on a text file.

Multiplication: Following snapshot data is for multiplication operation.



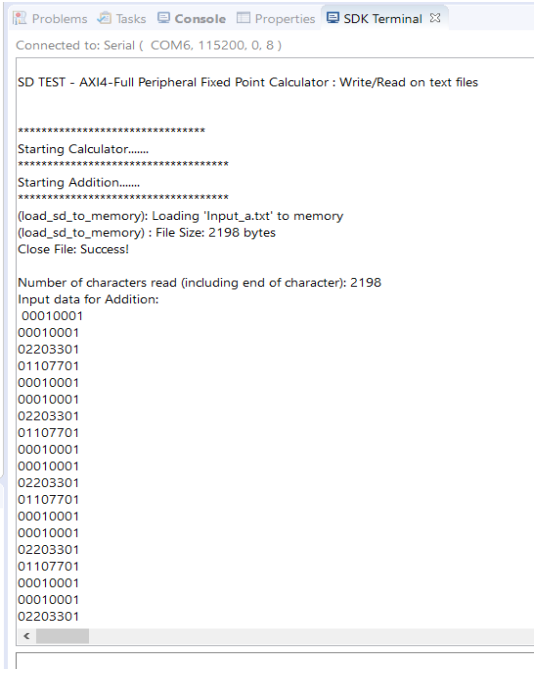
Again, we read 4 input numbers from the text file and they are written on axi bus along with the operation instruction which is 2 for multiplication. Output of multiplication is provided in format [32 16] which are written on output file “Output_m.txt”

Division: Following is the snapshot for division

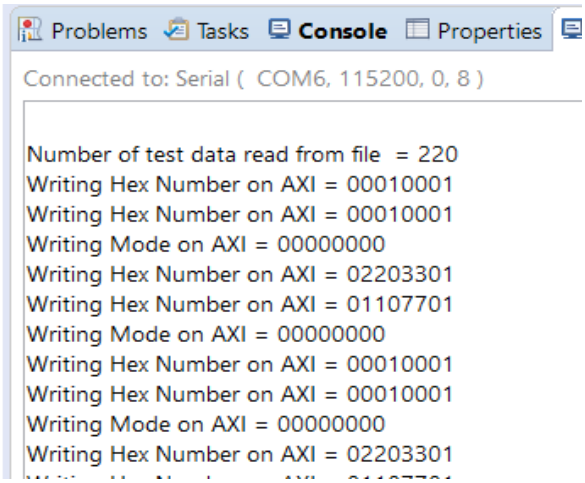


4 input numbers are read from a text file which are written on axi bus along with 3 which represents division operation. Two outputs are obtained in format [32 4] which are written on a text file.

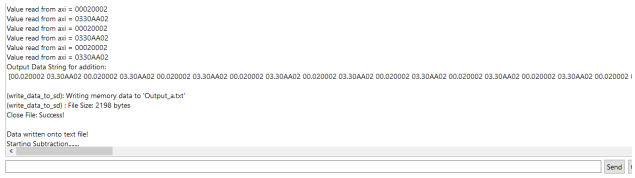
Following are the SDK snapshots when we read 220 input numbers to perform addition operation.This snapshot shows that 2198 bytes are read from the input text file which is a list of 220 numbers.



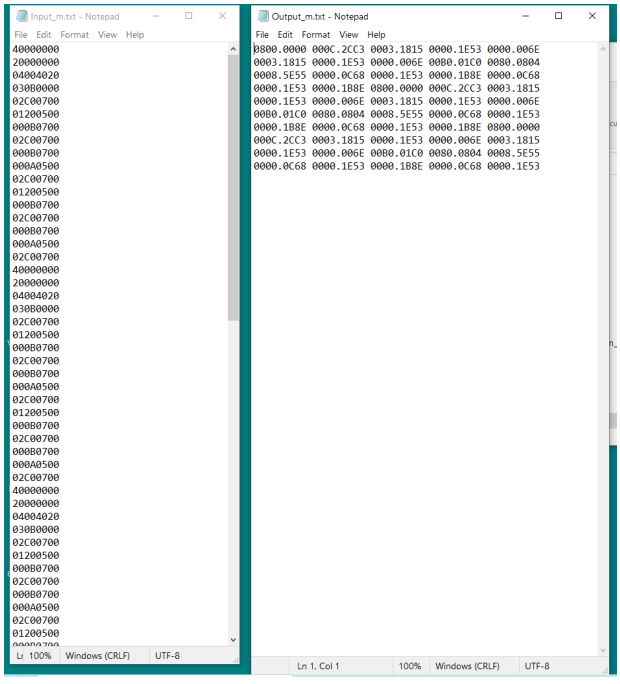
Following snapshot shows writing of 220 input numbers to the axi bus along with mode (0 for addition).



Following snapshot shows the output data which is written on the output text file (truncated in image due to large size)



Following is the snapshot of input and output text files used for multiplication operation. In the input text file, 100 numbers are present and the output test file has 50 numbers.



V. Conclusion

By finishing this project we fully implemented the fixed point calculator that can perform addition, subtraction, multiplication and division.

The original idea was to implement the calculator without the division portion, but eventually we implemented that part as well. After adding the subtraction part, we created the completed calculator fully integrated calculator to the ZYNQ board.

There's a few adjustments that might be done to perfect this calculator such as adding the overflow or improve the SDK terminal part or the output text files design, so that the results would be easily readable.

Another improvement which can be done is to use binary files instead of text files for reading input data and writing output data. We chose text files because they are easily readable. However, the professor suggested that binary files are more efficient as they do not store data in character format and no new line characters are required at the end of each line. We were able to read large amounts of data using text files but for industry applications, binary files are preferred as they can store more in small size compared to text files.

However, they are mainly additional changes and even without them the calculator is fully functional and performs its main functions well.

VI. References

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

[2] Daniel Llamocca, "Unit 5 - Embedded Systems in PSoC" in ECE-5736 Reconfigurable Computing.

[3] Manual by Digilent, "Zybo Reference Manual," *Digilent Reference*. [Online]. Available: https://digilent.com/reference/programmable-logic/zybo/reference-manual [Accessed: 10-June-2022].

[4] Martin A. Enderwitz, Crockett H. Louise, Ross A. Elliot, "The Zynq Tutorials For Zybo And Zedboard," Strathclyde Academic Media: August 12, 2015

[5] Martin A. Enderwitz, Ross A. Elliot, Crockett H. Louise, Robert W. Stewart, "The Zynq Book: Embedded processing with the ARM® Cortex®-A9 on the Xilinx® Zynq®-7000 All Programmable SoC," Strathclyde Academic Media: 2014

[6] Daniel Llamocca, "Embedded System Design for Zynq PSoC" in ECE-5736 Reconfigurable Computing.