CMPE 260 Laboratory Exercise 4

Execute Stage

Aden Crimmins
Performed: March 16, 2021
Submitted: March 30, 2021

Lab Section: 2
Instructor: Richard Cliver
TA: Corey Sheridan
Justin Soler
Jake Michalski

Lecture Section: 2
Professor: Mr. Cliver

Your Signature: *Aden Crimmins*

Abstract

The purpose of this exercise was to design the Execute Stage of the MIPS architecture. It utilized the ALU in tandem with its own processes in order to completely implement the logic for this stage. This stage takes in the register values, Memory and register control bits, a conditional bit for immediate, as well as the destination bits and ALU Opcode. Using the Opcode the ALU performs the desired operation providing the answer as the output. The memory and register control bits are passed through in this stage without modifications and the write data and address are updated with the correct values. This component and the ALU were tested using test benches designed to check the functionality of the design for example instruction opcodes with changing inputs. This lab was successful in creating the execution stage as well as properly testing it to ensure it functions as intended.

Design Methodology

The design was created using the Xilinx Vivado Design Suite and implemented on the Basys3 FPGA. The ALU was designed by combining logic functions created during the Introduction to Vivado and Simple ALU Exercise, in addition to a Adder/Subtractor and a Multiplier.

The Adder/Subtractor was created for a generic bit width by combining Full adders generated with the carry starting as 0 for addition and 1 for subtraction based on the OP input. The block diagram for this is shown in Figure 1.

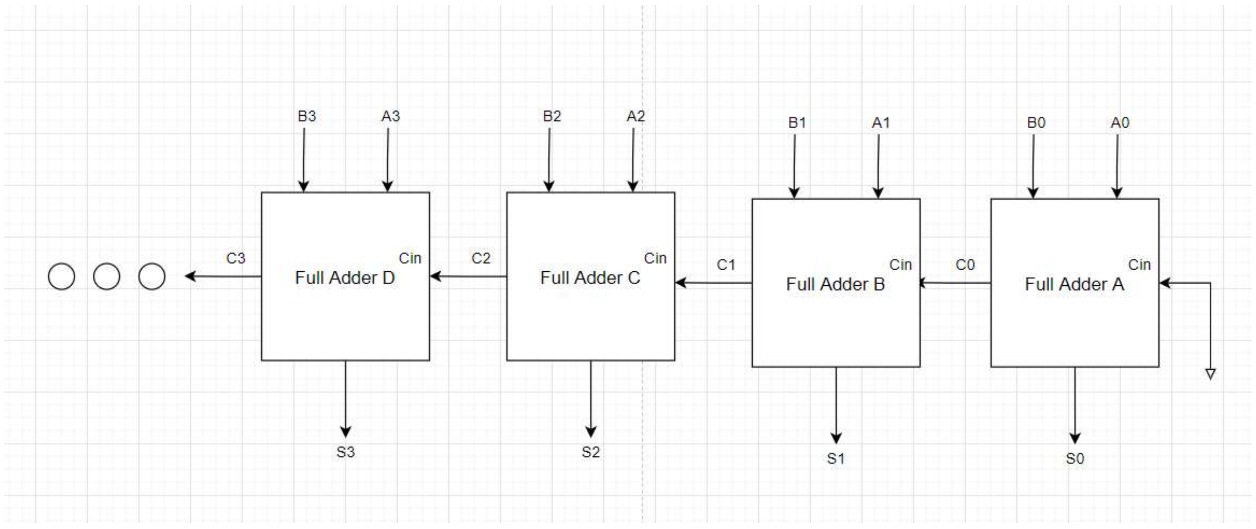


Figure 1: Adder/Subtractor Block Diagram

In order for the Full adder to switch between addition and subtraction, each bit of the second input is put through logic in which it is compared to the OP using an XOR. This determines if the 2's compliment is added or if it is the normal bit added. The sum is generated from these as each individual output is assigned to the Sum output.

The Multiplier was Created from a number of Full Adders that were connected in a number of rows and columns equal to the number of bits. The block diagram for this is shown in Figure 2.

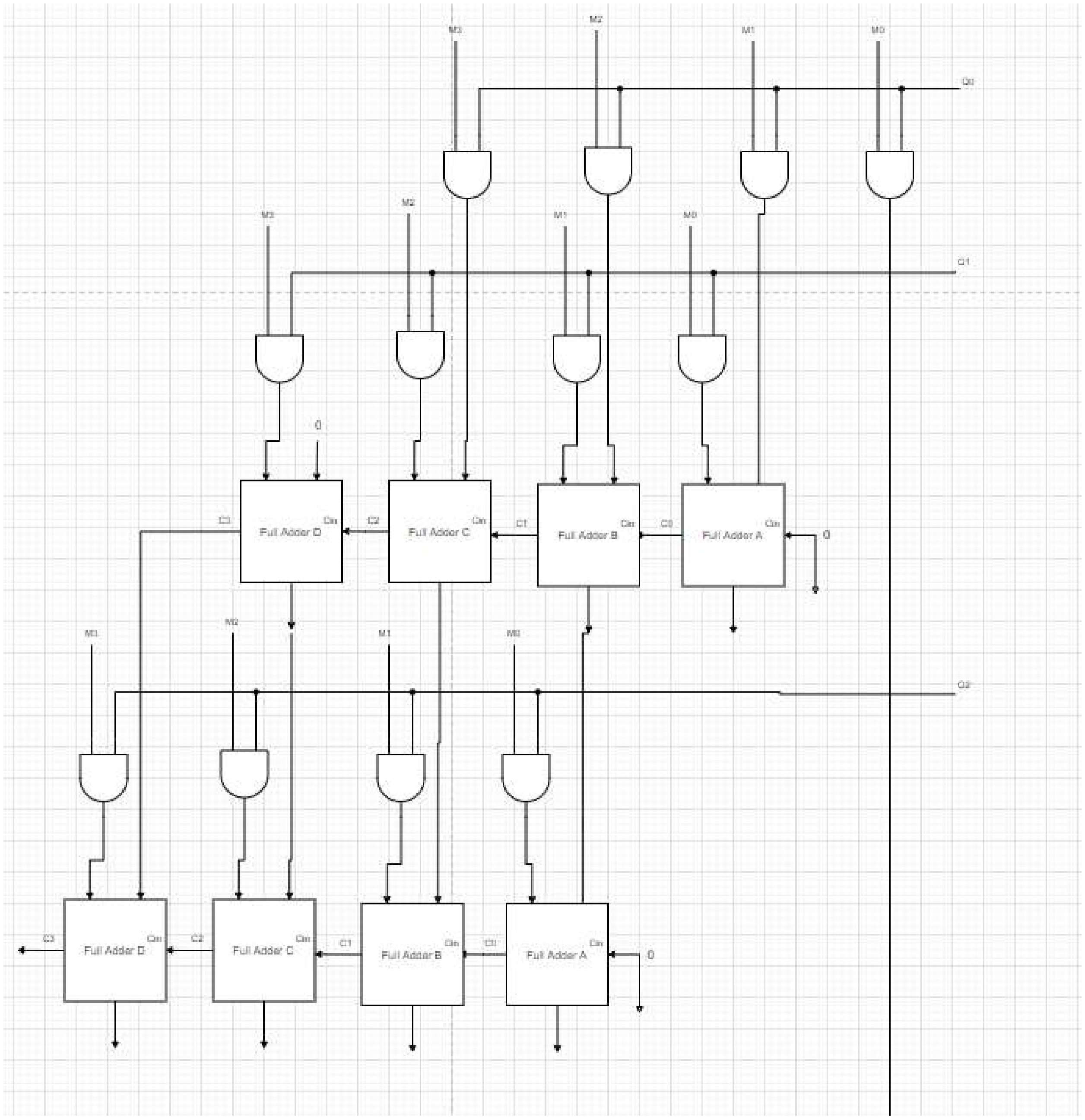


Figure 2: Multiplier Block Diagram

For the First Row the logic of each column of the first input was put through an AND gate with the first row of the second input. The first column of this process is used as the first bit of the output, and for each row following the first column is used as the bit in the sum indicated by the number of the row. For the first row it takes in the result of the AND operation as the input for every column excluding the last as there is not enough result bits in the previous row to cover that. Instead, the last column takes in a value of ‘0’ in place of the value from the AND operation. The second input comes from an AND operation between each column of the first input and bit specified by the row for the second input. The Carry for each operation goes into the next full adder in the same row until it gets to the final full adder in the row which has its carry routed as the second input to the last columns’ full adder in the next row. For the final row in addition to the first column, each column after that has its sum directly added to the output with the bit defined by adding the row and the column of the full adder. The final adder within the last row has both its sum and the carry added to the output, with its carry going into the final bit of the output.

The ALU logic was created for the logical gates NOT, OR, AND, XOR, Shift Left, Shift Right, Arithmetic Shift Right, Multiply, and the Add/Subtract. There are two sets of vector inputs A and B which contain a generic bit width to allow the code to be modular, and a 4-bit input OP to select which operation to complete. The Block diagram for the ALU is shown in Figure 3.

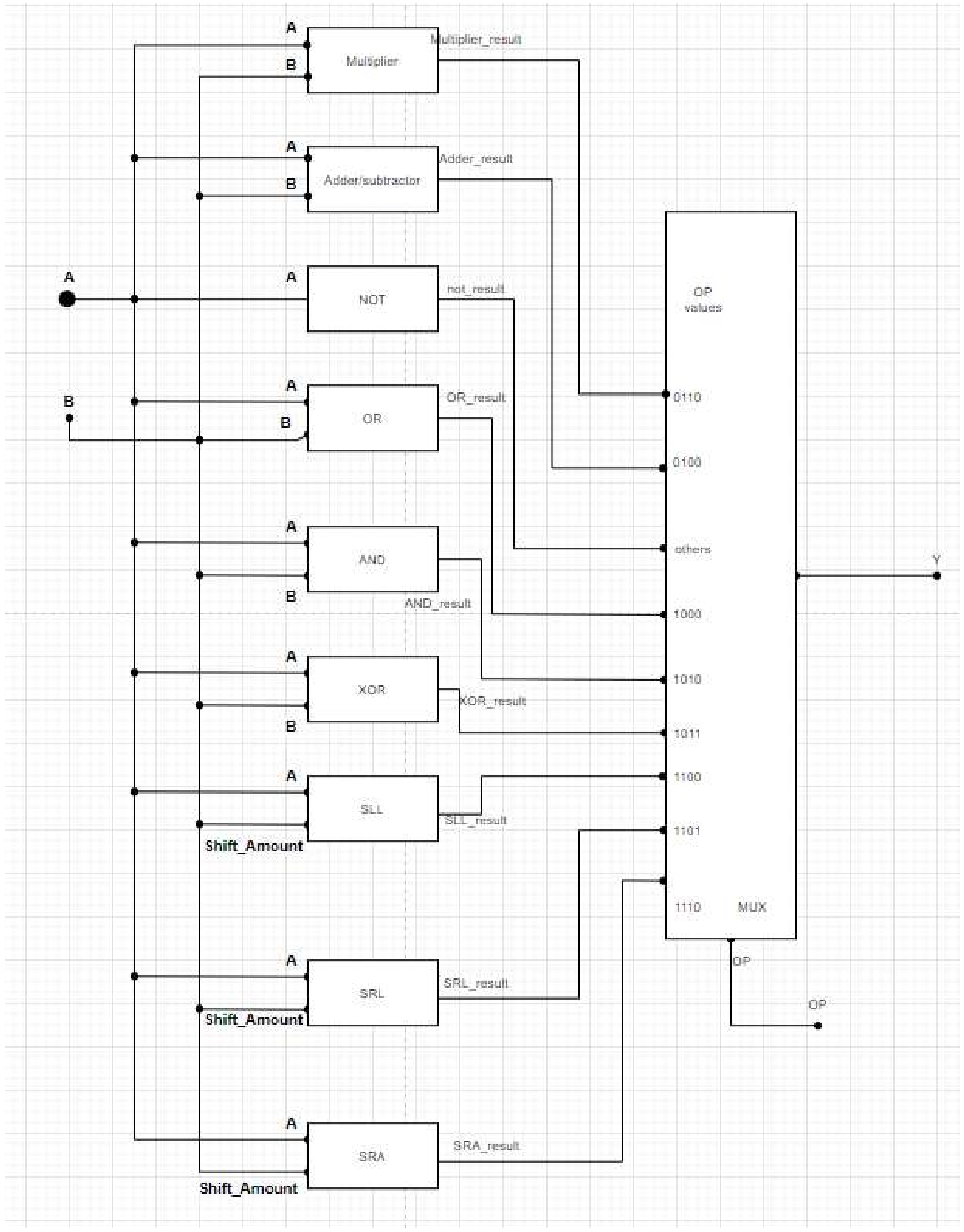


Figure 3: ALU Block Diagram

Within the ALU each of the operations are performed based on the inputs A and B, and the desired operation is selected through the use of a MUX that routes only the desired operation into the output.
The Execute Stage takes in the outputs from the Instruction Decode stage and performs all the necessary instructions. This is where the data is routed into the ALU as well as passing some of the provided bits through for future use, which are the RegWrite, MemtoReg, and MemWrite bits that were provided from the Instruction Decode stage. The WriteData and WriteReg outputs are also created for the SW instruction as it needs to write directly into the memory.

Results and Analysis

The testbenches created for this lab tested a variety of cases of each of the input values for the functionality of the ALU, Multiplier, and the overall Execute stage, as well as different operations within each to ensure full functionality of each stage and their components. Figure 1 shows the behavioral waveform for the ALU stage.

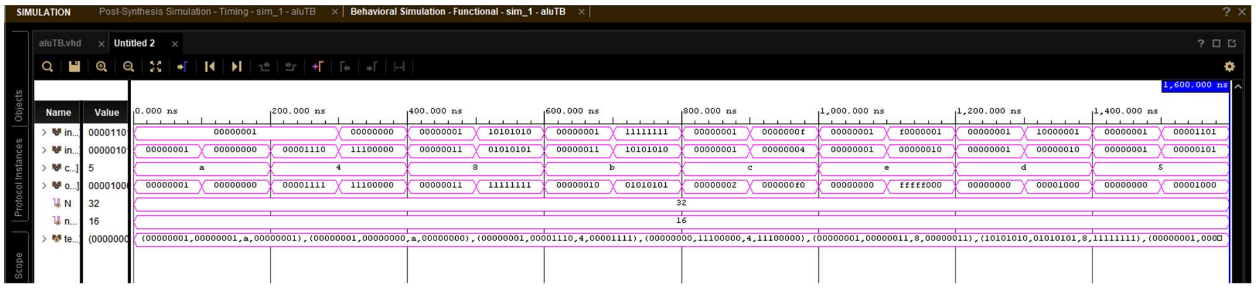


Figure 4: ALU Behavioral Simulation

The simulation begins by testing in sets of two for the ALU that does not have the multiplier added. The simulation in order tests the AND, ADD, OR, XOR, SLL, SRA, SRL, and SUB function in sets of two tests for each operation. Each of the operations tested by the test bench matched the output specified by the testbench without any errors being thrown.

Figure 5 shows the Post-Synthesis Timing simulation of the ALU setup using the same inputs.

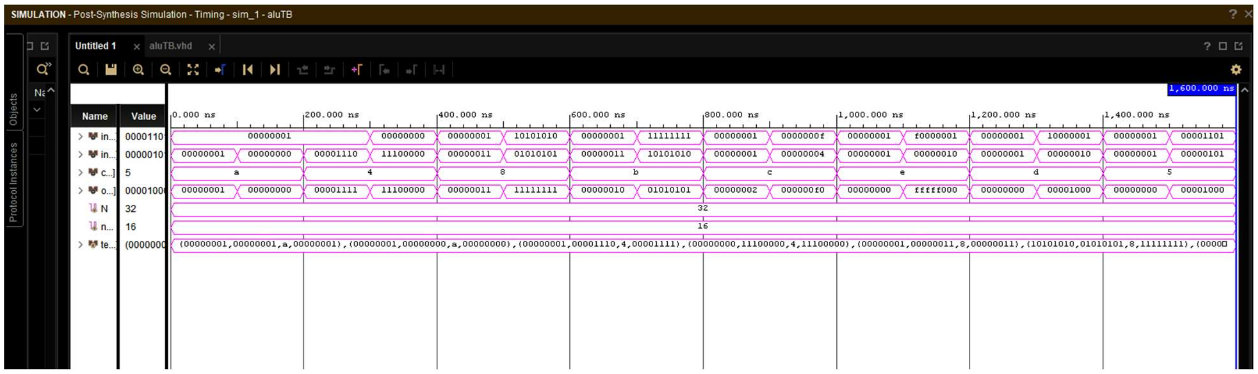
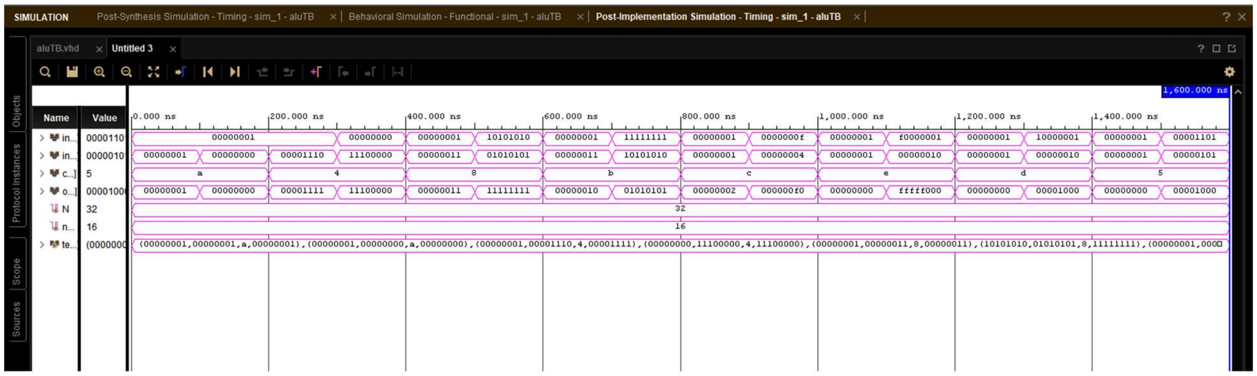


Figure 5: ALU Synthesis Simulation

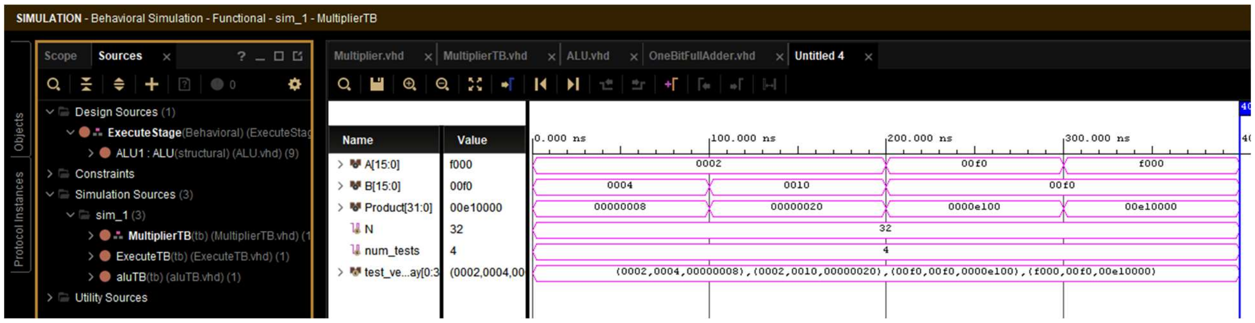
The Post-Synthesis timing simulation tests the gate level model of the code without accounting for any delays. This waveform shows the same results as the behavioral simulation which means that there arent any major design flaws that are visible within the synthesis simulation.
The next test was the Post-Implementation Timing Simulation shown in figure 6 which tests the same set of inputs as the behavioral simulation.


SIMULATION
Post-Synthesis Simulation - Timing - sim_1 - aluTB
Behavioral Simulation - Functional - sim_1 - aluTB
Post-Implementation Simulation - Timing - sim_1 - aluTB
aluTB.vhd
Untitled 3
Objects
Protocol Instances
Scope
Sources
1,600.000 ns

```
Name   Value     |0.000 ns            |200.000 ns          |400.000 ns          |600.000 ns          |800.000 ns          |1,000.000 ns        |1,200.000 ns        |1,400.000 ns
> in...  0000110   00000001                        00000000   00000001   10101010   00000001   11111111   00000001   0000000f   00000001   f0000001   00000001   10000001   00000001   00001101
> in...  0000010   00000001   00000000   00001110   11100000   00000011   01010101   00000011   10101010   00000001   00000004   00000001   00000010   00000001   00000010   00000001   00000101
> c...]  5         a                     4                     8                     b                     c                     e                     d                     5
> o...]  0001000   00000001   00000000   00001111   11100000   00000011   11111111   00000010   01010101   00000002   000000f0   00000000   fffff000   00000000   00001000   00000000   00001000
  N      32        32
  n...   16        16
> te...  (0000000  (00000001,00000001,a,00000001),(00000001,00000000,a,00000000),(00000001,00001110,4,00001111),(00000000,11100000,4,11100000),(00000001,00000011,8,00000011),(10101010,01010101,8,11111111),(00000001,000
```

Figure 6: ALU Implementation Simulation

The Post-Implementation timing simulation tested the implemented model as it would behave on the actual hardware. This includes the delays resulting from each gate in the model which can be seen whenever there is a change in the inputs as there is delay between that change and the resulting output. This delay is relatively small as there is relatively few gates that the logic has to move through due to the simple nature of the Instruction Fetch stage.
The Multiplier has its own testbench to confirm its' functionality and the behavioral simulation is shown in Figure 7.


SIMULATION - Behavioral Simulation - Functional - sim_1 - MultiplierTB
Scope
Sources
Design Sources (1)
ExecuteStage(Behavioral) (ExecuteStag
ALU1 : ALU(structural) (ALU.vhd) (9)
Constraints
Simulation Sources (3)
sim_1 (3)
MultiplierTB(tb) (MultiplierTB.vhd) (1
ExecuteTB(tb) (ExecuteTB.vhd) (1)
aluTB(tb) (aluTB.vhd) (1)
Utility Sources
Objects
Protocol Instances
Multiplier.vhd
MultiplierTB.vhd
ALU.vhd
OneBitFullAdder.vhd
Untitled 4

```
Name              Value        |0.000 ns             |100.000 ns           |200.000 ns           |300.000 ns
> A[15:0]         f000         0002                                         00f0                  f000
> B[15:0]         00f0         0004                  0010                  00f0
> Product[31:0]   00e10000     00000008              00000020              0000e100              00e10000
  N               32           32
  num_tests       4            4
> test_ve...ay[0:3 (0002,0004,00  (0002,0004,00000008),(0002,0010,00000020),(00f0,00f0,0000e100),(f000,00f0,00e10000)
```

Figure 7: Multiplier Behavioral Simulation

The Multiplier was tested in two sets to ensure correct functionality in both methods of use. The first test multiplied 2 by 4 to get an output of 8, then It multiplied 2 by 16 and got the correct result of 32. The next two tests were to ensure that the 4 most significant hex bits were working correctly by first multiplying the hex value of x"F0" by x"F0" with the output being the hex value of x"E100" which confirmed the correct result of the multiplication which is then shifted over my now multiplying x"F0" by x"F000" to get the result of x"E10000" which moves into the more significant 4 hex bits. These tests confirmed that the Multiplier functions as desired in this operation.

The Multiplier was then added to the ALU to complete its functionality. The ALU was then tested using the same behavioral simulation that had cases added to it to test the multiply functionality. This behavioral simulation is shown in figure 8.

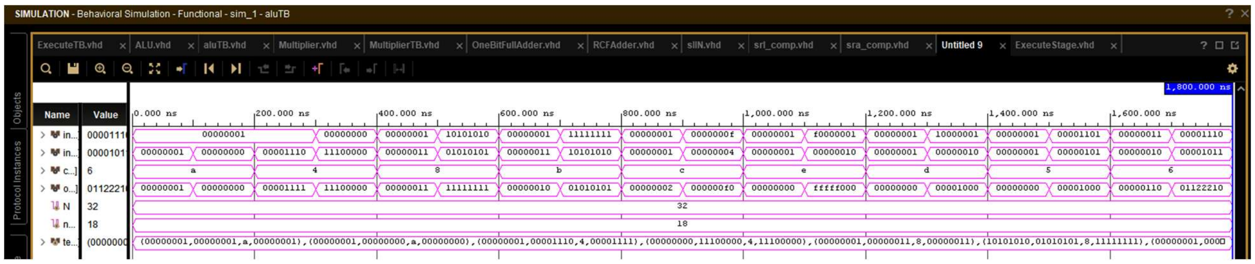

SIMULATION - Behavioral Simulation - Functional - sim_1 - aluTB
Name
Value
N
32
18

Figure 8: Complete ALU Behavioral Simulation

The Behavioral simulation shown in figure 8 contains the same tests as the original ALU simulations, with the added cases at the end for the multiplier in which x“11” is multiplied by x“10” to get x“110” and x“1110” is multiplied by x“1011” to get x“1122210” which are both the correct outputs for the multiplication.

The ALU was then put into the full Execute Stage and the behavioral test was run on the entire bench to test all possible Input combinations including those of R-type and I type instructions. This simulation is shown in Figure 9.

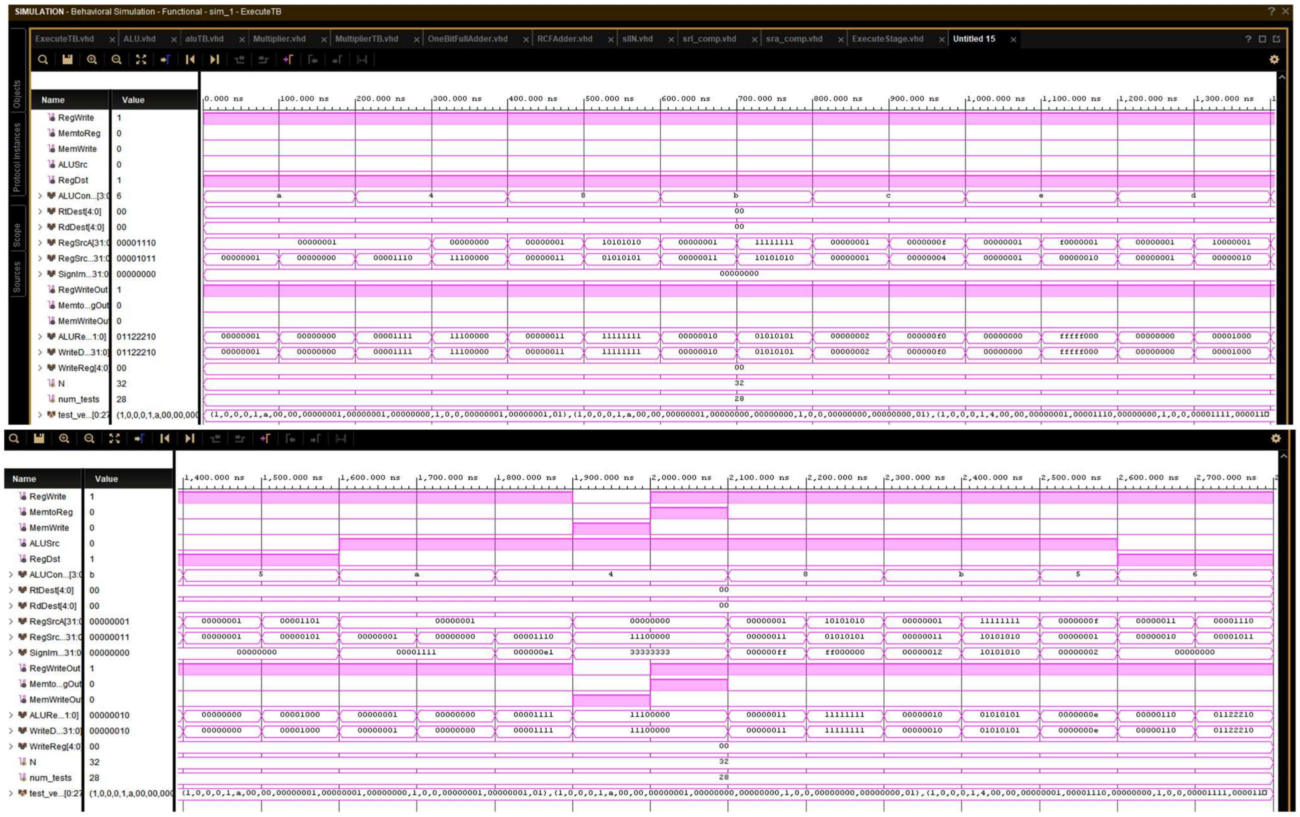
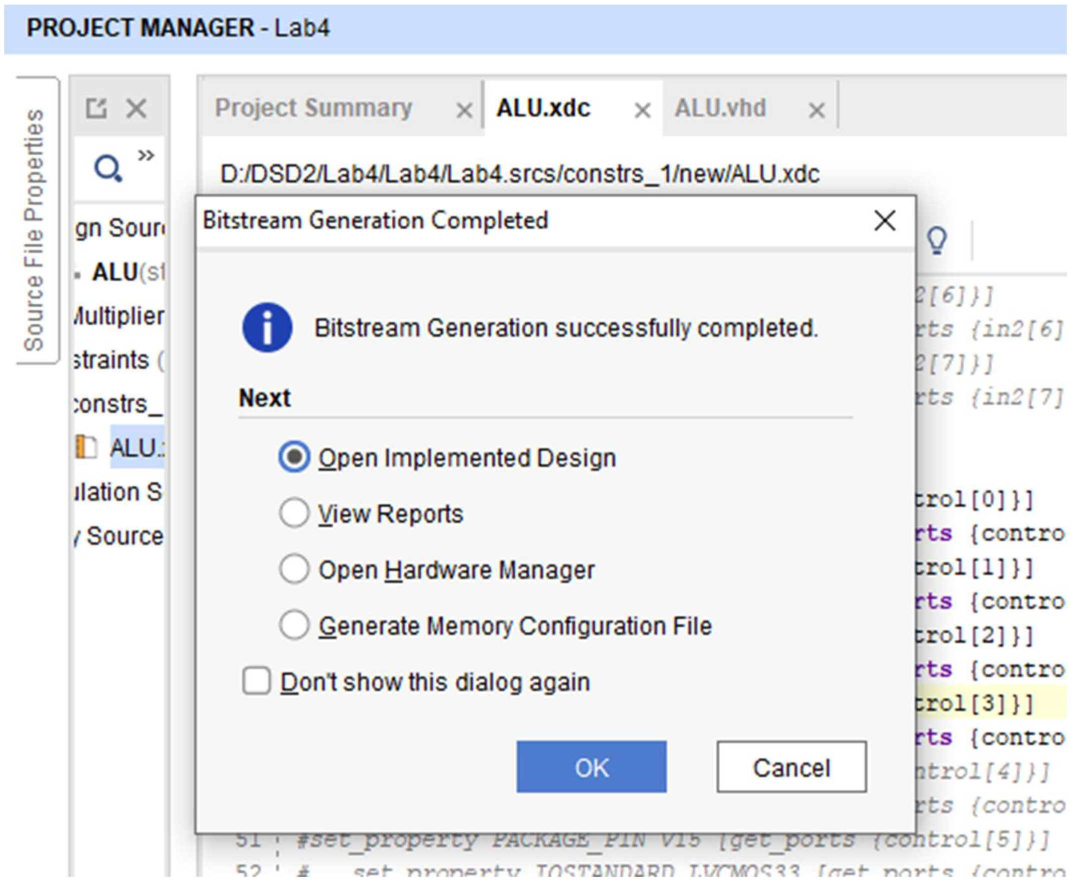


Figure 9: Complete Execute Behavioral Simulation

The Behavioral Simulation begins by testing sets of two in each of the R-type instructions, then the I-type instructions, then the final two tests are on the Multiplication operation. Each of the outputs had their results compared to those declared by the test bench, showing that each of the outputs matched the test bench outputs exactly showing that there were no errors in the overall design on the Execute stage.

In order to test if the ALU would be able to be implemented onto a FPGA the bitstream had to be generated. The message for this is shown in figure 10.


PROJECT MANAGER - Lab4

Project Summary | ALU.xdc | ALU.vhd

D:/DSD2/Lab4/Lab4/Lab4.srcs/constrs_1/new/ALU.xdc

Bitstream Generation Completed

Bitstream Generation successfully completed.

Next

[✓ Open Implemented Design]
View Reports
Open Hardware Manager
Generate Memory Configuration File

Don't show this dialog again

OK | Cancel

Figure 10: Bitstream Generation message.

The Bitstream was successfully generated for the ALU as shown in figure 10. This means that the design was synthesized successfully into a form that could be directly inputted to an FPGA to do physical testing.

Conclusion

The Execute Stage created was implemented successfully to complete the functionality to perform the provided instructions with the given inputs, outputting the correct values for each set of inputs. Each of these operations within the stages were tested in the Xilinx architecture through the Behavioral, Post-synthesis and Post-Implementation simulations with the test benches designed for both stages. The simulations showed that both stages were designed correctly within the Multiplier, ALU, and the Execute Stage. There were minor issues when getting the correct multiplier outputs, but after closer inspection a there was an overlapping assignment in one of the generation methods within the multiplier that was removed, after which it compiled correctly and there were no other major issues. This exercise was successful in both the design of the Execute stage, as well as the principles necessary to create it including the generate statements and multiple structural models within each other. It was also successful in providing more practice in the creation of functional test benches to test the created code.