CMPE 260 Laboratory Exercise 1

Introduction to Vivado & Simple ALU

Aden Crimmins
Performed: January 26, 2021
Submitted: February 9, 2021

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

Lecture Section: 2
Professor: Mr. Cliver

By submitting this report, you attest that you neither have given nor have received any assistance (including writing, collecting data, plotting figures, tables or graphs, or using previous student reports as a reference), and you further acknowledge that giving or receiving such assistance will result in a failing grade for this course.

Your Signature: *Aden Crimmins*

Abstract

The purpose of this exercise was to create, simulate, synthesize, implement, and load a VHDL file using the Xilinx Design Suite. The project was created for a 4-bit ALU which supports the Logical OR, AND, XOR, Shift Left, Shift Right, and the Arithmetic Shift Right. This was then increased to a width of 32 bits as the components were declared with a generic bit width, then instantiated with a width of 32 bits. This was tested using a testbench designed to confirm the successful function of each operation including edge cases. This lab was successful in showing how to utilize the Xilinx software to create, simulate, and test hardware code.

Design Methodology

The design was created using the Xilinx Vivado Design Suite and implemented on the Basys3 FPGA. This logic was created for the logical gates NOT, OR, AND, XOR, Shift Left, Shift Right and the Arithmetic Shift Right. 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 below in Figure 1.

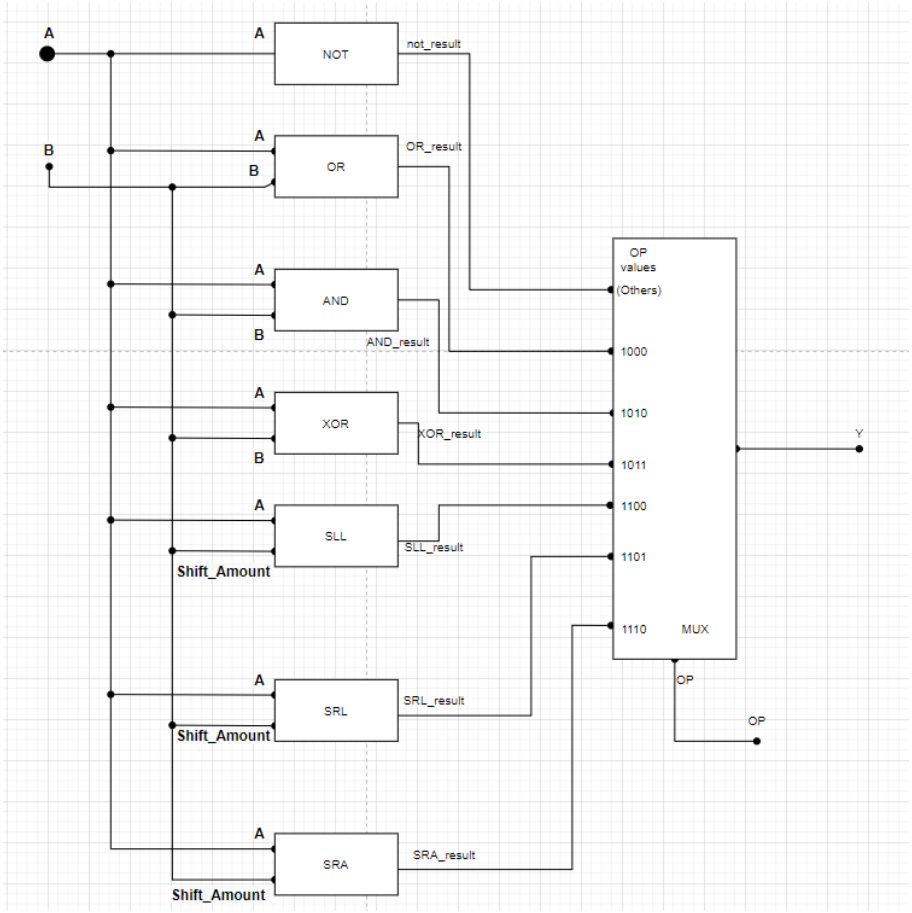

Figure 1: Block Diagram for the ALU

Figure 1 shows that the two inputs A and B are used as standard logic inputs for the NOT gate through the XOR gate, but for each of the shift operations the B input acts as the shift amount while A remains as a standard logic input. Each of the outputs goes into a MUX and the desired operation is selected using the OP input resulting in the chosen operation being routed to the output.

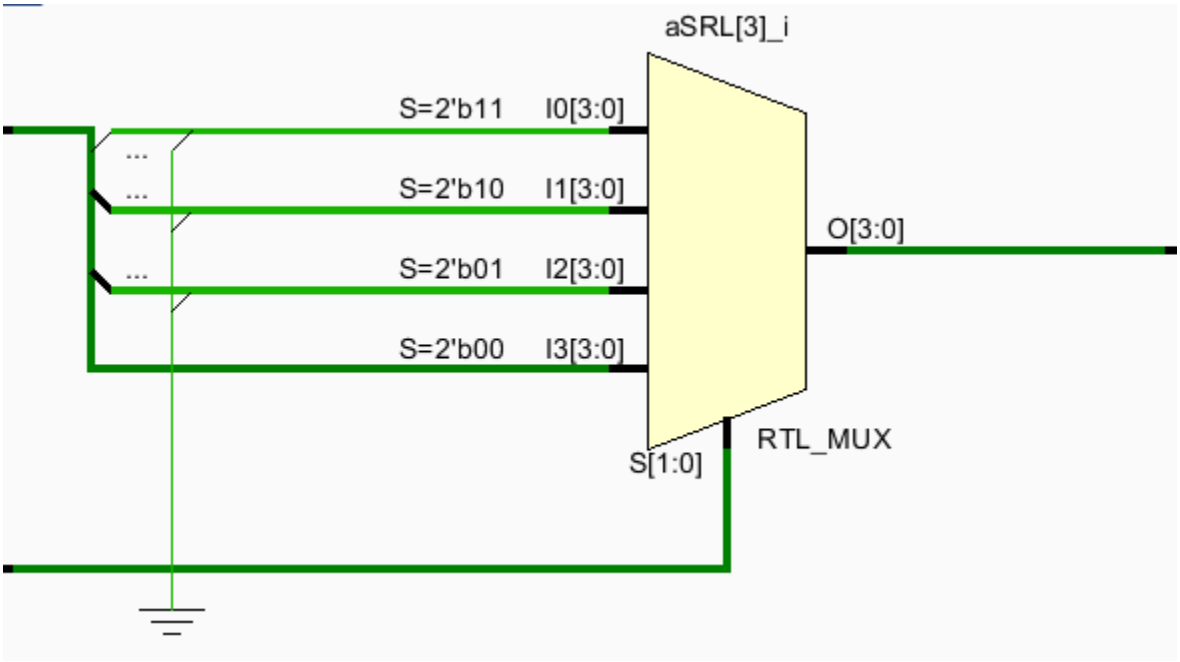

Figure 2: Logical Shift Right RTL Schematic

The Logical shift right as seen in Figure 2 was created by setting the initial shift values to be moved to the largest bit down to 0 from the largest bit down to the shift amount. This moves the data to the right, then the left bits from the largest bit down to the array size minus the shift amount are filled with the logic value 0. The arithmetic shift was created in the same way, but in order for it to be arithmetic, the fill value was determined by the most significant bit of the input, then that value was put into the spots that were shifted from to fill them in.

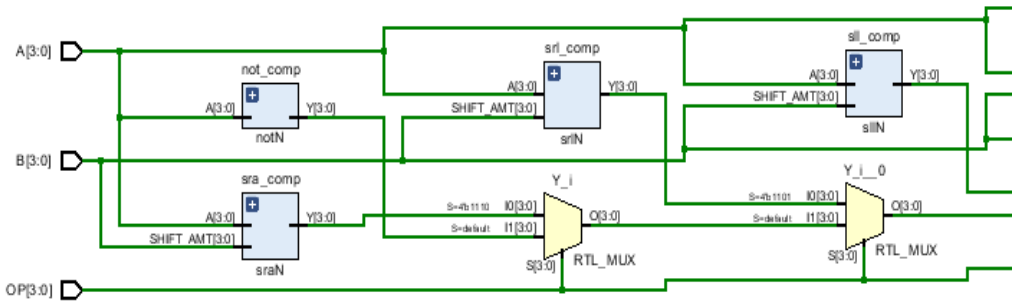

Figure 3: RTL Schematic for the ALU(1/2)

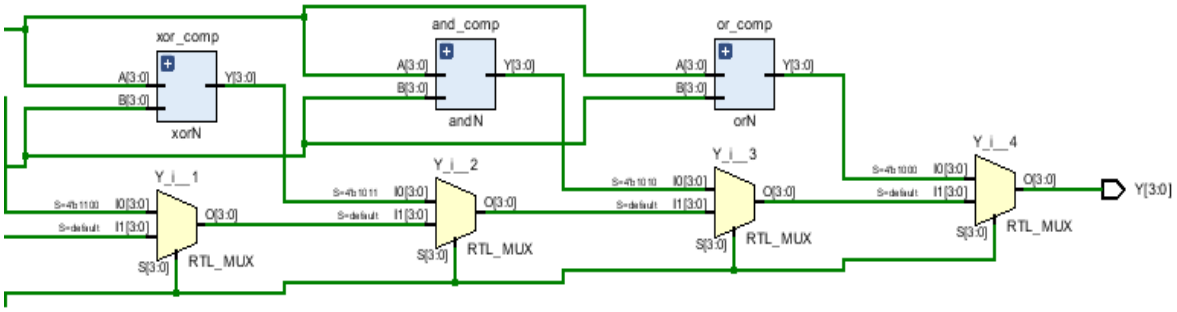

Figure 4: RTL Schematic for the ALU(2/2)

The schematic shown in figures 3 and 4 represent the full RTL schematic of the ALU. Each of the blue boxes represent the logic design of one of the operations while the MUX's below represent the selection of the desired operation through the OP input.

Results and Analysis

The testbenches created for this lab tested a variety of cases of A and B input array values, as well as different operations to ensure full functionality of the ALU and its components.

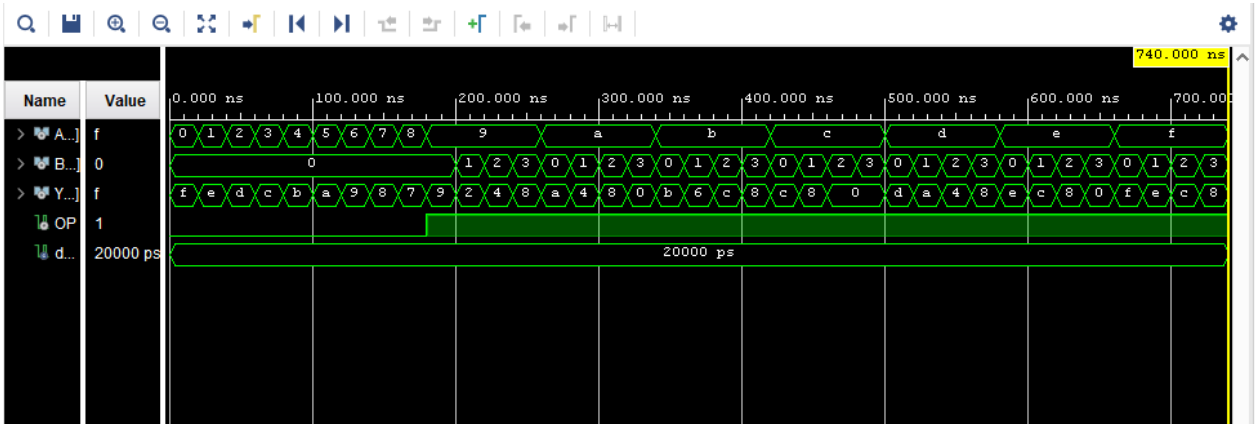

Figure 5: 4-bit Behavioral Simulation

The first test bench created tested each variation of OP with different value sets for A and B. Figure 1 shows the proper Operations were completed for the NOT circuit when OP is set to '0', and the correct Shift Left Operations when OP is '1'. Each output shows the correct function of the NOT and SLL operations with the given inputs A and B.

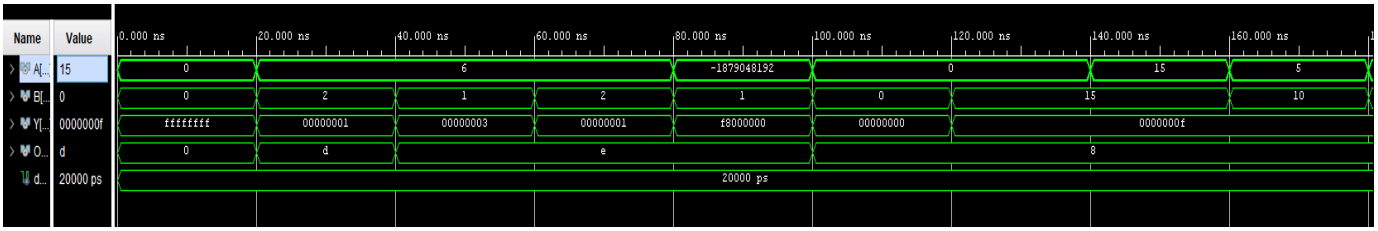

Figure 6: 32-bit Behavioral Simulation (1)

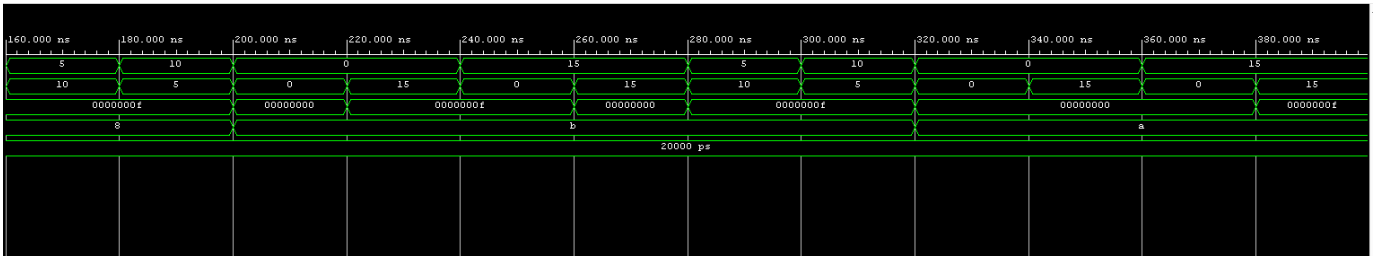

Figure 7: 32-bit Behavioral Simulation(2)

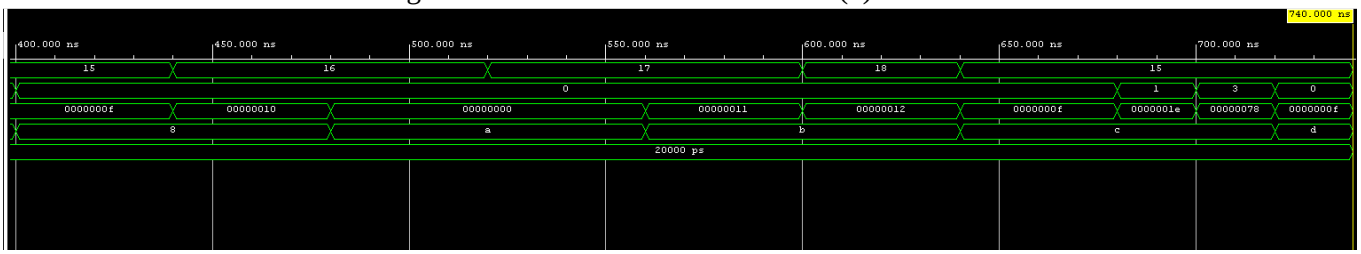

Figure 8: 32-bit Behavioral Simulation(3)

The second behavioral Simulation tested the 32-bit ALU with specific edge cases up to 400ns, then extra cases from 400-750ns. It started at 20ns by testing the function of the shift right logical, shifting the value of 6 by 2. The correct output value of 1 shows that the correct operation was performed in this test case. Shift right arithmetic is tested next with three cases. The first two cases show the value of 6 being shifted by 1 then 2, with correct outputs of 3 and 1. The final test for this operation shifted a value of 0xF0000000 by 1, with the output showing 0xF8000000 which is the correct value for the requested operation.

The OR operation tests are shown between 100-200ns, with each combination having an output value of 0xF after the initial output of 0x0. This is the correct result for the operation of OR performed on the values of (0x0+0x0, 0x0+0xF, 0xF+0x0, 0xF+0xF, 0x5+0xA, 0xA+0x5). The same tests were performed on the XOR function from 200-320ns with each output showing the correct values for the given input. from 320-400 ns the AND operation was tested with values of (0x0·0x0, 0x0·0xF, 0xF·0x0, 0xF·0xF). Each of the results were 0x0 except for the 0xF·0xF operation which resulted in 0xF.

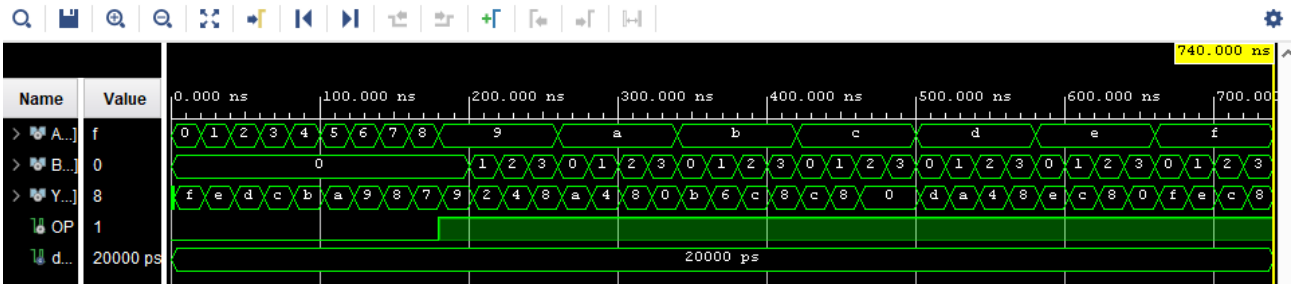

Figure 9: Post-Synthesis Timing Simulation

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.

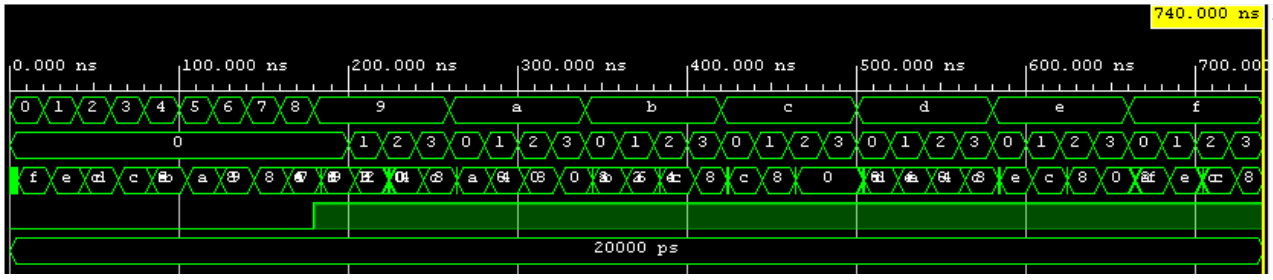

Figure 10: 4-Bit Post-Implementation Timing Simulation

The Post-Implementation timing simulation test 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 only two relatively simple operations to complete.

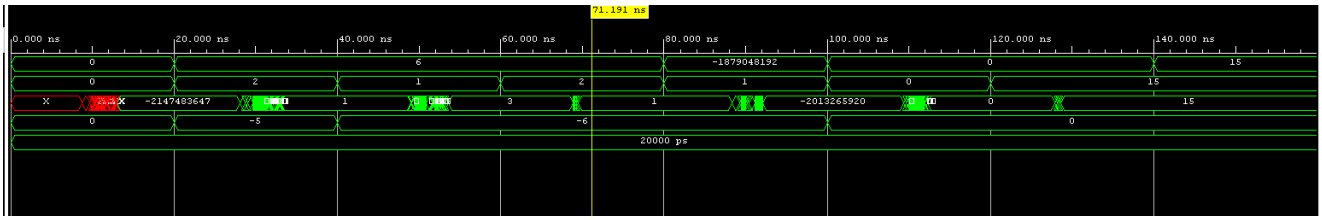

Figure 11: 32-Bit Post-Implementation Timing Simulation(1)

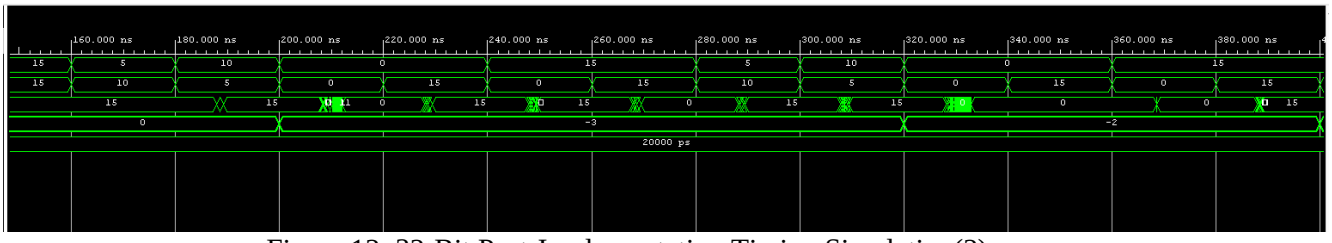

Figure 12: 32-Bit Post-Implementation Timing Simulation(2)

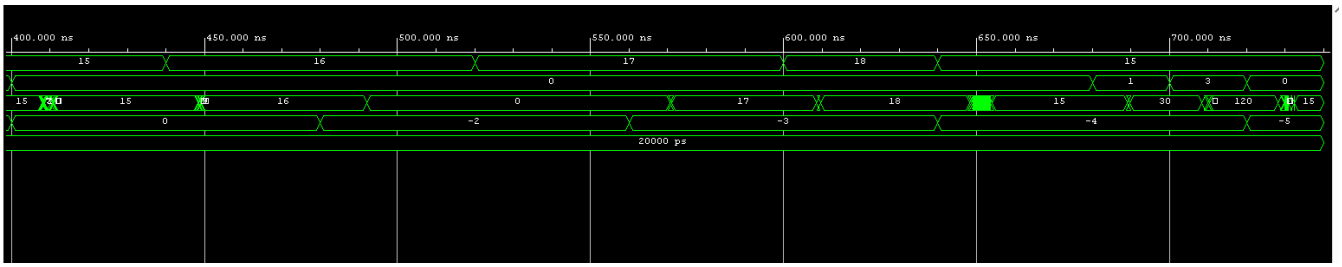

Figure 13: 32-Bit Post-Implementation Timing Simulation(3)

The 32-bit Post-Implementation timing simulation shown in figures 11,12, and 13 tested the implemented model as it would behave on the actual hardware. This test has greater delay shown in the waveforms as there are more complex operations in the 32-bit version with more gates adding to the total delay time. If delay is taken into account then the output of this waveform is the same as tha of the behavioral simulation proving that there aren't any obvious design flaws with the circuit.

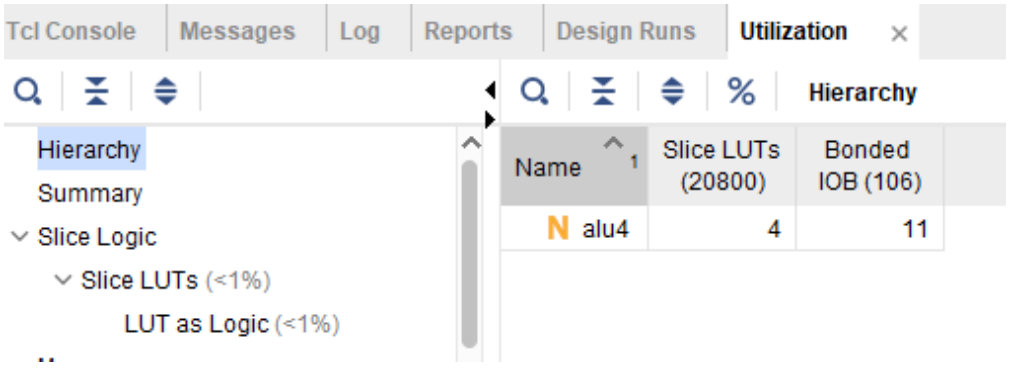

Figure 14: Synthesis Utilization Report

The Synthesis Utilization Report shown in Figure 14 shows the number of slice LUTs utilized by the design. These are Look up tables create with small asynchronous SRAM and are used to implement combinational logic. The more LUTs used to implement the circuit the more resource intensive the design is.

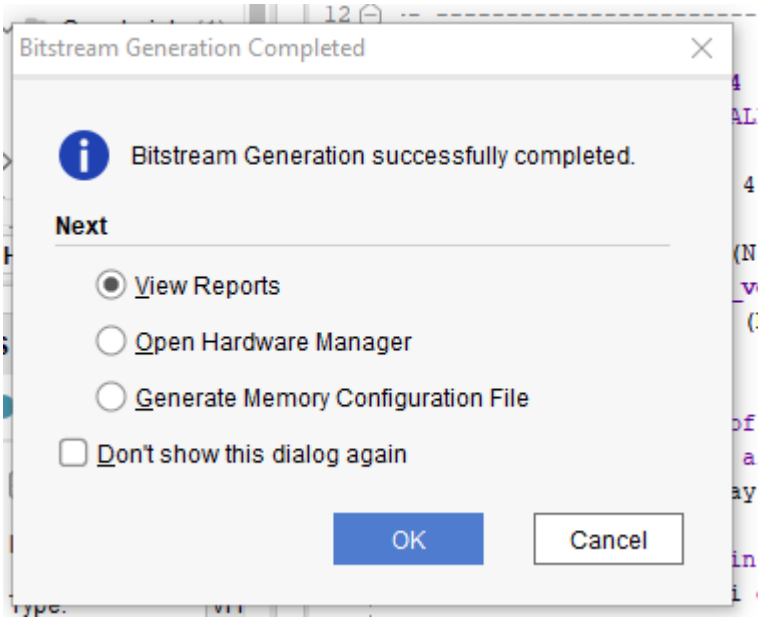

Figure 15: Bitstream Generation Successful

The Bitstream generation message in Figure 15 shows the successful generation of the Bitstream file, which is the file containing the necessary Information to program the FPGA with the created design. If there was an available FPGA to load the bitstream onto then the FPGA would be able to perform the operations designed when given the inputs of A, B, and OP, with the output Y would be produced in accordance with the simulated designs.

Conclusion

The ALU created was implemented successfully to complete the NOT, OR, AND, XOR, SLL, SRL, and SRA operations. Each of these operations within the ALU were tested in the Xilinx architecture through the Behavioral, Post-synthesis and Post-Implementation simulations when tested with the designed test bench. The circuits were inspected through the different design schematics to ensure that they were built and connected correctly. Both the schematics and simulations showed that the ALU was designed correctly both in the 4-bit implementation and the 32-bit implementation. A bitstream was also successfully generated which, if a physical board was present, would have allowed the design to be imported into an FPGA and tested in practice. There were minor issues when converting to the 32-bit implementation of the ALU, but after closer inspection the bit values were corrected and it compiled correctly and there were no other major issues.