**CMPE-260 Laboratory Exercise 03**

**ALU Design**

By submitting this report, I attest that its contents are wholly my individual writing about this exercise and that they reflect the submitted code. I further acknowledge that permitted collaboration for this exercise consists only of discussions of concepts with course staff and fellow students; however, other than code provided by the instructor for this exercise, all code was developed by me.

John Judge

John Judge

Performed 3/3/16

Submitted 3/31/16

Lab Section 04

Instructor: Dr. Lou Beato

TAs: Nick Barlow

Sergio Martin

Lecture Section 01

Professor: Dr. Reza Azarderakhsh

**Table of Contents**

1. Abstract …………………………………………………………………………...1
2. Design Methodology………………………………………………………………1
3. Results……………………………………………………………………………..3
4. Conclusion………………………………………………………………………...4
5. Appendix…………………………………………………………………………..5

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Abstract:**

The purpose of this exercise was to design an Arithmetic Logic, or ALU, in VHDL using sub-blocks. In this implementation of the exercise five sub-blocks were needed: the 16-Bit Adder/Subtractor, the Logical Unit, the 8-Bit Multiplier, the 16-Bit Variable Shifter, and a generic 7-1 Multiplexer. Once created in VHDL, the design was then thoroughly tested using multiple test benches and ModelSim. The waveforms generated by ModelSim verified the success of the implementation and the design was then transferred to the NEXYS 3 FPGA. The hardware was then tested to once again verify the success of the implementation.

**Design Methodology:**

*The ALU (Top Level Module)*

During this exercise an ALU would be created in VHDL. The ALU, or Arithmetic Logic Unit, had two 16-bit inputs as well as a 4-bit control unit. The ALU had a single 16-bit output. The unit would preform one of ten different operations when a valid control signal was detected. The different operations and their corresponding operations can be seen in Table (1.0). The ALU was designed so that if any other control signal other than those specified in Table (1.0) the output would result in a 16-bit output of all zeros.

|  |  |  |
| --- | --- | --- |
| Description | Instruction | Control |
| Addition | ADD | 0100 |
| Subtraction | SUB | 0101 |
| Multiplication | MUL | 0110 |
| Bit-Wise OR | OR | 1000 |
| Bit-Wise NOT | NOT | 1001 |
| Bit-Wise AND | AND | 1010 |
| Bit-Wise XOR | XOR | 1011 |
| Logical Left Shift | LLS | 1100 |
| Logical Right Shift | LRS | 1101 |
| Arithmetic Right Shift | ARS | 1110 |

Table (1.0): ALU Operations

In order to achieve this functionality the ALU needed several sub blocks. A capture of the design’s RTL schematic showing the various sub blocks can be seen in Figure (1.0). The sub blocks were a 16-Bit Adder/Subtractor, 8-Bit Multiplier, 16-Bit Shifter, a 16-Bit Logical Unit, and a generic 7 to 1 multiplexer used to select which sub block’s output the ALU would output

*The 16-Bit Adder/Subtractor*

In order to handle the addition and subtraction function of the ALU a 16-bit ripple carry adder (using 16 full adders) with a subtraction operation was designed using a structural model implementation. The Adder/Subtractor had two 16-bit inputs, a carry-in signal, and a single 16-bit output. The schematic of the Adder/Subtractor can be seen in Figure (1.1). In order to handle both the addition and subtraction functionalities XOR gates were used. A carry-in signal of 0 drove the addition function and a carry-in of 1 drove the subtraction function. By XORing each bit of the second input signal by the carry-in and then mapping the temporary signal leaving the XOR gates to one of the inputs of the full adders the addition and subtraction functionality could be handled by a ripple carry adder. The first 16-bit input signal was also mapped to the inputs of the full adders with each bit corresponding to the same bit of second input signal. By using two “FOR GENERATE” statements, one for the XOR gates and one for the full adders, the design created and each of the full adders and XOR gates were connected correctly. The output of the adder/subtractor was the sum calculated by each full adder. The carry out of the last full adder was neglected, as it was not necessary to the overall functionality of the ALU.

*The 8-Bit Multiplier*

By far the most challenging part of the design, the 8-bit Multiplier required fifty-six full adders and sixty-four AND gates. Designed as an Carry-save Array Multiplier the Multiplier had two 8-bit inputs and a 16-bit output. In order to handle the signals between each of the components three 2-D arrays were used to store the temporary signals. The 2-D arrays were used for the signals leaving the AND gates, the carry-out/carry-ins of full adders, and the sums calculated by the full-adders.

A carry-save multiplier works by mimicking binary multiplication. Consider a slanted table of full adders (the result of which would be shaped as a rhombus). The carry-in and carry-out connection between full adders are diagonal as opposed to the horizontal connection used in the Adder/Subtractor. This creates the Wallace Tree structure needed for division. Although the RTL schematic did not generate the Multiplier in exactly this way the correct schematic can be seen in Figure (1.2). Then the sum of each full adder was connected to the next row of full adder vertically. This is true for all of the full adders except the adders along the edges of the table and the full adders in each corner. The adders along the top edge of the table have inputs and carry-ins coming from the AND gates. The bottom row is essentially a ripple-carry adder. In this row the carry in of each full adder was mapped to the carry out of the preceding full adder, when considering the row from right to left. This resulted in the structure for a multiplication circuit.

Initially the Multiplier was designed using “FOR GENERATE” statements to handle the generation of full adders in each horizontal line but this proved to be extremely difficult. Initially, the Multiplier did not have the correct interconnections between its parts to function correctly. As a result, the multiplier was redesigned with each component hardcoded. This proved to be more successful and the implementation was verified to be correct later in the exercise.

*The 16-Bit Shifter*

The shifter had three inputs: a 16-bit data input, a 4-bit shift amount, and a 2-bit shift type. It had only one 16-bit output. The 16-bit shifter required three independent sub blocks to handle the three shifting operations. The shift type signal was used to select which sub block would operate. Each of the shifter’s sub blocks were written using behavioral modeling. In order to handle the logical shifters the input signal was reduced based on the shift amount signal and the then the resulting vector was concatenated with zero’s until the signal 16-bits long. This vector was then assigned to the outputs. In order to achieve the arithmetic right shift operation the same method was used except the shortened vector was concatenated with one’s in the most significant bit until the desired 16-bit output had been reached.

*The 16-Bit Logical Unit*

The logical unit, which had two 16-bit inputs, was written using behavioral modeling. In order to achieve the AND, OR, NOT, and XOR functions VHDL’s built in bit-wise Boolean operations were used. When synthesized, the program would create 16 logic gates for each operation. The Logical unit had four 16-bit outputs, one for each operation, these four outputs were mapped to the multiplexer.

*The Multiplexer*

A generic 16-bit 7-1 multiplexer was used to select which signal would be output by the ALU. The Multiplexer was written using dataflow architecture. Using the ALU’s control signal as a select the correct output could be chosen. If a non-valid control signal was used the Multiplexer would output all zero’s. This essentially meant that the multiplexer was a 16-1 multiplexer with nine inputs connected to ground.

**Results:**

As the sub blocks of the ALU were created they were tested using smaller test benches to test their individual operations. Using trial and error each of the individual components were corrected until they preformed the correct operations. The once the top-level model was implemented the ALU could be tested. Using a test bench the ALU was tested to ensure the design preformed the correct operations on the input vectors.

*The Design of the Test Bench*

The test bench created to check all of the operations in one file. The test bench changed the control signal and the input vectors so that multiple test cases were generated for each operation. The test cases were chosen to test specific cases that were likely to cause incorrect results. For instance, the test cases for the Adder/Subtractor contained cases that would result in negative numbers and overflow. The multiplier was tested using cases that would result in overflow and a multiplication by zero.

*Results of the Test Bench*

As seen in Figure (1.3) a waveform was generated by ModelSim to describe the result of the test bench. The results of the test bench were represented in hexadecimal for ease of reading. The results of the test bench matched the expected outputs and we’re verified to be correct by hand calculations and a programmer’s calculator.

Following the initial test bench, a post route simulation took place in order to see the expected delays that would occur in hardware. Using ModelSim and the same test bench the post route simulation took place. The resulting waveform can be seen in Figure (1.4). The delays labeled to highlight the difference in time to be expected in a hardware impliminataion.

*FPGA Implementation*

Following the post route simulation the design was routed to the NEXYS 3 FPGA and was tested using the board’s switches to represent the input vectors, the control vector, and the LEDs were used to represent the output. Since the FPGA only had eight switches the control signal used four and the two inputs we’re truncated to only 2-bits. The implementation was then tested using the same test cases as the test bench and some others. The resulting lit and unlit LEDs matched the expected results when the inputs were truncated to 2-bits each. The design could then be verified to be functioning and the design summary was then consulted.

*Design Summary*

The Design Summary tab of the project was then consulted to check the occupied number of slices, flip flops, and LUTs. Based on the Design summary the project used a total of 201 LUTs and Flip Flops as well as 90 slices

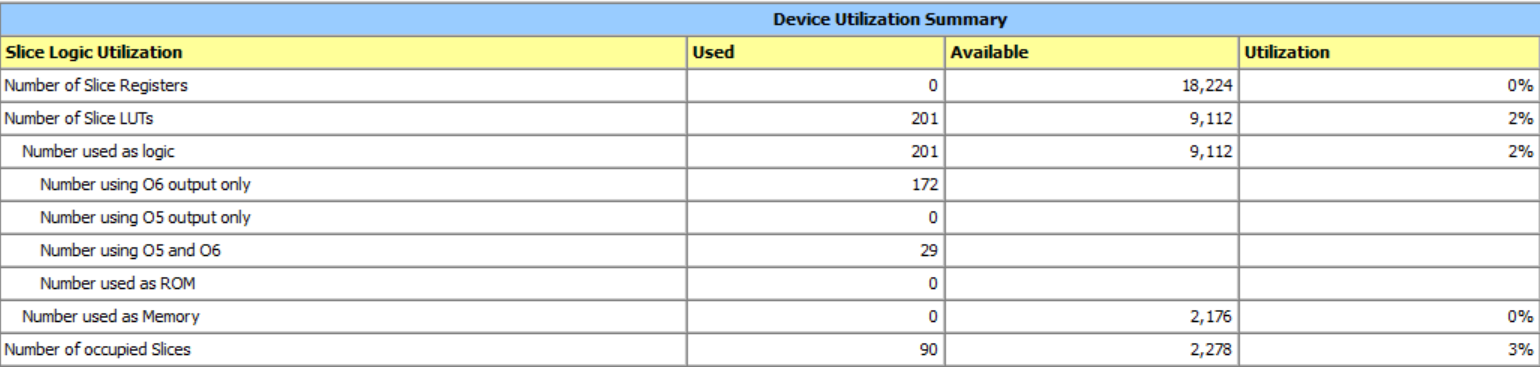
**

Figure (1.5): The Design Summary

**Conclusion:**

There were some lessons to take from doing the exercise. The most major of which was the use of “FOR GENERATE” statements and the interconnections between components in larger designs. The problems encountered in the multiplier resulted from incorrectly using the “FOR GENERATE” statement and incorrectly mapping components together with temporary signals. These techniques were used correctly in the Adder/Subtractor but proved to be difficult when there was a large amount of components. The exercise also used 2-D arrays at multiple points to hold temporary signals. This technique had not been used in previous exercises and was vital for the Multiplier. Finally this exercise also provided a successful example of FPGA use. The FPGA had not been used in the previous exercise and during the first exercise steps for implementing a design on the FPGA had been given.

Overall, the exercise was conducted successfully, with the ALU resulting in the expected outputs for the given inputs. In the future this exercise can be used as an example of how to correctly map a large series of components together and FPGA use.

**Appendix:**

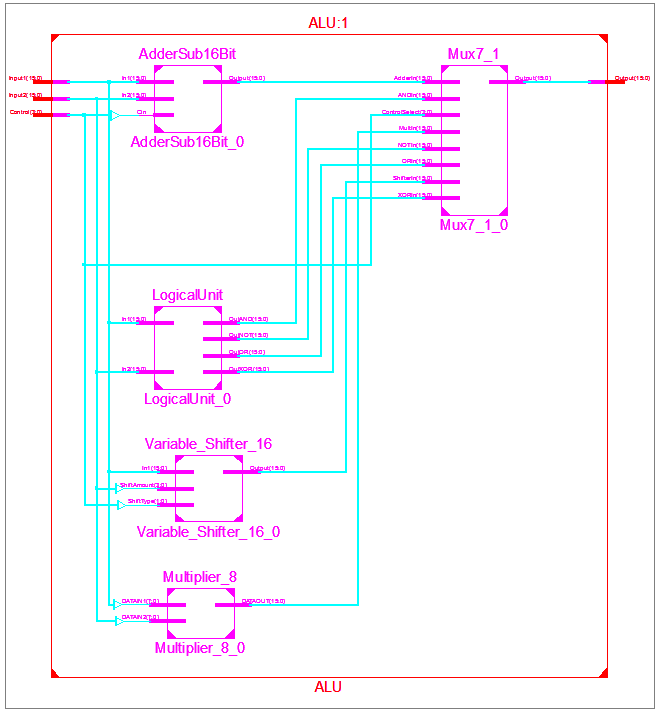
****

Figure (1.0): The ALU with Sub Blocks

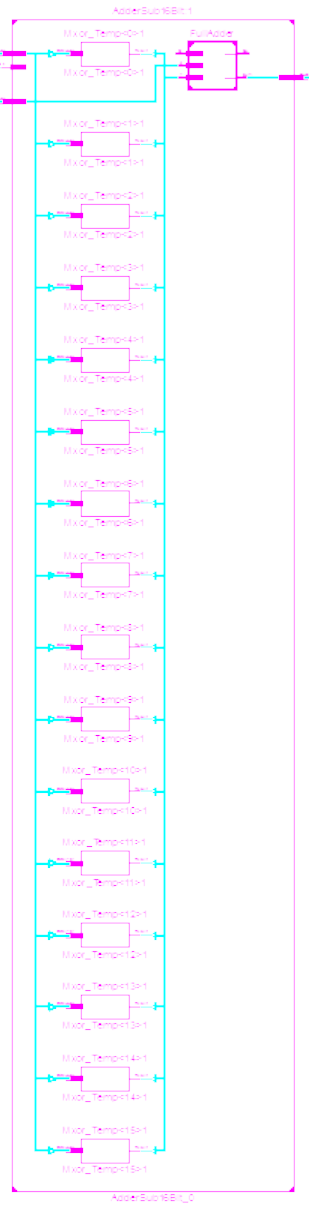


Figure (1.1): The Adder/Subtractor

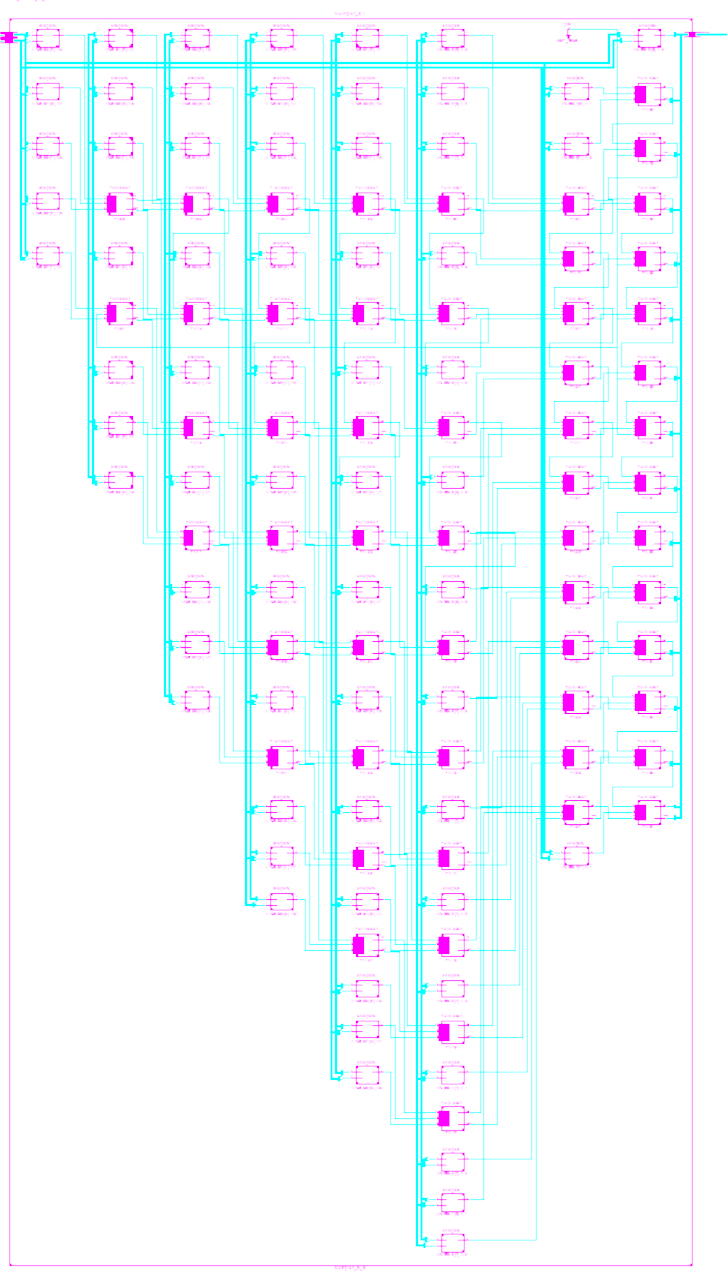


Figure (1.2): The Multiplier

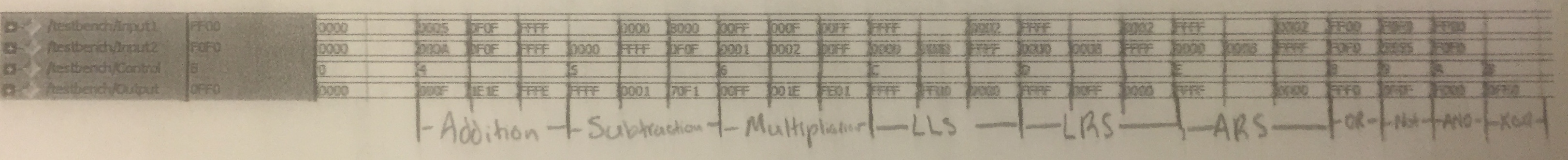


Figure (1.3): Test Bench Results

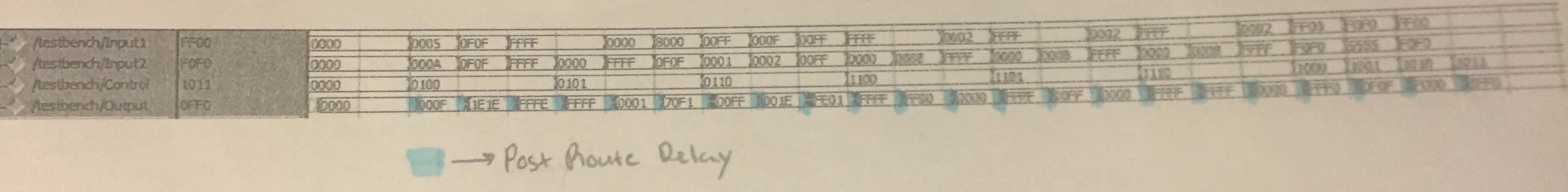


Figure (1.4): The Post Route Simulation Results

