Katy Kahla
James Kress
4 April 2012

Part 3 – ALU

Design

This part of the project asked us to create an ALU that implements all of the arithmetic and logic functions of BSUCA12. This included implementing ADD, SUB, NAND, XOR, SHL, and SHRA. The ALU implemented the adder from part 2 to implement addition and subtraction and can detect overflow from these two operations. The outputs from the ALU operation are aluOutput, isEqual, isGreaterThan, and isOverflow.

We began this portion of the project by creating the ALU module. From there, we created two claAdder instances, one to perform addition and one to perform subtraction of the input. At the time when the ALU is called, it will calculate both of these values but only output the correct value.

After creating the claAdder instances, we defined the cases that the ALU must perform. This was implemented using a simple case statement in the final version though originally it was implemented using if else statements. The case statement would simply match the input at which point the ALU would implement the correct operation and assign it to the output. The most involved portion to implement was the subtraction case which had to assign the isEqual and isGreaterThan outputs. It was rather simple to implement as the cases where given in the project documentation.

One error that we ran into while implementing the case statements was implementing the NAND statement as we discovered that ~& was not a recognized symbol in Xilinx. Instead, we had to AND the inputs and then inverse them. We also found that we needed to use a signed register in order to implement isGreaterThan as on edge cases we were getting invalid data and for SHRA since without a signed register we could not shift in 1's.

We had a challenging time making sure that our overflow detection was working properly. While determining how overflow should be detected, we found ourselves confused on whether the input numbers were signed or unsigned. We found that we were getting FAIL on addition cases such as `0xFFFF + 0xFFFF` as the overflow bits were not matching though this addition should cause an overflow. We researched online how overflow should be detected and ultimately we went to Dr. Rafla's office hours to receive help. He informed us that the numbers should be unsigned and that truly the only case we would have overflow was in addition as we could not subtract two negative numbers from each other. Based on this, we fixed our overflow by simply using the carryOut value from the claAdder for addition though in the original test bench it will FAIL on overflow, as the current test bench never expects an overflow.

Another error that we encountered was the implementation of our always block. Originally we were using certain inputs to test when to implement the always block but this would cause test cases to fail. We found that you had to use always(*) in order to get all test cases to work as this would detect any change.

Performance

We tested the performance of each operation that the ALU could perform. Through testing edge cases and normal cases, we found that the ADD operation was optimized at 65 nanoseconds. Originally it was optimized at 65 nanoseconds when solely testing the claAdder from part 2 and we are unsure as to why the time increased. Through testing the SUB operation, we found it was optimized at 70 nanoseconds.

We also test the NAND, XOR, SHL, and SHRA operation's performance. SHRA, SHL, and XOR was optimized at 10 nanoseconds while NAND was optimized at 15 nanoseconds.

Schematic

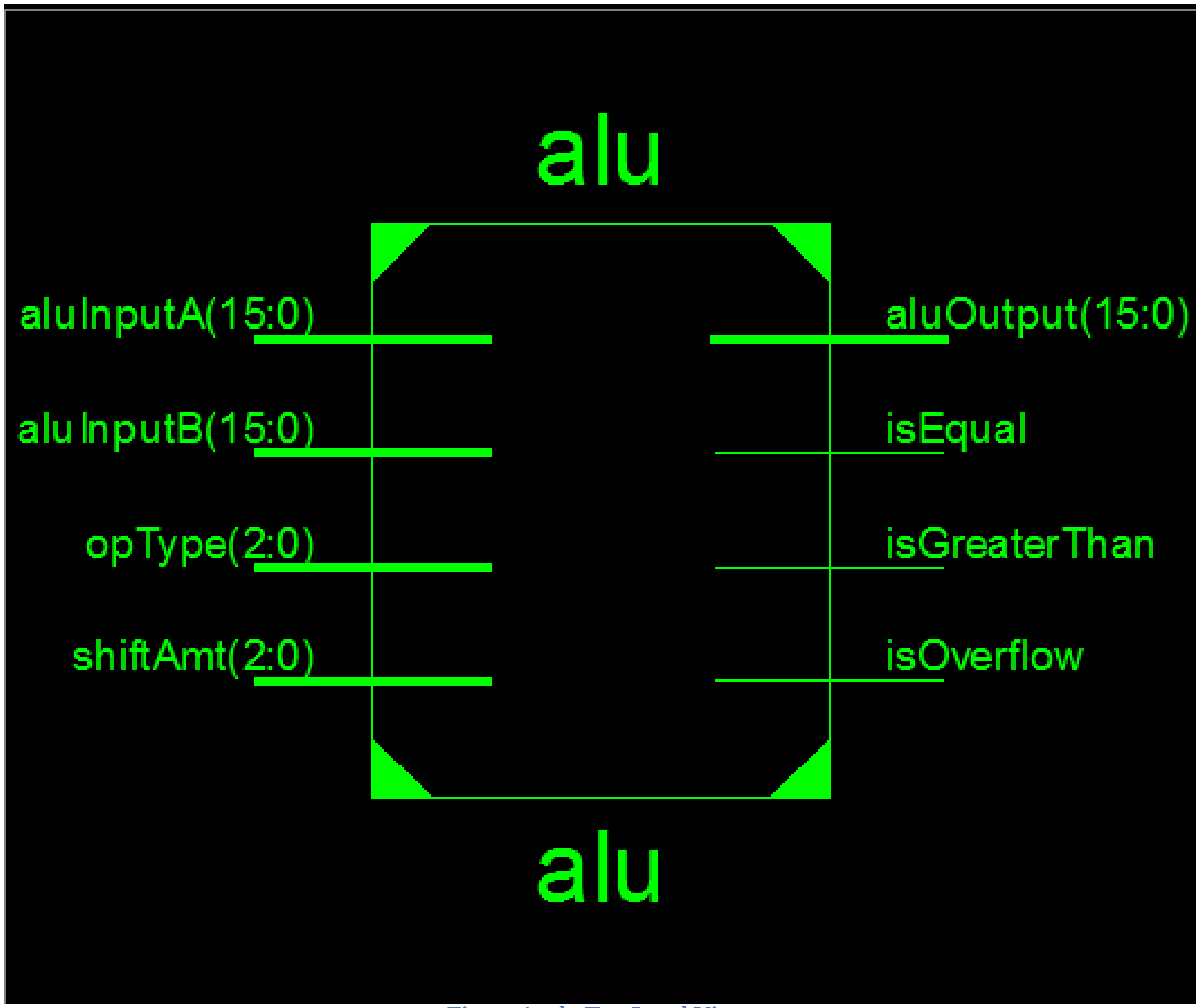

Figure 1: alu Top Level View

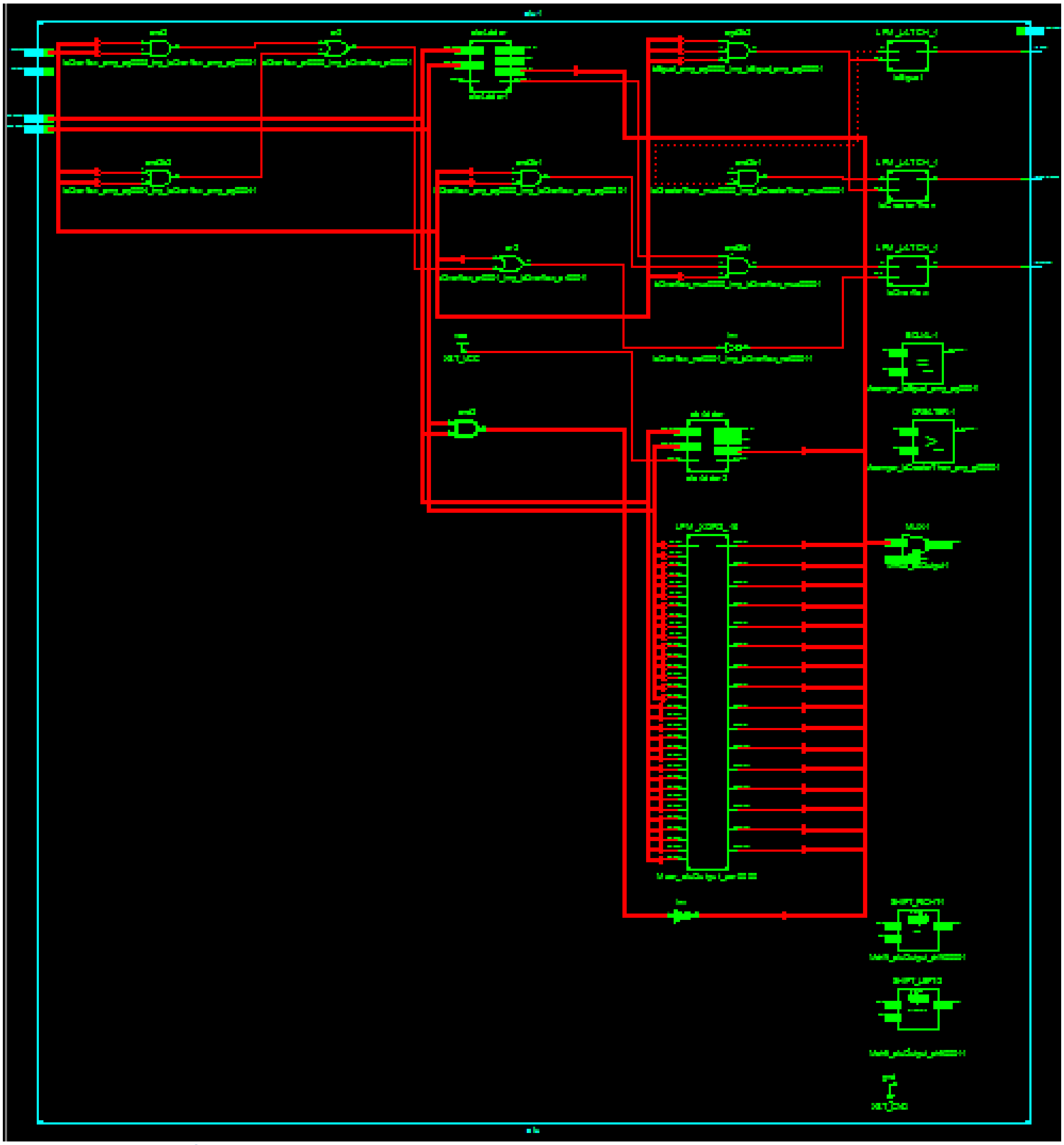
Figure 2: Pushed in view of alu

Testing

We tested out project extensively to ensure that all of the program requirements were met. To start, we tested using the simple testing script that had been provided with the program. Once we passed all of the tests it contained, we decided to add to the given tests to come up with a file that would test for all of the required functionality.

For each ALU operation, we tested base cases and as many edge cases as we could determine. Each section is delineated in the output file by section headers. For the ADD and SUB operations, we modified the test bench to display what the overflow values were to make sure that overflow was being calculated correctly. For the test bench that was given at the beginning of the program, we will fail ADD cases on overflow detection though our ALU is correctly detecting overflow.

We found an error while testing the SHRA where original code of simply doing `aluInputA >>> shift` would not shift in a 1 properly and we were able to fix it through our testing.