**Homework3 (take-home Quiz)**

**SystemVerilog HDL combinational logic modeling and simulation**

Name: \_Alex Blake\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Since this is a take-home quiz:**

**please work alone – do not collaborate!**

|  |  |  |  |
| --- | --- | --- | --- |
| **Exercise** | **Course outcome** | | **Grade** |
| Homework3 | | 2.a, 7.b | /30 |

2.a. Define engineering problems from specified needs for digital systems including implementation on FPGAs using HDL programming.

7.b. Employ appropriate learning strategies such as communicating with an expert, using external resources, experimentation, simulation, etc.

**Introduction**

In this homework assignment, students were asked to work on their own to create a module that can do multiple different operations on inputs. Then, they must create a testbench to thoroughly test the module for all conditions to verify it works correctly with assert statements. To accomplish this, students are required to understand. The assignment will test all previously learned material from the course including always blocks, begin/end statements, if/else statements, switch cases, loops, and immediate assertions.

**Methods**

The first step to completing this assignment is to create the digital logic module. The module needs to accomplish the following (Figure 1):

|  |  |
| --- | --- |
| Inputs | a[3:0], b[3:0], s[1:0] |
| Outputs | y[3:0] |
| Function | |  |  | | --- | --- | | s | function | | 0 | y = a + b, unsigned add | | 1 | y = a << b[1:0], a shifted left by the amount specified by the 2 LSB’s of b | | 2 | y = a AND b | | 3 | y = 1, ignore inputs a and b | |

**Figure 1: Logical Module Input and Output Table**

To set this up, a generic module is created, and named simpleALU. The input and output logic were labeled to match the table (Figure 1) to avoid any confusion and be consistent. A simple switch case is used and each case is selected by s, and each s value selected the corresponding assign to y within the always\_comb block. This switch case within the always\_comb produces all the necessary logic to match Figure 1.

The second task is to create the test bench. This takes a little more effort because the test bench is required to test every single possible input and verify the output matches. In other words, different inputs are required to be tested, so loops are an obvious choice here.

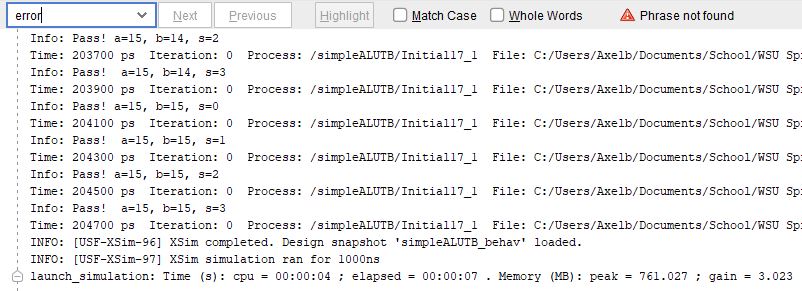
To test every single possible a, b, and s value, nested loops were used: one for every possible value of a, and for that, every possible value of b, and for both of those, every value of s. In the loop, a switch case similar to the simpleALU module is used, but instead of assigning values to y, assert statements are used to ensure the correct outputs match Table 1.

The assert statements are setup to print an “info” type statement when the assert statement is correct but prints an “error” type statement when it fails. In either case, the value of a, b, and s are printed for debugging purposes. There is also a default case with error messages in case s is out of bounds.

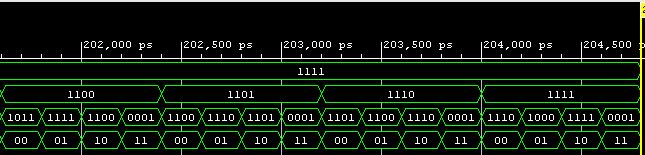
The time is incremented with #1 within the loop (after values increment) and at the end of the loop. This ensures the assert statement is reading the value of y between changes and not during. This means there is no ambiguity like there would be trying to read on a change of any inputs.

**Results**

The code works and does not print a fail, which can be seen from the Figure 2. All the messages (most you cannot be seen because there are so many) are info statements and can be verified by searching and finding no results for errors within the console. Figure 3 is the output waveform towards the end of the simulation; the total waveform is too lengthy to show in detail across the entire simulation.



**Figure 2: Console Output from Test Bench Shows 0 Errors**

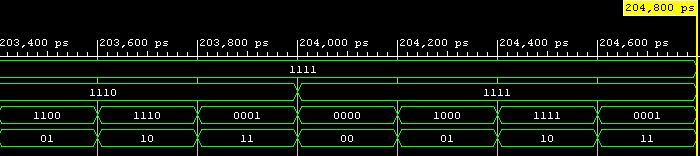
****

**Figure 3: Waveform (a, b, y, s top to bottom)**

Figures 2 and 3 show that the code operated as expected, as well as he waveform ended with a, b, and s being their max values (all 1’s in binary), which is how the loops were expected to work. This part of the lab required some modification of the time domain of the simulation to be manipulated so the entire set of assert statements were executed. The timescale function at the top of the testbench file was changed to 100ps. This fixed the problem by having the time increment function increment by a smaller value of time because the simulation will stop if it hits 1000ns, which it did if this timescale was not changed.

Finally, to examine how well the testbench file catches errors, an error was introduced to the module (brokenALU); when s==0, y=a-b instead of y=a+b. Figure 4 shows the errors caught in the console along with figure 5 showing the last part of the waveform for the broken ALU module similarly to figure 3.



**Figure 4: Console from Broken ALU File Catching 224 Errors**

**Figure 5: Waveform from Broken ALU Module (a, b, y, s top to bottom)**

Just to ensure that the ALU was catching real errors, figures 3 and 5 can be used to compare when s==0, figure 3 has a y value of 1110 while figure 5 shows 0000 for the equivalent a, b values (1111 for both). This concludes the assignment.

**Conclusion**

This homework assignment was a great way to test and practice all the tools learned from the semester so far. This includes instantiating modules to perform digital logic which includes loops, always\_comb statements with begin/end statements, test bench simulations, etc. Combining all of these tools together allows a simple yet functional ALU to accomplish digital logic to match a given truth table requirement.