ELE 206 Lab 2 - Oliver Schwartz (*os4*)

Review Questions

1.6)

a) The result will be different. X is the bitwise AND of A and B, and will therefore evaluate to 6'b000000. However, Y is the logical AND of A and B, and will be 6'b111111.

b) & is the **bitwise** AND operation. This means it AND's two wire vectors bit-by-bit (compares each value in the wire vector and AND's it with the corresponding wire in the other wire vector). However, && is the **logical** AND. This treats any wire vector with a non-zero value as a logical 1, and zero values as a logical 0 (i.e. it would treat both A and B as true).

c) Since line 7 uses logical negation, z will be 6'b000000 (as Verilog will treat A as TRUE in logical negation as it has at least one 1 in it). This is not the desired result. You could fix this without using B by using the bitwise NOT operation (~) which would produce 6'b010101 (which is the desired result).

2.2) What is a port? What is the difference between a module input and output port?

A port is a connection used to send data to the module and receive data from the module. As an example, an AND gate has two input ports and one output port. When listing ports in a module, it is best practice to put input ports first. Input ports can only even be of type *wire*, and are read-only. However, output ports (which are wires by default) can also be made into *reg* types by specifying this explicitly.

2.4) How do you reference another file in Verilog? Is instantiating a module from another file different from instantiating a module from within the same file?

To reference another file (i.e. if we need another module in that file), we can using an *include* statement like so: **`include "path/to/file/filename.v"** with the ` as the preprocessor directive. Once you have done this include statement, instantiating a module from another file is the same as instantiating a module from within the same file.

2.5)

a) A programmer could abstract this design into modules by declaring a module for each type of gate: i.e. a module for the OR gate. A programmer could also declaring a module for the NOT gate. This would mean implementing the XOR gate would be simple and easy to understand as modularity and abstracting are useful in understanding what a program does.

b) A topmodule is the hierarchical root of the module hierarchy. It is a module that has no ports and which cannot be instantiated. Instead, it is used as the starting point for the full description of a circuit. Most often, a topmodules is used for actually testing a circuit via simulation.

7.1) For software developers, testing is an important part of the development process, but even if bugs are later found in the software, developers are able to release an update or patch. Is this the same case for hardware developers? How could not

rigorously testing a Verilog circuit design be financially costly, especially for hardware developers like Intel?
Software developers are able to fix buggy code and release updates at little cost - updates simply consist of code being sent out and reassembled. On the other hand, hardware developers cannot issue fixes nearly as easily. Because the circuits they simulate are built into physical circuits, 'updates' in the software development sense cannot simply be issued. In such a case, the hardware itself would have to be completely remade from scratch, which would be far more costly for hardware developers (like Intel).

Therefore, it is important that hardware developers rigorously test the modules they create - because 'updates' cannot be made to physically manufactured circuits without remaking the circuits altogether (which would be financially exhausting).

7.2) Read and understand the following Verilog code: *`timescale 10ns/10ps*

a) To specify a delay in Verilog, you simply write a hashtag and a number for where you would like the delay to occur (e.g. ***#5***). This will delay the execution of the *initial* block where the delay is located for the specified amount of time. The number next to the hashtag represents the units of time you wish to delay (in the aforementioned case, the execution will delay by 5 time units).
b) The *timescale* directive indicates the length of time you would like 1 unit of time to be, as well as the precision you would like the precision of the delays to be. In the above example, the time scale is 10 ns, and the precision is 10ps. Therefore, we could use a delay of #1.001 but not #1.0001 (as this would be beyond the precision level we specified). The precision and time scale can only be 1, 10 or 100 in whatever units you use.
c) Delays are useful in Verilog circuits because the results from particular inputs will take a bit of time to propagate through a circuit. Because of this, we want to make sure our inputs have enough time to propagate to the outputs so we are measuring what we want to measure.

7.4) Read and understand the following Verilog code:
reg[3:0] x = 4'b0001;
reg[3:0] y = 4'b1111;
$display("Test Number %d Completed. Result is %h.", x, y);

a) The final text would be: "Test Number 1 Completed. Result is f."
b) To change the code to print test number as an octal value, you would change the %d to %o.

7.5) How do you properly end a simulation? How does the simulation behave if you fail to add this directive?
To properly end a simulation, you must add the **$finish** directive to the end of the simulation's **initial** procedure. If you fail to add this directive, the simulation will not exit.

7.6) Why should you use !== and === when checking equality in testbenches?
When checking equality in test benches, it is important to use the operators !== and === as these check for a bitwise exact match, including both X and Z (unknown and high impedance) values. This is important when checking for errors. If you check the condition

when **out != 1** this statement will evaluate as false if **out** is unknown (which is not what we want to happen, so we check for an exact bitwise match instead). This is because the == and != operators will return Z if the relation they are testing is ambiguous due to Z or X values, causing the error check to behave unexpectedly.

Write Up:

1) **What are the two different types of assignment in Verilog? Which datatypes does each style use?**

The two different types of assignment are procedural and continuous. Continuous assignment uses wire data types, whereas procedural uses primarily registers but can also use wires.

2) **In the last lab, we discussed two different styles of writing combinational logic in Verilog. The hierarchical 8-bit adder is best described as which of the two? How about the procedural 8-bit adder? Justify your answers.**

The hierarchical 8-bit adder is best described as structural Verilog because from the code the design of the circuit can easily be inferred. The procedural 8-bit adder is best described as behavioral Verilog because from the code the behavior of the circuit can easily be inferred, although the underlying implementation of the circuit is not at all obvious from the code.

3) **In the hierarchical 8-bit adder, it can be seen that the circuit's gate-level structure is explicitly described. On the other hand, the procedural 8-bit adder does not describe much of the gate-level structure. What are the advantages and disadvantages of each implementation approach (hierarchical and procedural)? Furthermore, what are the advantages and disadvantages of using higher-level operators in Verilog, such as using arithmetic operators instead of bitwise operators?**

The advantages of a hierarchical approach in Verilog are such that it is easy to infer the underlying structure of the circuit that the program is trying to describe. In other words, the circuit design is easily inferred from the code. The disadvantages of such an approach are that the code is far more extensive.

The advantages of a procedural approach are such that the code is compact and succinct. However, the disadvantages are such that it is more difficult to infer the actual implementation that the hardware designer is intending.

The advantages of using higher-level operators like arithmetic operators instead of bitwise operators is that the code will perform the same function, and it will be far simpler than if bitwise operators were used. On the other hand, there is a bit of a trade-off: using arithmetic operators means a Verilog synthesizer is free to use any adder circuit it sees fit during the synthesis process (so you give up control of exactly what the final circuit after synthesis will look like). On the other hand, when using bitwise operators, the actual circuit you intend to be implemented will be synthesized exactly as you intend.

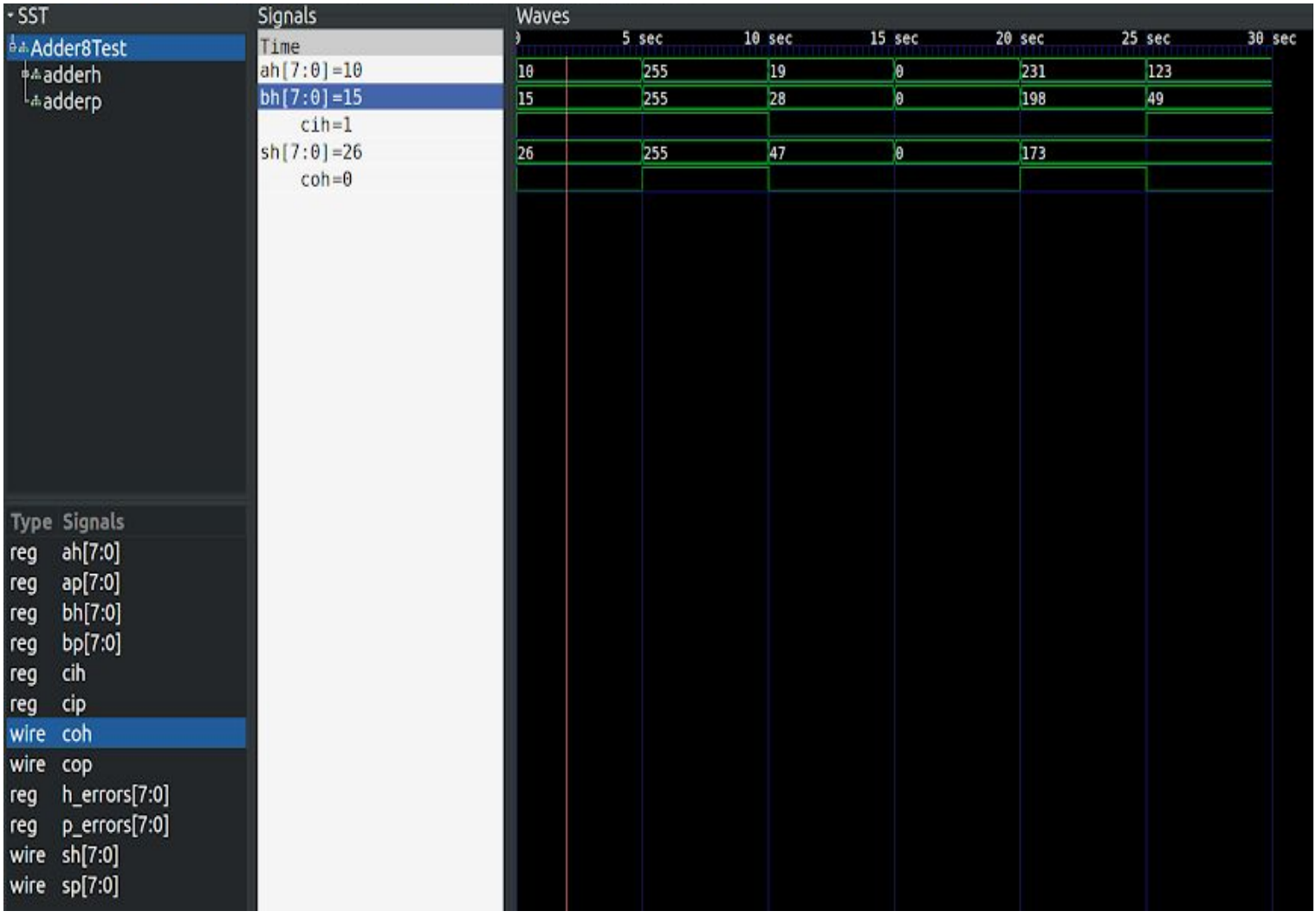
Describe the test cases that you chose for your adder. Explain why you chose each and how each contributes to testing your adder's correctness. It's okay if a few of your choices are random, but at least a few should have been chosen intentionally to test specific behaviors.

4) Describe the test cases that you chose for your adder. Explain why you chose each and how each contributes to testing your adder's correctness. It's okay if a few of your choices are random, but at least a few should have been chosen intentionally to test specific behaviors.

My first test case was to test the edge case (inputs were a = 255, b = 255, ci = 1), where the carry out bit should have been 1 and the sum should have been 255 (i.e. the maximum possible sum). My second test case was a random test case (summing 19 and 28) which did not use either ci or co. My third test case was a sanity check (summing 0 and 0 with ci = 0) to make sure the output was all 0. My fourth test case was again a random test case, which used numbers large enough so the co = 1 (the test was 231 + 198 with ci = 0). My final test case used the carry in bit (a = 123, b = 49, ci = 1) but not the carry out bit (sum = 173, co = 0).

These tests were deliberately constructed to test both edge cases, a sanity check, as well as various combinations of ci and co to make sure all parts of the adder circuit worked.

5) Screenshot:



The given test case adds the three numbers 10, 15 and the carry in bit, 1. These three numbers sum to 26. As can be seen from the waveform, the inputs match the test case (i.e. ah = 10, bh = 15, and cih = 1). Since the carry-out bit is only necessary if the sum ‘overflows’ (i.e. when 2 8-bit numbers are added the sum is 9-bits long and therefore the carry-out bit is needed to represent the largest bit), the carry-out bit in this test case is 0 (as 26 does not need 9 bits to represent it in binary). Furthermore, we can see the waveform is correct as the sum displays 26 (10 + 15 + 1 = 26).

6) Feedback

This lab was useful. There wasn’t very much info on how to instantiate a module in the tutorial, so I had to go to the lab TA’s for that.