ELE 206 Lab 3 - Oliver Schwartz

Review Questions

1.10

a) **What will the value of x be while this circuit is in operation? Why?**

As there are two assignment statements, x will take the value of the first statement. I.e. x will take the value of 5'b00001.

b) Why can there not be two assignments to the same wire in a circuit? How does this extend to the physical properties of digital circuits on hardware?

An assignment statement corresponds to connecting a wire to other components (either other wires or registers). There cannot be two assignments to the same wire, because this would mean connecting a wire to 2 separate external drivers, which would mean you would have to duplicate the wire (so the assignments don't interfere with one another).

1.12

Under what circumstances does the *if* block execute?

The if block will execute if any of the bits in the 4-bit register vector x is a 1.

3.1

List the characteristic differences between combinational and sequential logic.

Combinational logic refers to logic where the output can be computed based solely on the current inputs. Unlike combinational logic, sequential logic requires memory. Sequential logic uses relevant history as well as the present inputs to determine an output. This memory usually comes in the form of registers which are controlled by a clock signal.

3.2

a) **What are the differences between the sensitivity list of an always procedure in a sequential circuit implementation and in a combinational circuit implementation?**

The sensitivity list of an always procedure in combinational logic contains right-hand-signals (i.e. the always block is triggered by a change in any number of various wires and registers). In sequential logic, an always procedure contains a special modifier (either posedge or negedge) which means the procedure will run on the rising/falling edge of the indicated signal (usually the CLK signal).

All elements in an always block (sequential or combinational) must be reg datatypes. However, in combinational logic these are just treated as wires. In sequential always procedures, on the other hand, they are treated as flip-flop style registers (i.e. these reg types will only be assigned a new value on the specified clock edge).

Finally, the operators in combinational/sequential always procedures are different. In combinational logic, the blocking assignment (=) operator is used, whereas the non-blocking operator is used in sequential logic (<=). These two differ in their operation. Blocking assignment evaluates and stores immediately, with statements executed in order.

Non-blocking assignments are also executed in order, but the values are not updated until the whole block has executed.

3.3

a) **WHy is it good practice to use the non-blocking assignment when representing sequential logic?**

Non-blocking assignments are also executed in order, but the values are not updated until the whole block has executed. Because of this property, it is good practice to use non-blocking assignment when representing sequential logic because it allows a programmer to represent things like shift registers, as well as various other register types where the values are designed to propagate sequentially through the circuit.

b) **How would you change the code above to implement the desired sequential circuit using non-blocking assignment?**

You would rewrite it as follows: (by removing register b)

```
module FixMe(
  input clk,
  input a,
  output reg c
);
  always @(posedge clk) begin
    C <= a;
  end
end module
```

3.4

We have seen earlier that the reg datatype is coerced to behave like a wire in implementations of combinational logic. How does this differ in implementations of sequential logic?

In sequential logic, we can use the non-blocking operator to delay the updates of registers. This differs from using reg datatypes in combinational logic, where the value of a reg is updated immediately (and simply behaves like a wire). In sequential logic, we can simulate feedback loops and delay the updates in registers, which is unlike the instantaneous behaviour of wires (or a reg in combinational logic).

7.3

Assume you are asked to design a circuit with a clock period of 20ns.

a) **What time scale would you specify for the circuit?**

I would specify a timescale of 10ns, i.e. **`*timescale 10ns/10ps***

b) In your testbench you would need to cycle a clock. Write the code that you would use to do this. How did you determine the delay?

Assuming we want the clock to start low,

reg clk = 0;

always begin

#2 clk = ~clk;
end

This will make the clock's value alternate every 20ns (as 1 time unit is 10ns) and start with a value of 0. I determined the delay from the given period: i.e. 20ns.

Lab Write-up

1) **Show how the circuit is bistable when the input is (S^-, R^-) = (1, 1), that is, the circuit could be in two different stable states with different output values. Describe how the latch could be operated to store one bit of information.**

Case 1:
Let's assume Q is initially 1. This means NQ = 1 NAND 1 = 0. When this feeds back to the top NAND gate, Q then gets the value of 0 NAND NS = 0 NAND 1 = 1 (which was Q's initial value). Therefore, (NS, NR) = (1, 1) is a stable state as NQ and Q are fed back their initial values.

Case 2:
Let's assume Q is initially 0. This means that NQ = 1 NAND 0 = 1. When this feeds back to the top NAND gate, Q then gets the value of 1 NAND 1 = 0 (which was Q's initial value). Therefore, (NS, NR) = (1, 1) is a stable state as NQ and Q are fed back their initial values.

Therefore, (NS, NR) = (1, 1) is a bistable state as it can assume 1 of 2 stable states depending on the initial values present in the feedback loops.

This latch could be operated to store one bit of information through the following method. If (NS, NR) are both 0, 1 is stored. If NS is then changed to 1, 0 is stored. Therefore, one could simply keep NR at 0, and input the NOT of whatever value you want stored to NS (i.e. if you want 1 stored, simply input this 1 and then a not gate, and a 1 will be stored at Q, or vice versa if you want 0 stored).

2) What happens when the input is (S^-, R^-) = (0, 0)? What would happen if the input instantaneously transitions from (0, 0) to the "store" input, (1, 1)? Would something be stored? What if the transition is not instantaneous, and one bit in the input transitions from 0 to 1 some time before the other?
If the input is (0,0), then Q = 0 NAND NQ = 1. Therefore NQ = 1 NAND 0 = 1. Thus, for (0, 0), both Q and NQ take the value 1. If NS and NR instantaneously change to both (1, 1), then Q and NQ will then take the value of 1 NAND 1 = 0. This will feed back to the NAND gates, and Q and NQ will then take the value of 1. Thus, NQ and Q will rapidly oscillate between 0 and 1.

If the input changes to (1, 1) but one bit transitions before the other, then a stable state will be achieved. If NS changes before NR, then Q will take the value 0, and NQ will take the value 1. If NR changes before NS, then Q will take 1, and NQ will take 0. Either way, a stable state will be achieved. However, this stable state is only achieved if the new value of

Q/NQ propagates through the NAND gate before the other input switches (otherwise the circuit will oscillate as above).

3) The simulation avoids the problematic transition from (0, 0) to (1, 1) discussed in the previous write-up question. The simulator is a single-threaded program running on your computer, meaning it cannot perform two computations in parallel. Instead, after each change in input value, it works its way through the circuit, evaluating changes in signal values one after another. Explain how different simulators might give different results when attempting to simulate the problematic transition in input values. You do not need to check your answer using simulation.
When attempting to simulate the problematic transition from (0,0) to (1,1), different simulators may give different results depending on how the circuit is simulated - i.e. as values cannot be computed in parallel, the value which is computed first will influence the likely behaviour of the circuit.

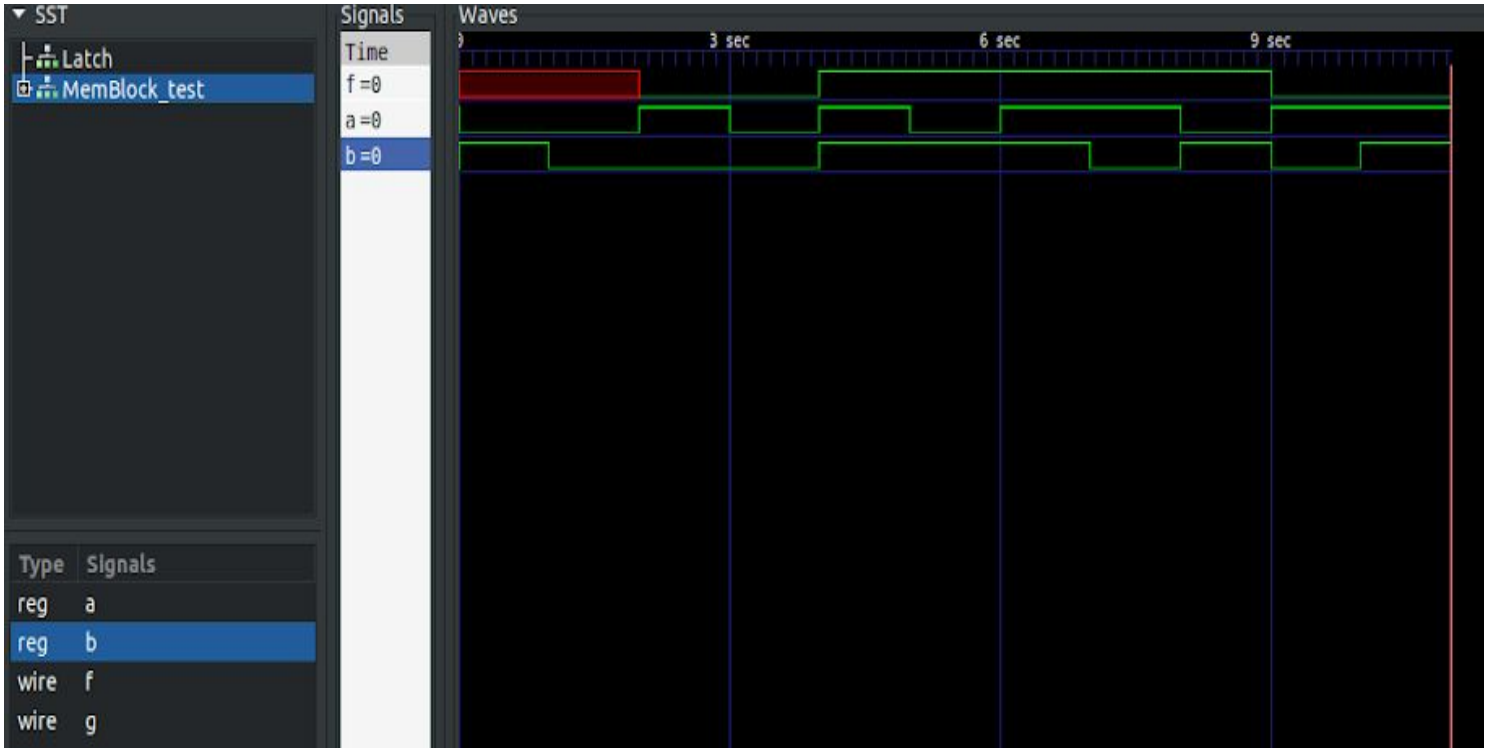
4) In a large computer system, memory units must be occasionally replaced or upgraded, e.g. upgrading a hard drive in a data center. These memory units behave similarly, sometimes following an industry standard, but they may be implemented very differently at the gate-level. Similarly, in a larger circuit design, a designer may want to substitute one latch for another. This would need to be done without causing functional changes in the larger design. What conditions and/or restrictions on input behaviors should be maintained across the design in order for a designer to be able to successfully substitute an $S^{-}R^{-}$ latch for another?
When substituting an SR latch for another latch, a designer must make certain that the latch will behave under all the same inputs as the initial latch. Otherwise, the new latch's behaviour would not match the behaviour of the old latch - the mismatch in behaviour might propagate through the whole circuit and cause unwanted errors and unpredictable behaviour. Furthermore, if substituting an SR latch for another latch, the designer must be sure that the latch's inputs will never make the problematic transition from (0,0) to (1,1), or else risk further unpredictable behaviour.

5) What are the advantages and disadvantages of implementing this latch structurally and behaviorally? Which implementation do you think is best? Why?
There are advantages to both: both require very little logic and are very easy for a hardware designer to implement. However, implementing this latch structurally makes it a little more obvious as to what is going on in the latch. Furthermore, implementing the latch structurally is probably a little more 'true' to what is going on in the circuit - because it uses only wires and blocking assignment, whereas my behavioural implementation used non-blocking assignment and registers (whereas in actual fact there are only wires and no registers in the circuit). In this case, as the circuit itself is so simple, I believe there is no 'best' way - both are relatively straightforward to implement and understand.

6) What 1-bit memory block that was discussed in lecture and in the textbook does this circuit behave the most like? Identify one of the inputs of this circuit as either an enable or clock signal. To support your answer, include one or more screenshots of

waveforms in GTKWave recorded from the simulation of the circuit with your testbench.

This circuit behaves like a rising-edge triggered flip-flop. As seen in the screenshot below, where input a is the clock signal, the output (f) changes only only the rising clock edge (assuming the input, b, differs from the previous value of f). Initially, the output of the circuit is unknown as it is only set on a rising edge of the clock (at t=0 we have a falling edge, not a rising edge). At t=4, we have a rising edge, and the input goes from 0 to 1, so the output also changes from 0 to 1. At t = 9, we have a rising edge, and the input falls from 1 to 0, so the output also falls from 1 to 0.



7) In terms of the number of two-input gates that make up the circuit, compare this circuit with the memory block that it behaves similarly to that was discussed in lecture and in the textbook. Which likely uses fewer transistors?

The memory block that it behaves similarly to is the D flip-flop, which consists of 2 D latches connected to a clock (with one clock input, to the master, NOT'ed). A D latch consists of two AND gates, as well as 2 NOR gates. So two of them make 8 2-input gates. Compares to the circuit in this lab, which is functionally the same, the lab circuit consists of 5 two-input NAND gates and 1 3-input NAND gate (which can be broken down into a 2-input AND gate and a 2-input NAND gate). This makes a total of 7 gates - so the two circuits are similar in terms of gates.

The circuit from lab likely uses fewer transistors, as it has fewer overall gates. Additionally, it has more NAND gates than the D flip-flop, and NAND gates use fewer transistors than AND gates (of which there are more in the D flip-flop).

8) Feedback

This lab was fine. Not very difficult or confusing.