ECEN 5863 Homework Set 2

Matt Hartnett

Q1)
Here is my gate level simulation that runs through the operation of the FIFO:

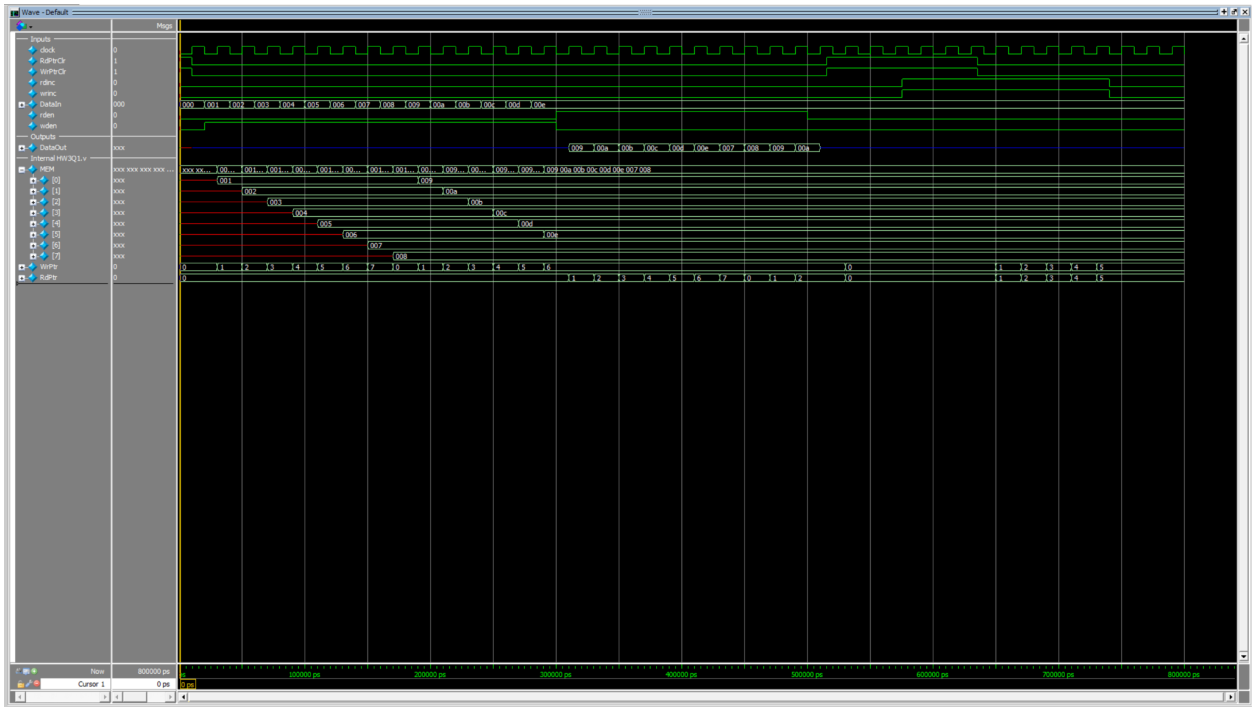

Fig 1. A screenshot of the functional Q1 FIFO.

This simulation can be created by running the HW3Q1_run_msim_rtl_verilog.do file in modelsim which automatically configures the dividers and signals.

As you can see from the above simulation screenshot, my testbench begins by running through the FIFO in a for loop and writing the value of the incrementor (i) from 1 to 14 into the FIFO. You can see that behavior in the MEM section, which runs through each of the memory addresses and writes to them, overwriting once the write pointer overflows back to 0. From there, the test bench reads out 10 values from the FIFO, which loop in a similar fashion. These read operations occur synchronously to the clock (it was not specified in the documentation if the read is asynchronous or not). When the read enable is not active, then the DataOut signal is high Z. It then tests that the synchronous resets of the pointers work as expected, as well as the incrementing enables, rdinc and wrinc. This test bench runs through the full gamut of the FIFO, and it demonstrates that it's functionality is as designed.

The operation of the FIFO is rather simple. It operates completely synchronized to the clk signal. On a positive edge of the clock, it will run through three if statements to create its behavior.

The first if statement determines what should happen to the read pointer based on the RdPtrClr reset, the rdinc signal, and the rden signal. If it determines that there should be a read operation, then it takes the data stored in the 2 dimensional MEM array, which acts as the storage of the FIFO, and outputs it to the Data register.

The second if statement determines if the DataOut signal should contain a value or be high Z. The DataOut output is tied to a register Data, which helps with this process. This is a fairly simple if statement, as all it does is check if the rden value is low, and if so, assign the Data register to high Z.

The last if statement determines what should happen to the write pointer based on the appropriate signals. If a write is to occur, then it takes the value of DataIn and stores it in the MEM array at the location of the write pointer. It then increments the write pointer.

Q2)
Here is a simulation screenshot showing the fully functional non-clocked ALU. This testbench works by running through every possible opcode with two randomly generated (but fixed) values for A and B. For every combination of opcode, it computes the expected value from the ALU. It then compares this expected value with the actually returned value. If the values are different, then it increments the error_counter integer. This helps keep track of the total number of errors in the design.

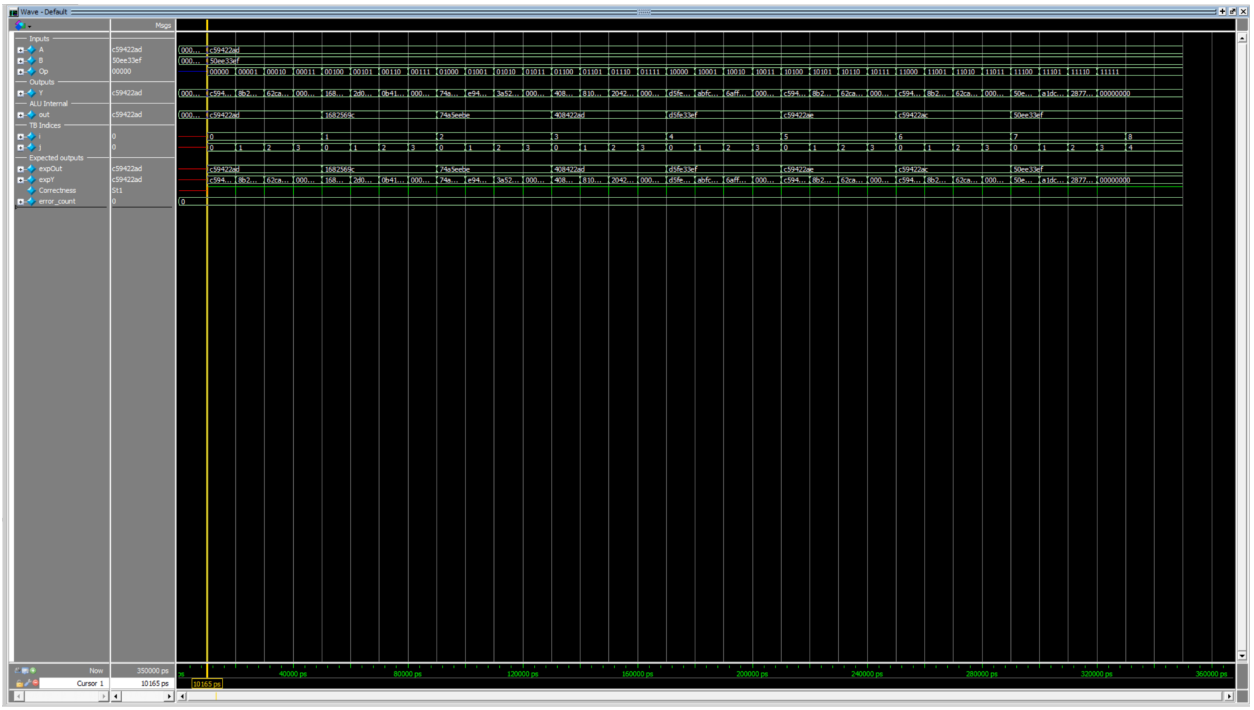

Fig 2. A screenshot of Q2's test bench for the non-clocked ALU.

As shown in Fig 4, the clocked ALU behaves as expected as well. Even though the inputs are placed on the bus before the rising edge, the output of the ALU isn't updated until the rising edge of the clock. Originally, this was reporting as incorrect, as shown in Fig 3. This is because the expected output of the ALU was being calculated independently of the clock. Once the ALU expected value was adjusted in the test bench, everything was correct.

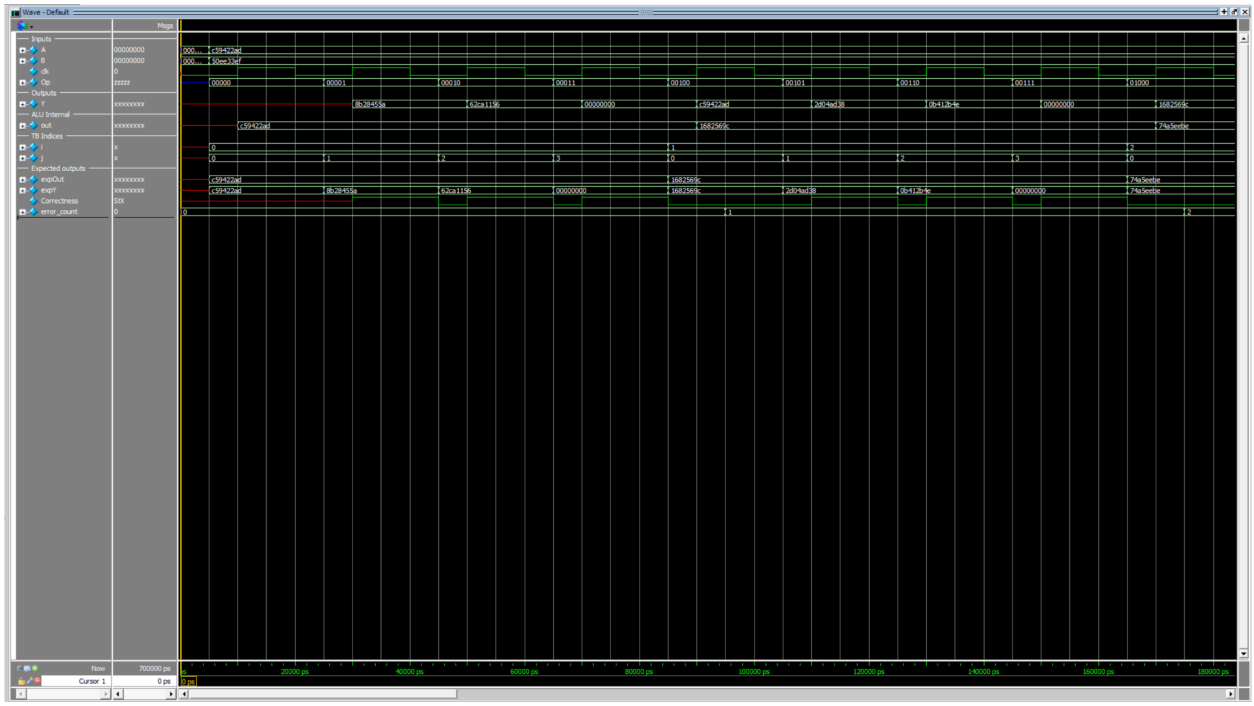

Fig 3. A screenshot of a test bench for the clocked ALU with an unclocked test bench.

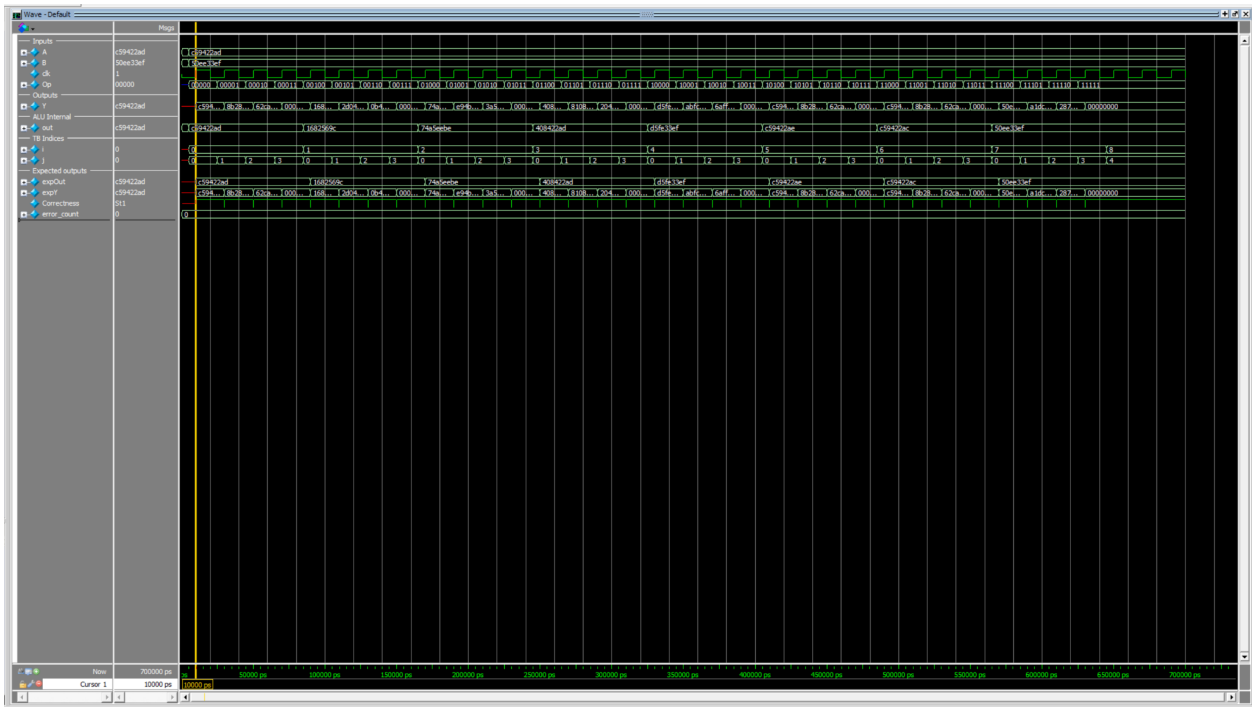

Fig 4. A screenshot of a test bench for the clocked ALU with a clocked test bench.

The clocked and the unblocked ALU operate in a very similar fashion. Both have 2 case statements that govern the behavior. The first case statement determines the mathematical operation based on the highest 3 bits of the opcode. This generates a value stored in a “out”

register. The second case statement changes what the actual output of the ALU is based on the lower 2 bits of the opcode. This is the case statement that determines if the output is passed, shifted right or left, or if 0s are output instead.

The primary difference between the clocked and the unblocked is in the sensitivity list of the second case statement. By modifying the sensitivity statement to require that the output is only updated on the positive edge of the incoming clock, it switches the ALU from unclocked to clocked.

The Fmax of the clocked ALU is: 108.46 MHz, as shown in Fig 5
The number of LUTs used for the clocked ALU is: 167

The Fmax of the unclocked ALU is: 64 MHz, as shown in Fig 6
The number of LUTs used for the unclocked ALU is: 199

As shown by the number of LUTs and the Fmaxs, the clocked design is obviously superior to the unclocked design. This may be due, in part, to the fact that all FPGA tools are optimized for synchronous logic as opposed to combinational logic.

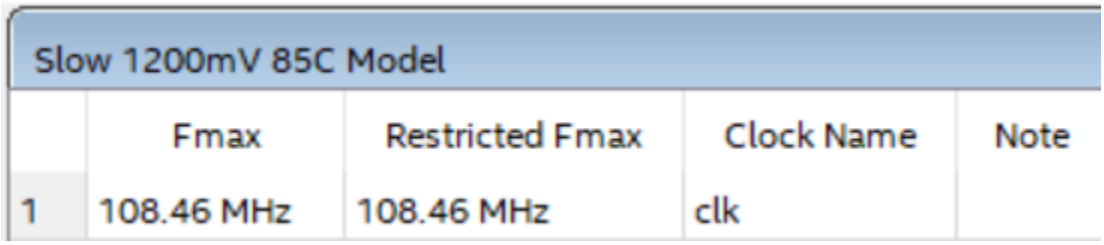

Fig 5. A screenshot of the timing analyzer showing the Fmax of the clocked ALU.

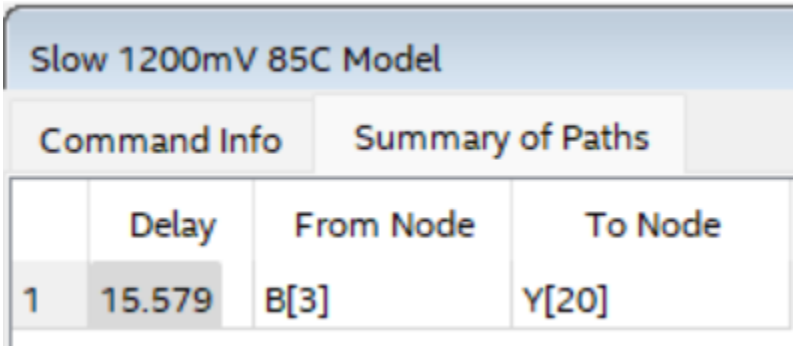

Fig 6. A screenshot showing the path with the largest delay of 15.579ns, which equates to an Fmax of 64MHz.

Q3)
For this testbench, I started with the template and made modifications to fit the comparator design. For this comparator, there are only two input signals and one output signal. With for loops, the test bench runs through every possible combination of the two inputs. It then compares the output of the DUT with the expected result of a comparison of the two values. If the output does not match the expected output, then an error is generated by incrementing the error counter and printing out the word “error”. The result of the error counter is also written to a log file. Fig 7 shows the output of the test bench, where you can see that the test bench runs through every value of A and B possible.

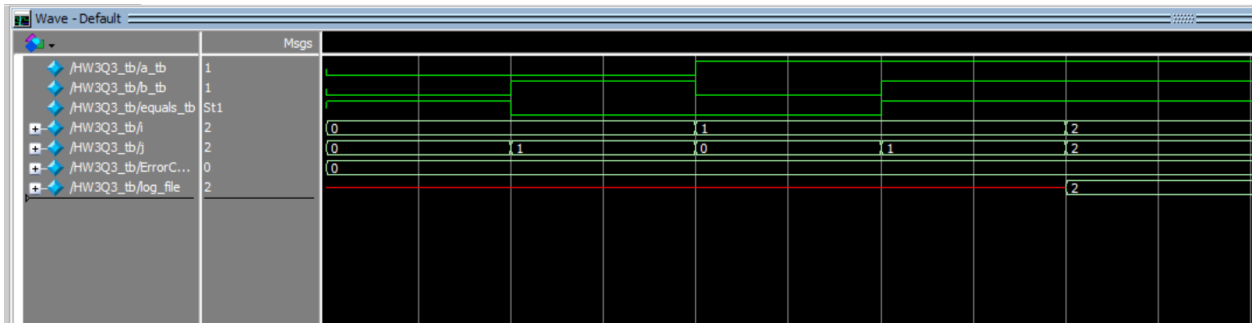

Fig 7. A screenshot of the test bench used to verify the 2 bit comparator.

Q4)
In order to run the simulation with my project name, I had to modify the command to the following:
vsim -t 1ps +transport_int_delays +transport_path_delays -sdftyp /HW2VerilogRAM=RAMVerilogProject_7_1200mv_125c_v_slow.sdo -L altera_ver -L cycloneive_ver -L gate_work -L work -voptargs=\"+acc\" work.HW2VerilogRAM

This allowed Modelsim to find the right libraries and the right modules to run the simulation.

Once the simulation began, I started by forcing the following inputs: address = 0, clk = clock with a 10ns period, data = 0xAAAAAAAA, wren = 0. I ran in 10ns time steps until the output bus q was properly updated to the value in memory at the address of 0. I then turned on the wren enable signal for a single clock period and ran until the test data showed up on the simulation. This represents the amount of time it requires to both read and write from the RAM. I then changed the address and ran until I saw the updated value on the bus. This represents the amount of time required to read from the RAM. After that, I wrote my test value into the RAM on the new address, and ran until I saw it appear on the bus. All of this can be seen below in Fig 8.

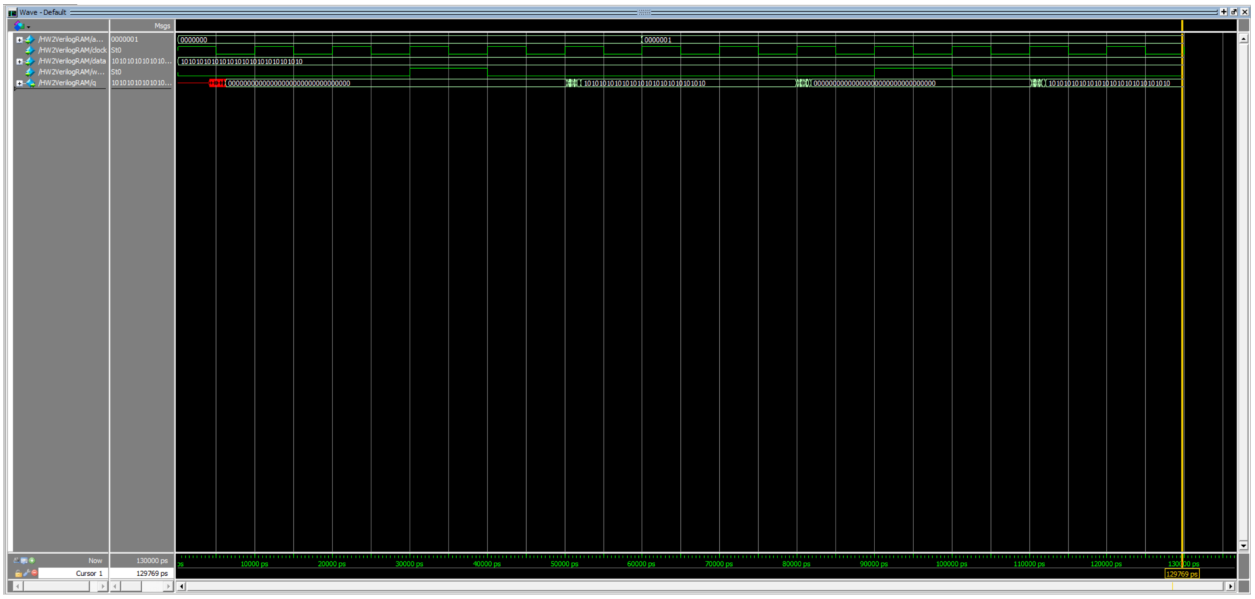

Fig 8. A screenshot of the gate-level simulation of the RAM module created in Homework2.

Q5)
My implementation of the state machine is very similar to that of the textbook. The main difference is that the A train goes in the opposite direction. This effectively switches out sensor 1 and 4. The structure of my Verilog is very similar to that of the textbook, where there is one always block controlling the state machine, and another controlling the output of the state machine. I also utilize some assignments and registers to help simplify everything down.

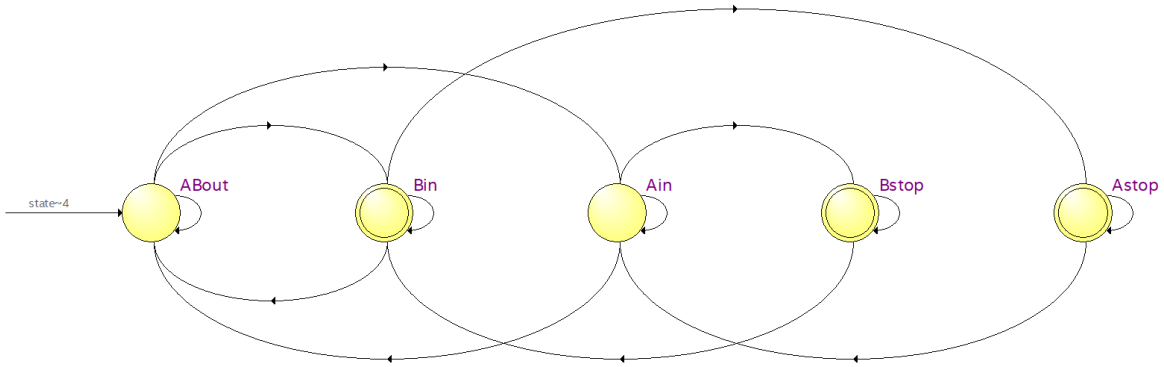

Fig 9. A screenshot of the state machine diagram generated by Quartus.

In order to properly simulate the design, 4 tests were conducted. The 4 tests were designed to run the state machine through all of its possible environments. The first two tests check that A and B can freely enter and leave the middle track. The second two tests check that when one train is in the shared track, the other train stops. As you can see from Fig 10, all these states operate as expected, with track switches and directionality control behaving as specified.

Fig 10. A screenshot of Modelsim with the testbench running through all possible state paths.

Q6)
Shown in Fig 11 is a screenshot of the Quartus state machine editor, with the completed state machine. Creating this state machine was relatively simple, as all of the information required was in the state diagram of the problem. I began by creating all of the input and output ports, and giving them the appropriate names. One thing to note is that the problem specified CON as an input port, but the diagram had several references to CONT and no references to CON. With that in mind, the state machine that I designed only uses CONT, and there is no CON port. From the port declaration, I moved on to create each of the states and their appropriate transitions. For state0, I set it as the default state so that a reset would bring the state machine back to state 0. I then defined all of the transitions with the equations provided in the problem statement. The final thing to do was to define the state outputs and convert it to an HDL file. Once the Verilog file (HW3Q6.v) was generated, I set it as my project's top level entry and compiled.

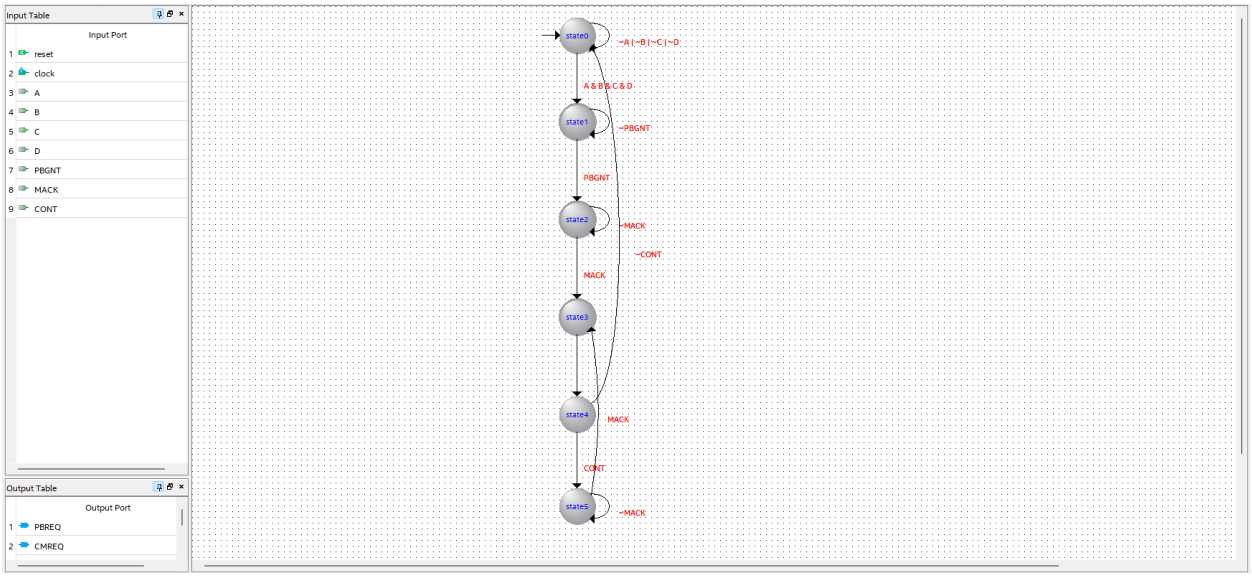

Fig 11. A screenshot of the Quartus State Machine Editor, with the completed state machine.

Q7)
Shown below in Fig 12 is a timing diagram of the circuit and the pulse of A. As you can see, the large variance in signal delay between the gates leads to very large changes in potential outcomes. Although difficult to convey by the time that the outcome is calculated, there are many possibilities for glitching and inconsistent behavior. This erratic behavior is caused by both a long path (in terms of # of gates), large variance in gate delay, and a very small relative pulse. If some of these factors were mitigated, by either slowing down the signal, adding registers, or by having more reliable gates, then this would be a more acceptable result.

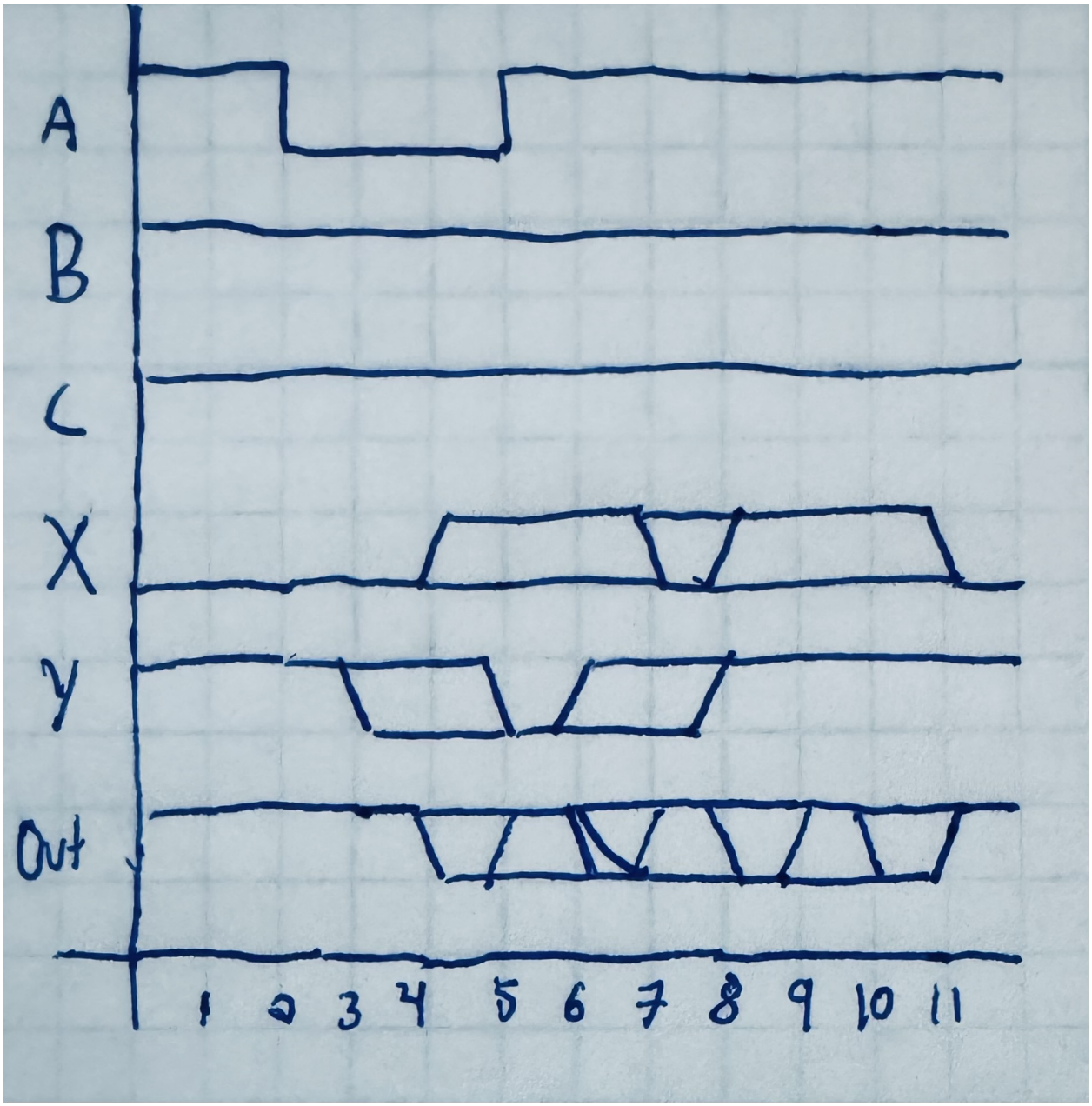

Fig 12. The full timing diagram of the design.

Q8)
This design is more complicated, as it now incorporates registers. In the analysis, it's easier to break the total path into three segments: register 1, register 2, and the final output. The full timing diagram is shown in Fig

For register 1, there is a lot of uncertainty due to the 2 gate path on the data signal. If the pulse makes it through on the minimum delay of 1 ns, then the pulse is entirely missed with no setup violation. However, if both of the delays are 3ns, then the rising edge of A wouldn't result in a signal change of D for register 1 until 11ns. This means that the register could have or could not have captured the value of A, it depends on the timing. In addition, there are many scenarios of in-between delay that result in a setup violation. Since there are three possibilities: A is missed, A is captured, or a setup violation occurs, there is no way to tell what the value of X is after that rising edge.

For register 2, the pulse on A has completely settled by the 8ns mark. While this doesn't result in a setup violation, since the clock rising edge occurs at 10ns, it does mean that the pulse is completely missed by the Y net. If the A pulse was longer, it could have resulted in a potential setup violation, as A would have had to be low until 9ns to completely avoid a violation while still being captured by register 2.

On the output path, it's clear that the output is low for the majority of the signal path, but as soon as the uncertainty of register 1 enters the equation, it's impossible to know what the output is. In this regard, this design does not meet timing requirements and doesn't have timing closure. This is a very low reliability design, caused by a variety of factors.

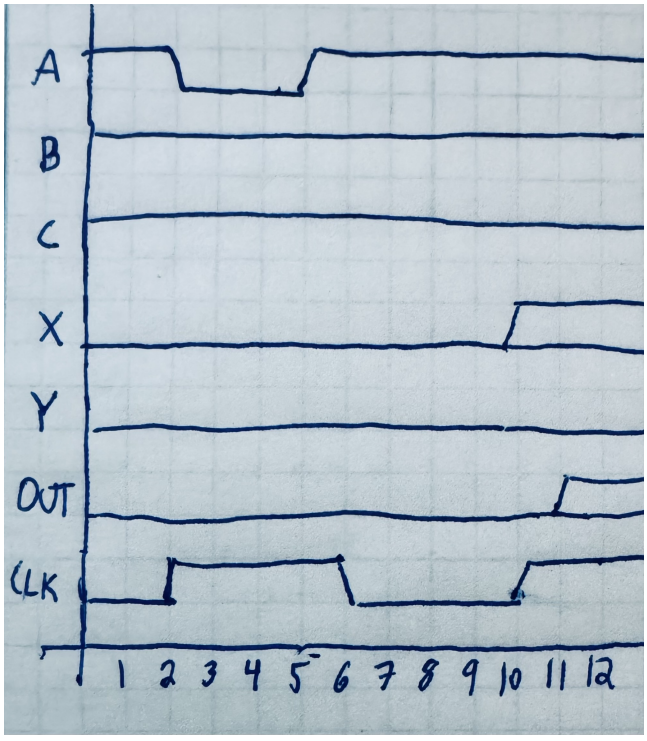

Fig 13. A timing diagram of the design with registers.