Name:

ECE 3544: Digital Design 1

Homework Assignment 6 (100 points)

*When writing a Verilog module or naming a file, replace YOURPID with your Virginia Tech PID. Name the file according to the module name that you use.*

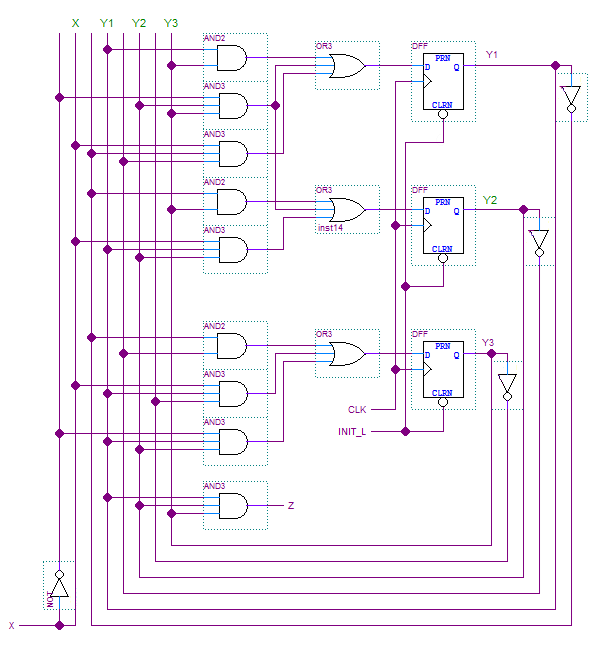
For each design that you represent in a Verilog module, copy your source code into the document. For each design that you simulate in ModelSim, include waveforms displaying the correct operation of each module

In addition, submit the .v file containing each design and each test bench that you write. Your files should contain header information as described in Project 1 and be neatly formatted and commented. Submit your files separately on Canvas. Do not put them into an archive. *Make sure that you upload all your files.*

*Use this logic circuit in Problem 1.*

This sequential circuit has input X and output Z. For clarity, input X is labeled twice.

The circuit uses a positive-edge triggered clock and an active-low asynchronous reset.



1. (25 points) The previous page shows a state machine. It has one input (X), a three-bit state (Y1 Y2 Y3), and one output (Z).
2. Derive a state diagram for the state machine. Use states S0-S7 to represent the binary codes for Y1Y2Y3 from 000 to 111. For example, S0 = 000, S1 = 001, and so forth.
3. Use the state diagram or table to create a Verilog model of the circuit and verify that its operation is the same as described by your answer to part (a). Use a procedural approach for implementing the state machine. Use parameters to define the states. (You may use continuous assignment instead of procedural assignment to describe the output.)

module problem1\_YOURPID(clock, init\_l, in, out);

input clock; // System clock

input init\_l; // Asynchronous active-low init

input in; // FSM input

output out; // FSM output

***You may add the state as an output port to the module and port declarations for debugging.***

1. Write a test bench (tb\_problem1\_YOURPID) to verify that your circuit implements the state machine model you found in part a.

*This is an analysis problem. You have a logic circuit. Can you derive logic equations from it? Can you map these equations? Can you use the logic values to reconstruct an encoded state table: one where you have values Y1+Y2+Y3+ and Z for all combinations of X, Y1, Y2, and Y3 – that is, for all combinations of input and present state? Can you use the state assignment described above to reverse the encoding and obtain the state table? Can you use the state table to draw the state diagram?*

*It is also an implementation problem. Can you use the state machine model directly to write a Verilog procedural model for the state machine?*

|  |
| --- |
| **Tip for Problems 2, 3, and 4:**  For your homework submission, write a test bench to verify that the counters in Problems 2, 3, and 4 work correctly. Even though all of the counters have slightly different input and control interfaces, if you are careful you should be able to use the same test bench to demonstrate the operation of all three counters. Of course, if you are uncomfortable writing a master test bench, you can always write separate test benches for each counter.  Regardless of your approach, you should still *write and test the counters one at a time*. |

One common problem in writing counters involves writing a counter value whose terminal value is not a power of two. The approach to solving this problem depends on whether you are writing a counter “from scratch” or using an existing model, as you might be when using an IC package.

1. (5 points) Write a synchronous decimal counter. This counter increments by one on each clock edge where an active-high enable input is asserted. If the counter state is binary 1001, the state should become 0000 on the next clock trigger where enable is asserted.

Besides the counter state, the counter has a (combinational) “carry-out” whose value should equal 1 if the counter state is 1001 and the enable is asserted.

module problem2\_YOURPID(clock, reset\_l, count, state, carry);

input clock; // System clock

input reset\_l; // Asynchronous active-low reset

input count; // Synchronous active-high count enable

output [3:0] state; // Counter state

output carry; // Counter carry-out

1. (5 points) Write a synchronous 4-bit counter model with parallel load. The operation of the counter is described in the following table:

|  |  |  |
| --- | --- | --- |
| load | count | state after a clock edge |
| 1 | 0 | ins |
| 0 | 1 | state + 1 |
| 0 | 0 | state |

Besides the counter state, the counter has a (combinational) “carry-out” whose value should equal 1 if the counter state is 1111 and the count enable is asserted.

module problem3\_YOURPID(clock, reset\_l, load, count, ins, state, carry);

input clock; // System clock

input reset\_l; // Asynchronous active-low reset

input load; // Synchronous active-high load enable

input count // Synchronous active-high count enable

input [3:0] ins; // Parallel load inputs

output [3:0] state; // Counter state

output carry; // Counter carry-out

1. (5 points) *Using the counter you wrote in Problem 3, create a synchronous decimal counter.*

module problem4\_YOURPID(clock, reset\_l, count, state);

input clock; // System clock;

input reset\_l; // Asynchronous active-low reset;

input count; // Synchronous active-high count enable

output [3:0] state; // Counter state

Your model should instantiate exactly one instance of your counter from Problem 3. Connect the ports of the top-level module to the appropriate ports of the counter instance. Use the control inputs of the counter instance to force the counter to exhibit the desired behavior. *Do not rewrite the logic of the counter from Problem 3!*

(In your model, the carry output of the instantiated counter will likely have nothing to do. This is fine.)

Another problem when writing counters is keeping track of the nature of the control inputs. The user must take care to observe when particular control inputs are synchronous or asynchronous.

1. (5 points) Write a synchronous up-down counter with parallel load. The operation of the counter is described in the following table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| clear | load | count | updn | state after a clock edge |
| 1 | × | × | × | 0000 |
| 0 | 1 | × | × | ins |
| 0 | 0 | 1 | 0 | state + 1 |
| 0 | 0 | 1 | 1 | state – 1 |
| 0 | 0 | 0 | × | state |

Besides the counter state, the counter has a (combinational) “carry-out” whose value should equal 1 if the counter state is 1111, the count enable is asserted, and updn is 0 OR if the counter state is 0000, the count enable is asserted, and updn is 1.

module problem5\_YOURPID(clock, clear, load, count, updn, ins,

count, carry);

input clock; // System clock

input clear; // SYNCHRONOUS ACTIVE-HIGH clear

input load; // Synchronous active-high load enable

input count; // Synchronous active-high count enable

input updn; // Synchronous up-down control

input [3:0] ins; // Parallel load inputs

output [3:0] count; // Counter state

output carry; // Counter carry-out

Write a test bench (tb\_problem5\_YOURPID) to verify that your counter works according to the function table above.

1. (15 points) Use the counter of Problem 5, along with appropriate control logic, to implement a counter that generates the following sequence:

0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9, 8

Upon reaching the last value listed, the sequence should repeat in the same manner.

module problem6\_YOURPID(clock, reset\_l, state);

input clock;

input reset\_l; // SYNCHRONOUS ACTIVE-LOW RESET;

output [3:0] state;

Your model should instantiate exactly one instance of your counter from Problem 5. Connect the ports of the top-level module to the appropriate ports of the counter instance. Use the control inputs of the counter instance to force the counter to exhibit the desired behavior. *Do not rewrite the logic of the counter from Problem 5!*

*This counter you wrote in Problem 5 has synchronous functions: it can be made to count up, to count down, to load a value as its new state, and to clear its state. To generate the desired sequence, you can describe the conditions for which you want to assert as combinational logic functions of the count value. In fact, you can use dataflow operators to describe them that aren’t bitwise logic functions.*

*The reset of the top-level module is a board-level reset. You should probably not connect it directly to the reset port of the counter instance. Instead, think about modifying the clear logic so that it operates under the previous condition OR when the board-level active-low clear asserts. (The main purpose of this top-level reset is to permit you to reset the counter independently of how the clear is used in generating the sequence.)*

Write a test bench (tb\_problem6\_YOURPID) to verify that your counter works according to the function table above.

1. (20 points) For an FPGA whose clock frequency is 50 MHz, write a synchronous counter that produces a one-clock pulse wide output signal every *second* when enabled. Use the module declaration shown below.

(This is a very useful circuit called a *clock divider*. The output pulse can be used as a synchronizing enable for synchronous sequential circuits on the same clock.)

module problem7\_YOURPID(clock, reset\_l, enable, state, out);

input clock; // System clock

input reset\_l; // Asynchronous active-low reset

input enable; // Synchronous active-high count enable

output [?:0] state; // Counter state.

output out; // Output pulse.

As usual, the question mark is not an element of Verilog syntax. It’s there because I am purposefully not telling you what the precision of the counter state is. Work it out.

*Remember: the secret handshake of being a computer engineer is that we start counting at 0.*

1. (10 points) Reconsider the system that you implemented in Project 2 consisting of the transmit and receive modules.



For reference, the counter has a clock-to-output propagation delay of 15 ns. The parity checker has a propagation delay of 18 ns. (Even though this was the typical propagation delay value, let’s use it as though it were the worst-case propagation delay.)

Even though we could find values on the datasheet for the 74FCT821 10-bit register, let’s suppose that the register has a setup time of 8 ns and a worst-case maximum propagation delay of 20 ns. Let’s also suppose that the XNOR gate used as a comparator has a propagation delay of 6 ns.

How do these values change the clock period with which the circuit can be safely clocked?

*Hint: Remember that there are three pathways that go from a clocked circuit element to another clocked circuit element. Recalculate the required period for each pathway and determine which one is the value that limits the speed of the entire system.*

1. (10 points) Consider Drill Problem 13.9 from the text. (It should be on page 730 of the 5th Edition text.)
2. What is the frequency of the clock that used by the flip-flops *in the synchronizer*? (The problem indicates what the frequency of the system clock is. Analyze the circuit and determine how many cycles of the system clock it takes to generate one clock period in the synchronizer.)
3. Using your answer to part (a) along with the remaining details in the problem, calculate the MTBF of the synchronizer. (The electrical characteristics for the flip-flop are in a table on page 706 of the 5th Edition text. Calculate a metastability restoration time instead of using the one from the same table.)