EC311 Introduction to Logic Design
Spring 2022
Lab 3: Designing Finite State Machine (FSM)

Goal

- Design and write behavioral Verilog code for Mealy and Moore state machines
- Understanding counters and debouncers

Abstract

A finite-state machine (FSM) or simply a state machine is used to design both computer programs and sequential logic circuits. It is conceived as an abstraction of a finite number of states. The machine can be only in one state at a time; the state it is in at any given time is called the **current state**. It can transition from the current state to another state depending on some triggering conditions. Whenever a transition happens, the FSM generates an output. There are two basic types of sequential networks—Mealy and Moore machines—that are used to model an FSM. In **Mealy** machines, the output depends on both the **current state** and the **current inputs** whereas in **Moore** machines, the output depends only on the **current state**.

Tasks

1. **Mealy Machine**

A general model of a Mealy sequential machine which generates the outputs and the next state, and a state register that holds the current state. The state register is normally modeled using D flip-flops. The state register must be sensitive to a clock edge. The other block(s) can be modeled either using the always procedural block or a mixture of the always procedural block and dataflow modeling statements; the always procedural block must be sensitive to all the inputs being read into the block and must have all the outputs defined for every branch in order to model it as a combinatorial block.

Design a Mealy state machine that holds "1" as an output for 1 clock cycle any time a sequence of 01 is is received on a bit stream, with no overlap between sequences. For example, if the input bit stream is 011001, then output bit stream is 010001. The Mealy state machine has one input (`ain`) and one output (`aout`). Write Verilog code for this state machine.

Develop a test bench and verify the state machine model through a behavioral simulation.

2. **Moore Machine**

In a Moore sequential machine, the output is generated from the state register block. The next state is determined using the current state. Here the state register is also modeled using D flip-flops. Moore machines are usually described by using three blocks, one of which must be sequential and the other two can be modeled using always blocks or a combination of always and dataflow modeling constructs.

Design a sequence detector implementing a Moore state machine which has a two bit input (`ain[1:0]`) and one output (`aout`). The output `aout` begins as "0" and remains constant unless one of the following input sequences occurs:

(i) The input sequence `ain[1:0] = 00` causes no change in the output
(ii) The input sequence `ain[1:0] = 01` causes the output to become 0
(iii) The input sequence `ain[1:0] = 11` causes the output to become 1
(iv) The input sequence `ain[1:0] = 10` causes the output to toggle.

Develop a test bench and verify the model through a behavioral simulation.

3. Button Trigger Fibonacci Counter

a. Fibonacci counter

Write behavioral Verilog code for the design of Fibonacci series counter and utilize the internal clock and the push-buttons on the Nexys-3 FPGA board.

This counter should output (one output per cycle) the first 10 numbers of the Fibonacci series and then loop back. You can use the global clock, a reset, and a count trigger as inputs. At the positive edge of the clock, if the count trigger is high then the counter is incremented. At the positive edge of reset, the counter should be reset to 0. Connect the reset and count trigger respectively to two push-buttons. You can assume that the first two numbers of the Fibonacci series are 0 and 1. You will need to figure out how many bits will be required to represent the first 10 numbers of the Fibonacci series. Output those bits onto the LEDs.

b. Debouncer

You will notice that one push of the trigger button does not always increment the counted number (on LEDs) in the correct order, but rather in random order. This happens because of the mechanical "bouncing" of the button. For the reset input, you can eliminate this issue by holding the reset button for more than one second. But for the count trigger input, you need to implement **a debouncer**—a low-pass filter that cleans the input signal.

As a very basic debouncer, we can implement a second counter that increments every clock cycle after the 1st push-button was pressed. Only once the second counter has reached its maximum value, we determine that the push-button has changed state. At this point we can also reset the second counter. This technique utilizes the fact that our global clock is significantly faster than the time it takes a person to press the push-button repetitively.

Your debouncer module should have global clock and reset inputs, as well as a push-button input. Its output should be the clean push-button signal. A sample algorithm for implementing the debouncer is shown in Figure 1 below.

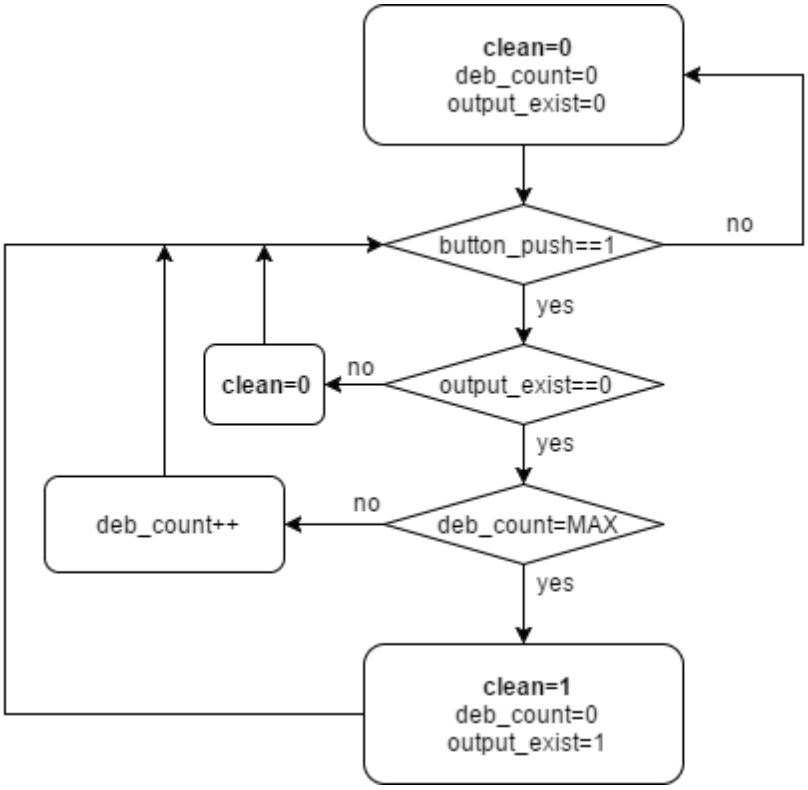

Figure 1: Sample algorithm for implementing the debouncer

Be sure to test your bouncer with different max values until you find one that works well.

For more information on debouncers and possible solutions, see
http://www.fpga4fun.com/Debouncer.html

c. Final Design

Your final design for the counter (i.e. "debounced counter") should instantiate two modules: debouncer and counter. It should have global clock and global reset inputs, a push-button input, and an Fibonacci series counter output. Refer to figure 2 below.

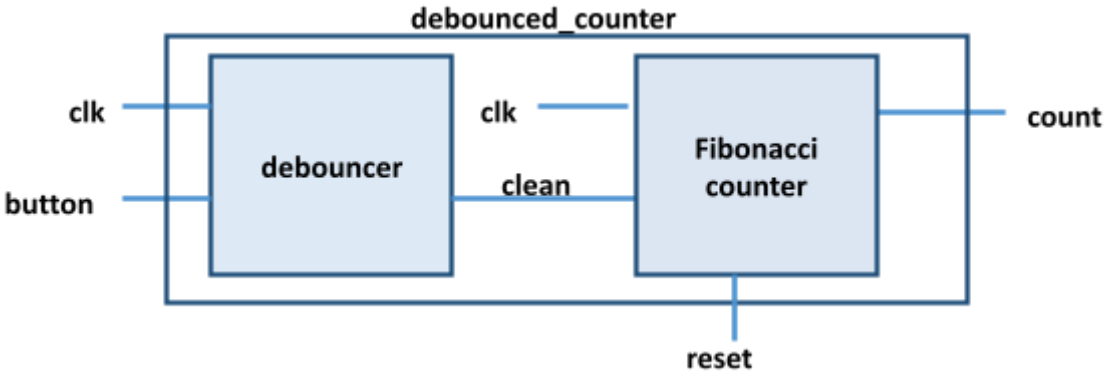

Figure 2: Block diagram for the final design

Deliverables

a. Show the simulation waveforms for all three parts to the TAs.
b. Demo your final design in Task 3 uploaded to the Xilinx FPGA board.
c. Submit your Verilog codes on Blackboard. It is recommended to tar or zip all your code together before uploading it.