Lab 6 - Verilog FSM Tutorial

OVERVIEW

A finite state machine is simply a collection of states and the transitions which allow the machine to go from one state to another based on the current value(s) of the machine's input(s). It is also the basis for incredibly powerful computational models. As you delve further into sequential logic you'll begin to see how essential and useful FSMs are, so it's important that you be able to create one at the hardware level (since you might not always have the luxury of a modern PC's internals). A FSM is composed from a collection of flip-flops (similar to the ones you used in Part 1 of the lab). This collection is often referred to as a register. State is represented by a unique value stored in this register. Connecting the outputs of the state register's flip-flops to the inputs is a web of combinational logic which incorporates the machine's inputs. So at each positive edge of the clock the combinational logic is evaluated and the state is physically changed.

You can imagine the horror of determining the "next state" logic for a large FSM with many inputs. Thankfully there's a 'programming' language that vastly simplifies this task: Verilog. (Programming is in quotes because Verilog is actually a hardware description language, not a programming language like Java or C).

IMPLEMENTATION

There are several Verilog constructs that are useful in implementing a FSM:

Always blocks: The always block is special for two reasons: 1. It is re-evaluated every time the value of anything on its *sensitivity list* changes and 2. It allows you to do if-else logic and case statements.

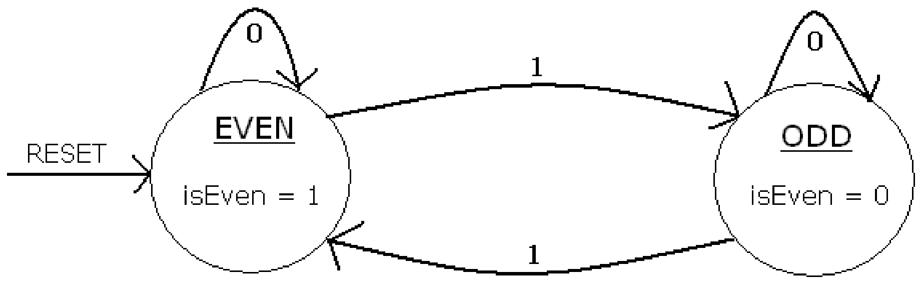
posedge CLK: The Verilog keyword **posedge** allows the variable following it to trigger the re-evaluation of the always block whose sensitivity list contains it when the value of that variable (which is really a wire) goes from 0 to 1. Similiarly **negedge** will trigger the re-evaluation when the variable goes from 1 to 0.

<= vs. = : Both '<=' and '=' are assignment operators. '<=' is commonly referred to as a "non-blocking assignment operator" and '=' a "blocking assignment operator". '<=' is very useful for assignment inside always blocks which are doing sequential logic (that is, always blocks which are using something like a clock as a trigger to update a stored value). '=' is perfectly suitable for assignment inside an always block doing purely combinational logic. '<=' differs from '=' by order of execution. At a timing event, all '=' assignments are completed BEFORE any '<=' assignments. When designing FSMs this ensures that the combinational logic to determine next state is performed before any attempt is made to update the machine's state. But, if you only remember one thing: Use '<=' in sequential always blocks and '=' in purely-combinational always blocks and never mix '<=' and '=' assignments inside the same always block!

When implementing a FSM in Verilog it is often easiest to use two always blocks. The first always block will perform the sequential tasks for the machine: doing synchronous reset or updating the state. The second block will perform all the combinational logic to determine the value of the machine's output(s) and what state the machine will transition to next based on the current state and the current value of the machine's inputs. Both of these always blocks will be in the same Verilog module. Since they will be re-evaluted anytime a value changes on their sensitivity lists, the state will be updated on each positive clock edge to the computed next state and the output value(s) will change concurrently.

AN EXAMPLE

Now that you have an idea of how to go about implementing a FSM here's an example to guide your thoughts:

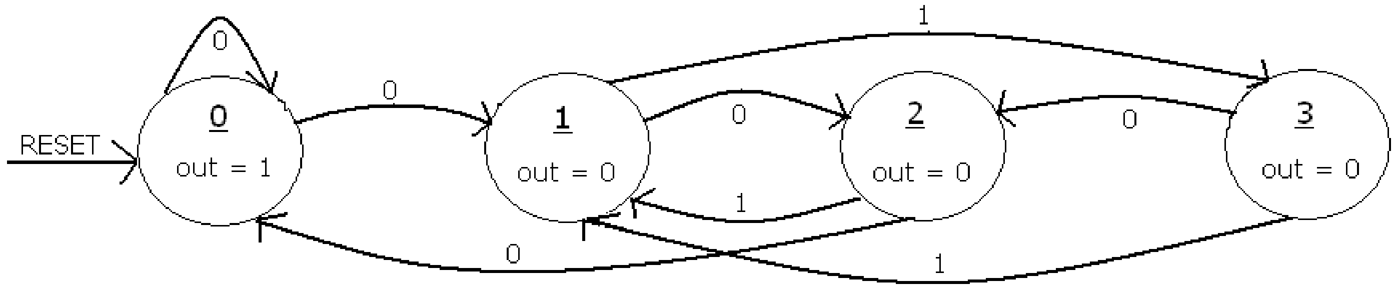


This FSM asserts its output whenever it has seen an even number of 1's. Otherwise the output is low.

It is also is an example of a Moore-style FSM since the output is assigned inside the states and the output values are only dependent on the current state. This is in contrast with a Mealy-style FSM where the output would be assigned on the transitions between states and the output values are dependent on both the current state and the input values.
Here is the well-commented Verilog code for this FSM: evenOddChecker.v.

YOUR TURN

Now you get the chance to practice implementing a FSM using Verilog. You need to describe the following Moore-style FSM which asserts its output when the bitstream it has seen so far is divisible by four, where the oldest bit seen is the most significant and the newest bit the least.



Resources

- FSM.v - A skeletal Verilog code file to help you get started.
- fpga_clock_sim.v - A clock simulator.
- FSM_tf.v - A test fixture for your completed FSM module.
- test.bde - A connected test diagram.

Download the above files to a location of your choice. Create a new Design in ActiveHDL and use 'Add Existing File' to add each of the above 4 files. Remember to check the box to make a local copy in this design's src directory! Then complete the code in FSM.v based on the above image. When you're done, set the FSM_test as your top-level module and hit the Play button to run the tests. If you see the "All Tests Passed!" message ask your TA for ...

CHECK-OFF

Demonstrate your ability to describe a FSM using Verilog to your TA by showing a successful iteration of the test fixture.

Comments to: cse370 - webmaster@cs.washington.edu