Exercise 2: State Machines

Overview

In this exercise, we will be building a simple counter for the number of cars in a parking lot based on the status of two signals. It will involve working with behavioral Verilog as well as implementing a state machine, which a very common design pattern used in hardware.

Mechanical Goals

By the end of going through this exercise, you should:

- Be comfortable implementing both sequential and combinational logic in synthesizable behavioral Verilog
- Know how to implement a state machine with good style practices

Conceptual Goals

- Understand the difference between combinational and sequential logic
- Understand the concept of a state machine and how to design a basic state machine

Problem

Implement the hardware to maintain a count of cars in a parking lot based on the inputs from two sensors. There are two sensors, an outer sensor and an inner sensor. A car has entered after the pattern {`outer, inner`} = {1,0; 1,1; 0,1; 0,0} has been seen. A car exiting is the pattern {`outer, inner`} = {0,1; 1,1; 1,0; 0,0}. Cars may stop, but they will not change direction, so they may spend multiple cycles at any part of the pattern, but they will not go backwards in the pattern. The counter should not be changed until the full sensor pattern has been seen.

Files

- `lot_counter_pkg.sv`: Used for definitions of constants
- `lot_counter_state_logic.sv`: The bulk of the logic should be implemented in this file. This will contain all the combinational logic for the state machine

- `lot_counter_state_regs.sv`: This is the file that contains all the sequential logic needed. Do not change this file. It has a register to store the current count of cars and a register to store the current state.
- `lot_counter_top.sv`: This file should instantiate both the state logic and the state regs and is the top-level wrapper for simulation

Steps

Step 0: Design your state machine
Outline the states that you want and what each of them should do and then fill in the enum in `log_counter_pkg.sv`. The first state (`READY`) that the state machine will initialize to has already been provided.

Think about the states that you go through when you see the entry and exit patterns and what signals cause you to transition states. Also consider when to set signals going from the combinational logic file to the sequential logic file. There are formal diagrams to do this (see ASM charts), but whatever type of diagram or outline is going to work for you is good.

Step 1: Implement the combinational logic
In `lot_counter_state_logic.sv`, implement the logic for the state machine following the design pattern shown in the state machine section. Use a case statement and make sure that every signal used in the `always` block is assigned a value on every path through the code.

Step 2: Instantiate the state logic module
In `lot_counter_top.sv`, instantiate the `log_counter_state_logic` module and connect the ports to the appropriate signals.

Step 3: Build and test
As before, run `make build` to compile the design. Then `make run` to actually execute the tests. Running `make all` will do both steps.

Running the tests will produce a waveform for debugging in `logs`
`vlt_dump.vcd` as well as print out some state of the design. Feel free to edit `sim_main.cpp` to add more tests or print out different state of the design.