Name: Ramon Costa-Patel

Student ID: 2377344

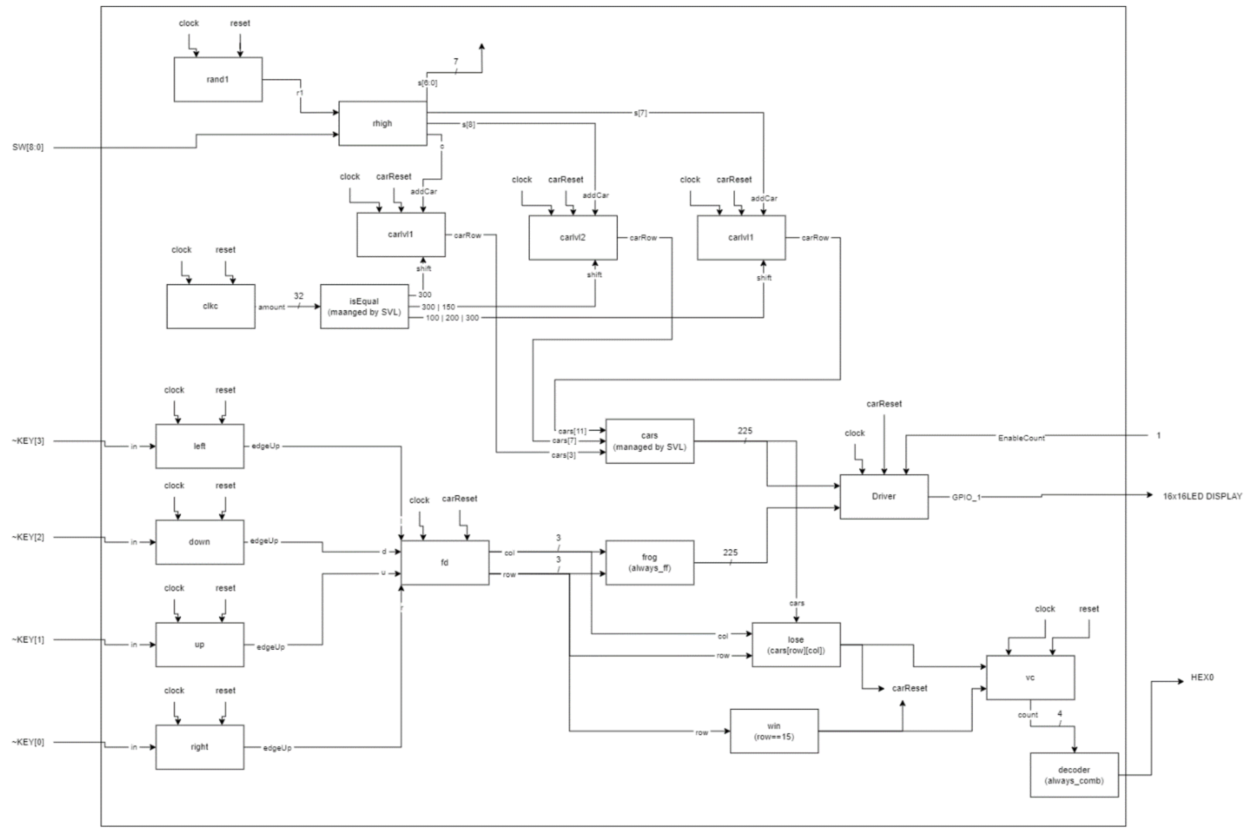
Frogger: User's Manual

Instructions

Frog: green ; Car: red

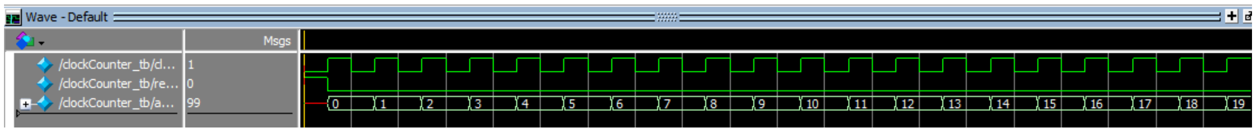
1. Win condition: move the frog to the other side of the board
2. Lose condition: get hit by a car
3. Reset: SW[9]
4. Move the frog:
    a. KEY[3] → move left
    b. KEY[2] → move up
    c. KEY[1] → move down
    d. KEY[0] → move right
5. The HEX0 display shows the number of consecutive wins you have. When you lose or reset, the counter is set back to 0.
6. You can change the difficulty of the game at any point using SW[8:0]. Turning switches on increases the difficulty:
    a. The left-most switches are a bigger jump in difficulty
    b. the more turned on switches you have the harder the game is

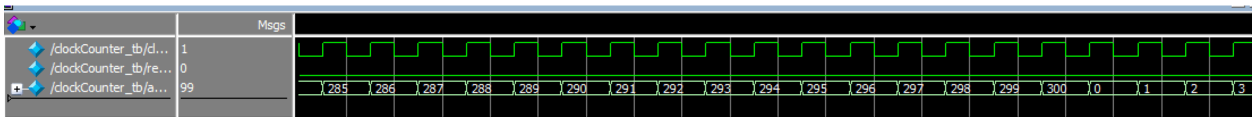
Block Diagram



There are 3 rows of cars (carlvl1, carlvl2, carlvl3) that go into the LEDDriver (Driver). Carlvl1 shifts every 300 clockcycles, carlvl2 does so every 150, and carlvl3 does so every 100 clock cycles. There are 4 edge detectors for the user input (left, down, up, right), which all go into the fd module that keeps track of the frog's column and row. That information is used for the lose and win conditions, and to update the position of the frog on the Driver. Finally the win and lose conditions go into a counter (vc) that counts the number of consecutive wins and displays it on the HEX0 display.

ClockCounter Modelsim (clkc)

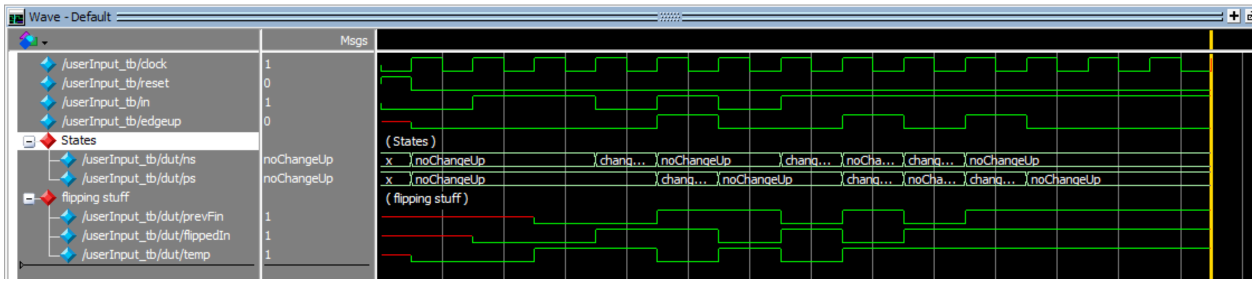




This counter counts how many clock cycles have passed (we can see the number increase every clock cycle) and resets when the number of clock cycles reaches 300.

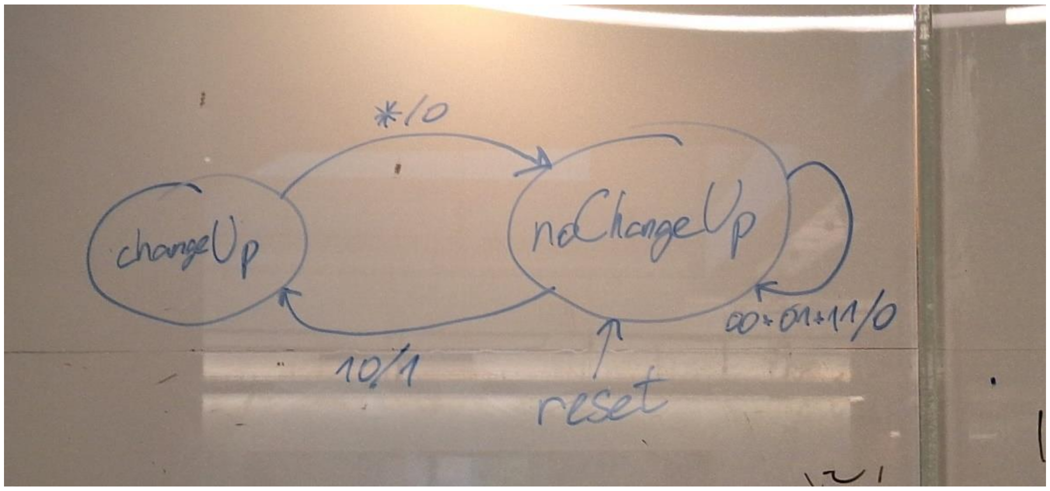
UserInput module (left, down, up, right)

ModelSim Simulation:


Wave - Default
Msgs
/userInput_tb/clock 1
/userInput_tb/reset 0
/userInput_tb/in 1
/userInput_tb/edgeup 0
States
/userInput_tb/dut/ns noChangeUp
/userInput_tb/dut/ps noChangeUp
flipping stuff
/userInput_tb/dut/prevFin 1
/userInput_tb/dut/flippedIn 1
/userInput_tb/dut/temp 1
(States)
x noChangeUp chang... noChangeUp chang... noCha... chang... noChangeUp
x noChangeUp chang... noChangeUp chang... noCha... chang... noChangeUp
(flipping stuff)

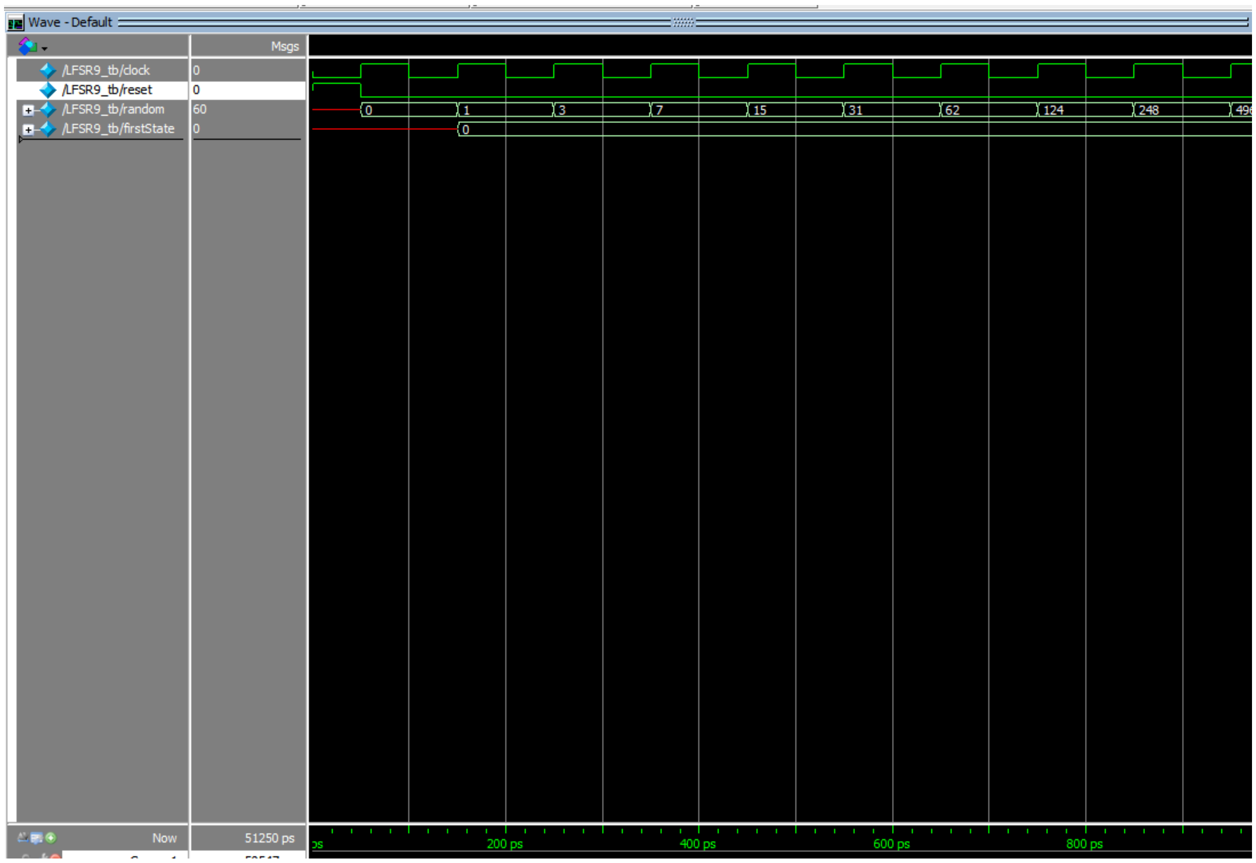
This module passes an input (a button press in this case) through 2 DFFs and acts as an edge detector of the input (it’s only high for 1 clock cycle for every button press, no matter how long). Between the input and the output there’s a delay of 3 clock cycles (2 for the DFFs, and 1 for the FSM itself to update). To detect the edge, this module stores the input value in the previous clock cycle and passes both current and previous inputs into an FSM.

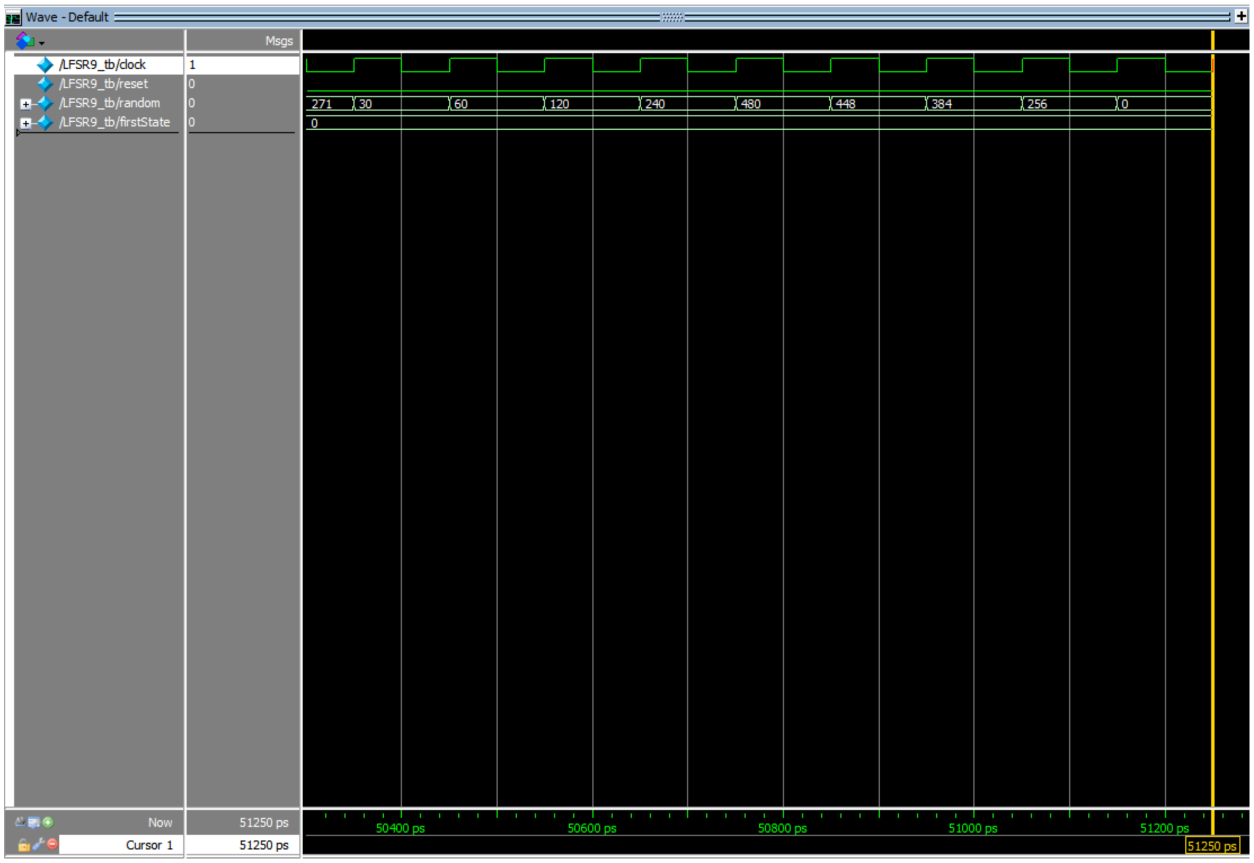
FSM State Diagram:



This FSM has 2 states: changeUp (a positive edge is detected, 1), and noChangeUp (no positive edge is detected, 0). There are 2 1-bit inputs, represented in the following order: current user input, and previous user input. The FSM changes from changeUp to noChangeUp in the next clock cycle (regardless of inputs). To change from noChangeUp to changeUp, the current input must be high and the previous must be low (positive edge). Otherwise, the state doesn’t change. The “reset” state is noChangeUp.

9-bit LFSR Modelsim (rand1)

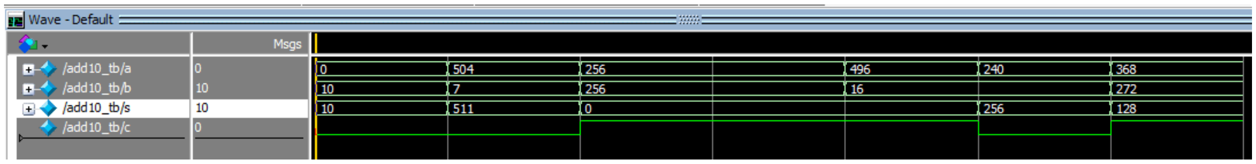




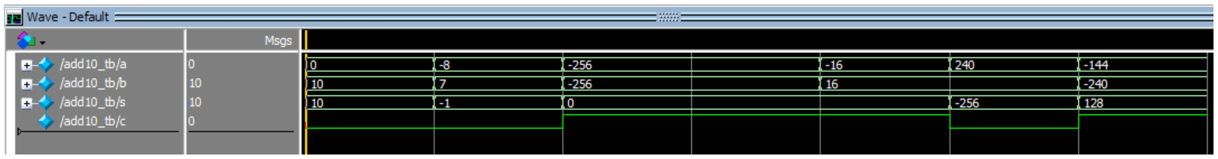
The simulation for this LFSR keeps going until the current output is the same as the output from the first state (stored in firstState). The LFSR states that are non-X while firstState isn't X is the number of different states the LFSR has. We see that firstState is non-X from 150ps to 51250ps. Since each clock cycle is 100ps, we can calculate the number of different states for the LFSR: (51250-150)/100 = 511 different states. Said otherwise, in this LFSR a state reappears on the 512th cycle.

9-bit Adder Carry-out Modelsim (rhigh)

Unsigned encoding


Wave - Default
Msgs
/add10_tb/a 0 | 0 504 256 496 240 368
/add10_tb/b 10 | 10 7 256 16 272
/add10_tb/s 10 | 10 511 0 256 128
/add10_tb/c 0

Signed encoding (Two's complement)

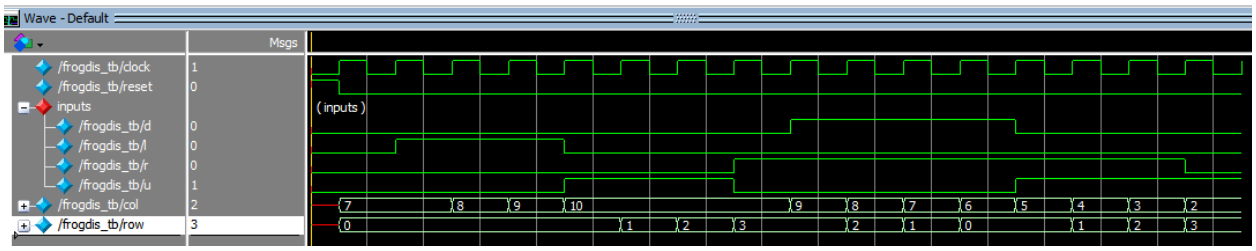

Wave - Default
Msgs
/add10_tb/a 0 | 0 -8 -256 -16 240 -144
/add10_tb/b 10 | 10 7 -256 16 -240
/add10_tb/s 10 | 10 -1 0 -256 128
/add10_tb/c 0

Inputs a and b are added and give the sum s and carry-out c. This simulation covers the 6 required cases:

1. Addition with one input being 0: 0+10
2. Addition with result 511: 504+7
3. Addition whose result is 0 (not 0+0): 256+256, -16+16
4. Example of unsigned overflow: 256+256
5. Example of positive signed overflow (pos+pos=neg): 240+16
6. Example of negative signed overflow (neg+neg=pos): -144-240

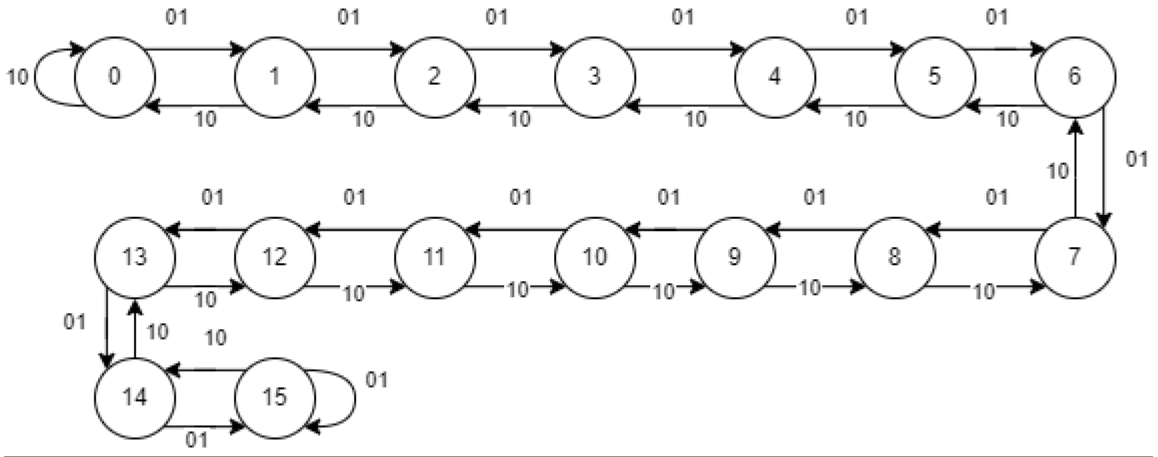
FrogDis (fd)

ModelSim Simulation:



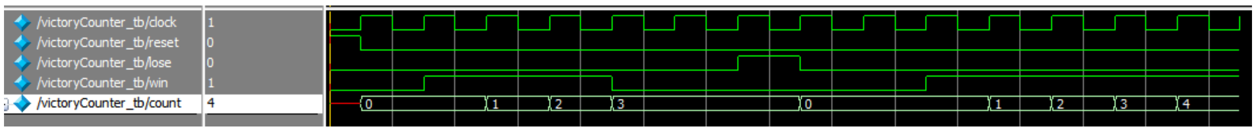
The high rows are up and the low rows are down. The high columns are left and the low columns are right. There is a 1 clock-cycle delay between the input and the updated value of col/row. We can see that row increases with the input u (up) and decreases with d (down). We also see that col increases with l (left) and decreases with r (right). On reset we also see that row is 0 (first row) and col is 7 (approximately the middle column).

State Diagram:



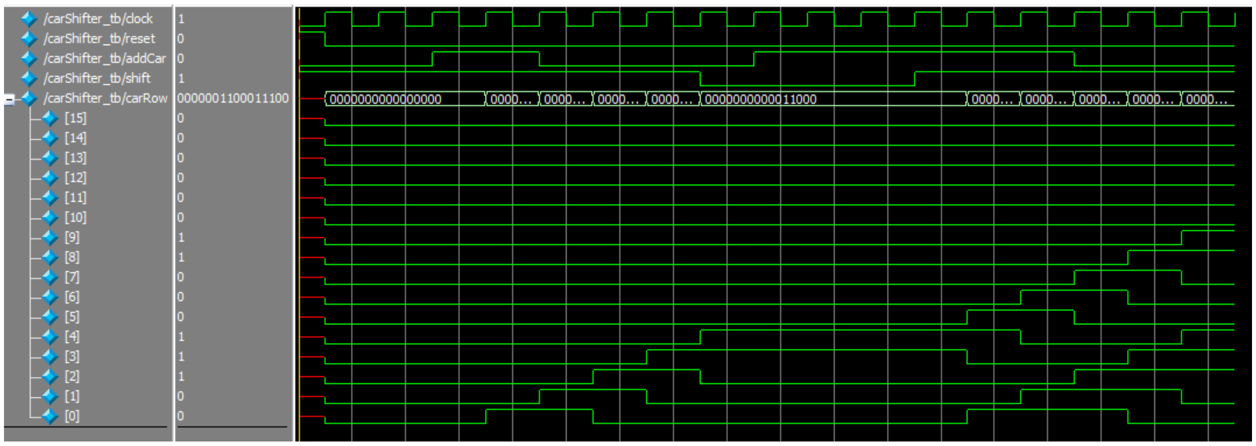
This is the state diagram for both col and row. For col the 2-bit input represents rl (right-left), and for row the input represents du (down-up). Although they are not shown, the reset for row is at the 0 state and the reset for col is at the 7 state. If both inputs are high the behavior is “don’t care”, so those transitions aren’t shown. The output for each transition is the number of the state it’s going into.

VictoryCounter Modelsim (vc)



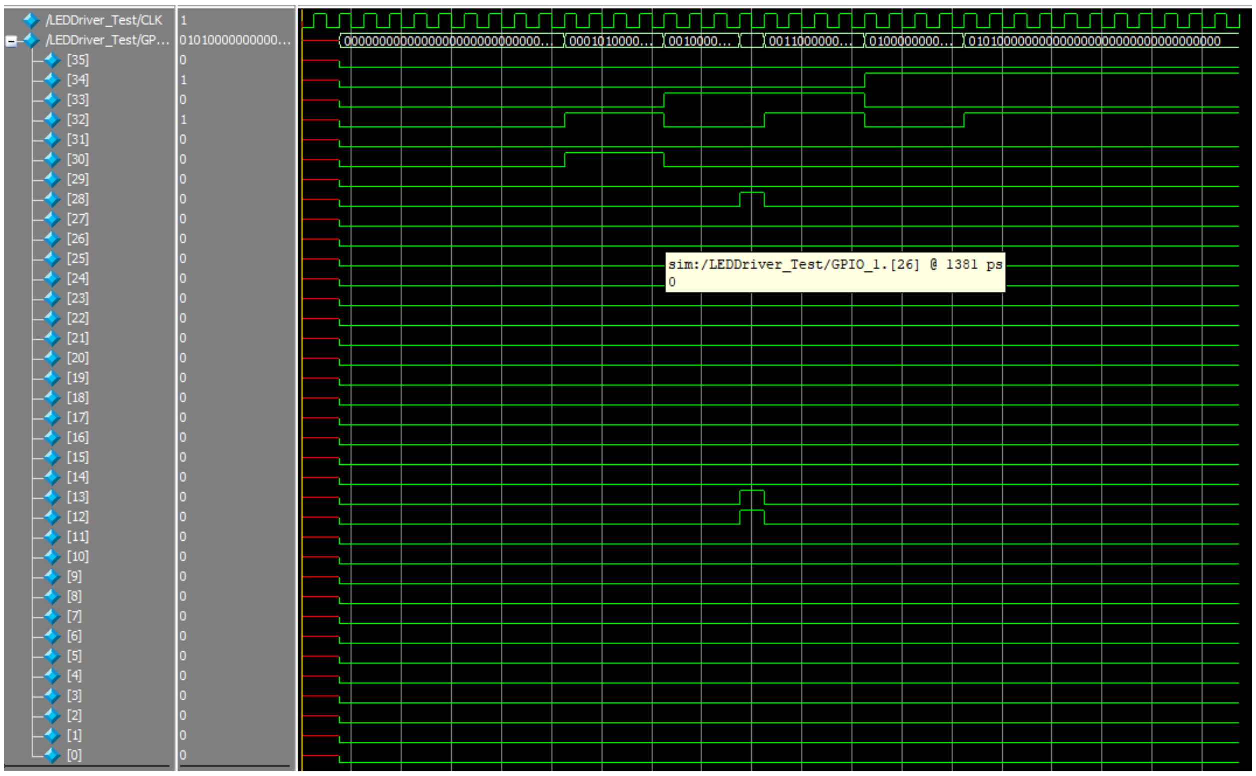
This counter only adds 1 to count when win is high, and resets to 0 when either lose or reset is high.

CarShifter ModelSim (carlvl1, carlvl2, carlvl3)



This is a shifter that shifts its bits left every clockcycle in which shift is high. Instead of a combination of XNORs, this shifter appends the value of addCar to the right on each shifting clockcycle.

LedDriver ModelSim (Driver)



```
/LEDDriver_Test/CLK      1
/LEDDriver_Test/GP...    01010000000000...
    [35]  0
    [34]  1
    [33]  0
    [32]  1
    [31]  0
    [30]  0
    [29]  0
    [28]  0
    [27]  0
    [26]  0
    [25]  0
    [24]  0
    [23]  0
    [22]  0
    [21]  0
    [20]  0
    [19]  0
    [18]  0
    [17]  0
    [16]  0
    [15]  0
    [14]  0
    [13]  0
    [12]  0
    [11]  0
    [10]  0
    [9]   0
    [8]   0
    [7]   0
    [6]   0
    [5]   0
    [4]   0
    [3]   0
    [2]   0
    [1]   0
    [0]   0
```

```
000000000000000000000000000...  0001010000...  0010000...  0011000000...  0100000000...  010100000000000000000000000000000000
```

```
sim:/LEDDriver_Test/GPIO_1.[26] @ 1381 ps
0
```

This is the modelsim for the provided LedDriver file. It shows how the output to the physical pins of the FPGA changes based on what value needs to be shown. (This is provided and I can’t provide further explanation)

Testing

To test the overall system (top level module), besides the test benches for the sub-modules, I tested directly on the FPGA by playing the frogger game and seeing if it worked properly. During this testing I also checked for edge cases to make sure the behavior shown was the one intended.

Hours Spent

Total hours spent on lab: 24