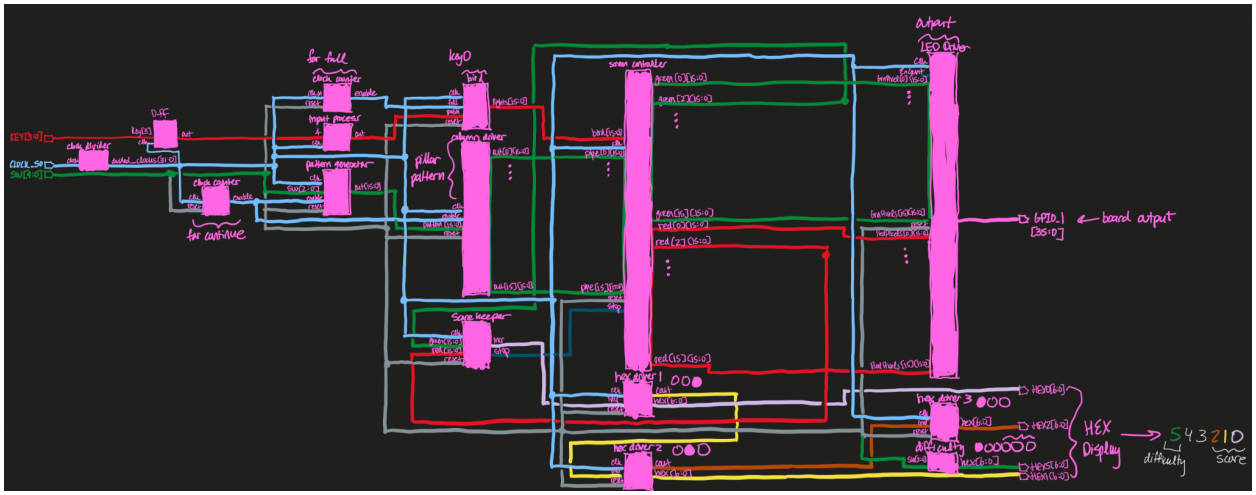
Audrey Yip
2121871
Lab 8

Q1

For someone to use the design, there is a reset switch on SW9 that will restart the game (initially the 16x16 LED extension board will have GAME OVER spelled out). KEY3 is used to click a red dot (the bird) to go through the green pillars which are generated by a random pattern. The HEX5 display shows the difficulty, ranging from 0 to 1. SW0 to SW3 can be switched on or off to toggle on the difficulty (which makes the distance between the green pillars closer as the difficulty increases). The HEX0 to HEX2 display shows the score, which increments as the red dot passes a green pillar. If the red dot touches a green pillar or KEY0 is pressed when the red dot is at the very edge of the LED board, the game stops with GAME OVER.

Q2

Block Diagram of Entire System



We first have 3 inputs, KEY[3:0], CLOCK_50 (the 50 MHz clock), and SW[9:0]. We will only use KEY[0], SW[0], SW[1], SW[2], and SW[9]. First, the KEY input goes through the doubleFlip module so that it can handle the metastability. CLOCK_50 also goes through a clock divider to create lower frequency clock signals that will be used in modules that need a clk. From the doubleFlip, the input is processed in input_processing so that one click on the key signifies the bird moving up once (so it prevents holding the key as input). There are also two clock counter modules, one for the bird moving up and down and the other for the pillars that move towards the bird. The bird will be on a faster clock compared to the pillars. The clock counter of the bird will output an enable signal that passes through the bird module that also has the stabilized user input. The bird module will determine if the bird is falling down or flying up, depending on if there is user input from KEY0. The output of the bird module connects to the board_controller where during the game it updates the bird's position (to the LEDDriver) and when it's game over it

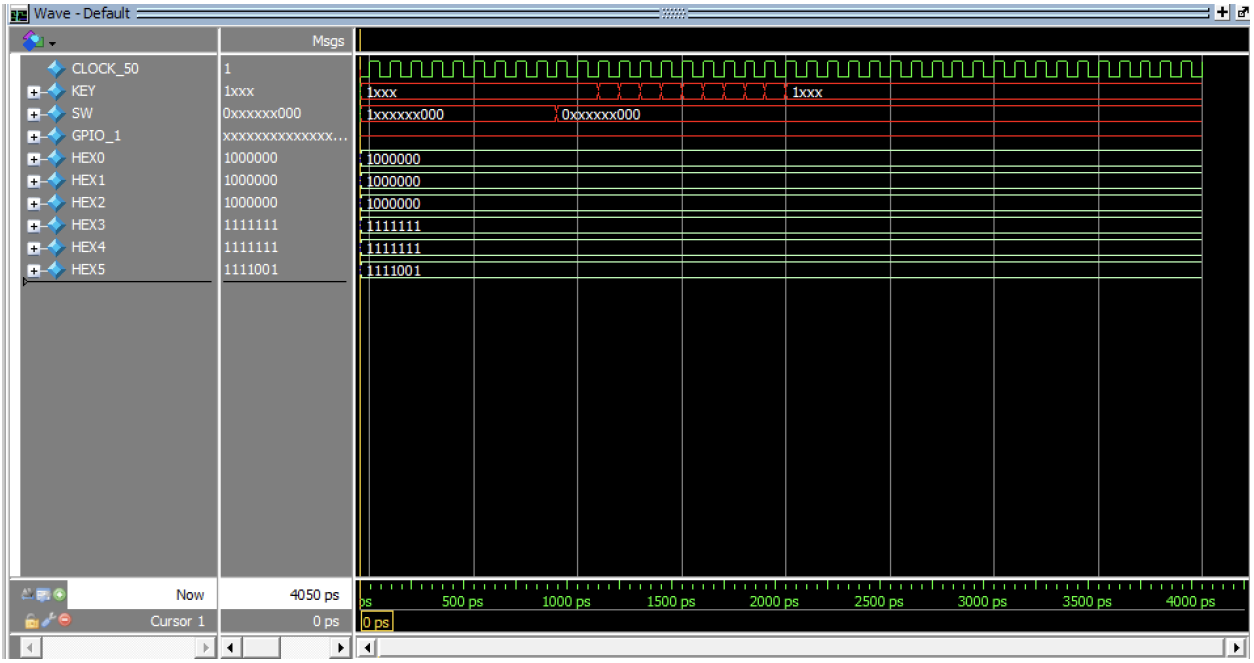
stops and outputs GAME OVER on the board. The clock counter of the pillar will also output an enable signal (different to the bird) that passes through the pattern_generator module, which will use an LFSR to generate a “random” pattern for the pillar (to which the gap of each pillar will be consistently 5 spaces apart). The random pattern is transferred to the column_driver module which sends the pattern from the pattern_generator to both the next driver so that the pillars shift to the left as time passes. The column_driver also transfers the pattern to the board_controller, which has the start state (the bird by itself at the center of the LED board) and the game over state. The board_controller output for the red and green pixels then go through the LEDDriver which outputs to the 16x16 board. To keep track of score, it has input from the green pixels on the fourth column and the red pixel (that is stationary going up and down only on the fourth column of the board). If the red pixel is in a position where the green pixel is not on, then the score is incremented (sent to the single_hex_display module). If the red pixel is in a position where the green pixel is on, then there is a stop switch that will stop the game and output the GAME OVER state on the board.

Q3

Top-Level Module

This module defines the inputs and outputs for the DE1 SoC board. It runs the whole flappy bird design to be output on the DE1_SoC. The inputs are KEY[3:0], SW[9:0], and CLOCK_50, where KEY[3] and SW[2:0] are used for user input. The outputs are HEX0, HEX1, HEX2 for score, HEX5 for difficulty level, and GPIO_1 for the 16x16x2 LED expansion board.

ModelSim



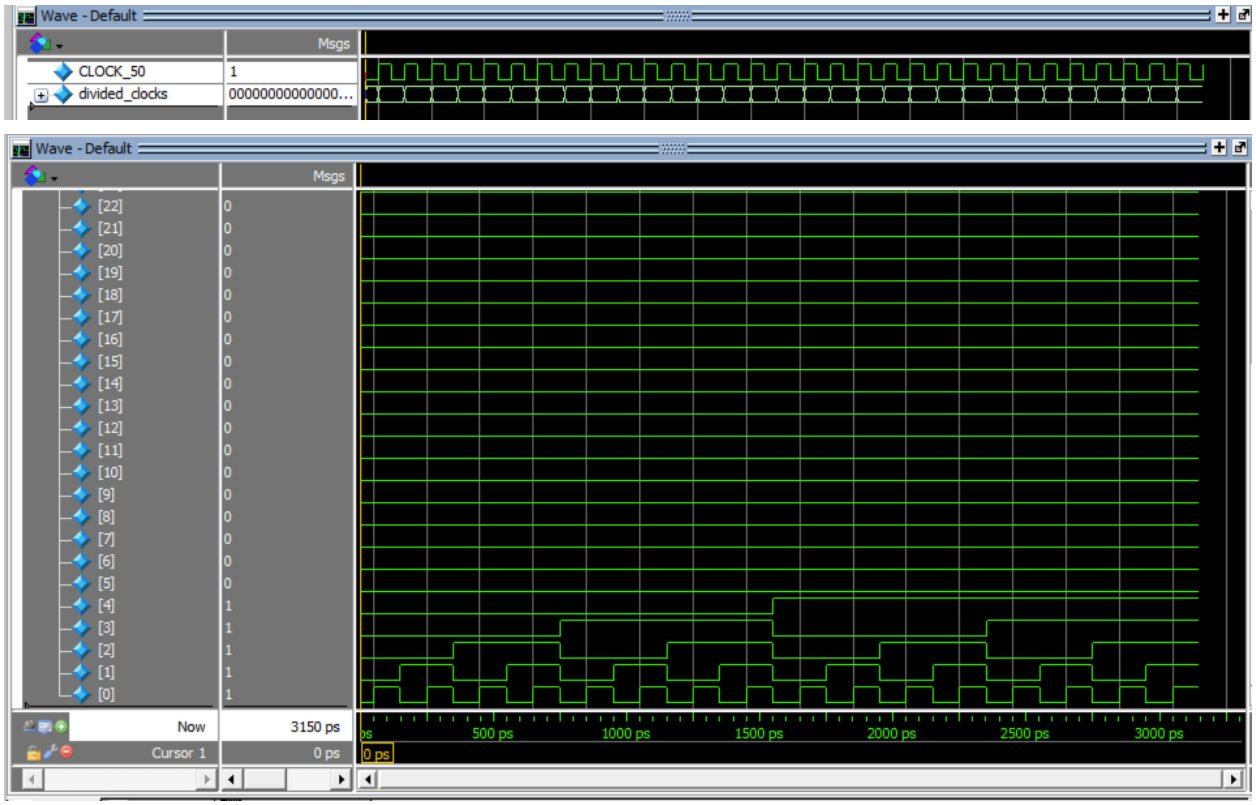
Unfortunately, since my top level module runs on multiple clocks (with the divided clock and two clock counters for the bird and pillars), it is quite complex and difficult to have the proper timing

for the testbench to produce output on the GPIO_1. So, I found other ways to test my design (mentioned in Q4).

Clock Divider Module

This module creates a lower frequency signal from an input clock source, where in this case the input is CLOCK_50. For the game, the frequency of the clock is clk[14], or a 1562 Hz clock signal that is used in all the modules that need a clk.

ModelSim

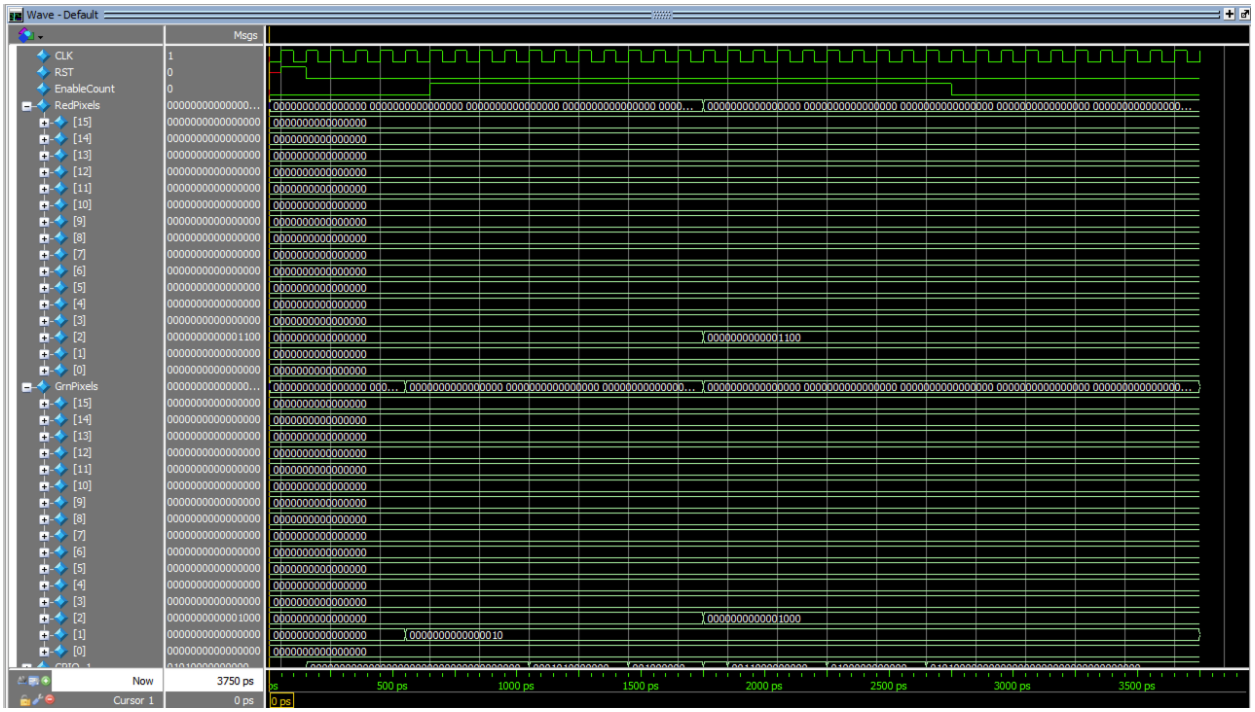


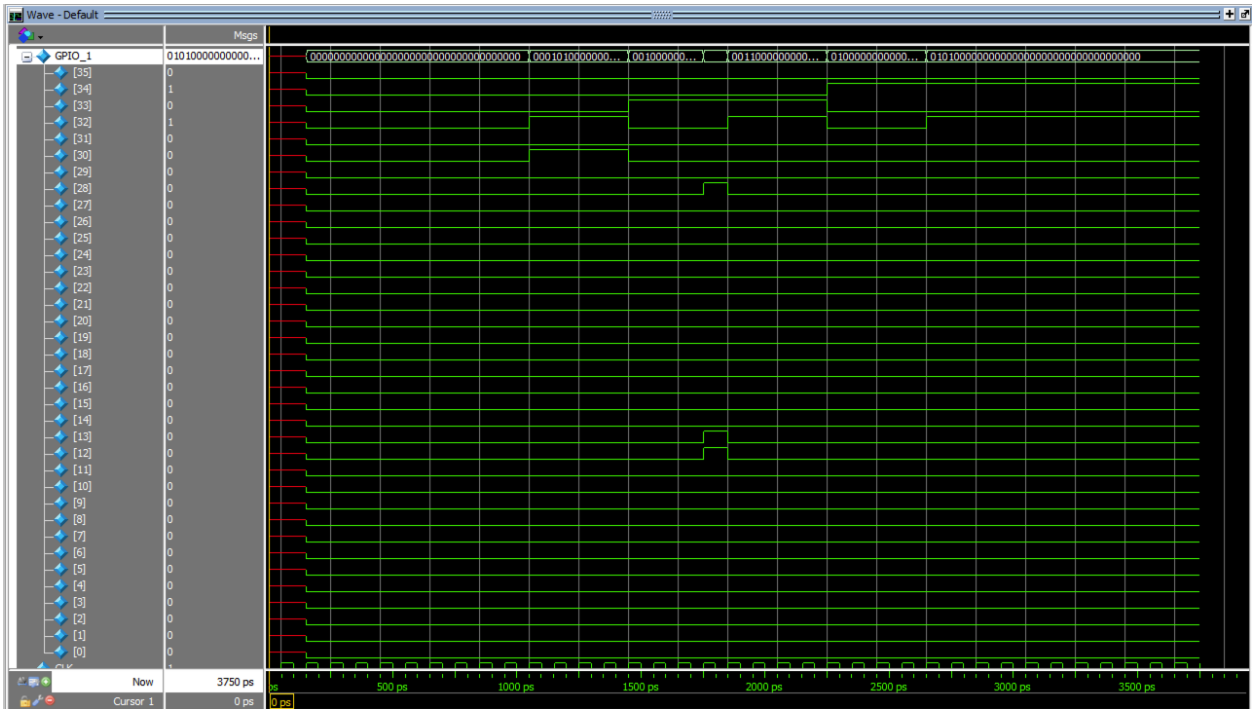
For the testbench, I had input be CLOCK_50 so the ModelSim above shows the possible clocks that have decreasing frequency for each increasing index of divided_clocks. It can be seen that the frequency decreases by 2 as the index of divided_clocks increments.

LED Driver Module

This module is the driver for the 16x16x2 LED display expansion board.

ModelSim



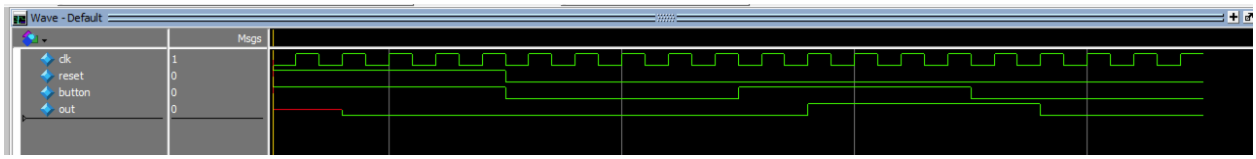


When EnableCount is set to false and the Reset is true, the lights are all off on the board (default). But when EnableCount is set to true, and Reset is false, we are able to change certain elements of the GPIO_1 to be green or red, like RedPixels[2][2] or GrnPixels[2][3].

D Flip Flop Module

This module handles metastability of user input with a double flip flop.

ModelSim

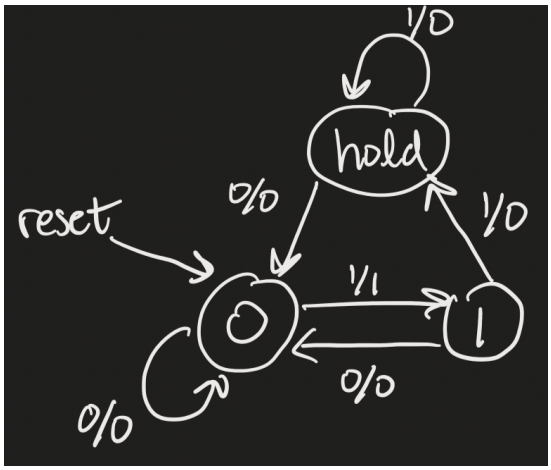


Since this is a double flip flop, when the button is pressed the output will occur 2 positive clk edges later. The reset works since when the button is pressed there is no output.

Input Processing Module

This module handles if the button is held, so input only counts when the button is clicked (the duration of the click does not count).

FSM state diagram of module



The state diagram has 3 states, each representing when the key is pressed (1) or not pressed (0), or held (hold). The reset/initial state is not pressed (0). When the input is 0, then the next state is always unpressed. When the present state is 0 and the input is 1, then the next state is 1. But if the present state is 1 and the input is 1, then the next state is hold so that it does not count as an input in the game.

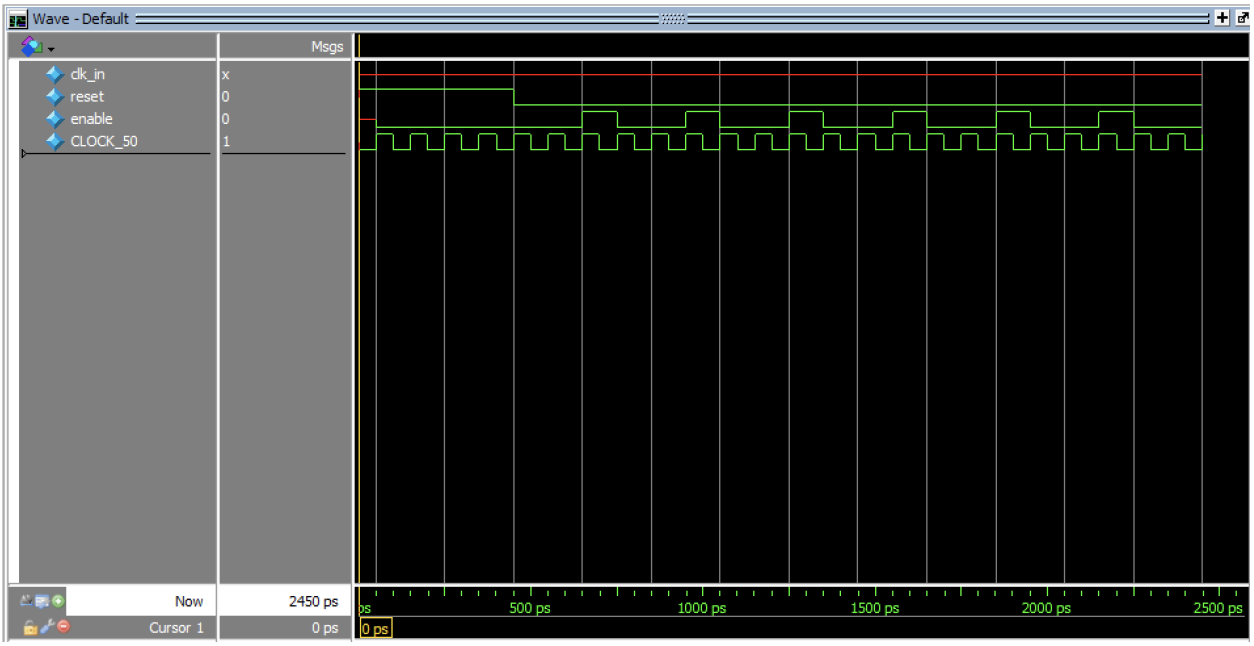
ModelSim

The ModelSim above has clk running with period 50ps. It tests the button input, which is when the key is pressed. The key/button is held for 4 periods, to which out is 1 when the key/button is unpressed, showing that the duration of the key being pressed is not accounted for in the output.

Clock Counter Module

This module slows down the clock to a desired rate, which is shown with the parameter CYCLE.

ModelSim

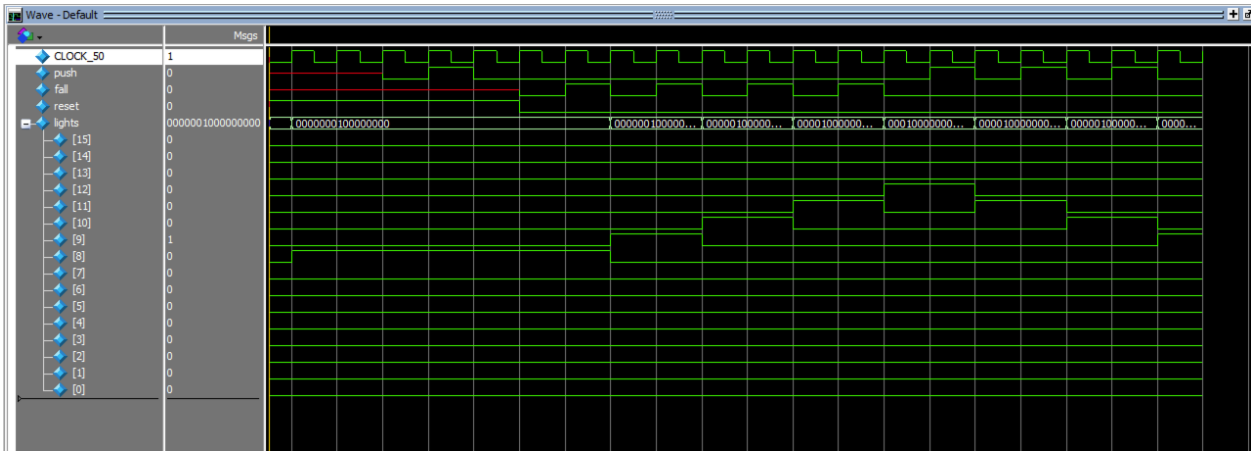


First the reset switch is tested to show that enable will not be true. Once the reset is false, with CLOCK_50 running, I had the parameter CYCLE at 2, so every 2 cycles of the clock the enable switch is on for 1 cycle, showing the decreased frequency.

Bird Module

This module implements the bird_light and bird_light_start modules where the position of the bird (vertically) depends on if user input KEY[3] is pressed (thus the bird flies up) or not pressed (thus the bird falls down).

ModelSim



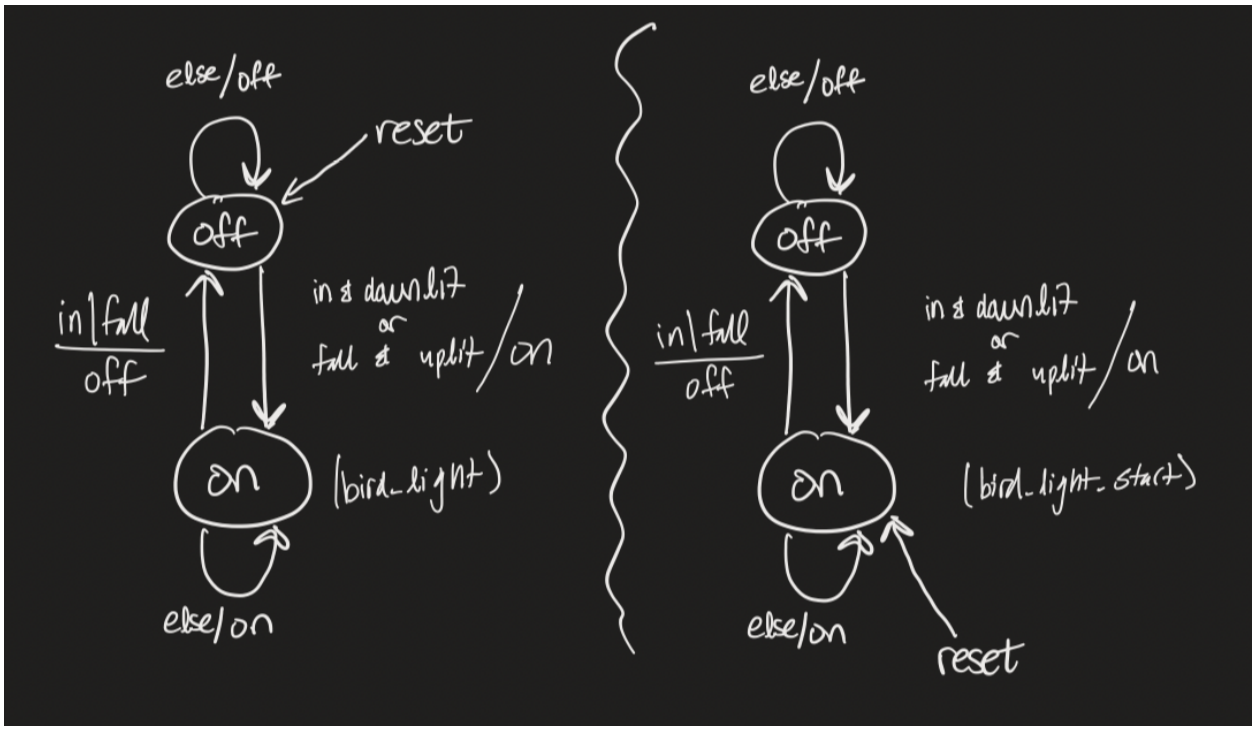
First the reset switch is tested to show that the bird is at its starting position at the vertical center position, regardless of if the input is pushed. Then, it is shown that when fall is true every other

clock cycle, the bird “falls” down to the lower vertical position (which is increasing in the lights[]). Then when input is pushed every other clock cycle, the bird “flies” up to the high vertical position (which is decreasing in the lights[]).

Bird Light and Bird Light Start Modules

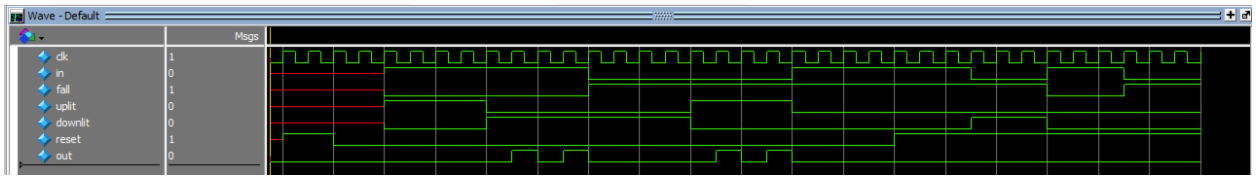
These modules use FSMs for when the LED light is on or off, depending on the bird’s position and reset switch.

FSM state diagram of modules

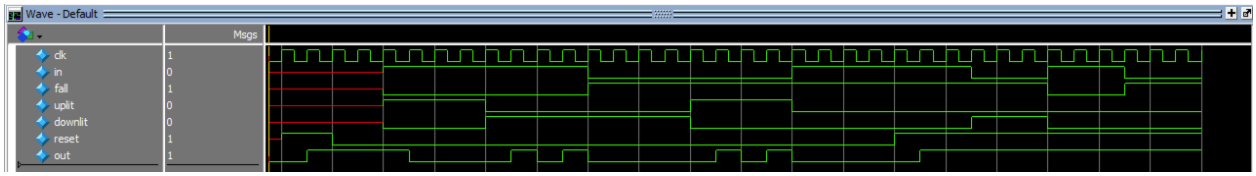


Above are the state diagrams for bird_light and bird_light_start. They both have 2 states, one for on, one for off. For bird_light, the reset makes the present state turn off, so that the bird_light_starts’s present state is turned on (since only one LEDR is on during the span of the game). Both state diagrams have the next state as off if the bird is falling or input is pressed, and the next state as on if the light below is on and input is pressed OR if the bird is falling and the light above is on. If there is a different input, (like a different key is pressed), then the present state/next state stays the same since it should not impact the LEDR.

ModelSim for Bird Light



ModelSim for Bird Light Start

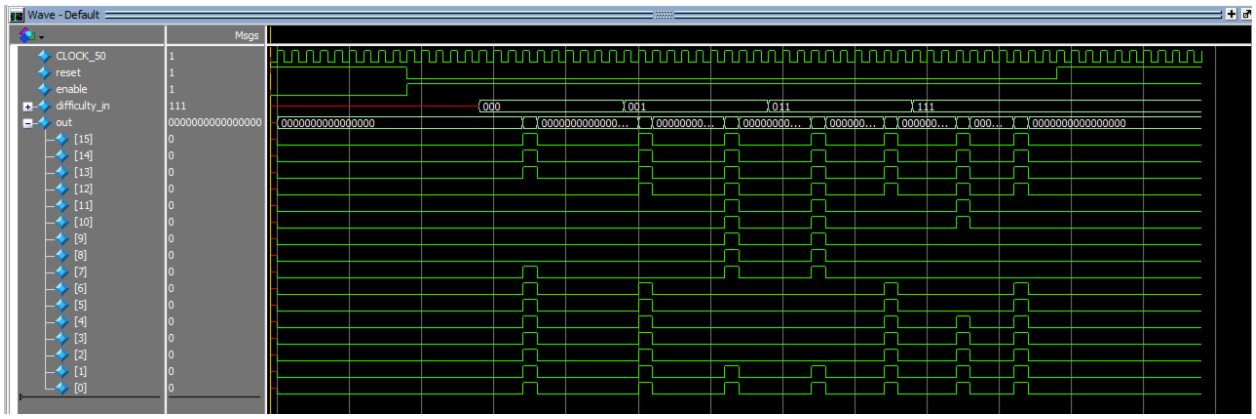


The ModelSim above has clk running with period 50ps. It first tests the reset switch (turned on and off), which makes out (if the light is on) true (indicating that the light at the center is the only light on, so the normal lights should be off).
From there, out is true when in | fall (input is pressed or not pressed), which then makes the next state false. out is also true when in & downlit OR fall & uplit, meaning the light is on when the light below this light is on and the input is pressed OR when the light above this light is on and input is not pressed. The reset switch is tested again, this time held on to show that the out is true (for the center light) regardless of the inputs.

Pattern Generator Module

This module utilizes a 9-bit LFSR to produce “random” output for the green pillars that will be used in the game. There is also a difficulty level, where the difficulty level 1 has 7 cycles between the pillars, level 2 has 5 cycles between the pillars, level 3 has 4 cycles between the pillars, and level 4 has 3 cycles between the pillars.

ModelSim

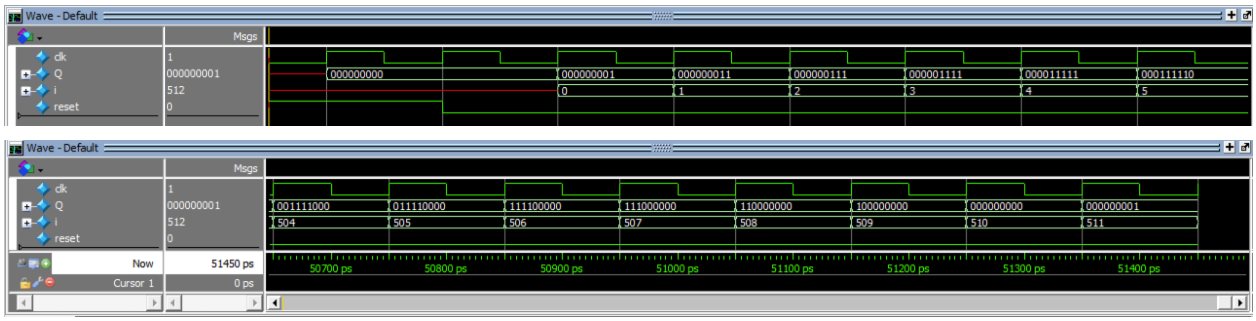


With the test bench running with CLOCK_50, reset is first tested to show that there is no output. Then, reset is off and enable is turned on. Starting with level 1 (so the difficulty has 000), there are 7 cycles that space between the out[]. Then at level 2 (difficulty 001), the spacing is decreased to 5 cycles. Then at level 3 (difficulty 011) the spacing is 4 cycles, and level 4 (difficulty 111) the spacing is 3 cycles. The spacing is also consistently at 5 consecutive spaces for the bird to pass through.

LFSR Module

This module is a 9-bit LFSR to “randomize” the green pillars that are produced in the game by producing a “random” number that may be used in the pattern generator module.

ModelSim

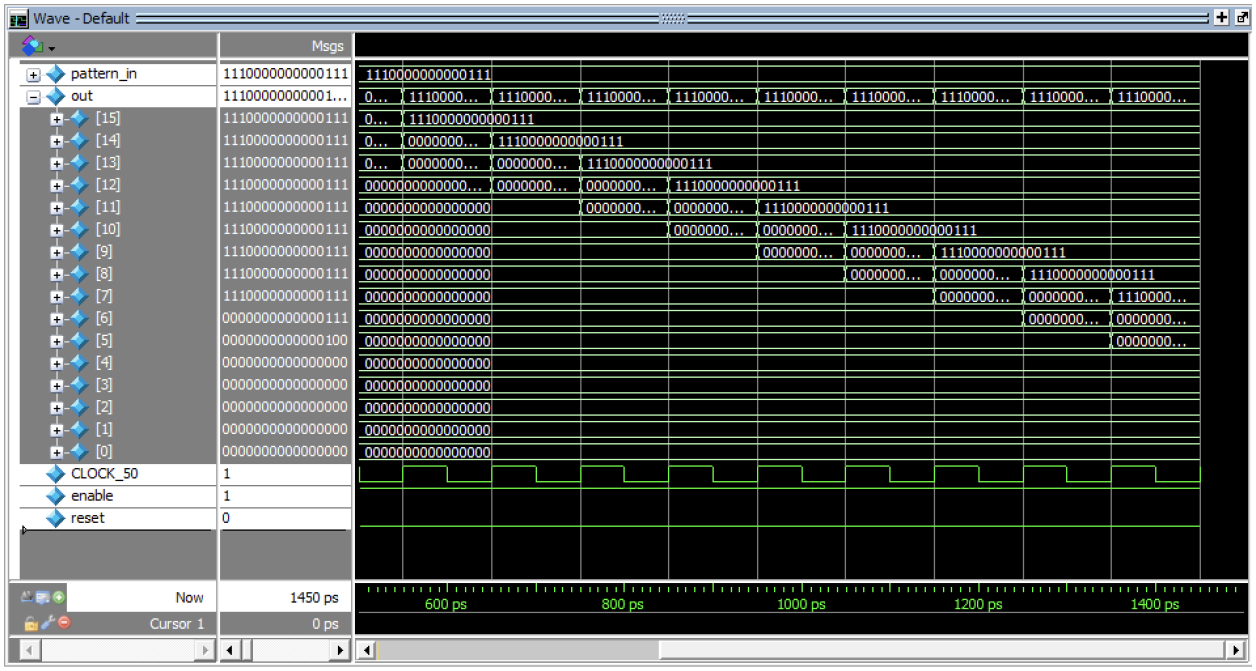


The ModelSim above shows the state bits changing as it shifts to the right, with the MSB being 0th bit XNOR 4th bit as shown in the lab spec for which bits to XNOR from when there are 9 state bits. With the initial state as 000000000, the 0th and 4th bit are 0, so 0 XNOR 0 produces output 1, which is added to the LSB and shifts the other bits to the left so the next state is 000000001. This is continued, as shown with 00001111, where the 0th bit is 0 and the 4th bit is 1, so 0 XNOR 1 produces output 0, which is added to the LSB and shifts the next state to be 00011110. The maximal sequence occurs at time 51200ps (if subtracting the reset test, that is 51100ps) , which is essentially the 2^9-1 = 511 states to reach the max sequence, 100000000, before going back to its initial state 00000000.

Column Driver Module

This module shifts the pillars to the left towards the bird to pass through at each positive clock edge. There may or may not be green lights on in the column driver (as there is space between each pillar in the game).

ModelSim

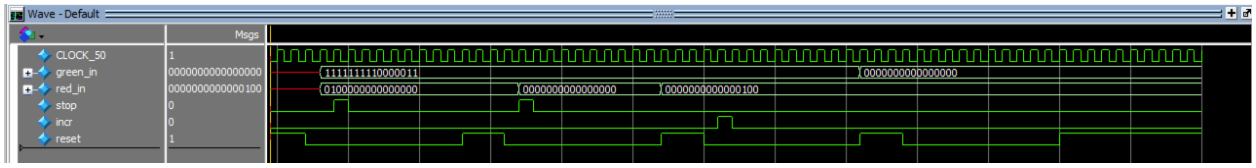


I did test the reset, which is shown to have all output be off (so all 16 bits are 0). Then, there are different inputs in pattern_in. It can be seen that every period on the clock, the pillars shift to represent the previous pattern_in inputs moving towards the “left” of the board.

Scorekeeper Module

This module keeps track of the score with an increment output when the bird passes the pillar and a stop output when the bird hits a pillar.

ModelSim

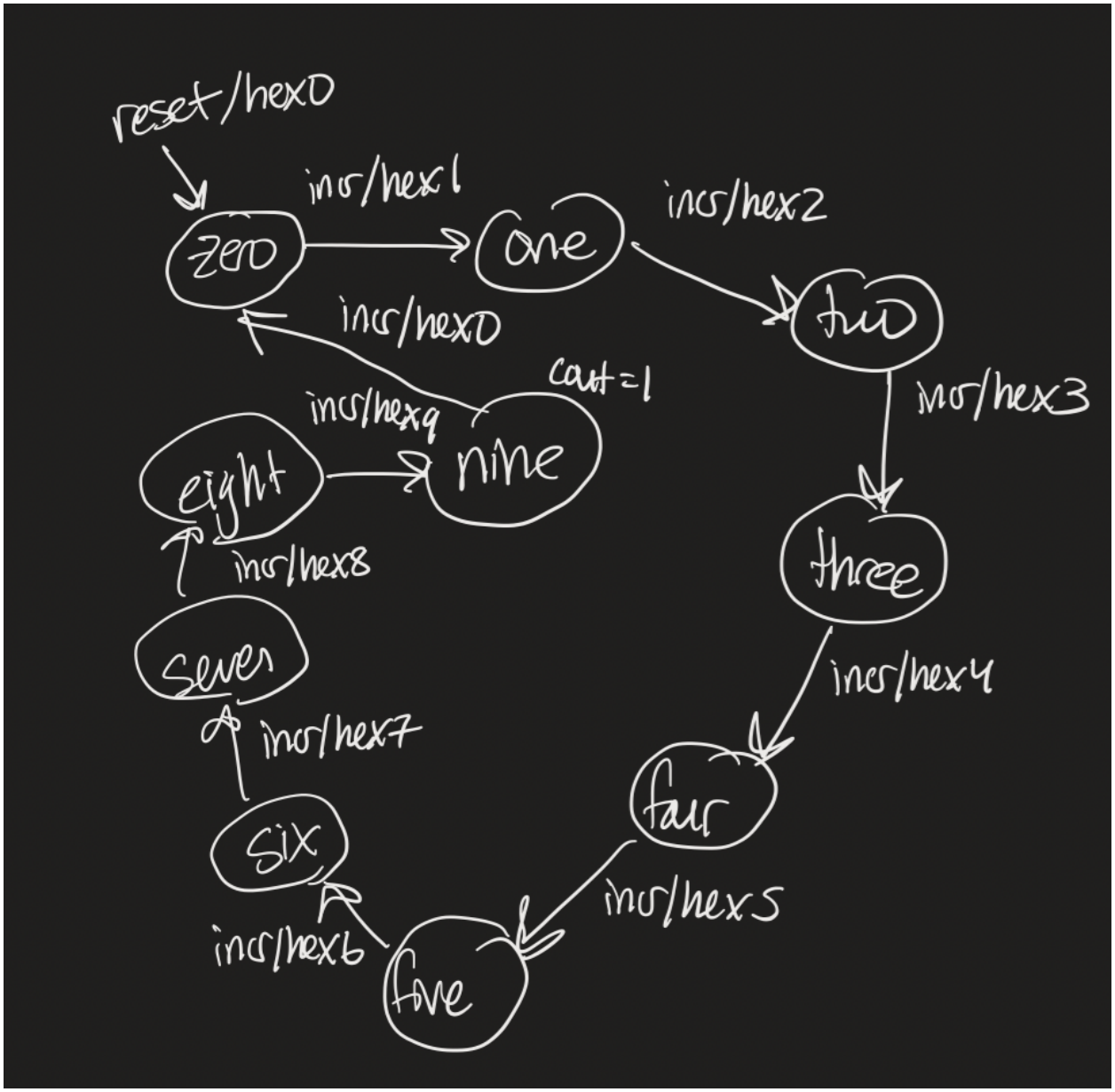


Above, the reset switch is tested to show that stop and incr are both false. Then, when there is green_in and red_in, the first input tests when the bird hits the pillar (which is true since the second bits of the green and red are both true), so the stop signal is true. Then, when red_in has input where the bird does not hit the pillar, so the third to last green bit is false but the third to last red bit is true. This has the incr signal true, which is expected.

HEX Score Display Module

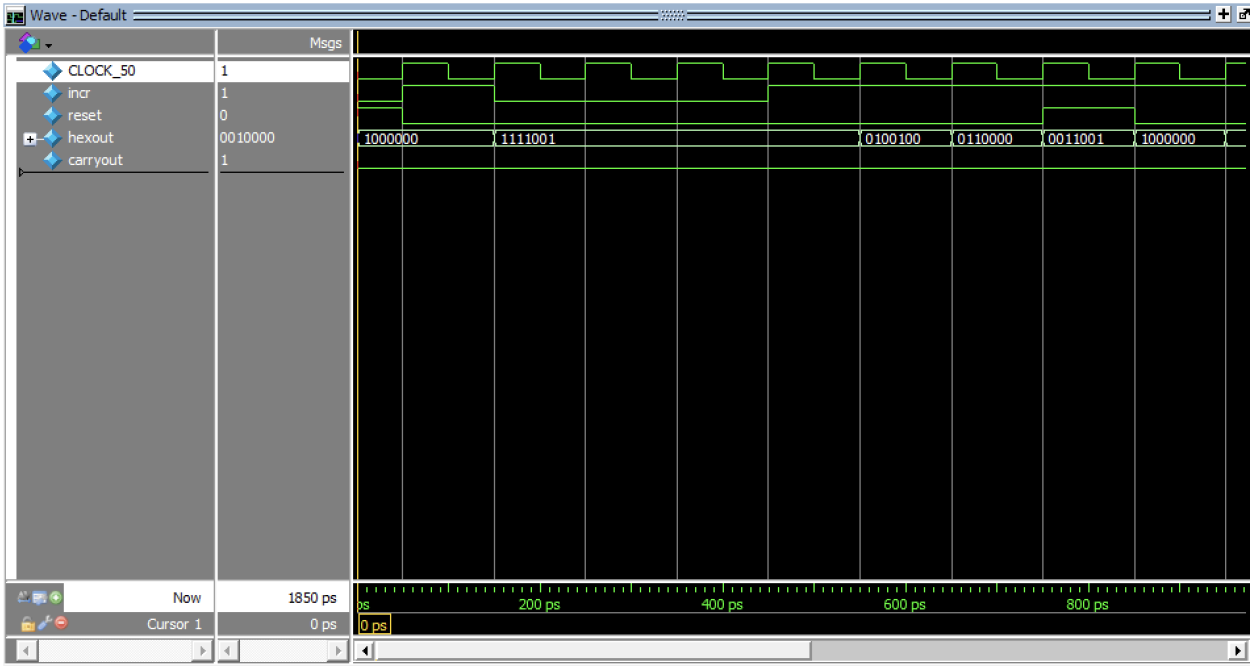
This module displays the score of the user, which increments from the scorekeeper module . HEX0 to HEX2 utilize the HEX score display module, so the max score is 999. There is also a carryout bit so when HEX0 reaches 9, then the next increment will make HEX0 0 and HEX1 1 so the score is 10.

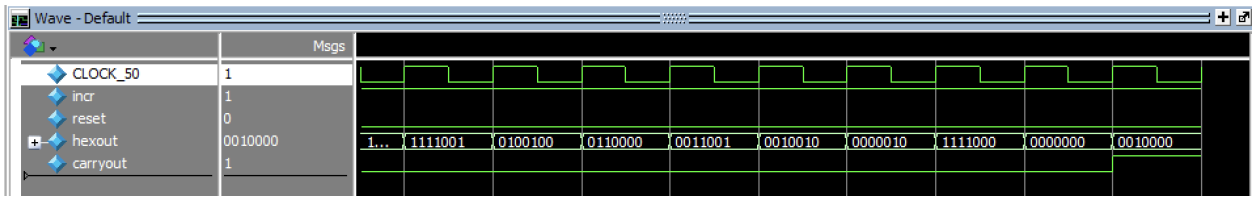
FSM state diagram of module



At reset, the state is 0 so the output on the HEX display is 0. Each time incr is true, the HEX display also increments (which is shown with the states one through nine). When state nine is reached and incr is true, then the next state is 0 with cout being true.

ModelSim





First the reset switch is tested to show that the output of the HEX display is 0. Then, when incr is true, the HEX display changes to be 1. When incr and reset are both true, reset has the priority so the HEX display changes to be 0. Over time, when incr is kept true past 9, the HEX display goes back to 0 but the carryout is true so that the next HEX display can be used.

Difficulty Display Module

This module uses HEX5 to display the difficulty level of the game, which is from 1 to 4.

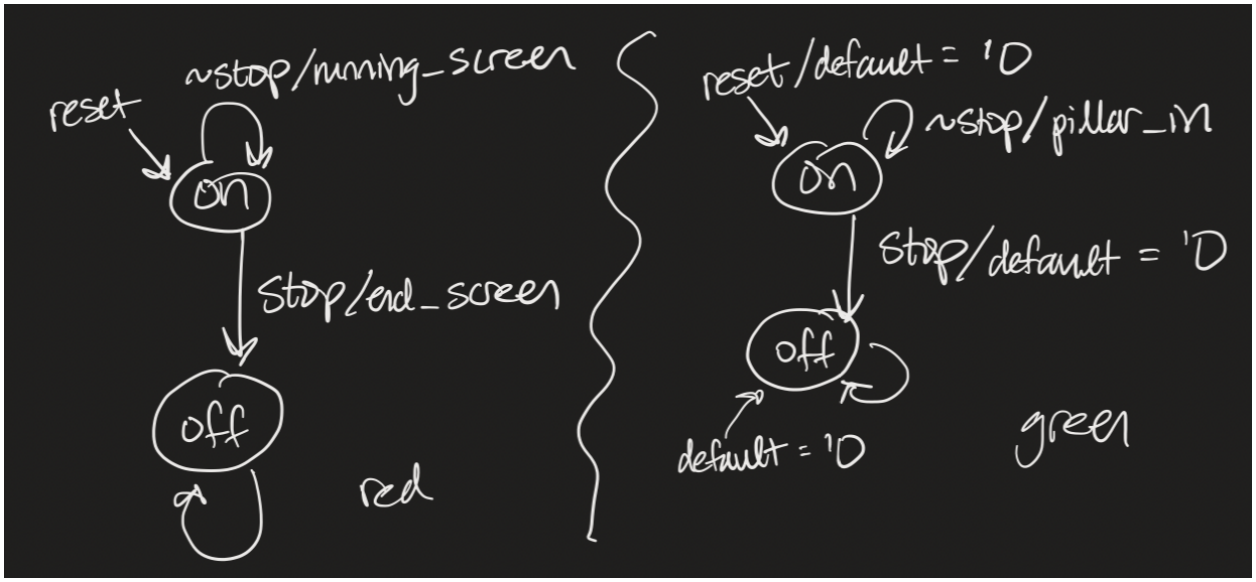
ModelSim

The default display when all inputs are off is 1 (which is level 1). With input SW[2:0], it can be seen that when one switch is on, the out on the HEX display is 2. When two switches are on, the out on the HEX display is 3. When all switches are on, the out on the HEX display is 4. The order the switches are turned on does not matter, we can think of a switch being on as incrementing the difficulty level by 1.

Board Controller Module

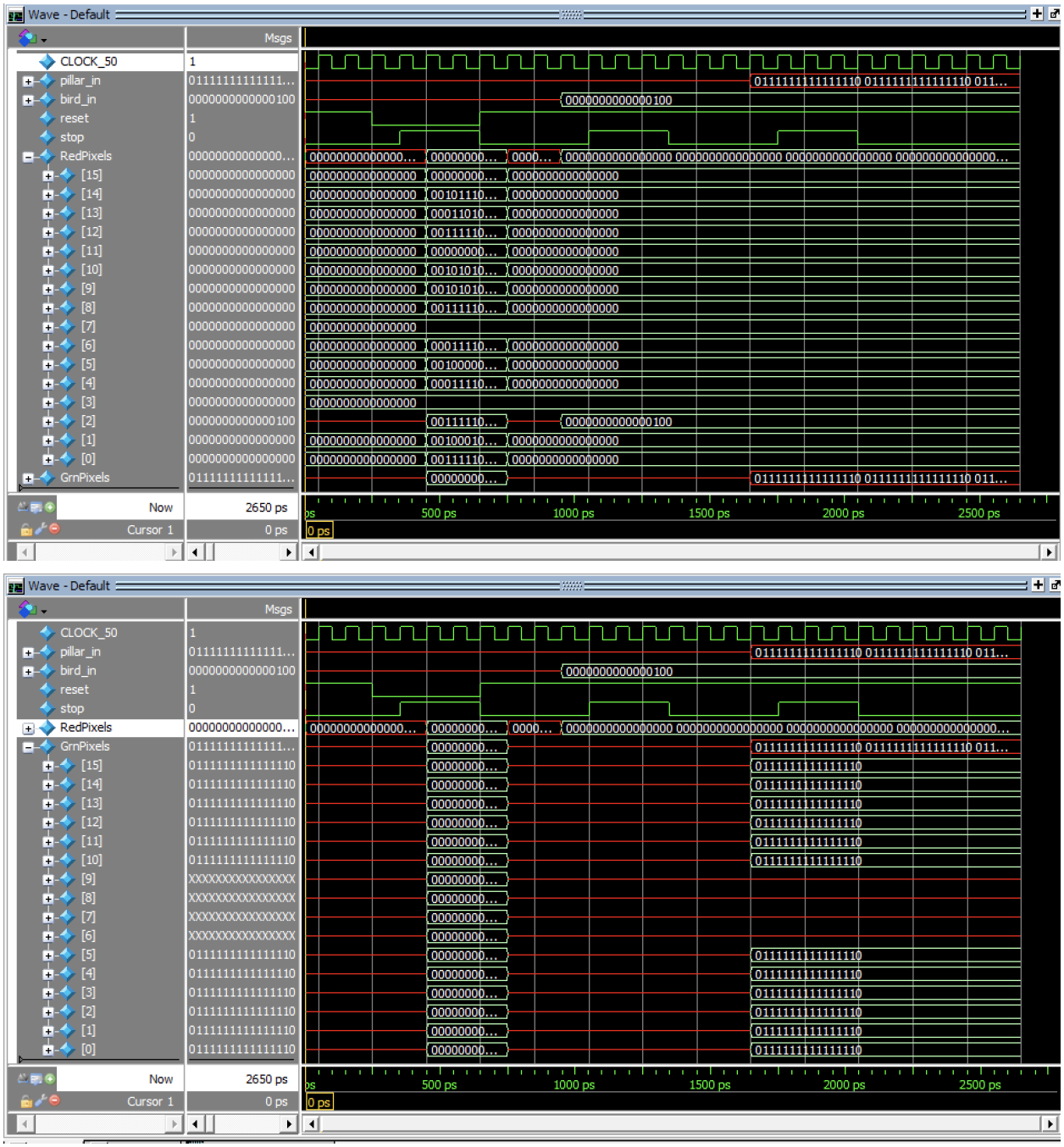
This module uses FSM to have the running screen for the bird, ending screen to display GAME OVER when the bird hits a pillar for the RedPixels, and input pillar_in from the column driver module that will output the GrnPixels.

FSM state diagram of module



Above are two FSMs, one for the RedPixels and one for the GrnPixels. The RedPixels at reset have their running_screen (to which in the module I had it where the bird was at its starting position). Otherwise, the running_screen will be bird_in inputs from the bird module. When stop is true, the next state is off which will produce the end_screen and stay there until the reset switch is true again. For the GrnPixels, its default is having all its display off (both at reset and when it is out of the on state). But in the on state, it will output what the pillar_in input is from the column driver module. When stop is true, the next state will be the default which is all GrnPixels off.

ModelSim



As shown above, when the reset switch is true, it has the GrnPixels all off but the RedPixels at its start state (so the bird is that the bird_light_start). Then, when stop is true, the GrnPixels are all off and the RedPixels has the output of GAME OVER. I then tested when there were pillar_in and bird_in inputs so that the RedPixels and GrnPixels would have those inputs (which would then transfer to the LEDDriver GPIO_1). When stop is true but the reset is also true, reset has priority so the RedPixels will just have the bird_light_start and the GrnPixels will be off.

Q4

I tested the system by first implementing every module with a test bench, which would test all states if there was a FSM or relevant inputs that would produce output that was expected behavior. I started creating test benches first for modules that were lone (did not rely on other modules). If there was a problem with a module that relied on other modules, I had the test benches to locate where the problem occurred (whether it was in the independent or dependent modules). I also tried to implement a test bench for the top level module, but that was difficult to do with the multiple clock counters and enables. So I decided to actually go to testing on hardware once all the modules from the top level were tested on the ModelSim with the correct behavior. I found that the hardware did work on the first try of testing, and tested with different levels of difficulty and trying to get a “high” score on the game (my highest was 162). I also tested if I left the bird to fall to the bottom of the board, or fly all the way up to the top of the board to ensure that it would be GAME OVER if the bird continued to fall or fly past the board.

Time Spent

This lab took me approximately 40 hours to complete.

Thank you for the great quarter! This was definitely my most favorite class.