Lab 8 Report

Hemkesh Bandi - 2322959

May 25th, 2024

1. Instruction Manual

The instructions are based on the fact that the code is already loaded correctly to the DE1-SoC with the LED-Expansion module.

This is a normal game of tic-tac-toe where a player tries to make a line of 3 of their own symbols before the other player. Player 1 is X's, and Player 2 is O's.

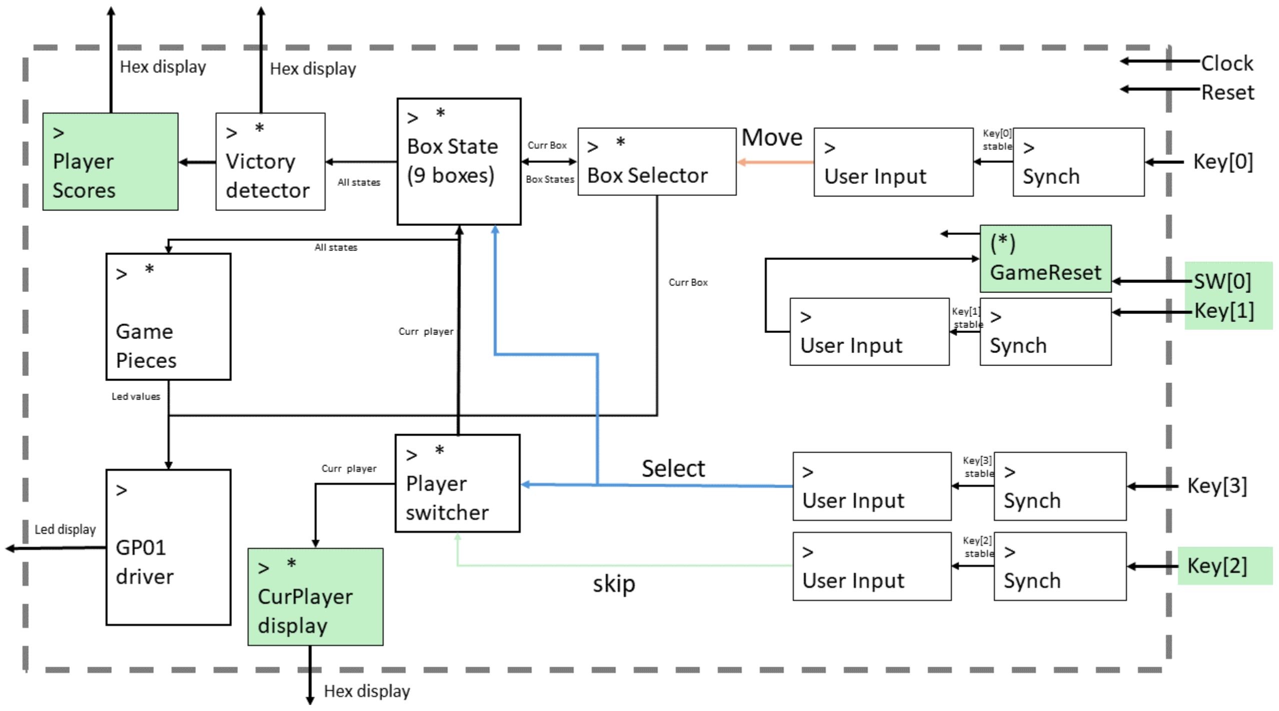
Inputs:
Key[0]: is used to move to where the current player wants to claim.
Key[1]: is used to restart the current game or go to the next game.
Key[2]: is used to switch which player it is.
Key[3]: is used to pick the current space as the current players.
SW[0]: is a toggle so a new game starts right after a game is finished.
SW[9]: is a reset switch, so the game is reset along with the scores.

Game Board:
The 16x16 LED board shows the current state of the game.

Hex Displays:
HEX0: the number of draws.
HEX1: the number of players 1 wins.
HEX2: the number of players 2 wins.
HEX4: who currently wins.
HEX5: the current player.

2. Top-Level block diagram

Top-Level block diagram:



This is the top-level block diagram for the tic-tac-toe system; that I decided to do for my final project, using a mix of old modules and new code.

Additionally, the 9 possible spots on a tic-tac-toe board are described as boxes for this diagram.
Modules:
The ”Synch” module alleviates the meta-stability issue for pressed buttons on clock edges. Taking in the button input and outputting a stabilized equivalent. Which introduces a cycle of delay. Already tested code from lab 6.

The ”User Input” modules take in the stabilized input, so a held-down button is only recognized at the rising edge. The button has to be pressed repeatedly instead of just held down. That ”pulse” is the output. Already tested code from lab 6.

The ”Player Scores” module takes in input if a win is counted and then uses a 3-bit counter to track scores from 0 to 7. This is then displayed via the 7-seg display module. This module/code was already tested in lab 7. A little bit of code was added from the ”User Input” module to detect a win only once, but that is also tested.

The ”Player switcher” module takes in both a select and skip input, which switches to the next player from the press of either one.

The ”Box State” module takes the cur box from the ”Box Selector,” the select signal, and the current player signal to determine the state of a box. Which that box’s state is outputted.

The ”Box Selector” Takes in both the current state of a box from the ”Box State” module, either already picked (X or O), and the move input to determine where to place the indicator for the current player. It then sends this position to the GP01 driver module to output to the LED board. The ”Box Selector” deals with display output and determining which box to go to (next open box). This module happened to serve the purpose of also a semi-display module where it deals with the indicator location in pixels along with initializing the game lines to make a 3x3.

The ”Victory Detector” module gets the current state of the current game where if a three in a row is found, then victory is assigned to the respective player and displayed on a 7-seg display.

The ”Game Pieces” module gets the state of each box and then assigns a part on the 16x16 display to the respective game piece. This is then sent to the GP01 driver module, which outputs the correct game piece on our 16x16 led board. This is also a quite trivial module as it is just is a modified version of the 7-seg display and LEDtest.sv that were given to us.

More detailed interconnects are explained in the module descriptions.

All modules with the > will have an input for the Clock and the Reset.

All modules with the * will have an input for the GameReset value, which essentially wipes the current game board. This is for going to the next game and keeping the scores.

The ”CurPlayer display” module gets input from the ”Player Switcher” module, which then outputs that number (the current player) to a 7-seg display. This is another trivial module that just changes the 0 or 1 input to 1 or 2 and inputs that value to a 7-seg module.

The ”GameReset” module is just a logical state meant for the SW[0], which acts as a toggle to restart the game instantly after the last game, and the Key[1] input makes it a button instead. This is a very basic and trivial module that isn’t even an actual module in the code.

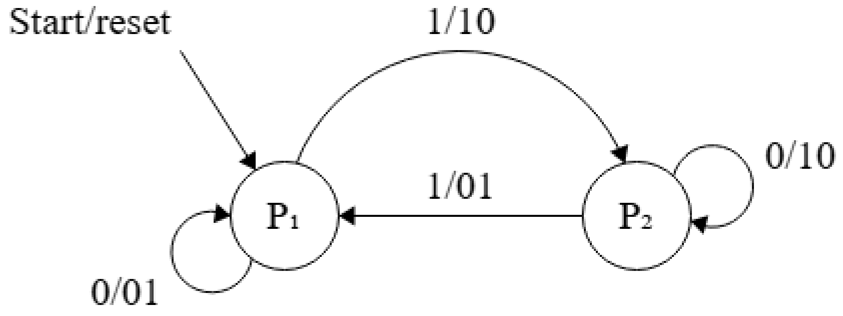
Additionally, everything in green is extra modules and inputs for ease of use/extra credit and not needed for core functionality/requirements.

2. Module Player Switcher

This module is made to determine which player's turn it currently is; if the current game has ended, the output of this module doesn't matter to us. This is done by an input called select which was initially just the button press, but to add a skip player function we the input is still select, but the interconnect is $.select(select||skip_s)$ at the top level. This was done as I needed to switch the current player without affecting the other modules that still use $select$. For the FSM we will call this interconnect $select$ for simplicity purposes.

2.a. FSM Player Switcher

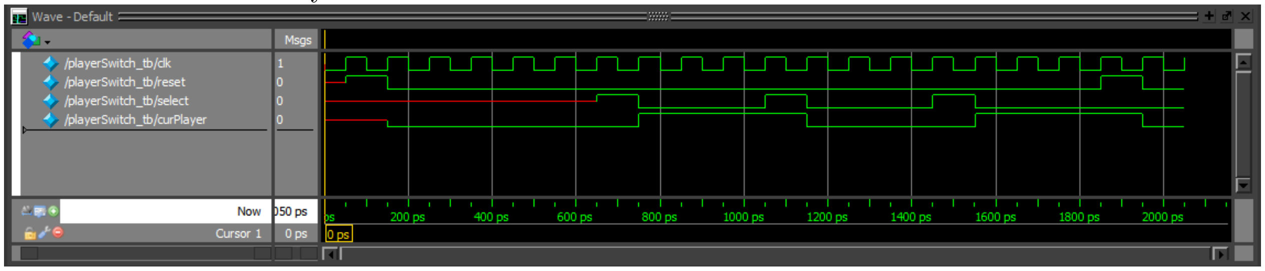
FSM of Player Switcher:



This FSM is pretty simple where we start at P_1 which is player 1 if we get an input of a 1 or *true* signal then we go to the other state of P_2 player 2 where when transitioning to a state, we output 01 for going to P_1 and 10 for going to P_2. We also do the same when there is no signal to switch (input), where we will output the value of the state we are in 01 for P_1 and 10 for P_2. If we get a reset signal, we again reset to P_1.

2.b. ModelSim of Player Switcher

This ModelSim is of Player Switcher:



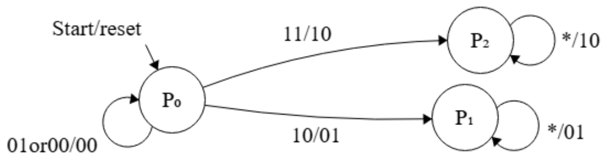
The topmost signal is the clock, followed by the reset signal, then by the input of select, and lastly by the output signal of the curPlayer. This ModelSim picture shows that when select is 1, we transition to the next player and output that where 0 is player 1 and 1 is player 2. This was done to reduce memory usage as it will always be player 1 or 2 (2 states). We can also say that when the reset signal is true, the counter resets to 0.

3. Module Box State

This module is made to determine the current state of each box, where there are three possible states such as empty, player 1 ”owned” (X), and player 2 ”owned”(O). Since there are 9 spots for any player to select, equal 9 box modules to make up the high-level module of ”Box State”, in the top-level diagram. So, a box module works with a select and curPlayer input and an 2-bit output value to represent the three previously mentioned states. The interconnect that is a little different is that $.select(select$ && $curBox == 0)$ curBox equals whatever box this module is $0 - 8$. This reduces the complexity of the module itself. For the FSM, we call this interconnect just *select* as this input will be true if the box is genuinely ”selected.”

3.a. FSM of each box

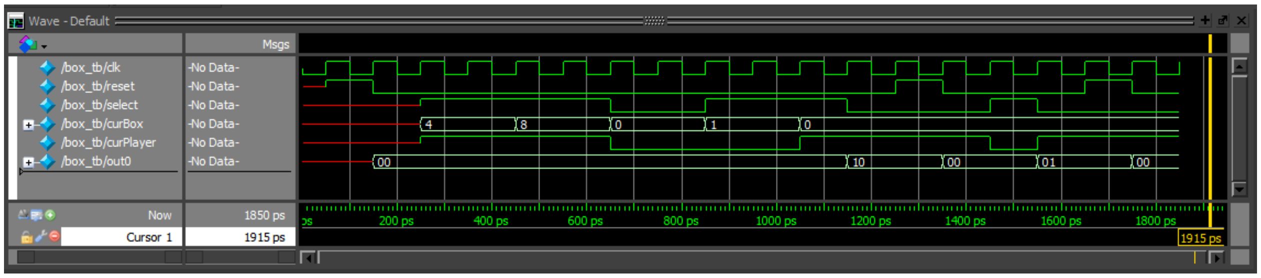
This is the state diagram of box:



The first bit of the input is the *select* value, and the second bit is the curPlayer (0 or 1 for 1 or 2). Then the output who is now "owning" that box, which is just if there should be an X or O at that box, in a simple format. then we also know that unless the box is reset the box will be in a sink state once picked by either player; characterized by the * input which is all inputs. Other than that, if the box is not selected (the first bit is 0), we stay in the starting state.

3.b. ModelSim of each box

This ModelSim is of box module:



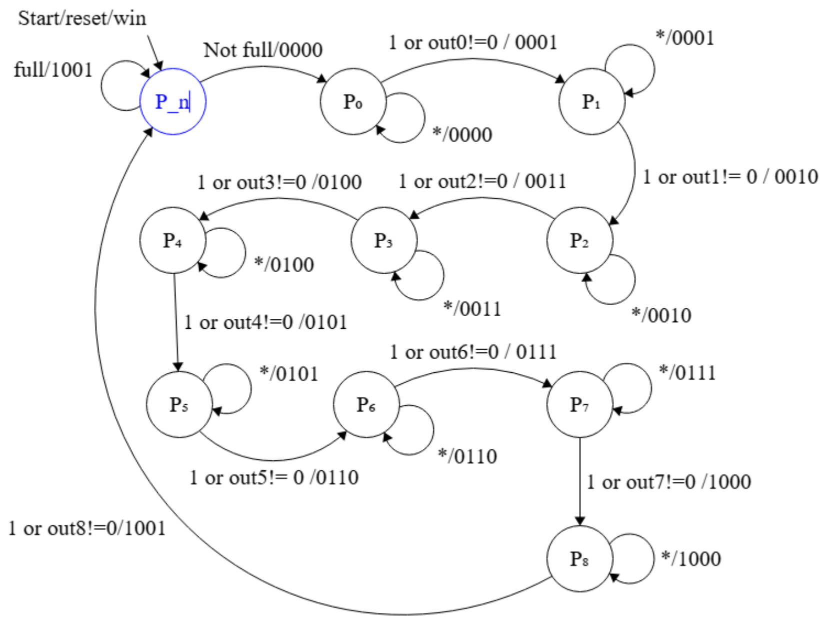
The topmost signal is the clock, followed by the reset signal, then by the input of select and curBox, then the input signal of the curPlayer, and lastly by the output (this box is initialized to be box0 so curBox has to be 0 for the output to change in theory). We can see that we only switch the state of the current box when both select and curBox are 1, which is in line with the FSM. Then based on what the curPlayer signal is (0 or 1 for p1 or p2), the output matches that or stays 0, again in line with the FSM.

4. Module Box Selector

This module is essentially a combination of two modules. One determines what box the current player is on, and the other displays that along with the lines that define the 3x3 tic-tac-toe board. We will focus most of my explanation on the first half of this module, as the other is just the ”LEDtest.sv” with if statements to fit our use case. The interconnects for this module are the next, along with the current state of each box, then the outputs are the current box the player is on and the GrnPixels determined by the trivial part of our module. Then, for reset, we also include the select signal and the win signals, as those require the same behavior as the reset signal.

4.a. FSMs of Box Selector

This is the state diagram of the Box Selector, without the pixel output:



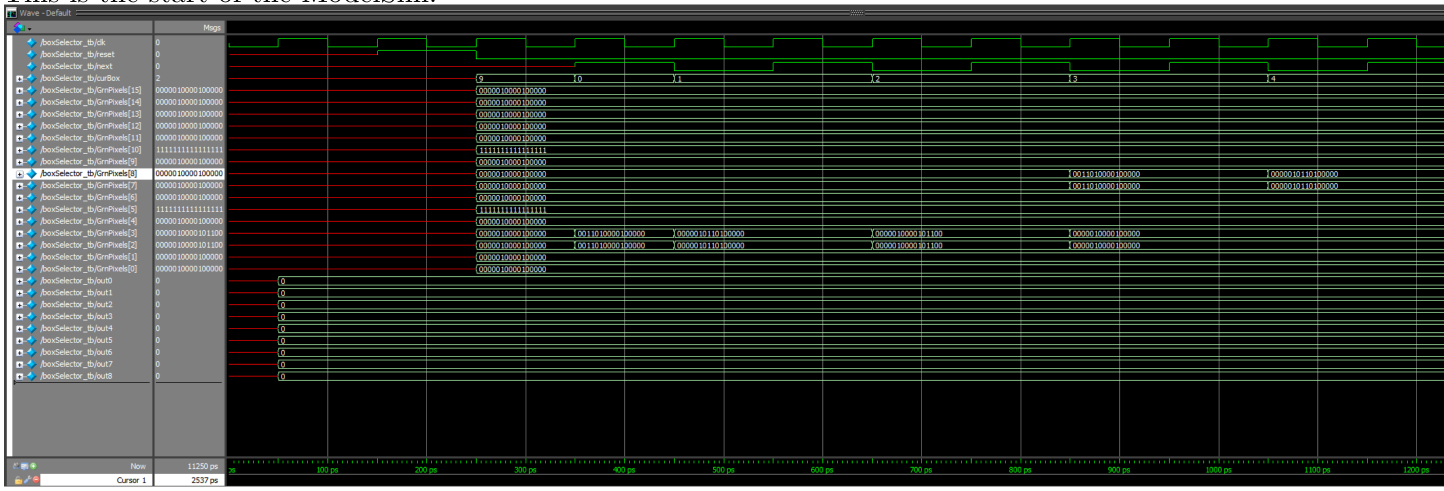
We start and loop back to the P_n state as that checks if all boxes are full. If not, we go to the next state, and then for all the other boxes, if the next button or the current box is filled, we again transition to the next state. Whatever state we are in/going to be in will be outputted through the current box signal. In this case we will say that a * is all other inputs other than the one defined.

The trivial part of this state diagram is that when in a state, we make a green 2x2 set of LEDs on the 16x16 LED board to show the user. We can say that this is determined by the output signal of this FSM at each state as that is what it essentially boils down to. Also we can say that it also sets the borderlines for the game in each state. Then, for P_n, it only has border lines.

4.b. ModelSim of Box Selector

This ModelSim is of Box Selector module:

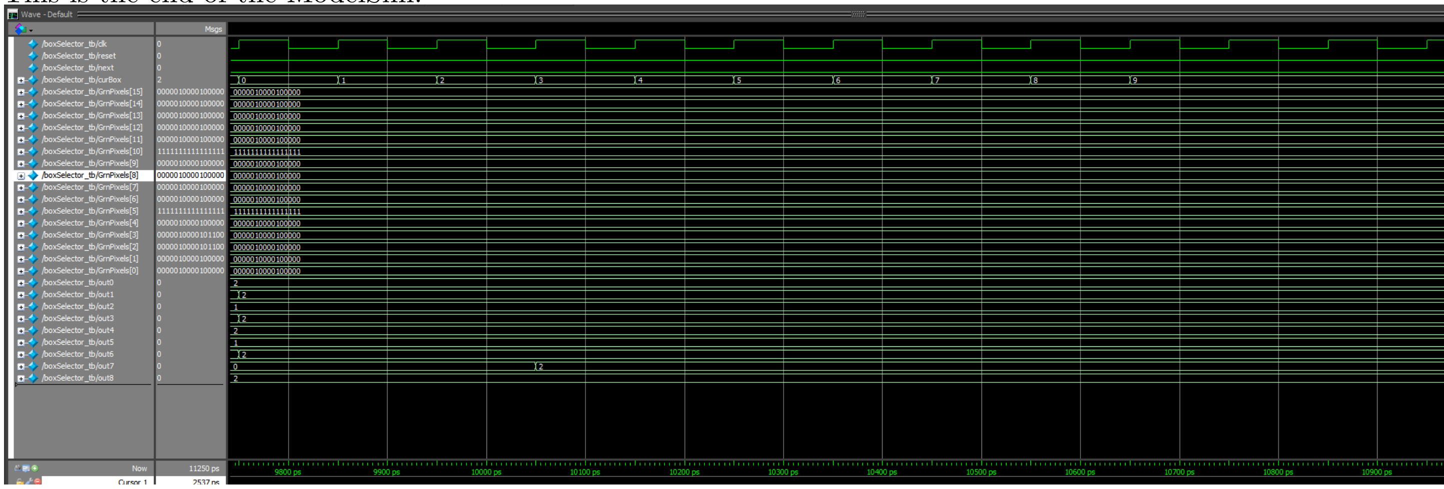
This is the start of the ModelSim:



```
Wave - Default
                                      Msgs
/boxSelector_tb/clk                   0
/boxSelector_tb/reset                 0
/boxSelector_tb/next                  0
/boxSelector_tb/curBox                2
/boxSelector_tb/GrnPixels[15]         0000010000100000
/boxSelector_tb/GrnPixels[14]         0000010000100000
/boxSelector_tb/GrnPixels[13]         0000010000100000
/boxSelector_tb/GrnPixels[12]         0000010000100000
/boxSelector_tb/GrnPixels[11]         0000010000100000
/boxSelector_tb/GrnPixels[10]         1111111111111111
/boxSelector_tb/GrnPixels[9]          0000010000100000
/boxSelector_tb/GrnPixels[8]          0000010000100000
/boxSelector_tb/GrnPixels[7]          0000010000100000
/boxSelector_tb/GrnPixels[6]          0000010000100000
/boxSelector_tb/GrnPixels[5]          1111111111111111
/boxSelector_tb/GrnPixels[4]          0000010000100000
/boxSelector_tb/GrnPixels[3]          0000010000101100
/boxSelector_tb/GrnPixels[2]          0000010000101100
/boxSelector_tb/GrnPixels[1]          0000010000100000
/boxSelector_tb/GrnPixels[0]          0000010000100000
/boxSelector_tb/out0                  0
/boxSelector_tb/out1                  0
/boxSelector_tb/out2                  0
/boxSelector_tb/out3                  0
/boxSelector_tb/out4                  0
/boxSelector_tb/out5                  0
/boxSelector_tb/out6                  0
/boxSelector_tb/out7                  0
/boxSelector_tb/out8                  0
                              Now     11250 ps
                         Cursor 1     2537 ps
```

The topmost signal is the clock, followed by the reset signal, then by the input of next, the curBox output, and the actual GrnLED output (keep in mind that it goes from index 15 to 0, so the bottom should be the top, not too relevant for testing purposes) which is the job of our trivial module at the very bottom we have the state of each box (the outs). We can see here that in the initial testing we do go from the P_n (curBox = 9) state to P_0 without needing next to be 1. Then we can see that each time the next goes to 1, we go to the next box. Then we also see that the 2x2 green LED box also moves relating to the value of curBox.

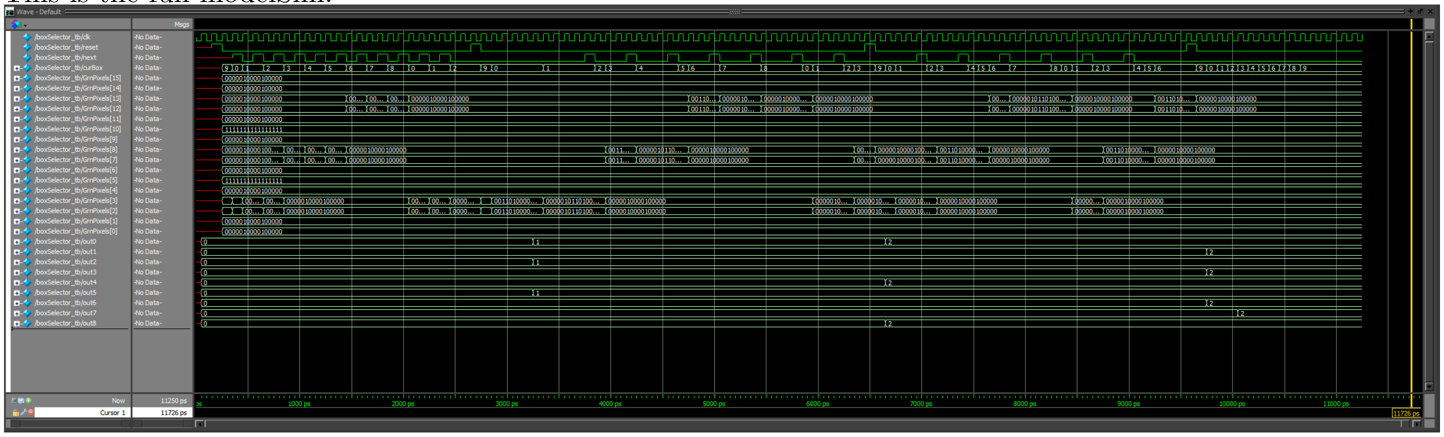
This is the end of the ModelSim:



```
Wave - Default
                                      Msgs
/boxSelector_tb/clk                   0
/boxSelector_tb/reset                 0
/boxSelector_tb/next                  0
/boxSelector_tb/curBox                2
/boxSelector_tb/GrnPixels[15]         0000010000100000
/boxSelector_tb/GrnPixels[14]         0000010000100000
/boxSelector_tb/GrnPixels[13]         0000010000100000
/boxSelector_tb/GrnPixels[12]         0000010000100000
/boxSelector_tb/GrnPixels[11]         0000010000100000
/boxSelector_tb/GrnPixels[10]         1111111111111111
/boxSelector_tb/GrnPixels[9]          0000010000100000
/boxSelector_tb/GrnPixels[8]          0000010000100000
/boxSelector_tb/GrnPixels[7]          0000010000100000
/boxSelector_tb/GrnPixels[6]          0000010000100000
/boxSelector_tb/GrnPixels[5]          1111111111111111
/boxSelector_tb/GrnPixels[4]          0000010000100000
/boxSelector_tb/GrnPixels[3]          0000010000101100
/boxSelector_tb/GrnPixels[2]          0000010000101100
/boxSelector_tb/GrnPixels[1]          0000010000100000
/boxSelector_tb/GrnPixels[0]          0000010000100000
/boxSelector_tb/out0                  0
/boxSelector_tb/out1                  0
/boxSelector_tb/out2                  0
/boxSelector_tb/out3                  0
/boxSelector_tb/out4                  0
/boxSelector_tb/out5                  0
/boxSelector_tb/out6                  0
/boxSelector_tb/out7                  0
/boxSelector_tb/out8                  0
                              Now     11250 ps
                         Cursor 1     2537 ps
```

The topmost signal is the clock, followed by the reset signal, then by the input of next, the curBox output, and the actual GrnLED output (keep in mind that it goes from index 15 to 0, so the bottom should be the top, not too relevant for testing purposes) which is the job of our trivial module at the very bottom we have the state of each box (the outs). Here we are showing that without the select signal, since a box is occupied, we move to the next box on each clock edge, then the last box also gets filled (not realistic but irrelevant for testing purposes). At that point, we reach the null state where curBox is equal to 9, which is not a value of importance as we only have boxes from 0 to 8 and stay there since all boxes are filled. Here we can see again the trivial part of our module the output of GrnLEDs is working properly.

This is the full modelSim:


Wave - Default
Msgs
/boxSelector_tb/clk
/boxSelector_tb/reset
/boxSelector_tb/next
/boxSelector_tb/curBox
/boxSelector_tb/GrnPixels[15]
/boxSelector_tb/GrnPixels[14]
/boxSelector_tb/GrnPixels[13]
/boxSelector_tb/GrnPixels[12]
/boxSelector_tb/GrnPixels[11]
/boxSelector_tb/GrnPixels[10]
/boxSelector_tb/GrnPixels[9]
/boxSelector_tb/GrnPixels[8]
/boxSelector_tb/GrnPixels[7]
/boxSelector_tb/GrnPixels[6]
/boxSelector_tb/GrnPixels[5]
/boxSelector_tb/GrnPixels[4]
/boxSelector_tb/GrnPixels[3]
/boxSelector_tb/GrnPixels[2]
/boxSelector_tb/GrnPixels[1]
/boxSelector_tb/GrnPixels[0]
/boxSelector_tb/out0
/boxSelector_tb/out1
/boxSelector_tb/out2
/boxSelector_tb/out3
/boxSelector_tb/out4
/boxSelector_tb/out5
/boxSelector_tb/out6
/boxSelector_tb/out7
/boxSelector_tb/out8
Now 11250 ps
Cursor 1 11726 ps
1000 ps
2000 ps
3000 ps
4000 ps
5000 ps
6000 ps
7000 ps
8000 ps
9000 ps
10000 ps
11000 ps

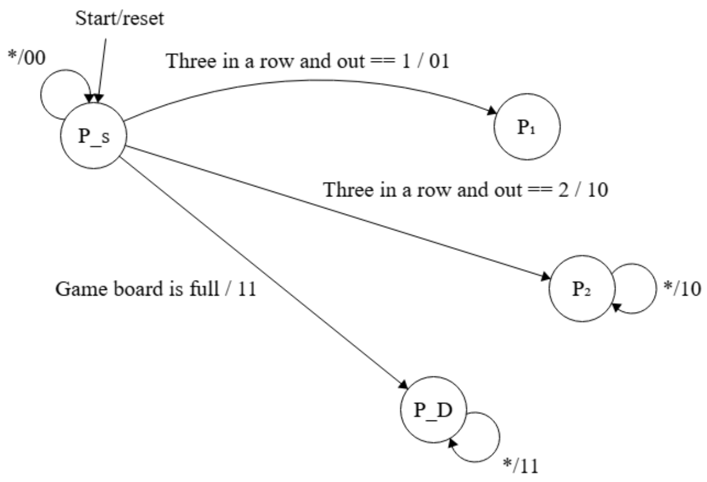
The topmost signal is the clock, followed by the reset signal, then by the input of next, the curBox output, and the actual GrnLED output (keep in mind that it goes from index 15 to 0, so the bottom should be the top, not too relevant for testing purposes) which is the job of our trivial module at the very bottom we have the state of each box (the outs). This is just the complete simulation as we have verified the trivial part, and now we can see that again, the module is working in line with the FSM.

5. Module Victory Detector

This module checks the state of all the boxes. If a three-in-a-row (tic-tac-toe kind) of the same value is found, then it sets the winning output to the respective values $01, 10, 11$ for players 1, 2, and a draw, respectively. This was simple and was done before in lab 7, so little testing was needed. We also take part in lab 6's victory module by displaying the winner on a HEX output. The interconnects are the inputs of all box states, an HEX display output, and the win state output. The reason for testing was the introduction of a draw state.

5.a. FSM of Victory Detector

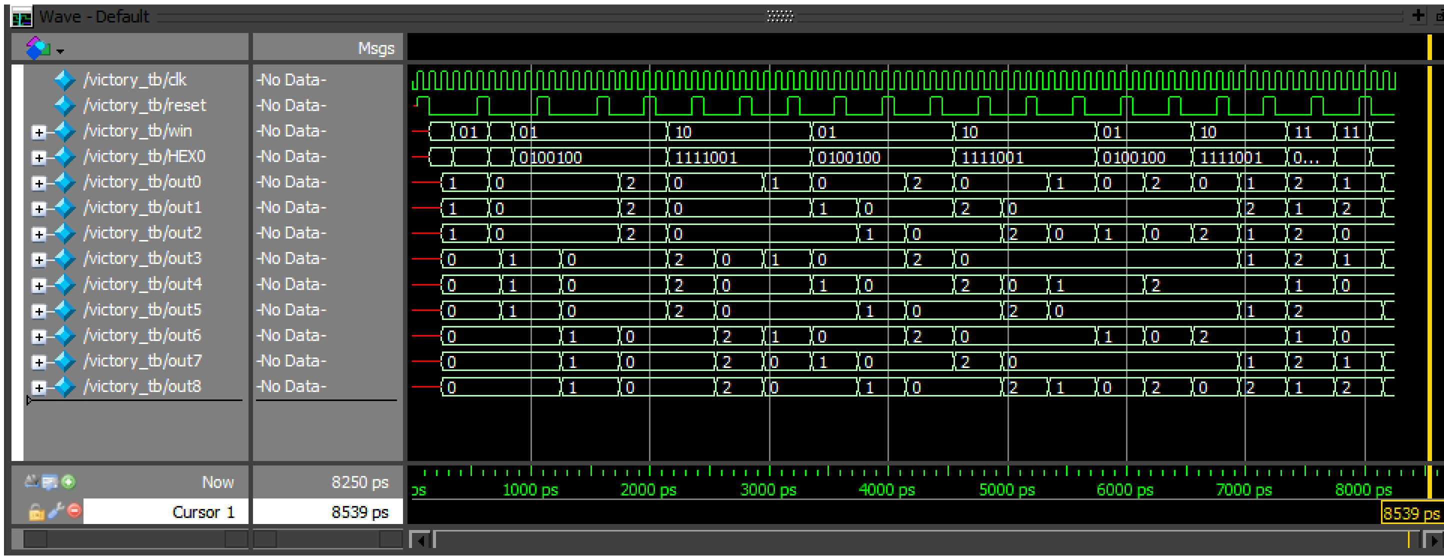
This is the state diagram of the Victory Detector:


Start/reset
*/00
Three in a row and out == 1 / 01
P_s
P₁
Three in a row and out == 2 / 10
Game board is full / 11
P₂
*/10
P_D
*/11

We start in the P_s state, and then we see that if a three in a row is detected, we go to the state representing the value of that three in a row. We then output the value representing that player. If all outputs (boxes) are filled and no three in a row, then we go to P_D, which is a draw. These are all sink states as well, so the only way to clear the win, in a sense, is to restart the game (reset).

5.b. ModelSim of Victory Detecto

This ModelSim is of the Victory Module:


Wave - Default
Msgs
/victory_tb/clk -No Data-
/victory_tb/reset -No Data-
/victory_tb/win -No Data-
/victory_tb/HEX0 -No Data-
/victory_tb/out0 -No Data-
/victory_tb/out1 -No Data-
/victory_tb/out2 -No Data-
/victory_tb/out3 -No Data-
/victory_tb/out4 -No Data-
/victory_tb/out5 -No Data-
/victory_tb/out6 -No Data-
/victory_tb/out7 -No Data-
/victory_tb/out8 -No Data-
Now 8250 ps
Cursor 1 8539 ps
1000 ps
2000 ps
3000 ps
4000 ps
5000 ps
6000 ps
7000 ps
8000 ps
8539 ps

The topmost signal is the clock, followed by the reset signal, then by the values of the win signal and HEX0 display, and then the input signal of all the boxes (out0-8). We can see that for all tic-tac-toe winning states (non-realistic), we do output a win, which is the same for both players; we also check that we stay starting if the game hasn't reached an end state and check a draw state. We also output a $1, 2, -$ for player 1 or 2 win and a draw on the HEX0 output.

6. Trivial Modules / Other Modules

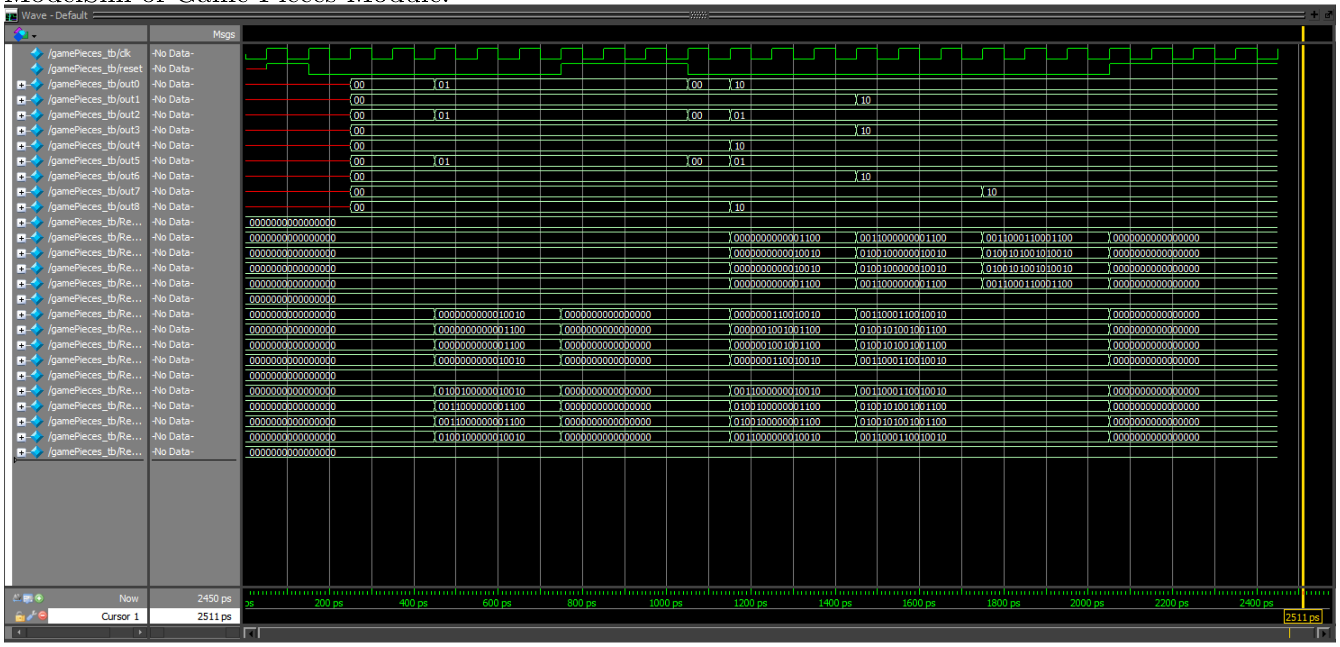
Multiple modules and code had already been tested and were labeled as trivial; this was listed in the top-level diagram description. Additionally, ”GameReset” is just a module in name and is really just a signal defined by ”assign $game_{reset}$ = (SW[0] && win != 2’b0) || $clear_s$;” This is then or-ed with the RST signal for all modules that aren’t the player scores. So that a new ”game” can start but the whole system isn’t reset. Furthermore, the current player was labeled as trivial as it did very little, and all it did was assign an output value to 1 or 2 respective to an input of 0 or 1 and then sent the output to a seg7 display module.

7. ModelSim (FSM) of Game Pieces module

Description:
This module is quite basic, where all it does is output the correct symbol for the state of a box (out0-8). This is done through the RedPixel output, which then goes to the GP01 driver. This is done by defining X, O to be players 1 and 2, respectively.

FSM explanation:
The FSM of this module would start at P_s be an empty output of $0's$ in the respective out's box if the value is 0, but if not 0, then output the respective symbol of the value of out P_1 or P_2 which are each sink states. Which is just a bunch of if statements.

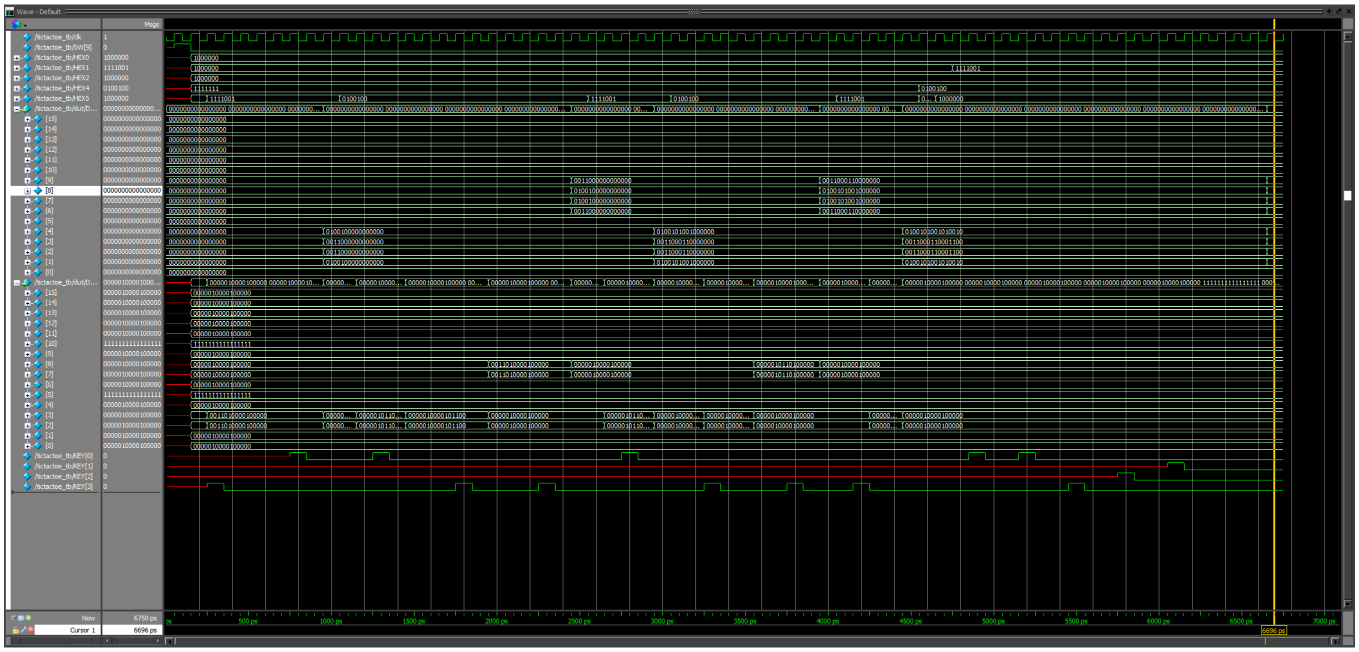
ModelSim of Game Pieces Module:



The topmost signal is the clock, followed by the reset signal; then we have the state of each box (the outs), then we have the RedLED output, which, while hard to see, displays O or X based on a box's value. Reset being on clears the RedPixels.

7 . ModelSim of Top-Level module (Tic-Tac-Toe), Testing overview

ModelSim of my Tic-Tac-Toe project:


Wave - Default
Msgs
/tictactoe_tb/clk 1
/tictactoe_tb/SW[9] 0
/tictactoe_tb/HEX0 1000000
/tictactoe_tb/HEX1 1111001
/tictactoe_tb/HEX2 1000000
/tictactoe_tb/HEX4 0100100
/tictactoe_tb/HEX5 1000000
/tictactoe_tb/dut/D... 00000000000000...
/tictactoe_tb/dut/D... 0000010000 1000...
/tictactoe_tb/KEY[0] 0
/tictactoe_tb/KEY[1] 0
/tictactoe_tb/KEY[2] 0
/tictactoe_tb/KEY[3] 0
Now 6750 ps
Cursor 1 6696 ps

The topmost signal is the clock, followed by the reset signal, we have our HEX outputs, then our GrnLED and RedLED outputs, followed by our 4 key inputs. The HEX outputs are HEX0, the number of draws; HEX1, the number of players 1 wins; HEX2, the number of players 2 wins; HEX4, who currently wins; and HEX5, the current player. This simulation mainly shows the interconnect between the modules along with checking the game reset functionality of KEY[1] (Which is also able to be a toggle with SW[0]). Other than that, we can see that the player who won makes a row of X's at the top (the led rows are flipped) and wins, changing the related HEX displays as well. We also check for inputs doing anything after a win and they do do nothing like expected/wanted.

This is my virtual testing to ensure basic interconnects are working. But I have also tested the code on the DE1-SoC myself for various other situations and random inputs someone could provide. I have also given the board to a friend who is not in the class and let them play the game with their girlfriend. Nothing went wrong, and things reacted in line with expectations.

How many hours (estimated) did it take to complete this lab in total, including reading, planning, designing, coding, debugging, and testing?

(a) This lab took me **30** hours to complete.