Anthony Humay, Andrew Musk, Clayton Green

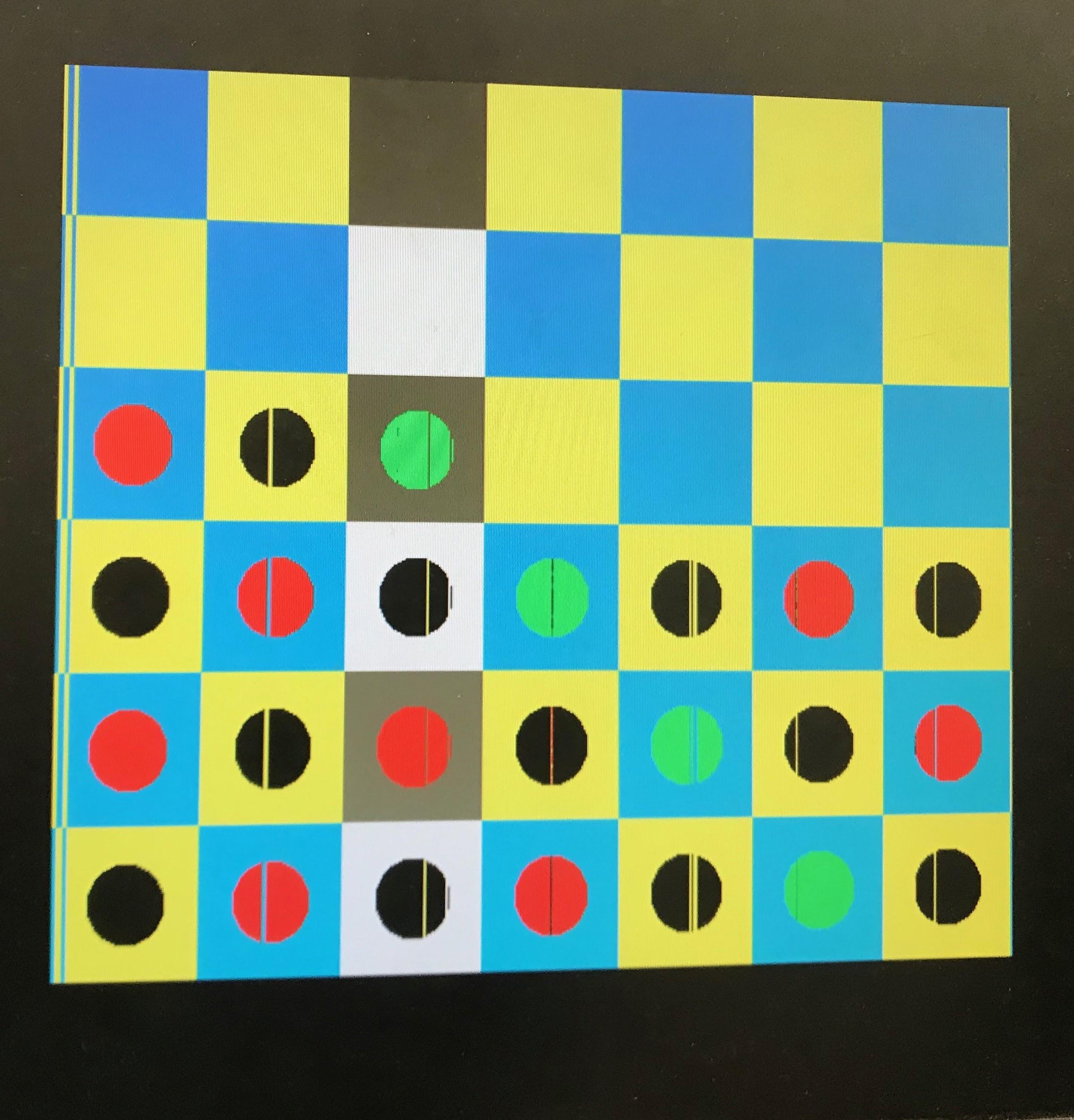
M152A Lab 6

Mastorakis

Lab 4 Report

**Introduction**

In this lab, we implemented the game Connect 4. Connect 4 is a two player game. Each player gets a set of colored circular coin-like pieces, with each set being a different color. The maximum size of the set of pieces is 21. There is a 7 column by 6 row board. Players take turns dropping pieces into one of the 7 columns, one piece at a time. Each of these turns are a maximum of 30 seconds long. The time remaining is shown on the seven segment display, with the first two digit slots showing either “P1” or “P2” depending on who’s turn it is, and the time remaining showing in the latter two digits. The background of the board also changes based on whose player turn it is; it will always be the same color as the player’s pieces. The object of the game is to get 4 of the same pieces horizontally, vertically, or diagonally in a row on the board. The game ends once this happens, and the player who dropped the 4 pieces in a row wins, causing the winning pieces to turn green (see figure 0.1). The game can also be restarted using a reset button.



*Figure 0.1* The game in it’s final state.

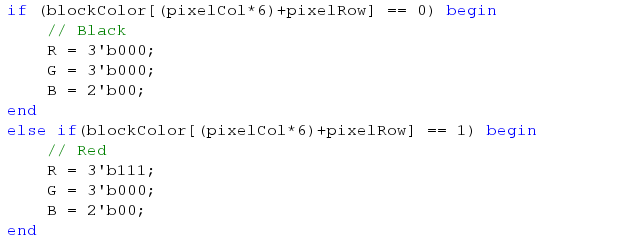
In terms of the game implementation, there is a one hot encoding of the first seven slider inputs for selecting the column to drop a piece into. For example, when slider 1 is flipped up, and the select (center) button is pressed, a coin will be dropped into the first column. The game board is drawn using the VGA connector. When a player has selected a column using the slider, we indicate this on the grid by highlighting the corresponding column in black and white (as opposed to the normal blue and yellow), as seen in figure 0.1. After every move, we check to see if the player has won the game. Once the game has been won, no more pieces can be dropped until reset is pressed.

**Design Description**

connect4.v

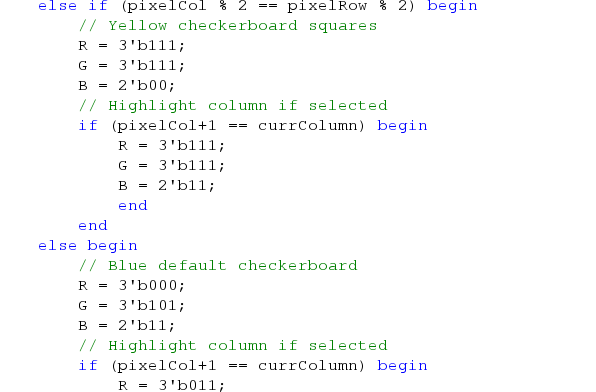
The board was represented in a one dimensional array named blockstate. It holds binary values; it is high only when occupied by a piece. There is an accompanying one dimensional array named blockcolor that designates which player the piece belongs to; it is low when it represents a player 1 piece (black), and high for player 2 (red). Both of these arrays hold all 42 of the cells that represent the 7x6 board.

Therefore, the location on the board will only be colored in if that position in blockstate is 1. Later in the module, we go through both arrays and, based on which player it is, color it accordingly. These RGB signals are sent to the VGA module.



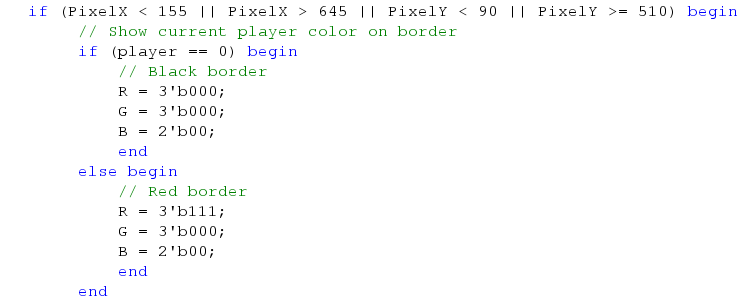
*Figure 1.1* When displaying a pixel that is at a location where has been placed on the board, check which player the piece belongs to and display accordingly.

The rest of the board is colored in a checkerboard style along the diagonals, done using modulus to only color every other cell in alternating rows. These RGB signals are likewise sent to the VGA.



*Figure 1.2* Creating the checkerboard effect.

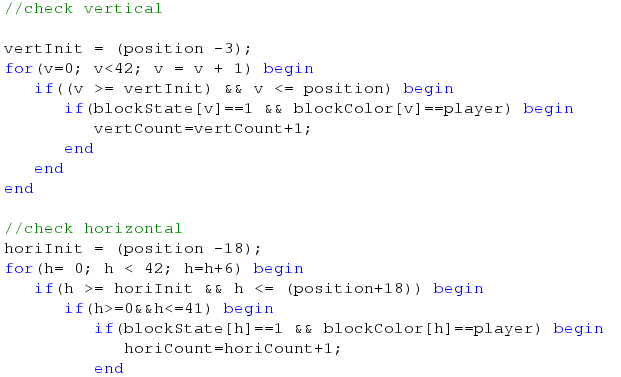
We decided to color the border based on the current player. Therefore, when displaying a pixel outside of the board, the code must check for the current player and display either red or black accordingly.



*Figure 1.3* The 7x6 board does not take up all of the screen on the monitor, leaving a border around it. The border color is based on the current player: black during Player 1, and red for Player 2.

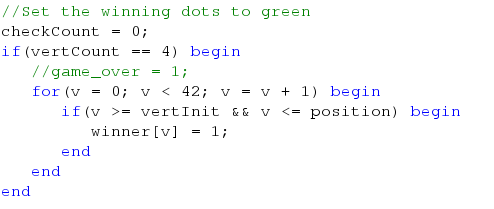
Since there are 7 columns and only 8 slider switches, one of the sliders must be ineffective. We chose the 8th slider, and had it be represented by a ‘nonexistent’ zeroth column in the code (we also use this zeroth column if there are multiple sliders, or none, set to high when select is hit).

connect4 also contains the logic for checking if there is a winner. This means checking if there are four of the same pieces in a row, in any orientation: horizontal, vertical, or diagonal. This is done separately in 3 for loops that cycle through blockstate in various ways depending on the orientation being checked.



*Figure 1.3* Checking if the last piece placed results in 4 pieces in a row vertically or horizontally for the same player.

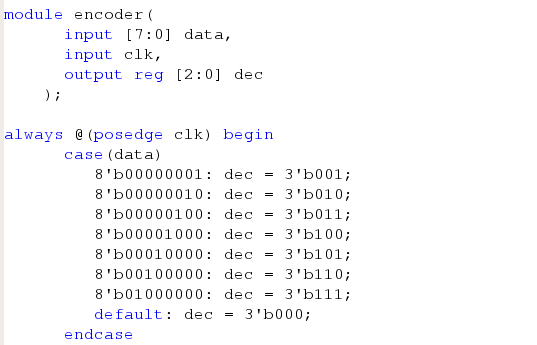
If we find that a player has won the game, we then color in the winning 4 pieces green.



*Figure 1.4* Setting the winning dots to green.

encoder.v

This module deciphers the one hot input into the binary representation of what column has been selected by the user. This is done through a straightforward case statement (figure 1.5). If multiple sliders are selected, or none, the encoder will return 0, which we have implemented to represent an invalid move, and nothing will happen.



*Figure 1.5* Checking which column is selected using slider switches.

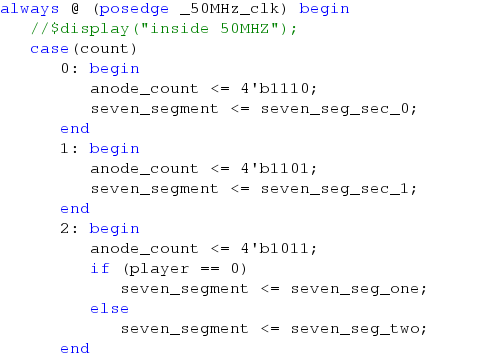
nexys3.v

This is the top-level module where we instantiate the btnS\_debounced, btnR\_debounce, \_clock, \_vga and \_connect4 modules. In other words, we create the VGA controller and initialize game logic here.

timer.v

In the end, most of the work that timer does was moved into connect4. We originally had this as a separate module that would deal with the counter for each player’s turn and deal with the logic of the seven segment display. However, the clock is reset to 30 and the player changes in two situations: a player makes a move, or the counter runs down to 0 and the player misses their turn. Because the logic for a player making a move and the player switching is done in connect4,we had to have these two module communicate with each other. We attempted to have connect4 tell timer when a player makes a move, and have timer tell connect4 when the clock runs out and the player needs to switch. However, we could not get them to work together properly and ultimately decided to insert the logic in timer directly into connect 4 so that they both had access to each other's variables. This worked much better.

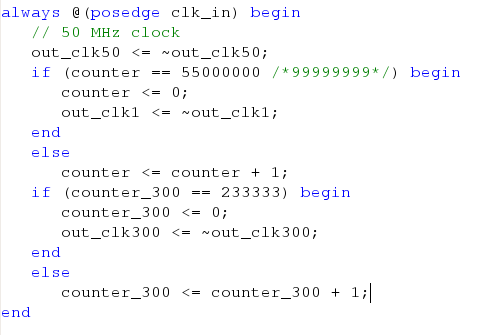
This was after we had trouble getting the two modules to communicate. To avoid these connection issues, we just put all the signals and computation of signals in connect4.



*Figure 1.6* A segment of code originally in timer that was moved into connect4. This is the logic for displaying to the seven segment display. The first digit is always a ‘P’ for player, the second was a ‘1’ or ‘2’ depending on whose turn it was, and the last two were used for the counter to represent the time left to make a move.

clock.v

In clock.v, we receive a 100 MHz signal. We divide this signal into 3 clocks that we need in our project: 300 Hz, 50 MHz, 1 Hz. The 300 Hz was for the seven segment display, the 50 MHz was for displaying using the vga, and the 1 Hz clocking was for decrementing the timer during a players turn.



*Figure 1.7* A simple clock divider that creates 50 MHz, 300 Hz, and 1 Hz clocks from the inputted 100 Mhz clock

vga.v

As mentioned in the Lab 4 document, we were allowed to use an outside repository to program VGA. In our case, this came from: <https://github.com/ThePedestrian/FPGA-Simple-Maze-Game-Using-VGA-Output>

This was connected to the rest of our implementation in nexys3.v, with the R, G, and B variables that are being manipulated throughout.

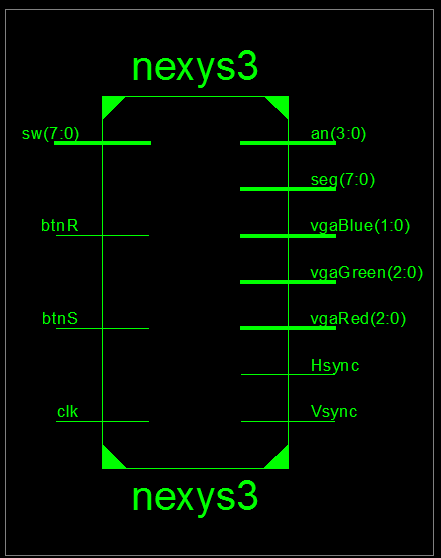
nexys3.ucf

We used the slider switches for selecting a column, the central button for dropping a piece, and the right button for resetting the board.

debouncer.v

This debouncer was largely based on Lab 3. It was used for the two input buttons.

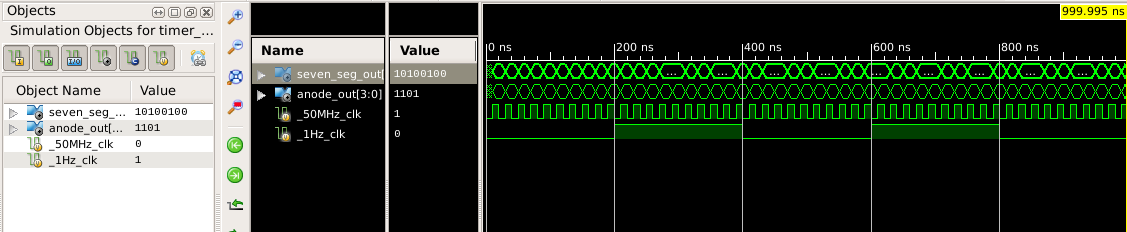
The design schematic is shown below:



*Figure 1.8* The schematic for our project.

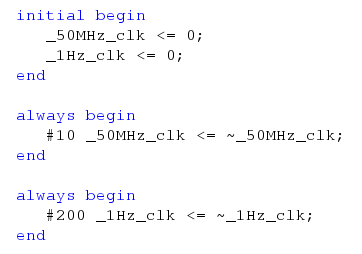
**Simulation Documentation**

One of the first things we tested was to make sure that our counter worked. We did this by creating a test bench called timer\_test(). The iSim waveform can be found below.



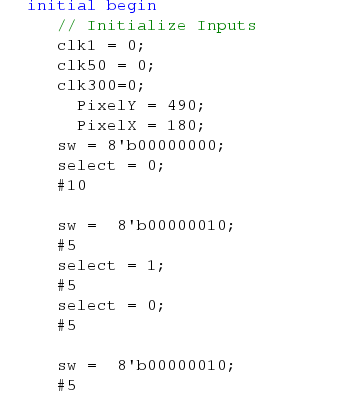
*Figure 2.1* timer.v testing iSim.

As shown by the code snippet below, timer\_test(), the counter was just initialized and the clocks were started. Surprisingly, this gave us a lot of trouble in Lab 3, so we wanted to ensure that we tested it thoroughly. During the simulation, the seven segment display initially shows “P1 30” to represent that it is player 1’s turn and they have 30 seconds to make their move. The number of seconds decrements at the positive edge of the 1Hz clock. Once the clock reaches zero, it then displays “P2 30” to show that it is now player 2’s turn and that they have 30 seconds to play. It worked as expected.



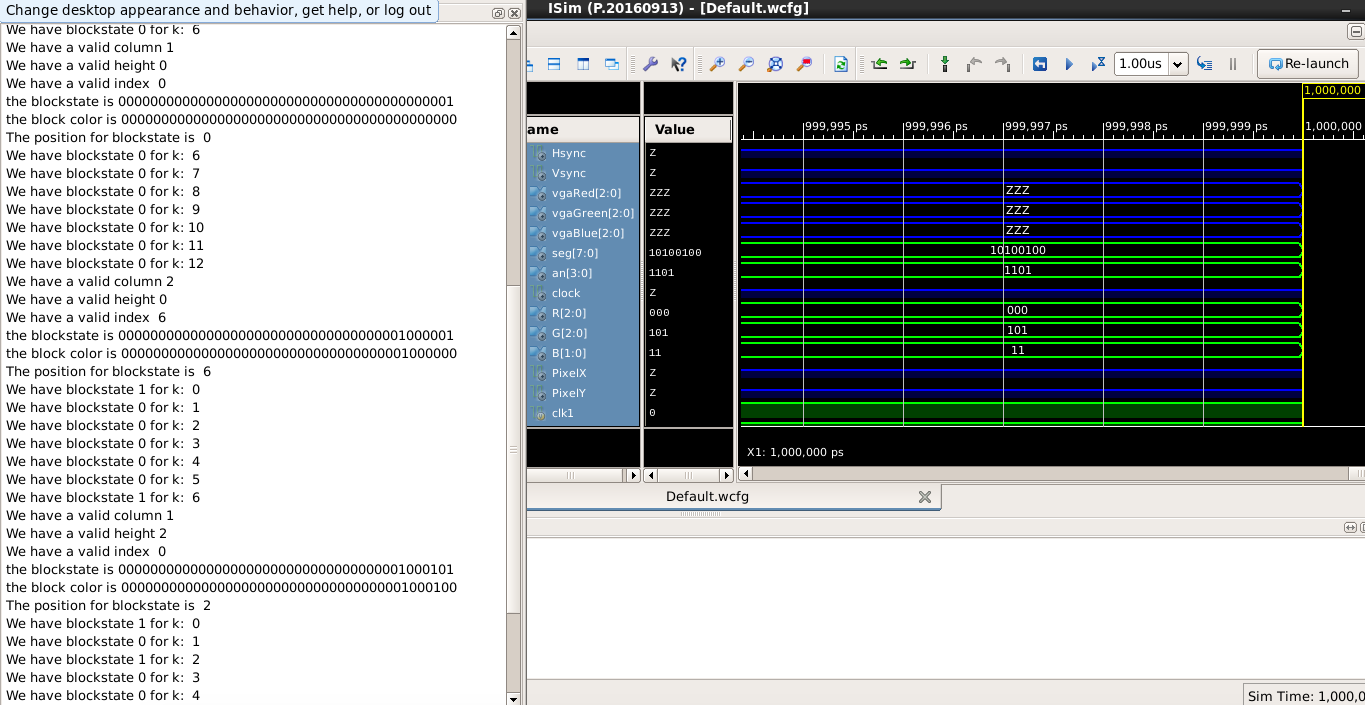
*Figure 2.2* Test case code for the timer that counts down the time left in a player’s turn.

We also created another testbench specifically for connect4. We primarily tested that blockstate and colorstate were correctly populating, often by outputting information (this is elaborated later). Since this was our primary testing module, we set the clocks to zero to begin, and then started sending in inputs from the slider switches. We waited to see if those were correctly received, and then sent in a selection button press. This was done in various styles of how it is shown in Figure 2.3.



*Figure 2.3* Test bench code for connect4.

The iSim waveform below shows our debugging process for our states (filled by a piece or empty) of the cells in the 7 by 6 structure, which was represented as a one dimensional array. We used the $display() to output after each move, as we were getting pieces populating from the top down initially, and then only populating the 3rd row. We needed to get a better understanding of how the array was actually getting filled, and we found the best way to do that was by outputting blockstate and colorstate everytime a move was made.

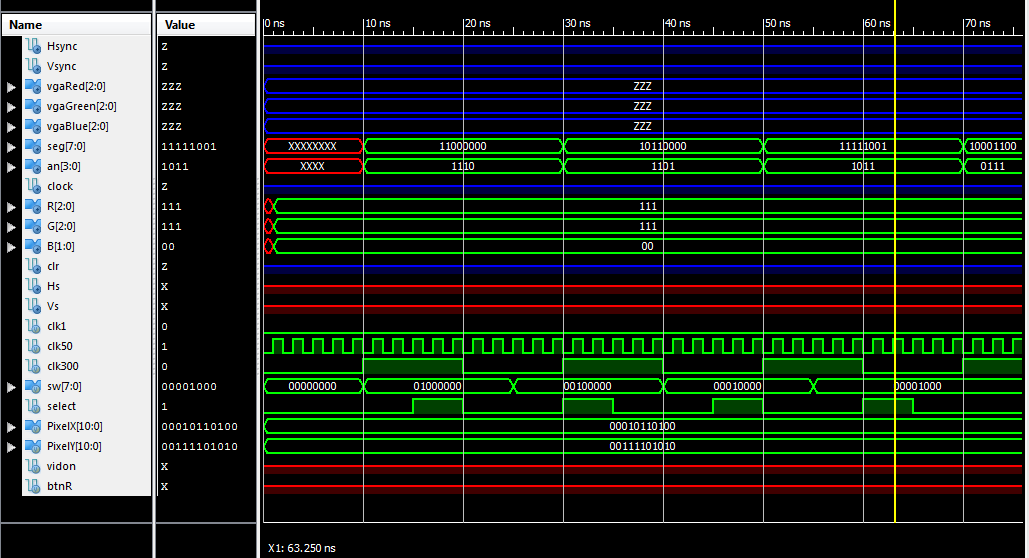


*Figure 2.3* iSim window with the display statements shown on the left. We outputted every time select is hit and a piece is dropped into the board in order to check the array and ensure that blockstate and colorstate were updated properly each time.

We were getting undefined behavior from the seven segment display after changing timer.v around. We suspected that this came from the inability of timer and connect4 to communicate changes to the game state to each other. Like mentioned previously, we ended up putting most of timer’s functionality into connect4 to avoid this.

Something that perplexed us was why our display was so dimly lit. The colors were all correct, but it seemed that the physical display was off. Initially, we passed it off as a hardware issue, but after trying a new monitor, took a look at our top module (nexys3.v). Everywhere else in the project, we used R, G, B variables as multiple bits, but when actually using the wires in nexys3 to instantiate the connect4 module, we truncated them to one bit. By changing these from wire x to wire [y:0] x in order to match the number of bits, we fixed the issue.

After fixing some of these primary issues, we created a more comprehensive test bench that simulated a winning scenario. This required several moves and we monitored the signals on the iSim below.



*Figure 2.4* The final iSim of our project.

**Conclusion**

For our final project we created Connect 4. Instead of the traditional 8 by 8 board, we used a 7 column by 6 row board. The classic rules were implemented: take turns dropping pieces into one of the 7 columns, one piece at a time, and the first player to get four in a row wins. An extra twist we added was that each player had 30 seconds for each, displayed on the seven segment display. The board can be reset using a separate button. One hot encoding of the first seven slider inputs is used for selecting the column to drop a piece into. All graphics of the game board are drawn using the VGA connector.

A major source of confusion for us was how to declare arrays. We did not know that in order to use 2D arrays, we needed to use an updated version of Verilog. Instead of doing this, we just consolidated our data and made a 1D array.

The only suggestion we would have is to perhaps allow open lab time more often during the final weeks of the class, as many of the things we worked on in the project could not just be done in simulation and we needed the actual lab equipment.