Final Project
RGB LED Connect 4 Game

12/11/2016

Nick Vaughn
&
Caleb Rash

Problem:

The goal of this project is to create a digital version of Connect 4 where the board is shown with an RGB matrix display. The game is to be controlled by tactile switches on a cyclone FPGA board. The players' turns alternate after each move and each player drops differently colored coins. Player 1 has the first turn. The game should also self-check for win scenarios for each player. The win scenarios should be 4 of the same type of coin in a vertical, horizontal, or diagonal row. In addition, the board should not allow invalid moves, such as moving the coin off the grid or dropping the coin on a full column. If the move is invalid, the player's turn should not alternate until a valid move is performed. Finally, the board needs to have a reset switch, which resets all values and messages as well as re-initializing the display to show grid lines.

System Design:

A Cyclone IV E board was used to synthesize the hardware needed for the system. Three tactile switches on the board were used to represent the user interface left, right, and enter. A toggle switch on the Cyclone was used to activate the active low reset signal to restart the entire game. The external hardware that was used for the system was a 32x32 RGB LED matrix from Sparkfun. It was connected to the Cyclone board's GPIO pins in order to provide it the signals it needs to generate the LED patterns for the game's display.

In order to achieve full movement functionality, a module was needed to store and adjust the column the player's coin was located, with left or right placement, before the coin was dropped. A Moore Finite State Machine, called the FSM control column module, was used to keep track of the player's movement on the board. The column control FSM, seen in figure 8 below, determines where the player is before the coin is dropped. All that it must do is keep track of where the player is to be moved or not moved depending on whether the right or left button was pushed. Figure 1 below shows the state diagram of the FSM. It lays out the condition that the player may go right until it is in the right most column and that the player can go left as long as he/she is not in the start column or the left most column.

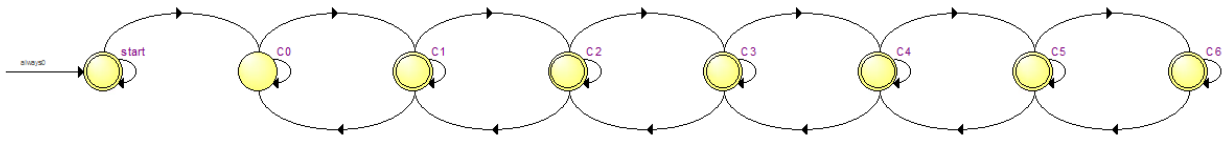


Figure 1 - state diagram of the column control module

In addition, a control module was used to check that the placement of the coin was a valid move before allowing the coin to drop when the player pressed the enter button on the Cyclone board. The control module was designed to modify the write address entering a memory module to adjust the "height" of a coin in a certain column. The placement of a coins write address inside the control module was designed to depend upon how many coins had already been dropped in that particular column. For example, if there were no coins in a column, the control module would give the entered coin a write address that placed it to the bottom of the displayed board. However, if there was already a coin placed at a column, the control module was designed to place the coin above the previous entered coin by sending the proper write address. If any column is completely filled with coins, the control module does not allow the player to enter the invalid move. The control module is also responsible for sensing when a win condition occurs from the win checking module, and passing the correct player win flag (player 1 or player 2) to the memory unit.

A memory unit was connected to the control module in order to store values for the game. This was implemented with a 64 by 32 memory block which stored 32 "pixels" vertically and 32 "pixels horizontally to represent the 32 x 32 RGB matrix. There were two bits per pixel to represent 4 different color states (off, purple, light blue, and white). That is why the memory entries are 64 bits wide. The memory unit reads off of a .mif text file at the start of the game, which initializes the white grid lines needed for the board. The memory unit also has pre-programmed win messages that are written to memory if the win flag is received from control. This way it can display a win message for the individual player that won the game. Since there were combinational and additional flip flop blocks that executed functions outside of what a typical memory unit performs, the unit was not inferred memory. Instead logic registers were utilized in the place of RAM. When the reset switch is activated for the system

A win module was used to check for win scenarios inside of the game. This module was designed to read off of the control module's write addresses and column values as it passes them into the main memory unit. It builds and modifies its own matrix from the control module's data, which is a smaller representation of the full memory matrix. The win module is also designed to then scan its own smaller copy of the board matrix for win scenarios of either player. It utilizes combinational logic to scan for diagonal, vertical, and horizontal win scenarios in each spot of the board. If one player had 4 in a row then it would send a win flag to control. When the control module receives a win flag, it checks to see which player's turn it is and can determine who won. This way, the control can send a flag to memory so that memory can write in an appropriate win message for player 1 or 2. When the active low reset signal is activated, the win module clears its board signal with zeroes to assist in restarting the game. The flowchart of the win module is shown below in figure 2.

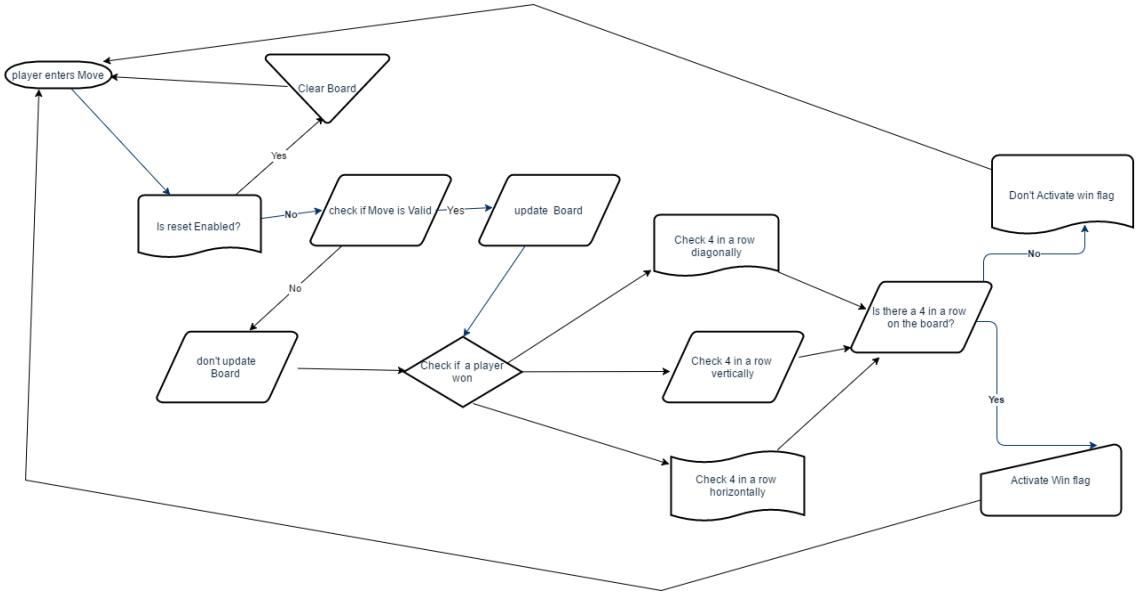


Figure 2 - FlowChart of the Win Module

The last module implemented was the display module, which connected to the memory unit and the RGB matrix. The RGB matrix operates by lighting up a single upper and lower row of leds at a time. It then cycles through all the rows quickly to make it appear that all the RGB matrix LEDs are lighting up simultaneously. The lower row is always 16 lines below the upper line, and the upper line loop from row 0 to 15.

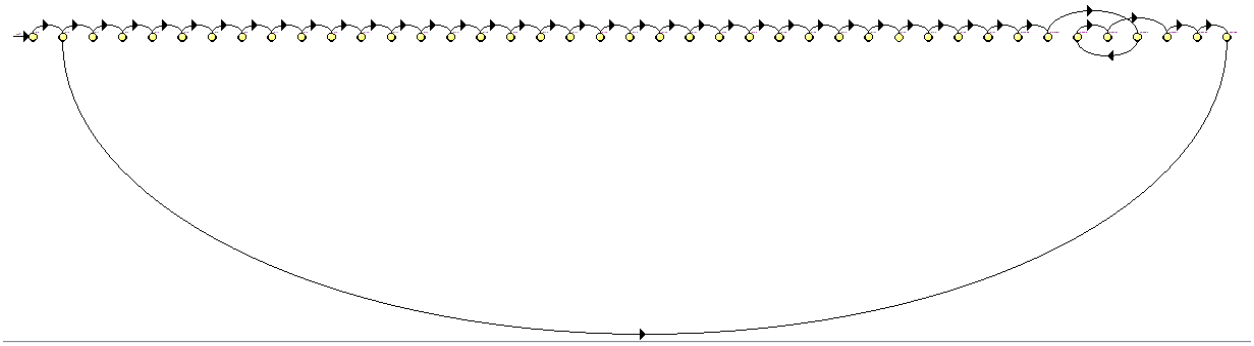


Figure 3 - state diagram of the display module

The state diagram for the display is seen above in figure 3. The module loads in 32 bits of data for each color pin on the matrix (R1, G1, B1, R2, G2, B2). After this, the loading register is disabled as well as the clock signal. Next, the address is incremented. The data is latched to the display register and the clock is restarted as the process repeats. The block diagram for the system is shown below in figure 4, and the overall flowchart of the design is shown in figure 5.

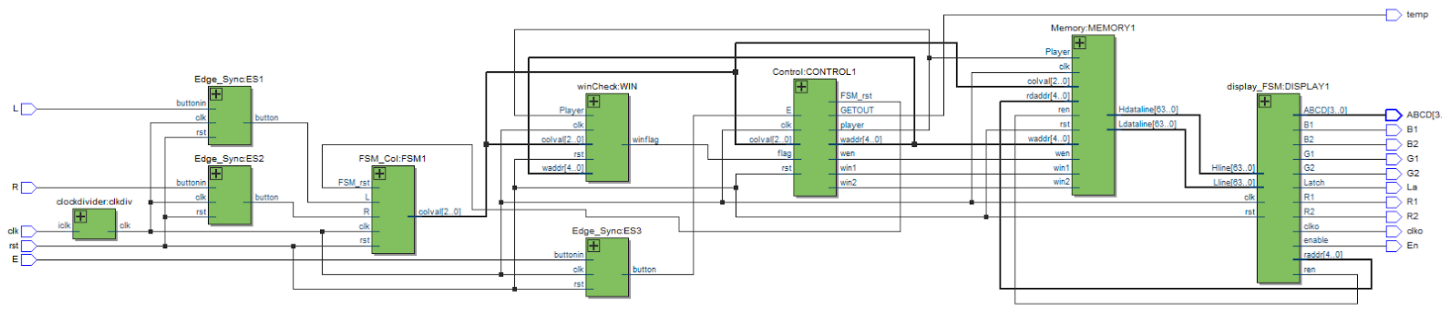


Figure 4 - Block Diagram of the complete system from the RTL viewer

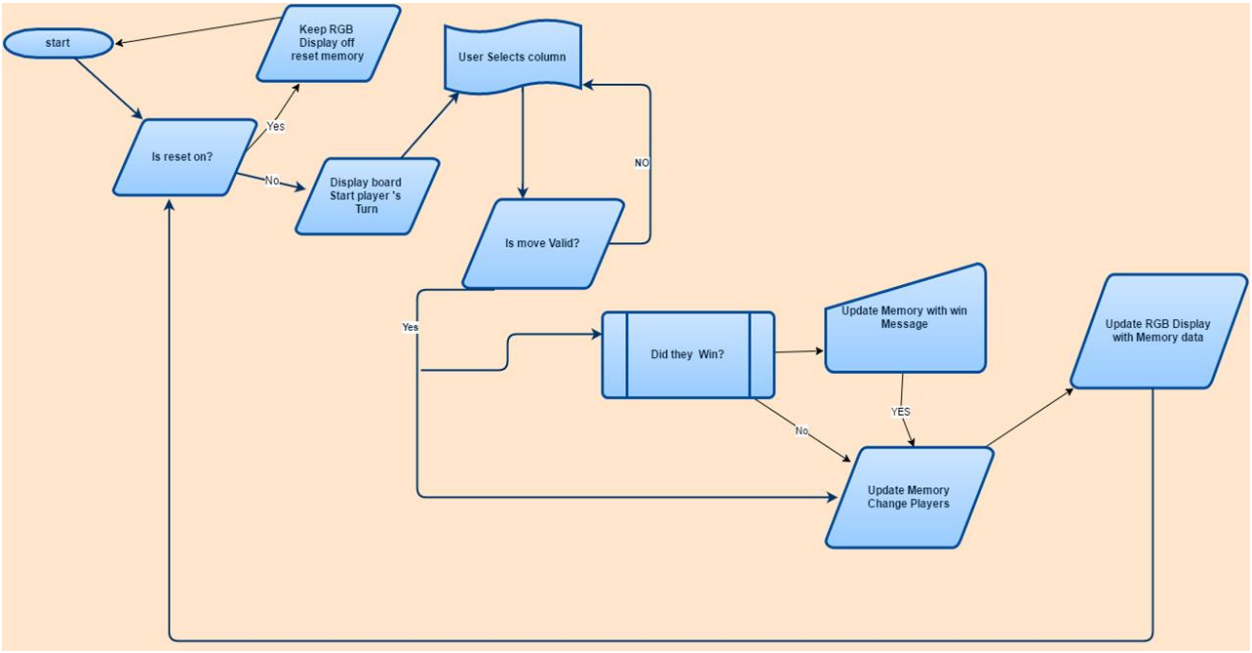


Figure 5 - Flowchart of the overall design

For the matrix to work, 6 bits needed to be loaded in per clock edge 32 times to light up 2 rows at a time. This is because there are loading 32 bit shift registers for red, green, and blue LEDs for the upper and lower rows of the display; this means that 6 bits are needed per edge of the clock (1 bit for each color). These bits determine whether that particular LED is on or off. The sets of 6 bits are loaded 32 times to fill all columns of LEDs in that row. There are other registers used to display the LEDs, which are then disabled along with the incoming clock signal. The address of the row is then incremented, and the data from the loading registers are latched into the display registers. When the data is valid, the display registers and the clock signal are enabled in order to display the freshly loaded data. After this, the cycle repeats and continues.

Additionally to all of this, there are synchronizers that are attached to the buttons on the board for Left, Right, and Enter. The synchronized output of Left and Right go to the column FSM module to determine the column where the coin is to be dropped. The synchronized Enter signal is then passed to the control module to drop the coin.

Testing Approach:

The testing of the system was simulated with separate ModelSim test benches for each module aside from the synchronizers. Once the test benches were simulated for each module, the Quartus software was then used to check if the hardware coded was synthesizable.

The FSM and control modules were both analyzed using randomized self-checking test suites. The FSM module randomized the left and right inputs that the user would trigger by pressing the buttons on the Cyclone board, and confirmed that the columns changed to the correct position in the simulation. The control module used the same logic, except it randomized the column values, and simulated the incrementation of the write addresses would happen in response to the enter button being pushed. The randomization provided all the possible button combinations that the user could accomplish on the board. This covered more than what a direct testing method could have done, because it generated enough cases to provide a larger area of coverage over the conditions that could occur.

The win module was tested by entering the board with values that produced 3 different kinds of win cases: diagonal, vertical, and horizontal. By using randomization within the test bench, over 1000 cases were generated to test the module with all possible inputs. The write addresses and column values received from the other modules were simulated to ensure that there was proper translation to the row and column values stored inside the board. The player value was also fluctuated to simulate when a player changed their turn during the game. This test was efficient enough to ensure the module design was functional, because it covered all the different unique win scenarios that could be achieved inside of the game. A direct test method was used to ensure that the module did not produce win flags in scenarios that did not contain four in a row.

The memory module was tested by writing values directly inside it through the test bench, and reading the outputs. Randomization was not necessary for testing this individual module, because it only needed to prove that values were being stored inside it properly. Due to this reason, a direct testing method was used for the memory module. Data was inputted inside the module, and observed through ModelSim's waveforms.

The display module was created using a direct testing method, because the real testing was observing the RGB matrix itself light up with the proper LEDs. However, the test did confirm that the address signals were sending the inputted data lines correctly to the shift registers, and that the latch and enable bits were activated appropriately. Self-verification and randomization was not used solely because the bulk of testing this module was involved with running it on the RGB matrix itself.

The top-level module of the system was tested with a randomized test suite. The left, right, and enter buttons were simulated being triggered in order to examine the signals being outputted to the RGB matrix. This test was sufficient for the design, because it implemented every possible move that could be made while interacting with the game. In addition, the signals of the inner modules were observed in ModelSim to ensure that the complete testing of the system was as functional as the individual tests of the modules.

Results:

The entire system was successfully simulated with the ModelSim software, and provided the required signals for the RGB LED matrix. Each individual module confirmed their functionality in their individual test benches. The win module's transcript and simulation are below in figures 6 and 7.

```
# The Vertical Test player WORKS! Player:         1 row:         2 col:         0
# The Horizontal Test player Works! Player:         0 row:         5 col:         2
# The Diagonal Down Right Test player WORKS! Player:         1 row:         2 col:         1
# The Diagonal Up Right Test player WORKS! Player:         1 row:         3 col:         0
# The Vertical Test player WORKS! Player:         1 row:         1 col:         1
# The Horizontal Test player Works! Player:         1 row:         5 col:         1
# The Diagonal Down Right Test player WORKS! Player:         0 row:         2 col:         1
# The Diagonal Up Right Test player WORKS! Player:         0 row:         4 col:         1
# The Vertical Test player WORKS! Player:         0 row:         0 col:         5
# The Horizontal Test player Works! Player:         1 row:         4 col:         2
# The Diagonal Down Right Test player WORKS! Player:         0 row:         1 col:         3
# The Diagonal Up Right Test player WORKS! Player:         0 row:         3 col:         1
# The Vertical Test player WORKS! Player:         0 row:         2 col:         6
# The Horizontal Test player Works! Player:         0 row:         3 col:         1
# The Diagonal Down Right Test player WORKS! Player:         1 row:         2 col:         0
# The Diagonal Up Right Test player WORKS! Player:         1 row:         3 col:         1
# The Vertical Test player WORKS! Player:         0 row:         0 col:         3
```

Figure 6 - Transcript window running tests on Win module

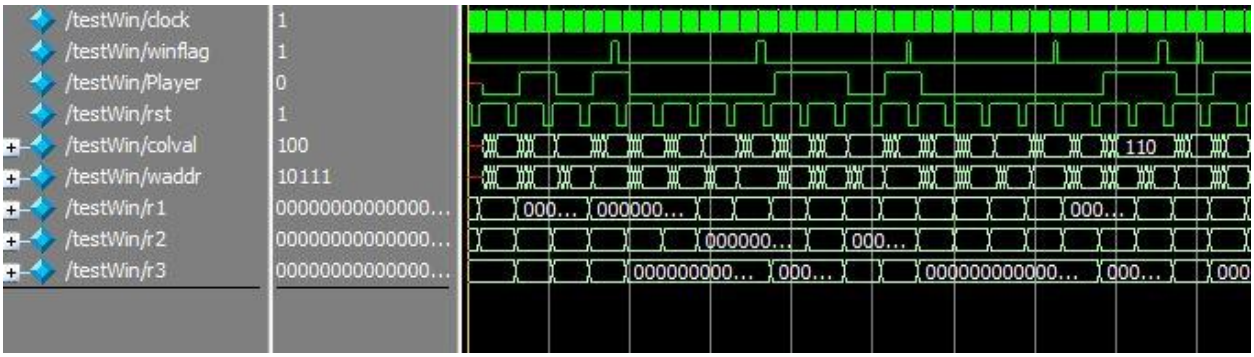


Figure 7 - Simulation of Win module

The win module simulated all the possible outcomes of the win scenarios, and showed that the win flag signal was being triggered at the correct times. Doing a randomized test suite verified every scenario, and saved time from having to debug the win module.

The win module had to be updated a few time during the project due to formatting issues with the memory. In addition, there was a problem with running the simulations, and synthesizing the hardware inside Quartus. The original designs for the win module would simulate in ModelSim, but infer latches inside of Quartus afterwards. To fix this, the design was implemented to ensure that signals were not being used by two different hardware block simultaneously.

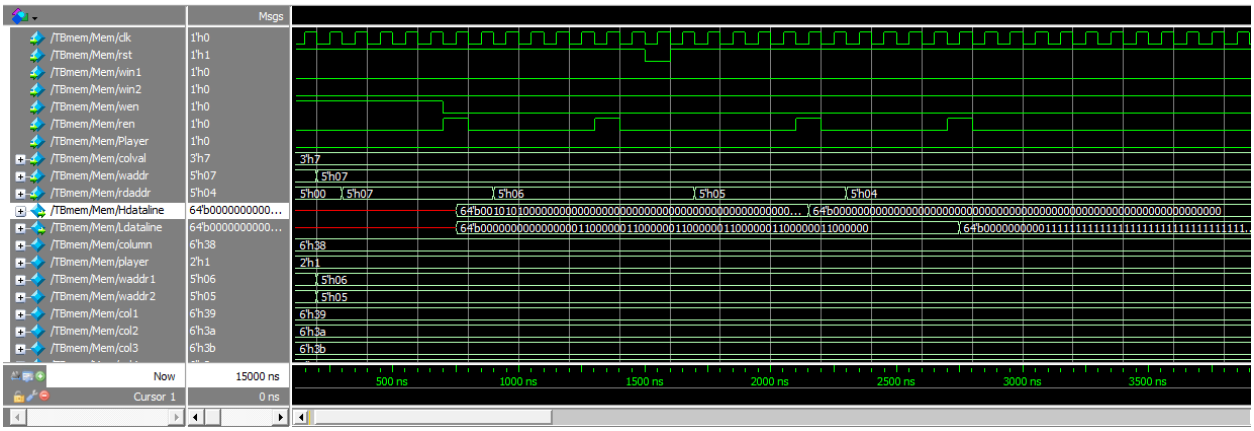


Figure 8 - Simulation of memory module

The memory module was tested by writing in data into memory and then reading the data at the address of the write. The data shows the written data correctly. As can be seen above in Figure 8, data for player 2 was written into the "start" column, which is two bits from the edge of memory which is reflected in Hdataline. The Ldataline shows the grid lines with code "11" for white pixels spaces 6 bits/3 pixels apart. The next address below the first, shows no data in Hdataline because there was no data written here and the Ldataline is all 1's past the leftmost blank columns because it is a solid white horizontal grid line from initialization.

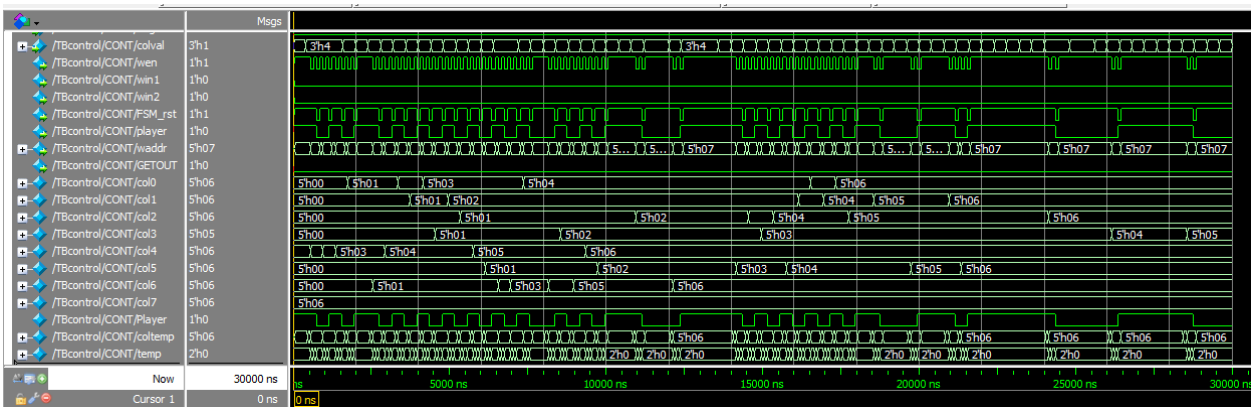


Figure 9 - Simulation of control module

```
#       TBcontrol
# End time: 20:42:15 on Dec 12,2016, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# End time: 20:42:16 on Dec 12,2016, Elapsed time: 0:25:33
# Errors: 0, Warnings: 5
# vsim -coverage TBcontrol
# Start time: 20:42:16 on Dec 12,2016
# Loading sv_std.std
# Loading work.TBcontrol
# Loading work.Control
# ** Warning: (vsim-WLF-5000) WLF file currently in use: vsim.wlf
#           File in use by: crash  Hostname: MDSOECSLABS-17  ProcessID: 12436
#           Attempting to use alternate WLF file "./wlft5srmh1".
# ** Warning: (vsim-WLF-5001) Could not open WLF file: vsim.wlf
#           Using alternate file: ./wlft5srmh1
# .main_pane.structure.interior.cs.body.struct
# .main_pane.objects.interior.cs.body.tree
# .main_pane.wave.interior.cs.body.pw.wf
# 0 ns
# 31500 ns

VSIM 4>
```

Figure 10 - Transcript window of control test bench

The control module test bench randomizes the coin drops by selecting a random column and simulating an ENTER button press afterwards. The transcript window shows no assert messages, which means that the output successfully passed the test bench's check assert block. This can be seen above in figure 9. The 2 warnings seen in the transcript window are likely from the virtual machine which was used for simulations. The shared memory on the VM filled up and despite trying to allocate more memory aside the error/warning persisted. However, the data from the test bench is still good and performing as expected. The FSM column control module's simulation is shown below in figure 11.

Figure 11 - Simulation of randomizes left and right movements in the column control module

The FSM column control module performs all of its desired functionality as seen above in figure 11. The player's column value increments upwards and downwards as well as stopping at the boundaries set inside of the game. The display module's output to the RGB matrix is shown below in figure 12.

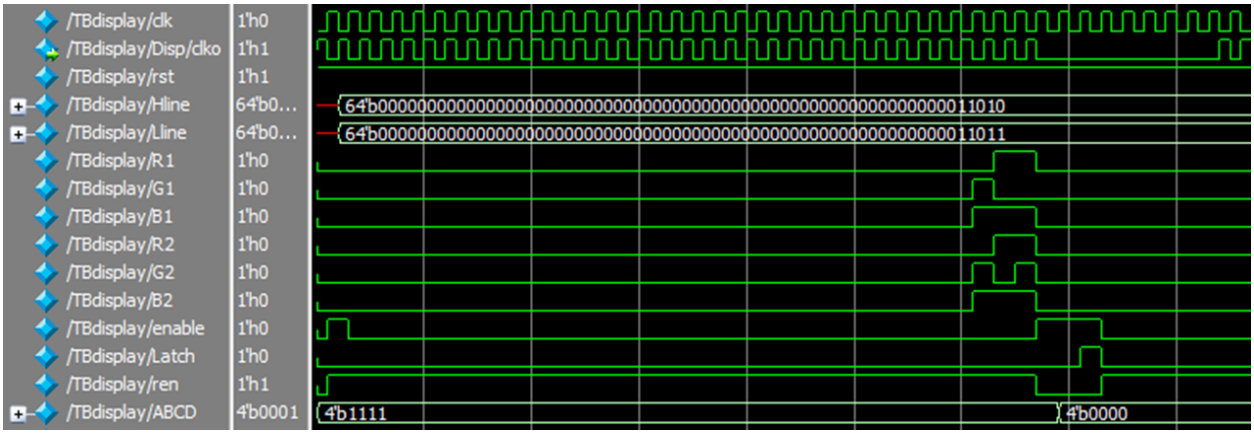
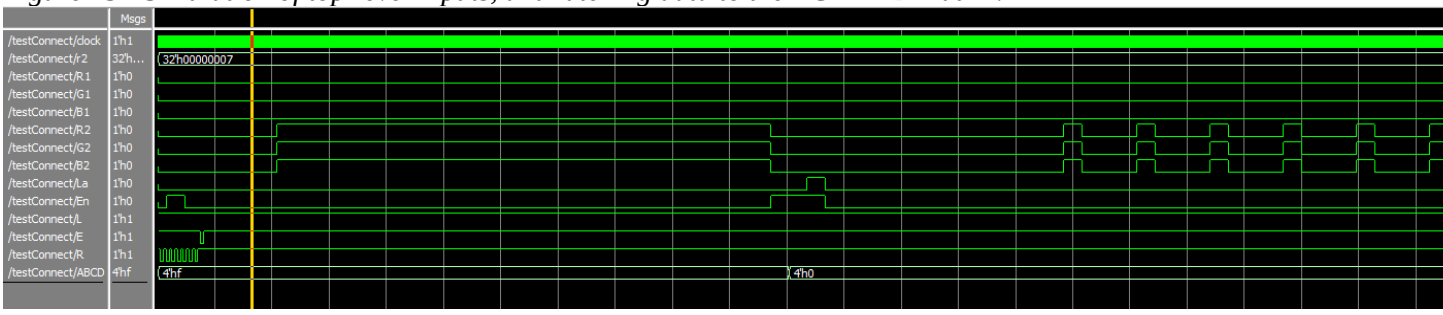


Figure 12 - Simulation of Display module

The upper and lower data lines given to the display provided the values for the LED positions and corresponding colors on the RGB matrix. Figure 4 shows an example of the colors cyan, purple, and another purple being latched to the last 3 RGB matrix pixels for the upper data line by passing values into the 32 bit shift registers for the colors red, green, and blue. In figure 4, the lower data line passes in cyan, purple, and white RGB pixels at the end of the lower data line.

Many obstacles came up with connecting the display module properly to the RGB matrix. The first problem that came up involved the transconductance of the clock signal with the RGB matrix. The board was originally trying to send a 50MHz clock to the RGB matrix, which was an unrealistic thing to expect over a 6-inch wire. The Oscilloscope showed that the clock signal was being distorted over the wire. This cause data to be latched to odd places on the RGB matrix. To fix this problem, a clock divider module was used to bring the clock speed down to 1.5MHz instead. In addition, since there was no documentation ever published with the device being connected to a FPGA board, reverse engineering tactics had to be used to uncover the protocol the board followed for communication. An Arduino microcontroller C-language library was discovered with definitions already created for communicating with the RGB matrix. By using a logic analyzer, the protocol used between the Arduino and the board was able to be observed. Using the information gathered from the logic analyzer, the display module was redesigned to mimic the behavior the Arduino used with the RGB matrix. The results for the overall system are shown below in figure 13.

Figure 13 - Simulation of top-level inputs, and latching data to the RGB LED matrix.



The top-level module of the design simulated successfully, and provided all the necessary signals the RGB matrix required. The user's signals for left, right, and enter provided the proper display signals for the game to function correctly.

The game is fully functional, and runs the entire game without any errors or flaws inside the display. Each player was able to insert their coin represented by a color block to the RGB matrix board. All blocks were able to go inside of any column, and the blocks properly stacked on one another in different rows. The self-checking logic was able to sense all three win scenarios (diagonal, vertical, and horizontal), and the customized win message for each player was projected on the top of board to indicate the winner. An example of the completed system when the win flag is triggered is shown below in figure 14.

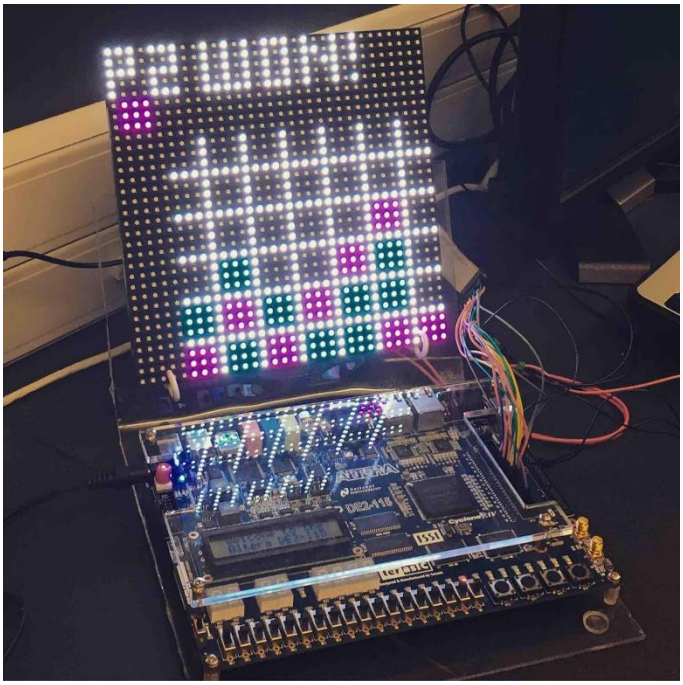


Figure 14 - example of the win flag being triggered by player 2

Analysis:

Overall, the design worked exactly as planned, and efficiently executed all the features required by the problem statement. With memory not being inferred, it was easier to do the reset logic to restart the game. It also made it less complicated to fill out the coin values need for the RGB matrix LEDs. Displaying the correct sequence on the RGB matrix was the biggest challenge of the design, but reverse engineering the Arduino fixed the problem entirely. The system's spacing requirements are shown below in figure 15.

Total logic elements	1,200 / 114,480 (1 %)
Total combinational functions	1,157 / 114,480 (1 %)
Dedicated logic registers	464 / 114,480 (< 1 %)
Total registers	464
Total pins	19 / 529 (4 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 15 - Spacing requirements for the entire design.

The entire design used 1,200 logic elements, and 464 of those elements were logic registers. This is logical given the fact that the design did not infer memory properly as most design would have. This design was not optimized for spacing in general. Many opportunities could have been taken to make the hardware more efficient with styles such as flattening logic, or unrolling. To optimize the 464 registers, the design could have utilized RAM more. Additional modules could have been created to perform the tasks that were stuck inside memory, and preventing memory from being inferred. Time constraints however, influenced most the design decisions when it came to spacing. The timing analysis and longest path of the design are in the figures 16 and 17 below.

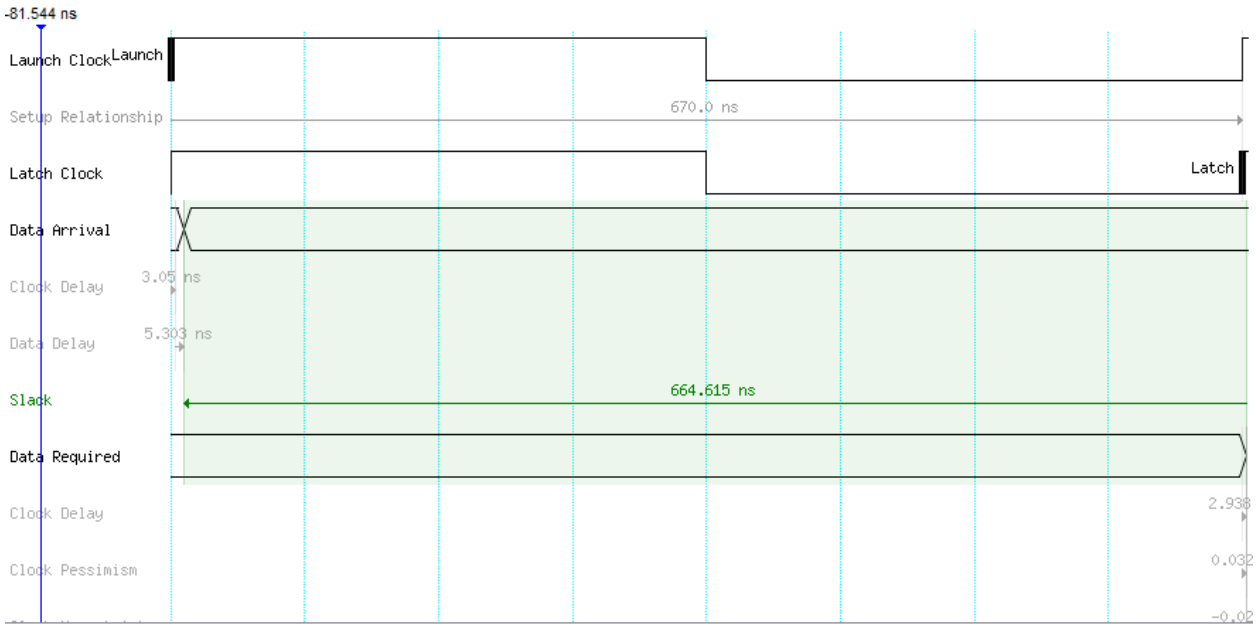


Figure 16 - Timing Analysis of the Complete System

Command Info | Summary of Paths

	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	664.615	clockdivider:clkdiv|clkstate	clockdivider:clkdiv|clkstate	Clock1	Clock1	670.000	-0.080	5.303
2	665.265	clockdivider:clkdiv|count[16]	clockdivider:clkdiv|clkstate	Clock1	Clock1	670.000	-0.078	4.655
3	665.284	clockdivider:clkdiv|count[19]	clockdivider:clkdiv|clkstate	Clock1	Clock1	670.000	-0.078	4.636
4	665.289	clockdivider:clkdiv|count[13]	clockdivider:clkdiv|clkstate	Clock1	Clock1	670.000	-0.078	4.631
5	665.294	clockdivider:clkdiv|count[12]	clockdivider:clkdiv|clkstate	Clock1	Clock1	670.000	-0.078	4.626
6	665.295	clockdivider:clkdiv|count[26]	clockdivider:clkdiv|clkstate	Clock1	Clock1	670.000	-0.078	4.625
7	665.308	clockdivider:clkdiv|count[24]	clockdivider:clkdiv|clkstate	Clock1	Clock1	670.000	-0.078	4.612
8	665.309	clockdivider:clkdiv|count[18]	clockdivider:clkdiv|clkstate	Clock1	Clock1	670.000	-0.078	4.611
9	665.317	clockdivider:clkdiv|count[25]	clockdivider:clkdiv|clkstate	Clock1	Clock1	670.000	-0.078	4.603
10	665.318	clockdivider:clkdiv|count[15]	clockdivider:clkdiv|clkstate	Clock1	Clock1	670.000	-0.078	4.602

Figure 17 - Longest Path inside the Complete System

As shown above in Figure Y, the design has a propagation delay (slack) of 664.615 ns. This make sense given the fact that the system is designed to run with a clock speed of 1.5 MHz. The maximum clock speed the system can run is (1/5.303ns) 188.6 MHz. Figure Z shows that the longest path within the system is located inside the input of the clock divider module to the output of the clock divider module, with a propagation delay of 665.318 ns. This is most likely because the project utilized many flip-flops between most of the combinational logic. The amount of flip-flops inside of the original design without the clock divider must have cause the combinational logic to have less propagation delay than what was contained inside the clock divider module itself. The design has a total latency of 9 clock edges, and a throughput of 32. The latency is from the amount of flip-flops connected to get towards the output of the system, and the throughput was calculated by examining how many clock edges are required to fill out entire row on the RGB matrix.

In the future, it is planned to have the project perform the game while having sound effects activated for each of the button presses. Signals can go high, and activate mp3 files when the player performs a certain action. This would make the game more fun for younger audiences. Also another additional feature than can be added to this project is animation to the coin drops on the board. Every time a player enters a move, an FSM can be used to show the coin flashing though other spots on the board vertically until it reaches the correct one. This would give the game a closer feeling to the original connect 4 game it was modelled by. When someone wins the game on the current design the win message appears, but player are still able to fill in spots afterwards. Another feature that could be implemented inside of the game is a game over state, that clears the board and has the user select whether or not they want to play the game again.

Task Breakdown:

Nicholas Vaughn's main goals were creating the self-checking win logic inside of the design, and testing the overall design. He created the win module, and the test benches required to confirm its functionality. He also assisted in designing the memory module, and came up with the concepts of how to connect the data between the win and memory modules. He researched the Arduino libraries needed for reverse engineering portion of the project, and assisted with interpreting the protocol needed to communicate with the RGB matrix. He designed the final test bench for the overall project, and confirmed the simulation's functionality for the entire design. After his portion simulations were confirmed, he assisted Caleb in any debugging issues that were left with the Display. He also documented the team's progress, and created the content necessary for project presentations, and reports.

Caleb Rash's main goals were creating the display module for the game's board, and implementing the control logic for the design. He created the memory unit, the column FSM, the control unit, and the display modules needed for the system. He also provided test benches for all his individual modules, and confirmed the functionality of each one. He researched the methods that were used to store

data on the RGB matrix, and came up with structure of the design that was needed to communicate through the Cyclone's GPIO pins. He constructed the top-level module that was used to implement the entire design properly. He wired the Cyclone IV E board, and created a Plexiglas stand used to present the project for peers. He implemented the reset feature of the design that allowed the board to play multiple game sessions. He also figured out how to implement the various colors shown on the RGB matrix with the display module.

Conclusion:

The design was able to fulfill all the expectations that were set out in the beginning of the design process. It is not as optimized with its hardware as it could potentially be, but it meets all the functionality one would desire from a connect 4 game. The RGB matrix was largest challenge the team had with the design. There was no guarantee that the display would function to how the team imagined it. Therefore, features such as animation for the game had to be moved to the future work category. Realizing the time constraints helped the team make more realistic goals, and thus contributed toward the overall success of the project. There were a few problems with design style that occurred in some of the modules. One teammate would envision solving the problem a certain way, while the other teammate attempted to solve the problem another. Keeping clear communication within the team prevented this from becoming a problem, and strengthened the efficiency of the group. When it comes to testing designs, team organization is key to reaching peak performance. Having team members work on individual tasks is much more efficient than having the entire team stay stuck debugging a single design flaw. Test benches are not just useful for on verify modules for themselves, but they also provide clarity to teammates that need to understand how certain parts of the hardware work. Having the entire team working in the same mindset, helps bring a design together. Any group project can be done with proper time management, and organization within a team.