Connect 4

Jonathon Glover, Ryan Sherry, Sony Mathews and Adam McNeily
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails:jvglover@oakland.edu, rgsherry@oakland.edu, smathews@oakland.edu, armcneil@oakland.edu

***Abstract*—A Connect 4 emulator that is displayed on a VGA monitor. The game works with two players and is capable of understanding who has won and various other processes with the storage of data in an array of registers. The purpose of this game was purely for entertainment. The code was challenging to integrate into a VGA controller due to the dynamic image processing from the screens horizontal and vertical sync. I would recommend making this game in software with a GUI instead of hardware descriptive language through a FPGA and VGA display interface. Overall this project was an excellent learning experience and the project itself incorporated almost every aspect of combinational and synchronous circuitry learned within the ECE 378 class.**

I. Introduction

This report will cover the coding structure used to display a fully functional Connect 4 game onto a VGA monitor. The report will go into great detail as to how the three main components used in the top file of the Connect 4 game work together. Said components are; IndexState, Compute and Graphics.

A very important aspect of this game was the use of state machines and combinatorial logic circuits to understand where the user wants to drop their piece during their turn and then seamlessly displaying this data onto a 6x7 grid on the VGA monitor. The game is also capable of knowing and displaying if Player 1 (or Player 2) has won .

Added feature to this game is being able to change the default colors of red and blue to shades of orange and pink (for red) and shades of teal and purple (for blue). This additional feature gives the game a more visually appealing and versatile playing experience that can accommodate the already well known and loved game of Connect 4.

II. Methodology

The major task to begin making this game was the design of the data path circuitry. This first task was the most fundamental step in beginning the design stage of the game, as it laid out the foundation of which the code was meant to be written. The idea of the game was very simple and the main objectives to create a visually working and logically correct game. Where the workload of the game was divided into three main groups. The index selector, computation and graphics output components.

The operational concept of this game was meant to flow in this order;

1. Have a home screen displaying the games title and wait for the two players to press begin.
2. Have an easy to use interface with left, right and drop as well as be able to change the coloring of the player's desired game piece.
3. Have the players take turns playing the game until completing the game.
4. Have the game understand when a player has won, and present a screen displaying which player was the first to place four of their pieces in a row, either horizontally, vertically or diagonally.

In the following subcategories the components; IndexSelector, Compute and Graphics are comprehensively analyzed to allow you the reader to understand the grand scheme of how this game was built and better the readers understand of what went into making this game function. See below a simplified top level block diagram that displays the core components and their interconnections within the top file. Also note that the debouncer components are mainly for eliminating mechanical bounce for active high button inputs and play no role in the logic functions of this game.

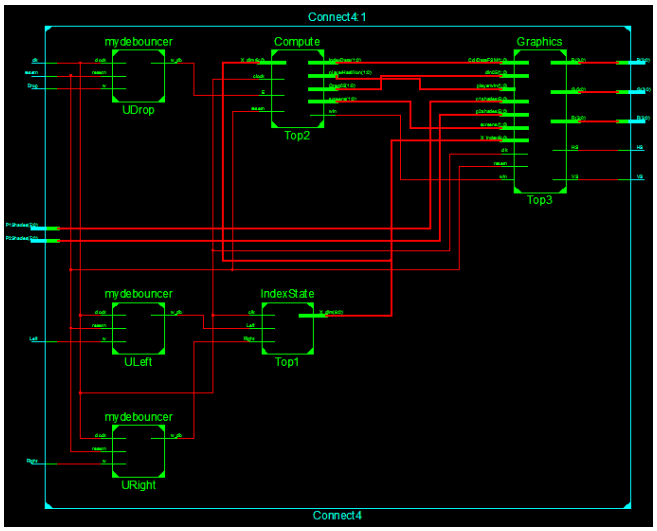

Figure 1. Top level simplified block diagram.

A. *IndexSelector*

When playing connect 4 there are seven possible columns of which the user can choose to drop their piece. The function of the IndexSelector was to allow the user to visually see what row they are hovering above by sending data signals into the Graphics component, as well as sending this data into the Compute component so that it understand which column that the user is going to drop their piece into. The input data comes from two main sources which are active high buttons on the Nexys 4 FPGA.

In the top file these buttons are known as Left and Right. The IndexSelector is a Moore state machine that responds to these Left and Right inputs by comparing them to the current state and the positional movement of which the user has entered. The output consists of a decoded signal. For example, if the user has chosen to move his/her piece into the Left most Index, this is known as Index0. When in Index0 the output consists of a 7-bit number of "0000001" which only outputs a high for the most significant bit that the index represents. Another example would be that the user has chosen to begin at Index0 and press and release the Right button six times. This sequence of inputs will cause the IndexSelector to be in a state known as Index6 and the corresponding decoded output will be "1000000".

Below is a state machine describing the input logic used in the state machine as well as all possible outputs. Notice that there are sub states in between each index. For example X_01R this is an in-between state which represents the movement from Index0 to Index1 by pressing the Right button input. This state acts as a wait state and waits until Right has a value of '0'. This functionality prevents the Index from being swept from 0 to 6 instantaneously.

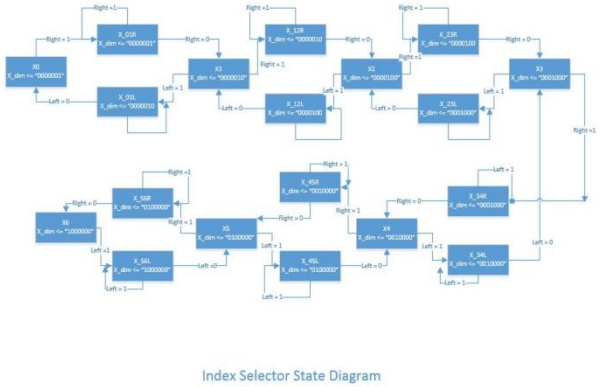

Figure 2. Index Selector Moore State machine.

B. *Compute*

The compute file is the heart of this program. It takes the decoded 7-bit data stream from the IndexSelector about the players column selection and another active high input from the top file known as Drop, it controls visual screen states of the game such as the; Home screen, Gameplay Screen, player1 wins screen and player2 wins screen. The compute file does this through three main components and an array of 42 registers. These three main components are called PlayerSwitcher, RowSelector and Winner. Below is a very simplified image of this file consisting of only one register(instead of 42). As you can see, the inputs to the compute file are the clock, resetn, X_dim and E (for dropping piece).

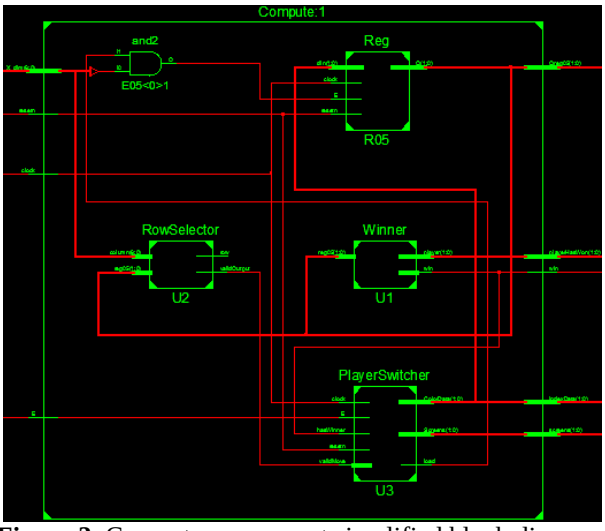

Figure 3. Compute component simplified block diagram.

In order to fully describe this logic circuitry it is necessary to first describe the components mealy state machine sub-component known as the PlayerSwitcher. This components main objective was to answer the following questions. Is it in the home screen, gameplay or a winner screen? Whose turn is it to play? When the user wants to drop their piece, how do I save that players data into the correct register and when am I allowed to do that? These questions were developed into an algorithmic state machine which takes signals from the RowSelector component called "valid move" which tells the state machine "yes you can place a piece in this column if it is not full" and from a signal called hasWinner which tells the state machine "No player has won this round".

The Winner and rowSelector components do both of these tasks by taking in 2-bit data inputs from each and every one of the 42 registers contents. Where "00" represents empty, "10" represents player1 has filled this space and "01" represents player2 has filled this space. Once the player has decided upon a row and pressed enter, the state machine outputs either a "10" or a "01" depending on whose turn it is for its signal called ColorData as well as a '1' for its signal called Load.

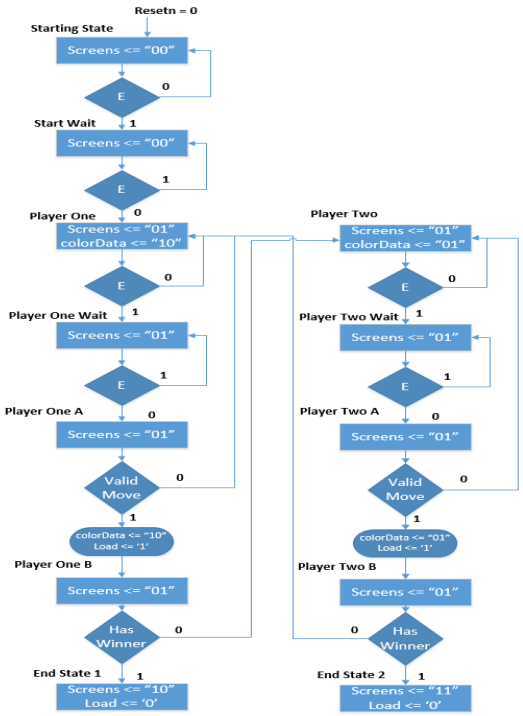

Figure 4. PlayerSwitcher State machine.

At the same time the rowSelector component understands that the user has decided to choose whichever column they are in by looking at the X_dim value for the current IndexState of and then it outputs a decoded signal representing the first row, starting from the bottom up, that is empty (has a corresponding register value of "00"). The row selector does this with 7 case statements representing each possible index position, whilst inside each of these case statements it processes an if statement that logically checks to see which row in that column is the smallest y_index position and is empty. Once it has found its first empty row a decoded output signal representing the most significant bit of that row is output.

For example if the bottom row of that players column is empty the rowSelector output will be a 6-bit number with a value of "000001", and if the first 5 rows of that column are filled and the top row is empty the rowSelector will output a "100000". This data is used as the Enable for the corresponding, as well as corresponding X_dim input and the load signal from the PlayerSwitcher Component. See below a representation of the Enable for register position 00.

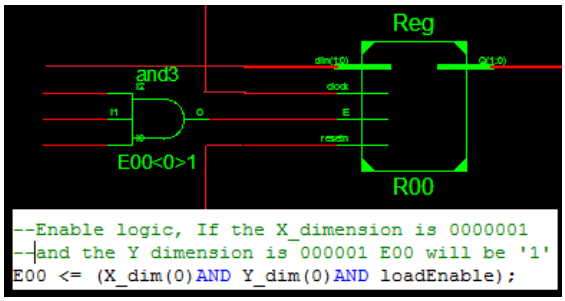

Figure 5. Compute component register enable.

The reasoning behind an enable that is based off of a 3 input AND gate was done like this because the registers enable was only supposed to take inputs of the players current colorData if and only if the user has chosen and selected that exact register from 42 possible choices. The colordata signal and the load signal from the PlayerSwitcher state machine are fed into each registers enable and dataIn inputs.

After all of this is said and done the register contents are now updated and are sent into a component known as the Winner component. This component logically checks the contents of every single register and compares them with any of the 69 possible ways to win in Connect 4. The "brute force" algorithm used as a reference to code this component can be found in the reference section listing [2].

Once all of the possibly ways have been checked, the Winner component outputs a '1' if a player has won or a '0' if no player has won this round. The signal is sent into the PlayerSwitcher component and used in a decision diamond for the next state. (Either player win screen or next players turn).

The final process that the Compute file does is outputting the 2-bit data contents of each register as well as a signal from the PlayerSwitcher state machine called screens which helps the graphics component choose which screen to display in its video control ram component.

C. Graphics

The graphics component takes in data from the compute file for every registers contents as well as the current state of the screen being displayed. The graphics component also has inputs for clock, resetn, X_index position and 2 6-bit input sources called p1shades and p2shades. Below is a simplified model of the Graphics component containing only 1 IndexPosition instead of 7, and one color decoded component instead of 42. This is purely for visual purposes and is only meant to present the reader with a simplified model of the Graphic component's block diagram.

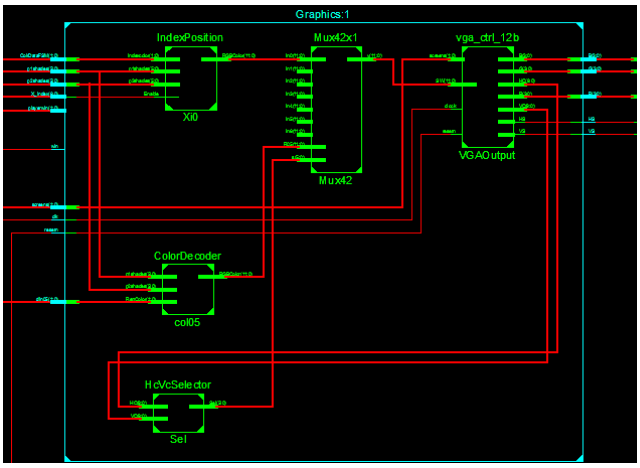

Figure 6. Graphics component simplified block diagram.

The first thing that happens with all of the data inside of the 42 registers is that it is decoded into 12-bit RGB. The current index position is also decoded into 12-bit RGB with and enable, which is the MSB that the index correlates to. This was done so that only the current index of the player would be decoded into color representing the current players unique color whilst the remaining six indexes remained black.

The best way to present the logic behind each of these components is to present the code used to decode this data. Note that "01" and "10" represent a key color for each of the player's data that has been input into each register in the Compute component.

```
process(RegColor)
begin
   case RegColor is
      when "00" => RGBColor <= x"FF0"; -- yellow
      when "10" => RGBColor <= shades1; -- red
      when "01" => RGBColor <= shades2; -- blue
      when others => RGBColor <= x"000";
   end case;

end process;
```

Figure 7. Register Color 12-bit RGB decoder.

```
process(Indexcolor, Enable)
begin
if Enable = '1' then
   case IndexColor is
      when "10" => RGBColor <= shades1; -- red
      when "01" => RGBColor <= shades2; -- blue
      when others => RGBColor <= x"000";
   end case;
elsif Enable = '0' then
   RGBColor<=x"000";
end if;
end process;
```

Figure 8. IndexPosition 12-bit RGB decoder

Note that the shades component is based off of the user's selection of the switches from the top file. They allow the user to change the color of their game piece from a standard red to shades of pink and orange. And from the standard blue into shades of teal and purple. Below is a descriptive picture of the RGB bit-manipulation done by each of the switch inputs.

Player one color shading

R	G	B
1 1 1 1	0 sw(15 downto 13)	0 sw(12 downto 10)

Player two color shading

R	G	B
0 sw(5 downto 0)	0 sw(2 downto 0)	1 1 1 1

Figure 9. p1shades and p2shades technique.

After each of these colors is decoded into 12-bit RGB they are fed into a multiplexor, which is effectively a 49 to 1 multiplexor. It views the 7 index locations and 42 register components and the selector decides when to choose each of these decoded color signals is appropriate to be displayed on the VGA screen.

The method through which this is done is in the HcVcSelector component. This component takes X and Y coordinate data position from the vga_ctrl_12b component as a reference of where the data from the screen is coming from in the standard 640x480 pixel grid. The process of choosing the selector line to the appropriate position was done rather easily by choosing the bounded regions on the screen to which these signals apply on a normal Connect 4 board. Below is an image appropriate describing the regions that were displayed onto the VGA by these select lines. However the only image displaying technique used was a 4 point bounded square which took HC and VC coordinates for each representative square of data. This was done because it was significantly easier to draw a square than it would have been to draw a circle; the HcVcSelector code would have been very cumbersome and would have to account for approximately sixty times as many regions of display.

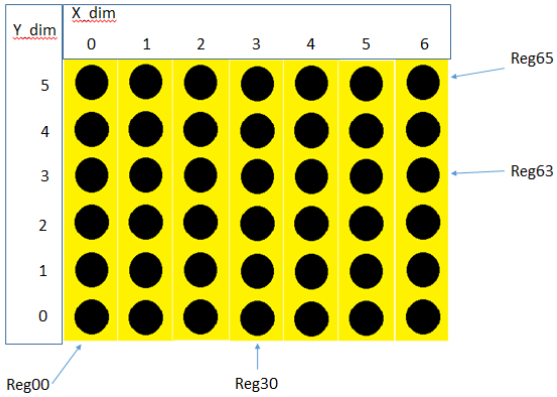

Figure 10. Connect 4 grid layout and register naming.

Up to this point of the game, everything surmounts to a 12-bit input signal into the vga_ctrl_12b component. Below is a comparison to the file,[2] which took a 12 bit input signal and image data from a txt file. The main differences are that the Connect 4's 12-bit input signal is based upon the entire Connect 4 game being selected in 49 regions, 3 Img Rom files, and a 4x1 multiplexor whose select line is based off of the Screens signal coming from the PlayerSwitcher state machine in the Compute component.

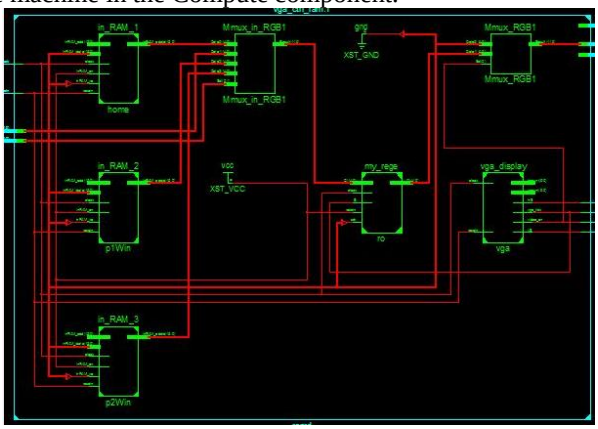

Figure 11. Connect 4 vga_ctrl_12b Top file.

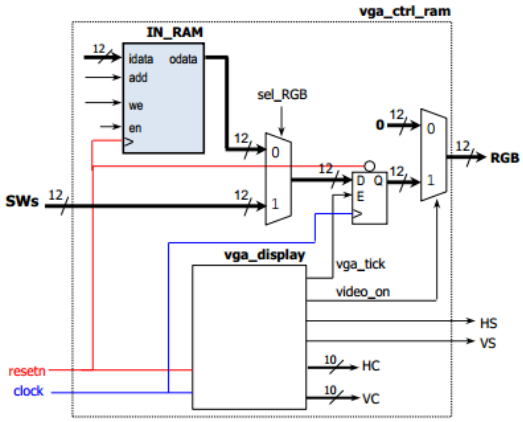

Figure 12. Standard Vga_controller [2]

III. Experimental Setup

A VHDL test bench on the Compute file was created in order to create an end to end test for a game that ended in 7 turns. This was done to determine whether or not the registers were storing data in the correct place, as well as determining if the Winner component was capable of determining a winner. The first move was made in the bottom left corner and the second player simply played its piece on top of player 1. The next move done by player one was 1 index to the right, this pattern was done 7 times until player1 had won. Below is a Xilinx timing simulation and a description of the register format used for data analyzing in the Compute file.

reg05, reg15, reg25, reg35, reg45, reg55, reg65,
reg04, reg14, reg24, reg34, reg44, reg54, reg64,
reg03, reg13, reg23, reg33, reg43, reg53, reg63,
reg02, reg12, reg22, reg32, reg42, reg52, reg62,
reg01, reg11, reg21, reg31, reg41, reg51, reg61,
reg00, reg10, reg20, reg30, reg40, reg50, reg60

Timing Simulation Depicting each of the 7 turns and their corresponding values of "red=10" and "blue=01" being saved into each register.

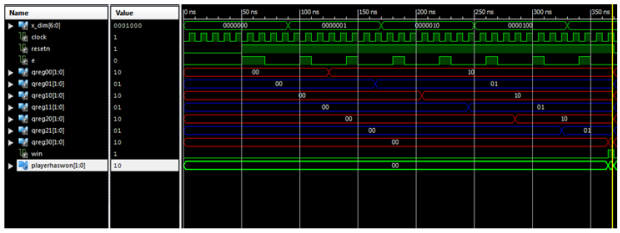

Figure 13. Compute test bench simulation for winner.

IV. Results

The project was then tested on the VGA in an end to end test where the exact same sequence of moves was performed. As shown, the game correctly shows the sequence of home screen- to gameplay- to win screen as well as that the data is indeed being placed into the correct registers and output accordingly.

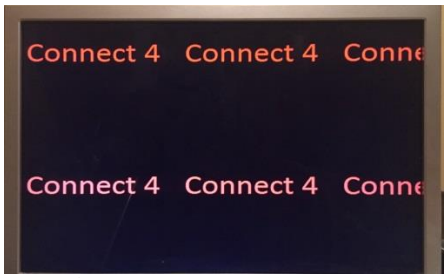

Figure 14. Home screen image

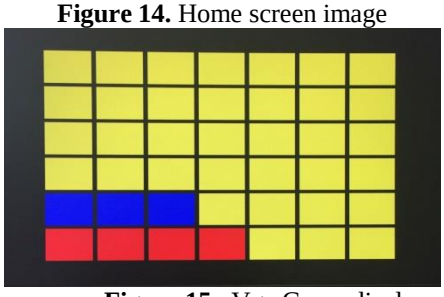

Figure 15. Vga Game display

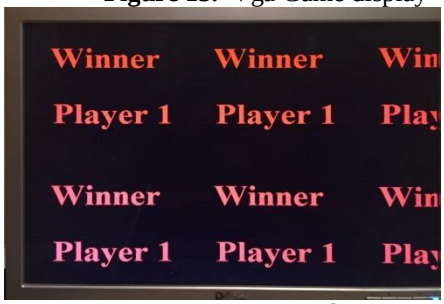

Figure 16. Winner player 1

Conclusions

The project was a great exercise in learning how to create large projects such as games and proved to be a humbling experience when thinking about computer engineers who made games in the 80's this way. The main goal of this project was to create a very simple game that everyone knows such as Connect 4 and interface it with the Nexys 4 onto a VGA monitor. The biggest milestone that arose was The creation of the data path circuitry and developing all of the logic components and state machines that are used to control the games Compute component.

Further work that needs to be resolved before this game is a top notch piece of art are as follows; The game must display circles onto the VGA display (this is easily done it is just very time consuming), The selection screens such as home, player1win and player2win need to be reformatted so that there is not duplicates of them and they more aesthetically display the winner, and the addition of P1 and P2 displayed onto the VGA monitor during gameplay in each players respective corner. Once these graphical changes have been accomplished this game should display what the compute file is capable of storing. It's also worth mentioning that the addition of a video game controller into the USB port could serve as a nicer way to play this game instead of using the push buttons on the Nexys 4.

Overall this project was a success as shown in figures 14, 15 and 16 the game works as intended. The project was an enormous learning experience and it taught the importance of planning and preparation when trying to solve a problem. The game was very fun to make and has been a very entertaining project to design.

References

[1] Daniel Llamocca VGA_Controller VHDL code.

[2] Connect 4 check for winner Algorithm "http://stackoverflow.com/questions/20201216/connect-4-check-for-winner-algorithm"