**Connect Four**

by

C2C John Terragnoli

A Technical Report Submitted to the Faculty of

Electrical and Computer Engineering

United States Air Force Academy

Submitted in partial fulfillment for the requirements of

ECE 383 – Embedded Systems Design II

May 8, 2015

1. **Chapter 1: Design Goals**
   1. **Need Statement**

The demand for simple electronic games has increased in the past decade. Business owners are eager to recreate board games in an electronic fashion so they can be used and stored by families easier and cheaper.

* 1. **Marketing Requirements**

It should be able to be played on the average monitor through an HDMI cable. Aside from the purchased kit, which includes the FPGA, a power cord, two remotes, and an HDMI cable, all that is needed is a wall outlet and some batteries for power. The game itself will look much like the traditional Connect Four game, except the grid will have eights rows and ten columns.

* 1. **Level-0 Description**

The Level-0 description, shown in Table 1, describes the overall inputs, outputs, and behavior of the Connect Four game.

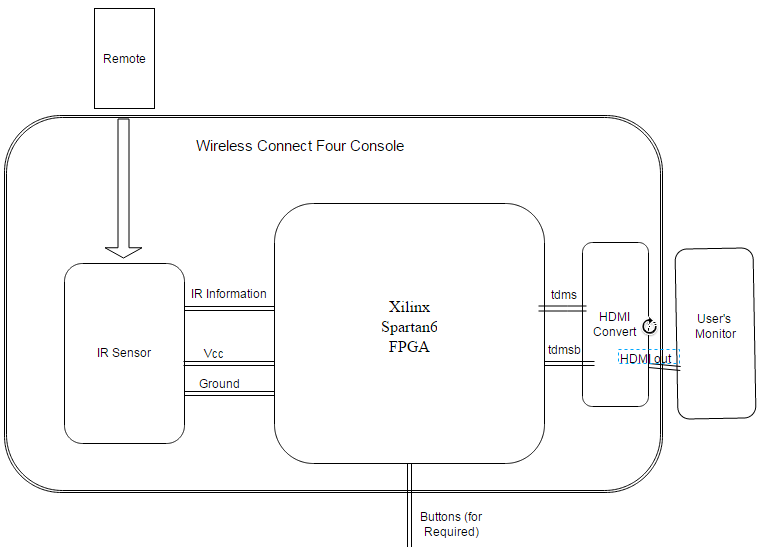
Table 1: Level-0 Description

|  |  |
| --- | --- |
| Inputs | Remote Infrared (IR) signal, directional and select buttons are required; buttons on FPGA |
| Outputs | An HDMI signal is output displaying the game. |
| Behavior | Should decode the IR signals, determine if it is a viable move or not, determine which space should be lit up, determine if the player has won the game or not, and end the game if a player has won.  Required functionality is to get the program to work using the buttons on the FPGA. Advanced Functionality was to use the remote to control the game through an IR signal. |

1. **Chapter 2: Detailed Design**

Figure 1 describes the Level-1 Architecture, which describes how the different hardware comments will be connected.

* 1. **Figure 1: Level-1 Architecture**



Below are the behavioral function tables are shown for the major components of the Level-1 Architecture. Tables 2, 3, 4, 5, and 6 relate to the remote, IR sensor, Xilinx Board, HDMI Convert, and Monitor, respectively, and describe their appropriate function behavioral tables.

* + 1. **Table 2: Remote:**

|  |  |
| --- | --- |
| **Input** | **Output** |
| User presses right directional button | IR bit-packet for right directional button press |
| User presses center directional button | IR bit-packet for column selection button press |
| User presses left directional button | IR bit-packet for left directional button press |
| User presses up directional button | IR bit-packet for up directional button press |

* + 1. **Table 3: IR Sensor**

|  |  |
| --- | --- |
| **Input** | **Output** |
| IR bit-packet for right directional button press | Right button press |
| IR bit-packet for column selection button press | Center button press |
| IR bit-packet for left directional button press | Left button press |
| IR bit-packet for up directional button press | Enable button press |
| Vcc | Nothing |
| Ground | Nothing |

* + 1. **Table 4: Xilinx Board (Spartan 6)**

|  |  |
| --- | --- |
| **Input** | **Output** |
| Right button press | Marker above potential column shifts right one |
| Center button press | Current player color written to BRAM based on the marker and how full the column is |
| Left button press | Marker above potential column shifts left one |
| Enable Button press | Enables the player to move horizontally or drop a piece |

Note: all outputs for the Xilinx board are based on what would appear on the screen due to the signal made by the “tdms” and “tdmsb” signals.

* + 1. **Table 5: HDMI Convert**

|  |  |
| --- | --- |
| **Input** | **Output** |
| Tdms | Combined with “tdmsb” to make HDMI signal |
| Tdmsb | Combined with “tdms” to make HDMI signal |

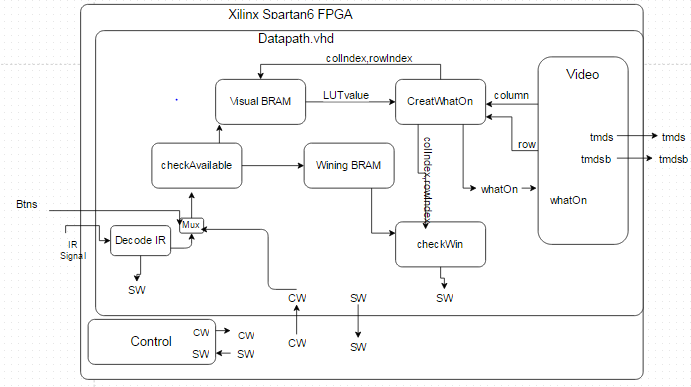
* + 1. **Table 6: Monitor**

|  |  |
| --- | --- |
| **Input** | **Output** |
| HDMI signal | Image on the monitor |

* 1. **Datapath**

The complete datapath can be seen below in Figure 2.

Figure 2: Datapath

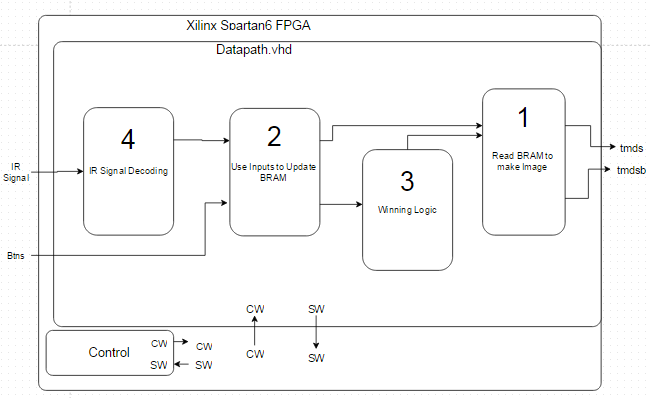


The bit-lengths of the signals of the datapath can be seen below:

* colIndex = 4 bits unsigned
* rowIndex = 4 bits unsigned
* LUTvalue = 2 bits unsigned
* whatOn = 2 bits std\_logic\_vector
* row = 10 bits unsigned
* column = 10 bits unsigned.
* Cw = 7 bits std\_logic\_vector
* Sw = 3 bits std\_logic\_vector
* Tdms = 4 bits std\_logic\_vector
* Tdmsb = 4 bits std\_logic\_vector
* IR Signal = 1 bit std\_logic
* Btns = 5 bits std\_logic\_vector
  + 1. **Simplified Datapath**

While the first datapath shown might include all vital signals, a simplified a behavioral version is shown in Figure 3 on the next page to give the reader a better understanding of how the project was approached.

Figure 3: Simplified Datapath

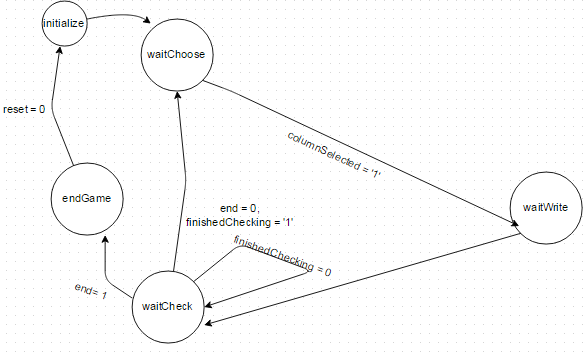


Obviously, there are four main parts to this program: reading BRAM to make an image; using inputs to update BRAM; winning logic; and IR decoding. The four boxes correspond to Milestone 1, Milestone 2, Basic Functionality, and Advanced Functionality, in order from one to four, as labeled in the schematic. Using BRAM to draw an image (1) allows the program to keep track of where each players’ pieces are on the board, so they can be draw each time the screen is refreshed. The use inputs to update BRAM box (2) describes the ability for a user to choose a column and for the final location of the piece dropped to be recorded. The winning logic block (3) reads where each player’s pieces are in BRAM and decides if a win has occurred or not. The IR signal decoding block (4) receives a bit-packet from the IR sensor, decodes this packet to either be a left, right, center, or up button press, and sends the results to the second box above to move the marker and drop pieces. A more detailed description of the development process and functionality of each box can be seen in Chapter 3.

* + 1. **Control Unit**

The master finite state machine for the control unit of the Connect Four game is shown below in Figure 4. Everything else in the program is subordinate to this finite state machine.

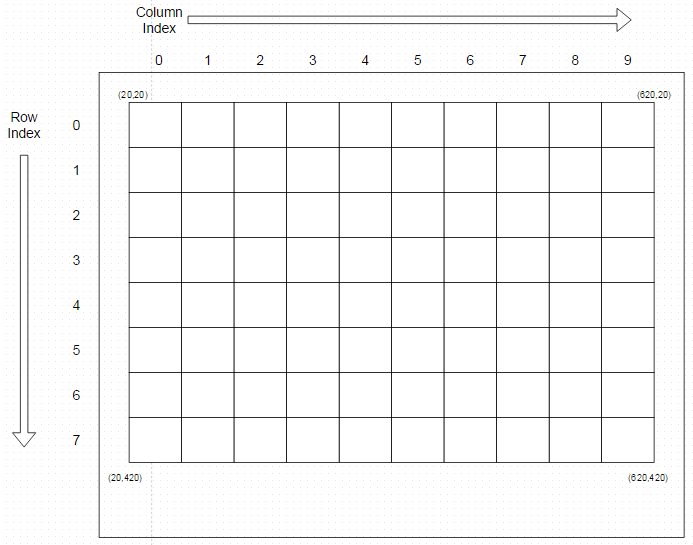
Figure 4: Main Control Unit Finite State Machine



Note: the signals columnSelected, finishedChecking, and win are the status words (sw) 1, 2, and 3, respectively. Also, later in the design process, two other finite state machines (FSMs) were created in the winning logic and IR signaled decoding blocks. Both of these FSMs are subordinate to the master FSM shown above.

* 1. **Calculations**

The pixel locations of the grid marks on the monitor were calculated so that the board could be drawn and that boundaries could be made to describe the indexes for the game. These measurements and indexes are shown in Figure 5 on the next page.

Figure 5: Monitor Map and Index Characterization

Note: the monitor was 640 pixels wide by 480 high, row indexes were 50 pixels apart, and column indexes were 60 pixels apart. The row and column size were just calculated so that the board would be 8 by 10. The location of the grid lines were later used to classify which indexes each pixel belonged to. The formula for the row index based on pixel row is in Equation 1.

(1)

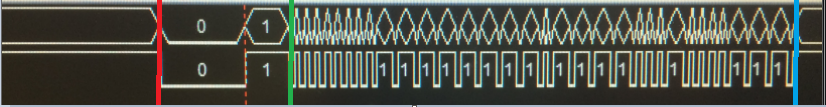
Equation 2, below, shows the column index based on the pixel column is below.

(2)

Note: column and row are in pixels; rowIndex and colIndex are both integers. Pairs of rows and columns which do not fit on the board are given the value (15,15) for their (rowIndex, colIndex) coordinates.

As for Advanced Functionality, the IR signals average lengths and boundaries for each part of the signal were calculated so that the bit-packet could be interpreted properly later. A typical IR signal can be seen in Figure 6.

Figure 6: A Generic IR Signal



Everything before the red line is in the period “indefinitely high.” Between the red and the green is the start bit. Between the green and the blue lines is the packet of data, which is 32 bits long for the remotes used in this project. In this section, shorter high periods correspond to zeros and longer high periods correspond to ones. Therefore, the identity of the high of the start-bit, short-bit, and long-bit are all characterized by their length. Then can then be identifies by timing how long the IR signal is high at a time. The average and standard deviation was found for each of these three parts in many trials.

The way these measured and calculated values were used will be more accurately described in Chapter 3, under the implementation of Advanced Functionality. The description is more appropriate there and will make more sense to the reader within the context of the attaining Advanced Functionality. The tables of the measured and calculated values are also located in the aforementioned section.

* 1. **Technical Requirements**

For this design to be considered a success, a number of different requirements needed to be achieved. Due to the nature of the project, though, these technical requirements were qualitative rather than quantitative (Dr. York approved). Lists of these requirements can be seen below.

* + 1. **Milestone 1**
* Draw a grid on the screen with eight rows and ten columns
* Read values stored in BRAM and display them on the screen
  + 1. **Milestone 2**
* All of Milestone 1 requirements
* Have the cursor on top of the grid move left and right according to the buttons on the FPGA or with the remote
* Have the cursor initialize on top of the grid
* Have the appropriate piece fit into the appropriate spot when the column is chosen by the buttons or by the remote
* Save the pieces in that slot
* Be able to stack pieces on top of each other
* Reject illegal moves such as stacking more than can fit in a column or placing a piece in a place above an empty space
* Switch turns when a player places a token
  + 1. **Basic Functionality**
* All of Milestone 2 requirements
* Evaluate whether a player has won the game or not, or if a tie has occurred, and end the game appropriately. A win is having four of the same color in a row either diagonally, horizontally, or vertically
* Should follow all other conventional Connect Four rules, if any were missing from this description
* Play a full game flawlessly using the buttons on the FPGA board
  + 1. **Advanced Functionality**
* All of Basic Functionality
* Use an IR remote to play the game instead of the buttons on the FPGA
  1. **Bill of Materials**

The following materials will be required to complete the electronic Connect Four game:

* Two infrared (IR) remotes, similar to a typical TV remote.
* A simple three pin IR sensor
* A small breadboard to connect the sensor to the FPGA.
* FPGA, including the power cord, the USB cord to connect to a computer for programming, and an HDMI cable which can be connected to a monitor.
* Small collection of wires to connect the IR sensor to the FPGA.

1. **Implementation**

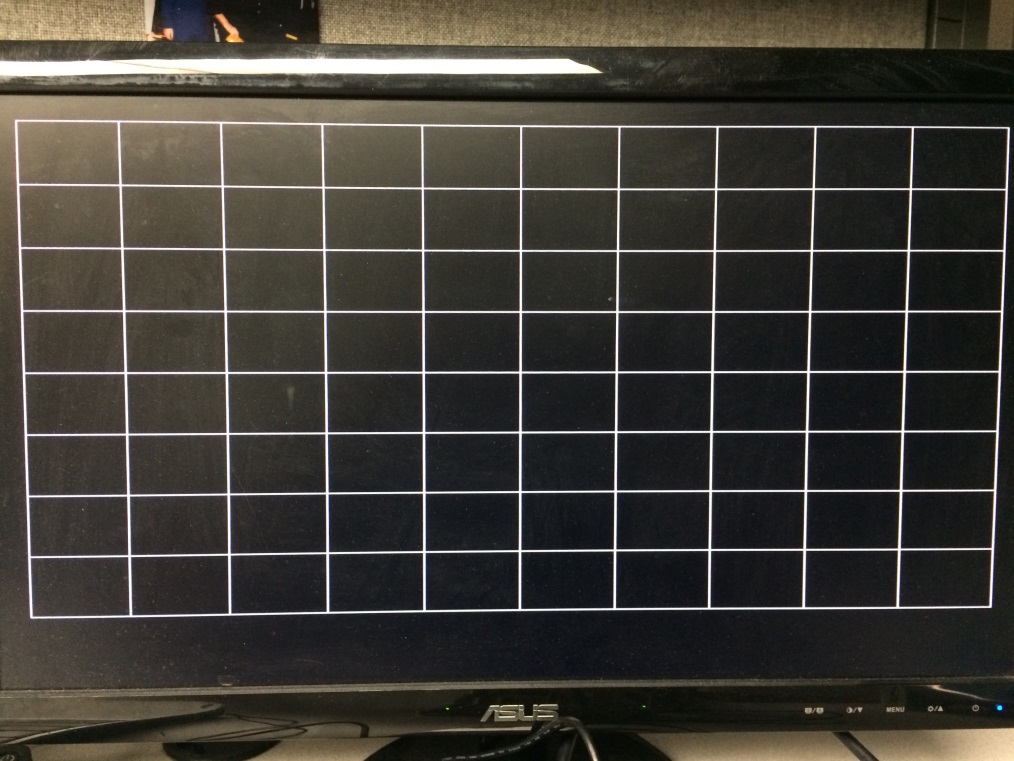
This chapter will elaborate on the specific steps performed to achieve the different levels of functionality. It includes calculations, schematics, testbenches, test results, and anything else involved in the design process.

* 1. **Milestone 1**

The overall purpose of Milestone 1 was first to develop an index system which could be synergized with BRAM to store the color of each square on the board. The second half of the functionality was utilizing this system to be able to read any pattern of red, blue, and empty spaces from BRAM and display them on the grid.

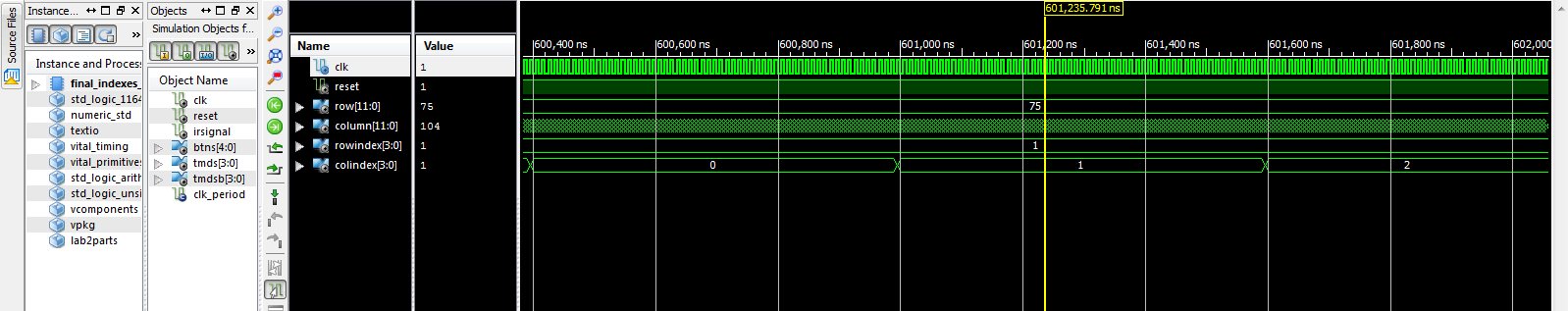
The modules video.vhdl, vga.vhd, counter\_glue.vhd, counter.vhd, h\_synch.vhd, v\_synch.vhd, scopeFace.vhd, dvid.vhdl, and tdms.vhdl were used from Lab02 when making an oscilloscope. Some of the irrelevant signals, such as the ch1 and ch2 values for incoming waveforms, were removed for simplicity. After this was done, an instance of video.vhd was created in the datapath module and a bit file was generated to ensure the old oscilloscope grid appeared on the screen. This test did work, and the grid originally produced, which was used throughout the duration of the project, can be seen in Figure 7.

Figure 7: Blank Board Screen



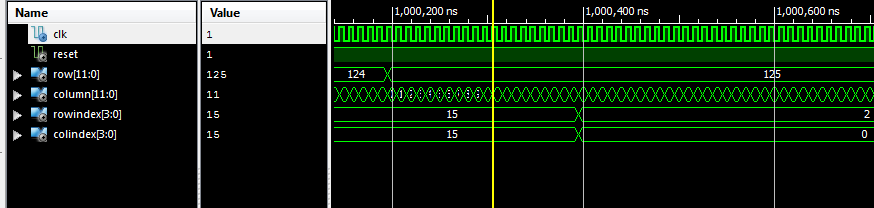
After this was done, the logic to classify each pixel into its respective indexes was created. This was done in the datapath and in accordance with Equations 1 and 2. A testbench showing the results of stepping through each pixel and referencing its appropriate indexes is in Figure 8 on the next page.

Figure 8: Giving Indexes to Pixels



Unfortunately this picture is difficult to see, but when the row was 75 and the column was 104, then the row/column index was (1,1). Also, when a pixel was referenced which was not on the game board face, an index of (15,15) was referenced. These numbers were chosen as they cannot relate to a specific square on the board and thus, if BRAM references them, they will always be returned as empty. This becames important later as the winning logic was developed. A testbench showing this phenomenon is shown in Figure 9.

Figure 9: Pixel Outside of Board Referenced



Then, the modules from Lab02 were slightly altered to accept a signal called “whatOn,” which was the value currently being referenced in BRAM according to the appropriate indexes. The value of the whatOn signal was sent to scopeFace.vhd, which ultimately maps which pixels will be what color. To test if the whatOn signal and the index system was written correctly, Code 1, shown below, was written in the datapath.

Code 1: Testing Index System’s Ability to Write

whatOn <=

"0001" when ((rowIndex = "0001") and (colIndex = "0001")) else

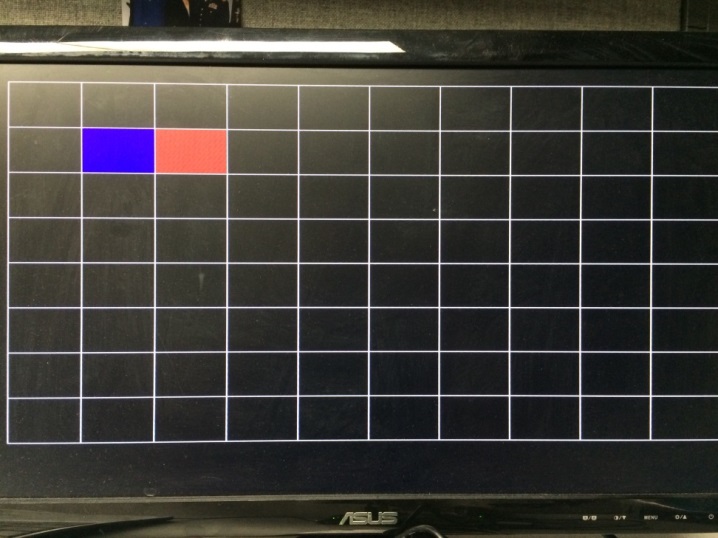
"0010" when ((rowIndex = "0001") and (colIndex = "0010")) else

"0000";

Combinational logic was chosen at first, instead of just using BRAM, because it is less complicated and less prone to user error in implementation. BRAM was used later once the ability to set the specific color of a space was verified.

The sequence 01 and 10 refers to blue and red, respectively. These lines just say that when the index coordinates are (1,1), then blue pixels should be drawn, and when the index is (1,2), red pixels should be drawn. Thus, space (1,1) is all blue and (1,2) is all red. The result from generating and running a bit stream with this code is shown in Figure 10.

Figure 10: Writing to Screen with Combinational Logic

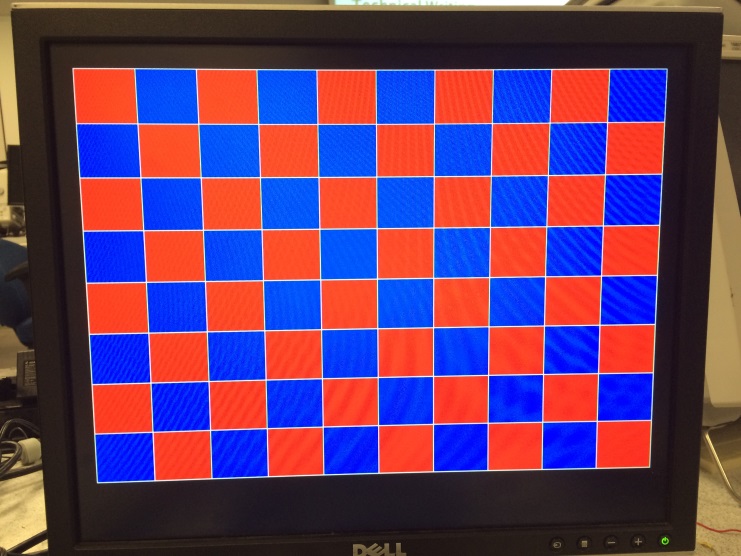


Clearly the blue and red squares are in the predicted locations. This meant that the whatOn signal worked correctly. Moving on, BRAM was made to replace the combinational logic. Then rowIndex and colindex were concatenated to form the read address of the BRAM. The equation for this is shown below in Equation 3.

Read address = “00” & rowIndex & colIndex (3)

Thus, every possible combination of the row index and column index could be related and saved to distinct locations in BRAM. To test this theory out, ones and twos were initialized in BRAM, alternating in a checkered pattern. The program was then able to read this pattern and print the checkered pattern to the screen. This screenshot can be seen in Figure 11.

Figure 11: Checkered Pattern on Monitor



After a couple more similar tests were made, it became quite apparent that any pattern of zeros, ones, and twos could be read from BRAM and displayed on the screen with the appropriate color. Thus, the functionality for Milestone 1 was obtained.

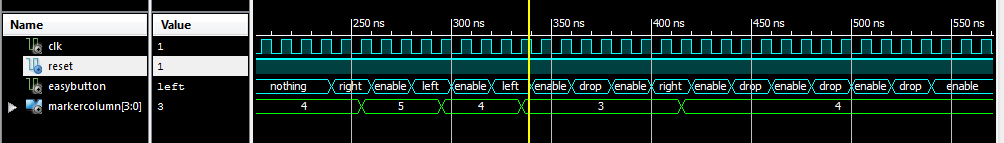
**3.2 Milestone 2**

The main purpose of Milestone 2 was to be able to use the buttons on the FPGA to write either a one or a two into BRAM with accordance to the rules of Connect Four. This capability, in combination with Milestone 1 functionality, allowed the user to use the buttons on the FPGA to place a piece and watch it appear on the screen, so long as that move abided by the rules of Connect Four.

Achieving this functionality involved breaking it into smaller steps. First, a marker needed to be created for the user to see which column is currently being considered. The marker is a yellow box which hovers above the column and represents the column the piece will drop in if the select button is hit. Second, the program needed to keep track of how many pieces were already in each column so that it knows which row to write to depending on which column is chosen.

The marker was created using the column indexes and a small lookup table. The marker is only allowed to be over one of the columns, never in between. Hitting the button left or right should decrease or increase the column index it is currently hovering over. Before this was implemented, a testbench was created to verify that the index the marker was at was being updated appropriately with button presses. This is shown in Figure 12:

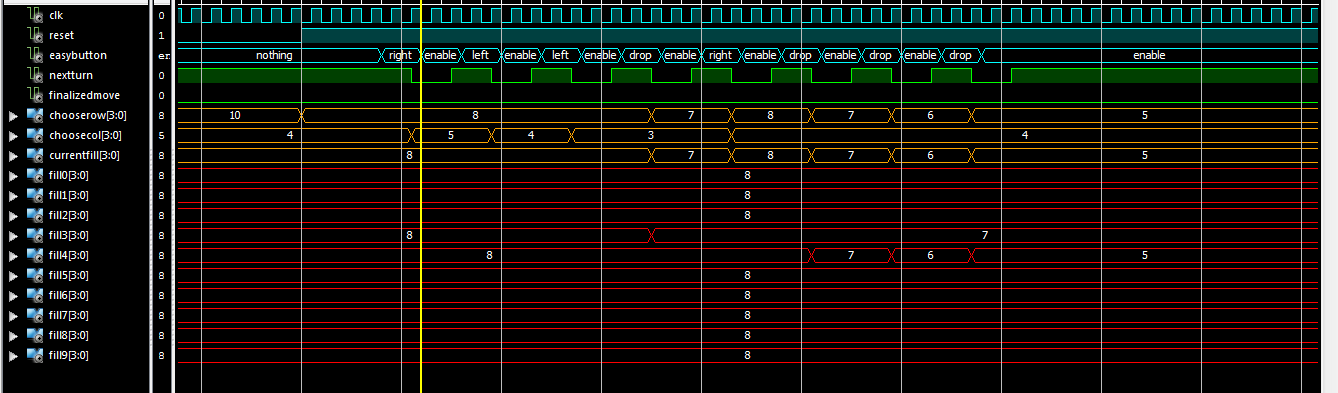
Figure 12: Marker Indexes Change Appropriately



The index of the marker changes as appropriate. This index was then passed down into the scopeFace module, where it was converted into the column value of the left grid line at that column index. For example, if the marker was at index 0, it would translate to column 20 out of 640 in the scopeFace module. Using relative addressing, any pixel with a column between 20 and 60 greater than 20 (80) and between row 10 and 20 would be colored yellow. Thus, only the index of the marker had to be updated in order for all the boundaries of the marker to be shifted and for the marker to be shifted to the next column.

To ensure the program wrote to the proper row, 10 fill signals, one for each column, were created. Their purpose was to keep track of the bottom-most space available in their respective column. They were initialized at the very bottom, at seven, and every time their column was selected, their value would decrease by one. A testbench proving this works is shown on the next page in Figure 13.

Figure 13: Fill Signals Updating After Drop



Thus, to find the row to write to, the program would check the fill signal associated with the column currently being considered, and have the writing rowIndex be equal to the value of the fill signal. This could then be combined with the column index currently being considered by the marker to create a write address for BRAM. The write address is shown below in Code 2

Code 2: Definition of Second BRAM Write Address

Write address = “00” & fillValue & markerIndex

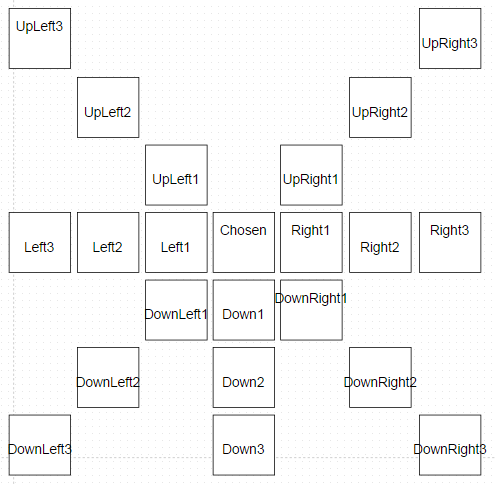
Then, when the master FSM in the control unit is in the waitWrite state, the write is enabled for BRAM. The value which gets put in BRAM is the same as the value for whose turn it is. The blue player is 01 and the red player is 10, which directly corresponds to the values representing blue and red when BRAM is being read and put on the screen.

Once this was achieved, it was impossible for the user to place a piece on the board and break a rule of Connect Four.

**3.3 Basic Functionality**

The only additional requirement for Basic Functionality was the ability to check and see if a player has won the game. This was the most difficult task up until this point. The easiest way to see if there is a winner or not in VHDL is to check every possible combination. Fortunately, all winning moves must involve the last piece placed. Thus, there are only thirteen possible combinations. To check these combinations, the values stored at 22 different locations need to be known. The spots which need to be checked are shown in Figure 14 on the next page.

Figure 14: Relative Addressing for Winning Logic



Once the values stored at each of these spots are saved in signals, combinational logic was applied to test for a win relatively easily. However, getting 22 values out of BRAM at the same time is impossible, so 22 separate trips to BRAM needed to be made to attain these values. Additionally, the original BRAM cannot be used, as the video module needs to constantly access the original BRAM to know which colors to print onto the screen. Thus, an exact replica of the first BRAM was created for the sole purpose of checking for winning combinations. The second BRAM had all of the same write signals as the first so they were identical.

To extract all 22 values out of BRAM a FSM was created within the winner.vhd module. Each state was named after the relative address it represented, such as state\_down2. As the program moved from state to state, the relative address being referenced by the BRAM was changing. Thus in each iteration through the process the address would change once and BRAM would spend one clock cycle outputting the data at the relative address.

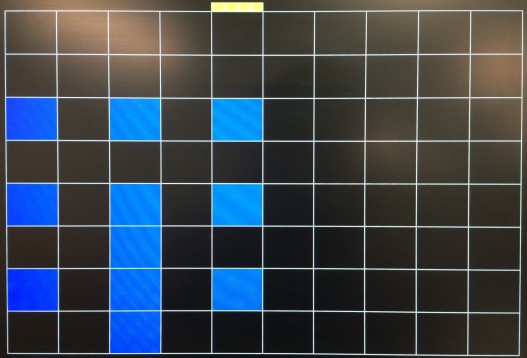
When the user selects a column, the winner logic assigns the row and column index of the piece placed to two signals called the anchorRow and anchorColumn. The idea was that the anchorRow/Column values would not change while a win was being checked, but a second set of values, ripRow and ripCol would be changed relative to the anchor signals. For example, if the anchor coordinates were (5,5) and the program wanted to read to the down1 signal, ripRow becomes anchorRow+1 for that state. The read address was written outside the process as the concatenation shown in Code 3.

Code 3: Definition of Second BRAM Read Address

RDADDR <= “00”&ripRow&ripCol.

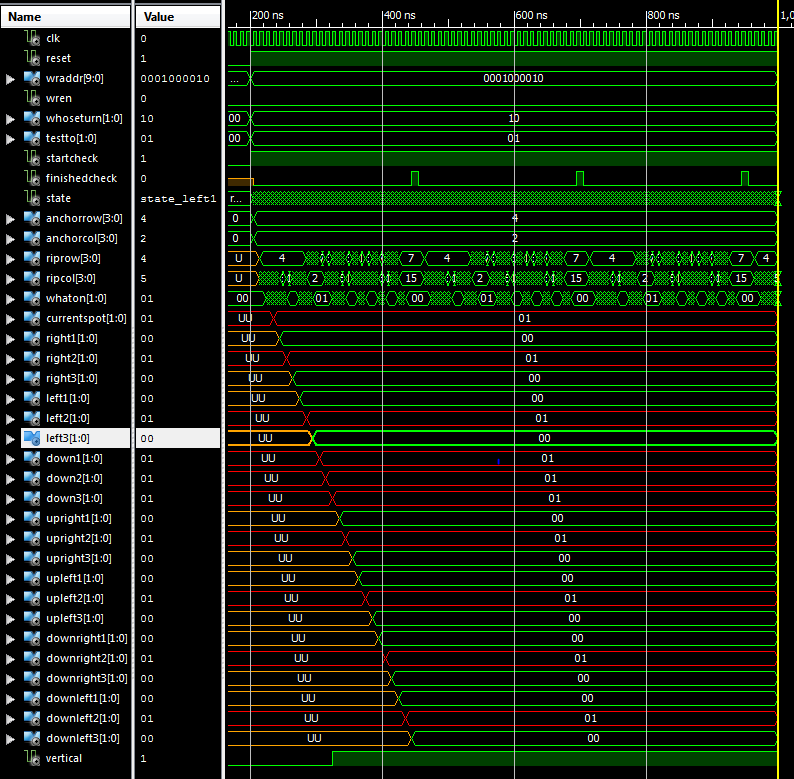
After climbing through 22 states and changing ripRow/Col 22 times, the values at each of the BRAM locations had been extracted and stored in the appropriate signals for combinational logic to take over. However, there was a very large issue with this process at first. At first, during tests it seemed like some of the 22 signals were being updated but the wrong ones were being changed. Then a pattern was noticed that instead of an address being read in its respective state, it would occur exactly two states later. Thus, if the states went from down1 to down2 to down3, then the value read at the location down1 would be placed into the signal down3. To fix this, all signals were simply read two states later, rather than in their respective state, with two dummy states on the end. Once this was done, all of the relative addresses could be access well and winning combinations were easy to spot. Verification of this process can be seen in the following test. BRAM was loaded with values which would produce the following image on the screen, shown in Figure 15.

Figure 15: Sample Pattern for BRAM Extraction Test



A testbench was created with the same values in the logic BRAM. When the anchor was located at (4,2) values residing in the surrounding boxes were read from the logic BRAM. The value for blue, 01, was read from the appropriate relative addresses and saved in the correct signals (currentSpot, down1, down2, down3, left2, right2, upRight2, upLeft2, downLeft2, downRight2) as seen in Figure 16.

Figure 16: Correct Values Extracted from BRAM



After this was achieved, a .bit file was generated and many games of Connect Four were played. The games finished when winning combinations were played every time. A video of this basic functionality can be seen [here](https://www.youtube.com/watch?v=hEnDyH8UMFo&feature=youtu.be).

**3.4 Advanced Functionality**

The main purpose of the Advanced Functionality was to replace the buttons on the FPGA with an IR remote to control the game. This was done in two parts. First, data had to be collected regarding the length of the start bit, the zero high, and the one high. This was essential because the length of the high portion of the signal is what classifies what value is currently being read from the signal. The second part was building a module which decoded the incoming IR signal and determining which button was hit.

**3.4.1 Data Collection**

The logic analyzer and IR sensor was used to view and measure the IR signal coming from the remote. Measurements were taken for the length of the start bit high, the zero high, and the one high. About 20 trials were done for each. The averages and standard deviations of this data can be seen below in Table 7.

Table 7: Duration of Vital Portions of IR Signal

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Lengths in Time (ms)** | | |
|  | **Zero High** | **One High** | **Start Bit High** |
| **Average** | 0.54403 | 1.68663 | 4.53703 |
| **Standard Deviation** | 0.00228 | 0.00939 | 0.01194 |

As mentioned earlier, the high parts of the signal are recognized by how long they are. Since there is not an actual timer available in VHDL, a counter was used to act as one. By counting the number of clock cycles which take place while the signal is high, it is possible to know the approximate time length of the signal high. The clock period is 10 ns. Therefore, the times in the table above can be converted into nanosecond by multiplying by 1000000. Then the lengths can be converted into clock cycles by dividing by10.

The standard deviation of each type of high was found in order to create a range for checking for that specific signal. Each IR output the remote creates will not be perfect or identical. Therefore, any high length which is within five standard deviations of the average, which accounts for 99.9999426697% of that type of high output, will still register as that bit [1]. In other words, if the high duration is very close to an expect value, it is alright assume that expected value was intended. The low and high boundaries in terms of clock cycles can be seen below in Table 8 on the next page. The upper bound is five standard deviations above the average and the lower bound is five standard deviations below the average.

Table 8: Clock Cycle Boundaries for Specific IR Bits

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Clock Cycles** | | |
|  | **Zero High** | **One High** | **Start Bit High** |
| **Upper Bound** | 55441 | 174475 | 459370 |
| **Lower Bound** | 53158 | 165084 | 447429 |

The above data makes it possible to convert an incoming IR signal into a string of zeros and ones. However, without anything to compare these strings to, it is impossible to know which button was pressed. Therefore, the logic analyzer was used again to view the resulting signal from left, center, right, and up button presses on the remote. Each of the signals looked similar to the general IR signal shown in Figure 6 in the calculations section. The signals were enlarged and visually decoded by recording the bits in the data packet from left to right. The results can be seen Table 9.

Table 9: Bit Strings Representing Information in Necessary IR Signals

|  |  |  |
| --- | --- | --- |
| **Button Press** | **Binary** | **Hexadecimal** |
| Left | 00000000111111110101000010101111 | 00FF50AF |
| Center | 00000000111111110000001011111101 | 00FF02FD |
| Right | 00000000111111110111100010000111 | 00FF7887 |
| Up | 00000000111111111010000001011111 | 00FFA05F |

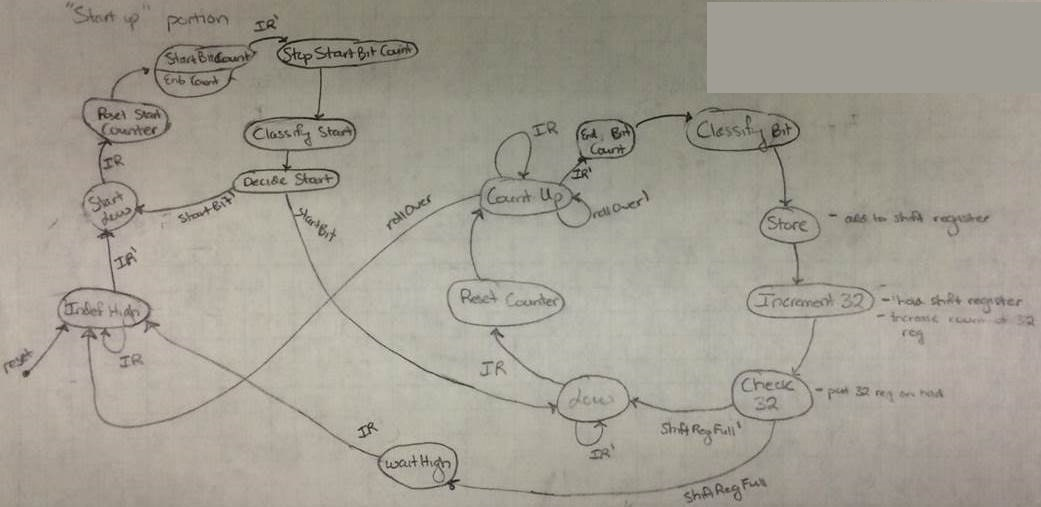
Note: The hexadecimal value is included in the table for convenience, although in the actual program the received bit string was just compared to the binary values in the second column.

Once these values were known, assuming a module could be written to decode the signal to a 32 bit string, the program could compare these stored values to the decoded string and determine which button was pressed.

**3.4.2 Decoding Incoming IR Signal**

In order to decode the incoming signal an additional FSM was created. There were two main parts of the FSM. The first half of the FSM checks for the start bit, so it can know when the signal begins and when to start decoding the bit-packet. The second half of the FSM actually decodes the bit-packet. A copy of the FSM can be seen in Figure 17.

Figure 17: FSM for Sensing and Decoding an IR Signal



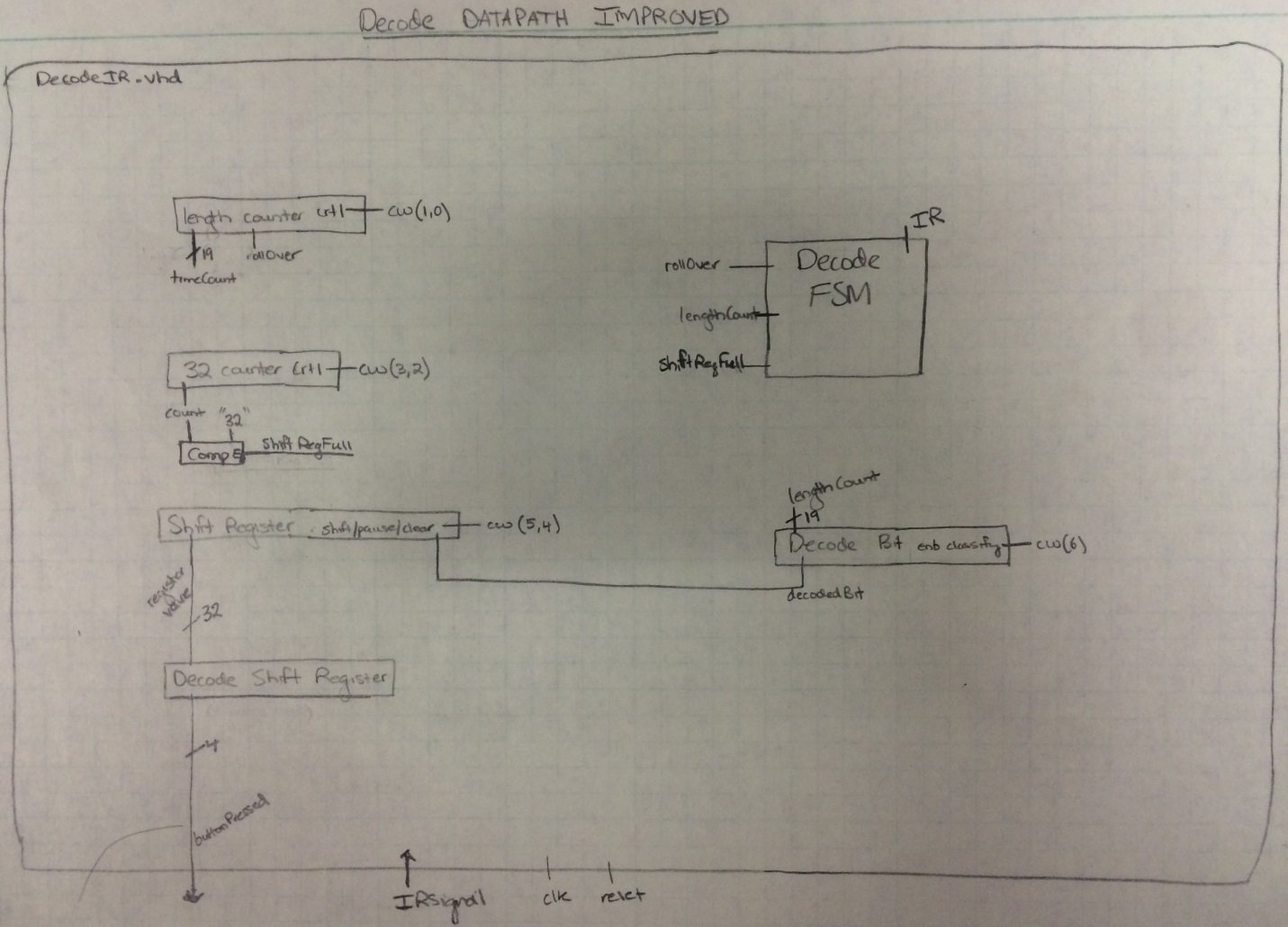
The basic gist of this FSM is that it will operate in the left half until the start bit is received. As soon as the start bit is confirmed, the machine will immediately begin to decode the 32 information bits. The decoded bits will be added to a shift register, as seen in Code 4.

Code 4: Shift Register to Collect the Decoded Bits

decodedSignal <= decodedSignal(30 downto 0) & decodedBit(0);

A second counter was instantiated to count every time a bit was added to the shift register. Once the counter got to 32, then the decoded signal was saved and combinational logic was used to determine if a button press occurred and which button was hit. A decoder-datapath can be seen on the next pag relating all of the processes and combinational logic inside the module. It is in Figure 18.

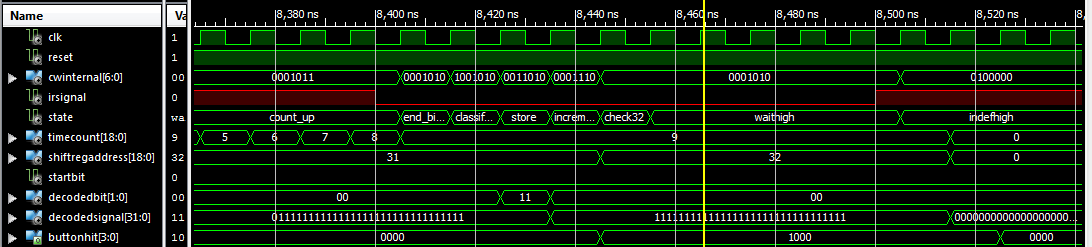
Figure 18: Decoding IR Datapath



Also, another noticeable aspect of the design was choosing a bit length of 19 for the counter size. This value was chosen as it was the smallest size capable of writing the number 460,000. If a rollover occurs, then the program returns to the beginning state. The only reason it would be rolling over was if it was fixated on counting something it was not supposed to. In this case, it is better to be prepared for just waiting for the next button press. However, if the size of the counter is too large, it might take too much time for the signal to roll over. If the delay is too large this might confuse the user and cause them to think that the remote is not working.

**3.4.3 Testbench for Decoding IR**

To prove that this program can actually work, a testbench was made with a fake IR signal sent in. The results of the testbench can be seen on the next page in Figure 19.

Figure 19: Testbench Decoding Fake IR Signal

This represents the very end of the decoding process when the shift register fills up. The bit string X“FFFFFFFF” was temporarily to be equal to be an up button press, or a “1000” in the program. This was done for the sake of making the fake IR signal less complicated.

This decoding module was instantiated in datapath. It accepted the IR signal and responded with what button was pressed. Then, inside datapath, the output from the IR decode module was sent to the variable “btns,” which represents button presses from the FPGA board buttons. Thus, the IR button press could directly replace the button logic which was already in place and known to be working.

**3.4.4 Implementation Advanced Functionality**

After this testbench worked correctly and instantiating the module, a .bit file was generated and sent to the FPGA. The IR sensor output was wired to JB(0) on the FPGA as an input, as seen in the Level-1 Architecture picture on the first page. However, when a button press occurred on the remote, nothing occurred on the screen at first. After pressing the buttons several times in a row, up to 20 repetitions, something would respond on the screen. Either a piece or two would drop or the marker would move. Additionally, a signal called “somethingDecoded” was created, which just detected if a decoded signal registered as a button press in general. When this signal was high, a green bar would light up on the bottom of the screen. After implementing this .bit file, the green bar did flash up occasionally, however, the program did not read it as a button press. A video of this green flash occurring can be seen [here](https://www.youtube.com/watch?v=knkCDsfrfm0&feature=youtu.be). Thus, Advanced Functionality was partially attained, as the signal was decoding the button press properly, although it was not nearly reliable enough to play a competitive game.

Another tactic used to understand the inside of the program was to decode the states of the IR decode module as a four bit string and output them to the JB ports. These ports were then read by the logic analyzer along with the actual IR signal itself. After running this test, the IR signal did show up on the logic analyzer, and the states changed appropriately, although not all of them could be seen as the logic analyzer does not sample quickly enough. It seemed like the IR signal was being decoded properly, but maybe that it was just happening too quick for the datapath to respond. To combat this, the decode signal was not reset until the startBitLow state. This should have given the program plenty of time to read the signal and return the appropriate button to the datapath to move something on the screen. However, this did not solve the issue.

After talking with Dr. Coulston, the answer to reliably decoding the IR did not get answered. However, this project will not be closed until a solution is found at Advanced Functionality can be achieved.

1. **References**

[1] "Standard Deviation." Wikipedia. Accessed May 8, 2015.

1. **Appendix A: Running the Project**

To duplicate the demonstration for Basic Functionality, just upload the final.bit file to the FPGA and connect the HDMI output on the FPGA to a standard monitor. The left and right buttons on the FPGA will be use to move the marker back and forth, and the center button is used to drop a piece in that column. In between every button press the top button needs to be pressed. This was done to eradicate the bouncing issues.

As for Advanced Functionality, upload the final.bit to the FPGA and the HDMI output to a monitor just as before. Then, use a breadboard to connect the IR sensor output to the JB(0) pin, the top right pin. The IR sensor needs 3.3v to be powered. Using the same remote I did for the presentation, the program should work upon loading the .bit file.

Notice that if the .bit file is giving the wrong functionality, the button input in the Datapath can replaced with the decoded IR signal or vice versa to get the right functionality. The only difference between Basic Functionality and Advanced Functionality is whether the IR signal or the FPGA buttons is driving the button logic in the datapath.

1. **Appendix B: Project Git Lab**

The source code, demonstration videos, the presentation, and a final copy of this report can be seen on at this [address](https://github.com/JohnTerragnoli/ECE383_FinalProject_Turnin).