Interactive Checkers

Michael Rodriguez

Ahmet Can Musabeyoglu

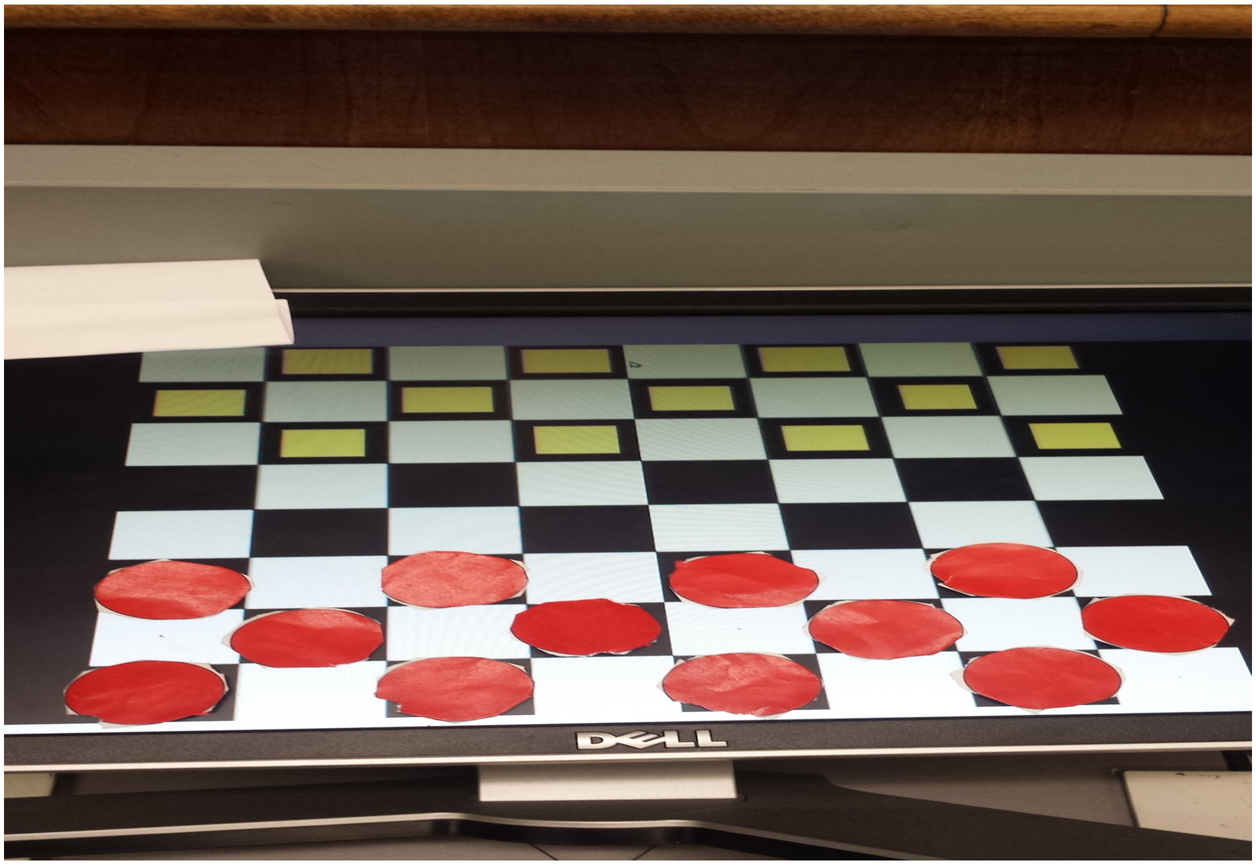

Figure 1: A picture of the Interactive Checkers game, which shows the starting round game board with actual human pieces (red ones) and computer pieces (yellow ones)

Overview-

This project seeks to create a new interactive way of playing checkers by combining FPGA driven image processing and an Artificially Intelligent checkers player. This hybrid game is a combination of a board game and a computer game. The game is set up by placing a flat computer screen horizontally on the table. An 8x8 checkers board will be displayed on this screen. The human player will place their 12 real red pieces on the checkers board and the computer's 12 yellow pieces will be displayed on the screen as seen in the Figure 1 above. When it is human player's turn to play, the player will move one of their own real pieces to the desired location and will press a "ready" button to inform the AI player that human's move is completed.

After human's move is completed, a webcam located above the screen will get an image of the screen and analyze what move has been made. When one of computer's pieces is removed from the game, it will vanish from the screen. When it is computer's turn to play, our computer AI player will calculate an optimal movement and will execute it on the screen. When one of human's pieces is removed from the game, the square that piece is located in will start blinking colorfully signaling the human player to remove their own piece from the game board. Combining all this functionality will create a completely new interactive way to play Checkers.

Design Description-

This checkers game is split into five main parts: the Image Capture Module with a camera that captures the current state of the board and sends this information to the FPGA, the Image Processing Module which takes in the captured images of the game board and determines which move has been made by the human player, the Computer AI Module that takes in the human player's move and outputs the new state of the board, the Serial Communication Module that connects the Computer AI Module to the FPGA through serial USB ports and finally the Display Module that draws all these changes to the computer screen. These five parts, as seen in the Figure 2 below, represent the flow of information that occurs in every round of game play.

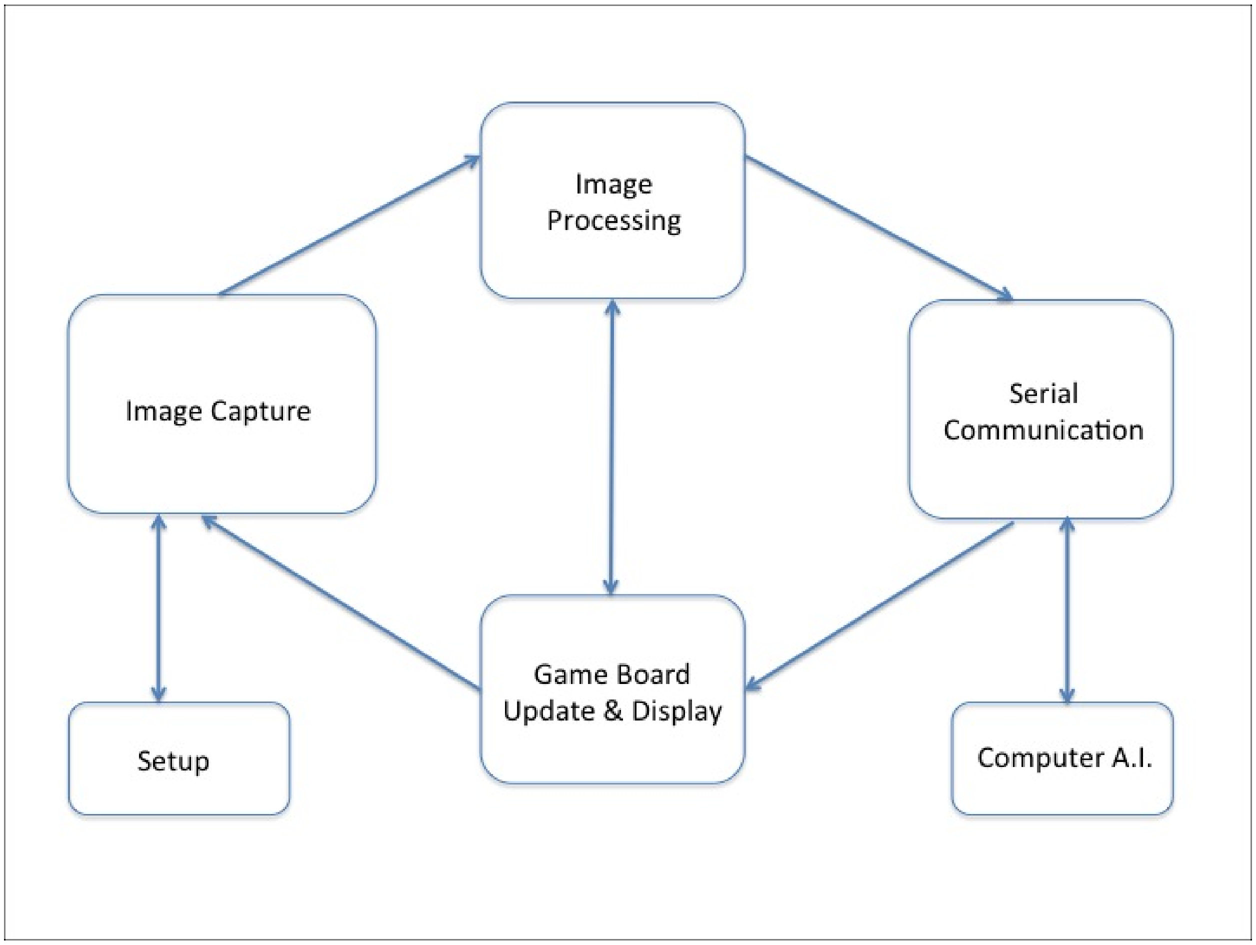

Figure 2: Illustrates the transfer of data amongst the four modules. Each cycle represents a turn of game play.

Image Capture Module(Michael)-

The Image Capture Module uses an NTSC camera interface to transfer live video of the game board to ZBT memory. The FPGA then reads from the zbt memory and is continually redrawing the screen. In the final implementation Switch 5 is used to designate the end of the Human's move. When the Human's move is done, a snapshot of the current state of the board is required. This is accomplished by using a simple conditional assign statement that sets the Write Enable from the camera to ZBT low if the humans move is completed. With the Camera no longer writing to ZBT the display will only read that snapshot of memory. The original NTSC camera code outputs a Black and White image straight to the screen using the YCRCB form of representing colors. The camera actually outputs the full YCRCB value which, if read in, would let a color image be displayed. Yet the original code only used the first few bits which represent the Black and White versions of the image. These B&W pixels were then sent into ZBT in sets of 4 per location. In order to get color video, the size of each pixel needed to be expanded to hold the entire 18 bit YCRCB value. This presented an interesting problem in that each address of ZBT memory can now only hold 2 (instead of 4) pixel values. In order for the display module to read the data correctly the clock needed to be changed to properly read and write to the ZBT. Once the pixel size and timing issues are resolved, the ZBT will hold two 18 bit YCRCB values in each memory address. This pixel (called Vr_pixel) is then outputted into the Labkit Module.

The YCRCB pixel is then converted to RGB using the YCRCB-to-RGB module from the 6.111 course website. This conversion is done using a simple matrix multiplication. After the RGB pixel is created it is then used for analysis by the image processing module.

Setup Module:

When the game is first initialized there is some setup and calibration work that needs to be done. First the initial state of the board is created in BRAM. Once the state of the board is created the computer screen will display a checkerboard with the starting state of the pieces. The monitor is then laid flat and the Human Player places their pieces on the screen. Once this is done the camera needs to be setup. The only real requirements for the setup of the camera are that it needs to be relatively close to perpendicular with the computer screen. The calibration module will then take care of making sure the FPGA knows where every square is on the screen. Once the screen is calibrated the setup is complete the game can begin.

State of the Board(Michael)

The state of the Board is kept in a BRAM. When the game first starts the BRAM is initialized to have the opponents pieces on the first 12 squares, followed by 8 empty squares, followed by 12 human player's pieces. Table 1 below, shows the encoding system we used per piece. The first bit encodes whether or not the space has a piece. The second bit determines whether a piece belongs to the human and the third bit tells you if the piece is a king or not. We decided to do it in this manner because a lot of the comparing we needed to do was to check if the square was empty or to check if there was a human's piece or not in that square. The code runs dramatically faster if you are comparing one bit vs comparing three bits 32 times. This encoding system minimized the length of the bits compared.

Object	Value in Game Board Array
Human Piece	3'b100
Human King	3'b101
Computer Piece	3'b110
Computer King	3'b111
Empty	3'b000

Table 1: The encoding system that is used to encode the state of the Board

The Display Module reads from BRAM in order to determine where to place the pieces on the screen. As moves occur throughout the game the BRAM is updated to reflect that move. An updated State of the Board is necessary for multiple steps in the process of creating this checkers game.

One of our major problems when creating this game was that we initially decided to keep the state of the board in an array of registers and have it update from there. We found out rather late in the game that the version of Verilog we were using did not allow for variable indexing into an array. More current versions of verilog have an operator that allows you to index into an array by designating the start position and a width (Array[StartVariable+:Width]) . The state of the board is never updated until a return move is sent from the computer AI so we did not know about this bug until the time came to integrate the computer AI to the Image Processing Module. In order to solve this problem, we had to completely redesign the code to use the BRAM instead of an array of registers.

Calibration Module(Michael)

The Calibration Module allows the FPGA to determine the locations of all the squares on the checkerboard. An important thing to note before explaining how the calibration module works is that in our final implementation Switch4 was used to alternate between displaying the live video and the checkerboard on the computer screen. In order to calibrate, the live video output is used to set the camera up in a position where it can see the entire computer screen. Once this is done, a snapshot of the screen is taken and the calibration is ready to begin. When the calibration starts, the FPGA displays an 8x8 square array of dots. The user then uses the Up, Down, Left and Right buttons to move the dots onto the image of the checkerboard. Button 2 and 3 are then used to expand and shrink the horizontal and vertical distances between neighboring points. Using this method for shifting and expanding the array of dots, the user can easily place a dot in the center of every square of the checkerboard. Figure 3 below, shows an ideal calibration screen. Once every square has a dot, the calibration is complete. The information of where every dot is can then be used by the FPGA to analyze the squares.

We decided to make this a manual process because since we are doing location based image processing it is vital to have the exact locations of every square on the board. Original ideas for calibration included some image processing to find the corners of the board and then use those values to determine where every square is. Methods like these can be very precise if the camera is setup perfectly but we did not want to have that limitation. Using this manual procedure the user has the flexibility the re-calibrate the screen at every turn, if necessary, and eliminated the necessity of having the camera at a fixed distance from the screen and at a directly perpendicular angle.

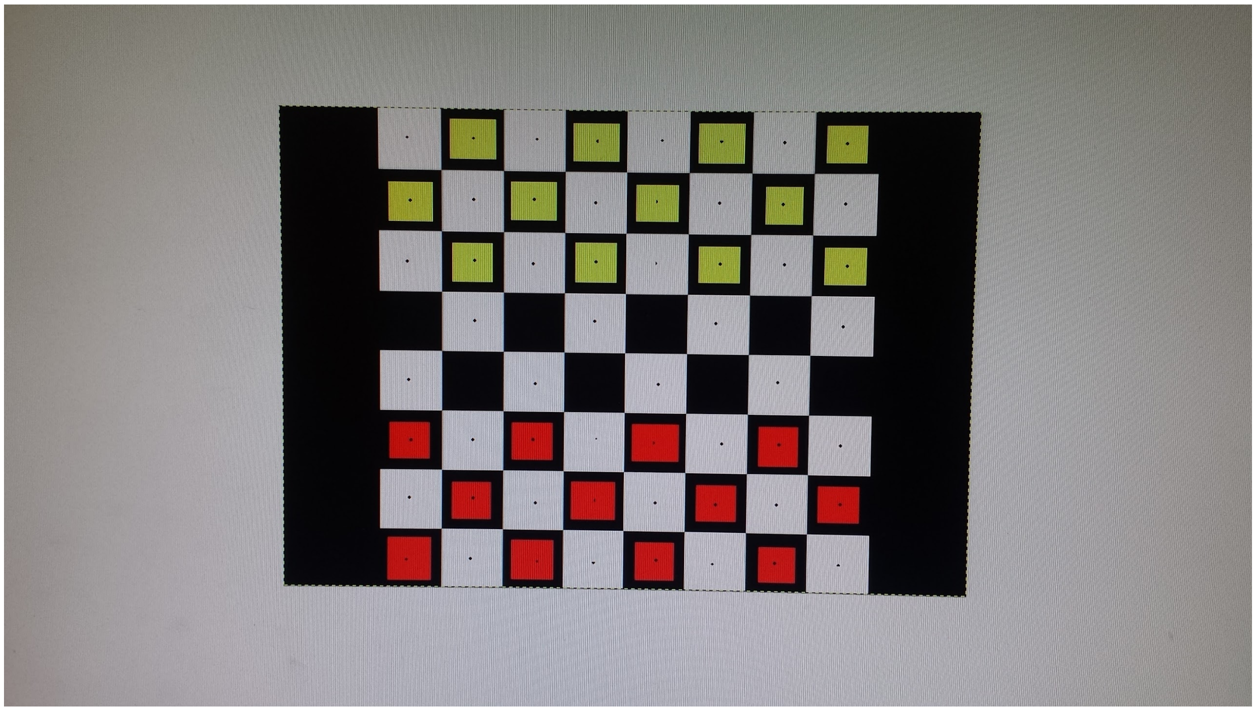

Figure 3: A picture of what the calibration screen looks like. A dot at the center of every square lets the FPGA know the location of every square.

<u>Image Processing Module (Michael)-</u>

The goal of the Image Processing Module is to determine which move the Human Player has just made. Figure 4 below, shows an illustration of the submodules necessary to complete this action.

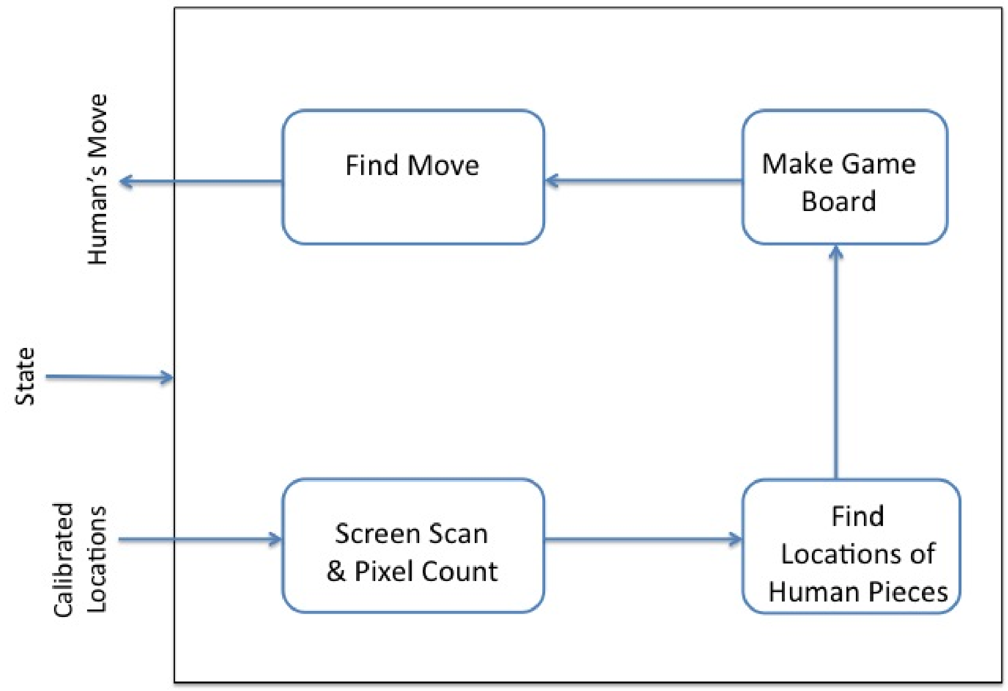

Figure 4: The submodules of the Image Processing Module.

Screen Scan Module(Michael)

The Image Processing Module takes in the RGB pixel data (that is converted from the ZBTs YCRCB data) and the locations of the dots from the calibration module. The first step in determining which move is made is to scan the screen and determine where the Human's piece are currently located. The screen is scanned by taking every 2x2 pixel dot from the calibration step and making a 9x9 pixel square around them. Each pixel in those 9x9 squares is then analyzed. In our implementation we used red pieces as the human's pieces and blue pieces for human's kings so the code checks to see if these pixels in the 9x9 square are either red or blue. To distinguish between white and any color the code not only checks for a value of red or blue higher than a threshold but it also checks to make sure the other values are less than a designated threshold. Each one of the 81 pixels is analyzed and if it meets the criteria it increments a counter for that square. This submodule outputs the ‘count’ per square of both red pixels and blue pixels.

When we originally tested the image processing module we had the FPGA draw the red human player’s pieces on the screen. Searching for these pieces was actually really simple. We were able to set up a high threshold for the red value of the pixel and get great results. But, when we began testing this module with the actual checkers pieces, the red in the video wasnt as prominent so we had to drop the threshold per pixel. Overall we were very surprised at how well this process worked for find counting pixels.

<u>Find Locations of Human’s Pieces Module(Michael)</u>

Once the screen scan is completed two arrays of 32 values is created. The first array is the count of red pixels per square on the board. The second is the count of blue pixels per square on the board. This information is then used to determine if there is a human’s piece in that position. In the following Module summary I will focus in on how the Human player’s non-king pieces get found. To find the Human’s Kings is the exact same process as finding the non-kings except the analysis is done for blue pixels instead of red pixels.

Using the 32 bit array of the number of red pixels in each square, the FPGA determines which squares have pieces in them. This is done by carefully setting a threshold for the number of red pixels needed to designate the square as a piece. There needs to be a careful balance between this threshold and the threshold per pixel set in the ScanScreen module to ensure that noise would not affect the gameplay. Using a 9x9 square means that the max count for any square on the board is 81. We determined, by outputting values of the counts to the hex display, that the scan screen would easily output counts of 50-60. With some counts even close to 70. With this information we decided to get the threshold at 48 of the 81 pixels.

This module loops through that 32 bit array and checks if the count per square is over 48. This information is the used to create a new 32 bit array called Locations which outputs the state of every square. Once this is completed the information is sent to the Find Move module.

Make GameBoard Array Module(Michael)

Before the Find Move module executes it needs the state of the board. The Make Gameboard Array Module loops through every address in BRAM and writes the gameboard into 3bitx32location array. This array holds the previous state of the board before the human's move was made.

Find Move Module (Michael)

The find move module takes in the previous state of the board from the Make Gameboard Array Module and the location of all the Humans pieces from the Find Location of Human's Pieces Module. It uses these two states to determine what move has been made. Every move that the human makes creates two differences between these arrays. A location which previously had a piece is now empty and a space that was empty now has a piece. The Find Move module goes square by square comparing the two states and determines what those two changes are. Once these two changes are determined a 'move' is created. A move is usually encoded as a 12 bit value where the first six bits represent the location where the piece move from and the last 6 bits represent the location of where the piece moved to. Most moves are encoded as 12 bits but in case of double, triple or even quadruple jumps we gave 'move' the ability to be up to 36 bits which makes a max of 5 jumps per turn. (Which i do not think is actually even possible). Once this move is made it is sent to two places. The first is the Board Update Module which takes in the move and adjusts the State of the Board to reflect the move. The second place the move goes to is the Computer AI so that the computer can calculate a counter move.

Serial Communication Module (Ahmet)-

The Serial Communication Module, is the module which sends and receives the information of human's move, computer's counter move and captured pieces between the

FPGA and the computer. We physically connected the FPGA to the computer's USB port using an FTDI UM245M USB-to-FIFO module and we implemented the 4 submodules (FPGA Data Sender/Receiver, Computer Data Sender/Receiver) in order to provide serial communication between two devices. The inner structure of the Serial Communication Module can be seen in the Figure 5 below.

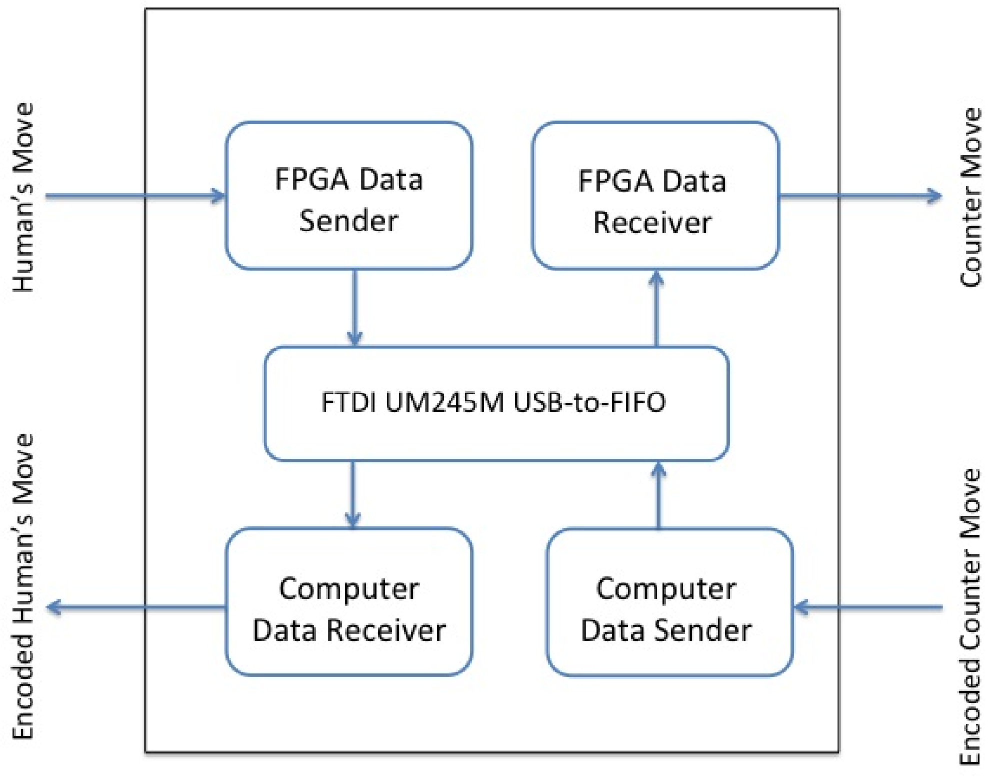

Figure 5: The submodules of the Serial Communication Module, including the physical USB-to-FIFO module FTDI UM245M.

<u>FPGA Data Receiver and Computer Data Sender (Ahmet)</u>

The aim of these two submodules are to send the counter move (which is encoded into 72 bits) from computer to FPGA. A single byte FPGA data receiver was already implemented by Edgar Twigg, who was a former 6.111 student. I downloaded this file from the “Tools” section of the 6.111 website and before writing my own code, I started by understanding this verilog module which receives a single byte of data from the computer to FPGA.

Since receiving a single byte of data at a time would not be enough for our purposes, we developed a new handshake protocol for data sending/receiving and we wrote a new module which receives 72 bits (this number can be easily changed to adjust the bandwidth) of data at a time. Actually, according to our protocol, we were sending 96 bits of data as 12 separate bytes from the Computer Data Sender Module and the useful 72 bits out of 96 bits was being received by the FPGA Data Receiver Module.

Our Computer Data Sender Module was placing a high bit, “1” , at the highest order digit of each byte, because there was a problem with the communication channel and if the highest order digit was not “1”, all the data sent or received was being corrupted. Also since the sending speed of our Python function “Computer Data Sender” was much slower than the receiving speed of our Verilog module “FPGA Data Receiver”, we needed to alternate the second highest order bit between “1” and “0” at each new byte, before sending the byte from Computer Data Sender Module. With the help of this alternating bit, our FPGA Data Receiver Module was able to recognize easily when a new byte is received and was storing the useful information in the new bit (lowest order 6 bits) into our 72 bits register buffer. When the buffer is filled with 72 bits of information, a “done” signal is asserted to inform other modules about the fact that the new counter move from computer is received completely and ready to be used.

<u>FPGA Data Sender and Computer Data Receiver (Ahmet)</u>

The aim of these two submodules are to send the human player’s move (which is encoded into 72 bits) from FPGA to computer. In order to accomplish sending multiple bytes of data at a time from FPGA, initially we wrote a single byte data sender. We realized the problem that, if the highest order bit of a sent byte was not “1”, the received byte was always being corrupted. That is why we decided to put “11” to the highest order 2 bits and to put the 6 useful data bits to the lowest order 6 bits of of every byte. By building on the single byte FPGA data sender, we implemented an FPGA Data Sender Module, which sends 96 bits (12 bytes) of data at a time from FPGA to the computer.

According to our protocol, the FPGA Data Sender Module always sends 96 bits of "1" (or 12 bytes of "0xff") when it doesn't have any new data to send. Using this protocol, the Computer Data Receiver Module understands that there is new useful data and starts receiving the data if there is not any "0xff" byte in the 12 bytes that is being received. Actually, if we wanted to send the 6 bits location "111111" (which represents, row = 7, column = 7) among these 72 bits, our protocol causes a problem (because our protocol can't transmit the byte "0xff") and no information gets transmitted. However, in the 8 by 8 checkers game, the square (7,7) is a white square and there is never a piece on this square, which means we never need to send "111111" through our channel. After removing the highest order 2 bits ("11") of each byte, Computer Data Receiver Module outputs the 72 bits that has the useful information. The information that is encoded in these 72 bits is actually the human player's next move.

When we first implemented the FPGA Data Sender and FPGA Data Receiver modules in Verilog, we thought that these modules would work with 27 MHz clock cycle. However, since in our project, we are also displaying a 1024*768 image on the screen, we needed to use a 65 MHz clock cycle. In order to satisfy the timing constraints of our FTDI UM245M chip with a faster clock speed, we needed to increase the number of clock cycles between data transmissions. By putting extra idle states between signal transitions in our Verilog implementations, we were able to solve this timing problem.

Computer AI Module (Ahmet)-

The Computer AI Module, is the module which receives the human player's move through the Serial Communication Module, calculates the optimal counter move against it and send this counter move to the FPGA through the Serial Communication Module. It also includes "Gameplay" submodule which runs the Checkers game on the computer side in an automated fashion once it gets started.

As seen in the Figure 6 below, the Computer A.I module can be divided into two submodules: Gameplay Module and “Cobra Draughts” A.I. module.

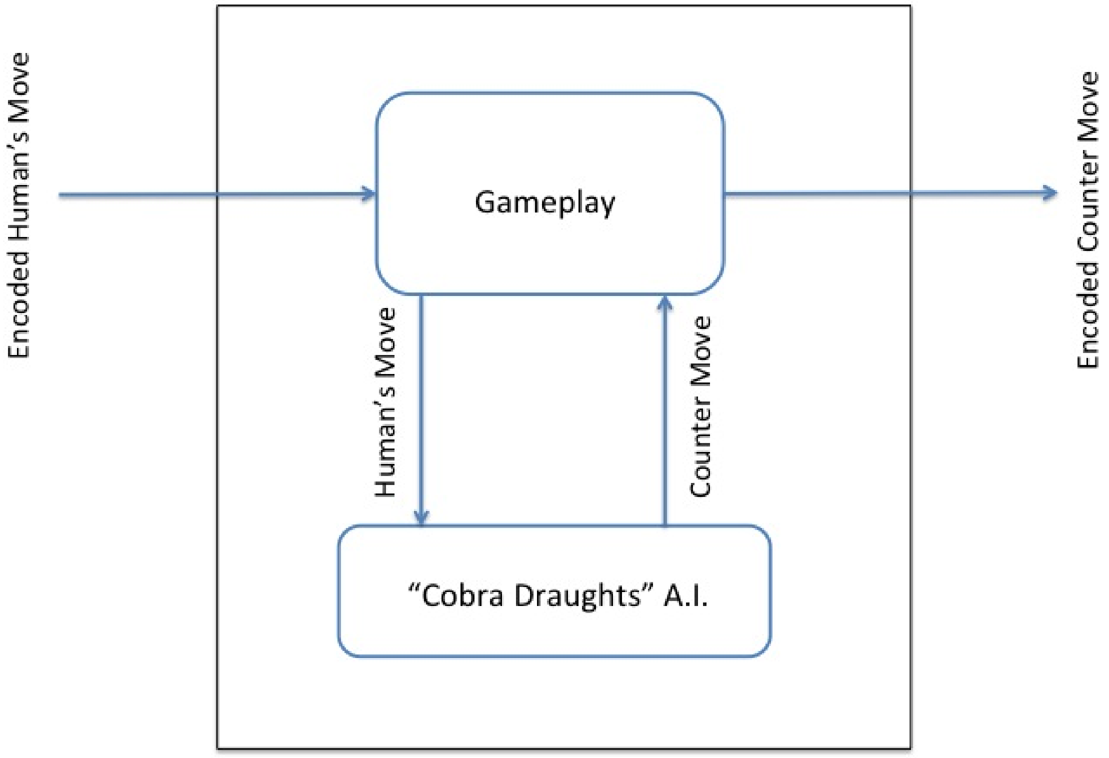

Figure 6: The Computer A.I. Module, which outputs a counter move for any given move of the human player

“Cobra Draughts” A.I. Module (Ahmet)

Cobra Draughts is an open source checkers A.I. algorithm which is written for Python and the code for this algorithm can be downloaded from its official project website https://code.google.com/p/cobra-draughts/ . This is a smart A.I. algorithm which uses the minimax search algorithm along with alpha-beta pruning method to calculate an optimal counter move given any input move.

We downloaded the algorithm from the official website and started to integrate it into other parts of our system. At this point we realized a problem that Cobra Draughts A.I. algorithm was written for a checkers game with a 10 by 10 board, however our interactive checkers game implementation was designed for an 8 by 8 board. In order to make the code applicable into an 8 by 8 board, we modified it. We also needed to readjust the value of “kings” and “regular pieces” in the algorithm, since with a smaller board, promoting from regular piece to king was easier and these values has a crucial effect in the intelligence of the algorithm.

In the Cobra Draughts A.I. algorithm, the moves that are applied onto the board had a Python class structure with the chain of moves and captured pieces being nested inside the original move. However, in our case we needed to encode this class structure into an array of bits, in order to send the moves into FPGA through our Serial Communication Channel. We developed an encoding which converts the location of jumps and location of captured pieces into 6 bits of information, first 3 bits representing the row and second 3 bits representing the column of a location. We thought that on an 8 by 8 board, probably the longest chain move would have less than 6 jumps and we decided to encode the moves into 72 bits. First 36 bits encodes up to 5 jumps of a chain move (first 6 bits being the departure location of the moving piece, second 6 bits being the first jump, third 6 bits being the second jump,...). Next 30 bits encodes up to 5 captured pieces in a chain move and the last 6 bits encodes the information if the moving piece got promoted to king or not at the end of the move. We implemented this encoding method, by writing the appropriate encoder and decoder functions in the Python.

Gameplay Module (Ahmet)

This module is a Python function which executes the game in an automated fashion in a Python Shell. Once this main function is executed, it initializes a game board in the memory of the computer and opens the serial port in order to connect to FPGA. At each round of the game, it runs through the same while loop until the game is over.

If it is human player’s turn to play, the Gameplay Module receives the encoded information of Human’s move from the Serial Communication Module and decodes it using the decoder Python function. After the move is decoded (by decoding we mean it is converted into the special class structure that can be comprehended by Cobra Draughts A.I. algorithm) it is inputted into the A.I. code. The A.I. algorithm outputs the counter move in a decoded style (special class structure) and this counter move is encoded into 72 bits as usual, with the encoder Python function. Then, the encoded information of the counter move is sent to the FPGA through the Serial Communication Module. At this point, a round of the game gets completed and Gameplay Module starts waiting for the human’s next move to start the next round.

Update Board/ Display Module (Michael)-

Whenever a new move is made, the data is sent to the Update Board Module which takes the encoded ‘move’ and makes the necessary changes in BRAM. The Display Module is responsible for drawing the state of the game board onto the screen.
We decided to group these two Modules together because the Display Module is constantly pulling the data from BRAM so that it can draw. So when the Board is updated the screen is also updated simultaneously.

There are three points per round of gameplay where the board needs to be updated. The first of which is when the Find Move Module outputs the human’s move. This move is taken in by the board update module and the change is made in BRAM. The second time the Board gets updates is when the Human’s move gets sent to the Computer AI. The Computer AI first outputs whether the Human’s move resulted in a captured Computer’s piece. This information is used to remove the piece from BRAM. The final time the board gets updated is when the Computer responds with a counter move. When the computer decides what counter move to make it sends that move and also sends information of any captured pieces. This information gets updated in the BRAM. Once those three changes are complete the State of the Board is ready for the next round of gameplay.

At those three points per round of gameplay, the Display Module also redraws itself using the updated game board as seen in the Figure 7 below. This is done by using an intermediate module which converts the newly updated BRAM to an 128 bit array. This array holds the state of each of the 32 squares in 4 bits. See Table 1 to understand how the state of each square is encoded. We use the same encoding as Table 1 except we add a 0 as the highest order bit. This is just done for debugging processes. It allows us to easily output the state of the board to the hex-display to see the contents of each square.

The Display Module uses this 128 bit array to redraw the screen. The way it does this is using a modified version of the Blob Module from the Pong game. Every square on the checkerboard is 96x96 pixels and in the center it has an 80x80 blob. All we do to update the screen is use the state of the board to set the color of each blob. For example if the board has a computer's non-king piece in a square the 128 bit array will have a 4'b0110 value for that square. The Display Module then sets the color of the Blob that is in that square to yellow. There are 5 possible values each square can take. In our final implementation Computer's pieces were yellow and computer kings were purple. Once the new state of the game board has been drawn, the Display Module sends a "Computer's move completed" signal to the Image Capture Module. This signal represents the end of one round of play.

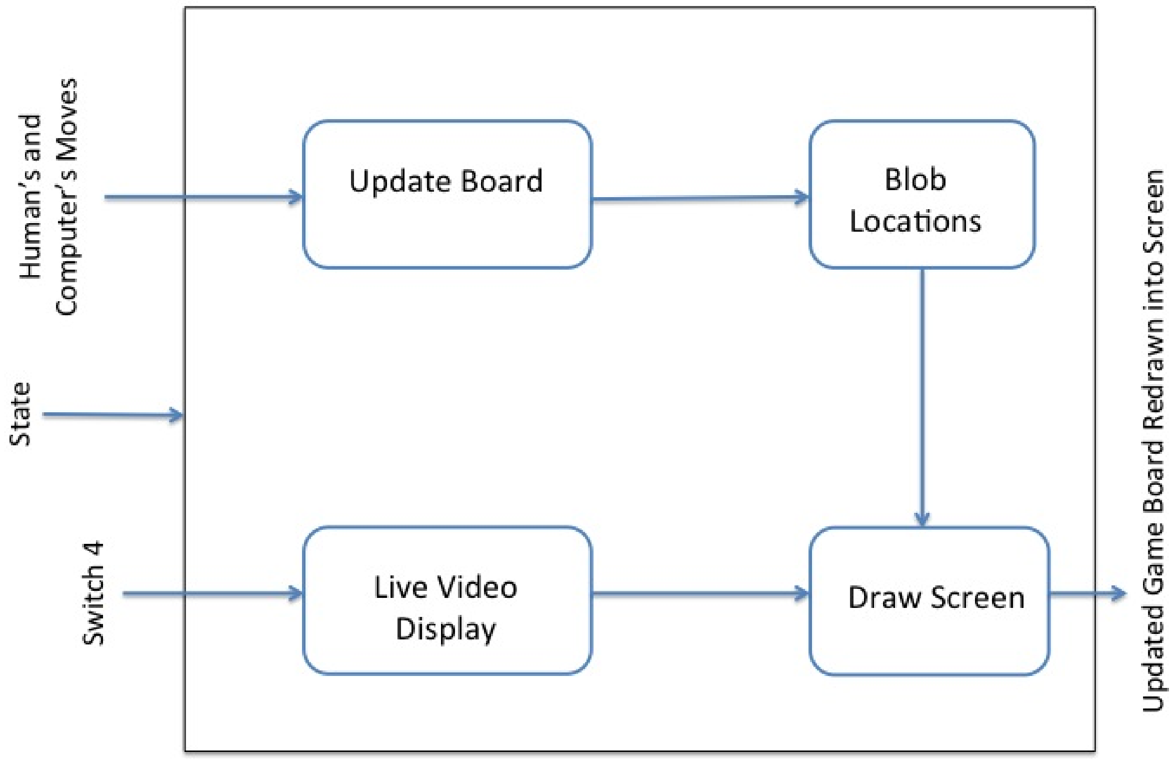

Figure 7: The Display Module with the components necessary to update and output the information to be displayed on the computer screen.

<u>The State Machine (Ahmet)-</u>

Our entire game of checkers is controlled by one state machine. Every submodule outputs a done signal when it is complete and that is read by the state machine. Implementing a high level state machine really simplified the transitions between the states. We were able to easily set the done bit for every state high, wait for the state to change and then set the done bit low again in preparation for the next round of play.

One of my biggest regrets in writing this code, and this is Michael speaking, was not having started the code with a high level state machine. I went in and started building modules and connecting them together in ways that ultimately became too complex to properly implement as the entire project grew. Implementing this state machine organized the whole code to run very smoothly. Also made it very easy to debug because we always knew what state we were in.

Conclusion-

Our “Interactive Checkers” game will create the feeling of playing checkers on an actual game board, which differs from the regular checkers games played through mouse inputs on a computer. The unique user interface and playing style of our game will make it more enjoyable and easier to play. Thus, our game will be a good alternative to other computer checkers games on the market.

Goals-

Our original goals for the project are listed below:

- Set up an interactive checkers environment by placing a flat computer screen horizontally, displaying the checkers board on the screen and placing a camera above the screen.
- Determine the human player’s moves efficiently, through advanced image processing techniques of the camera feed.
- Effectively communicate with a Computer AI.
- Create the basic playing functionalities of a checkers game.

Ultimately we feel like we accomplished every one of these goals and more. Our FPGA was able to effectively determine the locations of every piece on the board. We were able to compare two states to determine what move was made. The move was correctly sent to the Computer AI. The AI’s counter move was sent into the FPGA and the screen was redrawn. Our code worked perfectly for the first round of game play. In the end of the day we had a bug in the way the game board was updated in the final step of a turn.

When the human's move was made, the FPGA was able to correctly find the location of where the piece was moved to. This even worked very effectively with the actual real pieces. But the code was not able to determine where that piece was moved from which threw an error when sent to the computer AI. Besides that one bug in the Board Update Module, the entire game worked perfectly and we are excited about the project we were able to create.

<u>How the Interactive Checkers Game Can Be Improved-</u>

Our stretch goals for the project included:

- Implementing a logic that determines if an illegal move was made by the human player and requesting a new move from the human player.
- Using Image Processing techniques to determine when human player's move is completed.
- Converting the checkers AI code into a Verilog Module and eliminating the need for an extra computer which runs AI code.

Since our priority was to accomplish our original goals as good as we could, we did not have enough time to implement the stretch goals. These additional features, if implemented in the future, can improve the gameplay and increase the usability, functionality of the Interactive Checkers game.