Interactive 1 Player Checkers

Harrison Okun
December 9, 2015

Introduction

The goal of our project was to allow a human player to move physical checkers pieces on a board, and play against a computer's projected pieces. Unfortunately my partner became sick during the course of the project and had to drop the class. Thus our original goals became unfeasible. I modified the project to mainly include the modules I was originally focusing on. I believe the two major parts of our original project were the computer AI and interface and the camera and image processing modules. I was to focus on the computer AI and interface and my partner on the camera and image processing modules. Thus I scrapped the camera and its associated modules.

This meant the goal of the project became to play against a computer and have the board be displayed on a monitor through VGA. It also meant changing some modules and implementing ones I didn't expect to right at the end of the project. As a result I didn't get to implement a few features I would have liked to.

We originally intended to use the Nexys 4 board. However we decided to switch to the labkit because it was much easier to interface with, both for the camera and the computer. We were going to use the microSD card on the Nexys 4 to store the checkers images. However, this was unnecessary and we realized storing them in the labkit's memory was much simpler. Due to a lack of time, I ended up just representing checkers as different colored squares.

This project is split up into five major modules: the FSM, game clock, display module, USB interaction and button input. The FSM controls the flow and communication of the whole system. It stores the current board configuration and sends that to the display. When it is the player's turn it allows for the player to input their move. After they move it sends the new board to the computer through a USB adapter. The computer then runs a checkers AI program to determine its move. It sends the board configuration back to the FSM over USB. The FSM then sends this board to the display. The Game Clock is hooked directly into the signals for new moves. When the USB modules reads in a new move it turns an enable signal high to notify the FSM. The game clock switches to the player's turn on this signal. To move the player hits the enter button on the FPGA. The Game Clock is hooked directly into this button. All buttons were hooked into a debouncer module. However, I mostly needed them to signal once. For example, a selector was used to allow the player to input their move. This was moved by the up, down, right and left buttons. I needed one press of the button to correspond to moving one square. Thus all the buttons were hooked into a collector module which outputs a high signal for only one clock cycle for every half second the button is pressed. This way one can hold the button down to continuously move the selector or press and release to move one square.

Design

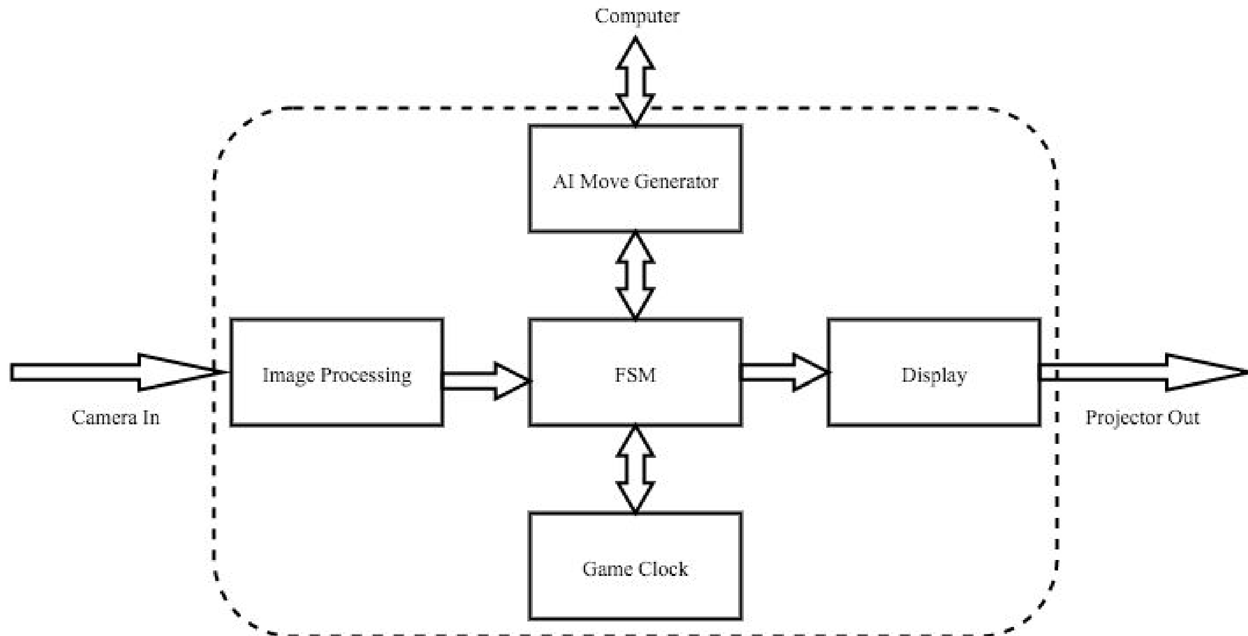

Figure 1: Original High Level Block Diagram

Project Overview and Block Descriptions

Originally there were five main components of our project as shown in Figure 1: Image Processing, AI Move Generator, FSM Logic, Display and Game Clock. The Image Processing block interfaces with the camera, telling it when to take images. It then processes these images to determine the board configuration. It transmits the board configuration to the FSM. The AI Move Generator interfaces with computer. It receives new board configurations from the FSM after the player has moved and sends this to the computer. It then receives the new board configuration from the computer and transmits this to the FSM. The Game Clock shows how much time each player has. It receives signals from the FSM to switch which clock is ticking and tells the FSM when one player's clock expires. The Display module receives the board configuration from the FSM, translates it into VGA and sends this to the projector. The FSM acts as the control unit for the whole systems. It allows communication between the modules and keeps track of the state of the game.

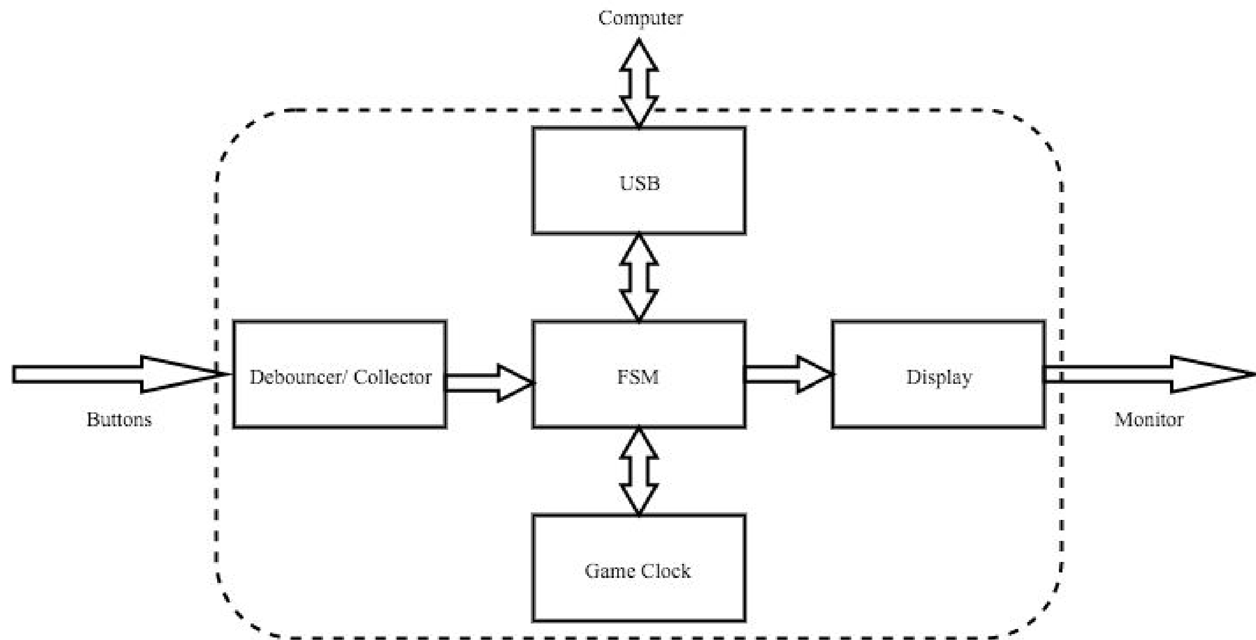

Figure 2: Final High Level Block Diagram

In the final version, three of the modules are very similar to above. The game clock module is identical to the original version. The Display model now outputs an entire board to the monitor instead of just the computer's pieces to a projector. The FSM model now allows for the user to input moves from the input of the buttons. The debouncer/ collector module just cleans up button input and turns it into a single clock cycle high signal. The USB module allows the FSM to send and receive boards from the computer. The design here is the same as proposed above, the name is just changed to represent the actual function of the module.

Design Decisions

In designing our project we had to make some decisions on how certain features would work. We decided to implement the AI on the computer because of our belief that doing so on the FPGA would be impractical. The algorithm would require quickly accessible memory, which the FPGA has a very limited memory (flash memory for example would be way too slow). We will represent the board configuration as a 96-bit binary number. There are 32 valid board locations in checkers. In each location, there can be five possible items: nothing, red piece, black piece, red king or black king. Thus we can represent each location with 3 bits. We construct our board representations by starting in the upper left corner and going row by row down the board, appending the code of the item in each location.

Move input was a significant decision not made at the beginning. I decided to use a selector to achieve this. The selector is essentially a highlighted square on the board. The player can move the selector around the board, remove pieces and add their own pieces back. Thus to input a move they select the piece they want to move, remove it, remove any pieces that are jumped and put the piece back in its new location. They then use the enter button to indicate they have moved. I represented the player's pieces in red and the computer's in green. Kings are represented by a darker shade of the same color.

Implementation

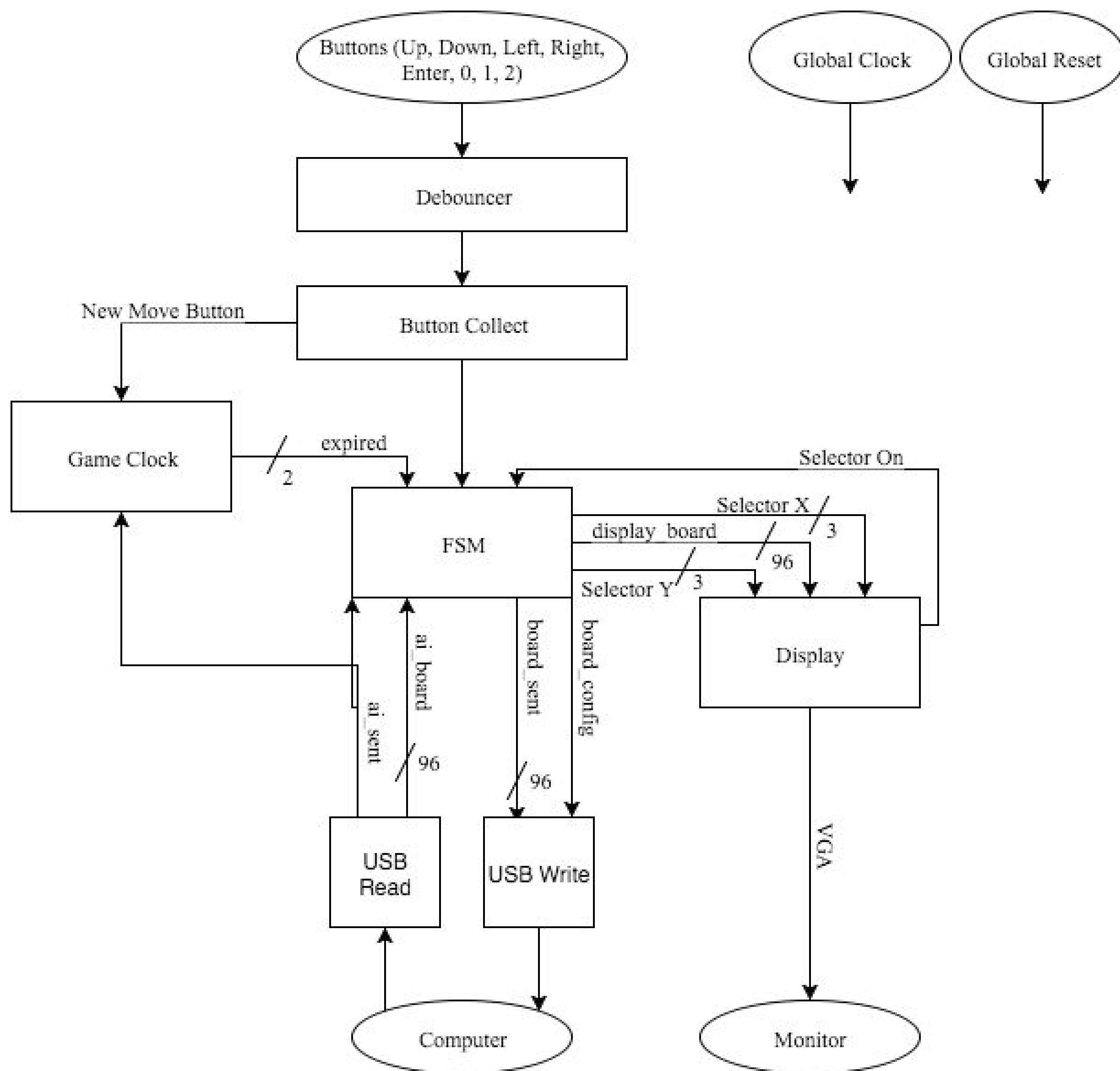

Figure 3: Final Detailed Block Diagram

Modules

The above diagram gives a detailed look at the modules that were used and how they communicated. I now give a description of the purpose of each of these modules and a brief explanation of how I tested them. The modules are organized in the order they were created.

Debounce module

The Debouncer debounces and synchronizes external button inputs so they can be used with the clock synchronized modules. I used the Debouncer module provided in lecture.

Button Collect

The button collector module takes in the output from the debouncer module. It turns the constantly high signal into one that is only high for one clock cycle every half second. It has a counter register. While the button is pressed the counter is incremented by one each clock cycle. When it hits 13,500,000 (half a second for a 27mhz clock) it resets and the output goes high for one clock cycle. If the button is ever not pressed the counter resets. I chose half a second because it is long enough that pressing and releasing won't accidently go high twice and short enough that it can reasonably be held down to simulated repeated presses.

This module was simple enough that it was tested on the FPGA. I displayed a counter for how many clock cycles the output of this was pressed on the hex display. The I can easily see if the desired one clock cycle high signal is achieved.

Game Clock module

The Game Clock module displays the time remaining for both players on the hex display. It has clock cycle counters for both players. It also has second, ten second, minute and ten minute counters for both players. This is because getting the amount of each remaining from the number of seconds or clock cycles is actually quite hard. Doing division by anything other than powers of two is very complicated. Implementing all the counters to increment the next one when they hit a set amount and then reset is much simpler and more efficient in terms of calculations.

The Game Clock takes input from the enter button which signifies a player move and the USB read done signal which signifies a computer move. On either one moving it pauses the current timers and switches to the other ones. This was actually implemented by a state register indicating whose turn it was. This was high for the player's turn. Then we add this value to the player clock cycle counter each time and one minus this value to the computer clock cycle counter. The other counters are incremented based on lower order counters independent of whose turn it is. Since these only can increment on the clock cycle

hitting the one second mark they will never increment on the other's turn. 48 is then added to the value of each counter to convert to ascii and then this is sent to the display string module that was provided on the course website.

The game clock sends a 2-bit signal to the FSM when one player's clock expires. The first bit indicates that a clock has expired and the second will indicate which player's clock expired. The game clock is mostly tested using the hex display. Using test benches and modelsim is unfeasible because all the counters increment on very different time scales. The logic behind this module isn't too complicated so just running it on the FPGA is very feasible. Also displaying the output on the hex display shows the internal state this module.

Display module

The Display module takes in the current board configuration and displays it on the monitor. It displays a checkerboard in the top left 512x512 pixels. Each square was 64x64 pixels. As the basis for displaying vga I used the code provided for lab3. To find the color of a pixel I first check if it's in the board range. This just consists of making sure the x and y coordinate are less than 512. If not it's black. If it is then I check digits 6 through 8. This gives the coordinate in the board because it is equivalent to dividing by 64 (there can't be any 1s past 8 because then it wouldn't be in the board range).

If the sum of the x and y board coordinates are odd then it's black. (This provides the checkering of the board). If the sum is even I have to look at the board configuration to determine the color. Looking at the three bits corresponding to (x,y) in the board turns out to be somewhat complicated. This is because it's not possible to index into a register with variables (i.e. call board[x+3:x]).

What I did was create a pointer wire which is assigned the value of x*12+(y/2)*3. Since y can be odd, y/2 evaluates to floor(y/2) as it's just a bit shift by one. This formula is used because there are 12 bits in each row and three per square in each row. However only every other square is used. This wire always points to the start of the square I care about in the board configuration. I have another wire which has the value of the square I care about. This is accomplished by assigning it to cur_square=(board >> pointer). It's a 3 bit wire so it only stores the value of one square.

Then to find the color of a pixel in an even square we just look at the value of cur_square. If it's 0 this means the square is empty and we make it white. 1 means player piece and is red. 2 means player king and is dark red. 3 means computer piece and is green. 4 means computer king and is dark green.

The display also displays the highlighted square when it is the player's turn. It takes in selector x, selector y and highilight_on from the FSM. After determining the pixel color, it checks if the square it's in is the selector square. If it is and highlight_on is high then it adds blue to the pixel. This includes pixels in black squares (otherwise the selector disappears when it's on dark squares).

USB Read module

The USB read module is based off of the usb_input module provided on the course website. On top of this I built a module which takes in 12 bytes of data and combines them into a 96 bit string. It then signals that it has read in a board. The way it works is that it's an FSM where the state is the number of bytes it has read in. It also stores an 96 bit output. The default state is 0. When a new byte is read in the state increments and the byte read in shifted by state times 8 is added to the output. When the state hits 12, it goes to state 13 and takes a done_reading signal high. From state 13 it goes to 0 and takes done_reading low sets output to 0. The purpose of state 13 is so that done_reading goes high for a clock cycle (it also resets output but that could be done by state 12). There is also a reset input. This resets the usb_input module and zeroes out the state, output and done_reading signals. When the FSM wants to get input from computer it resets the USB Read module to make sure it is cleared.

The USB read module was tested primarily through ModelSim. I assumed the usb_input module works correctly and assigns values to its outputs. Then I can test that the USB read module responds to these as I expect it to. Essentially I have it read in twelve bytes and check that the state is always what I expect and the outputs are as desired.

USB Write module

The USB Write module is based off a USB byte writing module which I wrote. The byte writing module is very similar to the byte reading module that was provided. It is an FSM which goes through the different stages of writing following the timing specifications given in the USB adapter data file. The USB Write module is also rather similar to the USB Read module. It is an FSM where the state is the number of bits written. It has a 96 bit input. To get the xth bit we take input >>(x*8) and store it in an 8 bit wire. The byte writing module has a success signal which goes high for a clock cycle after it has successfully written. The USB Write module increments the state on this signal. The default state is 0. When it gets a new_data input signal it switches to state 1 and sets the new_data input to the byte writing module high and then waits for success signals. This module also has a reset signal to be used before writing to the computer.

I first tested the USB byte writing module with ModelSim to make sure it behaves as I expect and satisfies the timing requirements. The real testing however is on the board and with the computer. It's very easy for it to seem like it should work on ModelSim and then not work on the actual thing. The testing for the USB write module was mostly done on ModelSim assuming that the USB write module works as desired.

FSM module

The FSM keeps track of the game state and provides a communication channel between modules. It will keep track of whose turn it is and what the current board state is. During the player's turn it allows the user to input a move. The way this works is that it outputs a selector x and y and a highlight_on signal to the display module. When it's the player's turn the FSM turns the highlight_on signal on. When it's the

computer's turn it turn off the signal. While in the player's turn pressing up, down, left or right will move the selector in that direction. This is incremented by just increasing or decreasing one of selector x and y depending on which button was pressed.

The player uses the numbered buttons to edit the pieces. If button 0,1 or 2 is pressed and the selector is pointing to a non-black square then the value of that square in the board representation will be changed to the value of the button pressed (these values correspond to empty, player checker and player king). Once again editing three variable bits in the middle of the string isn't trivial. We store a pointer analogous to the one used in the display module. Then when a button is pressed we update the board configuration to:
((board>>(pointer+3))<<(pointer+3))+((board << (96-pointer))>>(96-pointer)) + (button_value<<pointer)
The first term corresponds to all the digits after the three we care about. The second term corresponds to all the digits before the three we care about. The third term corresponds to the new value of the three digits we care about. The player hits the enter button when they have entered their move.

When the player has entered their move, the FSM switches to the computer's move, turns off the selector and tells the write module that their is a new board to be written. The FSM then waits for the read module to signal that the computer has moved at which point it switches back to the player's move and turn the selector back on. The presence of the selector has the added purpose of indicating that it is the player's move.

Since the FSM interfaces almost entirely with other modules and not outside components, testing for it was focused on ModelSim. Essentially I want to check that switch whose move it was and the player inputting their move worked correctly.

Checkers AI

The Checkers AI was written on my computer in C++. It is a pretty standard game AI. The basic idea is to search through the game tree using alpha-beta search (a modified version of mini-max search). The essential idea here is that if it's my turn I want to find my best move. Well my best move is the move which maximizes the goodness (from my point of view) of my opponents least good move. (They will play the move that is least good for me and I want to maximize this.) Hence the name mini-max. My program searches this way down six levels and then evaluates the position. The evaluation function my program uses is pretty simple, it assigns one point per piece, two per kind, minus one per opponent piece and minus two per opponent king and then sums these.

One situation we want to avoid is giving a good evaluation to positions where the other player could immediately respond by capturing a bunch of pieces. To avoid this my program simulates out all possible jumps before evaluating the position. This is known as a quiescence search since we want 'quiet' positions.

This program was tested by playing against it on my computer. The program is pretty simple but still beat me pretty handily every time I played it (I'm not a very good player).

USB Connection Program

I used a program very similar to one provided by the USB Adapter driver to control the USB adapter from the computer side. I modified to sit in a loop until the game was over where it would wait to read a board. Once it had read a board it would feed it to the computer AI, which would return a new board. That new board would then be written to the USB adapter for the FSM to read in. The actual methods for reading from and writing to the USB adapter were provided. However, it took a decent amount of tinkering on my part to setup up the drivers and get the code to compile. I believe the manual is written for a much older version of Mac OS X.

USB Adapter

The device used to transfer data between the labkit and computer was a FTDI UM245R USB-to-FIFO. It essentially consists of a first in first out (FIFO) buffer. When a byte is written it is added to the back of the buffer. When one is read it is taken from the front of the buffer.

Next Steps

There are several components I would have wanted to implement if I had more time. Obviously the camera and physical board would have been great. Even not including that though there are a few less ambitious features that I think could be added. Storing bitfiles of checker pieces on the board and then displaying those on the screen would have made the game board more recognizable and visually pleasing. In the current input setup the player can input any board they want. Thus they can easily make moves that are not allowed.

It would be preferable if the player were only allowed to enter legal moves. I think it would be very difficult for the FSM to check on its own if the new board was an allowed move. One possible solution for this is for the computer to send back the previous board configuration after an illegal move. This would cause the board to reset to before the illegal move. Ideally the system would indicate in some way to the player that there move was illegal.

Takeaways

Certain parts of the project turned out to very different than expected. I didn't think the USB connection would be a major component, but this turned out to be the most difficult part. The adapter was pretty finicky and I was unable to get it to work perfectly. Writing to the adapter sometimes didn't work. I added a feature where the FSM would resend the board if the write failed. However, there were already bits stored in the buffer of the USB adapter and so it would send a gibberish board. More robustness to the connection not working perfectly definitely would have benefited this project.

Conclusion

I very much enjoyed this class and working on this project. I felt I learn a tremendous amount over the course of this class. Due to the unfortunate circumstances I definitely accomplished much less than I would have hoped. While I scrapped the camera modules completely, there were a few other features which I didn't expect to implement (for example manual input for the player) which now had to. This meant that I didn't have time to implement several features I would have like to have implemented. While I obviously would have preferred having a physical board or some of the other features I didn't get to, it still felt pretty satisfying to be able to play against the computer AI on the monitor.