7x7 LED Matrix Connect 4

Charlee Van Eijk & Shiv Seetharaman

E155 Fall 2017

Abstract

The goal of this project was to implement the game "Connect 4" on a custom built 7x7 RGB LED matrix. The project involved building a custom RGB LED matrix using 49 WS2812D LEDs daisy chained together. To control this matrix, a driver module and a frame buffer to store the current display state was implemented on the FPGA. The Pi received player inputs through arrow keys and prompted players using an LCD screen. Received inputs were transmitted to the FPGA using SPI and then displayed on the LED matrix. The game state and game control was controlled by the Pi, while also implementing animations, and checking if a player had won the game. Overall, the project was a success and all the promised deliverables were met. Two users (red LED and blue LED) could successfully play a game, using arrow keys to select a column in which to drop a token, the dropping to the right position would be animated, and if a player won, the matrix would display the color of the winner.

1. Introduction

For our E155 final project we designed and built a custom two-toned (red and blue) 7x7 LED matrix on which the game Connect Four can be played by two users. The overall system includes arrow keys through which user input is collected, a Rapsberry Pi receiving user input and generating control signals, and an FPGA that uses these control signals to drive the 7x7 LED plane. Both the Pi and the FPGA store the game state, and the Pi detects when the game has been won. An LCD interface prompts users during gameplay.

This can be subdivided into front and back end gameplay systems. The front end consists of the arrow keys (collect user input, send to Pi), Pi (track game state, generate FPGA control signals), LCD screen (prompt users), win-detection. The back end consists of the FPGA and LED matrix. A visual representation of the subsystems is provided below.

1.1. Block Diagram

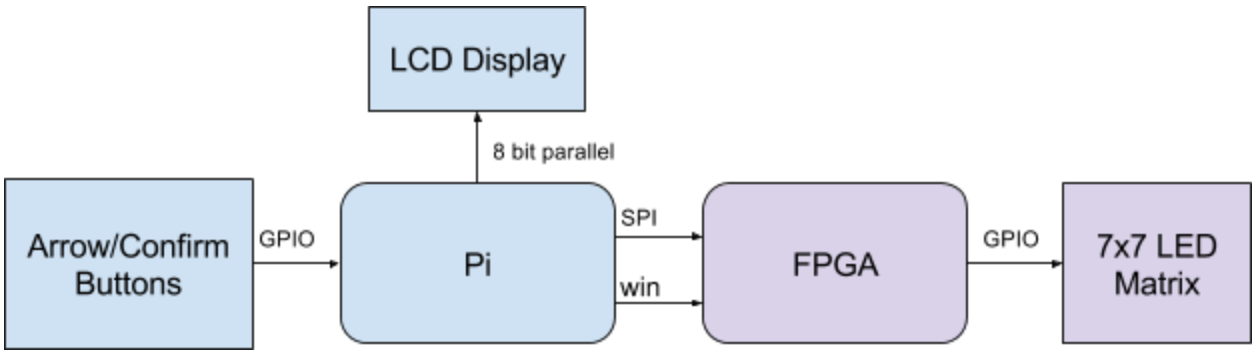


Figure 1: Overall system block diagram

2. New Hardware

One of the goals of the final project was to learn to interface with a new piece of hardware that involved looking at a datasheet and understanding its specifications. This project involved two pieces of new hardware.

2.1. RGB Addressable LED (Worldsemi WS2812D-F8)

These LEDs use a single-wire communication protocol requiring 24 bits of information (8 bits for red, 8 bits for blue, 8 bits for green) for each LED. The data transfer is in the form of a self-clocking signal, a type of NRZ (non-return-to-zero), where ones are represented as a pulse width modulated signal with a high duty cycle, and zeros are represented as a pulse width modulated signal with a low duty cycle. A one consists of a HIGH of 0.8 µs followed by a LOW of 0.45 µs - a zero consists of a HIGH of 0.45 µs, followed by a LOW of 0.8 µs. This resulted in each bit being 1.25 µs long, requiring a data transmission rate of 800 kHz.

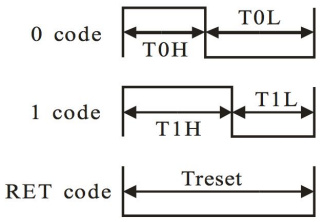


Figure 2: Timing Sequence of Data Transmission

The 24 bits of information are transmitted by sending these pulses serially to the Data-In pin of the LED. The LEDs can be daisy-chained, connecting the Data-Out of one LED to the Data-In of the other and so on. The first 24 bits are used by the first LED and the the rest of the 24 bit packets propagate to the subsequent LEDs. Through this, it is possible to have individual control of all the LEDs connected together. A reset pulse of around 50 µs must be sent if new data is being sent to refresh the LEDs. Although the datasheet says 50 µs, online information suggests that anything above 8 µs suffices.

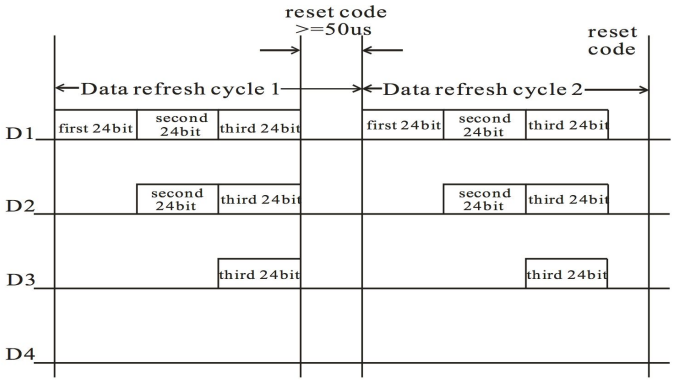


Figure 3: Data Transmission to Daisy-Chained LEDs

Using this technique, a 7 x 7 LED matrix was built using 49 LEDs all daisy-chained together. The matrix was controlled using one wire connected to the Data-In pin of the first LED, allowing individual control of all LEDs to light up with any color. The Altera Cyclone IV FPGA on the MuddPi was chosen to generate the specific waveforms with strict timing to drive these LEDs. The LEDs were placed in a laser-cut piece of acrylic with slots for each LED (pictured on the cover). Each LED represented a token, and lit red or blue up based on which token was dropped.

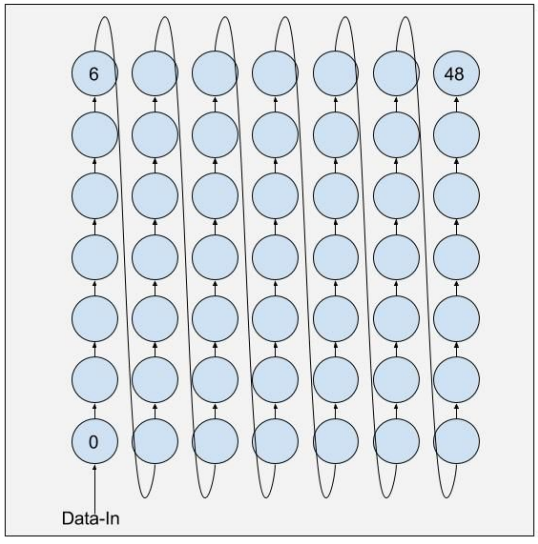


Figure 4: 7x7 LED Matrix design

2.2. 20x2 Parallel Character LCD (Crystalfontz CFAH2002A-TMI-JT)

To display user prompts during the game and show which column in the matrix a user was selecting to drop a token, a LCD display was used. The Crystalfontz display uses a 8 bit parallel interface to transmit data. These displays used the industry standard HD44780 LCD display controller. There is a potentiometer to set a voltage that controls contrast on the display. The LCD requires that information is sent in a particular sequence with appropriate delays between signals. Specifics of this information is outlined in Chapter 9: IO Systems of the Digital Design and Computer Architecture book (Harris and Harris).

The code provided in the Chapter 9 requires that a string, (*char* pointer) be sent to the display. Unfortunately when trying to display numbers (*int*), regular casting isn't sufficient. A common C++ standard library function itoa*(int* value*)* can be used to convert this *string* to an *int*, however this does not run on the Raspberry Pi, so an alternative C implementation was used, after consulting online forums. This allowed displaying numbers on the screens instead of having everything have to be a string.

3. FPGA Design

The Altera Cyclone IV on the Mudd-Pi board was used to implement a frame-buffer (RAM), a LED driver, and a SPI slave to interact with the Pi. The FPGA was chosen for these tasks because it is capable of generating precise waveforms and implementing a frame buffer on the FPGA would be educational.

3.1. Block Diagram

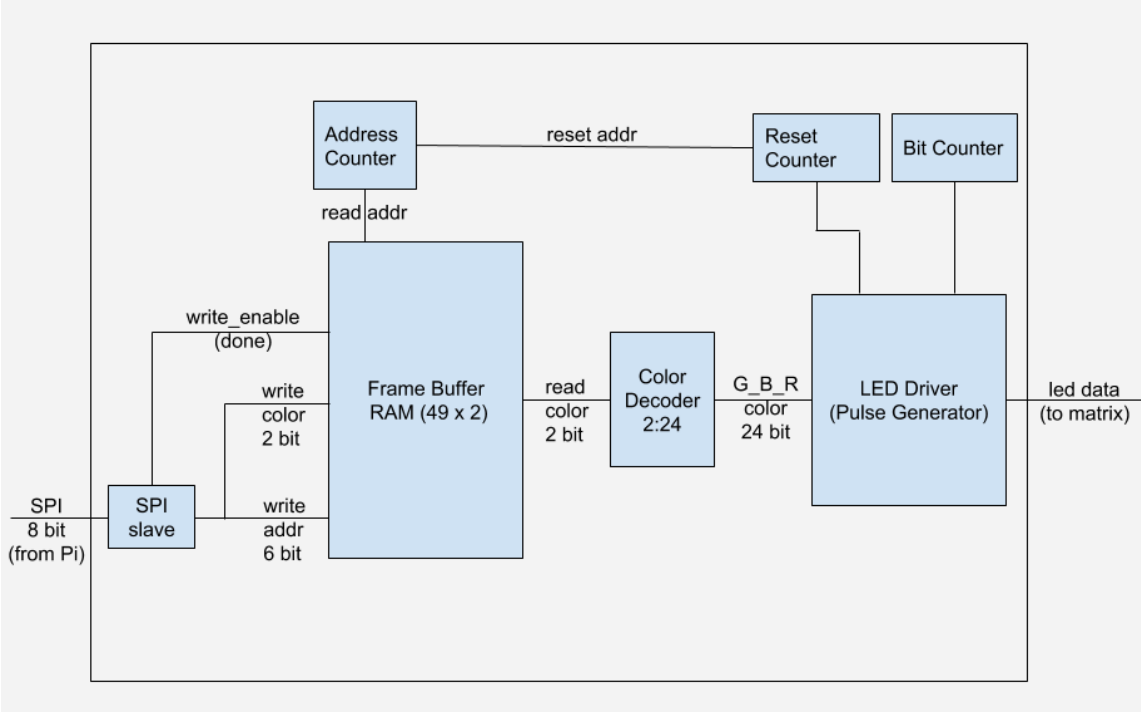


Figure 5: FPGA overall block diagram

3.2. Frame-Buffer (RAM)

The frame buffer implemented on the FPGA served as a way to store the current state of the LED matrix. The frame buffer was 49 entries deep (one for each of the 49 LEDs), where each entry was 2 bits wide. Each entry stores 2 bits for the color of each LED: 00 for off, 01 for red, 10 for blue. The chosen implementation was a dual-ported ram with individual read and write ports. To read a value from RAM, a read address was provided to then output the contents of RAM at that address. To write to RAM, a value and a write address were provided.

Upon each SPI signal, a two bit color value was stored at the write address in RAM. The address counter reset upon each SPI signal so that the new value stored in RAM was accounted for when displaying to the matrix. The LED driver module reads out all the values stored in RAM by incrementing the read address at 33.33 kHz - every 30 µs (time taken to transmit all 24 bits for an LED, 1.25 µs for each bit). In order to change the state of the matrix (when a new SPI signal was received), a reset pulse was sent for 60 µs. This resulted in a theoretical refresh rate of about ~650 Hz ((24 x 1.25 µs x 49 LEDs) + 60 µs), meaning that when everytime the contents of the frame buffer were displayed on the matrix they would be perceived as almost instantaneous.

3.3. LED Driver

The LED driver module consisted of a bit clock running at the required 800 kHz (1.25 µs per bit, fifty clock cycles of the master clock) for the data transmission. To generate the correct waveforms, with the precise timing, a decision to use counters was made instead of relying on the clock-divider formula. The driver consisted of a decoder module to first convert from the 2 bit color value stored in the frame buffer module to a 24 bit value. The approach taken was to increment an index from 0 → 23 at the rate of the bit clock. Using this, a 24:1 multiplexer was used to select which of the 24 bits was at the specified index. Based on whether the bit was a 1 or a 0, another multiplexer was used to select between two flags. A counter to 50 clock cycles (fiftyCount) of the main clock represents 1.25 µs. Based on which flag was selected, a comparator is used to check whether the current value of fiftyCount is less than the selected flag. For that period of time, the led data pin is written high, after which it is written low till fiftyCount overflows. This allows us to generate the variable duty cycle pulse for sending a 0 or a 1 as the LED datasheet specifies.

3.4. SPI-Slave:

The SPI-slave module consisted a sclk, and mosi as inputs, and the data_received as an output. The Raspberry Pi sent 8 bit SPI packets at 244 kHz, using default settings: clock polarity of 0 and clock phase of 0, 8 bits, and no parity. On each rising edge of sclk, data being received is loaded into a shift register. After 8 sclk cycles, the full byte has been received and a flag is asserted. The data received is 8 bits: the 6 least significant bits consist of the memory address that the value must be written at (a value between 0 and 48 for the 49 spots in the frame buffer). The last 2 bits (most significant) consist of the color of the LED being stored in the frame buffer.

These are passed to a 2:24 decoder to generate the correct 24 bit RGB signal that the LEDs require. The SPI-slave module provided the link between the Pi and the FPGA as the Pi could send any spot in the matrix a signal to light up to a color, and the FPGA would drive the matrix such that the LED lit up appropriately.

3.5. Simulation:

Simulation proved to be extremely beneficial for debugging, given the several counters and flags used in the implementation of the above modules. Additionally, it was noticed that things that worked in simulation didn't work in synthesis, calling on the use of a logic analyzer to debug. To simulate using ModelSim, a test bench was written, which included emulating SPI signals and checking to see whether things were being stored and read from the frame buffer (RAM). See appendix C for sample simulation waveforms.

4. Pi Design

The microcontroller design has four main facets: 1) LCD control, 2) button input handling, 3) SPI communication, and 4) game control. Each is discussed in the following sections, and the C code can be found in the appendix.

4.1. LCD

To utilize the LCD we wrote a C header file (lcdDisp.h) which contains several functions necessary for communicating with and controlling the display. Their objectives can be separated into initialization and printing, where each uses several helper functions. As mentioned before, specifics on these functions can be found in Chapter 9 of Digital Design and Computer Architecture. The general idea is that several helper functions helped to initialize the LCD screen in the function 'lcdInit' and once that had been accomplished several more were used to enable the function 'lcdPrintString'. The latter is used to write out prompts to the users during gameplay. One is displayed at the start of the game- "Start game!", and the other is used to indicate whose turn it is- "Red turn! Col x" or "Blue turn! Col x".

4.2. Buttons

Three buttons were used to collect user input. One to increment column number where token is to be dropped, one to decrement column number, and one to drop said token.

To use simple button inputs from Pi GPIO ports 'easyPIO.h' was included in the main C file 'game_control.c'. This enabled use of the 'pioInit' function, which initializes memory mapping for GPIO pins. At that point the 'pinMode' function was used to declare button inputs for incrementing and decrementing columns, as well as for dropping a token.

When a user drops a token in a given column a dropping animation takes place before the appropriate LED is turned on. To do this the Pi sends the FPGA information about which LED in the matrix should be lit up, and what color it should be.

To communicate this information Serial Peripheral Interface communication was used between the Pi and the FPGA, where the Pi was the master and the FPGA the slave. The function 'spiInit' from 'EasyPIO.h' was used to initialize communication between the master and the slave, details of which can be found in Chapter 9 of Digital Design and Computer Architecture. At that point any time information needed to be sent between the Pi and the FPGA a call was made to 'spiSendReceive'. The argument to this function was an eight bit char whose first two bits represented color (red, blue, and off, with an unused two-bit value as well), and whose last six bits represented a number from zero to forty-eight, corresponding to LEDs in the matrix (starting

in the bottom left with zero and ending in the top right with LED forty-nine). The FPGA would process this data, correctly store it in RAM, and then update the LED matrix.

4.3. Game Control

The Pi was responsible for keeping track of the game state, alternating user turns, and ending the game if a win was detected. The main mechanism for doing so was a 7 x 7 char array initialized with Xs. If a blue token was dropped in a given col, the char at the appropriate (correct height, given that a token should sink to the lowest unfilled row) col and row was updated to the char 'B'. The same was true for red tokens, except with the char 'R'. Another 7 x 7 char array stored 8 bit chars of the binary values between zero and forty-eight, representing the index of the LED on the matrix and the address of the LED in RAM on the FPGA. A char from this array would be or-ed with a char representing the color of the current users turn, and that was the char that would get sent over SPI to control individual LEDs in the matrix.

The function 'changeRowCol' in 'game_control.c' handled all game state control. It consisted of a while loop that ran until a win was detected. Three conditional blocks were contained within the loop, one for each button of user input, namely incrementing column, decrementing column and dropping token.

Incrementing and decrementing column allowed users to change the location users expected to place their token, within the bounds of 0 - 6. Hitting the 'confirm' button enabled a series of events to occur. An animation would take place, where an SPI signal lighting up every LED from the top of the column to the lowest unfilled LED in the column briefly and sequentially would be generated. The char array representing the game state would be updated. Next, the char spi signal determining the latest LED to be lit up in the appropriate color would be generated and sent to the FPGA. Finally, the updated game state array would be subject to a 'checkWin' function, which iterated over the char array to see if either player has won the game. If a game was won, the overarching while loop would break and a different animation would play out on the LED matrix, wherein the entire board is filled with the color of the winning player.

5. Results

The overall project was a success. All our promised deliverables were met. Connect 4 was implemented on the custom built RGB LED matrix and allowed two players to play the game together. Animations of token drops and win detection were also successfully implemented. A custom LED matrix driver was built allowing full control of it through an only an 8 bit SPI signal. The FPGA handled driving the matrix and acted as a frame buffer, while the Pi handled player inputs, controlled game state, sent animation signals, and checked for a winner.

The project was challenging in that controlling the LEDs required precise timing as well as a through understanding of how these LEDs could be daisy chained together. Although it became possible to easily control them by "hardcoding" a 24 bit value, making it dynamic and reading these values from RAM added complexity and changed the initial implementation of the LED driver. Integrating with the frame buffer proved to be harder than expected. Issues with timing often meant that it was impossible to debug using the physical hardware and so ModelSim was used as a good way to debug and see internal signals. After making sure things worked in ModelSim, the logic analyzer in the lab was used to ensure simulation and synthesis agreed with each other (in some occasions they did not: storing the "hardcoded" 24 bit value as a logic variable caused Quartus to give an error. Storing this value as a parameter worked instead.

That said, there are aspects of the project that could be improved. We noticed a slight flicker in LED zero (bottom left) whenever a drop animation occured. We presume this was because every time a SPI event occured, the led address counter (that iterated over RAM) reset to zero, and although ModelSim showed that nothing should have been read from RAM to led driver, there may not have been an exact match between physical phenomena and simulation.

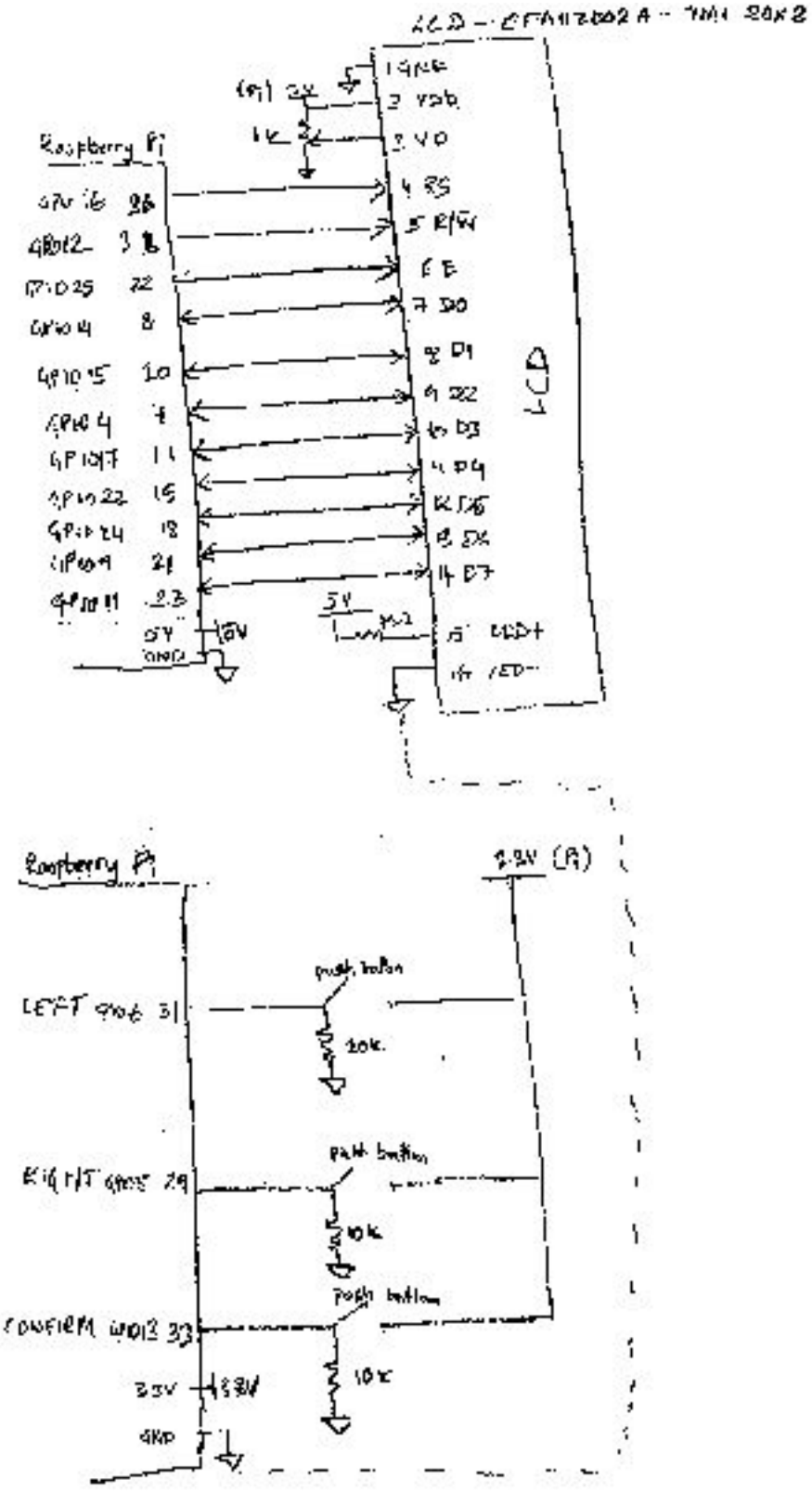
Also, we noticed during user testing on demo day that players sometimes dropped tokens in columns they did not intend to drop in. An animation that lit up the topmost LED in a column that the user was thinking about dropping a token in might serve as a better indicator than the LCD screen.

Given more time, the team would have liked to implement artificial intelligence for connect four, so that a player could play against a computer. But in conclusion, the proposed goals for the project have been met elegantly and robustly.

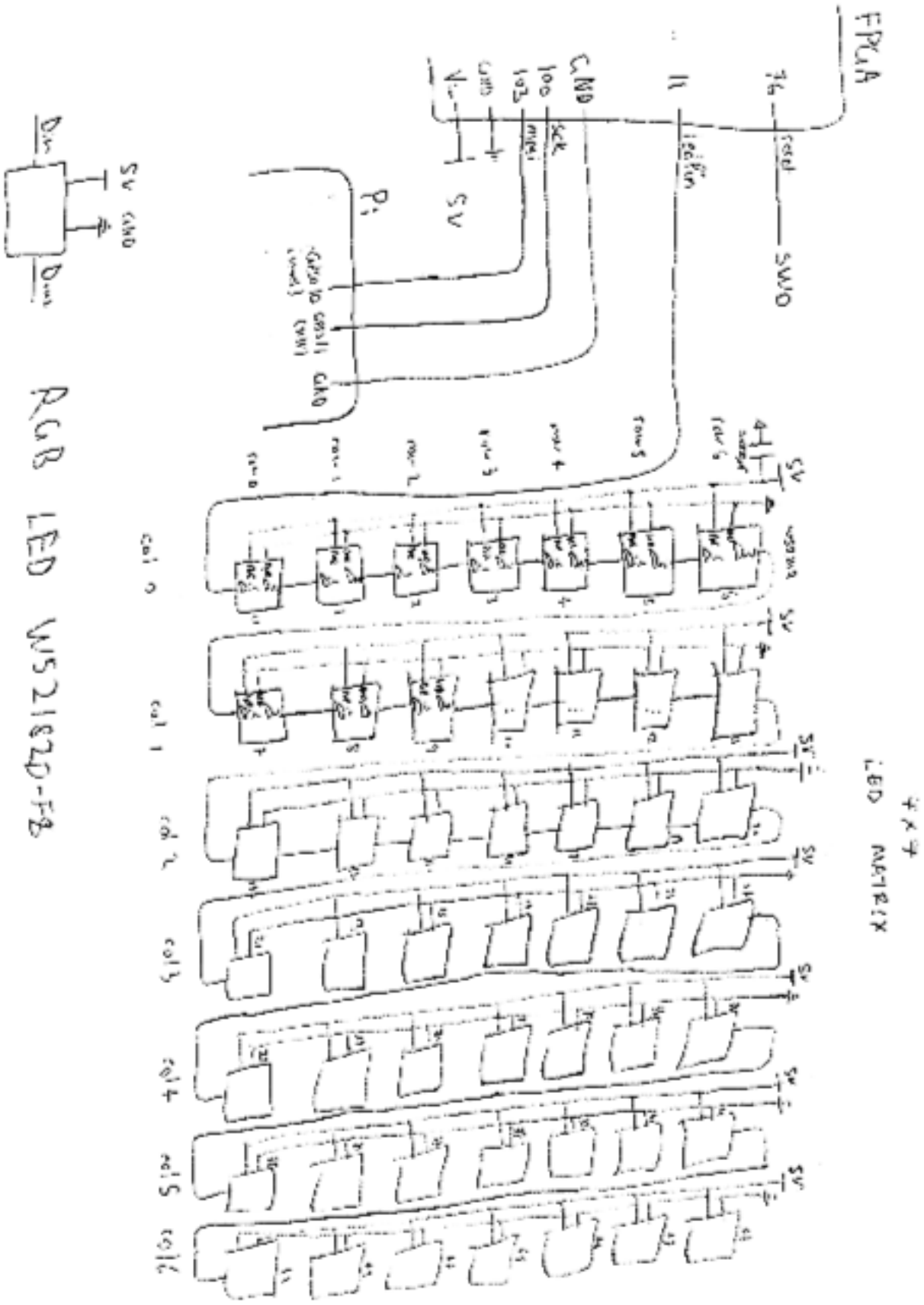
Bibliography

1. EasyPIO.h (http://pages.hmc.edu/harris/class/e155/EasyPIO.h)
2. Chapter 9: Digital Design and Computer Architecture (Harris and Harris)
3. WS2812D - DataSheet (http://www.world-semi.com/solution/list-4-1.html)

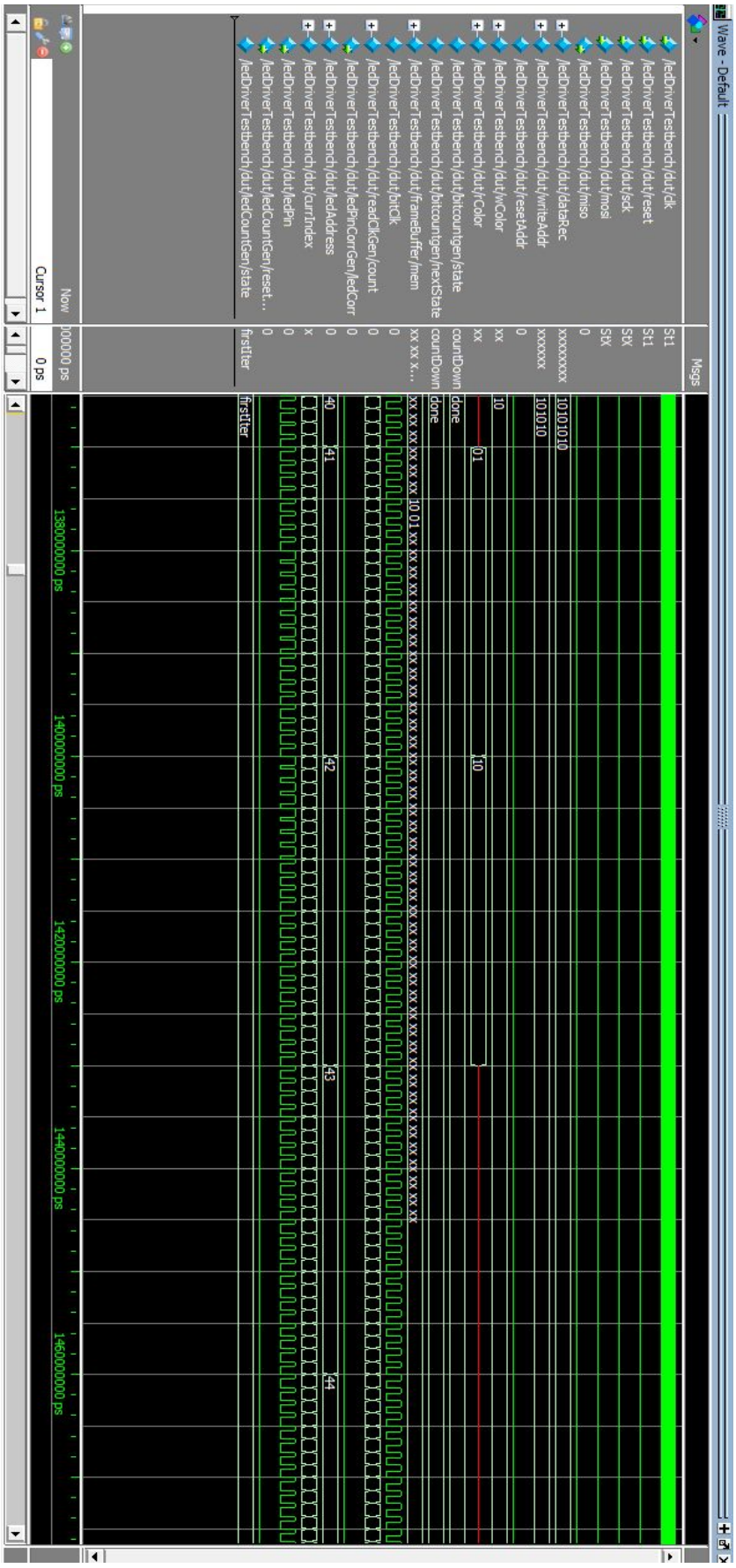
APPENDIX A: LCD + Pi SCHEMATIC



APPENDIX B: FPGA + LED MATRIX SCHEMATIC



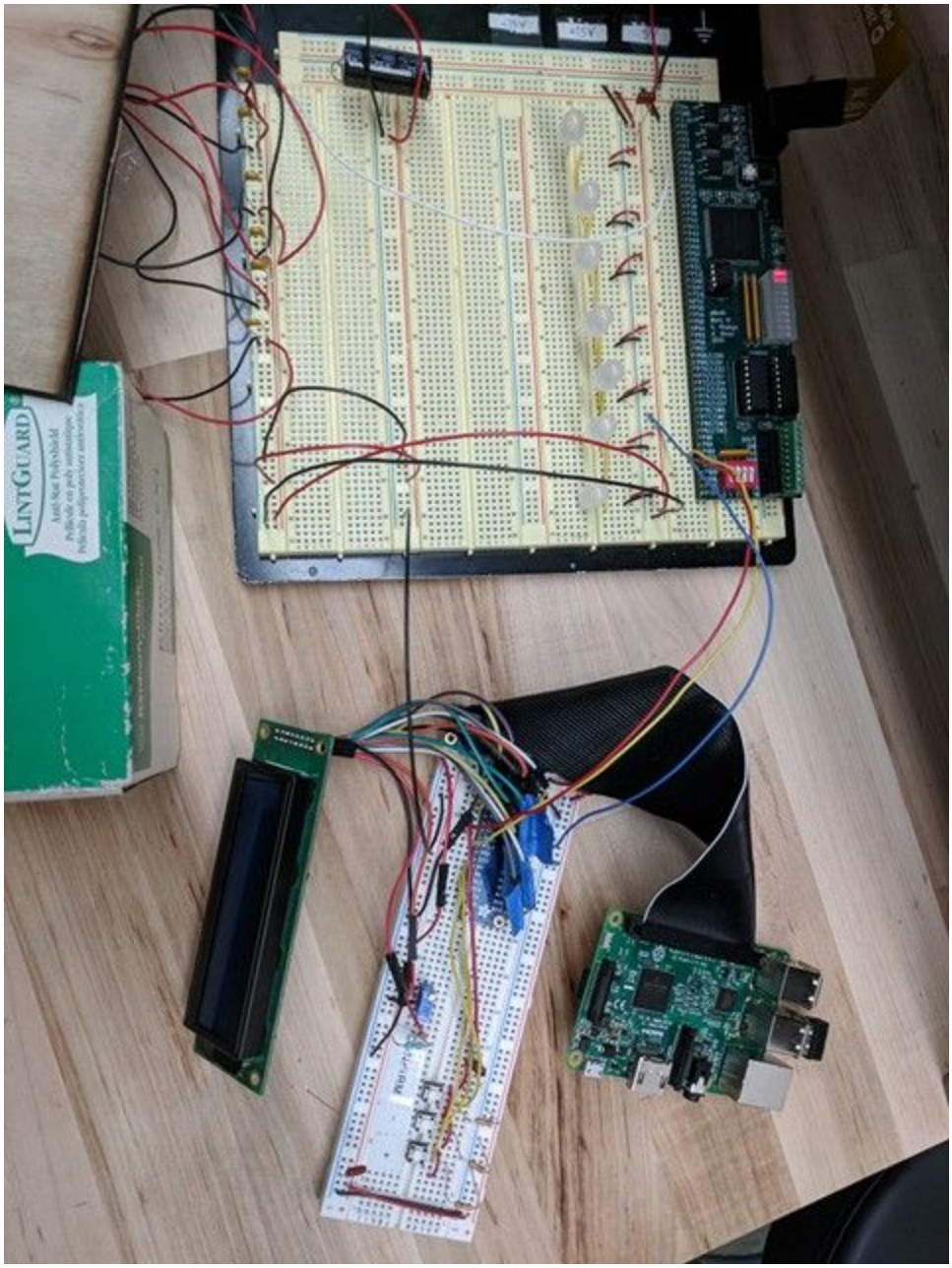
APPENDIX C: SIMULATION WAVEFORMS (Reading from RAM)



APPENDIX D: BILL OF MATERIALS

Item	***Part Number***	***Price ($)***
Addressable RGB LEDs	WS2812D	12 + shipping
LCD Display	CFAH2002A-TMI-JT	8.81 + shipping
Acrylic	-	10.95 + shipping
Buttons	(available in stockroom)	0
		Total = 28.72 + shipping

APPENDIX E: LCD + BUTTONS IMAGE



APPENDIX F: System Verilog Code

```
// E155 - MicroPs Final Project: Connect 4 on LED Matrix
// Charlee van Eijk and Shiv Seetharaman
// cvaneijk@hmc.edu & sseetharaman@hmc.edu
// Date Created: November 12th 2017
// Led Matrix Driver Module

////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////// TESTBENCH ///////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////

// Testbench
module ledDriverTestbench();

	// variables
	logic clk, sck, mosi, miso, reset, ledPin, rClk;
	logic [7:0] led_red_zero_zero, led_blue_zero_one;
	logic [3:0] i;

	// load LED Signals (blue to 41, red to 40)
	initial
		begin
			led_red_zero_zero <= 8'b01_101_001;
			led_blue_zero_one <= 8'b10_101_010;
		end

	// clock
	always
		begin
			clk = 1; #12500;   clk = 0; #12500;
		end

	// dummy SPI Signals
	initial
		begin
			i = 0;
			reset = 1;#5000000; reset = 0;
			sck = 1; mosi = led_red_zero_zero[7]; #2049180; sck = 0; #2049180; // 1
			sck = 1; mosi = led_red_zero_zero[6]; #2049180; sck = 0; #2049180; // 2
			sck = 1; mosi = led_red_zero_zero[5]; #2049180; sck = 0; #2049180; // 3
			sck = 1; mosi = led_red_zero_zero[4]; #2049180; sck = 0; #2049180; // 4
			sck = 1; mosi = led_red_zero_zero[3]; #2049180; sck = 0; #2049180; // 5
```

```
                    sck = 1; mosi = led_red_zero_zero[2]; #2049180; sck = 0; #2049180; // 6
                    sck = 1; mosi = led_red_zero_zero[1]; #2049180; sck = 0; #2049180; // 7
                    sck = 1; mosi = led_red_zero_zero[0]; #2049180; sck = 0; #2049180; // 8
                    sck = 0;
                    #50000000;
                    sck = 1; mosi = led_blue_zero_one[7]; #2049180; sck = 0; #2049180; // 1
                    sck = 1; mosi = led_blue_zero_one[6]; #2049180; sck = 0; #2049180; // 2
                    sck = 1; mosi = led_blue_zero_one[5]; #2049180; sck = 0; #2049180; // 3
                    sck = 1; mosi = led_blue_zero_one[4]; #2049180; sck = 0; #2049180; // 4
                    sck = 1; mosi = led_blue_zero_one[3]; #2049180; sck = 0; #2049180; // 5
                    sck = 1; mosi = led_blue_zero_one[2]; #2049180; sck = 0; #2049180; // 6
                    sck = 1; mosi = led_blue_zero_one[1]; #2049180; sck = 0; #2049180; // 7
                    sck = 1; mosi = led_blue_zero_one[0]; #2049180; sck = 0; #2049180; // 8
                    sck = 0;
            end

  // device under test
        cve_ss_ledMatrix dut(clk, reset, sck, mosi, miso, rClk, ledPin);
endmodule



////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////// TOP LEVEL MODULE ////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////

module cve_ss_ledMatrix(input logic  clk, reset,
                                                                input logic  sck,
                                                                input logic  mosi,
                                                                output logic miso,
                                                                 output logic rClk,
                                                                output logic ledPin,
                                                                output logic [1:0] debug1,
                                                                output logic [1:0] debug2);

        // internal variables
        //                spi
        logic wEn;
        logic [7:0] dataRec, dataToSend;
        logic [5:0] writeAddr;
        logic [1:0] wColor, rColor;
        //                counters
        logic bitClk, twelveCountFlag;
```

```
logic [5:0] ledAddress;
logic [5:0] fiftyCount;
logic [4:0] currIndex;
//          reset signals
logic ledCorr, resetAddr, resetWave;
//                  decoder
logic [23:0] selectedBits;
//          led pulse generation
logic [5:0] ledCount;
logic [5:0] sixteenFlag, thirtyFourFlag, countSelect;
logic currBit;

// spi slave module to determine write address and color
assign dataToSend = 8'b0;
spiSlave dropTokenSPI(sck, mosi, miso, reset, dataToSend, dataRec, wEn, resetAddr);
assign writeAddr = dataRec[5:0];
assign wColor = dataRec[7:6];

// make sure led pin is correct upon SPI
ledPinCorr ledPinCorrGen(rClk, reset, resetAddr, ledCorr);

// creates bitClk (800 kHz) and counter to count to fifty (1.25us)
// necessary for ledPin timing
bitCycle bitCycleGen(clk, reset, bitClk, fiftyCount);

// set rate of generating read addresses
readClk readClkGen(bitClk, reset, rClk);

// iterate through bits in 24 bit value to send to LEDs
bitCounter bitcountgen(bitClk, reset, twelveCountFlag, currIndex);
twelveCounter twelveCounterGen(bitClk, reset, twelveCountFlag);

// generate read addresses every rClk (0 - 48)
ledCounter ledCountGen(rClk, reset, resetAddr, ledCorr, ledAddress, resetWave);

// read and write to frame buffer (RAM)
mem frameBuffer(rClk, wEn, ledAddress, writeAddr, wColor, rColor, debug1, debug2);

// decode 2 bit frame buffer data to 24 bit data for ledDriver
ramDecoder colorToGRB(rColor, selectedBits);

// all verilog below generates ledPin pulses
assign sixteenFlag    = 6'b010000;    // Flag for reaching 16 (used in bitCycle)
```

```
        assign thirtyFourFlag = 6'b100010;    // Flag for reaching 34 (used in bitCycle)

        // dictates which of the 24 bits to choose in GRB and gets that bit from GRB
        assign currBit = selectedBits[currIndex];

        // choose if generating 0.85us high or 0.4us high based on current bit (specific to
WS2812 LED)
        assign countSelect = currBit ? thirtyFourFlag:sixteenFlag;

//      // flop to drive ledPin with waveform of signal
//      always_ff@(posedge clk, posedge reset, posedge resetWave)
//              if (reset)                      ledPin <= 0;
//              else if (resetWave) ledPin <= 0;
//              else                            ledPin <= ~(countSelect < fiftyCount) & ~ledCorr
& wEn; // & ~resetWave; //& ~led_reset_wave;

        // drive ledPin
        assign ledPin = ~(countSelect < fiftyCount) & ~ledCorr & wEn & ~resetWave;
endmodule

////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////// SUB MODULES ////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////


// Module to genrate a reset signal for ledPin and keep it off longer when receiving SPI
module ledPinCorr(input logic  rClk,
                                        input logic  reset,
                                        input logic  resetAddr,
                                        output logic ledCorr);

        // flop to run on rClk but asynchronously reset with resetAddr
        always_ff @(posedge resetAddr, negedge rClk)
                if (resetAddr) ledCorr <= 1;
                else                    ledCorr <= 0;
endmodule

// spiSlave to receive data from Pi and send back dummy signal
module spiSlave(input logic   sck,              // from master
                                input logic  mosi,      // from master
                                output logic miso,      // to master
                                input logic  reset,     // system reset
                                input logic  [7:0] d,   // data to send
```

```
                                        output logic [7:0] q,   // data received
                                        output logic wen,       // write enable
                                        output logic resetAddr); // reset signal

        // internal variables
        logic [2:0] cnt;
        logic       qdelayed;

        // 3-bit counter tracks when full byte is transmitted
        always_ff @(negedge sck, posedge reset)
                if (reset) cnt <= 0;
                else       cnt <= cnt + 3'b1;

        // loadable shift register
        // loads d at the start, shift mosi into bottom on each step
        always_ff @(posedge sck)
                q <= (cnt == 0) ? {d[6:0], mosi} : {q[6:0], mosi};

        // align miso to falling edge of sck
        // load d at the start
        always_ff @(negedge sck)
                qdelayed = q[7];
        assign miso = (cnt == 0) ? d[7] : qdelayed;

        // convey that full byte is transmitted and ready to be written
        assign wen = (cnt == 3'b000);
        assign resetAddr = (cnt == 3'b111);
endmodule

// Frame buffer(RAM) (49x2) module to store current state of each LED in matrix
module mem(input logic       rClk,          // read clock
                        input logic       we,           // write enable
                        input logic [5:0]  rAddr,   // from spi
                        input logic [5:0]  wAddr,   // from counter
                        input logic [1:0]       wd,           // data to write
                        output logic [1:0] rd,           // data read
                        output logic [1:0] debug1,    // debugging led
                        output logic [1:0] debug2);   // debugging led read data

        // 49 entries deep, 2 bits wide
        logic [1:0] mem [48:0];

        // read from RAM at read address
```

```
        assign rd = mem[rAddr];

        // debug LEDs on MuddPi (see what's stored at addr 0 and 1)
        assign debug1 = mem[6'b000000];
        assign debug2 = mem[6'b000001];

        // write to memory at write address on write enable
        always_ff @(posedge we)
          mem[wAddr] <= wd;
endmodule

// Decode 2 bit color signal to generate 24 bits for led driver to use
module ramDecoder(input logic [1:0]   color,
                                      output logic [23:0] GRB);
        always_comb
                case(color)
                        2'b00:   GRB = 24'b00000000_00000000_00000000; // off
                        2'b01:   GRB = 24'b00000000_11111111_00000000; // red
                        2'b10:   GRB = 24'b00000000_00000000_11111111; // blue
                        default: GRB = 24'b00000000_00000000_00000000;
                endcase
endmodule

// Counter to generate and increment addresses in RAM and handle resetting
module ledCounter(input logic  clk,
                                      input logic  reset,
                                      input logic  resetAddr,
                                      input logic  ledCorr,
                                      output logic [5:0] ledCount,
                                      output logic resetWave);

        // fsm to make sure we reset only ONCE after we get an SPI signal
        typedef enum logic {firstIter, rest} statetype;
        statetype state, nextState;

        // state reg
        always_ff @(posedge clk, posedge reset, posedge resetAddr)
                if (reset)                state <= firstIter;
                else if (resetAddr)  state <= firstIter;
                else                      state <= nextState;

        // next state logic
        always_comb
```

```
		case(state)
			firstIter:  if (ledCount == 50)  nextState = rest;
								else		nextState = firstIter;
			rest:	if (resetAddr)	nextState = firstIter;
								else
nextState = rest;
		endcase

	// counter for generating LED addresses
	always_ff @(negedge clk, posedge reset, posedge resetAddr)
		if (reset)							ledCount <= 0;
		else if (resetAddr)				ledCount <= 0;
		else if (ledCount== 50)		ledCount <= 0;
		else if (ledCorr)		ledCount <= 0;
		else 							ledCount <=
ledCount+ 6'b1;

	// flop to control reset wave signal
	always_ff @(negedge clk)
		if((ledCount== 48 | ledCount == 49) & state == firstIter)	resetWave <= 1;
		else
								resetWave <= 0;
endmodule


// Module to count to 12 and is used in bitCounter as a flag for its FSM
module twelveCounter(input logic  clk,
								input logic  reset,
								output logic twelveCountFlag);

	logic [3:0] twelveCount;

	//count to twelve
	always_ff @(posedge clk, posedge reset)
		if (reset)		twelveCount <= 0;
		else if (twelveCount == 11) twelveCountFlag <= 1;
		else		twelveCount <= twelveCount + 4'b1;
endmodule

// Module to count from 0 to 23 to select a particular bit to send signals to led driver
// makes sure to wait 12 clocks at startup to ensure currIndex is synced
module bitCounter(input logic  clk,
							input logic  reset,
```

```
                                        input logic  twelveCountFlag,
                                        output logic [4:0] currIndex);

        // state machine to ensure currIndex always resets when reading from RAM
        typedef enum logic {countDown, done} statetype;
        statetype state, nextState;

        // state reg
        always_ff @(posedge clk, posedge reset)
                if (reset)                      state <= countDown;
                else                                    state <= nextState;

        // next state logic
        always_comb
                case(state)
                        countDown:    begin
                                                        if (twelveCountFlag)   nextState =
done;
                                                        else
nextState = countDown;
                                                end
                        done:
        nextState = done;
                        default:
        nextState = countDown;
                endcase

        // current index counter
        always_ff @(posedge clk)
                if(state == countDown)
currIndex = 0;
                else                                                    begin
                                                                        if (currIndex == 23)
currIndex = 0;
                                                                        else
currIndex = currIndex + 5'b1;
                                                                       end
endmodule



// Generate clock to increment led addresses for frame buffer
// period is 24 bit clock cycles == time taken to send one LED data packet
```

```
module readClk (input logic  clk,
                                    input logic  reset,
                                    output logic clkOut);

        logic [4:0] count;

        // count to 23 (24 bit clks == 30 us)
        always_ff @(posedge clk, posedge reset)
                if (reset)                                      count <= 0;
                else if (count == 23)   count <= 0;
                else                                                    count <= count + 5'b1;

        // generate clk with period of 30us (33.33 kHz)
        always_ff @(posedge clk, posedge reset)
                if (reset)                                  clkOut <= 0;
                else if (count < 12)  clkOut <= 1;
                else                                                clkOut <= 0;
endmodule

// Module to count from 0 to 50, and generate clock signal with 1.25us (800kHz)
// signal (required by WS2182 LED) to drive LEDS
module bitCycle(input logic   clk,
                                    input logic   reset,
                                    output logic  bitClk,
                                    output logic  [5:0] fiftyCount);

        // count to 50 (50 master clks == 1.25us)
        always_ff@(posedge clk, posedge reset)
                if (reset)              fiftyCount <= 0;

                else if (fiftyCount == 49)  fiftyCount <= 0;
                else                    fiftyCount <= fiftyCount + 6'b1;

        // generate clock 800 kHz clock signal

        always_ff@(posedge clk, posedge reset)
                if (reset)              bitClk <= 0;
                else if (fiftyCount < 24) bitClk <= 1;
                else                    bitClk <= 0;
endmodule


//// Simple mux to select 1 of 24 bits of GRB data
```

```
//module mux24_1(input logic [23:0] GRB,
//                                  input logic [4:0] currIndex,
//                                  output logic currBit);
//
//        assign currBit = GRB[currIndex];
//endmodule
```

APPENDIX G: game_control.c

```
// game_control.c
// Charlee Van Eijk
// Shiv Seetharaman

// Necessary includes
#include <stdio.h>
#include "lcdDisp.h"

// Define constants
#define NUM_ROWS 7
#define NUM_COLS 7
#define DELAY    500

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////    SETUP       ////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

// Define boolean variables
typedef enum {false, true} bool;

// Char array representing game board
char board [NUM_ROWS][NUM_COLS] =
          {{'X', 'X', 'X', 'X', 'X', 'X', 'X'},
           {'X', 'X', 'X', 'X', 'X', 'X', 'X'},
           {'X', 'X', 'X', 'X', 'X', 'X', 'X'},
           {'X', 'X', 'X', 'X', 'X', 'X', 'X'},
           {'X', 'X', 'X', 'X', 'X', 'X', 'X'},
           {'X', 'X', 'X', 'X', 'X', 'X', 'X'},
           {'X', 'X', 'X', 'X', 'X', 'X', 'X'}};

// Char array for RAM addresses
char addressBoard [NUM_ROWS][NUM_COLS] =
          {{0b00000000, 0b00000111, 0b00001110, 0b00010101, 0b00011100, 0b00100011,
0b00101010},
           {0b00000001, 0b00001000, 0b00001111, 0b00010110, 0b00011101, 0b00100100,
0b00101011},
```

```
        {0b00000010, 0b00001001, 0b00010000, 0b00010111, 0b00011110, 0b00100101,
0b00101100},
        {0b00000011, 0b00001010, 0b00010001, 0b00011000, 0b00011111, 0b00100110,
0b00101101},
        {0b00000100, 0b00001011, 0b00010010, 0b00011001, 0b00100000, 0b00100111,
0b00101110},
        {0b00000101, 0b00001100, 0b00010011, 0b00011010, 0b00100001, 0b00101000,
0b00101111},
        {0b00000110, 0b00001101, 0b00010100, 0b00011011, 0b00100010, 0b00101001,
0b00110000}};

// Define constants
char off  = 0b00000000;
char red  = 0b01000000;
char blue = 0b10000000;


bool hasWon = false;
bool blueTurn = true;
bool placedToken = false;
bool animate = false;

int col = 0;
char* colChar = "0";
char* colTxt  = " Col: ";
char* turnTxt = " Blue Turn! ";
char* blueTurnTxt = " Blue Turn! ";
char* redTurnTxt  = " Red Turn! ";

////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////  HELPER FUNCTIONS  //////////////////////////////
////////////////////////////////////////////////////////////////////////////////

// Print the state of the board
void printBoard() {
        int row, col;

        printf("  ");
        for(col = 0; col <NUM_COLS; col++){
```

```
                printf("%d ", col);
        }
        printf("\n");
        for (row = NUM_ROWS-1; row >= 0; row--){
                printf("%d ", row);
                for (col = 0; col < NUM_COLS; col++) {
                        printf("%c ", board[row][col]);
                }
                printf("\n");
        }
}

// Display text on the LCD
void dispText(void){
        if (col == 0) colChar = "O";
        else          colChar = i_to_a(col);
        lcdClear();
        lcdPrintString(turnTxt);
        lcdPrintString(colTxt);
        lcdPrintString(colChar);
        delayMillis(DELAY);
}

// Switch text to display on turn switches
void switchTurnTxt(void){
        if (blueTurn) turnTxt = blueTurnTxt;
        else          turnTxt = redTurnTxt;
}

// Switch turns
void switchTurn(void){
        if (blueTurn)   blueTurn = false;
   else           blueTurn = true;
        switchTurnTxt();
}

// Determine what row token will land in
int getDropHeight(void){
        int row = 0;
```

```
        int i;
        for (i = 0; i < 7; i++){
                if (board[i][col] == 'X') return row;
                else row++;
        }
        return row;
}

// Return char at given index
char boardIndexLookup(int idx){
      int row = idx % NUM_ROWS;
      int col = idx / NUM_ROWS;
      return board[row][col];
}

// Check if four chars at four indeces are the same
bool checkFour(int a, int b, int c, int d){
        char first, second, third, fourth;
        first = boardIndexLookup(a);
        second = boardIndexLookup(b);
        third = boardIndexLookup(c);
        fourth = boardIndexLookup(d);
        if  (first != 'X' && first == second && second == third && third == fourth)  return true;
      else return false;
}

// Check for win in collumns
bool verticalCheck(){
      int row, col, idx;
      for (row = 0; row < NUM_ROWS; row++){
            for (col = 0; col < 4; col++){
                idx = NUM_ROWS * row + col;
                if (checkFour(idx, idx + 1, idx + 2, idx + 3)) return true;
                printf("checking indexes %d ", idx," %d", idx + NUM_COLS, " %d", idx +
NUM_COLS*2, " %d", idx + NUM_COLS*3);
            }
                printf("\n");
      }
        return false;
```

```
}

// Check for win in rows
bool horizontalCheck(){
        int row, col, idx;
    int midpoint = 3;
    for (col = 0; col < NUM_ROWS; col++){
        for (row = 0; row < 4; row++){
            idx = NUM_ROWS * col + row;
            if (checkFour(idx, idx + NUM_COLS, idx + NUM_COLS*2,
idx+NUM_COLS*3)) return true;
            printf("checking indexes %d ", idx," %d", idx + NUM_COLS, " %d", idx +
NUM_COLS*2, " %d", idx + NUM_COLS*3);
        }
        printf("\n");
    }
    return false;
}


// Check for win in diagonals
bool diagonalCheck(){
    int row, col, idx;
    for (row = 0; row < 6; row++){
        for (col = 0; col < 4; col++){
            idx = NUM_ROWS * col + row;
                if (row < 3) if(checkFour(idx, idx + 8, idx + 8*2, idx + 8*3)) {
                    printf("checking index %d ", idx);
                printf(" %d", idx + 8);
                printf(" %d", idx + 8*2);
                printf(" %d \n", idx + 8*3);
                    return true;
                    }
                if (row == 3){
                if (checkFour(idx, idx + 8, idx + 8*2, idx + 8*3)){
                    printf("checking index %d ", idx);
            printf(" %d", idx + 8);
            printf(" %d", idx + 8*2);
            printf(" %d \n", idx + 8*3);
```

```
                                        return true;
                                        }
                                        if (checkFour(idx, idx + 6, idx + 6*2, idx + 6*3)) return true;
//Diag down
                                }
                                if (row > 3) if (checkFour(idx, idx + 6, idx + 6*2, idx + 6*3)) return true;
                }
                printf("\n");
        }
        return false;
}


// Credit for win detection to
codereview.stackexchange.com/questions/27446/connect-four-game
bool checkWin(void){
        if (diagonalCheck() || verticalCheck() || horizontalCheck()) return true;
        else return false;
}

// Return char at row, col in game board
char lookupAddress(int row){
        return addressBoard[row][col];
}

// Return char at top row of given col
char lookupTopAddress(void){
        return addressBoard[6][col];
}

// Return color of current player
char assignColor(void){
        char color;
        if (blueTurn) color = blue;
        else          color = red;
        return color;
}

// Generate char for right RAM address and with the correct color
```

```
char generateTokenSPIsig(int row){
        char addr  = lookupAddress(row);
        char color = assignColor();
        return addr|color;
}

// Put correct char in game board based on turn
void updateBoard(int row){
      char token;
      if (blueTurn) token = 'B';
      else        token = 'R';
      board[row][col]    = token;
      if( checkWin()) hasWon = true;
      else          hasWon = false;
}

// Generate and send SPI signals to simulate dropping token
void animateDrop(int row){
        int i;
        char dropSPIsig;
        char color = assignColor();

        for (i = NUM_ROWS; i > row; i--){
                spiSendReceive(addressBoard[0][0] | ledOneColor);
                delayMillis(5);
                spiSendReceive(addressBoard[1][0] | ledTwoColor);
                delayMillis(5);

                dropSPIsig = addressBoard[i][col] | color;
                spiSendReceive(dropSPIsig);
                delayMillis(20);

                dropSPIsig = addressBoard[i][col] | off;
                spiSendReceive(dropSPIsig);
                delayMillis(10);
        }
}

// Fill board with specific color, or turn all off
```

```
void floodBoard(char color){
        char spiSig;
      int row, col;
      int addr = 0;

      for (col = 0; col  < NUM_COLS; col++){
            for (row = 0; row < NUM_ROWS; row++){
                                          if (color == off) board[row][col] = 'X';
                  spiSig = addressBoard[row][col] | color;
               spiSendReceive(spiSig);
                                          delayMillis(50);
            }
      }
}

////////////////////////////////////////////////////////////////////////////////
//////////////////////////////   MAIN GAMEPLAY FUNCTIONS   ////////////////////////////
////////////////////////////////////////////////////////////////////////////////
char changeRowCol(void) {
 // Enter user interaction while loop
 while(1) {

          // Decrement collumn button
          if (digitalRead(LEFT)) {
      if (col == 0) col = 0;
                  else        col = col - 1;
                  dispText();
          }

    // Increment collumn button
          else if (digitalRead(RIGHT)) {
      if (col == 6) col = 6;
                  else        col = col + 1;
                  dispText();
          }

          // Confirm drop button
          else if (digitalRead(CONFIRM) & getDropHeight() < NUM_ROWS){
                  lcdClear();
```

```
                lcdPrintString("Dropping token");

                int row = getDropHeight();

                char tokenSPIsig = generateTokenSPIsig(row);

                animateDrop(row);

                spiSendReceive(tokenSPIsig);

                // For Debugging
                printf("row %d",row);
                printf(" col %d\n",col);
                updateBoard(row);

                delayMillis(1000);

                // Break loop if someone wins
                if (hasWon == true) return;

                // Otherwise switch turns
                switchTurn();

                printBoard();
                printf("\n");

                dispText();
        }
 }
}

void playGame(void){
        // Reset game state
        hasWon = false;

         // Initialize LCD screen
        lcdInit();
        lcdPrintString("Start game!");
   changeRowCol();
```

```
   floodBoard(assignColor());
}

////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////     TESTING      ////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
void boardTestComplete(void){

        char spiSig;
        int row, col;
        int addr = 0;

        for (col = 0; col  < NUM_COLS; col++){
                for (row = 0; row < NUM_ROWS; row++){
                        spiSig = addressBoard[row][col] | off;
                        spiSendReceive(spiSig);
                        printf("lighting up led %d", addr);
                        printf(" in pos row %d", row);
                        printf(" col %d", col);
                        printf(" by sending %c\n", spiSig);
                        delayMillis(10);
                        addr++;
                }
        }
        printf("done filling ram w zeros \n");
}

void boardTestSingle(void){
        char spiSig;
        spiSig = addressBoard[0][0] | red;
        spiSendReceive(spiSig);
        printf("sending spi to row 0 col 0 \n ");

        delayMillis(5000);

        spiSig = sunkenAddr | red;
        spiSendReceive(spiSig);
        printf("sending spi to sunken addr \n");
        delayMillis(5000);
```

```
        spiSig = addressBoard[1][0] | blue;
    spiSendReceive(spiSig);
    printf("sending spi to row 0 col 0 \n ");

    delayMillis(5000);

    spiSig = sunkenAddr | blue;
    spiSendReceive(spiSig);
    printf("sending spi to sunken addr \n");
    delayMillis(5000);
}

////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////  MAIN  ////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
void  main(void) {
        pioInit();
        arrowKeyInit();
        spiInit(240000,0);
        playGame();
}
```

APPENDIX H: lcdDisp.h

```
// Final Project: MicroPs Fall 2017
// 3D Connect Four on 7*7*7 LED cube
// LCD Display Module
// Shiv Seetharaman: sseetharaman@hmc.edu
// Nov 4 2017

#include "EasyPIO.h"
#include "stdio.h"
#include "string.h"

int LCD_IO_Pins[] = {14,15,4,17,22,24,20,21};

typedef enum {INSTR, DATA} mode;
#define RS 16
#define RW 12
#define E 25
#define LEFT 6
#define RIGHT 5
#define CONFIRM 13

// Convert int to char pointer
// Credits: Srikant Aggarwal: (careercup.com/question?id=2794)
int no_of_digits(int num) {
 int digit_count =  0;

 while (num > 0) {
  digit_count++;
  num /= 10;
 }
 return digit_count;
}

char* i_to_a(int num) {
 char* str;
 int digit_count = 0;
```

```
 if (num < 0) {
  num = -1*num;
  digit_count++;
 }

 digit_count += no_of_digits(num);
 str = (char *)malloc(sizeof(char)*(digit_count+1));
 str[digit_count] = '\0';

 while (num > 0) {
  str[digit_count-1] = num%10 + '0';
  num = num/10;
  digit_count--;
 }

 if(digit_count == 1)
  str[0] = '-';

 return str;
}


char lcdRead(mode md) {
  char c;
  pinsMode(LCD_IO_Pins, 8, INPUT);
  digitalWrite(RS, (md == DATA)); // Set instr/data mode
  digitalWrite(RW, 1);            // Read mode
  digitalWrite(E, 1);             // Pulse enable
  delayMicros(10);                // Wait for LCD Response
  c = digitalReads(LCD_IO_Pins, 8); // Read a byte from parallel port
  digitalWrite(E, 0);             // Turn off enable
  delayMicros(10);
  return c;
}

void lcdBusyWait(void) {
  char state;
  do {
    state = lcdRead(INSTR);
```

```
  }while (state & 0x80);
}

void lcdWrite(char val, mode md) {
  pinsMode(LCD_IO_Pins, 8, OUTPUT);
  digitalWrite(RS, (md == DATA));    // Set instr/data mode.OUTPUT=1,INPUT=0
  digitalWrite(RW, 0);
  digitalWrites(LCD_IO_Pins, 8, val);// Write the char to the parallel port
  digitalWrite(E, 1);               // Pulse E
  delayMicros(10);
  digitalWrite(E, 0);
  delayMicros(10);
}

void lcdClear(void) {
  lcdWrite(0x01, INSTR);
  delayMicros(1530);
}

void lcdPrintString(char* str) {
  while (*str != 0) {
    lcdWrite(*str, DATA);
    lcdBusyWait();
    str++;
  }
}

void lcdInit(void) {
  pinMode(RS, OUTPUT);
  pinMode(RW, OUTPUT);
  pinMode(E, OUTPUT);
  // send init routine
  delayMicros(15000);
  lcdWrite(0x30, INSTR);
  delayMicros(4100);
  lcdWrite(0x30, INSTR);
  delayMicros(100);
  lcdWrite(0x30, INSTR);
  lcdBusyWait();
```

```
  lcdWrite(0x3C, INSTR);
  lcdBusyWait();
  lcdWrite(0x08, INSTR);
  lcdBusyWait();
  lcdClear();
  lcdWrite(0x06, INSTR);
  lcdBusyWait();
  lcdWrite(0x0C, INSTR);
  lcdBusyWait();
}

void arrowKeyInit(void) {
  pinMode(LEFT, INPUT);
  pinMode(RIGHT, INPUT);
  pinMode(CONFIRM, INPUT);
}
```