Multi-Device Connect 4

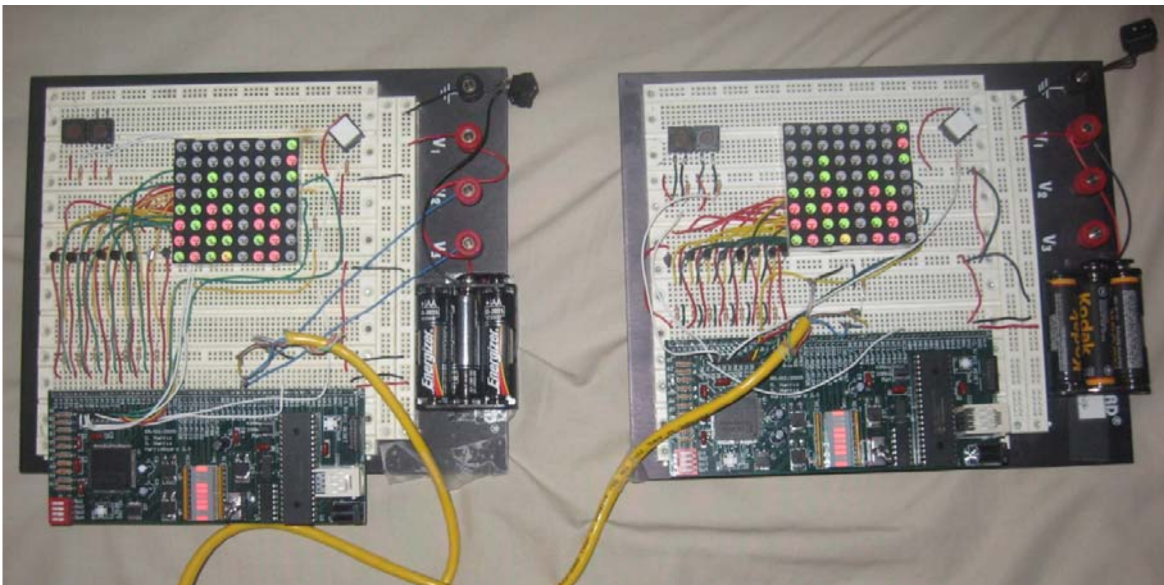
Final Report

12/11/2009

E155: Microprocessor-Based Systems

Professor Harris

Narayan Propato and Jason Kang



Abstract:

The goal of this project is to create a pair of identical handheld gaming devices that allow two players to play Connect 4 against each other. Players take turns dropping chips into a six-by-seven game board using two directional buttons and a select button. The game board is displayed on an eight-by-eight bi-color red/green dot-matrix LED display. The first player to connect four chips in a line is victorious. If no player is able to achieve this when the game board is full, the game declares a tie. The device uses a PIC to control game flow and inter-device communication and an FPGA to control the LED display. The two prototype devices are able to carry out multiple Connect 4 games, albeit with disconnections occurring approximately every twenty matches.

Introduction

The goal of this project is to create a pair of handheld gaming devices that allows two players to play six-by-seven Connect 4 against each other. Both devices are identical in both hardware and software such that if additional prototypes are created any combination of them will successfully connect. The user interface consists of two directional buttons (left and right) and a select button. The game board is displayed on an eight-by-eight bi-color red/green dot matrix LED display (Figure 1). Once the two devices are connected, the first player to press select is allowed to make the starting move. Additionally this player is assigned the color red, and his opponent is assigned green. When one player wins the match, a special animation displays the winning combination. A new match begins when both players have pressed select, with the green player now making the first move. The device uses a PIC to control game flow and inter-device communication and an FPGA to control the LED display (Figure 2).

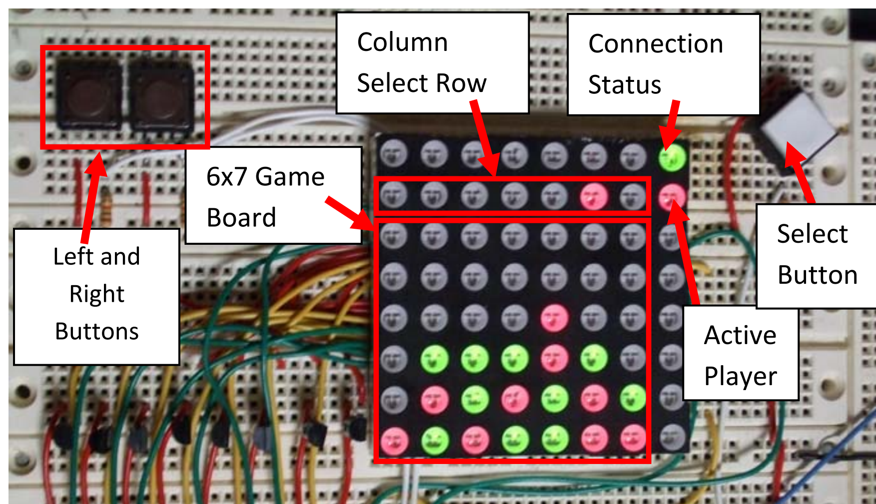


Figure 1: Connect 4 inputs and outputs. Inputs consist of two directional buttons and a select button. The output is an 8x8 dot matrix LED display.

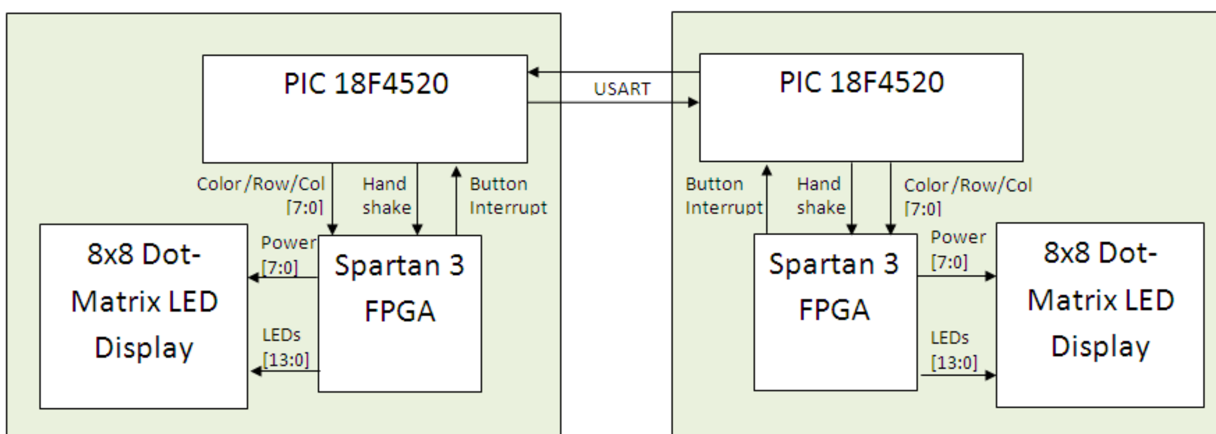


Figure 2: Overview schematic showing connection between devices, and the interaction between the three main pieces of hardware (PIC, FPGA, and LED display)

New Hardware: 8x8 Dot Matrix LED Display

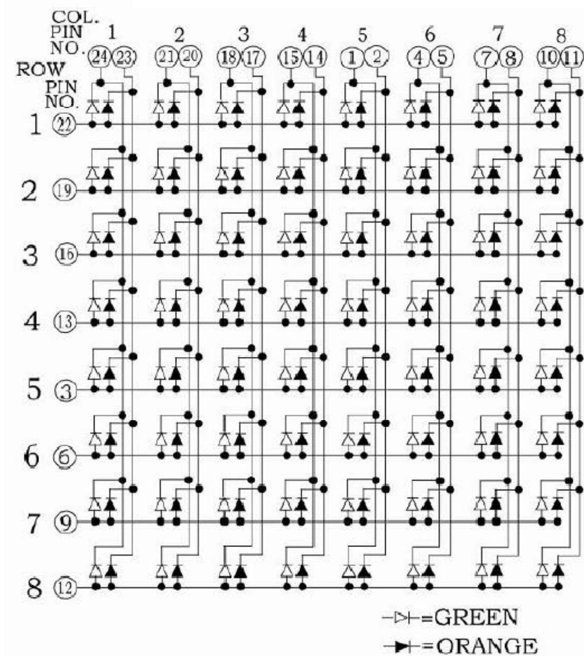


Figure 3: Internal Circuitry for 8x8 bi-color LED Display. Obtained from data sheet [1].

This display works similarly to the matrix keypad used in lab 3 except it is used as an output instead of an input (Figure 3). To light up an individual LED the correct anode (row) must be powered and the correct cathode (column) must be grounded. Time multiplexing is used to light up LEDs in multiple rows. One must scan power through the row pins while outputting the logic 0s to the LEDs that should be turned on in the powered row. This organization allows many LEDs to be controlled by relatively few pins.

Breadboard Schematic

The game device displays information through an 8x8 dot matrix LED display. Each dot on the display contains two LEDs, a green one and a red one. Therefore the display needs eight common anode control wires, eight green cathode wires, and eight red cathode wires (Figure 4). The desired color is chosen by grounding the appropriate cathode and powering the appropriate anode. In addition, a yellow color can be produced by turning on both the red and yellow LEDs. In order to ensure that all of the LEDs are supplied with enough current, the anodes are tied to the 3.3V source through a 2N3960 PNP transistor. Since the red LEDs in the display run on a lower voltage than the green ones, the red LEDs have a tendency to overpower the green LEDs. Therefore, 390Ω resistors are added to the red cathodes in order to even out the brightness.

To enable communication between two devices the Rx and Tx pins of the Harris Board must be connected to the Tx and Rx pins of another device's Harris Board respectively. In addition the two

devices must share a ground over the same cable so that the data is received using the same voltage reference. Additionally the Rx pins are pulled up with a 1kΩ resistor. This causes the Rx pin to remain in an idle high state even if the UART connection is dropped.

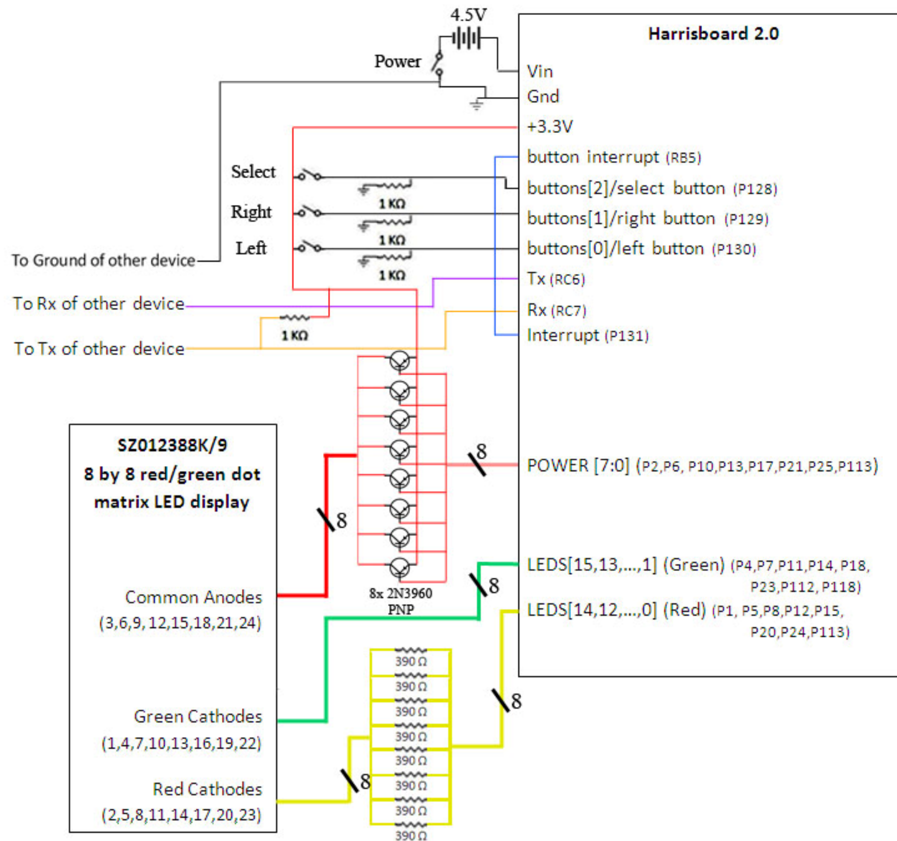


Figure 4: Breadboard Schematic. Transistor, resistor, and switch pictures taken from The Electronics Club [2].

Three user input push button switches are mounted on the board. They are all active high and are pulled to ground using 1kΩ resistors. The board is powered with 4.5 V from three 1.5V AA batteries. The Harris board's voltage regulators are used to drop that voltage down to 3.3V, which is outputted from the Harris board to power the LED display and provide the button voltage. A power switch was added to turn the device on and off.

PIC

The PIC has six major roles:

1. Inter-Game Initialization
2. User Input

3. Inter-PIC Communication
4. Game Mechanics
5. Victory Conditions
6. Display

Their relationship is shown in Figure 5.

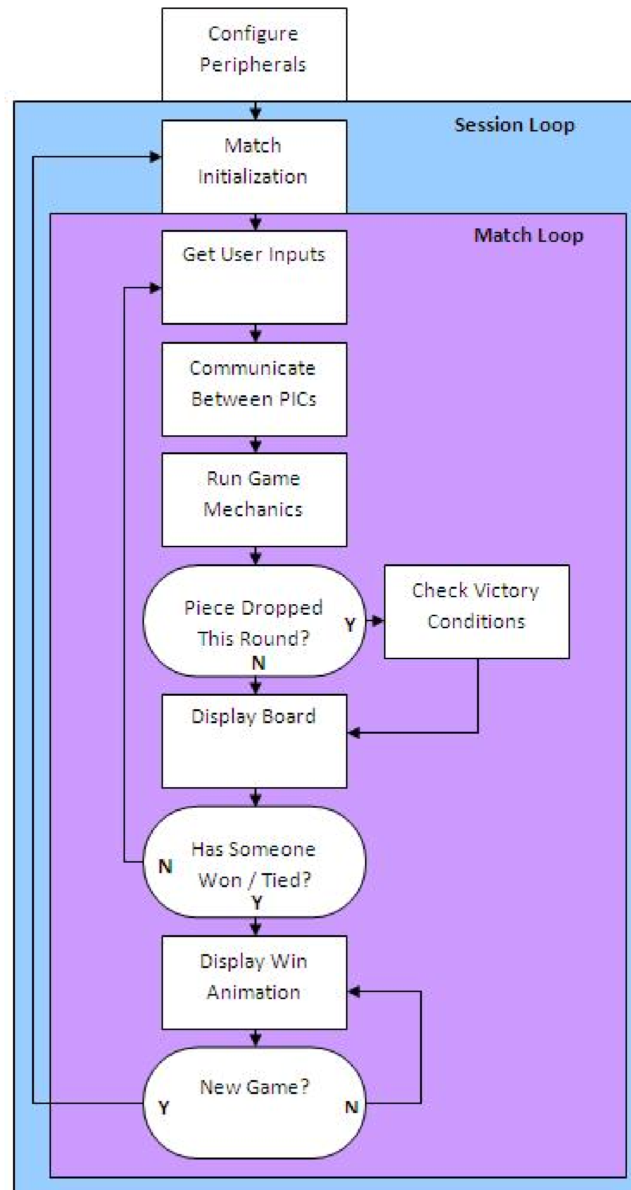


Figure 5: Flowchart of PIC code. Match loop cycles play continuously regardless of user inputs.

The PIC carries out its roles through two nested loops. The outer loop runs the entire session (which consists of multiple matches of Connect 4). The first time the session loop is called it initializes communication with the opponent PIC and assigns turn order and colors. This process is described in more detail in the Inter-PIC Game Initialization section. All successive times that it is called the session loop simply switches turn order and instantiates variables to what they should be at the start of the game.

The inner loop, the match loop, manages the current match. At the beginning of a loop the PIC parses the user inputs (User Input). Then the PIC attempts to communicate with the opponent's PIC (Inter-PIC Communication). It sends player data and waits to receive and parse the opponent's data. Now that the PIC has data from both the player and the opponent it proceeds to carry out the active player's move if a move has been made (Game Mechanics). It then checks to see if the new move causes a victory or tie (Victory Conditions). Finally, the PIC informs the FPGA how to update the display (Display) before starting the next cycle of the loop.

The two devices communicate using the UART module built into the PICs. The connection operates at a baud rate of 115.2k. This connection allows the players to communicate their moves to one another and is also used to determine player colors and turn order.

Additional inputs come from the FPGA including denounced button signals (left, right, and select) into Port C, and a button interrupt signal is wired to Port RB5 which takes advantage of that port's interrupt on change feature. This is described further in the PIC-User Input section.

The PIC also outputs information to the FPGA including eight data bits transmitted on Port D over parallel to the FPGA. These bits tell the FPGA which spaces on the LED display should be altered and to what color. Additionally, the PIC outputs a handshaking bit so that the FPGA knows when the parallel data can be read. This is described further in the PIC-Display section.

PIC - Match Initialization:

At the start of each new game the program needs to initialize variables including making a clear game board, clearing the previous winner, and assigning turn order.

The first time this module is used in a session the PIC devices need to establish a connection and assign colors and a turn order to each player. A connection status light on the top right of the display will be yellow to show that a connection has not been established yet. When either player presses their select button the two PICs attempt to synchronize. The PIC which had its select button pressed sends an op code (0b10100000), showing that it wants to start a connection, out on the transmit line of its UART. If the other PIC is on and able to recognize the op code it replies with another op code (0b01010000), showing that it is also ready to initialize a connection. When the first PIC receives this op code then the status light turns green showing that a connection has been established. Additionally the player who pressed select is assigned the color green and is the first player for the first match. The other player is red and the second player.

The program returns to this module when the players decide to start a new match in a session. This occurs when both players press select at the win screen. Upon returning to this module the turn order is switched while keeping player colors.

PIC-User Input:

The PIC samples the debounced button inputs every time the interrupt signal is toggled by the FPGA. This was done by connecting the interrupt signal to Port B and turning on the Port B interrupt, which triggers whenever a Port B input is toggled. The input values recorded when the interrupt is triggered is then processed by the dataSetup function, such that the variables “left,” “right,” and “select” are assigned their corresponding values. These variables can then be used whenever they are needed by other functions.

Inter-PIC Communication:

Once the game is in progress the UART connection is used to share the players’ moves. The most significant bit is a valid bit. The PICs send data on every cycle of the match loop regardless of whether new user input has been observed. If the players have not yet made a move then the valid bit will be cleared so that sent data will be ignored. When the player chooses a column to drop a piece into, the valid bit is set high and the selected column is sent over UART in the three least significant bits. The remaining bits are used to check the quality of the signal. If bits 7-3 are not 0b1010 then the data packet is considered to be corrupted and is therefore ignored.

This module also deals with synchronizing match loops between the two devices and detecting lost connections. A delay loop is built into this module. The PIC breaks out of this delay loop when the UART receive interrupt flag is set. This way if one PIC is running faster it will first send out its data then wait in the loop to receive data, such that both devices are always processing the same match loop and consequently stay synchronized. When the connection is lost the delay loop will time out causing the connection status light to become red. When this happens the PICs will continue to attempt to communicate so that the game can resume when the connection is re-established.

PIC - Game Mechanics:

The game mechanics module of the PIC processes the changes made to the game board as defined by the user input. This module stores a virtual copy of the Connect 4 board in the form of an 8x8 array of characters. The value of each character shows what color should be displayed in that spot (0b00=none, 0b01=green, 0b10=red).

The light in the second row from the top and the right-most column shows the color of the active player. A cursor is shown on the active player’s display indicating which column of the game board the piece will be dropped into. Pressing the left button moves this cursor left and pressing the right button moves it right. If the select button is pressed this module causes a piece to fall into the selected column. Also, this event causes turns to change.

PIC - Victory Conditions:

The game uses the endgameModule function to check the victory conditions every time a new chip is placed on the board. It does this by checking whether there is a four-space-long (or longer) line of chips that includes the most-recently placed chip. Additionally, it checks for winning combinations in all

four directions (horizontal, vertical, and the two diagonals), so if multiple combinations are created simultaneously all will be detected. endgameModule also keeps track of which chips are part of a winning combination, so that they can then be highlighted by the endAnimationModule. Finally, if victory conditions aren't met, endgameModule checks the top row of the board. If it is full, then a tie has occurred and a special animation is played.

PIC – Display:

Based on information from the game mechanics module, the display module of the PIC informs the FPGA what color each pixel on the LED display should. The PIC packages the color, row, and column into 8 bits and outputs this word in parallel to the FPGA through PORTD. Then it toggles a hand shaking signal on and off to inform the FPGA that it can read the data pins. Since the instruction clock is $1/4^{th}$ the speed of the board clock, the FPGA is able to consistently detect the change.

FPGA

The FPGA is responsible for debouncing user input, providing an interrupt signal to the PIC whenever a button press is detected, and lighting up the dot matrix LED display with the contents of the game board matrix.

It takes inputs from the left, right, and select buttons. Additionally the FPGA takes in data from the PIC telling it what to display and when it can read this new data. The FPGA has 24 outputs to control the dot-matrix LED display. Eight of them control the common anodes, eight control the green cathodes, and eight control the red cathodes. In addition, the FPGA outputs the debounced buttons to the PIC and toggles an interrupt output to tell the PIC when new user input data is available.

FPGA - User Input:

The FPGA debounces all three buttons by sampling them every 2000 clock cycles of a 20 Mhz clock. Additionally, it provides the PIC with an interrupt signal which is toggled on the rising edge of any of the three button inputs. The FPGA also masks the debounced button inputs such that only those buttons that have undergone a rising edge are held high; buttons that were already high the previous time they were sampled are transmitted as zeroes to the PIC.

FPGA - Display:

The FPGA is responsible for reflecting the matrix stored in the PIC on the dot matrix LED display at the speed that is required. Matrix information consisting of the colors that should be displayed for each element is stored in a 16 by 8 memory block. This module is time-multiplexed to light up individual rows in the dot matrix LED display at a time. When the FPGA detects the handshaking signal, it samples the data transmitted by the PIC and spends one clock cycle updating the contents of the appropriate row in the matrix. Consequently the FPGA needs only be informed of any changes to the matrix, for it will continue to display the old values otherwise. The FPGA does not read from a row while it is updating

to prevent incorrect information from being displayed on the dot matrix LED display. Additionally, the FPGA will clear the contents of its memory block when the reset signal is held high.

Results:

The two prototype devices function as desired. Users are able to connect the two devices and play an unlimited number of Connect Four matches (as long as the devices are powered, of course). The connection between the two devices is fairly noisy, however, and therefore there is a small chance that a spike of noise will produce faulty information to be received by one of the FPGAs. While the use of three-bit op codes has drastically reduced the frequency of these occurrences, they do still happen approximately every twenty games.

The most difficult part of the design was getting the communication to work reliably. If movement caused the Tx and Rx UART pins to disconnect the receive interrupt flag stopped triggering. A lot of this difficulty was due to the fact that if breakpoints are used in debug mode the connection is dropped since the PIC can no longer send data at regular intervals. This causes the same problem that we were trying to solve in the first place. Eventually we learned of two ways to fix this problem. One is placing a pull-up resistor on the Rx pin so that when the Tx and Rx pins disconnect the Rx pin stays high. Since the UART is idle high, the UART will see an idle signal on the Rx pin instead of a start bit. The second solution was clearing the UART overrun flag. Originally we ignored the fact that this was triggered believing that the UART would just ignore that data if the flag was set (similar to the framing error flag). We discovered, however, that if the overrun flag is set the receive interrupt will not be set when data is received.

The only major difference between the initial and final proposal (besides making the writing clearer) was adding the goal of a status light for the connection. We realized that both debugging and playing the game would be frustrating without it.

References:

Datasheets

[1] Wuxi Ark Technology Electronic Co. LTD. SZ012388K/9 Datasheet. http://www.seeedstudio.com/depot/datasheet/SZ012388K9.pdf

Images:

Images of resistors, switches, and transistors obtained from

[2] Hewes, John. Circuit Symbols of Electronic Components. 2009 http://www.kpsec.freeuk.com/symbol.htm

APPENDIX A: PIC C Code

```c
//Created By: Jason Kang (jkang@hmc.edu) and Narayan Propato (npropato@hmc.edu)
//Date: 11/25/2009
//Function:
//  This is code for the 18f4520 PIC microcontroler in a multiplayer
// connect 4 handheld device. It deals with communication over the USART
// to an opponent's device. It also has functions to accept button press
// inputs, and to communicate with an FPGA that handles display to an
// 8x8 bi-color dot-matrix LED display.


#include <p18f4520.h>
#include <stdio.h>
#include <usart.h>

// Function prototypes
void isr(void);                          //interrupt handler
void config(void);                       //Periferal configuration.
void coltrans(void);                     //Transmit column over USART
char findRow(char);                      //find row that that piece ends in
char dropAttempt(char, char);            //attempt to drop a piece
void updateDisplay(char, char, char);    //show a piece in LED display
void initialization(void);               //initializes a new game
void dataSetup(void);                    //parses user input. creates packet
void communicationModule(void);          //trades data between PICs
void gameplayModule(void);               //uses user & opp inputs to move
void endgameModule(void);                //checks victory conditions
void displayModule(void);                //updates the whole LED display
void endAnimationModule(void);           //win screen animation
void main(void);

// Definitions
char MIN_ROW = 2;                        //play game on rows 2-7
char MAX_COL = 7;                        //play game on columns 0-7
char DROP_ROW = 7;                       //The row that holds the piece to drop

//Global Variables
char fail;                               //Is 1 if the drop attempt was invalid
char playerReady, oppReady;              //Are the players ready to start game
char turnOrder;                          //First player(0) or Second Player(-1)
char newGame = 0;                        //Do players whant to play again
char lastColor = 0;                      //last color to make a move
char totalWinningChips = 0;              //number of chips that cause victory
char winningChips[22][2];                //Stores location of winning chips (22 max)
char victory = 0;                        //Has someone won the game?
char tie = 0;                            //Have the players tied?
unsigned int t=0;                        //an timing variable
char sync = 0;                           //Are the PICs connected and synced.
char colHighlight;                       //Column currently being highlighted
char rowPlaced;                          //Row that piece ends up in
char playerColor, oppColor;              //the color assignments to each player
char turn = 0;                           //current turn: 0,my turn; -1,opp. turn
char left = 0, right = 0;                //Are directional btns being pressed
char playerSelect = 0, oppSelect = 0;    //Has either player pressed select
char buttons;                            //Buttons that are currently pressed
char playerMove, oppMove;                //Column selected by each player
char playerData=0, oppData=0;            //players' column
char board[8][8];                        //the current state of the board

// When an interrupt happens, jump to the timer_isr function
#pragma code high_vector = 0x08
void high_interrupt(void){
    _asm GOTO isr _endasm
}
#pragma code

#pragma interrupt isr
// Function runs when an interrupt is triggered
```

```
void isr(void){
    char maxcol;
    char fake;
    if (INTCONbits.RBIF == 1) {          // whan a button is pressed
        buttons = PORTC;
        buttons = buttons | 0b00000100;
        fake = PORTB;
        INTCONbits.RBIF = 0;             //clear interrupt flag
    }
}

//main/////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////

void main(void){
    //wait a while for the FPGA to get programmed. This is necessary if the game
    //is run on batteries
    for (t = 500000;t>0;) {
        t = t + 1;
    }
    //configure nessary PIC periferals
    config();
    while(1){
        initialization();                //initialize the game at the start of each match
        while(newGame == 0){             //while a match is currently running
            dataSetup();                 //parse user inputs and prepare USART data packet
            communicationModule();       //Communicate with opponent
            if (sync == 1) {             //if the connection is maintained
                gameplayModule();        //run the turn of the game
                if ((playerSelect != 0 || oppSelect != 0) && fail == 0) {
                    endgameModule();     //check victory conditions
                }
                displayModule();         //Update LED display
            }
            if (victory || tie) {        //if someone wins
                while(newGame == 0){
                    dataSetup();             //maintain communication
                    communicationModule();
                    endAnimationModule();    //while displaying animation
                }
            }

        }
    }
}
//config//////////////////////////////////////////////////////////////////////////

void config(void){
    // Interrupt configuration
    // Enable global interrupts and PortB interrupts, and clear PortB interrupt flag
    INTCON = 0b11001000;
    PIE1bits.RCIE = 0;       // Enable USART reciever interrupt
    PIR1bits.RCIF = 0;       // Clear USART reciever interrupt flag


    // USART configuration
    TRISC=0b10000000;        // Sets RX to input and TX to output
    TXSTA=0b00100100;        // Enables TX and sets to high speed.
    RCSTA=0b10010000;        // Enables serial ports and RX.
    SPBRG=10;                // Sets baud rate to approximately 115.2k.

    //make port D output ports (for the parallel comm to FPGA)
    PORTD=0b00000000;        //set port D to a known value
    TRISD=0b00000000;

    //make RE2 an output port (for telling FPGA about change)
```

```
    PORTEbits.RE2=0;        //set RE2 to a known value
    TRISEbits.TRISE2=0;

    //Set up button interrupts
    TRISBbits.TRISB5=1;

    //Set up button inputs
    TRISC=0b10111000;       // Sets RX to input and TX to output

    //Reset the dot matrix display
    TRISEbits.TRISE1=0;
    PORTEbits.RE1=1;        //set RE1
    PORTEbits.RE1=0;        //clear RE1
}



//initialization////////////////////////////////////////////////////////////////////

//  Initializes game by establishing player turn order and color. This function handles
// two cases the first is when the game is started for the first time. In this case the
// first person to press select is assigned to be the first player and assigned the color
// green. The other player is second and is assigned red.
//  The second case deals with games that are not the first in a set. In this case turn
// order is switched but color is not.


void initialization(void){
    char i, j;          //initialize some counter variables

    //clear board
    for (i=0; i<8; i++){
        for (j=0; j<8; j++){
            board[i][j]=0;
        }
    }

    colHighlight = 3;   //initialize the starting column to the third
    playerReady = 0;    //clear the flags that would start a new game
    oppReady = 0;

    //Goes through this loop if it is the first game in a session
    if (newGame == 0) {
        board[0][7] = 3;                    //display a yellow status light that
        updateDisplay(board[0][7],0,7);     //shows that connection is not yet made

        //wait here until connection with other PIC has been made
        while(sync == 0) {
            oppData = ReadUSART();                  //read USART to check for sync. code
            INTCONbits.RBIE = 0;
            playerSelect = buttons & 0b00001000;    //is player's select pressed?
            buttons = 0;
            INTCONbits.RBIE = 1;
            //if the opponent is sending the initialization code
            if (oppData == 0b10100000) {
                //this PIC starts as the second player
                turn = -1;
                playerColor=2;                  //...and the red player
                oppColor=1;                     //...the opponent is green
                turnOrder = -1;                 // and this keeps the starting turn
                board[1][7] = oppColor;         // show that it starts as the opp. turn
                printf("%c", 0b01010000);       // send an acknowledge string
                sync = 1;                       // now the PICs are synchronized
            }
            //  if the opponent has not sent the initialization code and
            // the player has pressed select...
            else if (playerSelect != 0){
```

```
                //send initialization code
                printf("%c", 0b10100000);
                //wait for the acknowledge string from the opponent
                //do nothing if not acknopwledged
                for (t = 50; t > 0;) {
                    oppData = ReadUSART();
                    t = t - 1;
                    // if the initialization is acknowledged
                    if (oppData == 0b01010000) {
                        turn = 0;                           //you are the the 1st player
                        playerColor=1;                      //and are assigned green
                        oppColor=2;                         //the opponent is red
                        turnOrder = 0;
                        board[1][colHighlight] = playerColor;   //column select display
                        board[1][7] = playerColor;              //show your turn
                        sync = 1;                               //PICs are syhnchonized
                        break;
                    }
                }
            }
        }
    }
    else {
        //this code is used when the game is not the first in the session
        //clear newGame flag that showed that the players wanted to start a new game
        newGame = 0;
        buttons = 0;
        //switch the starting player
        if (turnOrder == 0) {                       //if you were the first player
            turn = -1;                              //you are now the second player
            board[1][7] = oppColor;                 //show opponent's turn
            victory = 0;                            //delete last game info.
            tie = 0;
            sync = 1;
        }
        else {                                      //if you were the second player
            turn = 0;                               //you now become the first player
            board[1][colHighlight] = playerColor;   //activate column select display
            board[1][7] = playerColor;              //show that it is your turn
            victory = 0;                            //adelete last game info.
            tie = 0;
            sync = 1;
        }
        //show that the turn order has been switched in this turn
        turnOrder = ~turnOrder;
    }
}

//dataSetup//////////////////////////////////////////////////////////////////////////

// Module responsible for decoding player input and readying inter-PIC data.

void dataSetup(void){
    //disable the input button interrupt so that the buttons don't change
    //while processing
    INTCONbits.RBIE = 0;
    left = buttons & 0b00100000;            // get the left button
    right = buttons & 0b00010000;           // get the right button
    playerSelect = buttons & 0b00001000;    // get the select button
    INTCONbits.RBIE = 1;
    //clear buttions so that it doesn't get double read.
    buttons=0;
    //if the player passes select than add the move to the packet to be sent over
    //USART (playerData)
    if (playerSelect != 0) {
        playerMove = colHighlight;
//      playerData = 0b10000000 | playerMove;
```

```
        playerData = 0b11010000 | playerMove;
    }
    //or else send useless data
    else {
        playerData = 0b00000000;
    }
}

//communicationModule/////////////////////////////////////////////////////////////////////

// Module responsible for synchronizing players and transmitting and receiving player
// moves.
void communicationModule(void){
    // send the playerData package to opponent's PIC
    printf("%c", playerData);
    //wait for a little while until data is recieved from the opponent. This loops allows
    // the two PICs to synchronize since it delays the faster one.
    for (t = 1000; t>0;) {
        if (PIR1bits.RCIF == 1){
            break;
        }
        t= t -1;
    }
    //if the connection times out...
    if (t == 0) {
        sync = 0;                           //the boards are not synchronized
        RCSTAbits.CREN = 0;                 //clear the overrun error if there is one
        RCSTAbits.CREN = 1;
        board[0][7] = 2;                    //they are not connected so show red
        updateDisplay(board[0][7],0,7);
    }
    //if the connection does not time out
    else {
        sync = 1;                           //the boards are synchronized
        board[0][7] = 1;                    // show this on the board with a green light
        updateDisplay(board[0][7],0,7);
    }
    oppData = ReadUSART();              // Read opponent data
    if ((oppData & 0b01110000) == 0b01010000) {
`       oppSelect = oppData & 0b10000000;   // Parse it into oppSelect and oppMove
        oppMove = oppData & 0b00000111;
    }
    else {
        oppSelect = 0;                      // Parse it into oppSelect and oppMove
        //oppMove = oppData & 0b00000111;
    }
}


/////////////////////////////////////////////////////////////////////////////////////////

// Module responsible for enforcing game rules.
void gameplayModule(void){
    //  Check if it is this player's turn.
    if (turn == 0){
        //  Check if the player has attempted to make a move, then check whether the move
        // is valid and modify the board if it is. Additionally if it is valid, change
        // turns, remove the cursor, and display the opponent's color to signify it is
        // his turn.
        if (playerSelect != 0) {
            fail = dropAttempt(playerMove, playerColor);
            if (fail == 0) {
                turn = ~turn;
                board[1][colHighlight] = 0;
                board[1][7] = oppColor;
            }
        }
```

```
        //  If the player instead has pressed the left button, and the cursor is not
        // located at the left edge of the screen, remove the previous position of the
        // cursor, shift the position of the cursor one space to the left, and display
        // the new position of the cursor.
        else if ((left != 0) && ((colHighlight & 0b00000111) > 0)) {
            board[1][colHighlight] = 0;
            colHighlight = colHighlight - 1;
            board[1][colHighlight] = playerColor;
        }
        //  If the player instead has pressed the right button, and the cursor is not
        // located at the right edge of the screen, remove the previous position of the
        // cursor, shift the position of the cursor one space to the right, and display
        // the new position of the cursor.
        else if ((right != 0) && ((colHighlight & 0b00000111) < (MAX_COL-1))) {
            board[1][colHighlight] = 0;
            colHighlight = colHighlight + 1;
            board[1][colHighlight] = playerColor;
        }
    }
    //  Check if it is the opponent's turn and the opponent has attempted to make a
    // move. If both are true, check that the move is valid and modify the board if
    // it is. Additionally if it is valid, change turns, add the cursor, and display
    // this player's color to signify it is his turn.
    else if (turn == -1 && oppSelect != 0){
        fail = dropAttempt(oppMove, oppColor);
        if (fail == 0) {
            turn = ~turn;
            board[1][colHighlight] = playerColor;
            board[1][7] = playerColor;
        }
    }
}


//endgameModule//////////////////////////////////////////////////////////////////////////

// Module responsible for determining whether endgame (victory/defeat) conditions
// have been met.
void endgameModule(void){
    char i,j;
    char index = 0;
    char lastMove;
    char topleft, left, bottomleft, bottom, bottomright, right, topright;

    playerSelect = 0;

    // If it is this player's turn, then use his color and check for his pieces.
    if (turn == -1) {
        lastMove = playerMove;
        lastColor = playerColor;
    }
    // If it is the opponent's turn, use his color and check for his pieces.
    else if (turn == 0){
        lastMove = oppMove;
        lastColor = oppColor;
    }


    // Victory condition check.
    //  To check if the victory condition has been met, the program check for four
    // continuous pieces arranged in a horizontal or vertical line or one of
    // the two diagonal lines. More specifically, it checks if there are three
    // pieces total to the topleft/bottomright, left/right, bottomleft/topright,
    // or bottom of the most recently-placed piece. Additionally, the program
    // keeps track of where the winning pieces where located using the matrix
    // winningChips, which  stores the row and column coordinates of each winning
    // piece. If multiple winning combinations are  found (such as simultaneous
```

```
    // horizontal and vertical combinations), winningChips stores chips belonging
    // to all successful combinations, not just the first one found. When the
    // program checks a line and finds that the victory condition has not been
    // met, it restores the indexing of winningChips to what it was before checking
    // said line. This ensures that only successful checks affect the winning
    // chips displayed, since when victory is achieved only the first [index]
    // elements of winningChips are displayed. Only the first check is fully
    // commented; the other three checks follow the same logic as the first.

    //// Topleft/bottomright victory check. ////
    //  Topleft check. //
    // Only checks up to three spaces away from the most recently-placed piece.
    // If the active player doesn't have a piece [topleft] places to the topleft
    // of his/her most recently-placed piece, stop the loop early. Else increase
    // the indexing of winningChips and store the location of the piece.
    for(topleft=1; topleft < 4; topleft++) {
        if (board[rowPlaced - topleft][lastMove - topleft] != lastColor) {
            break;
        }
        index = index + 1;
        winningChips[index][0] = rowPlaced - topleft;
        winningChips[index][1] = lastMove - topleft;
    }

    //  Bottomright check. //
    // If the active player doesn't have a piece [bottomright] places to the bottomright
    // of his/her most recently-placed piece, stop the loop early. Else increase the
    // indexing of winningChips and store the location of the piece.
    for(bottomright=1; bottomright < 4; bottomright++) {
        if (board[rowPlaced + bottomright][lastMove + bottomright] != lastColor) {
            break;
        }
        index = index + 1;
        winningChips[index][0] = rowPlaced + bottomright;
        winningChips[index][1] = lastMove + bottomright;
    }
    //  If the total number of pieces in this line (including the most recently-placed
    // one) is four or more, the active player is deemed the victor. The program still
    // checks for other winning combinations in other lines.
    if (topleft+bottomright-1 >= 4) {
        victory = lastColor;
    }
    //  If the total number of pieces is three or less, victory has not been achieved
    // and the index of winningChips is restored to what it was before the line was
    // checked (zero).
    else {
        index = 0;
    }

    // Left/right victory check.
    //// Left check.
    for(left=1; left < 4; left++) {
        if (board[rowPlaced][lastMove - left] != lastColor) {
            break;
        }
        index = index + 1;
        winningChips[index][0] = rowPlaced;
        winningChips[index][1] = lastMove - left;
    }
    //// Right check.
    for(right=1; right < 4; right++) {
        if (board[rowPlaced][lastMove + right] != lastColor) {
            break;
        }
        index = index + 1;
        winningChips[index][0] = rowPlaced;
        winningChips[index][1] = lastMove + right;
```

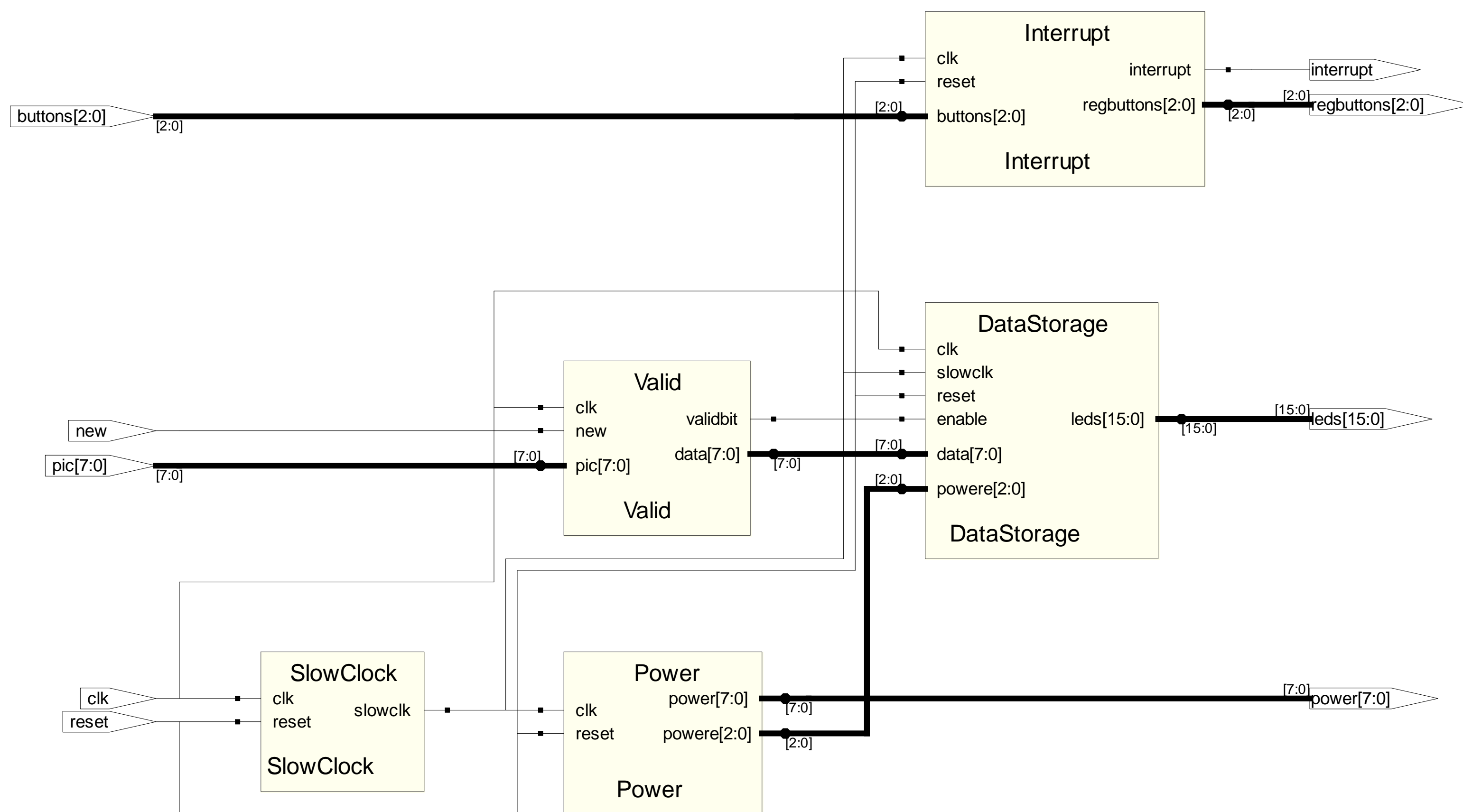
```
    }
    if (left+right-1 >= 4) {
        victory = lastColor;
    }
    else {
        //  The index is not reset to zero, because this could negate a previously-found
        // set of winning chips. Instead index is returned to what it was before this
        // line was checked.
        index = index - (left + right - 2);
    }

    // Bottomleft/topright victory check.
    //// Bottomleft check.
    for(bottomleft=1; bottomleft < 4; bottomleft++) {
        if (board[rowPlaced + bottomleft][lastMove - bottomleft] != lastColor) {
            break;
        }
        index = index + 1;
        winningChips[index][0] = rowPlaced + bottomleft;
        winningChips[index][1] = lastMove - bottomleft;
    }
    //// Topright check.
    for(topright=1; topright < 4; topright++) {
        if (board[rowPlaced - topright][lastMove + topright] != lastColor) {
            break;
        }
        index = index + 1;
        winningChips[index][0] = rowPlaced - topright;
        winningChips[index][1] = lastMove + topright;
    }
    if (bottomleft + topright-1 >= 4) {
        victory = lastColor;
    }
    else {
        index = index - (bottomleft + topright - 2);
    }

    // Bottom victory check. Note that top is not checked because there can be no pieces
    // above the most-recently placed one.
    for(bottom=1; bottom < 4; bottom++) {
        if (board[rowPlaced + bottom][lastMove] != lastColor) {
            break;
        }
        index = index + 1;
        winningChips[index][0] = rowPlaced + bottom;
        winningChips[index][1] = lastMove;
    }
    if (bottom >= 4) {
        victory = lastColor;
    }
    else {
        index = index - (bottom - 1);
    }

    // Victory effects.
    //  If victory has been achieved the winner's color is shown on an LED and the most
    // recently-placed piece's position is stored. Additionally the total number of
    // winning chips is set equal to the current value of index. Any additional
    // (erroneous) information stored in the winningChips matrix will thus be ignored,
    // and there is no need to reset winningChips. Lastly, the cursor should not be
    // displayed when the user can no longer control it.
    if (victory != 0) {
        board[2][7] = lastColor;
        winningChips[0][0] = rowPlaced;
        winningChips[0][1] = lastMove;
        totalWinningChips = index;
        board[1][colHighlight] = 0;
```

```
    }


    // Tie condition check.
    //  If victory is not found, it is possible a tie has occurred. This is tested by
    // checking if the top row is filled with pieces. If it is then a tie has occurred.
    else {
        for (i=0;i<7;) {                //  Only check the first seven spaces, since
                                        // the eighth isn't part of the game board.
            if (board[2][i] == 0) {     // If a space is empty, break out of the loop.
                break;
            }
            i = i + 1;                  //  If it isn't move on to the next.
        }
        if (i == 7) {                   //  If all seven spaces are filled, then a
            tie = 1;                    // tie has occurred.
            board[1][colHighlight] = 0; //  Remove cursor.
            board[2][7] = 3;            //  Display tie LED.
        }
    }
}

//displayModule/////////////////////////////////////////////////////////////////////////

//  Module responsible for keeping LED display up-to-date it goes through every
// element of the matrix and tells the FPGA to update the LED display to the appropriate
// value based on what the PIC has stored on its software copy of the board.

void displayModule(void){
    char i, j;                              //initialize some indicies
    for(i=0;i<8;i++){                       //look at every element on the display
        for(j=0;j<8;j++){
            updateDisplay(board[i][j],i,j); //update the FPGA's copy of the board based
                                            //      on PIC's copy
        }
    }
}

//endAnimationModule////////////////////////////////////////////////////////////////////

//  When a player wins, this function runs the animation which alternates the color
// of the winning pieces between the winning player's color and yellow. If it is a tie
// a different animation play highlighting all of the pieces on the board. It also waits
// for both players to press select. At this point the animation breaks so that the
// players can start a new game.


void endAnimationModule(void) {
    char i;         //instantiate indicies
    char j;

    //This is the code that runs if there is a victory
    if (victory) {
        // highlight all of the winning chips in yellow
        for (i = 0; i <= totalWinningChips;) {
            updateDisplay(3,winningChips[i][0],winningChips[i][1]);
            //wait a while before changing the next one so that the animation was visible
            for (t = 100000;t>0;) {
                t = t + 1;
            }
            i = i + 1;
        }
        //return all the pieces back to the winning player's color
        for (i = 0; i <= totalWinningChips;) {
            updateDisplay(lastColor,winningChips[i][0],winningChips[i][1]);
            // wait so that the time between changes is visible
            for (t = 100000;t>0;) {
```

```
                t = t + 1;
            }
            i = i + 1;
        }
        //Look for both players to press select
        if (playerSelect != 0) {
            playerReady = 1;
        }
        if (oppSelect != 0) {
            oppReady = 1;
        }
        if (playerReady && oppReady) {
            //Start a new game when both players have pressed select
            newGame = 1;
        }
    }

    //This controls the tie animations
    else if (tie) {
        //color the whole board yellow
        for(i=2;i<8;i++){
            for(j=0;j<7;j++){
                updateDisplay(3,i,j);
                //wait so that time between changes is visible
                while(t<15000){
                    t++;
                }
                t=0;
            }
            for(j=0;j<7;j++){
                updateDisplay(board[i][j],i,j);
                //wait so that time between changes is visible
                while(t<15000){
                    t++;
                }
                t=0;
            }
            //Look for both players to press select
            if (playerSelect != 0) {
                playerReady = 1;
            }
            if (oppSelect != 0) {
                oppReady = 1;
            }
            if (playerReady && oppReady) {
                //Start a new game when both players have pressed select
                newGame = 1;
                break;
            }
        }
    }


}


//Functions not directly called by main()////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////

//updateDisplay///////////////////////////////////////////////////////////////////////////
//updateDisplay(color, row, col)
//   This function takes in the color, row, and column then tells
//the FPGA this information. It also tellss the FPGA that it can read
//by setting RE2, waiting for a while for the FPGA to read it, then
//turning RE2 back off to complete transmsision.
//The data is sent over PORT D with the following packet structure...
//{Color1, Color0, Row2, Row 1, Row0, Col2, Col1, Col0}
```

```
void updateDisplay(char color, char row, char col){
    //convert row and column to FPGA's matrix indexing
    row=row-1;
    col=-(col-7);

    //mask out the relevant bits and shift to appropriate location
    color = (color & 0b00000011) << 6;  //mask 2 color LSBs and shift
    row = (row & 0b00000111) << 3;      //mask 3 row LSBs and shift
    col = col & 0b00000111;             //mask 3 col LSBs

    //combine into portD
    PORTD=(color^row^col);

    //Let FPGA read data
    PORTEbits.RE2=1;                    //set RE2
    PORTEbits.RE2=0;                    //clear RE2
}

//dropAttempt//////////////////////////////////////////////////////////////////////

//dropAttempt(col, color)
//   Attempts to drop a piece into the column (col). If it fails this
// function does nothing. If it suceeds it will update the board with the
// appropriate color and tell the FPGA what to display
char dropAttempt(char col, char color){
    char minrow;    //Get the MIN_ROW
    minrow=MIN_ROW;
    rowPlaced= findRow(col);
    if (rowPlaced >=minrow && rowPlaced <=7){       //if it is a valid row
        board[rowPlaced][col]=color;                //update the board
        return 0;
    }
    else {
        return 1;
    }
}

//findRow//////////////////////////////////////////////////////////////////////////

//(char)findRow(col)
//   This function takes in a column (that the user dropped the
//piece into), looks at the board (a global variable), then finds
//the row that the piece should be in using connect 4 rules

char findRow(char col){
    char row;                           //the row to put the piece in

    //look for the highest numbered row with a piece
    for (row=7; row>=MIN_ROW; row--){   //start from the bottom and search up
        if (board[row][col] == 0){
            break;                      //break out when you find the first no zero row
        }
    }
    return row;
}
```

APPENDIX B: FPGA RTL Schematic


Interrupt
clk
reset
buttons[2:0]
interrupt
regbuttons[2:0]
Interrupt
buttons[2:0]
interrupt
regbuttons[2:0]
DataStorage
clk
slowclk
reset
enable
data[7:0]
powere[2:0]
leds[15:0]
DataStorage
leds[15:0]
Valid
clk
new
pic[7:0]
validbit
data[7:0]
Valid
new
pic[7:0]
SlowClock
clk
reset
slowclk
SlowClock
clk
reset
Power
clk
reset
power[7:0]
powere[2:0]
Power
power[7:0]

APPENDIX C: FPGA Verilog Code

```
1        `timescale 1ns / 1ps
2        ////////////////////////////////////////////////////////////////////////////////
3        // Company: Harvey Mudd College
4        // Engineer: Narayan Propato
5        //
6        // Create Date:    18:56:09 11/20/2009
7        // Design Name:    Top Module
8        // Module Name:    TopModule
9        // Project Name:   FPGA LED Display controller
10       // Target Devices: XC3S400
11       // Tool versions:
12       // Description: This module is responsible for time-multiplexing the data storage
13       // module, debouncing the buttons, and providing an interrupt signal to the PIC.
14       // Revision: 1.0
15       // Revision 0.01 - File Created
16       // Additional Comments:
17       ////////////////////////////////////////////////////////////////////////////////
18       module TopModule(
19           input clk,
20           input reset,
21           input [2:0] buttons,
22           input [7:0] pic,
23           input new,
24           output interrupt,
25           output [2:0] regbuttons,
26           output [7:0] power,
27           output [15:0] leds
28           );
29
30          wire slowclk;
31          wire [7:0] data;
32          wire [2:0] powere;
33
34          //Instances
35          Valid Valid(clk, pic, new, data, validbit);
36          SlowClock SlowClock(clk,reset,slowclk);
37          Interrupt Interrupt(slowclk,reset,buttons,interrupt,regbuttons);
38          Power Power(slowclk, reset, power, powere);
39          DataStorage DataStorage(clk, slowclk, reset, data, validbit, powere, leds);
40
41       endmodule
42
```

```
1	`timescale 1ns / 1ps
2	//////////////////////////////////////////////////////////////////////////////////
3	// Company: Harvey Mudd College
4	// Engineer: Narayan Propato
5	//
6	// Create Date:    19:00:49 11/20/2009
7	// Design Name:    Valid Module
8	// Module Name:    Valid
9	// Project Name:   FPGA LED Display controller
10	// Target Devices: XC3S400
11	// Tool versions:
12	// Description:    This module is responsible for determining whether the PIC data
13	// is new (and therefore valid) or not. If it is, VALIDBIT is high and DataStorage
14	// writes the PIC data into memory.
15	// Dependencies:
16	// Revision: 1.0
17	// Revision 0.01 - File Created
18	// Additional Comments:
19	//////////////////////////////////////////////////////////////////////////////////
20	module Valid(
21	    input clk,
22	    input [7:0] pic,
23	    input new,
24	    output reg [7:0] data,
25	    output validbit,
26	    );
27	   reg [1:0] valid;
28	
29	   // validbit is held high when PIC input data is new. Note that new is provided
30	   // by the PIC to notify the FPGA that new data is available.
31	   always@(posedge clk)
32	      begin
33	         data <= pic;
34	         valid[1] <= new;
35	         valid[0] <= valid[1];
36	      end
37	
38	assign validbit = valid[1]&~valid[0];
39	
40	endmodule
41
```

```
1        `timescale 1ns / 1ps
2        //////////////////////////////////////////////////////////////////////////////////
3        // Company: Harvey Mudd College
4        // Engineer: Narayan Propato
5        //
6        // Create Date:    19:00:49 11/20/2009
7        // Design Name:    Slow Clock Module
8        // Module Name:    SlowClock
9        // Project Name:   FPGA LED Display controller
10       // Target Devices: XC3S400
11       // Tool versions:
12       // Description:    This module is responsible for making a slow clock signal from
13       // the input clock signal. This slow clock signal is used to cycle power between
14       // the LED display rows and to read the row data from DataStorage at the same
15       // speed.
16       // Dependencies:
17       // Revision: 1.0
18       // Revision 0.01 - File Created
19       // Additional Comments:
20       //////////////////////////////////////////////////////////////////////////////////
21       module SlowClock(
22           input clk,
23           input reset,
24           output slowclk
25           );
26
27          reg [11:0] counter;
28          wire [11:0] ncounter;
29
30
31          // Slow clock logic, used to power the rows on the LED display at a reasonable rate.
32          // Counter will automatically reset once it overflows.
33          always@ (posedge clk, posedge reset)
34             if (reset == 1)
35                counter <= 12'b011111111111;
36             else
37                counter <= ncounter;
38
39          assign ncounter = counter + 12'b000000000001;
40          assign slowclk = counter[11];
41
42
43       endmodule
44
```

```
1        `timescale 1ns / 1ps
2        ////////////////////////////////////////////////////////////////////////////////
3        // Company: Harvey Mudd College
4        // Engineer: Narayan Propato
5        //
6        // Create Date:    19:00:49 11/20/2009
7        // Design Name:    Interrupt module
8        // Module Name:    Interrupt
9        // Project Name:   FPGA LED Display controller
10       // Target Devices: XC3S400
11       // Tool versions:
12       // Description:    This module is responsible for debouncing the user button
13       // input as well as providing an interrupt signal to the PIC that is dependant
14       // on user input.
15       // Dependencies:
16       // Revision: 1.0
17       // Revision 0.01 - File Created
18       // Additional Comments:
19       ////////////////////////////////////////////////////////////////////////////////
20       module Interrupt(
21           input clk,
22           input reset,
23           input [2:0] buttons,
24           output reg interrupt,
25           output reg [2:0] regbuttons
26           );
27
28          reg [1:0] leftreg;
29          reg [1:0] rightreg;
30          reg [1:0] selectreg;
31          reg [2:0] btnvalid;
32
33          // These registers keep track of the last two sampled values
34          // of each button.
35          always@(posedge clk)
36             begin
37                leftreg[1] <= buttons[2];
38                leftreg[0] <= leftreg[1];
39                rightreg[1] <= buttons[1];
40                rightreg[0] <= rightreg[1];
41                selectreg[1] <= buttons[0];
42                selectreg[0] <= selectreg[1];
43             end
44
45          // A valid button input is defined as a change from zero
46          // (not pressed) to one (pressed).
47          always@(*)
48             begin
49                if (leftreg == 2'b10)
50                   btnvalid[2] <= 1;
51                else
52                   btnvalid[2] <= 0;
53                if (rightreg == 2'b10)
54                   btnvalid[1] <= 1;
55                else
56                   btnvalid[1] <= 0;
57                if (selectreg == 2'b10)
58                   btnvalid[0] <= 1;
59                else
60                   btnvalid[0] <= 0;
61             end
```

```
62
63          // The interrupt signal is toggled on the positive edge of
64          // any of the button inputs. Additionally, the buttons sent
65          // to the PIC are masked such that only those undergoing a
66          // change from zero to one are held high. Those that were
67          // already previously one are held low.
68          always@(posedge clk, posedge reset)
69             if (reset == 1)
70                interrupt <= 0;
71             else if (|btnvalid)
72                begin
73                   regbuttons <= {btnvalid[2]&leftreg[1],btnvalid[1]&rightreg[1],btnvalid[0]&
       selectreg[1]};
74                   interrupt  <= ~interrupt;
75                end
76             else
77                regbuttons <= 0;
78
79       endmodule
80
```

```
1    `timescale 1ns / 1ps
2    //////////////////////////////////////////////////////////////////////////////////
3    // Company: Harvey Mudd College
4    // Engineer: Narayan Propato
5    //
6    // Create Date:    19:00:49 11/20/2009
7    // Design Name:    Power module
8    // Module Name:    Power
9    // Project Name:   FPGA LED Display controller
10   // Target Devices: XC3S400
11   // Tool versions:
12   // Description:    This module is providing power to the different LED display
13   // rows in order to time-multiplex the DataStorage module. Two registers are
14   // defined: POWER and POWERE. POWER is outputted to the LED display, while POWERE
15   // is the encoded version that is used for preventing read/write conflicts in the
16   // Data Storage module.
17   // Dependencies:
18   // Revision: 1.0
19   // Revision 0.01 - File Created
20   // Additional Comments:
21   //////////////////////////////////////////////////////////////////////////////////
22   module Power(
23       input clk,
24       input reset,
25       output reg [7:0] power,
26       output reg [2:0] powere
27       );
28
29      // Power is provided by cycling the active bit in power, which corresponds
30      // to the row that is being powered. Note that the power bits is active low,
31      // because they are connected to PNP transistors. powere is used for internal
32      // logic purposes.
33      always@(posedge clk, posedge reset)
34         if(reset)
35            begin
36               power <= 8'b01111111;
37               powere <= 3'b000;
38            end
39         else
40            begin
41               power <= {power[0],power[7:1]};
42               powere <= powere + 1;
43            end
44
45   endmodule
46
```

```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////////////////////
3   // Company: Harvey Mudd College
4   // Engineer: Narayan Propato
5   //
6   // Create Date:    19:00:49 11/20/2009
7   // Design Name:    Data Storage Module
8   // Module Name:    DataStorage
9   // Project Name:   FPGA LED Display controller
10  // Target Devices: XC3S400
11  // Tool versions:
12  // Description:    This module is responsible for storing the matrix information
13  // pertaining to the game board in a memory block, as well as updating this matrix
14  // with changes provided by the PIC.
15  // Dependencies:
16  // Revision: 1.0
17  // Revision 0.01 - File Created
18  // Additional Comments:
19  //////////////////////////////////////////////////////////////////////////////////
20  module DataStorage(
21      input clk,
22      input slowclk,
23      input reset,
24      input [7:0] data,
25      input enable,
26      input [2:0] powere,
27      output reg [15:0] leds
28      );
29
30     reg [1:0] color;
31     reg [2:0] row;
32     reg [2:0] column;
33     reg [15:0] memory [7:0];
34     integer i;
35
36     // Dividing PIC data into its sub-components.
37     always@(*)
38        begin
39           color <= ~data[7:6];
40           row <= data[5:3];
41           column <= data[2:0];
42        end
43
44     // Data storage memory block write logic. When an element in the matrix needs
45     // to be changed, the corresponding row is read, altered, and stored again.
46     always@(posedge clk, posedge reset)
47        if (reset == 1)
48           begin
49              for (i = 0; i < 8; i = i +1)
50                 memory[i] <= 16'b1111111111111111;
51           end
52        else if (enable == 1)
53           begin
54              case(column)
55                 3'b111: memory[row] <= {color, memory[row][13:0]};
56                 3'b110: memory[row] <= {memory[row][15:14],  color, memory[row][11:0]};
57                 3'b101: memory[row] <= {memory[row][15:12],  color, memory[row][9:0]};
58                 3'b100: memory[row] <= {memory[row][15:10],  color, memory[row][7:0]};
59                 3'b011: memory[row] <= {memory[row][15:8] ,  color, memory[row][5:0]};
60                 3'b010: memory[row] <= {memory[row][15:6] ,  color, memory[row][3:0]};
61                 3'b001: memory[row] <= {memory[row][15:4] ,  color, memory[row][1:0]};
```

```
62                      3'b000: memory[row] <= {memory[row][15:2] ,  color};
63                      default: memory[row] <= 0;
64                   endcase
65                end
66 
67          // Memory block read logic. Reads are only performed when a write is not.
68          always@(posedge slowclk, posedge reset)
69             if (reset == 1)
70                begin
71                   leds  <= 16'b1111111111111111;
72                end
73             else if (enable == 0)
74                begin
75                   leds <= memory[powere];
76                end
77 
78       endmodule
79 
```

APPENDIX D: Pin-Out

FPGA pin	PIC pin	Function
P124	Clk1	Clock
p131	RB5	interrupt/Button Change Interrupt
p80	RE2	NEW / PIC-FPGA handshaking
p79	RE1	FPGA reset
	RC7	UART Rx
	RC6	UART Tx
p87	RC5	buttons[2]/Left Button
p86	RC4	buttons[1]/Right Button
p85	RC3	buttons[0]/Select Button
p97	RD7	PIC[0]/color[1]
p98	RD6	PIC[1]/color[0]
p99	RD5	PIC[2]/row[2]
p100	RD4	PIC[3]/row[1]
p102	RD3	PIC[4]/row[0]
p103	RD2	PIC[5]/col[2]
p104	RD1	PIC[6]/col[1]
p105	RD0	PIC[7]/col[0]
p13		power[0]
p17		power[1]
p21		power[2]
p25		power[3]
p116		power[4]
p2		power[5]
p6		power[6]
p10		power[7]
p11		leds[0]
p12		leds[1]
p14		leds[2]
p15		leds[3]
p18		leds[4]
p20		leds[5]
p23		leds[6]
p24		leds[7]
p112		leds[8]
p113		leds[9]
p118		leds[10]
p1		leds[11]
p4		leds[12]
p5		leds[13]
p7		leds[14]
p8		leds[15]

APPENDIX E: Parts List

Part	Source	Manufacturer Part ID	Number	Total Price
Pushbuttons	R-VAC Electronics	8161J81ZQE22/ D6R10 F2 LFS	4/2	$8.35
Power Switches	R-VAC Electronics	PRCSA1-20L-BB0CW	2	$1.90
Battery Holders	R-VAC Electronics	BC3AAW	2	$2.50
Dot Matrix LED Displays	Seeed Studio Depot	LED203A58	2	$11.00

Note: values given are for assembly of two prototype devices, not one.