PIC Tetris

Final Project Report
December 6, 2006
E155: Microprocessor-Based Systems

Philip Amberg and Kevin Zielnicki

Abstract

Tetris is a classic video game where falling blocks are arranged to complete lines on the playing board. The goal is to get to the highest score possible before the falling pieces fill up the board. In our project, the PIC is responsible for running the game code and accepting the input from the controller. The FPGA is responsible for accepting the board data from the PIC and displaying the graphics on a VGA monitor. We were successful in creating a fully functioning Tetris game that is controlled with an NES controller and displays 256 color graphics on a VGA monitor.

Introduction

Tetris is a falling block puzzle, developed in 1985 by Alexey Pazhitnov. The goal of the game is to get as many points as possible before the board fills up with pieces. Points are scored by completing lines and accelerating pieces down the board. When a line of the board is completely full of pieces, that line is cleared from the board, making more room above to stack pieces. A popular implementation of the game on the Nintendo Entertainment System (NES) is shown in Figure 1.

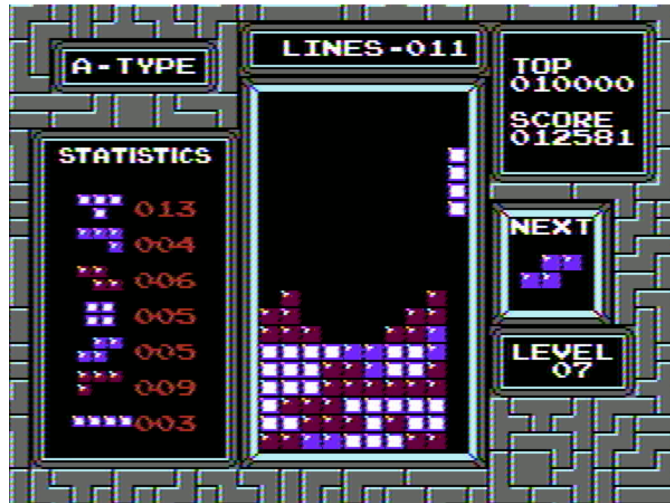

Figure 1: Screenshot of the NES version of Tetris.

To implement this game in our hardware, we split the required tasks between the PIC and the FPGA. The PIC is responsible for running the game code which computes things such as the position and speed of the falling block, current score, and the current level. The PIC is also responsible for polling the NES controller to determine which buttons are pressed. The FPGA is responsible for receiving data about the state of the board from the PIC and processing this into a signal that can be displayed on a VGA monitor at a 640x480 pixel resolution. The layout of our system is shown in Figure 2.

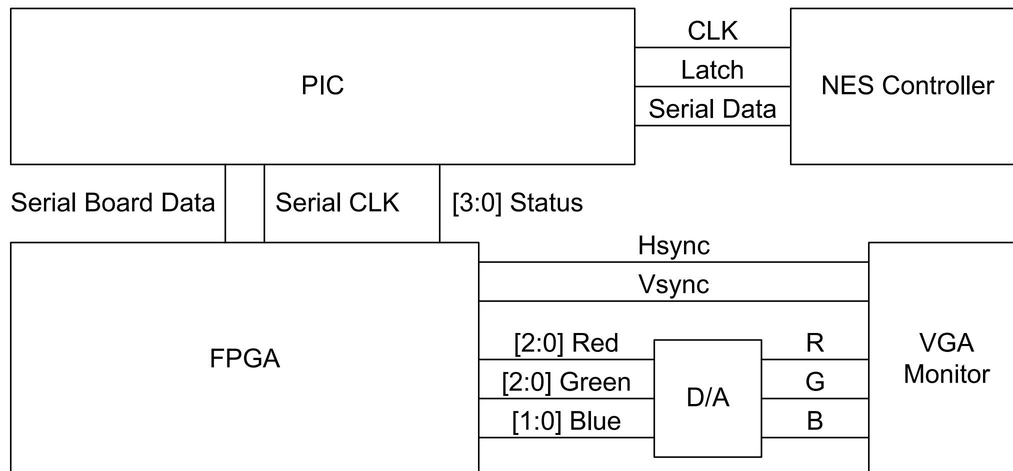

Figure 2: Block Diagram of System.

New Hardware

Our project used an NES controller and a VGA monitor with 256 color graphics, which have not been used in an E155 project before.

To use the NES controller in our design, we have to mimic what the NES does to receive the output. When the NES is ready to receive controller data, the Latch line is pulsed high. This causes the state of the buttons to get latched into an 8-bit shift register within the controller. After the latch line is pulsed, the state of button A can be read on the data line. A low signal means the button is pressed while a high signal means the button was not pressed. To obtain the states of the rest of the buttons, the Clk line is pulsed. Pulsing the clock line will shift the state of the next button onto the Data line. The order of button states on the Data line goes A, B, Select, Start, Up, Down, Left, then Right.

A function, `pollController()`, was written in C to handle the input for our game. This function is available in appendix B. The pinout of the NES controller is shown in Figure 3, and the timing diagram is shown in Figure 4. In our case we used a Clk frequency of 120.5 kHz.

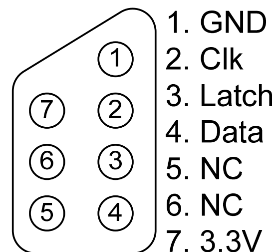

Figure 3: NES Controller pinout.

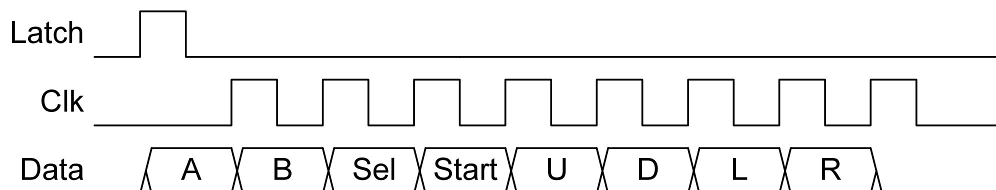

Figure 4: Controller Timing and Interface signals.

Implementing color on a VGA monitor simply involves converting a digital signal representing each color into an analog signal for the monitor. The VGA monitor has three pins for color, red, green, and blue. To control the color displayed on the screen, an analog voltage ranging from 0-0.7 V controls the intensity of each color. To display 256 unique colors, we used an 8-bit number to represent each color, 3 bits for red, 3 bits for green, and 2 bits for blue. To convert our 3.3 V digital output from the FPGA to a 0-0.7 V analog signal, we used an R-2R D/A converter as shown in Figure 5. This gives each bit a binary weight in the overall analog signal.

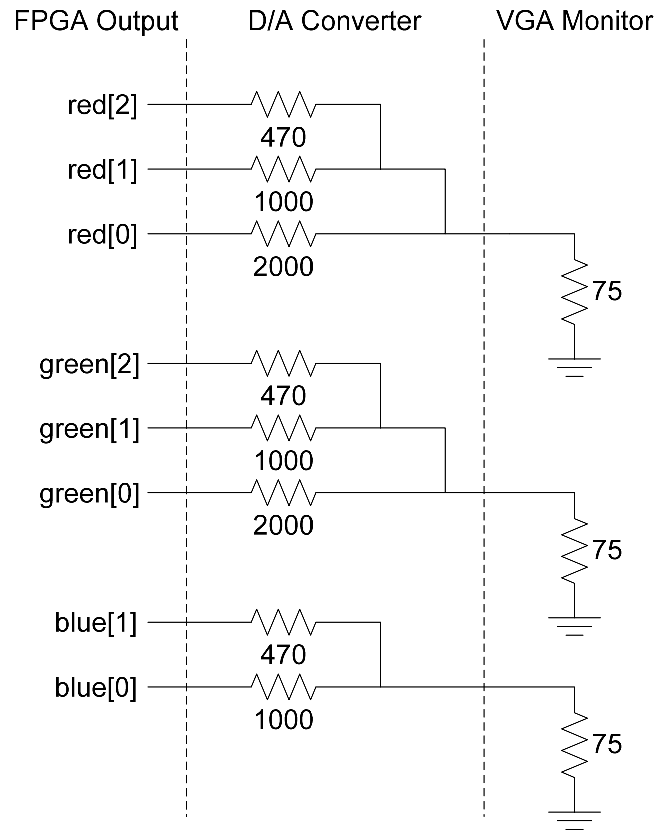

Figure 5: R-2R D/A converter used in our design.

Schematics

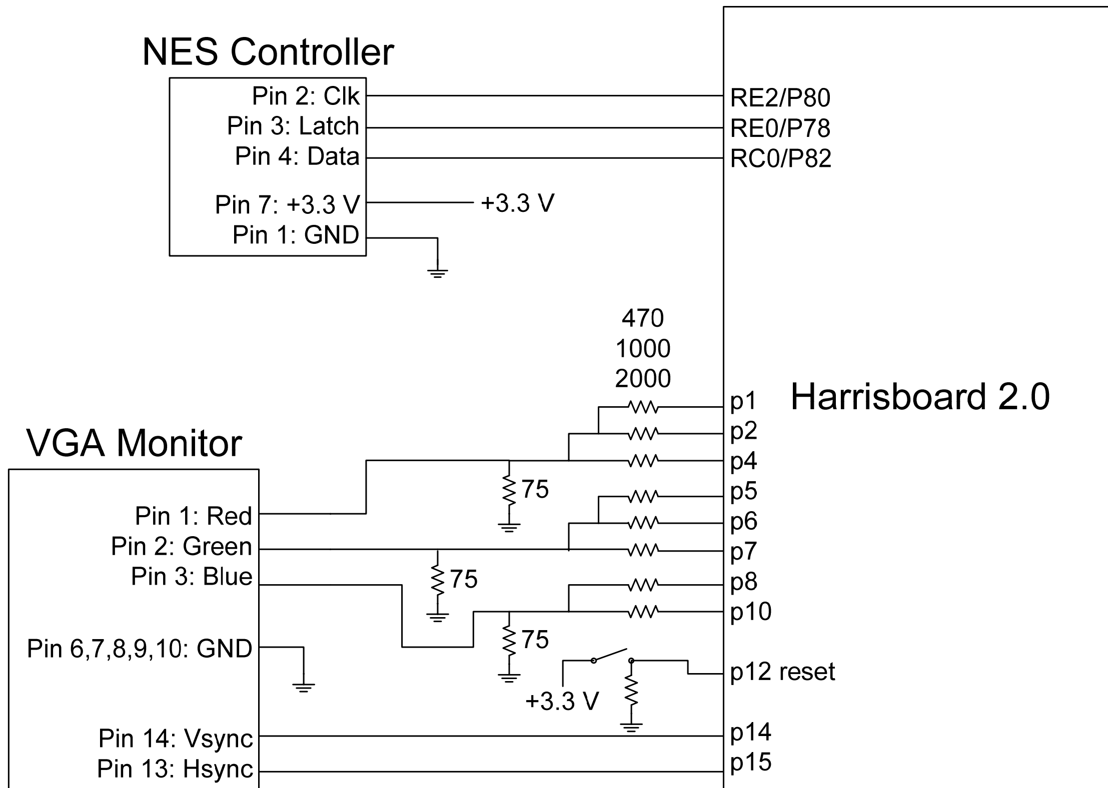

Figure 6: Schematic of System Interconnections and Board Layout

Microcontroller Design

Inputs

- NES controller data (1 bit)

Outputs

- Latch signal for NES controller (1 bit)
- Clock signal for NES controller (1 bit)
- Game status for controlling FPGA (4 bits)
- Serial clock for sending data to FPGA (1 bit)
- Serial data out to FPGA (1 bit)

The microcontroller must accomplish three primary tasks. First, it must run the game of tetris, which involves generating random pieces (tetrominoes) and advancing them down the screen, clearing completed rows, keeping track of the score and level, and checking if a block has extended past the top of the screen, causing the game to end. Secondly, it must receive input from the user via a NES controller, which is used to direct the active tetromino left or right across the screen, rotate it, or accelerate it downwards towards the bottom of the screen. Finally, it must transmit information about the game to the FPGA, which handles graphics processing. This involves a serial connection which sends data about the type of tiles located at various positions on the screen, and a status signal which gives the FPGA information about the current state of the game or what type of data is currently being sent over the serial port.

The basic structure used to accomplish these tasks is evident in the `main()` function of the program. After some basic I/O and timer configuration, the program enters an infinite loop. In the first stage of this loop, the FPGA is directed to display the start screen, and microcontroller waits until the user presses the start button. After this occurs, some initialization values are set, and the actual game begins. In the game loop, the controller is first polled to determine which buttons the user pressed. Based on those inputs, the program will optionally call various functions to rotate or move the piece, or pause the game if the user pressed 'start'. Next, the program checks if the time elapsed is greater than a cutoff time, and if so advances the tetromino down to the next row. The exact amount of time between piece advances depends on the level, and becomes shorter as the game progresses. Finally, the current state of the board is sent to the FPGA.

Although `main()` ties everything together, some of the microcontroller's behavior is not evident from this overview because several important things occur in the `advanceBlock()` function, which is called by main every time the current block needs to move down a row. Additionally, `advanceBlock()` serves as an archetype for the other piece movement functions, `moveBlock()` and `rotate()`, which behave similarly. When this function is called, it copies the currently active tetromino into a temporary variable, advances the y coordinate of the temporary piece, and checks if it has collided with anything, implying that it has hit bottom or another piece. On most calls to this function, there will be no collision, so the modified block is accepted and displayed. However, if there was a collision, several tasks need to be performed. First, any bonus points accumulated by accelerating the piece downwards must be added to the score. Next, the

current piece needs to be replaced with the next piece, and a new next piece must be generated and sent to the FPGA. Then, the program must check if any rows have been completely filled, and if so, clear them and update the score and possibly the level accordingly. Finally, if any part of old piece extends beyond the top of the screen, the game has ended and a gameover flag is set to indicate this.

Going back to `main()`, once the gameover flag is set, the FPGA is signaled to let it know the game has ended, and the microcontroller waits until start has been pressed again to restart the game. For more detail, see the listing of the C code in appendix B.

FPGA Design

The FPGA is responsible for receiving the board data from the PIC and using this to generate the graphics to display on the VGA monitor. Figure 7 shows an overview of the 4 main units of the FPGA.

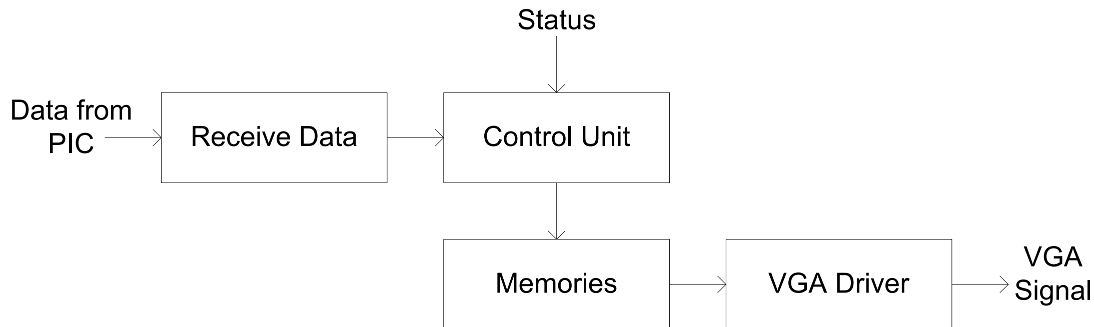

Figure 7: Block Diagram of FPGA System

Receive Data

The receive data unit of the FPGA is responsible for receiving the 10 MHz serial signal from the PIC and converting it to a parallel signal at 25 MHz. This is accomplished by using an 8-bit shift register, clocked to the serial clock of the PIC. Once the 8 bits are filled, the parallel data passes through a synchronizer running at 25 MHz to mitigate metastability and have the signal running at the speed of the FPGA.

Control Unit

The control unit is responsible for determining which of the board data memories is displaying on the screen and which is being written. The control unit consists of two finite state machines, the control FSM and the gamestate FSM. The control FSM reads the status bits from the PIC to determine what data the PIC is about to send or which state the game should enter. For example, when the status bits for writing a new board are asserted, the FSM enters a state and waits there until the status bits go low, meaning the PIC is done sending data. While in this state, the memory address where the received data should be written are computed and the data is written to the correct memory. Other status bits can indicate that the game should start or end. These signals tell the gamestate FSM to switch the board memories so either the start or ending screens are displayed. This unit also handles the displaying of the score, high score, level, and next piece.

Memories

Our system uses 5 memories to perform its graphics functions. One memory to hold the start screen, two memories to hold the game screen, one memory to hold the ending screen, and one large memory to hold the bitmap data for our tiles. The first 4 memories are structured as in Figure 8 and the bitmap data memory is structured as in Figure 9. To update the screen during gameplay, our system utilizes a common graphics

technique known in the game design world as double buffering. While one memory is being read to display the graphics, the next screen is being written to a second memory. When the second memory is ready, the active memory switches and the new image is shown on the screen. This eliminates the flicker that would result from writing and displaying from the same memory. When playing the game, the control unit controls which memory is sent to the display and which is being written to.

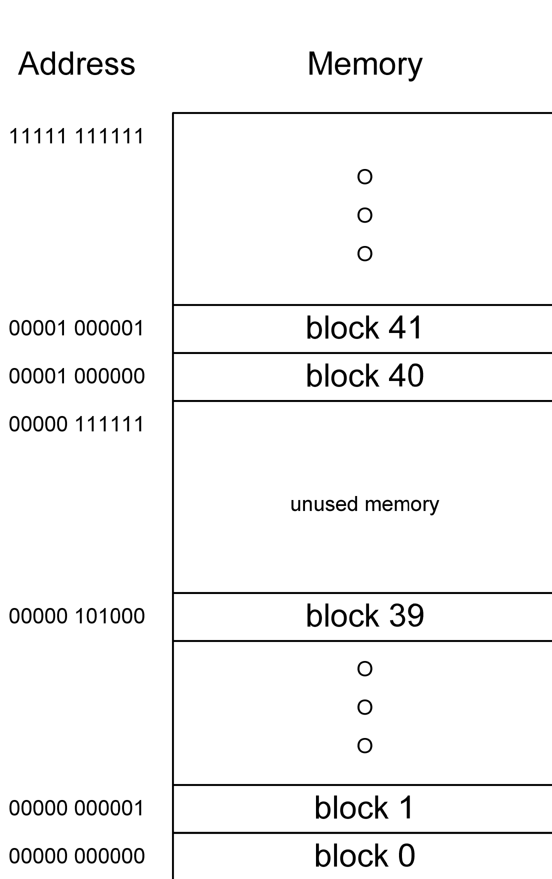

Figure 8: Layout of Boardstate Memory.

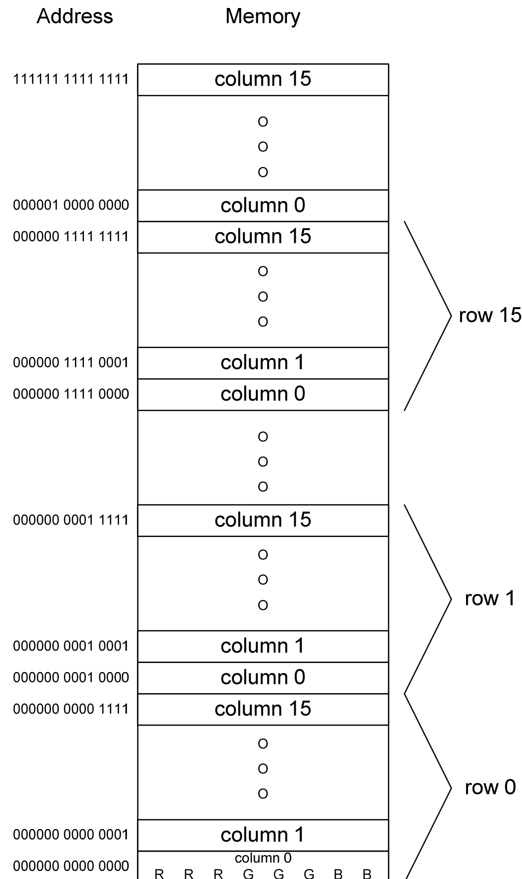

Figure 9: Layout of Tilemap Memory

VGA Driver

To display our graphics, we divided the 640x480 pixel screen into 1200 16x16 pixel tiles. Each of these tiles has an 8-bit number associated with it, indicating the type of block that it is. The block types for the 1200 blocks are stored in a memory which is used to draw the corresponding picture on the screen. A series of counters is able to calculate which of the 1200 blocks of the screen is active and which pixel within that block we are drawing. This data is used to generate an address which accesses a large 16 kB memory which accesses the pixel of the tile to be drawn in this location.

To step down the clock to 25 MHz and to generate the Hsync and Vsync signals for the monitor, the MicroToys VGA tutorial code was modified to suit our needs.

Results

Our project was successful. We implemented Tetris on the PIC using an NES controller for input and displayed the game in 256 colors on a VGA monitor.

We had two main difficulties with our design. The first was trying to use an on chip digital clock manager, DCM, to step down the 40 MHz clock on the FPGA to the 25 MHz required by the VGA signal. Coregen would not generate a DCM that would generate the correct clock frequency when it was downloaded to our FPGA. Instead it would generate clocks in the range of 180 MHz, which did not suit our needs. To fix this problem we used a Xilinx library template for a DCM and modified it to output the correct frequency. This method produced many warnings during implementation but the hardware worked fine.

The second difficulty was in receiving the serial data from the PIC. With our first design, we attempted to send our status codes to the FPGA over the serial port, before we sent the real data. This method did not work because data was being misinterpreted as status codes, causing unpredictable behavior. We then changed our design of this unit to read the status bits from PORTD in parallel. This design proved more manageable and we were able to correctly receive all the data from the PIC.

References

Microtoys VGA Tutorial. http://www4.hmc.edu:8001/Engineering/microtoys/VGA/MicroToys%20VGA.pdf.

NES Controller Pinout. http://www.diylive.net/index.php/2006/02/03/decoding-nes-controller/.

D/A Converter for use with a VGA monitor. http://www.stanford.edu/class/ee108a/documentation/vga.pdf

Tetris. http://en.wikipedia.org/wiki/Tetris.

Parts List

Part	Source	Price
NES Controller	eBay	$9.36
VGA Monitor	Lab	free

Appendix A – Reading bitmap tiles into coregen

To avoid entering our tile memory by hand, we developed a way to convert bitmap images into hex values in a coregen-readable csv file. To do this, we used the free program tbpaint (http://www.assembler.ca/) to create a bitmap image with our 64 16x16 tiles. Then, we wrote a java program to convert the bitmap to hexadecimal values representing the color of each pixel.

Figure A1 – bitmap tiles stored in FPGA memory

```
// Kevin Zielnicki <kzielnicki@hmc.edu>
// 11-13-06
// BMPread.java - convert a bitmap image titled "allbmp.bmp" to a list
// of hex color values
import java.awt.image.*;
import java.io.*;
import javax.imageio.ImageIO;
import java.awt.Color;

public class BMPread
{

	// convert a java color value to its hex representation
	public static void colorAsHex(Color pixel) {
		int color = ( ((pixel.getRed()/32)&0x7) << 5 ) | (
	((pixel.getGreen()/32)&0x7) << 2 ) | ((pixel.getBlue()/32)&0x3);
		System.out.println(Integer.toHexString(color));
	}

	// output all the hex color values in the 16x16 tile located
	// at the given horizontal offset
	public static void printColors(BufferedImage bi, int offset)
	throws IOException {
		for(int r=0; r < 16; ++r)
			for(int c=0; c < 16; ++c)
				colorAsHex( new Color(bi.getRGB(c+offset*16,r))
	);
	}

	// load the image file and print all 64 tiles
	public static void main(String[] args) throws IOException
	{
		File f = new File("allbmp.bmp");
		BufferedImage bi = ImageIO.read(f);

		for(int offset = 0; offset < 64; ++offset) {
			printColors(bi,offset);
		}
	}
}
```

In addition to creating the tiles, we wanted an interactive way to create images using our tiles. To do this, we used mappy, a freely available tile map editor (http://tilemap.co.uk/mappy.php). Mappy allows the user to select from a pallet of tiles, and paint these tiles onto a grid. As such it was a simple matter to import the bitmap image containing our tiles, and use it to create aggregate images for our game. To export the image in a format readable by the following java code, select export as text and in the dialog that comes up, unselect all check boxes besides 'Map Arrays', and select the '1D format" radio button, as shown in figure A2. This will create a text file that can be used as input for the java code below to generate a csv file for coregen.

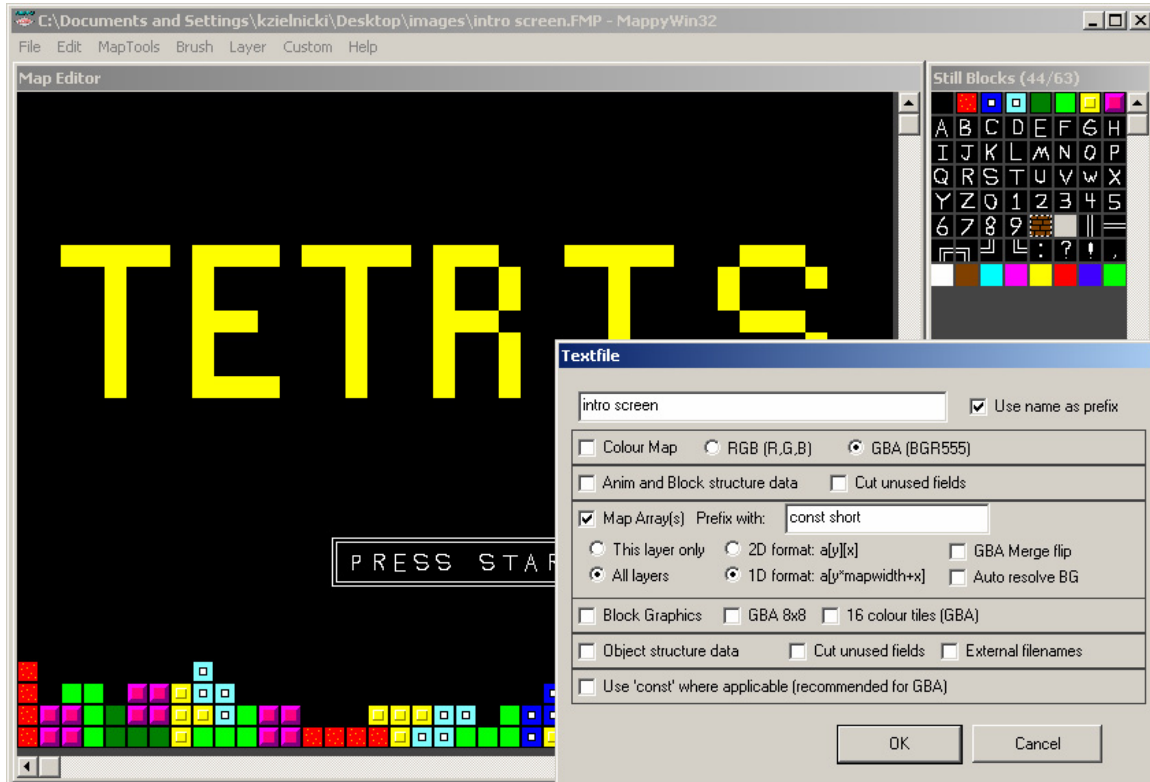

Figure A2 – using Mappy to create and export a tile map

```
// Kevin Zielnicki <kzielnicki@hmc.edu>
// 11-13-06
// creatememe.java - read a text file created with mappy and output a
// coregen readable csv file

import java.util.Scanner;

public class creatememe
{
	public static void main(String[] args) {
		// read in a 30x40 array of tiles
		int[] values = new int[1200];
		Scanner s = new Scanner(System.in).useDelimiter("[\\s,]");

		int loc = 0;
		while(s.hasNext()) {
			while(s.hasNext() && !s.hasNextInt()) {
```

```
                        s.next();
                  }
                  while(s.hasNextInt()) {
                        values[loc] = s.nextInt();
                        loc++;
                  }
            }

            // output the tile numbers as hexadecimal values
            // an offset of 1 is needed because of how mappy indexes
            // the tile pallet
            loc = 0;
            for(int i = 0; i < 64*30; ++i) {
                  if(i%64 >= 40)
                        System.out.println("0");
                  else {
                        System.out.println( Integer.toHexString(
                                                values[loc]+1) );
                        ++loc;
                  }
            }

            for(int i = 0; i < 64*2; ++i) {
                  System.out.println("0");
            }
      }
}
```

Appendix B – Code Listing

```
//////////////////////////////////////////////////////////////////////////////////
// Phil Amberg & Kevin Zielnicki
// 12/02/2006
//
// toplevel.v - top level module for tetris graphics
//////////////////////////////////////////////////////////////////////////////////
module
toplevel(clk,reset,serialclk,serialDataIn,status,sigR,sigG,sigB,Hsync,Vsync,clkfx,clklock
);

        input clk;
        input reset;
        input serialclk;
        input serialDataIn;
        input [3:0] status;
        output [2:0] sigR, sigG;
        output [1:0] sigB;
        output Hsync,Vsync;
        output clkfx,clklock;

        wire [7:0] dataOut;
        wire [10:0] address,writeaddress,readaddress,mem1address,mem2address;
        wire [3:0] state;
        wire [1:0] s;
        wire dataReadyPulse, we, memswitch, gostartScreen, stopGame, gameStart;
        wire [7:0] dout,blocktype,doutmemStart, doutmemGame1, doutmemGame2, doutmemEnd;

        dcmunit dcm1(clk,reset,clkfx,clklock);

        //recieve data from PIC, dataOut comes with dataReadyPulse
        recieve_data dataRecieve(clkfx, serialclk, reset, serialDataIn, dataOut,
dataReadyPulse);

        //control unit
        control_fsm controlUnit(clkfx, reset, status, dataReadyPulse,
stateChangePulse,state, gostartScreen, stopGame, gameStart,address, memswitch);
        statechangepulse_fsm statechangepulse(clkfx, reset, state, stateChangePulse);
        delay1cycle #(11) delayaddress(clkfx,reset,address,writeaddress);
        delay1cycle #(1) delaywriteen(clkfx,reset,dataReadyPulse,we);

        //memories
        startscr startScreenMem(readaddress,clkfx,doutmemStart);
        boardmem1 boardmem1(mem1address,clkfx,dataOut,doutmemGame1,s[1]&we);
        boardmem2 boardmem2(mem2address,clkfx,dataOut,doutmemGame2,~s[1]&we);
        endscr endScreenMem(readaddress,clkfx,doutmemEnd);

        //select read/write memories
        mux11 mem1mux(readaddress,writeaddress,s[1],mem1address);
        mux11 mem2mux(readaddress,writeaddress,~s[1],mem2address);

        gamestatefsm gamestate(clkfx, reset, memswitch,gostartScreen, stopGame, gameStart,
s);

        mux4 boardstateoutmux(doutmemStart, doutmemGame1, doutmemGame2, doutmemEnd, s,
blocktype);

        toplevelvga ppu(clkfx,reset,blocktype, Hsync, Vsync,readaddress, sigR, sigG,
sigB);

endmodule
```

```
////////////////////////////////////////////////////////////////////////////////
// Phil Amberg & Kevin Zielnicki
// 11/05/2006
//
// dcmunit.v - create a 25 MHz clock from our 40 MHz clock
////////////////////////////////////////////////////////////////////////////////
module dcmunit(CLKIN,RST, CLKFX,LOCKED);

input CLKIN;
input RST;
output CLKFX;
output LOCKED;
//  <-----Cut code below this line---->

   // DCM: Digital Clock Manager Circuit for Virtex-II/II-Pro and Spartan-3/3E
   // Xilinx HDL Language Template version 8.1i

   DCM #(
      .CLKDV_DIVIDE(2.0), // Divide by: 1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0,5.5,6.0,6.5
                          //   7.0,7.5,8.0,9.0,10.0,11.0,12.0,13.0,14.0,15.0 or 16.0
      .CLKFX_DIVIDE(8),    // Can be any integer from 1 to 32
      .CLKFX_MULTIPLY(5),  // Can be any integer from 2 to 32
      .CLKIN_DIVIDE_BY_2("FALSE"), // TRUE/FALSE to enable CLKIN divide by two feature
      .CLKIN_PERIOD(25),  // Specify period of input clock
      .CLKOUT_PHASE_SHIFT("NONE"), // Specify phase shift of NONE, FIXED or VARIABLE
      .CLK_FEEDBACK("1X"),  // Specify clock feedback of NONE, 1X or 2X
      .DESKEW_ADJUST("SYSTEM_SYNCHRONOUS"), // SOURCE_SYNCHRONOUS, SYSTEM_SYNCHRONOUS or
                                            //   an integer from 0 to 15
      .DFS_FREQUENCY_MODE("LOW"),  // HIGH or LOW frequency mode for frequency synthesis
      .DLL_FREQUENCY_MODE("LOW"),  // HIGH or LOW frequency mode for DLL
      .DUTY_CYCLE_CORRECTION("TRUE"), // Duty cycle correction, TRUE or FALSE
      .FACTORY_JF(16'hC080),   // FACTORY JF values
      .PHASE_SHIFT(0),     // Amount of fixed phase shift from -255 to 255
      .STARTUP_WAIT("TRUE")   // Delay configuration DONE until DCM LOCK, TRUE/FALSE
   ) DCM_inst (
      //.CLK0(CLK0),     // 0 degree DCM CLK output
      //.CLK180(CLK180), // 180 degree DCM CLK output
      //.CLK270(CLK270), // 270 degree DCM CLK output
      //.CLK2X(CLK2X),   // 2X DCM CLK output
      //.CLK2X180(CLK2X180), // 2X, 180 degree DCM CLK out
      //.CLK90(CLK90),   // 90 degree DCM CLK output
      //.CLKDV(CLKDV),   // Divided DCM CLK out (CLKDV_DIVIDE)
      .CLKFX(CLKFX),   // DCM CLK synthesis out (M/D)
      //.CLKFX180(CLKFX180), // 180 degree CLK synthesis out
      .LOCKED(LOCKED), // DCM LOCK status output
      //.PSDONE(PSDONE), // Dynamic phase adjust done output
      //.STATUS(STATUS), // 8-bit DCM status bits output
      //.CLKFB(CLKFB),   // DCM clock feedback
      .CLKIN(CLKIN),   // Clock input (from IBUFG, BUFG or DCM)
      //.PSCLK(PSCLK),   // Dynamic phase adjust clock input
      //.PSEN(PSEN),     // Dynamic phase adjust enable input
      //.PSINCDEC(PSINCDEC), // Dynamic phase adjust increment/decrement
      .RST(RST)        // DCM asynchronous reset input
   );

   // End of DCM_inst instantiation


endmodule
```

```
////////////////////////////////////////////////////////////////////////////////
// Phil Amberg & Kevin Zielnicki
// 11/29/2006
//
// basic_parts.v - a few basic components used in other modules
////////////////////////////////////////////////////////////////////////////////

// 8-bit shift register
module sr8(clk,reset,serialDataIn,parallelDataOut,dataReady);

	input clk;
	input reset;
	input serialDataIn;
	output [7:0] parallelDataOut;
	output dataReady;

	reg [7:0] data, dataProp, parallelDataOut;
	reg [2:0] count;

	wire dataStart;
	wire dataReady;

	always@(posedge clk, posedge reset)
		if(reset)
			begin
				data<=0;
				count<=0;
				dataProp<=0;
			end
		else
			begin
				data<={data[6:0],serialDataIn};
				count<=count+1;
				dataProp<={dataProp[6:0],dataStart};
			end

	always@(posedge dataReady, posedge reset)
		if(reset)
			parallelDataOut<=0;
		else
			parallelDataOut<=data;


	assign dataReady=dataProp[7];
	assign dataStart=(count==3'b000);

endmodule


module syncronizer(clk,reset,bitsIn,bitsOut);

	input clk;
	input reset;
	input [7:0] bitsIn;
	output [7:0] bitsOut;

	reg [7:0] bitsOut;

	always@(posedge clk, posedge reset)
		if(reset)
			bitsOut<=0;
		else
			bitsOut<=bitsIn;

endmodule

// create a 1 clock cycle pulse when dataReady goes high
module datareadypulse_fsm(clk, reset, dataReadyIn, dataReadyPulse);
    input clk;
    input reset;
    input dataReadyIn;
```

```
    output dataReadyPulse;

        reg[1:0] state,nextstate;

        //parameterize state encodings
        parameter S0=2'b00;
        parameter S1=2'b01;
        parameter S2=2'b10;

        //state register
        always@(posedge clk, posedge reset)
              if(reset)
                     state<=S0;
              else
                     state<=nextstate;

        //nextstate logic
        always@( * )
              case(state)
                     S0:nextstate= dataReadyIn?S1:S0;
                     S1:nextstate=S2;
                     S2:if(~dataReadyIn) nextstate=S0;
                     default:nextstate=S0;
              endcase

        //output logic
        assign dataReadyPulse=state[0];


endmodule

// create a 1 clock cycle pulse whenever state changes
module statechangepulse_fsm(clk, reset, stateIn, stateChangePulse);
    input clk;
    input reset;
    input [3:0] stateIn;
    output stateChangePulse;

        reg[3:0] prevstate;
        reg stateChangePulse;

        //state register
        always@(posedge clk, posedge reset)
              if(reset)
                     begin
                            stateChangePulse <=0;
                            prevstate<=0;
                     end
              else
                     begin
                            stateChangePulse <= (prevstate != stateIn);
                            prevstate<=stateIn;
                     end


endmodule
```

```
////////////////////////////////////////////////////////////////////////////////
// Phil Amberg & Kevin Zielnicki
// 11/29/2006
//
// mux11.v - a few muxes and tristates used in other modules
////////////////////////////////////////////////////////////////////////////////
module mux11(d0, d1, s, y);

	input [10:0] d0;
	input [10:0] d1;
	input s;
	output [10:0] y;

	tristate t0(d0,~s,y);
	tristate t1(d1,s,y);


endmodule

module tristate(d,en,y);

	input [10:0] d;
	input en;
	output [10:0] y;

	assign y=en?d:11'bz;

	endmodule

module tristate8(d,en,y);

	input [7:0] d;
	input en;
	output [7:0] y;

	assign y=en?d:8'bz;

endmodule

module mux2(d0, d1, s, y);

	input [7:0] d0;
	input [7:0] d1;
	input s;
	output [7:0] y;

	tristate8 t0(d0,~s,y);
	tristate8 t1(d1,s,y);

endmodule

module mux4(d0, d1, d2, d3, s, y);

	input [7:0] d0;
	input [7:0] d1;
	input [7:0] d2;
	input [7:0] d3;
	input [1:0] s;
	output [7:0] y;

	wire [7:0] low,high;

	mux2 lowmux(d0,d1,s[0],low);
	mux2 highmux(d2,d3,s[0],high);
	mux2 finalmux(low,high,s[1],y);

endmodule
```

```
////////////////////////////////////////////////////////////////////////////////
// Phil Amberg & Kevin Zielnicki
// 11/19/2006
//
// delay.v - registers to delay signals
////////////////////////////////////////////////////////////////////////////////
module delay1cycle
        #(parameter width = 8)
        (clk, reset, signal, delayed);
        input clk;
        input reset;
   input [width-1:0] signal;
        output reg [width-1:0] delayed;

        always @(posedge clk, posedge reset)
                if(reset)
                        delayed<=0;
                else
                        delayed <= signal;

endmodule

module delay2cycles
        #(parameter width = 8)
        (clk, reset, signal, delayedTwice);
        input clk;
        input reset;
   input [width-1:0] signal;
        output reg [width-1:0] delayedTwice;

        reg [width-1:0] delayedOnce;

        always @(posedge clk,posedge reset)
                if(reset)
                        begin
                                delayedTwice<=0;
                                delayedOnce<=0;
                        end
                else
                        begin
                                delayedTwice <= delayedOnce;
                                delayedOnce <= signal;
                        end

endmodule
```

```
/////////////////////////////////////////////////////////////////////////////////
// Phil Amberg & Kevin Zielnicki
// 11/29/2006
//
// recieve_data.v - reads data from microcontroller over SSP
/////////////////////////////////////////////////////////////////////////////////
module recieve_data(clk,serialclk,reset,serialDataIn,dataOut,dataReadyPulse);

	input clk;
	input serialclk;
	input reset;
	input serialDataIn;
	output [7:0] dataOut;
	output dataReadyPulse;

	wire [7:0] parallelDataOut;
	wire dataReady;

	//recieve data
	sr8 inputShiftRegister(serialclk,reset,serialDataIn,parallelDataOut,dataReady);
	datareadypulse_fsm dataReady2Pulse(clk, reset, dataReady, dataReadyPulse);
	syncronizer syncSignals(clk,reset,parallelDataOut,dataOut);

endmodule
```

```
/////////////////////////////////////////////////////////////////////////////////
// Phil Amberg & Kevin Zielnicki
// 12/04/2006
//
// control_fsm.v - create addresses for memory based on input data from microcontroller
/////////////////////////////////////////////////////////////////////////////////
module control_fsm(clk, reset, status, dataReadyPulse, stateChangePulse, state,
gostartScreen, stopGame, gameStart,address,memswitch);

        input clk;
   input reset;
        input [3:0] status;
        input dataReadyPulse;
        input stateChangePulse;
        output [3:0] state;
        output gostartScreen;
        output stopGame;
        output gameStart;
        output [10:0] address;
        output memswitch;

        reg [3:0] state,nextstate;
        reg [4:0] count;
        reg [10:0] address,baseaddress;
        reg overflow;

        wire boardReady,scoreReady,hiScoreReady,nextPieceReady,levelReady;
        wire startScreen,playGame,endGame;

        //parameterize state encodings
        parameter ready=4'b0000;
        parameter writeboard=4'b0001;
        parameter writescore=4'b0010;
        parameter writenextpiece=4'b0011;
        parameter writelevel=4'b0100;
        parameter gotostartscreen=4'b0101;
        parameter startgame=4'b0110;
        parameter gameoverscreen=4'b0111;
        parameter switchmem=4'b1000;
        parameter writehiscore=4'b1001;

        //state register
        always@(posedge clk, posedge reset)
                if(reset)
                        state<=ready;
                else
                        state<=nextstate;

        //nextstate logic
        always@( * )
                case(state)
                        ready:begin
                                                if(boardReady) nextstate=writeboard;
                                                else if(scoreReady) nextstate=writescore;
                                                else if(nextPieceReady)
nextstate=writenextpiece;
                                                else if(levelReady) nextstate=writelevel;
                                                else if(startScreen)
nextstate=gotostartscreen;
                                                else if(playGame) nextstate=startgame;
                                                else if(endGame) nextstate=gameoverscreen;
                                                else if(hiScoreReady)
nextstate=writehiscore;
                                                else nextstate = ready;
                                        end
                        writeboard:             nextstate = (~boardReady ? switchmem :
writeboard);
                        writescore:             nextstate = (~scoreReady ? switchmem :
writescore);
                        writehiscore:   nextstate = (~hiScoreReady ? switchmem :
writehiscore);
```

```
			writenextpiece:nextstate = (~nextPieceReady ? switchmem :
writenextpiece);
			writelevel:		nextstate = (~levelReady ? switchmem :
writelevel);
			gotostartscreen:nextstate = (~startScreen ? ready :
gotostartscreen);
			startgame:		nextstate = (~playGame ? ready : startgame);
			gameoverscreen:nextstate = (~endGame ? ready : gameoverscreen);
			switchmem:		nextstate=ready;
			default:			nextstate=ready;
		endcase

	always@(posedge dataReadyPulse,posedge reset)
		if(reset|overflow)
			count<=0;
		else if(boardReady | nextPieceReady)
			count<=count+1;

	always@(posedge clk,posedge reset)
		if(reset)
			overflow<=(count==4'd0);

		else if(boardReady)
			overflow<=(count==4'd9);

		else if(nextPieceReady)
			overflow<=(count==4'd3);

		else
			overflow<=(count==4'd0);

	always@(posedge clk, posedge reset)
		if(reset)
			address<=0;
		else if(stateChangePulse&boardReady)
			address<=11'd271;
		else if(stateChangePulse&scoreReady)
			address<=11'd1054;
		else if(stateChangePulse&hiScoreReady)
			address<=11'd1411;
		else if(stateChangePulse&nextPieceReady)
			address<=11'd543;
		else if(stateChangePulse&levelReady)
			address<=11'd1440;
		else if(dataReadyPulse&boardReady)
			address<=address+(overflow ? 11'd55:11'd1);
		else if(dataReadyPulse&nextPieceReady)
			address<=address+(overflow ? 11'd61:11'd1);
		else if(dataReadyPulse)
			address<=address+1;



	assign gostartScreen=(state==gotostartscreen);
	assign stopGame=(state==gameoverscreen);
	assign gameStart=(state==startgame);

	assign memswitch=(state==switchmem);
	assign boardReady=(status==4'd1);
	assign scoreReady=(status==4'd2);
	assign nextPieceReady=(status==4'd3);
	assign levelReady=(status==4'd4);
	assign startScreen=(status==4'd5);
	assign playGame=(status==4'd6);
	assign endGame=(status==4'd7);
	assign hiScoreReady=(status==4'd8);

endmodule
```

```
/////////////////////////////////////////////////////////////////////////////////
// Phil Amberg & Kevin Zielnicki
// 11/29/2006
//
// gamestatefsm.v - keep track of the state of the game based on signals from control_fsm
/////////////////////////////////////////////////////////////////////////////////
module gamestatefsm(clk, reset, memrdy, start, stopGame, beginGame, state);

	input clk;
   input reset;
	input memrdy;
	input start;
   input stopGame;
   input beginGame;
   output [1:0] state;

	reg [1:0] state, nextstate;

	parameter startScreen=2'b00;
	parameter game1=2'b01;
	parameter game2=2'b10;
	parameter endScreen=2'b11;

	always@(posedge clk,posedge reset)
		if(reset)
			state<=startScreen;
		else
			state<=nextstate;

	always@( * )
		case(state)
			startScreen:nextstate = (beginGame ? game1 : startScreen);
			game1:begin
					if(stopGame) nextstate=endScreen;
					else if(memrdy) nextstate=game2;
					else nextstate = game1;
				end
			game2:begin
					if(stopGame) nextstate=endScreen;
					else if(memrdy) nextstate=game1;
					else nextstate = game2;
				end
			endScreen:nextstate = (start ? startScreen : endScreen);
			default:nextstate=startScreen;
		endcase


endmodule
```

```
///////////////////////////////////////////////////////////////////////////////
// Phil Amberg & Kevin Zielnicki
// 11/05/2006
//
// toplevelvga.v - top level module for generating VGA signals to display tetris graphics
///////////////////////////////////////////////////////////////////////////////
module toplevelvga(clkfx, reset, blocktype, HsyncDelayed, VsyncDelayed, BoardmapAddress,
sigR, sigG, sigB);

        input clkfx;
        input reset;
        input [7:0] blocktype;
    output HsyncDelayed;
    output VsyncDelayed;
        output [10:0]BoardmapAddress;
    output [2:0] sigR;
    output [2:0] sigG;
    output [1:0] sigB;

        wire DataValid, Hsync, Vsync;
        wire [13:0] TilemapAddress;
        wire [7:0] memVal,Blocktype;
        wire [3:0] blockRow,blockCol, blockRowDelayed,blockColDelayed;
        wire [4:0] boardRow;
        wire [5:0] boardCol;
        wire [10:0] BoardmapAddress;

        // some signals need to be delayed because of the 1 cycle delay in reading memory
        delay2cycles #(2) delayTimingSignals(clkfx, reset,
{Hsync,Vsync},{HsyncDelayed,VsyncDelayed});
        delay1cycle #(8) delayBlockRowCol(clkfx, reset, {blockRow, blockCol},
{blockRowDelayed, blockColDelayed});

        // create timing signals for VGA connection
        GenSyncSignals GenSyncs1(clkfx,reset,Hsync,Vsync,DataValid);

        // cycle through each 16x16 block in the 40x30 board
        RowFSM rows(reset,DataValid, boardRow, blockRow);
        ColFSM cols(clkfx,reset,DataValid, boardCol, blockCol);

        // create an address to get the current tile from the board memory
        GenerateBoardmapAddress GenBoardAddress(boardCol,boardRow,BoardmapAddress);

        // create an address to get the current pixel from the tile memory
        // need to use delayed values for block row & col because of delay in reading
memory
        GenerateTilemapAddress GenTileAddress(blocktype,
blockRowDelayed,blockColDelayed,TilemapAddress);

        // Memory modules
        blockmem1 blockMemory(TilemapAddress,clkfx, memVal); // pixel values for each
16x16 tile

        // output the current pixel value as a {Red,Green,Blue} VGA signal
        OutputSignal dataout(memVal,DataValid,sigR, sigG, sigB);

endmodule
```

```
///////////////////////////////////////////////////////////////////////////////
// Phil Amberg & Kevin Zielnicki
// 11/05/2006
//
// GenSyncSignals.v - create timing signals for VGA connection
// Based on code by Dan Chan, Nate Pinckney, & Dan Rinzler Spring 2005
///////////////////////////////////////////////////////////////////////////////
module GenSyncSignals(clk, reset, Hsync, Vsync, DataValid);
	input clk;
	input reset;
	output Hsync;
	output Vsync;
	output DataValid;

	 reg [9:0] colCount;
	 reg [9:0] rowCount;
	reg Hsync;
	 reg Vsync;
	 reg Hdata;
	 reg Vdata;

	always@(posedge clk, posedge reset) //Hsync block
		begin
			if(reset || (colCount == 10'd799) ) //reset colCount on reset or
end of line
				colCount<=0;
			else
				colCount<=colCount+1;  //else increment colCount
		end
	always@(posedge clk)
		begin
			if( (colCount >= 10'd7) && (colCount < 10'd103) )
				Hsync<=0;
			else
				Hsync<=1;
			if( (colCount >=10'd151) && (colCount < 10'd791) ) //lines of
displayed screen
				Hdata<=1;
			else
				Hdata<=0;
		end
	always@(negedge Hsync, posedge reset) //Vsync block
		begin
			if(reset || (rowCount==10'd524) ) //reset rowCount on reset or end
of field
				rowCount<=0;
			else
				rowCount<=rowCount+1; //else increment rowCount
		end
	always@(posedge clk)
		begin
			if( (rowCount >= 10'd2) && (rowCount < 10'd4) )
				Vsync<=0;
			else
				Vsync<=1;

			if( (rowCount >= 10'd37) && (rowCount < 10'd517) ) //lines of
displayed screen
				Vdata<=1;
			else
				Vdata<=0;
		end

	assign DataValid = Hdata & Vdata;
endmodule
```

```
////////////////////////////////////////////////////////////////////////////////
// Phil Amberg & Kevin Zielnicki
// 11/05/2006
//
// GenSignals.v - cycle through each 16x16 block in the 40x30 board
////////////////////////////////////////////////////////////////////////////////

// boardCol is the column of the current tile in the board
// blockCol is the column of the current pixel in the current tile
module ColFSM(clk, reset, DataValid, boardCol, blockCol);

	input clk;
	input reset;
	input DataValid;
	output [5:0] boardCol;
	output [3:0] blockCol;

	reg[3:0] blockCol, nextblockCol;
	reg[5:0] boardCol, nextboardCol;

	always @(posedge clk, posedge reset)
		if(reset)
			begin
				blockCol<=0;
				boardCol<=0;
			end
		else if(DataValid)
			begin
				blockCol<=nextblockCol;
				boardCol<=nextboardCol;
			end

	always @( * )
		case(blockCol)
			4'd15:
				begin
					nextblockCol=0;
					nextboardCol=(6'd39==boardCol)? 0:boardCol+1;
				end
			default:
				begin
					nextblockCol=blockCol+1;
					nextboardCol=boardCol;
				end
		endcase

endmodule

// boardRow is the column of the current tile in the board
// blockRow is the column of the current pixel in the current tile
module RowFSM(reset, DataValid, boardRow, blockRow);

	input reset;
	input DataValid;
	output [3:0] blockRow;
	output [4:0] boardRow;

	reg[3:0] blockRow, nextblockRow;
	reg[4:0] boardRow, nextboardRow;

	always @(posedge DataValid, posedge reset)
		if (reset)
			begin
				blockRow<=0;
				boardRow<=0;
			end
		else if(DataValid)
			begin
				blockRow<=nextblockRow;
				boardRow<=nextboardRow;
			end
```

```
	always @( * )
		case(blockRow)
			4'd15:
				begin
					nextblockRow=0;
					nextboardRow=(5'd29==boardRow)? 0:boardRow+1;
				end
			default:
				begin
					nextblockRow=blockRow+1;
					nextboardRow=boardRow;
				end
		endcase

endmodule

// create an address to get the current pixel from the tile memory
module GenerateTilemapAddress(BoardData,blockRow,blockCol,address);

	input [3:0] blockRow;
	input [3:0] blockCol;
	input [7:0] BoardData;
	output [13:0] address;

	assign address= {BoardData[5:0], blockRow, blockCol};

endmodule

// create an address to get the current tile from the board memory
module GenerateBoardmapAddress(boardCol,boardRow,BoardmapAddress);

	input [5:0] boardCol;
	input [4:0] boardRow;
	output [10:0] BoardmapAddress;

	assign BoardmapAddress={boardRow,boardCol};

endmodule

// output the current pixel value as a {Red,Green,Blue} VGA signal
module OutputSignal(signal,DataValid,R,G,B);

	input [7:0] signal;
	input DataValid;
	output [2:0] R,G;
	output [1:0] B;

	assign R={signal[7]&DataValid,signal[6]&DataValid,signal[5]&DataValid};
	assign G={signal[4]&DataValid,signal[3]&DataValid,signal[2]&DataValid};
	assign B={signal[1]&DataValid,signal[0]&DataValid};

endmodule
```

```
1   /* tetris.c
2    * C implementation of the game of tetris
3    * Receives input from a NES controller
4    * Outputs board data over SSP to an FPGA
5    *
6    * Author: Kevin Zielnicki <kzielnicki@hmc.edu>
7    * Date:   December 4, 2006
8    */
9
10  #include <p18f452.h>
11
12  /* Constants */
13
14  // status signals sent to FPGA over port D
15  #define BOARDDATA 0x10
16  #define SCOREDATA 0x20
17  #define PIECEDATA 0x30
18  #define LEVELDATA 0x40
19  #define STARTSCREEN 0x50
20  #define PLAYGAME 0x60
21  #define ENDGAME 0x70
22  #define HISCOREDATA 0x80
23  #define NULL 0x00
24
25  // internal representations for controller status
26  #define FALSE 0x00
27  #define TRUE 0x01
28  #define LEFT 0x02
29  #define RIGHT 0x03
30  #define UP 0x04
31  #define DOWN 0x05
32
33  /* Block Data Structure */
34  typedef struct block {
35      char buff[8];       // occupied tiles relative to (x,y) coordinate of block
36      char type;          // block type numbered 1-7
37      char x;             // x coordinate of block on screen
38      char y;             // y coordinate of block on screen
39      unsigned char bonus;// bonus points awarded for speeding up
40  } block;
41
42  /* Function Prototypes */
43  void main(void);
44  void isr(void);
45  void setState(unsigned char state);
46  void sendSig(unsigned char);
47  void clearBoard(void);
48  void clearScore(void);
49  void sendData(void);
50  void sendBoard(void);
51  void sendScore(void);
52  void sendHiScore(void);
53  void sendLevel(void);
54  void sendNextPiece(void);
55  void pollController(void);
56  void delay(void);
57  void longDelay(void);
58  void addScore(unsigned int);
59  void newBlock(void);
60  void showBlock(void);
61  void hideBlock(void);
62  void setBoard(char);
63  void rotate(char);
64  void moveBlock(char);
65  void advanceBlock(void);
66  char checkCollision(block);
67  void checkLines(void);
68  void clearLine(unsigned char);
69  void checkGameOver(void);
70
71  /* global variables */
72  #pragma idata chardata
73  block cBlock;                   // current block
74  block nBlock;                   // next block
75
```

```
 76  char level;                      // current level
 77  char lines;                      // lines completed in current level
 78  char gameover = FALSE;
 79  unsigned char random ;           // use for pseudorandomness
 80  unsigned char delayTime;         // determines how long to wait before advancing tetromino
 81
 82  char startPressed = FALSE;
 83  char selectPressed = FALSE;
 84  char aPressed = FALSE;
 85  char bPressed = FALSE;
 86  char direction = FALSE;
 87
 88  #pragma udata bigarray
 89  char boardData[220];             // game board, first 2 rows are off-screen buffer
 90
 91  #pragma idata smallarrays
 92  unsigned char score[7];
 93  unsigned char hiscore[7] = {0,0,0,0,0,0,0};
 94  char blocks[42] = {-2,0,-1,0,1,0,
 95                     -1,0,0,-1,1,-1,
 96                     1,0,0,-1,-1,-1,
 97                     -1,-1,-1,0,1,0,
 98                     -1,0,1,0,1,-1,
 99                     -1,0,1,0,0,-1,
100                     0,-1,-1,0,-1,-1};
101
102
103
104
105  #pragma code low_vector = 0x08
106  void low_interrupt(void) {
107      _asm
108          GOTO isr
109      _endasm
110  }
111
112  #pragma code
113  // set state to let FPGA know what kind of data we're sending
114  void setState(unsigned char state) {
115      PORTD = state;
116  }
117
118  // send a signal to the FPGA
119  void sendSig(unsigned char signal) {
120      while(SSPSTATbits.BF == FALSE) {
121                                    // wait for any previous write to finish
122      }
123      SSPSTATbits.BF = FALSE;       // clear buffer full bit (data is meaningless)
124
125      SSPBUF = signal;              // send signal to FPGA over SSP
126  }
127
128  // initialize board data to all 0's
129  void clearBoard(void) {
130      unsigned char i;
131
132      for(i=0; i<240; ++i) {
133          boardData[i] = 0;
134      }
135  }
136
137  // initialize score to all 0's
138  void clearScore(void) {
139      score[0] = 0;
140      score[1] = 0;
141      score[2] = 0;
142      score[3] = 0;
143      score[4] = 0;
144      score[5] = 0;
145      score[6] = 0;
146  }
147
148  // send board, score, level, and next piece to the FPGA
149  void sendData(void) {
150      sendBoard();
```

```
151     sendScore();
152     sendHiScore();
153     sendLevel();
154     sendNextPiece();
155 }
156 
157 // send the board data to the FPGA
158 void sendBoard(void) {
159     unsigned char i;
160 
161     setState(BOARDDATA);    // let FPGA know we're about to send the gameboard
162 
163     for(i = 20; i < 220; ++i) {
164         // send everything except the off-screen buffer
165         // 0's have to be sent as 0x2d, the background grey color
166         sendSig(boardData[i] ? boardData[i] : 0x2d);
167     }
168 
169     setState(NULL);         // make the FPGA switch its buffer and draw the updated board
170 }
171 
172 // send the score to the FPGA
173 void sendScore(void) {
174     char i;
175 
176     setState(SCOREDATA);    // let FPGA know we're about to send the score
177 
178     for(i = 6; i >= 0; --i) {
179         sendSig(score[i] + 34);
180     }
181 
182     setState(NULL);         // make the FPGA switch its buffer and draw the updated board
183 }
184 
185 // send the hiscore to the FPGA
186 void sendHiScore(void) {
187     char i;
188 
189     setState(HISCOREDATA);  // let FPGA know we're about to send the hiscore
190     //setState(SCOREDATA);  // let FPGA know we're about to send the score
191 
192     for(i = 6; i >= 0; --i) {
193         sendSig(hiscore[i] + 34);
194     }
195 
196     setState(NULL);         // make the FPGA switch its buffer and draw the updated board
197 }
198 
199 // send level to FPGA
200 void sendLevel(void) {
201     unsigned char digit1, digit2;
202 
203     setState(LEVELDATA);    // let FPGA know we're about to send the level
204 
205     digit1 = level/10;
206     digit2 = level%10;
207 
208     // don't send leading 0's for level (instead send empty tile)
209     sendSig(digit1 ? digit1 + 34 : 0);
210     sendSig(digit2 + 34);
211 
212     setState(NULL);         // make the FPGA switch its buffer and draw the updated board
213 }
214 
215 // send next piece to FPGA
216 void sendNextPiece(void) {
217     char i;
218     char toSend[8] = {0,0,0,0,0,0,0,0};
219 
220     setState(PIECEDATA);    // let FPGA know we're about to send the next piece
221 
222     // create a 2x4 array of information to send the FPGA based
223     // on the occupied locations in the next block buffer
224     for(i = 0; i < 8; i+=2) {
225         toSend[nBlock.buff[i] + 6 + nBlock.buff[i+1]*4] = nBlock.type;
```

```
226     }
227
228     // send the data to the FPGA
229     for(i = 0; i < 8; ++i) {
230         sendSig(toSend[i]);
231     }
232
233     setState(NULL);         // make the FPGA switch its buffer and draw the updated board
234 }
235
236 // poll NES controller
237 void pollController(void) {
238     unsigned char i,j;
239
240     random++;   // increment pseudorandom counter every time we poll controller
241
242     // direction must be set to false if no direction is pressed
243     direction = FALSE;
244
245     // first, pulse the latch to get the first input (A)
246     PORTE = 0x0;
247     PORTE = 0x1;
248     delay();
249     PORTE = 0x0;
250     aPressed = !(PORTC & 0x01);     // low signal on RC0 indicates button pressed
251     delay();
252
253     // a clock pulse gets us the next signal (B)
254     PORTE = 0x2;
255     delay();
256     PORTE = 0x0;
257     bPressed = !(PORTC & 0x01);
258     delay();
259
260     // next comes select
261     PORTE = 0x2;
262     delay();
263     PORTE = 0x0;
264     selectPressed = !(PORTC & 0x01);
265     delay();
266
267     // then start
268     PORTE = 0x2;
269     delay();
270     PORTE = 0x0;
271     startPressed = !(PORTC & 0x01);
272     delay();
273
274     // then up
275     PORTE = 0x2;
276     delay();
277     PORTE = 0x0;
278     if( (PORTC & 0x01) == 0x00 )
279         direction = UP;
280     delay();
281
282     // then down
283     PORTE = 0x2;
284     delay();
285     PORTE = 0x0;
286     if( (PORTC & 0x01) == 0x00 )
287         direction = DOWN;
288     delay();
289
290     // then left
291     PORTE = 0x2;
292     delay();
293     PORTE = 0x0;
294     if( (PORTC & 0x01) == 0x00 )
295         direction = LEFT;
296     delay();
297
298     // then right
299     PORTE = 0x2;
300     delay();
```

```
301     PORTE = 0x0;
302     if( (PORTC & 0x01) == 0x00 )
303         direction = RIGHT;
304     delay();
305     PORTE = 0x2;
306     delay();
307     PORTE = 0x0;
308 
309     longDelay();
310 }
311 
312 // helper function for pollControl, delays a set amount of time between clk signals
313 void delay(void) {
314     unsigned char i;
315 
316     for(i=0; i<3; ++i) {
317     }
318 }
319 
320 // helper function for pollControl, delays a longer time after polling controller
321 // to prevent bouncing and overly sensitive controls
322 void longDelay(void) {
323     unsigned char i,j;
324 
325     for(i=0; i<255; ++i) {
326         for(j=0; j<100; ++j) {
327         }
328     }
329 }
330 
331 // add value s to score array, scaled by the current level
332 void addScore(unsigned int s) {
333     unsigned char i;
334     unsigned int temp;
335 
336     s *= (level+1);                 // you earn more points on higher levels
337 
338     for(i = 0; i < 7; ++i) {        // break score up into individual digits to send to FPGA
339         s += score[i];
340         temp = s%10;
341         score[i] = temp;
342         s = s/10;
343     }
344 
345     sendScore();                    // send score to FPGA twice to update both buffers
346     sendScore();
347 
348     // if score > hiscore, replace hiscore with score
349     for(i = 6; i >= 0; --i) {
350         if(score[i] > hiscore[i]) {
351             hiscore[0] = score[0];
352             hiscore[1] = score[1];
353             hiscore[2] = score[2];
354             hiscore[3] = score[3];
355             hiscore[4] = score[4];
356             hiscore[5] = score[5];
357             hiscore[6] = score[6];
358             sendHiScore();          // send hi score to FPGA twice to update both buffers
359             sendHiScore();
360             break;
361         } else if(hiscore[i] > score[i]) {
362             break;
363         }
364     }
365 }
366 
367 // generate a new block at random
368 void newBlock(void) {
369     char i;
370 
371     nBlock.type = random % 7; // get a block type 0-6
372     nBlock.buff[0] = 0;       // all types have origin occupied
373     nBlock.buff[1] = 0;
374     for(i = 0; i < 6; i+=2) { // assign positions from block definitions
375         nBlock.buff[i+2] = blocks[6*nBlock.type + i];
```

```
376         nBlock.buff[i+3] = blocks[6*nBlock.type + i + 1];
377     }
378 
379     nBlock.type = nBlock.type + 1;  // want block types 1-7, not 0-6
380     nBlock.x = 5;             // start block in middle/top of the board
381     nBlock.y = 1;
382     nBlock.bonus = 0;         // blocks have no bonus score when created
383 }
384 
385 // display contents of cBlock on the board
386 void showBlock(void) {
387     setBoard(cBlock.type);
388 }
389 
390 // remove contents of cBlock on the board
391 void hideBlock(void) {
392     setBoard(0);
393 }
394 
395 // helper function for hideBlock and showBlock, sets tiles referred to in cBlock to type
396 void setBoard(char type) {
397     unsigned char i, x, y;
398 
399     for(i=0; i < 8; i += 2) {
400         x = cBlock.x + cBlock.buff[i];
401         y = cBlock.y + cBlock.buff[i+1];
402         boardData[10*y + x] = type;
403     }
404 }
405 
406 // rotate current block left or right
407 void rotate(char direction) {
408     char i, tmp;
409     block tBlock;
410 
411     if(cBlock.type == 0x07)                 // can't rotate square piece
412         return;
413 
414     tBlock = cBlock;                        // operate on a copy of current block
415 
416     hideBlock();
417 
418     // swap x and y coordinates, negating one or the other depending on direction
419     for(i=0; i < 8; i += 2) {
420         tmp = tBlock.buff[i];
421         if(direction == RIGHT) {                // rotate right (CW)
422             tBlock.buff[i] = -tBlock.buff[i+1]; // x = -y
423             tBlock.buff[i+1] = tmp;             // y = x
424         } else {                                // rotate left (CCW)
425             tBlock.buff[i] = tBlock.buff[i+1];  // x = y
426             tBlock.buff[i+1] = -tmp;            // y = -x
427         }
428     }
429 
430     if(checkCollision(tBlock) == FALSE)
431         cBlock = tBlock;                        // if no collisions, adopt the modified block
432 
433     showBlock();
434 }
435 
436 // move the current block left or right on screen
437 void moveBlock(char direction) {
438     char i;
439     block tBlock;
440 
441     tBlock = cBlock;                        // operate on a copy of current block
442 
443     hideBlock();
444 
445     // add 1 to x coordinate to move right, subtract 1 to move left
446     if(direction == RIGHT) {                // move right
447         tBlock.x += 1;                      // x = x + 1
448     } else {                                // move left
449         tBlock.x -= 1;                      // x = x - 1
450     }
```

```
451
452     if(checkCollision(tBlock) == FALSE)
453         cBlock = tBlock;                        // if no collisions, adopt the modified block
454
455     showBlock();
456 }
457
458 // advance the current block down the screen, create a new block if it hits bottom
459 void advanceBlock(void) {
460     char i;
461     block tBlock;
462
463     tBlock = cBlock;                            // operate on a copy of current block
464
465     hideBlock();
466
467     // positive y is down -- add 1 to y to advance
468     tBlock.y += 1;                              // y = y + 1
469
470     if(checkCollision(tBlock) == FALSE)
471         cBlock = tBlock;                        // if no collisions, adopt the modified block
472     else {
473         showBlock();                            // otherwise, we're done with this block
474         addScore(cBlock.bonus);                 // accumulate any bonus points in score
475         cBlock = nBlock;                        // next block becomes current
476         newBlock();                             // make a new next block
477         sendNextPiece();                        // send next piece to FPGA twice to update both
478         sendNextPiece();
479         checkLines();                           // see if this completed any lines
480         checkGameOver();                        // see if we've lost the game :'(
481     }
482
483     showBlock();
484 }
485
486 // check if block is in a valid position, return TRUE if it collides with something
487 char checkCollision(block tBlock) {
488     unsigned char i, x, y;
489
490     // blocks collide if any (x,y) is out of bounds, or on top of another block
491     for(i=0; i < 8; i += 2) {
492         x = tBlock.x + tBlock.buff[i];
493         y = tBlock.y + tBlock.buff[i+1];
494         if(x < 0 || x >= 10 || y < 0 || y >= 22)
495             return TRUE;
496         if(boardData[10*y + x] != 0)
497             return TRUE;
498     }
499     return FALSE;
500 }
501
502 // remove any completed lines in the board and give the player points
503 void checkLines(void) {
504     unsigned char i, j, count, newLines;
505     int delayTemp;
506
507     newLines = 0;
508     for(i=2; i < 22; ++i) {
509         count = 0;
510         for(j=0; j < 10; ++j) {
511             if(boardData[10*i + j] != 0)
512                 count++;
513         }
514         if(count == 10) {
515             lines++;
516             newLines++;
517             clearLine(i);
518         }
519     }
520
521     // if more than 10 lines were completed, advance to next level
522     if(lines >= 10) {
523         lines = lines%10;
524         level++;
525         sendLevel();                    // send level to FPGA twice to update both buffers
```

```
526         sendLevel();
527         delayTemp = delayTime;
528         delayTime = (delayTemp*9)/10;
529     }
530 
531     // add points to score based on # of lines completed
532     switch(newLines)
533     {
534         case 1 : addScore(40);
535                  break;
536         case 2 : addScore(100);
537                  break;
538         case 3 : addScore(300);
539                  break;
540         case 4 : addScore(1200);
541                  break;
542     }
543 }
544 
545 // helper function for checkLines, removes a given row by shifting previous rows down
546 void clearLine(unsigned char line) {
547     unsigned char i, j;
548 
549     for(i = line; i >= 2; --i) {
550         for(j=0; j < 10; ++j) {
551             boardData[10*i + j] = boardData[10*(i-1) + j];
552         }
553     }
554 }
555 
556 // set the gameover global variable if we lost the game by going off the top of the screen
557 void checkGameOver(void) {
558     unsigned char i;
559 
560     for(i = 0; i < 10; ++i) {
561         if(boardData[i+10] != 0)
562             gameover = TRUE;
563     }
564 }
565 
566 
567 void main(void) {
568     unsigned char tmrHtmp, tmrLtmp, i, j, rotated, moved;
569 
570     // configure I/O
571     TRISD = 0x00;
572     TRISE = 0x00;       // RE0 used for controller latch, RE1 for controller clk
573     TRISC = 0x11;       // SPI on RC4, controller input on RC0, all others can be outputs
574 
575     // configure SPI for serial communication with FPGA
576     SSPCON1 = 0x20;     // configure PIC as master, no collision or overflow
577     SSPSTAT = 0x40;     // transmit data on rising edge of SCK
578 
579     SSPBUF = 0x00;      // send some dummy data so SSPSTATbits.BF gets initialized to true
580 
581     // timer0 determines when piece advances a line
582     T0CON = 0x87;       // 16-bit timer 0, 1:256 prescale
583 
584 
585     while(1) {
586         setState(STARTSCREEN);          // tell FPGA to show start screen
587         setState(NULL);
588 
589         while(startPressed == FALSE) {  // wait for user to press start
590             pollController();
591         }
592         // delay until the user lets go of pause button
593         while(startPressed == TRUE) {
594             pollController();
595         }
596 
597         random = TMR0L;                 // initialize pseudrandom counter
598 
599         setState(PLAYGAME);             // let's play TETRIS!
600         setState(NULL);
```

```
601
602         TMR0H = 0;                      // initialize timer
603         TMR0L = 0;
604
605         level = 0;                      // start at level 0
606         lines = 0;                      // start w/ no lines completed
607         delayTime = 0x6F;               // initial speed of pieces falling
608         clearScore();                   // start with 0 points
609         clearBoard();                   // play on an empty board
610         newBlock();                     // start the game off with fresh new blocks
611         cBlock = nBlock;
612         random = random << 3;           // shift random left to get a different block
613         newBlock();
614
615         sendData();                     // send initial data to FPGA twice to update both buffe:
616         sendData();
617
618         while(gameover == FALSE) {      // play until you lose
619             pollController();
620
621             if(aPressed == TRUE) {      // a button rotates right (CW)
622                 if(rotated == FALSE) {  // don't rotate if we rotated last cycle
623                     rotate(RIGHT);
624                     rotated = TRUE;
625                 }
626             } else if(bPressed == TRUE) {   // b button rotates left (CCW)
627                 if(rotated == FALSE) {  // don't rotate if we rotated last cycle
628                     rotate(LEFT);
629                     rotated = TRUE;
630                 }
631             } else {
632                 rotated = FALSE;        // neither a or b pressed, we didn't try to rotate
633             }
634
635             if((direction == LEFT || direction == RIGHT) & moved == FALSE) {
636                 moveBlock(direction);   // move in the appropriate direction
637                 moved = TRUE;           // move left / right at 1/2 maximum speed
638             } else {
639                 moved = FALSE;
640             }
641
642             if(direction == DOWN) {
643                 cBlock.bonus++;         // award bonus points for fast movement
644                 advanceBlock();         // advance the block down the screen if down is pressed
645                 random += tmrLtmp;      // increment pseudorandom using clock
646                 TMR0H = 0;              // reset delay timer
647                 TMR0L = 0;
648             }
649
650             tmrLtmp = TMR0L;            // save timer values in case user pauses
651             tmrHtmp = TMR0H;
652             if(startPressed == TRUE) {  // pause the game if the user presses start
653                 // delay until the user lets go of pause button
654                 while(startPressed == TRUE)
655                     pollController();
656                 do {
657                     pollController();
658                 } while(startPressed == FALSE);
659                 // delay until the user lets go of pause button
660                 while(startPressed == TRUE)
661                     pollController();
662             }
663             TMR0H = tmrHtmp;            // reload timer values in case user pauses
664             TMR0L = tmrLtmp;
665
666             if(tmrHtmp > delayTime) {
667                 advanceBlock();         // advance the block if enough time has elapsed
668                 random += tmrLtmp;      // increment pseudorandom using clock
669                 TMR0H = 0;              // reset delay timer
670                 TMR0L = 0;
671             }
672
673             sendBoard();                // update board on FPGA
674         }
675
```

```
676
677         setState(ENDGAME);              // you LOSE!
678         setState(NULL);
679
680         gameover = FALSE;
681         while(startPressed == FALSE) {  // wait for user to press start
682             pollController();
683         }
684         // delay until the user lets go of pause button
685         while(startPressed == TRUE) {
686             pollController();
687         }
688     }
689 }
690
691
692 #pragma interruptlow isr
693 void isr(void) {
694     // we shouldn't have any interrupts
695     PORTD = 0x0F;
696 }
697
```