Simon Game with VGA

Final Project Report
December 11, 2009
E155: Microprocessor-Based Systems

Becky Glick and Max Wishman

Abstract

This paper outlines the hardware and software utilized in designing an interactive Simon game with video output. Simon is a memory game that generates color and sound patterns for the player to try and memorize and repeat. In creating the simon game, we used a PIC18F452 microcontroller and a Xilinx Spartan-3 FPGA mounted on the HarrisBoard 2.0. We also used a Dell VGA monitor, lighted buttons, an 8Ω speaker, as well as various other pieces of hardware. We programmed the PIC microcontroller using C to implement all the normal features of a handheld Simon game and programmed the FPGA using Verilog to generate the signals that control a VGA monitor. We successfully demonstrated a working unit on Projects Day and implemented a feature beyond our original proposal that displays the players score in binary at the end of the game.

Introduction

Simon games challenge memory retention capacity by generating a sequence of colors for a player to repeat. After each successful series of presses, “Simon” repeats the list followed by an additional random color. Game play continues until the player makes a mistake.

This project fulfills the same objectives using a Xilinx Spartan-3 FPGA and a PIC microcontroller. The PIC pseudo-randomly generates a color that, via a VGA interface created by the FPGA, brightens a corresponding quadrant on a monitor.

Figure 1: CRT Display Layout: The monitor is divided into four quadrants, each color corresponding to one button on the input pad. The color pattern is presented to the player by increasing the brightness of the appropriate quadrant.

After the player inputs the correct color by pressing the appropriate key on the pad, “Simon” repeats the color pattern from the beginning.

Each game instruction and key press is coupled with a distinctive tone. Upon an incorrect key press, the speaker plays an ominous melody line, indicating the end of the game. After the player loses the game, the total score (the length of the longest sequence of correctly returned colors) is displayed in binary on an LED array.
The interaction between the main components is illustrated in the following diagram.

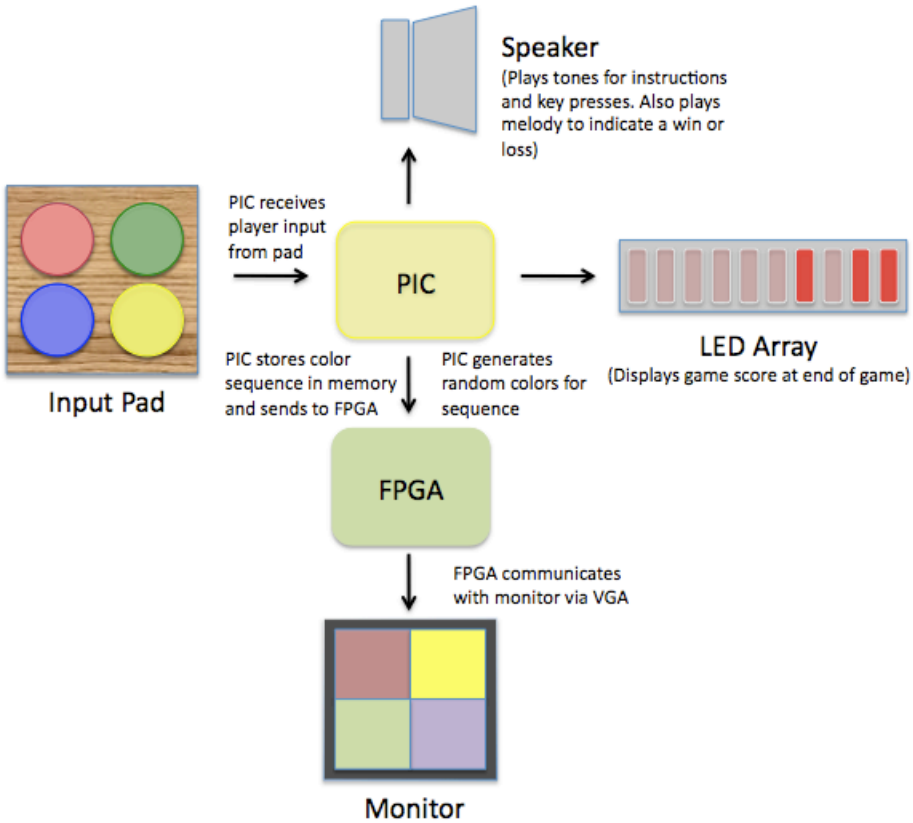


Figure 2: Block Diagram illustrating interactions between hardware

Schematics

Four off momentary LED switches are connected to the PIC at Port C as user inputs. Each switch has an RC de-bouncing circuit consisting of two 1kΩ resistors and one 10μF electrolytic capacitor. The circuit acts as a low-pass filter to ignore high frequency bounces between button contacts.

An 8Ω speaker is connected to the PIC at Port D as an output. The 1-bit signal sent to the speaker is amplified using an LM386 audio amplifier chip in an RC circuit with a gain on the order of about 50. The other speaker terminal is grounded.

R[2:0], g[2:0], and b[2:0] outputs from the FPGA are connected to a 510Ω resistor, a 270Ω resistor, and a 130Ω resistor. These binary weighted ladders convert each of the 3-bit values into a 0-0.7V voltage required for VGA.

A 1kΩ potentiometer provides an analog voltage from 0-3.3V in order to create pseudo-random games. After an analog to digital conversion in the PIC, the eight most significant bits of the digital signal seed rand in the C standard library. This allows a player to repeat the same pseudo-randomly generated game as many times as desired. When the player wants a new game, he or she can turn the potentiometer to create a new seed.

Microcontroller Design

The PIC microcontroller runs the Simon game. It takes inputs of the four buttons, the clock, and a potentiometer's analog voltage and outputs to four LEDs and a speaker. Game play is broken down by turns, and each turn involves two stages: the instructions portion and the user input portion. For a given turn, the PIC first plays a number of instructions from the array, and then waits for the user to repeat the pattern, checking each note the user inputs with the original array of instructions. If the user correctly replicates the pattern, the process repeats with one additional instruction appended to the end of the instruction sequence. If the user misses a note, the PIC plays an ominous melody, resets the turn counter, and displays the player's score. The following diagram illustrates of all of the PIC functions.

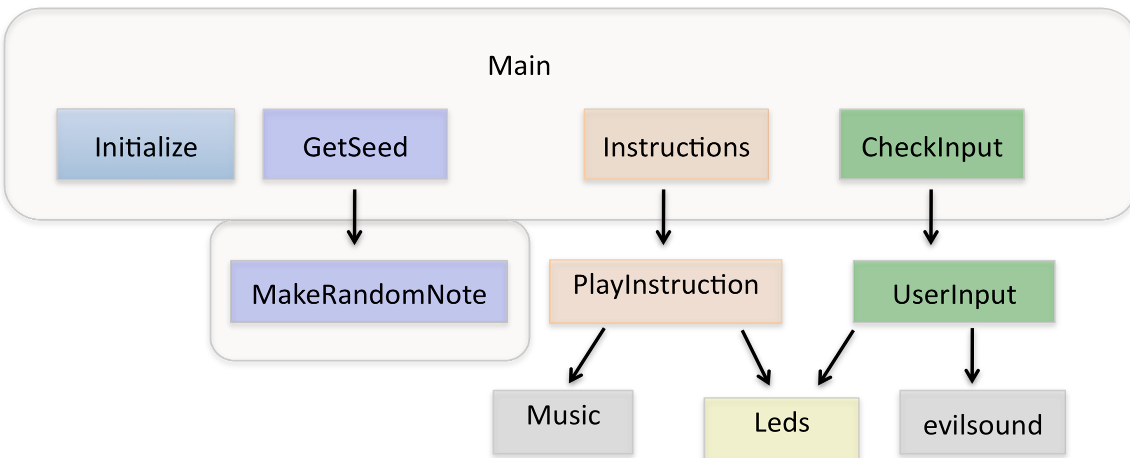


Figure 3: PIC Block Diagram: Each block represents an individual function in C code

Main:

The Main function runs once per turn, calling the functions Initialize, GetSeed, MakeRandomNote, Instructions, and CheckInput. Although the Initialize and GetSeed functions are only needed at the start of a new game, they are called from Main every turn. Instructions and CheckInput call various sub-functions including Leds, Music, KeyPressMusic, UserInput, PlayInstruction, EvilSound, and DisplayScore. Most of these functions involve the two global variables "Notes" and "Turn". "Notes" is a list of up to 50 instructions, stored as note periods for the speaker. "Turn" is a turn counter that starts at zero and increments at the end of each round before the Main function is called to begin the next set of instructions.

Initialize:

The Initialize function configures the two timers, the A/D converter (ADC), and the I/O ports by writing values to the configuration registers and I/O tri-state registers. Two timers are used to drive the speaker. Timer 0 is used for note duration and

Timer 1 is used for period duration. The ADC converts an arbitrary input voltage between 0V and 3.3V into a digital value to seed a pseudo-random number generator. I/O Port A is set as "input" to read in the analog voltage, I/O Port C is set as "input" to read in signals from each of the four buttons, and I/O Port D is set as "output" to light up the LEDs and drive the speaker.

<u>GetSeed:</u>
The GetSeed function starts the Analog-to-Digital conversion and waits in a loop until the conversion is finished. Then, the most significant bits are fed into srand. Without srand, rand assumes a seed of 0 and will play the same game after a reset. The MakeRandomNote function calls rand to get the next number from the pseudo-random array. The integer output from rand is then sorted into one of four instruction notes and adds that new instruction to the global variable "Notes."

<u>Instructions:</u>
The Instructions function begins the sequence of colors and sounds given to the user at the start of each turn. It creates a time delay between each instruction using empty FOR loops. PlayInstruction is called for each instruction in the global array "Notes." The number of iterations of this loop is based on the global value "Turn." This plays the sequence from the beginning and adds one more instruction to the end of the sequence each turn.

<u>PlayInstruction:</u>
PlayInstruction takes, as an argument, an index that accesses the appropriate instruction from "Notes." The function then lights up the corresponding LED and plays the appropiate note by calling Leds and Music.

<u>CheckInput</u>:
Once the Instructions function is finished, Main calls CheckInput. CheckInput waits for the player to repeat the color pattern. As the player presses each button, each input is checked against the global array "Notes." The UserInput function polls the buttons to determine which one is pressed. Since the RC debouncing circuit drives the button output high, key presses output a low. Therefore, the CheckInput function scans Port C for a zero.

<u>EvilSound:</u>
If the CheckInput function detects a mismatch in the user's playback of the color pattern, then the function EvilSound is called. This function calls Music to play an ominous melody that indicates a game over. EvilSound calls the function DisplayScore to print the turn counter value in binary to Port D, which displays the score on the LED array. From EvilSound, if the user presses one of the four buttons, the turn counter resets and the game restarts.

FPGA Design

The VGA standard uses an “active pixel” to display an image on a monitor. The active pixel scans through the rows at 25MHz and colors each pixel individually. This requires sending red, green, blue, hsync and vsynch signals. The FPGA design is constructed from two main modules, sync_controller and rgb_controller, to display four colored quadrants that brighten with instructions and user input.

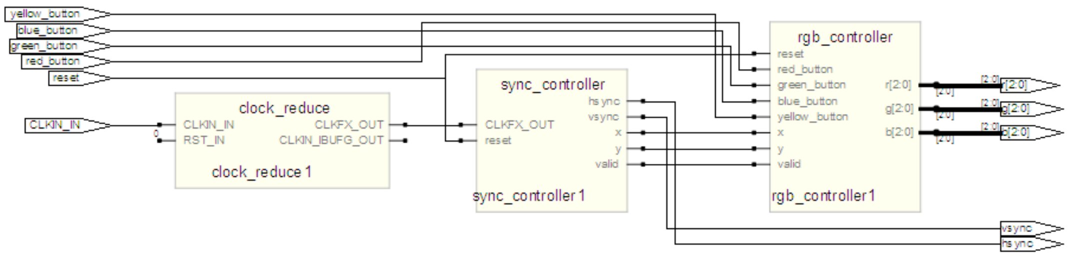


Figure 4: FPGA Block Diagram: Each block represents a module in verilog.

Generating Sync Signals:

Xilinx’s Digital Clock Manager (DCM) can be used to approximate the VGA internal clock frequency standard of 25.175MHz by multiplying HarrisBoard’s 40MHz clock by the integer ratio 5/8. Hsync controls when the active pixel moves to the next line and VSync controls when the active pixel starts back to the top left corner, beginning a new screen. With a pixel clock of 25MHz, Hsynch must run at 31.47kHz. This is based on a total of 800 pixels per row (including front and back porches). Vsynch must run at 59.94MHz for a total of 525 rows. Sync_controller also generates two signals, x and y, that act as pixel counters.

Generating RGB Signals:

The rgb_controller module takes the outputs x and y from synch_controller to track the horizontal and vertical coordinates of the active pixel. Rgb_controller then uses this location information to determine 9 bits of color with 3 bits red, 3 bits green, and 3 bits blue. The screen is divided into four equal sized quadrants with colors that are dependent on user input. When an instruction is given or key is pressed, the color value for the corresponding quadrant changes to represent a brighter shade. The nine bits of color pass through three binary weighted DACs, one for each color, to create the r[2:0], g[2:0], and b[2:0] analog signals between 0 and 0.7V. All area outside the four squares is kept black as a background.

Results

This project succeeded in meeting the specifications described in the project proposal. The game reliably challenges the player and provides two different visual interfaces, the VGA display as well as the LED buttons. Additionally, the sounds corresponding to each instruction and key press further enforce the pattern. The VGA display creates a more interactive gaming environment where observers can also participate in strengthening their memory.

The most challenging aspects of the project involved interfacing with VGA. Problems slowing the internal clock from 40 to 25MHz and debugging the vsync and hsync controllers could have been simplified had we discovered that the MicroToys tutorial described as "connect the PIC to VGA" concerned interfacing with an FPGA as opposed to a PIC microcontroller. Debugging problems in VGA required learning more about operating an oscilloscope because without generating signals matching the VGA standard, no image appears on the monitor.

An additional complication was due to the internal capacitance of the breadboard. Wiring the VGA hardware to a breakout board resolved these problems.

Instead of completing the stretch goal of displaying the final score on a seven-segment display, we chose to produce the score in binary on the HarrisBoard's built-in LED array. This was due to a limited number of pins remaining on the board as well as the benefit of requiring less hardware. Additionally, a binary representation of the final score is consistent with the digital nature of the project.

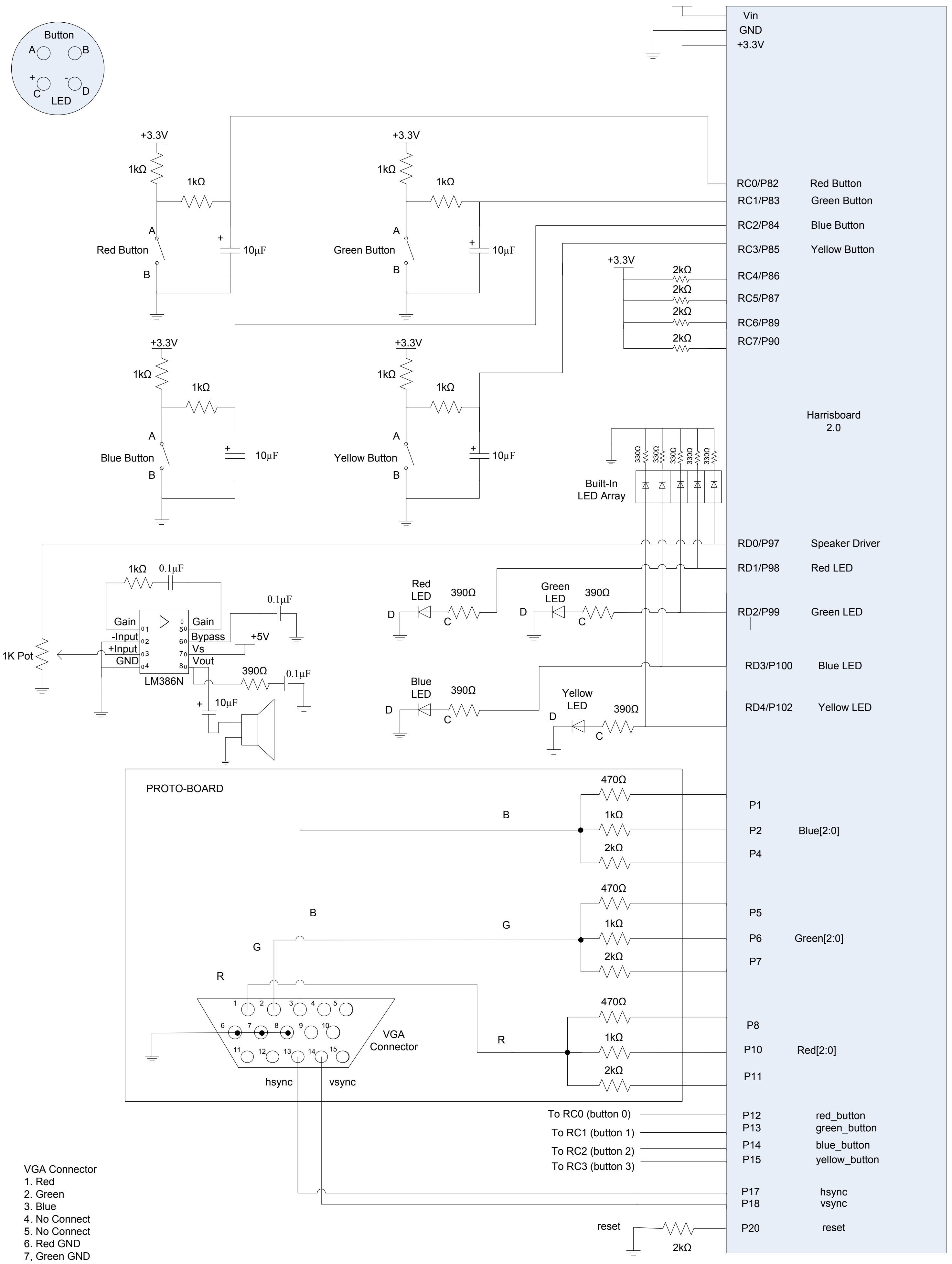
References

[1] MicroToys VGA, `http://www4.hmc.edu:8001/Engineering/microtoys/`

Parts List

Part	Source	Vendor Part #	Quantity	Price
Switch MOM-OFF Illum (Yellow)	Digikey	67-149-ND	2	$3.48
Switch MOM-OFF Illum (Red)	Digikey	67-150-ND	2	$3.48
Switch MOM-OFF Illum (Green)	Digikey	67-148-ND	2	$3.48
Switch MOM–OFF Illum (Blue)	Digikey	67-151-ND	2	$6.45

All other parts used could be found in the Lab.



VGA Connector
1. Red
2. Green
3. Blue
4. No Connect
5. No Connect
6. Red GND
7, Green GND
8. Blue GND
9. No Connect
10. No Connect
11. No Connect
12. No Connect
13. Hsync
14. Vsync
15. No Connect

```verilog
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company: Harvey Mudd College
// Engineer: Becky Glick and Max Wishman
//
// Create Date:    19:13:06 11/06/2009
// Design Name:
// Module Name:    vga
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module vga(
    input CLKIN_IN,
    input reset,
    input  red_button,
    input  green_button,
    input  blue_button,
    input  yellow_button,
    output [2:0] r,
    output [2:0] g,
    output [2:0] b,
    output hsync,
    output vsync
    );

   // Digital Clock Manager: Reduces 40MHz clk to 25MHz clock
   clock_reduce clock_reduce1(
    .CLKIN_IN(CLKIN_IN),               // 40MHz clk
    .RST_IN(RST_IN),
    .CLKFX_OUT(CLKFX_OUT),             // 25MHz clk
    .CLKIN_IBUFG_OUT(CLKIN_IBUFG_OUT)
    );

   wire [9:0] x;
   wire [9:0] y;
   wire valid;

   // Controls hsync and vsync pins: Position on screen
   sync_controller sync_controller1(CLKFX_OUT, reset, hsync, vsync, x, y, valid);

   // Controls R, G, and B pins: Pixel color
   rgb_controller rgb_controller1(reset, red_button, green_button,
      blue_button, yellow_button, x, y, valid, r, g, b);


endmodule

```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Harvey Mudd College
// Engineer: Becky Glick and Max Wishman
//
// Create Date:    14:42:26 11/24/2009
// Design Name:
// Module Name:    sync_controller
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments: Creates hsynch and vsynch to drive VGA
//
//////////////////////////////////////////////////////////////////////////////////
module sync_controller(
    input CLKFX_OUT,
    input reset,
    output hsync,
    output vsync,
    output [9:0] x,
    output [9:0] y,
    output valid
    );

   parameter maxcol = 800;                 //800 pixles per row
   parameter maxrow = 525;                 //575 pixles per column
   parameter HSYNC = 7'b1100000;           //witdh when cannot write to rows
   parameter VSYNC = 2;                    //height when we can write to cols
   parameter hbporch = 6'b101000;          //pixles off left side of screen
   parameter width = 640;                  //visible pixles per row
   parameter vbporch = 25;                 //pixles off of top of screen
   parameter height = 480;                 //visible pixles per column

   reg [9:0] row, col;                     //10 bit wire for row and col

   always@(posedge CLKFX_OUT, posedge reset)
         if (reset)                        //move cursor to upper left corner
            begin
               row = 0;
               col = 0;
            end
         else
            begin
               col = col + 1;              //move cursor to right by one unit per
                                           //clock cycle
               if (col == maxcol)          //once at the end of row
                  begin
                     col = 0;              //move cursor back to left of screen
                     row = row + 1;        //shift down one row
                     if (row == maxrow)    //once at bottom of screen
                        row = 0;           //go back to top!
                  end
            end

   assign hsync = (col > HSYNC);           //determines hsync frequency
```

```
        assign vsync = (row > VSYNC);           //determines vsync frequency
        assign x = (col - HSYNC - hbporch);     //x(0) at left side of writeable area
        assign y = (row - VSYNC - vbporch);     //y(0) starts at top of writable area
        //valid is high in writable area
        assign valid = ((x < height) & (y < width) & (x > 0) & (y > 0));

    endmodule
```

```verilog
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company: Harvey Mudd College
// Engineer: Becky Glick and Max Wishman
//
// Create Date:    14:58:12 11/24/2009
// Design Name:
// Module Name:    rgb_controller
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments: Creates color information for active pixel
//
////////////////////////////////////////////////////////////////////////////////
module rgb_controller(
    input reset,
    input  red_button,
    input  green_button,
    input  blue_button,
    input  yellow_button,
    input [9:0] x,
    input [9:0] y,
    input valid,           // high if hsync and vsync in writable area
    output reg [2:0] r,    // to be converted to analog for VGA color red
    output reg [2:0] g,    // to be converted to analog for VGA color green
    output reg [2:0] b     // to be converted to analog for VGA color blue
    );
                           //    rrrgggbbb
   parameter blue =           9'b010010011;
   parameter red =            9'b111010010;
   parameter green =          9'b1010110100;
   parameter yellow =         9'b001001100;
   parameter blue_bright =    9'b000000111;
   parameter red_bright =     9'b111000000;
   parameter green_bright =   9'b000011000;
   parameter yellow_bright =  9'b111111000;
   parameter border =         9'b000000000;
   parameter white =          9'b111111111;

   reg [8:0] color;


   always@(*)
      begin
         if ((valid))
            if (x < 240)                     // left half of screen
               if (y < 260)                  // top-left region
                  if (red_button == 1'b1)
                     color <= red_bright;    // red region "bright" on keypress
                  else
                     color <= red;           // assign red color to region
               else if (y > 250)             // bottom-left region
                  if (blue_button == 1'b1)
                     color <= blue_bright;   // blue region "bright" on keypress
                  else
```

```
                     color <= blue;          // assign blue color to region
               else
                  color <= border;           // black region around game board
            else                             // right half of screen
               if (y < 260)                  // top-right region
                  if (green_button == 1'b1)
                     color <= green_bright;  // green region "bright" on keypress
                  else
                     color <= green;         // assign green color to region
               else if (y > 259)             // bottom-right region
                  if (yellow_button == 1'b1)
                     color <= yellow_bright; // yellow region "bright" on keypress
                  else
                     color <= yellow;        // assign yellow color to region
               else
                  color <= border;           // black region around game board
         else
            color <= border;                 // black region around game board
      end

   always @(*)
      begin                                  // assign appropiate bits of color
      r <= color[8:6];                       // to r,g, and b
      g <= color[5:3];
      b <= color[2:0];
      end

endmodule
```

```
////////////////////////////////////////////////////////////////////////////////
// Copyright (c) 1995-2008 Xilinx, Inc.  All rights reserved.
////////////////////////////////////////////////////////////////////////////////
//   ____  ____
//  /   /\/   /
// /___/  \  /    Vendor: Xilinx
// \   \   \/     Version : 10.1.03
//  \   \         Application : xaw2verilog
//  /   /         Filename : clock_reduce.v
// /___/   /\     Timestamp : 12/01/2009 19:48:23
// \   \  /  \
//  \___\/\___\
//
//Command: xaw2verilog -intstyle
C:/glickwishman/final_project/vga/vga/clock_reduce.xaw -st clock_reduce.v
//Design Name: clock_reduce
//Device: xc3s400-5tq144
//
// Module clock_reduce
// Generated by Xilinx Architecture Wizard
// Written for synthesis tool: SynplifyPro
// Period Jitter (unit interval) for block DCM_INST = 0.03 UI
// Period Jitter (Peak-to-Peak) for block DCM_INST = 1.23 ns
`timescale 1ns / 1ps

module clock_reduce(CLKIN_IN,
                    RST_IN,
                    CLKFX_OUT,
                    CLKIN_IBUFG_OUT,
                    LOCKED_OUT);

    input CLKIN_IN;
    input RST_IN;
   output CLKFX_OUT;
   output CLKIN_IBUFG_OUT;
   output LOCKED_OUT;

   wire CLKFX_BUF;
   wire CLKIN_IBUFG;
   wire GND_BIT;

   assign GND_BIT = 0;
   assign CLKIN_IBUFG_OUT = CLKIN_IBUFG;
   BUFG CLKFX_BUFG_INST (.I(CLKFX_BUF),
                         .O(CLKFX_OUT));
   IBUFG CLKIN_IBUFG_INST (.I(CLKIN_IN),
                           .O(CLKIN_IBUFG));
   DCM DCM_INST (.CLKFB(GND_BIT),
                 .CLKIN(CLKIN_IBUFG),
                 .DSSEN(GND_BIT),
                 .PSCLK(GND_BIT),
                 .PSEN(GND_BIT),
                 .PSINCDEC(GND_BIT),
                 .RST(RST_IN),
                 .CLKDV(),
                 .CLKFX(CLKFX_BUF),
                 .CLKFX180(),
                 .CLK0(),
                 .CLK2X(),
                 .CLK2X180(),
                 .CLK90(),
                 .CLK180(),
                 .CLK270(),
```

```
                 .LOCKED(LOCKED_OUT),
                 .PSDONE(),
                 .STATUS());
   defparam DCM_INST.CLK_FEEDBACK = "NONE";
   defparam DCM_INST.CLKDV_DIVIDE = 2.0;
   defparam DCM_INST.CLKFX_DIVIDE = 8;
   defparam DCM_INST.CLKFX_MULTIPLY = 5;
   defparam DCM_INST.CLKIN_DIVIDE_BY_2 = "FALSE";
   defparam DCM_INST.CLKIN_PERIOD = 25.000;
   defparam DCM_INST.CLKOUT_PHASE_SHIFT = "NONE";
   defparam DCM_INST.DESKEW_ADJUST = "SYSTEM_SYNCHRONOUS";
   defparam DCM_INST.DFS_FREQUENCY_MODE = "LOW";
   defparam DCM_INST.DLL_FREQUENCY_MODE = "LOW";
   defparam DCM_INST.DUTY_CYCLE_CORRECTION = "TRUE";
   defparam DCM_INST.FACTORY_JF = 16'h8080;
   defparam DCM_INST.PHASE_SHIFT = 0;
   defparam DCM_INST.STARTUP_WAIT = "FALSE";
endmodule
```

```
/* simoncon.c: Final Project: VGA Simon game
 * Authors: Becky Glick <rglick@hmc.edu> and Max Wishman <mwishman@hmc.edu>
 * Date: October 30, 2009
 */

#include <p18f452.h>
#include <stdlib.h>

/* Function Prototypes */
void main(void);
void initialize(void);
int userinput(void);
void read(void);
void playinstruction(char index);
void instructions(void);
void checkinput(void);
void evilsound(void);
void leds(char color);
void music(int period, int duration);
void getseed (void);
void makerandomnote (void);
void srand( unsigned int seed );
int rand( void );
void keypressmusic(int period);
void displayscore(void);


int turn = 0;           //Makes game start from beginning upon reset or power on

int notes[50];          //instantiates array for random instructions

void main (void){       //runs once every turn
        initialize();
        getseed();
        makerandomnote();
        instructions();
        checkinput();
}

void initialize(void)   {
        /* Configure Timer 0 (T0CON)
         * T0CON(7):    TMR0ON  = 1 to enable timer0
         * T0CON(6):    T08BIT  = 0 for 16-bit mode
         * T0CON(5):    T0CS    = 0 for internal instruction clock
         * T0CON(4):    T0SE    = 0 n/a
         * T0CON(3):    PSA     = 0 to assign prescaler
         * T0CON(2-0):  T0PS    = 111 for 256 prescale value
         */
        T0CON = 0x87;                           //1000_0111
        /* Configure Timer 1 (T1CON)
         * T1CON(7):    RD16    = 1 to operate in 16-bit mode
         * T1CON(6):                    = 0 unimplemented
         * T1CON(5-4):  T1CKPS  = 10 for 4 prescale value
         * T1CON(3):    T1OSCEN = 0 to disable oscillator
         * T1CON(2):    T1SYNC  = 0 to synchronize external clock
         * T1CON(1):    TMR1CS  = 0 for internal clock
         * T1CON(0):    TMR1ON  = 1 to enable timer1
         */
        T1CON = 0xA1;                           //1010_0001
        /* Configure ADCON0
         * ADCS1:ADCS0 = 10 (Clock Conversion = F/32)
         * CHS2:CHS0 = 001 (Channel 1, AN1)
         * GO/DONE = 0 (A/D convesion status bit)
         * Unimplemented = 0
         * ADON = 1 (A/D converter module is powered up)
         */
        ADCON0 = 0b10001001;
        /* Configure ADCON1
         * ADFM = 0 (Left Justified)
         * ADCS2 = 0 (Clock Conversion = F/32)
```

```
         * Unimplemented = 00
         * PCFG3:PCFG0 = 0000 (configure A/D port bits for AAAAAAAA)
         */
        ADCON1 = 0b00000000;

        // Configure ports
        TRISA = 0xFF;                           // PorttA is input
        TRISC = 0xFF;                           // PortC is input
        TRISD = 0x00;                           // PortD is output
}

void getseed ( void ){
        //Sample output from A/D Converter to generate seed
        char Seed;
        if (turn == 0 ) {                               // Only seed rand() once
                PIR1bits.ADIF = 0b0;                    // Re-zero interrupt flag
                ADCON0 = 0b10001101;                    // Restart A/D conversion
                while (1)       {
                        if (PIR1bits.ADIF == 1) {       // Check A/D conversion
                                unsigned int Seed = ADRESH;
                                srand(Seed);            // Seed with A/D conversion
                                return;                 // Brake from loop once
                                                        //conversion completed

                        }
                }
        }
}

void makerandomnote (void) {
//Gets pseudo random number from rand and assigns to one of the four colors
        int newrandomnumber;
        newrandomnumber = rand();
        if ((newrandomnumber > 0) && (newrandomnumber < 8192)){
                notes[turn] = 0x03EC;   //note for green
        }
        else if ((newrandomnumber > 8192) && (newrandomnumber < 16383)){
                notes[turn] = 0x04F1;   //note for blue
        }
        else if ((newrandomnumber > 16383) && (newrandomnumber < 24576)){
                notes[turn] = 0x0768;   //note for yellow
        }
        else {
                notes[turn] = 0x0954;   //note for red
        }
}

void leds(char color)   {
//Controls which LED's light up with instructions or to match user input
        if (color == 0) {                       //red LED on
                PORTDbits.RD1 = 0b1;
                PORTDbits.RD2 = 0b0;
                PORTDbits.RD3 = 0b0;
                PORTDbits.RD4 = 0b0;
        }
        else if (color == 1) {                  //green LED on
                PORTDbits.RD1 = 0b0;
                PORTDbits.RD2 = 0b1;
                PORTDbits.RD3 = 0b0;
                PORTDbits.RD4 = 0b0;
        }
        else if (color == 2) {                  //blue LED on
                PORTDbits.RD1 = 0b0;
                PORTDbits.RD2 = 0b0;
                PORTDbits.RD3 = 0b1;
                PORTDbits.RD4 = 0b0;
        }
        else if (color == 3) {                  //yellow LED on
                PORTDbits.RD1 = 0b0;
                PORTDbits.RD2 = 0b0;
```

```
                PORTDbits.RD3 = 0b0;
                PORTDbits.RD4 = 0b1;
        }
        else {                                  //all LEDS off
                PORTDbits.RD1 = 0b0;
                PORTDbits.RD2 = 0b0;
                PORTDbits.RD3 = 0b0;
                PORTDbits.RD4 = 0b0;
        }
}

void music(int period, int duration) {
        unsigned int t0, t1; //timer values as 16-bit numbers
        unsigned int tl, th;

        //reset duration timer
        TMR0H = 0x00;
        TMR0L = 0x00;

        do { //repeat until the duration has elapsed
                if (period !=0) // if not a rest        {
                //set the output high for half the period, then low for half the period

                        PORTDbits.RD0 = 0;              //set output bit low

                        TMR1H = 0; TMR1L = 0;           //reset period counter
                        do {
                                tl = TMR1L;             //t1 = low bits of period timer
                                th = TMR1H;             //t2 = high bits of period timer
                                t1 = tl|th<<8;          //concatinate tl and shifted th
                        }       while (t1<period);      //wait for timer to match period
                        PORTDbits.RD0 = 1;              //set output bit high

                        TMR1H = 0; TMR1L = 0;           //reset period counter
                        do {
                                tl = TMR1L;
                                th = TMR1H;
                                t1 = tl|th<<8;          //concatinate tl and shifted th
                        }       while (t1 < period);    //wait for timer to match period

                tl = TMR0L;                             //set tl to low bits duration timer
                th = TMR0H;                             //set th to high bits of duration
                t0 = tl|th<<8;                          //concatinate tl and shifted th
        }       while (t0<duration);                    //play note for duration
}

void instructions (void){
        // calls playinstruction for each instruction for the appropiate turn
        // in game and creates appropiate delays
        char i;
        int delay;
        long int delay2;
        for (delay2=0; delay2<120000; delay2++){        //for delay between turns
        }
        for (i=0; i<=turn; i++){                        //for delay between instructions
                for (delay=0; delay < 20000; delay++){
                }
                playinstruction(i);
                leds(5);
        }
}


void playinstruction(char index)        {
        int duration = 10000;                   //sets standardized instruction duration
        int period = notes[index];              //translate instruction index to period

        if (period == 0x0954)   {               //note for red
                leds(0);
        }
```

```
        else if (period ==      0x03EC) {       //note for green
                leds(1);
        }
        else if (period ==      0x04F1) {       //note for blue
                leds(2);
        }
        else if (period ==      0x0768) {       //note for yellow
                leds(3);
        }

        //play note for duration
        music(period, duration);
}


void checkinput (void){
        //waits for user input in response to instruction
        int j;
        for (j=0; j<=turn; j++){
                while (PORTC == 0xFF){
                        leds(5);                                //turn off all LEDs
                }
                        if (userinput() != notes[j]) {          //if user inputs incorrectly
                                leds(5);
                                if (notes[j] == 0x0954) {       //check if red
                                        leds(0);
                                }
                                else if (notes[j] == 0x03EC) {  //check if green
                                        leds(1);
                                }
                                else if (notes[j] == 0x04F1) {  //check if blue
                                        leds(2);
                                }
                                else if (notes[j] == 0x0768) {  //check if yellow
                                        leds(3);
                                }
                        evilsound();                            //go to play ominous melody
                }
        }
        turn++;                                                 //increment turn

}

int userinput(void)     {
        if (PORTCbits.RC0 == 0b0) {                             //if red button is pressed
                leds(0);                                        //light red LED
                keypressmusic(0x0954);
                return 0x0954;
        }
        if (PORTCbits.RC1 == 0b0)       {                       //if green button is pressed
                leds(1);                                        //light green LED
                keypressmusic(0x03EC);
                return 0x03EC;
        }
        if (PORTCbits.RC2 == 0b0)       {                       //if blue button is pressed
                leds(2);                                        //light blue LED
                keypressmusic(0x04F1);
                return 0x04F1;
        }
        if (PORTCbits.RC3 == 0b0)       {                       //if yellow button is pressed
                leds(3);
                keypressmusic(0x0768);
                return 0x0768;
        }
        else    {                                               //otherwise, all LEDs are off
                leds(5);
                return 0x0000;
        }
}
```

```
void evilsound(void)    {                                           //plays ominous melody line
        music(0x0333, 0xAAAA);
        music(0x0A66, 0x0F54);
        music(0x0A66, 0xD700);
        displayscore();
        turn = -1;                                                  //resets turn
}

void displayscore(void) {
        PORTD = 0x00;
        PORTD = turn;                                               // turn starts at zero
        while (PORTC == 0xFF)   {
        }
}



void keypressmusic(int period)  {
        unsigned int t0, t1; //timer values as 16-bit numbers
        unsigned int tl, th;

        while (PORTC != 0xFF)   {
                PORTDbits.RD0 = 0;                                  //set output bit low
                                                                    //(driving speaker)
                TMR1H = 0; TMR1L = 0;                               //reset period counter
                do {
                        tl = TMR1L;
                        th = TMR1H;
                        t1 = tl|th<<8;
                }       while (t1 < period);
                PORTDbits.RD0 = 1;                                  //set output bit high
                                                                    //(driving speaker)
                TMR1H = 0; TMR1L = 0;                               //reset period counter
                do {
                        tl = TMR1L;
                        th = TMR1H;
                        t1 = tl|th<<8;
                }       while (t1 < period);
        }
}
```