The Laser Target Game

Final Project Report
December 9, 2000
E155

Zehao Chang and Ben Schmidel

Abstract:

With increasing presence of guns at our schools, more and more children may be inclined to shoot down their classmates. What better way is there to relieve the potentially deadly tension than a fast-paced but non-violent laser target game? This project is the design and implementation of a laser target game, which consists of target board that serves as a user interface, photo-detection circuitry, and an FPGA to control the game-play. Using a laser pointer, the player hits any of the nine target areas to start the game. The player then has sixty seconds to shoot as many targets as possible. At the end of the game, the player can reflect on his performance by looking at the score display.

Introduction

The goal of the game is to shoot as many targets as possible in sixty seconds. There are nine target areas, each with a ring of alternating red and green LEDs surrounded by a photodiode in the center. The gun used in the game is simply a laser pointer. On powering on the game, the player will be greeted by a pre-game sequence of all red LEDs followed by all green LEDs. Shooting any of the nine targets initiates the game. A random red LED ring will now be lit, designating the target that is to be hit. Upon hitting the correct target, the green LED ring corresponding to the target will light up, providing feedback to the player. The score is incremented every time a target is hit. A new target is then chosen and the game waits again for the player to successfully shoot it. During this time, the game keeps track of the number of successful hits and the time remaining. Once the timer reaches zero, an end game LED sequence will clearly indicate to the player that the game has ended. The game returns to the pre-game sequence but retains the score of the last player. The score is reset to zero when a new game is initiated.

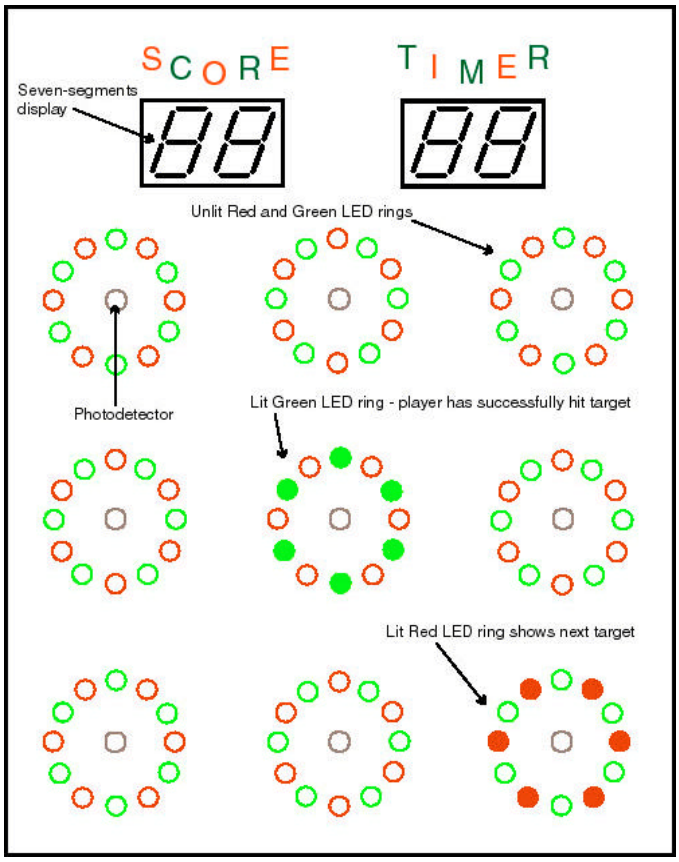


Figure 1: The target panel

The photodiodes situated in the center of each of the target ring are the sensing devices the player must hit. They are interfaced with the FPGA through op-amp circuitry that will determine whether or not to assert the “target-hit” signal. The FPGA controls the flow of the game by receiving input from the photo-detection circuit, randomly selecting targets to be hit, and sending the correct outputs to the LEDs and seven-segment displays.

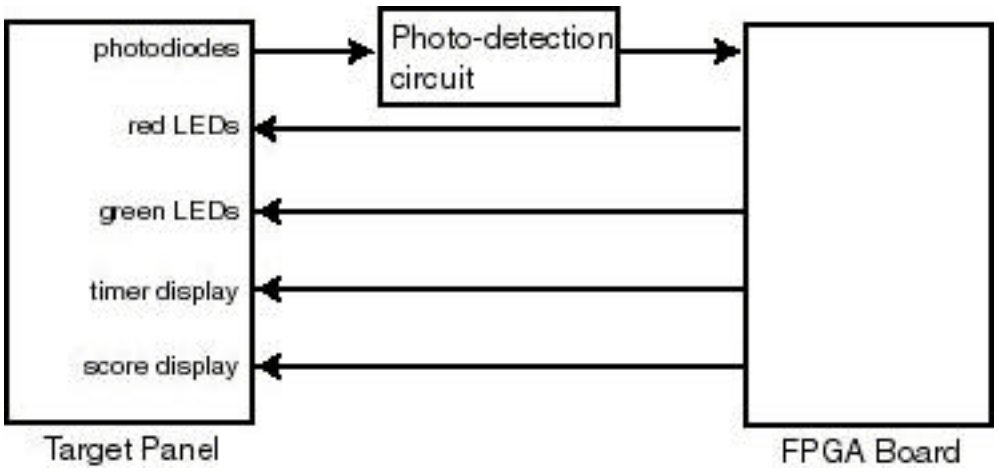
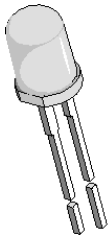


Figure 2: System block diagram

New Hardware



We used Vishay Siliconix BPW43 photodiodes as our photodetectors. The photodiodes provide good sensitivity at 650nm, which is ideal for detecting the output from a laser pointer. The peak sensitivity is at 900nm, which is expected since most photodiodes are intended to work in the infrared range. The following plot shows its spectral response characteristics:

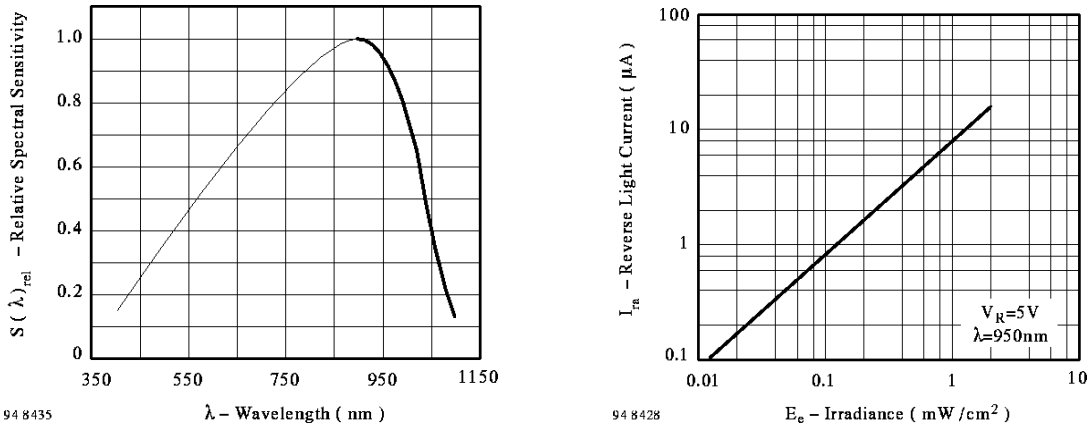


Figure 3: Photodiode response curves

These devices output a small current (in the order of microAmperes) proportional to the amount of light entering them. In our circuit, we use an op-amp to convert the current output to voltage linearly.

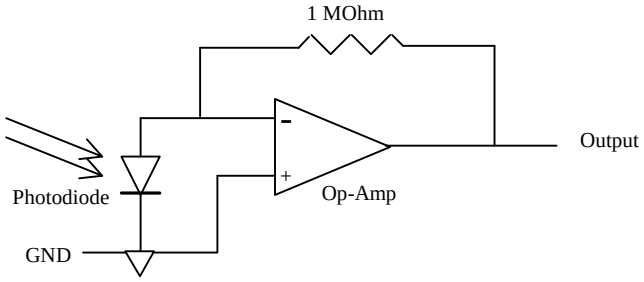


Figure 4: Current-to-Voltage converter for photodiode

For more information about this photodiode please visit:
http://www.vishay.com/temporary/datasheets/optoelectronics/detectors/bpw43.html

Schematics

We used analog circuitry to detect whether a target has been hit and send a binary TTL signal to our FPGA. There will be one sensor at each of our nine target areas, as well as one reference sensor. Each of the target photodiode's output is converted to voltage using the following op-amp circuit as discussed in the previous section (Figure 4).

The purpose of the reference detector is to provide a dynamic threshold for when to assert the "target-hit" signal. It is necessary that this voltage be slightly higher than the output of the current-to-voltage op-amp since to prevent fluctuations and non-uniformity of room lighting from triggering our system. Yet the threshold must be low enough such that the laser pointer can trigger a high in the comparator. Under typical room lighting conditions, the photo-detection circuitry outputs around 2V and increases to 4.5V when a laser pointer shines on it. We then chose the correct resistor value that increases our reference voltage level to 3V.

To obtain this reference voltage we modify the circuit to add a small current to the output from the reference photodiode.

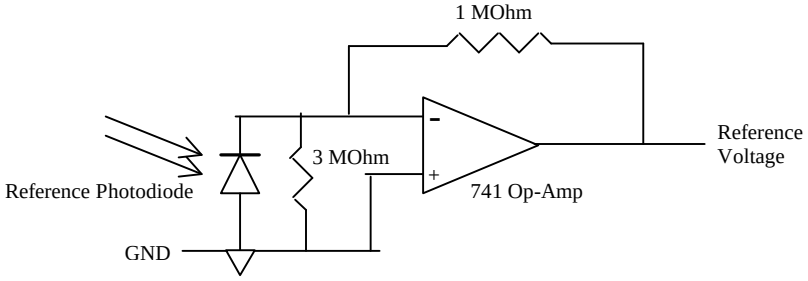


Figure 5: Reference voltage generator

Each of the target's output will then be compared with the reference voltage using an op-amp comparator circuit. The comparator will output a TTL high (5V) if the target output voltage is higher than the reference or a TTL low (0V) otherwise. This converts the analog photodiode output to digital signals that can be sent as input to the FPGA to tell it which target has been hit.

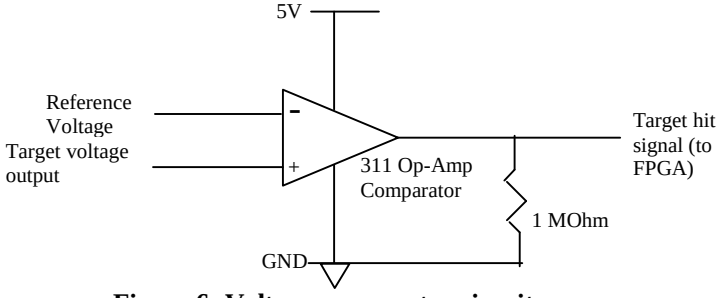


Figure 6: Voltage comparator circuit

All nine detection circuits are identical, and the reference voltage is distributed to each of them. The complete detection circuitry is shown in figure 7.

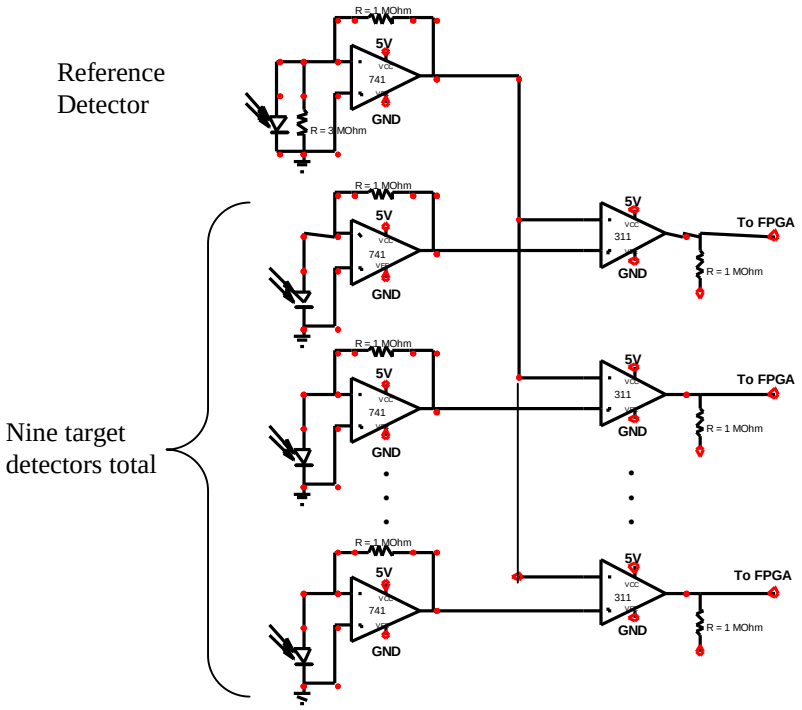


Figure 7: The complete photo-detection circuit

The LED rings consist of LEDs connected in parallel, each LED with its own current-limiting resistor. A single FPGA pin is able to provide enough current to power a ring of six LEDs. Thus no transistor switches are needed to power the target rings.

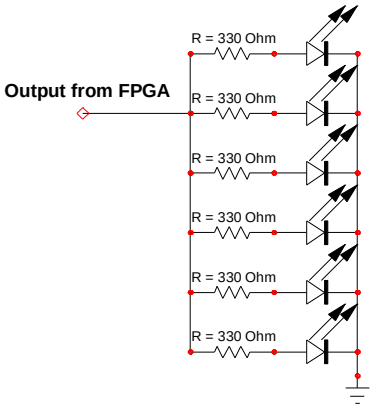


Figure 8: LED ring circuit

Both seven-segment displays (for score and timer) are two-way multiplexed in a common anode configuration. A PNP-transistor is needed at the anode to provide enough current to the multiplexed display.

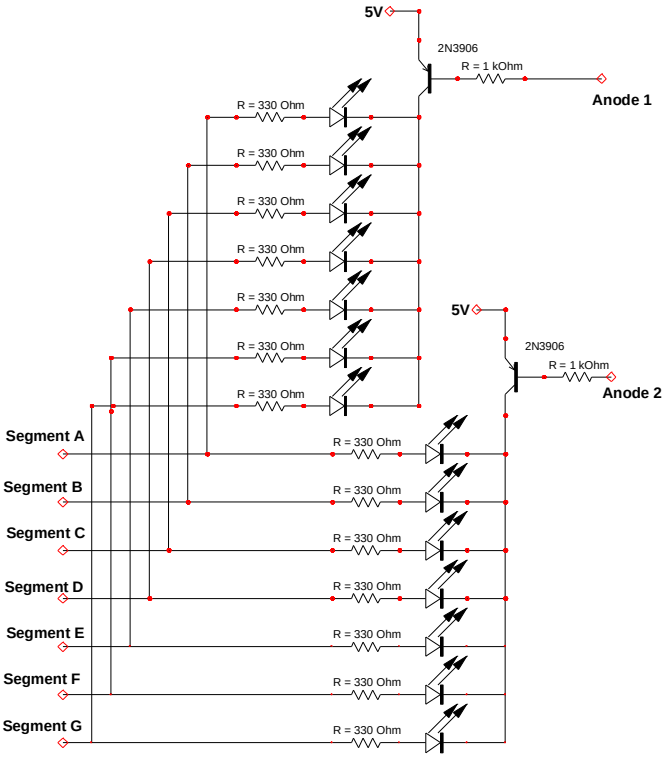


Figure 9: Multiplexed display

Appendix A shows which pins of the FPGA connects to each of the breadboard circuitry described above.

FPGA Design

The FPGA serves as the heart of the game. Input to the FPGA simply consists of a nine-bit bus with each bit corresponding to a unique target photodiode. A high bit is taken to mean that the light incident on the photodiode is above a preset threshold. Thus targeting a photodiode with a laser should cause the corresponding FPGA input bit to transition from low to high. This input bus is then converted into convenient boolean quantities indicating whether a new game should start if the current game has ended or whether the active target has been hit. The "start-game" signal is determined by checking to see whether any of the inputs from the target are high, since shooting any target should start a new game. The "target-hit" signal is determined by checking to see whether the target with the lit red LED ring is providing a high input bit.

Outputs from the FPGA include two nine-bit busses that are used to power target's red and green LEDs. Each bit corresponds to a unique target and lights six of that targets same-color LEDs. The game provides a score tabulation and a countdown game timer displayed on two seven-segment displays. Each display is multiplexed using an 8 ms clock such that its two digits are alternately powered. The FPGA outputs a seven-bit bus to each of the multiplexed displays and the two clock signals used to drive the anode of each seven-segment display. Since each segment of the display is connected to the cathode, a low output bit towards the segments will cause a segment to be lit. The anode signals are inverses of each other and have a period of 8 ms. The eight most significant bits of the game counter are wired to the LEDs on the FPGA board as an additional indication of time.

A block diagram of the verilog top-level module can be seen in Fig. 11. This diagram depicts the relationship between all key hardware modules. A few modules instantiated solely within GameFSM, however, are not depicted. System inputs and outputs can be clearly seen from this diagram.

Hardware Module Descriptions:

DebounceFSM:

This module debounces the input signal "hit" and provides output in the form of the debounced signal "hitConfirmed." The finite state machine used to debounce the input is depicted in Fig. X. It simply verifies that the input signal "hit" is high for two consecutive clock cycles before setting "hitConfirmed" high. When the input signal goes low, the output is set to low at the next rising edge of the clock. It was found that the input, if not properly debounced, is a source of failure. The detectors were found to provide a microsecond-width pulse sixty times a second, indicating that electrical and optical noise in the room is likely to affect the operation of the photo-detection circuitry. Debouncing input from this circuitry using an 8 ms clock successfully filtered out these pulses.

GameFSM:

This module is responsible for implementing the game rules. The finite state machine implied by this module can be seen in Fig. 10. Text in the figure enclosed by parentheses and including an assignment arrow (✍) indicate variable assignments made during the transition from one state to another. Text within the state "bubble" indicates variable assignments maintained as long as that state is the current state. The labeled states are essential to the game. Pregame FSM and Postgame FSM, however, are optional additions that make the verilog more complex but make the game more visually appealing. The Pregame FSM is implemented it as a two-state finite state machine where one state lights all targets red while the other lights all targets green. The two states alternate every half-second until a start-game signal is detected, at which point the current state transitions to state S4. Pregame FSM simply serves to provide the game player with some entertaining visual effects while at the same time indicating that the game is operational and waiting for a new game to start. Postgame FSM is a purely optional display of visual effects, with the purpose of entertaining the game player and informing him that his current game is over. It also serves to provide a delay between the end of the current game and Pregame FSM so that the player does not accidentally trigger the start of a new game while trying to score last-second points. It is implemented as a nine-state

finite state machine that sequentially lights all the targets' red LEDs in a snaking pattern, followed by lighting all the targets' green LEDs following a reverse snaking pattern.

Timing for this module is achieved through the use of timing modules GetTimerBig, GetTimerSmall. The user score is encoded for a seven-segment display using the module SevenSegDisplay and output as "scoreSegs." Similarly, the game time remaining is encoded for a seven-segment display and output as "timerSegs."

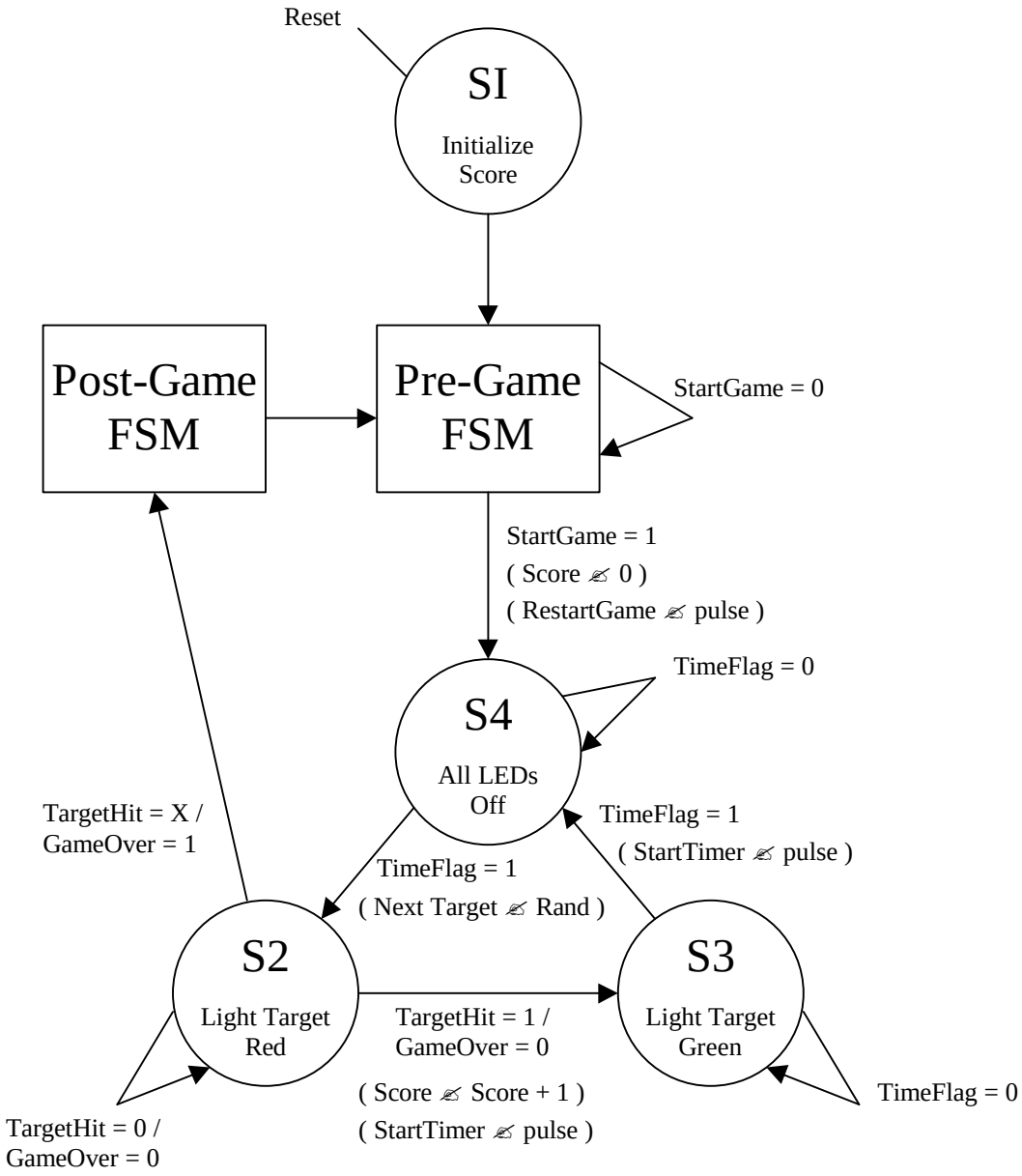


Figure 10: The GameFSM state transition diagram

GameTimerBig:

This module is used to specify the length of a game. When the input signal "restartGame" is pulsed, the module begins counting down from approximately sixty seconds. When the counter reaches zero, counting stops and the output flag "gameOver" is set high. The eight most significant bits of the counter are output as "leds." Furthermore, the amount of time remaining is also converted into a multiplexed seven-segment display encoding and output as timerSegs. A game is begun when "restartGame" transitions from high to low and persists until the next rising edge of the "gameOver" flag.

GameTimerSmall:

This module is used for small-scale, in-game timing. When the input signal "startTimer" is pulsed, the module begins counting down from approximately half a second. After approximately a sixteenth of a second has elapsed, the output flag "timeFlagS" is set high. When the counter reaches zero counting stops and the output "timeFlag" is set high. This timer is used by GameFSM to time state transitions.

GetClocks:

This module is used to obtain the slower clocks needed for input debouncing and seven segment display multiplexing. Currently these clocks both have periods of 8.192 ms, but using two separate clocks makes it easy to independently change debouncing and multiplexing times in the future.

GetRand:

This module aids in the selection of a random target. The output "rand" equivalent to a microsecond counter modulus nine. GameFSM chooses a new target based upon the timing of user input. Since user input is essentially random on the order of microseconds, "rand" is a simple yet effective way of choosing new targets.

SevenSegDisplay:

This module was initially written for Lab 3 and modified for use with this project. It encodes two decimal numbers for multiplexed display on a dual seven-segment display. Depending on the value of "clk," the output "segs" will either be an encoding of the input "data1" or "data2.

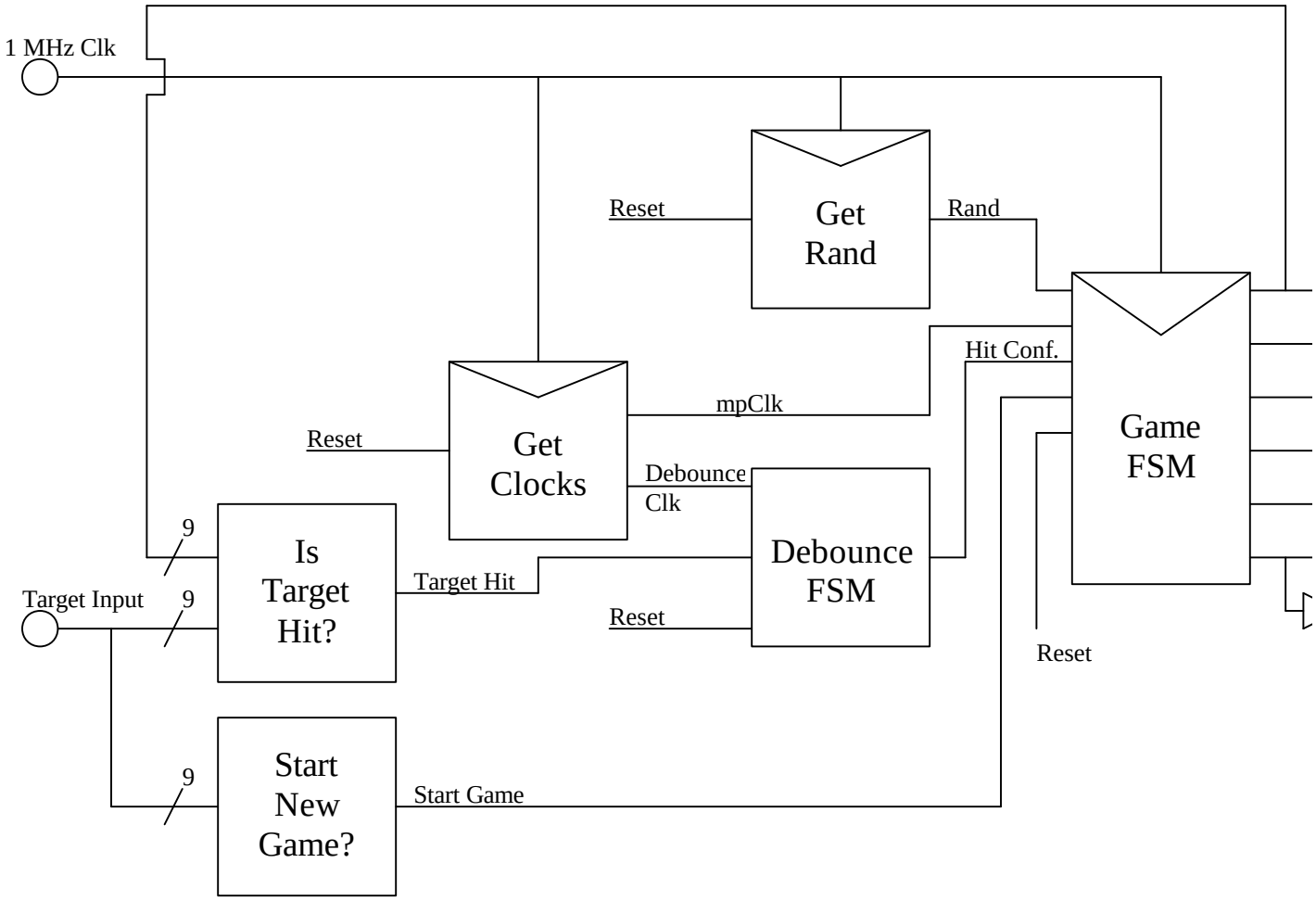
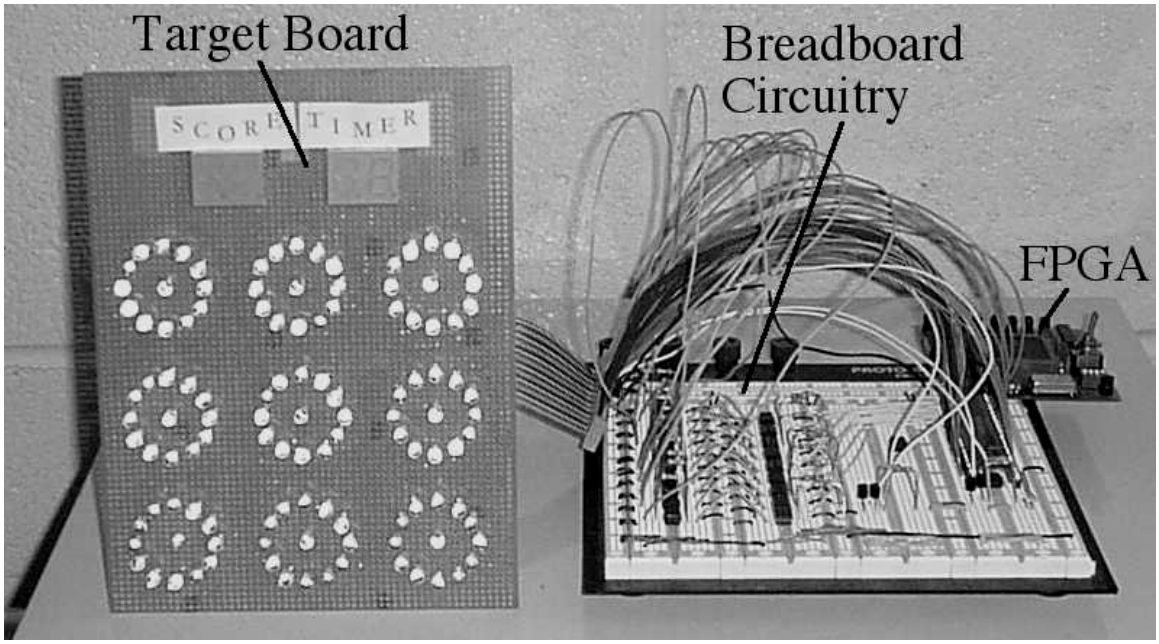


Figure 11: Block diagram of verilog top-level module

Results



The above picture shows the physical appearance of our final project. The laser target game is a success since it works very well and is fun to play, as evinced by the happy player shown on the right.

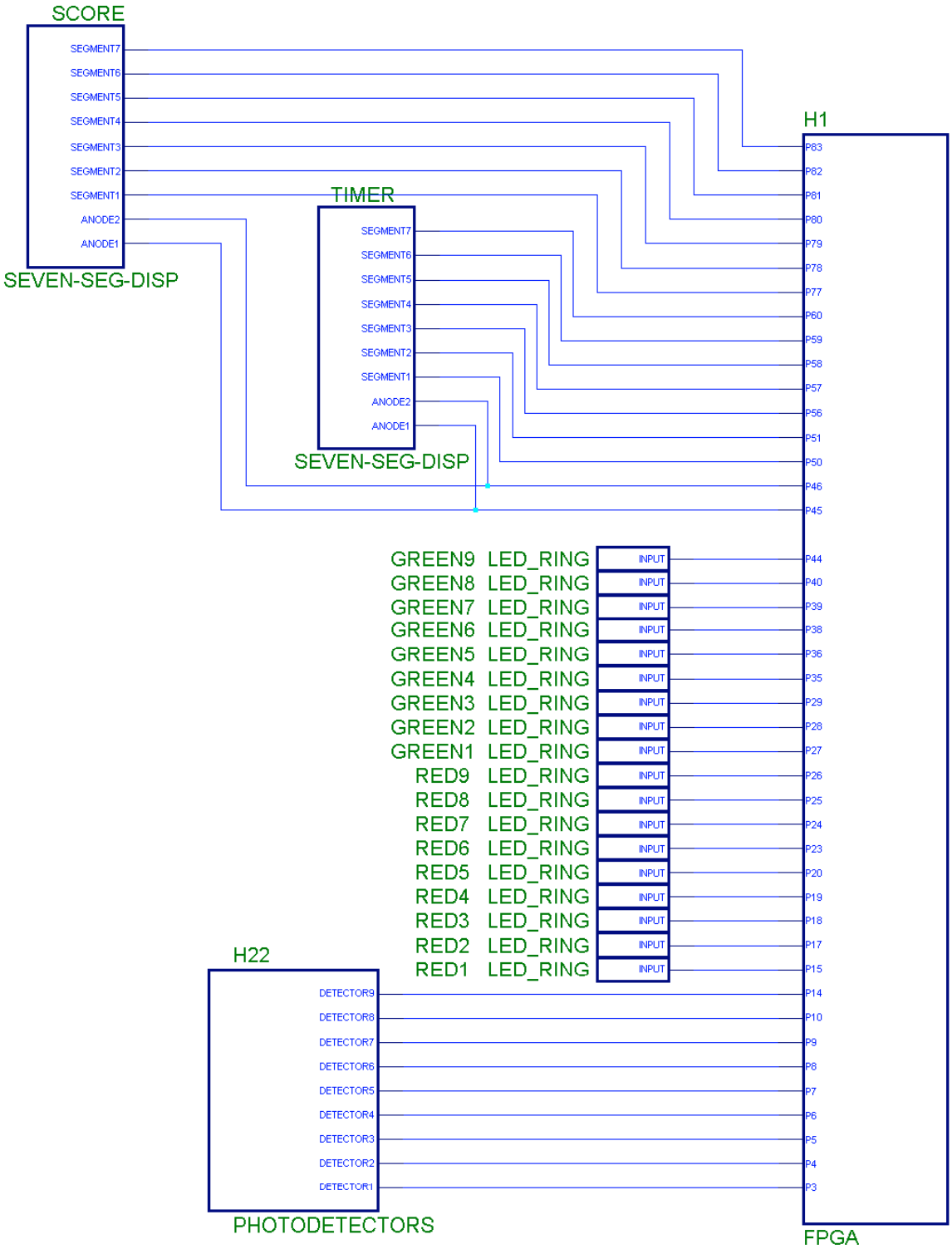
The breadboard circuitry and verilog for the FPGA required tinkering to work as intended, but there were no major problems. The soldering of components onto the target board, however, was odious time consuming. We originally envisioned a ring of eight red and eight green LEDs per target, but we quickly reduced that number to six as we realized the amount of soldering we had to do.

The use of the HC11 for scoring and timing as stated in our proposal was abandoned because we found the FPGA to be capable of both tasks elegantly. In addition, we also decided not to control the laser such that only pulses may be fired, as we found the continuous pressing of the trigger detracted the player from the fun of the game. In addition, the size of each photodiode is small enough such that very accurate shooting is required, and we felt that the game would be too hard if the player is restricted to pulses.

Parts List

Part	Source	Vendor Part #	Quantity	Price
Vishay Siliconix BPW43 Photodiodes	Arrow Electronics (www.arrow.com)	BPW43	20	$11.60
Ultrabright Red LED	Digikey (www.digikey.com)	HLMP3750A-ND	100	$14.65
Ultrabright Green LED	Digikey (www.digikey.com)	HLMP3950A-ND	100	$16.82
741 Op Amp	Stock Room		10	
311 Comparator	Stock Room		9	
Prepunched Perfboard	Prof. Harris		1	

Appendix A – FPGA pin assignments



Appendix B – Verilog Code

```
//  Written by Ben Schmidel, 11/30/00
//  File: tlm.v
//  email: bschmide@hmc.edu

//  Top level module for the laser target game


module TLM(reset, clk, targetInput, targetOutputRed, targetOutputGreen, mpClk, mpClk_b,
                scoreSegs, timerSegs, leds);

        input reset;                    // global reset
        input clk;                      // 1 MHz clock signal
        input [8:0] targetInput;        // one-hot encoding (photodetectors)
        output [8:0] targetOutputRed;   // one-hot encoding (red LEDs)
        output [8:0] targetOutputGreen; // one-hot encoding (green LEDs)
        output mpClk;                   // clk used for hex-display multiplexing
        output mpClk_b;                 // inverse of mpClk (b stands for "bar")
        output [6:0] scoreSegs;         // 7-seg display encoded score
        output [6:0] timerSegs;         // 7-seg display encoded game-time remaining
        output [7:0] leds;              // 8 most sig. bits of the game timer

        wire [3:0] rand;                // "random" number selecting next target
        wire debounceClk;               // clk used for target debouncing
        wire startGame;                 // possible game-start indication
        wire startGameConfirmed;        // definite game-start signal (debounced startGame)
        wire targetHit;                 // indicates a possible successful hit
        wire targetHitConfirmed;        // definite successful hit (debounced targetHit)


        GetRand randTarget(reset, clk, rand);
        GetClocks clks(reset, clk, debounceClk, mpClk);
        assign mpClk_b = ~mpClk;

        // any target can be used to start the game
        assign startGame = |targetInput;

        // only a hit on the active target counts
        assign targetHit = |(targetInput & targetOutputRed);

        DebounceFSM start_fsm(reset, debounceClk, startGame, startGameConfirmed);
        DebounceFSM target_fsm(reset, debounceClk, targetHit, targetHitConfirmed);
        GameFSM game(reset, clk, mpClk, startGameConfirmed, targetHitConfirmed, rand,
                        targetOutputRed, targetOutputGreen, scoreSegs, timerSegs, leds);

endmodule
```

```
// Written by Ben Schmidel, 11/30/00
// File: getclocks.v
// email: bschmide@hmc.edu

// If clk has a frequency of 1MHz, debounceClk and mpClk
// will have periods of 8.192 ms. debounceClk is used to
// debounce input from the target and mpClk is used to
// multiplex output to seven-segment displays. Though
// one clock could serve both purposes, having separate
// clocks makes it easier to change debounce time or
// multiplexing speed at a later date.


module GetClocks(reset, clk, debounceClk, mpClk);

        input reset;
        input clk;
        output debounceClk;
        output mpClk;

        reg [12:0] counter;


        always @ (posedge clk or posedge reset)

                if (reset)
                        counter <= 0;
                else
                        counter <= counter + 1;


        // debounceClk will have a period 2^13 - 1 times that of clk
        assign debounceClk = counter[12];
        assign mpClk = counter[12];

endmodule
```

```
// Written by Ben Schmidel, 11/30/00
// File: debouncefsm.v
// email: bschmide@hmc.edu

// debounceFSM is a finite-state machine that debounces a signal
// to confirm it is high. If two successive highs are sampled,
// the output is high. Otherwise the output is low.

module DebounceFSM(reset, debounceClk, hit, hitConfirmed);

	input reset;
	input debounceClk;
	input hit;
	output hitConfirmed;

	reg [1:0]state;
	reg [1:0]nextState;

	parameter S0 = 2'b00;
	parameter S1 = 2'b10;
	parameter S2 = 2'b01;


	// state register
	always@(posedge debounceClk or posedge reset)

		if (reset)
			state <= S0;
		else
			state <= nextState;


	// next state logic
	always @ (state or hit)

		case (state)
			S0:	if (hit) nextState <= S1;
				else nextState <= S0;

			S1:	if (hit) nextState <= S2;
				else nextState <= S0;

			S2:	if (hit) nextState <= S2;
				else nextState <= S0;

			default: nextState <= S0;
		endcase


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

endmodule
```

```
// Written by Ben Schmidel, 11/30/00
// File: gametimersmall.v
// email: bschmide@hmc.edu

// Performs all short-term game timing. When startTimer pulses,
// the timer starts. Half a second later, the timer stops and
// timeFlag is set high. Restarting the timer will set timeFlag low.
// timeFlagS goes high a sixteenth of a second after start timer
// transitions from high to low.

module GameTimerSmall(reset, startTimer, clk, timeFlag, timeFlagS);

	input reset;
	input startTimer; // flag used to start the timing process
	input clk;
	output timeFlag, timeFlagS;		// flags indicating a given time has elapsed

	reg [19:0] gameCounter;

	parameter HALFSEC_TIMER = 20'b1000_0000_0000_0000_0000;


	always @ (posedge clk or posedge reset)

		if (reset)
			gameCounter <= 0;
		else if (startTimer)
			gameCounter <= HALFSEC_TIMER; // init counter
		else if (gameCounter > 0)
			gameCounter <= gameCounter - 1; // count down
		else
			gameCounter <= gameCounter;


	assign timeFlag = (gameCounter == 0);
	assign timeFlagS = (gameCounter <= 20'b0111_0000_0000_0000_0000);

endmodule
```

```
//  Written by Ben Schmidel, 11/30/00
//  File: gametimerbig.v
//  email: bschmide@hmc.edu

//  times the span of the game and provides a flag to say when the
//  current game has ended.  The  gameOver flag will be set high
//  approximately sixty seconds after the input restartGame
//  transitions from high to low.  The time remaining is displayed in
//  binary on the eight FPGA board LEDs.  It is also output for multiplexed
//  display on a seven segment display.

module GameTimerBig(reset, restartGame, clk, mpClk, state, gameOver, leds, timerSegs);

        input reset;
        input restartGame;              // flag that starts the counter
        input clk;
        input mpClk;                    // multiplexing clock
        input [3:0] state;
        output gameOver;                // indicates the counter has finished
        output [7:0] leds;              // 8 most sig. bits of the counter
        output [6:0] timerSegs;         // 7-seg encoding of the counter

        wire secondClk;

        reg [3:0] timerData1;
        reg [3:0] timerData2;
        reg [3:0] nextTimerData1;
        reg [3:0] nextTimerData2;

        reg [26:0] gameCounter;
        reg [26:0] nextGameCounter;

        parameter COUNTER_START = 26'b11_1100_1111_1111_1111_1111_1111;


        // Maintain the game counter
        always @ (posedge clk or posedge reset)

                if (reset)
                        gameCounter <= 0;
                else if (restartGame)
                        gameCounter <= COUNTER_START;
                else if (gameCounter > 0)
                        gameCounter <= gameCounter - 1;
                else
                        gameCounter <= 0;


        assign gameOver = ((gameCounter == 0) && (state[2] | state[1]));
        assign leds = gameCounter[26:19];

        assign secondClk = gameCounter[19];


        // Maintain seven segment display output
        always @ (posedge secondClk or posedge reset)
```

```
        if (reset) begin
                timerData1 <= 0;
                timerData2 <= 0;
        end
        else if (restartGame) begin
                timerData1 <= 6;
                timerData2 <= 0;
        end
        else if (timerData2 == 0) begin
                if (timerData1 > 0) begin
                        timerData1 <= timerData1 - 1;
                        timerData2 <= 9;
                end
                else begin
                        timerData1 <= 0;
                        timerData2 <= 0;
                end
        end
        else begin
                timerData1 <= timerData1;
                timerData2 <= timerData2 - 1;
        end


    SevenSegDisplay timer(mpClk, timerData1, timerData2,  timerSegs);

endmodule
```

```
//  Written By Ben Schmidel, 9/20/00
//  File: sevensegdisplay.v
//  Lab 3 Verilog Source -- Adapted in part from
//   the verilog example "seven-seg-disp" given in E155 lecture
//  email: bschmide@hmc.edu
//  Initially written for Lab 3
//  Modified 9/29/00 for Lab 4
//  Modified again for final project on 12/2/00

//  This module represents a hexidecimal encoder.  When the clock is high, the 7-bit bus
//  segs encodes the binary number data1.  When clk is low, segs encodes the binary number
//  data2.  Our final project only requires the use of decimal numbers, so only values from 0 to 9
//  will be encoded.

module SevenSegDisplay(clk, data1, data2, segs);

        input clk;
        input [3:0] data1;
        input [3:0] data2;
        output [6:0] segs;
        wire [4:0] data;
        reg [6:0] segs;

        parameter ZERO  = 7'b000_0001;
        parameter ONE   = 7'b100_1111;
        parameter TWO   = 7'b001_0010;
        parameter THREE = 7'b000_0110;
        parameter FOUR  = 7'b100_1100;
        parameter FIVE  = 7'b010_0100;
        parameter SIX   = 7'b010_0000;
        parameter SEVEN = 7'b000_1111;
        parameter EIGHT = 7'b000_0000;
        parameter NINE  = 7'b000_0100;

        // multiplex to select the desired input data
        assign data = clk ? data1 : data2;

        // asynchronous hex encoder
        always @ (data)

                case (data)
                        0: segs <= ZERO;
                        1: segs <= ONE;
                        2: segs <= TWO;
                        3: segs <= THREE;
                        4: segs <= FOUR;
                        5: segs <= FIVE;
                        6: segs <= SIX;
                        7: segs <= SEVEN;
                        8: segs <= EIGHT;
                        9: segs <= NINE;

                        default: segs <= ZERO;
                endcase

endmodule
```

```
//  Written by Ben Schmidel, 11/30/00
//  File: getrand.v
//  email: bschmide@hmc.edu

//  rand will be a number between 0 and 8, inclusive.  rand
//  increments every millionth of a second (if clk has a
//  frequency of 1MHz).  If rand is used to select a new target
//  whenever the current target is hit, the new target should be
//  sufficiently random due to the uncertainty in the length of time
//  between successful target hits.

module GetRand(reset, clk, rand);

        input reset;
        input clk;
        output [3:0] rand;

        reg [3:0] rand;

        always @ (posedge clk or posedge reset)

                if (reset)
                        rand <= 0;
                else if (rand == 8)
                        rand <= 0;
                else
                        rand  <= rand + 1;

endmodule
```

```
//  Written by Ben Schmidel, 11/30/00
//  File: gamefsm.v
//  email: bschmide@hmc.edu

//  This module is the heart of the laser target game.  It is responsible
//  for implementing all of the game rules.  It is currently designed to
//  wait for a start-game signal.  Upon this signal, a game timer is started.
//  A target is randomly selected and illuminated red.  This state persists
//  until it is determined that the player has hit that target.  When this
//  happens, the target is illuminated green for half a second.  Then
//  all LEDs are turned off for half a second and a new target is chosen.
//  This process repeats until the game timer reaches zero (and the  gameOver
//  flag is set high).  At this point, some post-game visuals are displayed to
//  indicated that the current game has ended.  After completion, the game goes
//  back to waiting for a start-game signal.

module GameFSM(reset, clk, mpClk, startGame, targetHit, rand, targetOutputRed,
                targetOutputGreen, scoreSegs, timerSegs, leds);

        input reset;
        input clk;
        input mpClk;                    // multiplexing clock
        input startGame;                // start game flag
        input targetHit;                // target hit flag
        input [3:0] rand;               // random number from 0 to 8
        output [8:0] targetOutputRed;   // target area red  LEDs
        output [8:0] targetOutputGreen; // target area green  LEDs
        output [6:0] scoreSegs;         // seven segment display encoded score
        output [6:0] timerSegs;         // seven segment display encoded timer
        output [7:0] leds;              // 8 most sig. bits of game timer

        wire gameOver;                  // flag indicating game has ended
        wire timeFlag, timeFlagS;       // flag indicating short timer has ended

        reg eFlag;                      // flag used during post-game visuals
        reg nexteFlag;
        reg restartGame;                // flag used to restart the game timer
        reg startTimer;                 // flag used to start short timer
        reg [8:0] targetOutputRed;
        reg [8:0] targetOutputGreen;
        reg [8:0] nextTargetOutputRed;
        reg [8:0] nextTargetOutputGreen;
        reg [3:0] scoreData1;
        reg [3:0] scoreData2;
        reg [3:0] nextScoreData1;
        reg [3:0] nextScoreData2;

        reg [3:0] state;
        reg [3:0] nextState;


        //  State Encodings
        parameter SI = 4'b0101;     // initial state only reached by reset
        parameter S0 = 4'b0000;
        parameter S1 = 4'b0001;
        parameter S2 = 4'b0010;
```

```
parameter S3 = 4'b0011;
parameter S4 = 4'b0100;

//  states used to flash LEDs when a game ends
parameter SE0 = 4'b0111;
parameter SE1 = 4'b1000;
parameter SE2 = 4'b1001;
parameter SE3 = 4'b1010;
parameter SE4 = 4'b1011;
parameter SE5 = 4'b1100;
parameter SE6 = 4'b1101;
parameter SE7 = 4'b1110;
parameter SE8 = 4'b1111;
parameter SE9 = 4'b0110;


// Instantiate modules used by this module
GameTimerBig gtb(reset, restartGame, clk, mpClk, state, gameOver, leds, timerSegs);
GameTimerSmall gts(reset, startTimer, clk, timeFlag, timeFlagS);
SevenSegDisplay score(mpClk, scoreData1, scoreData2, scoreSegs);


//  State Change Logic: Sets the next stage and its corresponding outputs
always @ (posedge clk or posedge reset)

        if (reset) begin
                state <= SI;
                targetOutputRed <= 9'b111111111;
                targetOutputGreen <= 9'b111111111;
                scoreData1 <= 0;
                scoreData2 <= 0;
                eFlag <= 0;
        end
        else begin
                state <= nextState;
                targetOutputRed <= nextTargetOutputRed;
                targetOutputGreen <= nextTargetOutputGreen;
                scoreData1 <= nextScoreData1;
                scoreData2 <= nextScoreData2;
                eFlag <= nexteFlag;
        end


//  Next State Logic
always @ (state or startGame or timeFlag or gameOver or targetHit or eFlag or timeFlagS)

        case (state)
                // perform reset initializations
                SI:     begin
                                nextState <= S0;
                                startTimer <= 1;
                                restartGame <= 0;
                                nexteFlag <= 0;
                        end

                // pre-game visuals -- wait for start game flag
```

```
S0:     if (startGame) begin
                nextState <= S4;
                startTimer <= 1;
                restartGame <= 1;
                nexteFlag <= 0;
        end
        else if (timeFlag) begin
                nextState <= S1;
                startTimer <= 1;
                restartGame <= 0;
                nexteFlag <= 0;
        end
        else begin
                nextState <= S0;
                startTimer <= 0;
                restartGame <= 0;
                nexteFlag <= 0;
        end

// pre-game visuals -- wait for start game flag
S1:     if (startGame) begin
                nextState <= S4;
                startTimer <= 1;
                restartGame <= 1;
                nexteFlag <= 0;
        end
        else if (timeFlag) begin
                nextState <= S0;
                startTimer <= 1;
                restartGame <= 0;
                nexteFlag <= 0;
        end
        else begin
                nextState <= S1;
                startTimer <= 0;
                restartGame <= 0;
                nexteFlag <= 0;
        end

// wait for current target to be hit
S2:     if (gameOver) begin
                nextState <= SE0;
                startTimer <= 1;
                restartGame <= 0;
                nexteFlag <= 0;
        end
        else if (targetHit) begin
                nextState <= S3;
                startTimer <= 1;
                restartGame <= 0;
                nexteFlag <= 0;
        end
        else begin
                nextState <= S2;
                startTimer <= 0;
                restartGame <= 0;
```

```
                nexteFlag <= 0;
        end

// flash hit-target green
S3:     if (timeFlag) begin
                nextState <= S4;
                startTimer <= 1;
                restartGame <= 0;
                nexteFlag <= 0;
        end
        else begin
                nextState <= S3;
                startTimer <= 0;
                restartGame <= 0;
                nexteFlag <= 0;
        end

// choose next target; pause before illuminating
S4:     if (timeFlag) begin
                nextState <= S2;
                startTimer <= 1;
                restartGame <= 0;
                nexteFlag <= 0;
        end
        else begin
                nextState <= S4;
                startTimer <= 0;
                restartGame <= 0;
                nexteFlag <= 0;
        end


// Perform post-game visual effects (OPTIONAL)

SE0: if (timeFlagS) begin
                if(eFlag) begin
                        nextState <= S0;
                        startTimer <= 1;
                        restartGame <= 0;
                end
                else begin
                        nextState <= SE1;
                        startTimer <= 1;
                        restartGame <= 0;
                end
                nexteFlag <= eFlag;
        end
        else begin
                nextState <= SE0;
                startTimer <= 0;
                restartGame <= 0;
                nexteFlag <= eFlag;
        end

SE1: if (timeFlagS) begin
                if(eFlag) begin
```

```
                        nextState <= SE0;
                        startTimer <= 1;
                        restartGame <= 0;
                end
                else begin
                        nextState <= SE2;
                        startTimer <= 1;
                        restartGame <= 0;
                end
                nexteFlag <= eFlag;
        end
        else begin
                nextState <= SE1;
                startTimer <= 0;
                restartGame <= 0;
                nexteFlag <= eFlag;
        end

SE2: if (timeFlagS) begin
                if(eFlag) begin
                        nextState <= SE1;
                        startTimer <= 1;
                        restartGame <= 0;
                end
                else begin
                        nextState <= SE3;
                        startTimer <= 1;
                        restartGame <= 0;
                end
                nexteFlag <= eFlag;
        end
        else begin
                nextState <= SE2;
                startTimer <= 0;
                restartGame <= 0;
                nexteFlag <= eFlag;
        end

SE3: if (timeFlagS) begin
                if(eFlag) begin
                        nextState <= SE2;
                        startTimer <= 1;
                        restartGame <= 0;
                end
                else begin
                        nextState <= SE4;
                        startTimer <= 1;
                        restartGame <= 0;
                end
                nexteFlag <= eFlag;
        end
        else begin
                nextState <= SE3;
                startTimer <= 0;
                restartGame <= 0;
                nexteFlag <= eFlag;
```

```
        end

SE4: if (timeFlagS) begin
                if(eFlag) begin
                        nextState <= SE3;
                        startTimer <= 1;
                        restartGame <= 0;
                end
                else begin
                        nextState <= SE5;
                        startTimer <= 1;
                        restartGame <= 0;
                end
                nexteFlag <= eFlag;
        end
        else begin
                nextState <= SE4;
                startTimer <= 0;
                restartGame <= 0;
                nexteFlag <= eFlag;
        end

SE5: if (timeFlagS) begin
                if(eFlag) begin
                        nextState <= SE4;
                        startTimer <= 1;
                        restartGame <= 0;
                end
                else begin
                        nextState <= SE6;
                        startTimer <= 1;
                        restartGame <= 0;
                end
                nexteFlag <= eFlag;
        end
        else begin
                nextState <= SE5;
                startTimer <= 0;
                restartGame <= 0;
                nexteFlag <= eFlag;
        end

SE6: if (timeFlagS) begin
                if(eFlag) begin
                        nextState <= SE5;
                        startTimer <= 1;
                        restartGame <= 0;
                end
                else begin
                        nextState <= SE7;
                        startTimer <= 1;
                        restartGame <= 0;
                end
                nexteFlag <= eFlag;
        end
        else begin
```

```
                nextState <= SE6;
                startTimer <= 0;
                restartGame <= 0;
                nexteFlag <= eFlag;
        end

SE7: if (timeFlagS) begin
                if(eFlag) begin
                        nextState <= SE6;
                        startTimer <= 1;
                        restartGame <= 0;
                end
                else begin
                        nextState <= SE8;
                        startTimer <= 1;
                        restartGame <= 0;
                end
                nexteFlag <= eFlag;
        end
        else begin
                nextState <= SE7;
                startTimer <= 0;
                restartGame <= 0;
                nexteFlag <= eFlag;
        end

SE8: if (timeFlagS) begin
                if(eFlag) begin
                        nextState <= SE7;
                        startTimer <= 1;
                        restartGame <= 0;
                end
                else begin
                        nextState <= SE9;
                        startTimer <= 1;
                        restartGame <= 0;
                end
                nexteFlag <= eFlag;
        end
        else begin
                nextState <= SE8;
                startTimer <= 0;
                restartGame <= 0;
                nexteFlag <= eFlag;
        end

SE9: begin
                nextState <= SE8;
                startTimer <= 1;
                restartGame <= 0;
                nexteFlag <= 1;
        end

default: begin
                nextState <= S0;
                startTimer <= 0;
```

```
                        restartGame <= 0;
                        nexteFlag <= 0;
                end
        endcase


//  Output Logic, it determines the next Red or Green output
always @ (state or startGame or rand or timeFlag or targetHit or scoreData1 or scoreData2 or
                targetOutputRed or targetOutputGreen or eFlag)

        case (state)
                // perform reset initializations
                SI:
                begin
                        nextTargetOutputGreen <= 0;
                        nextTargetOutputRed <= 0;
                        nextScoreData1 <= 0;
                        nextScoreData2 <= 0;
                end

                // pre-game visuals -- wait for start game flag
                S0:
                if (startGame)
                begin
                        nextTargetOutputGreen <= 0;
                        nextTargetOutputRed <= 0;
                        nextScoreData1 <= 0;
                        nextScoreData2 <= 0;
                end
                else if (timeFlag) begin
                        nextScoreData1 <= scoreData1;
                        nextScoreData2 <= scoreData2;
                        // light all red LEDs
                        nextTargetOutputGreen <= 9'b111_111_111;
                        nextTargetOutputRed <= 0;
                end
                else begin
                        nextScoreData1 <= scoreData1;
                        nextScoreData2 <= scoreData2;
                        nextTargetOutputGreen <= targetOutputGreen;
                        nextTargetOutputRed <= targetOutputRed;
                end

                // pre-game visuals -- wait for start game flag
                S1:
                if (startGame)
                begin
                        nextScoreData1 <= 0;
                        nextScoreData2 <= 0;
                        nextTargetOutputGreen <= 0;
                        nextTargetOutputRed <= 0;
                end
                else if (timeFlag) begin
                        nextScoreData1 <= scoreData1;
                        nextScoreData2 <= scoreData2;
```

```
            nextTargetOutputGreen <= 0;
            // light all green LEDs
            nextTargetOutputRed <= 9'b111_111_111;
end
else begin
            nextScoreData1 <= scoreData1;
            nextScoreData2 <= scoreData2;
            nextTargetOutputGreen <= targetOutputGreen;
            nextTargetOutputRed <= targetOutputRed;
end

// wait for current target to be hit
S2:
if (targetHit) begin
            nextTargetOutputGreen <= targetOutputRed;
            nextTargetOutputRed <= 0;
            // increment score
            if (scoreData2 == 9) begin
                        nextScoreData1 <= scoreData1 + 1;
                        nextScoreData2 <= 0;
            end
            else begin
                        nextScoreData1 <= scoreData1;
                        nextScoreData2 <= scoreData2 + 1;
            end
end
else begin
            nextTargetOutputGreen <= targetOutputGreen;
            nextTargetOutputRed <= targetOutputRed;
            nextScoreData1 <= scoreData1;
            nextScoreData2 <= scoreData2;
end

// flash hit-target green
S3:
begin
            nextScoreData1 <= scoreData1;
            nextScoreData2 <= scoreData2;
            if (timeFlag) begin
                        nextTargetOutputGreen <= 0;
                        nextTargetOutputRed <= 0;
            end
            else begin
                        nextTargetOutputGreen <= targetOutputGreen;
                        nextTargetOutputRed <= targetOutputRed;
            end
end

// choose next target; pause before illuminating
S4:
begin
            nextScoreData1 <= scoreData1;
            nextScoreData2 <= scoreData2;
            if (timeFlag) begin
                        nextTargetOutputGreen <= 0;
                        case (rand)
```

```
                    0: nextTargetOutputRed <= 9'b100_000_000;
                    1: nextTargetOutputRed <= 9'b010_000_000;
                    2: nextTargetOutputRed <= 9'b001_000_000;
                    3: nextTargetOutputRed <= 9'b000_100_000;
                    4: nextTargetOutputRed <= 9'b000_010_000;
                    5: nextTargetOutputRed <= 9'b000_001_000;
                    6: nextTargetOutputRed <= 9'b000_000_100;
                    7: nextTargetOutputRed <= 9'b000_000_010;
                    8: nextTargetOutputRed <= 9'b000_000_001;
                    default:
                      nextTargetOutputRed <= 9'b101_010_101;
                    endcase
            end
            else begin
                    nextTargetOutputGreen <= targetOutputGreen;
                    nextTargetOutputRed <= targetOutputRed;
            end
end


// Perform post-game visual effects (OPTIONAL)

SE0:
if (eFlag) begin
            nextScoreData1 <= scoreData1;
            nextScoreData2 <= scoreData2;
            nextTargetOutputGreen <= 9'b111_111_111;
            nextTargetOutputRed <= 0;
end
else begin
            nextScoreData1 <= scoreData1;
            nextScoreData2 <= scoreData2;
            nextTargetOutputGreen <= 0;
            nextTargetOutputRed <= 9'b100_000_000;
end

SE1:
if (eFlag) begin
            nextScoreData1 <= scoreData1;
            nextScoreData2 <= scoreData2;
            nextTargetOutputGreen <= 9'b011_111_111;
            nextTargetOutputRed <= 9'b100_000_000;
end
else begin
            nextScoreData1 <= scoreData1;
            nextScoreData2 <= scoreData2;
            nextTargetOutputGreen <= 0;
            nextTargetOutputRed <= 9'b110_000_000;
end

SE2:
if (eFlag) begin
            nextScoreData1 <= scoreData1;
            nextScoreData2 <= scoreData2;
            nextTargetOutputGreen <= 9'b001_111_111;
            nextTargetOutputRed <= 9'b110_000_000;
```

```
end
else begin
        nextScoreData1 <= scoreData1;
        nextScoreData2 <= scoreData2;
        nextTargetOutputGreen <= 0;
        nextTargetOutputRed <= 9'b111_000_000;
end

SE3:
if (eFlag) begin
        nextScoreData1 <= scoreData1;
        nextScoreData2 <= scoreData2;
        nextTargetOutputGreen <= 9'b000_111_111;
        nextTargetOutputRed <= 9'b111_000_000;
end
else begin
        nextScoreData1 <= scoreData1;
        nextScoreData2 <= scoreData2;
        nextTargetOutputGreen <= 0;
        nextTargetOutputRed <= 9'b111_001_000;
end

SE4:
if (eFlag) begin
        nextScoreData1 <= scoreData1;
        nextScoreData2 <= scoreData2;
        nextTargetOutputGreen <= 9'b000_110_111;
        nextTargetOutputRed <= 9'b111_001_000;
end
else begin
        nextScoreData1 <= scoreData1;
        nextScoreData2 <= scoreData2;
        nextTargetOutputGreen <= 0;
        nextTargetOutputRed <= 9'b111_011_000;
end

SE5:
if (eFlag) begin
        nextScoreData1 <= scoreData1;
        nextScoreData2 <= scoreData2;
        nextTargetOutputGreen <= 9'b000_100_111;
        nextTargetOutputRed <= 9'b111_011_000;
end
else begin
        nextScoreData1 <= scoreData1;
        nextScoreData2 <= scoreData2;
        nextTargetOutputGreen <= 0;
        nextTargetOutputRed <= 9'b111_111_000;
end

SE6:
if (eFlag) begin
        nextScoreData1 <= scoreData1;
        nextScoreData2 <= scoreData2;
        nextTargetOutputGreen <= 9'b000_000_111;
        nextTargetOutputRed <= 9'b111_111_000;
```

```
                end
                else begin
                        nextScoreData1 <= scoreData1;
                        nextScoreData2 <= scoreData2;
                        nextTargetOutputGreen <= 0;
                        nextTargetOutputRed <= 9'b111_111_100;
                end

                SE7:
                if (eFlag) begin
                        nextScoreData1 <= scoreData1;
                        nextScoreData2 <= scoreData2;
                        nextTargetOutputGreen <= 9'b000_000_011;
                        nextTargetOutputRed <= 9'b111_111_100;
                end
                else begin
                        nextScoreData1 <= scoreData1;
                        nextScoreData2 <= scoreData2;
                        nextTargetOutputGreen <= 0;
                        nextTargetOutputRed <= 9'b111_111_110;
                end

                SE8:
                if (eFlag) begin
                        nextScoreData1 <= scoreData1;
                        nextScoreData2 <= scoreData2;
                        nextTargetOutputGreen <= 9'b000_000_001;
                        nextTargetOutputRed <= 9'b111_111_110;
                end
                else begin
                        nextScoreData1 <= scoreData1;
                        nextScoreData2 <= scoreData2;
                        nextTargetOutputGreen <= 0;
                        nextTargetOutputRed <= 9'b111_111_111;
                end

                default:
                begin
                        nextTargetOutputGreen <= targetOutputGreen;
                        nextTargetOutputRed <= targetOutputRed;
                        nextScoreData1 <= scoreData1;
                        nextScoreData2 <= scoreData2;
                end

        endcase

endmodule
```