Lab 4 Report

Introduction and Requirement

We will be creating a single player Whack-A-Mole game. Users will be tasked with whacking as many moles as fast they can under a minute. During the game, players will see a countdown and a count of how many moles they have whacked. At the end of the game, the player will be able to see how many points they got and will be able to restart the game to play again. Once players have mastered the easy level, they will be able to increase the difficulty level and compete for the most points.

Upon turning on the board, the player will be motioned to push a button to start the game. Upon pressing the start button, a 3 second countdown will begin that will say go at the end. At this point, the player will whack the moles as fast as they can under a minute. The LED row will randomly turn on a random amount of LEDs, that will serve as the indicators of where the moles have appeared. If the player flips the correct switch underneath the LED, the player will be awarded a point. If the player flips a switch that does not have a lit LED above it, the player's score will be decremented. During the game, the seven segment display will show the minute countdown with the left two digits and the current score with the right two digits. After the player has whacked all of the moles, the board will reset with new moles. If at any point during the game, the player wants to restart the game, the player can press a button to reset it. At the end of the minute, the player will be able to see their score and press the start button to begin the game again.

Here are the requirements:

1. LED Functionality (20%) - Display "moles" to be whacked by turning on the LED for the switch to be hit. Turns off the LED when the mole is whacked

2. Switch Functionality (20%) - Player "whacks moles" by hitting the switch. Whacking a mole results in an increase in score and the LED turning off. If player whacks a nonexistent mole, the score decreases instead

3. Reset Round (10%) - Start or restart a game round, player restarts game but retains high score and game count.

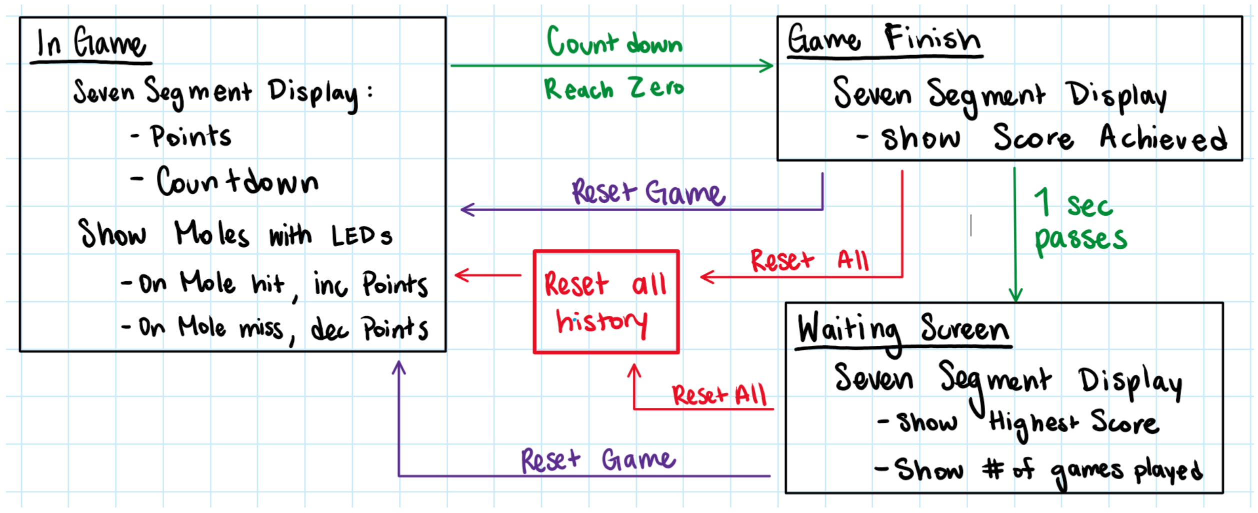
4. Reset All (10%) - round is regenerated, player high score and game count is restarted

5. Game Display (20%) - During game, counts down the number of seconds remaining and indicates game start / end. Displays current number of points in real time, decrement and increment as necessary

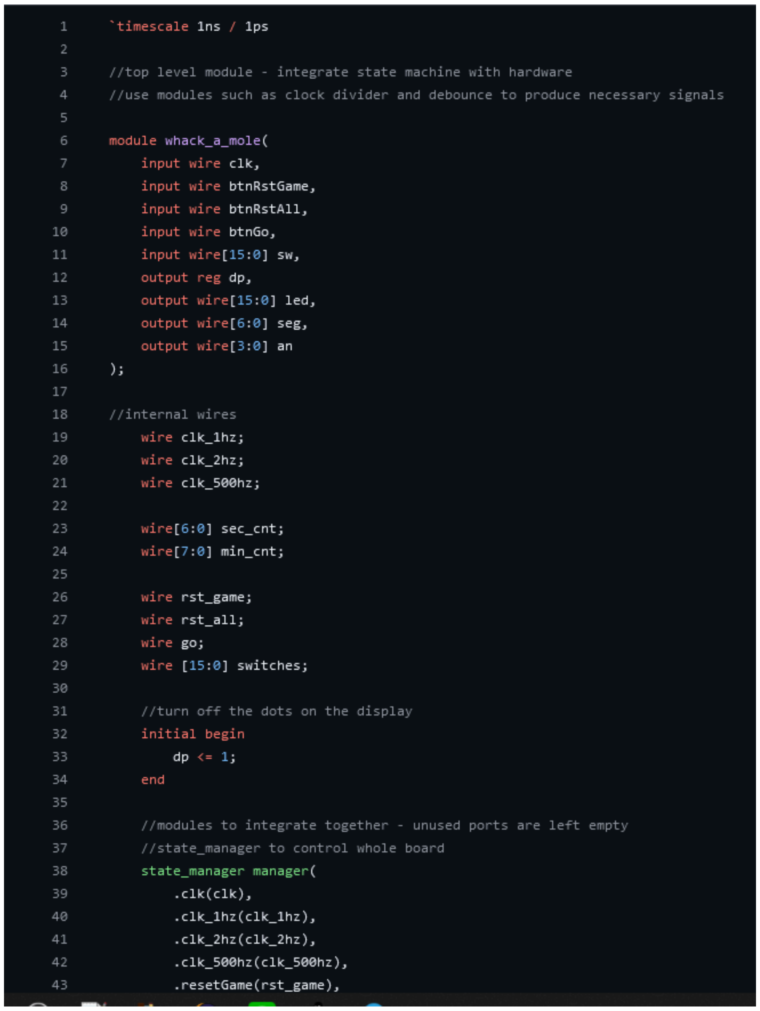
6. Status Display (20%) - Before any games, display instructions. After any games have concluded, display a high score and the number of games played so far.

Design Description

Whack-a-mole is implemented using several states (In Game, Game Finish, Waiting Screen) displayed by the black boxes in the picture below. Upon startup, the user plays the game by starting in the In Game state. Upon finishing the game (i.e., when the countdown runs out), the device will transition into the Game Finish state and briefly display the score the player got. Afterward, the device will transition into the waiting screen that will display the game history. At any point, the player can reset the game to play again or reset all in order to start everything fresh. This later command works by resetting the number of games played and the highest score.

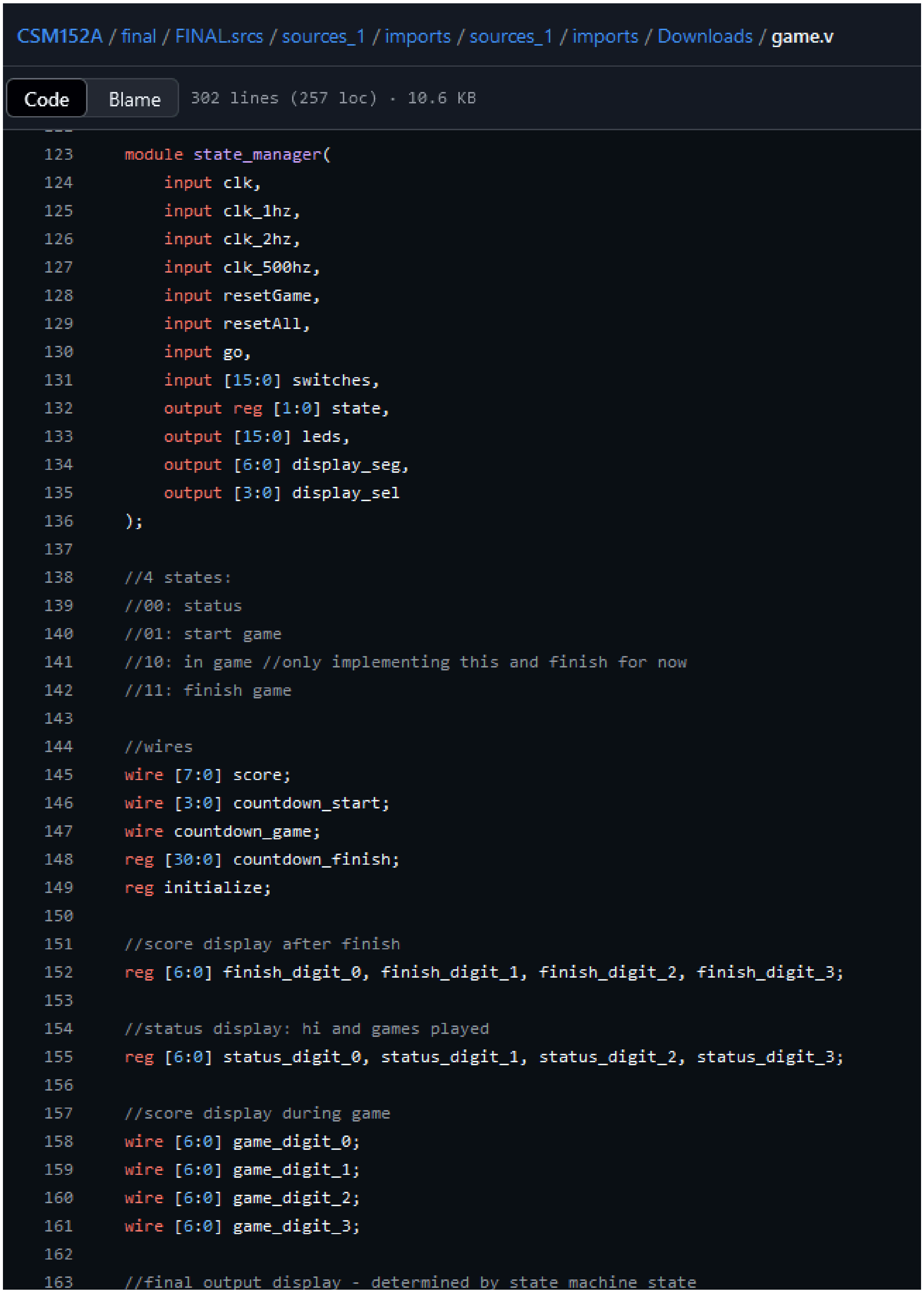


For this project, we utilize various modules to implement each of the functions. Firstly, we tie together everything using a top level module called whack_a_mole in whack_a_mole.v that runs the state machine module and provides it with the necessary clock and debounced hardware signals, and passes through the hardware outputs to the user:



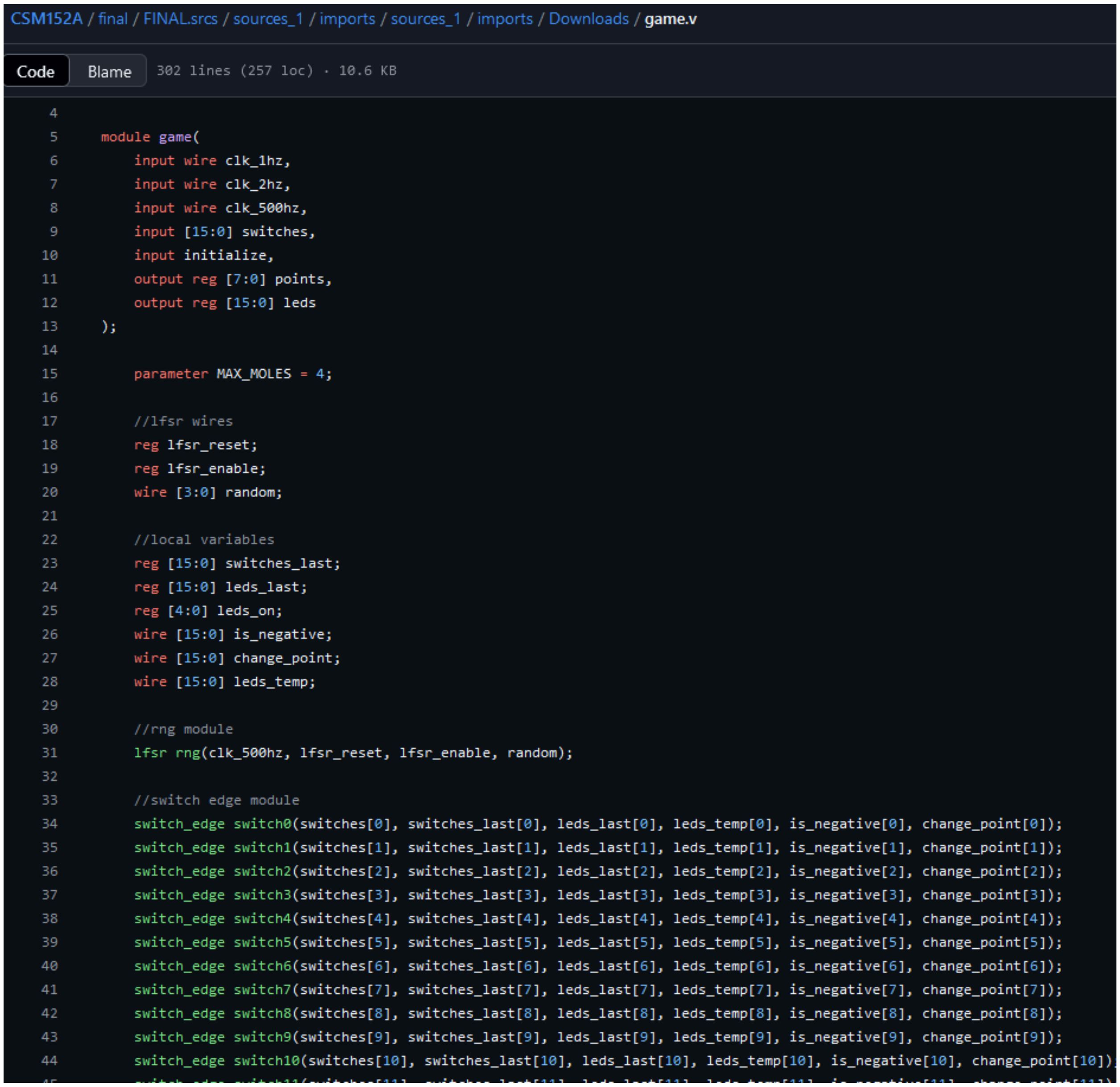
```
`timescale 1ns / 1ps

//top level module - integrate state machine with hardware
//use modules such as clock divider and debounce to produce necessary signals

module whack_a_mole(
    input wire clk,
    input wire btnRstGame,
    input wire btnRstAll,
    input wire btnGo,
    input wire[15:0] sw,
    output reg dp,
    output wire[15:0] led,
    output wire[6:0] seg,
    output wire[3:0] an
);

//internal wires
    wire clk_1hz;
    wire clk_2hz;
    wire clk_500hz;

    wire[6:0] sec_cnt;
    wire[7:0] min_cnt;

    wire rst_game;
    wire rst_all;
    wire go;
    wire [15:0] switches;

    //turn off the dots on the display
    initial begin
        dp <= 1;
    end

    //modules to integrate together - unused ports are left empty
    //state_manager to control whole board
    state_manager manager(
        .clk(clk),
        .clk_1hz(clk_1hz),
        .clk_2hz(clk_2hz),
        .clk_500hz(clk_500hz),
        .resetGame(rst_game),
```

Then, for the main game logic we have game.v, which includes the state machine module and implementation of each of the states (such as game), using helper modules.


CSM152A / final / FINAL.srcs / sources_1 / imports / sources_1 / imports / Downloads / game.v

302 lines (257 loc) · 10.6 KB

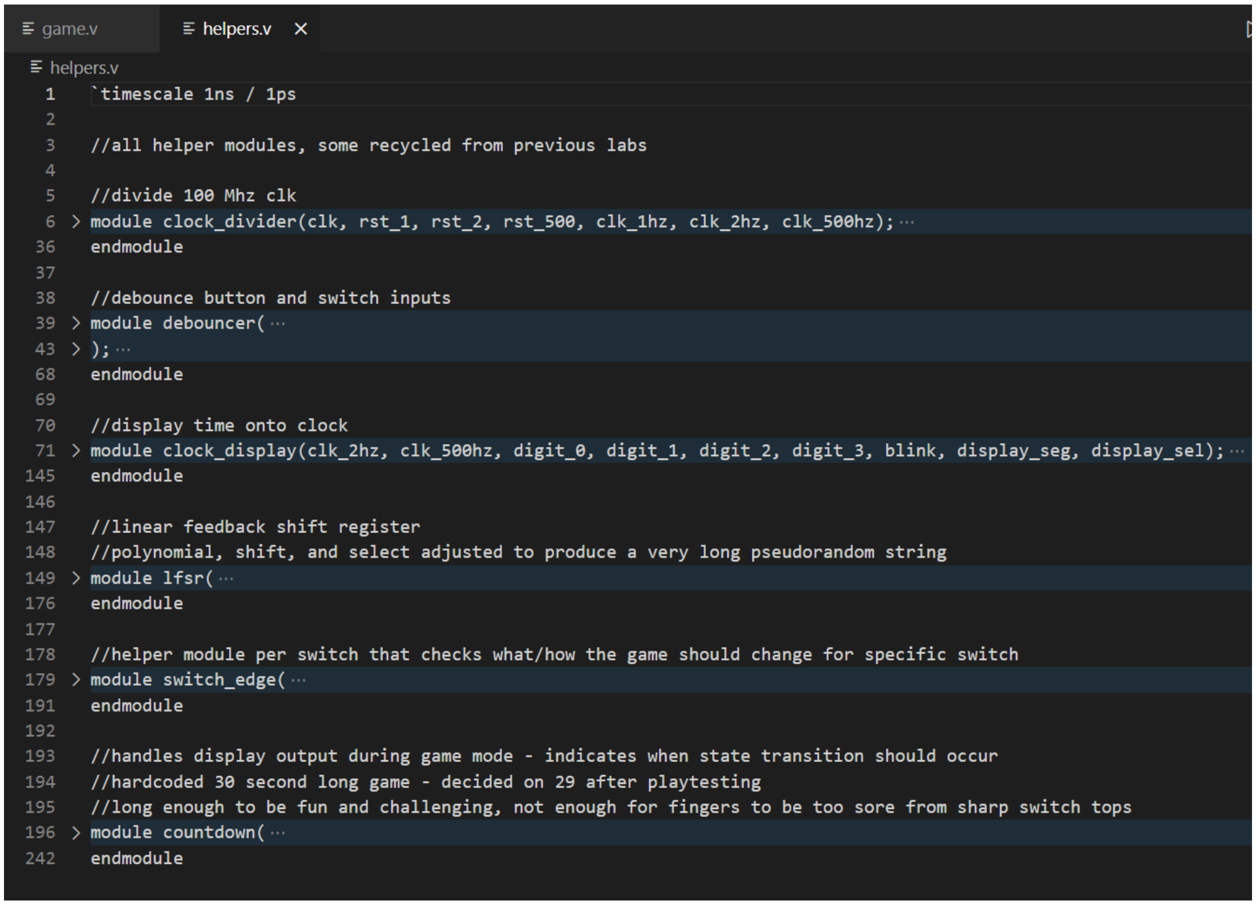
```verilog
module state_manager(
    input clk,
    input clk_1hz,
    input clk_2hz,
    input clk_500hz,
    input resetGame,
    input resetAll,
    input go,
    input [15:0] switches,
    output reg [1:0] state,
    output [15:0] leds,
    output [6:0] display_seg,
    output [3:0] display_sel
);

//4 states:
//00: status
//01: start game
//10: in game //only implementing this and finish for now
//11: finish game

//wires
wire [7:0] score;
wire [3:0] countdown_start;
wire countdown_game;
reg [30:0] countdown_finish;
reg initialize;

//score display after finish
reg [6:0] finish_digit_0, finish_digit_1, finish_digit_2, finish_digit_3;

//status display: hi and games played
reg [6:0] status_digit_0, status_digit_1, status_digit_2, status_digit_3;

//score display during game
wire [6:0] game_digit_0;
wire [6:0] game_digit_1;
wire [6:0] game_digit_2;
wire [6:0] game_digit_3;

//final output display - determined by state machine state
```


CSM152A / final / FINAL.srcs / sources_1 / imports / sources_1 / imports / Downloads / game.v

302 lines (257 loc) · 10.6 KB

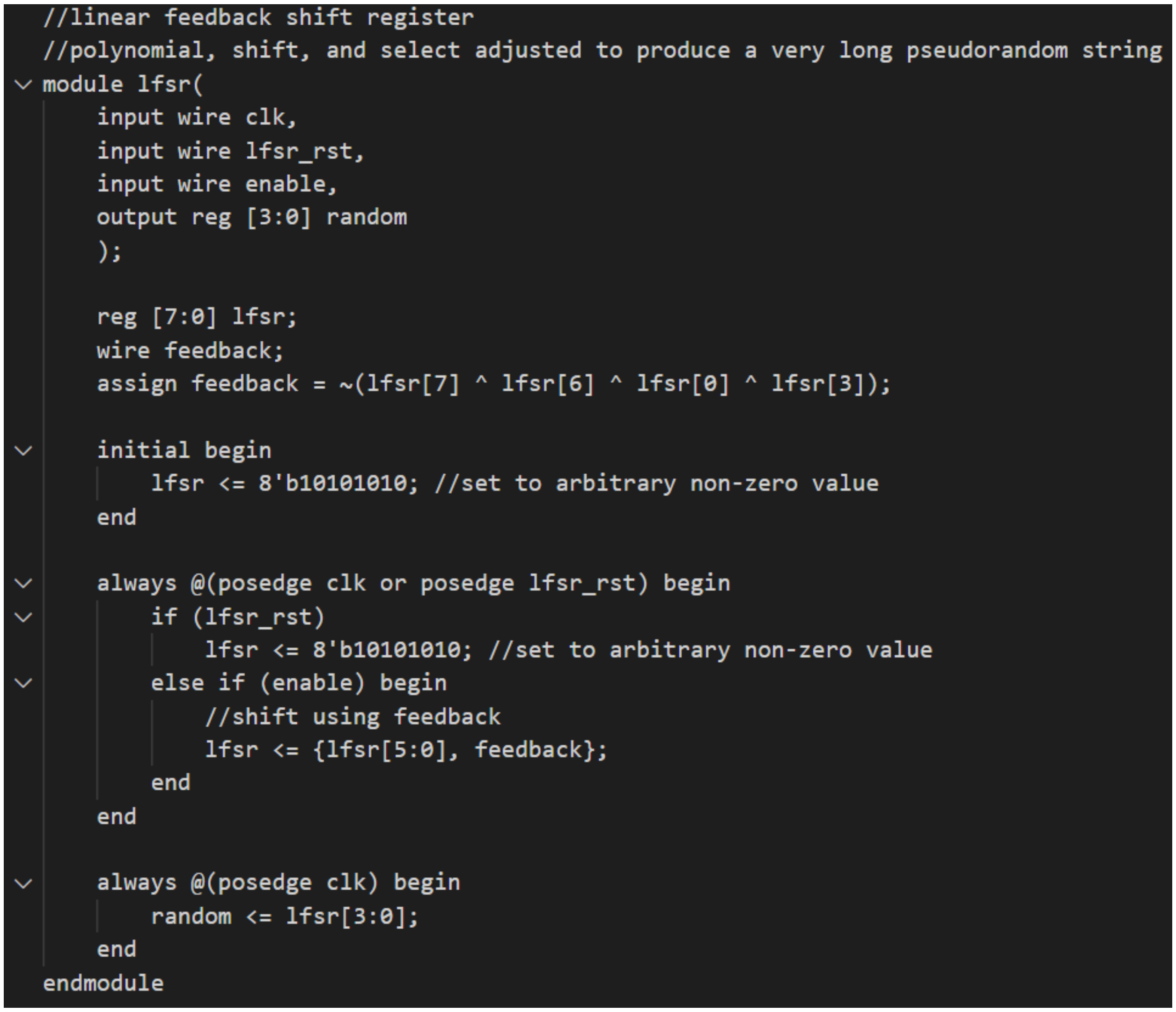
```verilog

module game(
    input wire clk_1hz,
    input wire clk_2hz,
    input wire clk_500hz,
    input [15:0] switches,
    input initialize,
    output reg [7:0] points,
    output reg [15:0] leds
);

    parameter MAX_MOLES = 4;

    //lfsr wires
    reg lfsr_reset;
    reg lfsr_enable;
    wire [3:0] random;

    //local variables
    reg [15:0] switches_last;
    reg [15:0] leds_last;
    reg [4:0] leds_on;
    wire [15:0] is_negative;
    wire [15:0] change_point;
    wire [15:0] leds_temp;

    //rng module
    lfsr rng(clk_500hz, lfsr_reset, lfsr_enable, random);

    //switch edge module
    switch_edge switch0(switches[0], switches_last[0], leds_last[0], leds_temp[0], is_negative[0], change_point[0]);
    switch_edge switch1(switches[1], switches_last[1], leds_last[1], leds_temp[1], is_negative[1], change_point[1]);
    switch_edge switch2(switches[2], switches_last[2], leds_last[2], leds_temp[2], is_negative[2], change_point[2]);
    switch_edge switch3(switches[3], switches_last[3], leds_last[3], leds_temp[3], is_negative[3], change_point[3]);
    switch_edge switch4(switches[4], switches_last[4], leds_last[4], leds_temp[4], is_negative[4], change_point[4]);
    switch_edge switch5(switches[5], switches_last[5], leds_last[5], leds_temp[5], is_negative[5], change_point[5]);
    switch_edge switch6(switches[6], switches_last[6], leds_last[6], leds_temp[6], is_negative[6], change_point[6]);
    switch_edge switch7(switches[7], switches_last[7], leds_last[7], leds_temp[7], is_negative[7], change_point[7]);
    switch_edge switch8(switches[8], switches_last[8], leds_last[8], leds_temp[8], is_negative[8], change_point[8]);
    switch_edge switch9(switches[9], switches_last[9], leds_last[9], leds_temp[9], is_negative[9], change_point[9]);
    switch_edge switch10(switches[10], switches_last[10], leds_last[10], leds_temp[10], is_negative[10], change_point[10])
```

Finally, we have helper.v, which has all the supporting modules that handle hardware inputs, generating new clock signals, determining the status of a specific switch/mole/led (switch_edge) in the next cycle, pseudorandom number generation using a linear feedback shift register, in-game display logic, and display output.

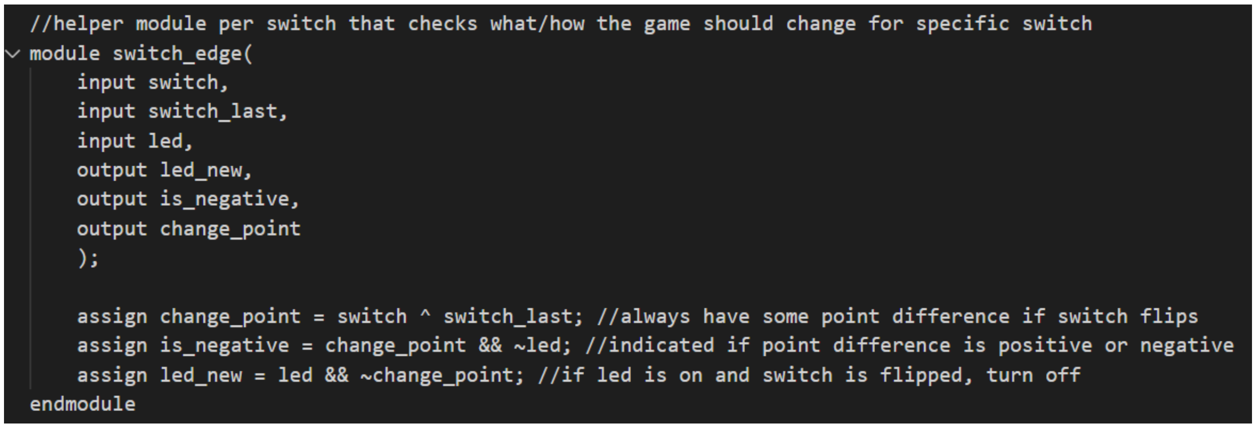

game.v helpers.v

helpers.v

```
`timescale 1ns / 1ps

//all helper modules, some recycled from previous labs

//divide 100 Mhz clk
module clock_divider(clk, rst_1, rst_2, rst_500, clk_1hz, clk_2hz, clk_500hz);⋯
endmodule

//debounce button and switch inputs
module debouncer(⋯
);⋯
endmodule

//display time onto clock
module clock_display(clk_2hz, clk_500hz, digit_0, digit_1, digit_2, digit_3, blink, display_seg, display_sel);⋯
endmodule

//linear feedback shift register
//polynomial, shift, and select adjusted to produce a very long pseudorandom string
module lfsr(⋯
endmodule

//helper module per switch that checks what/how the game should change for specific switch
module switch_edge(⋯
endmodule

//handles display output during game mode - indicates when state transition should occur
//hardcoded 30 second long game - decided on 29 after playtesting
//long enough to be fun and challenging, not enough for fingers to be too sore from sharp switch tops
module countdown(⋯
endmodule
```



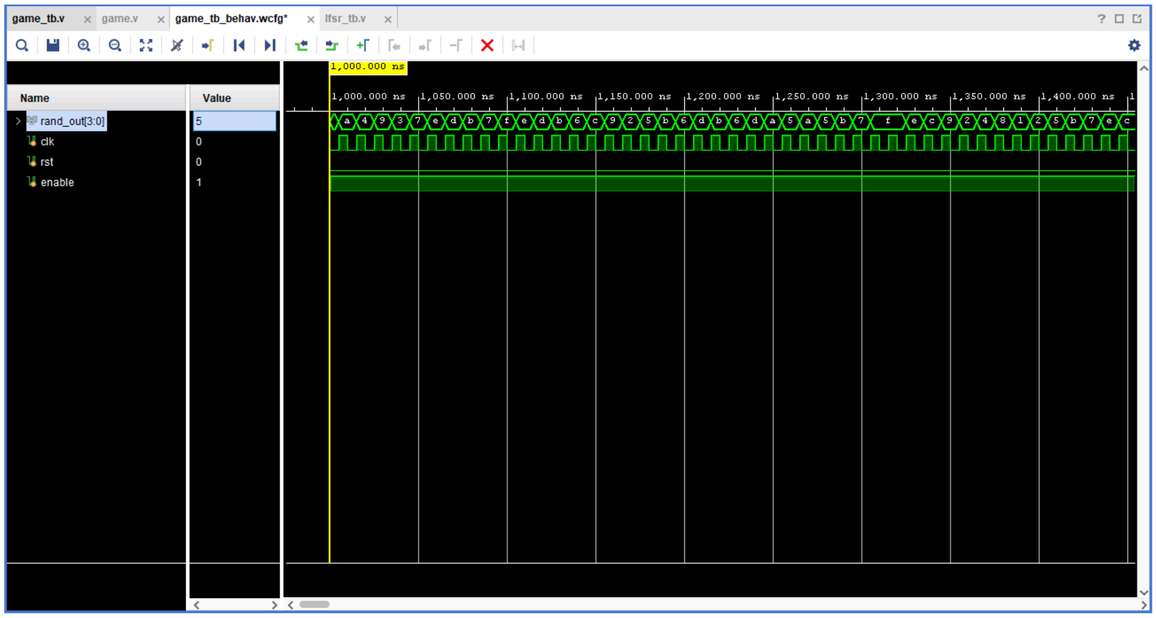
```
//linear feedback shift register
//polynomial, shift, and select adjusted to produce a very long pseudorandom string
module lfsr(
    input wire clk,
    input wire lfsr_rst,
    input wire enable,
    output reg [3:0] random
    );

    reg [7:0] lfsr;
    wire feedback;
    assign feedback = ~(lfsr[7] ^ lfsr[6] ^ lfsr[0] ^ lfsr[3]);

    initial begin
        lfsr <= 8'b10101010; //set to arbitrary non-zero value
    end

    always @(posedge clk or posedge lfsr_rst) begin
        if (lfsr_rst)
            lfsr <= 8'b10101010; //set to arbitrary non-zero value
        else if (enable) begin
            //shift using feedback
            lfsr <= {lfsr[5:0], feedback};
        end
    end

    always @(posedge clk) begin
        random <= lfsr[3:0];
    end
endmodule
```

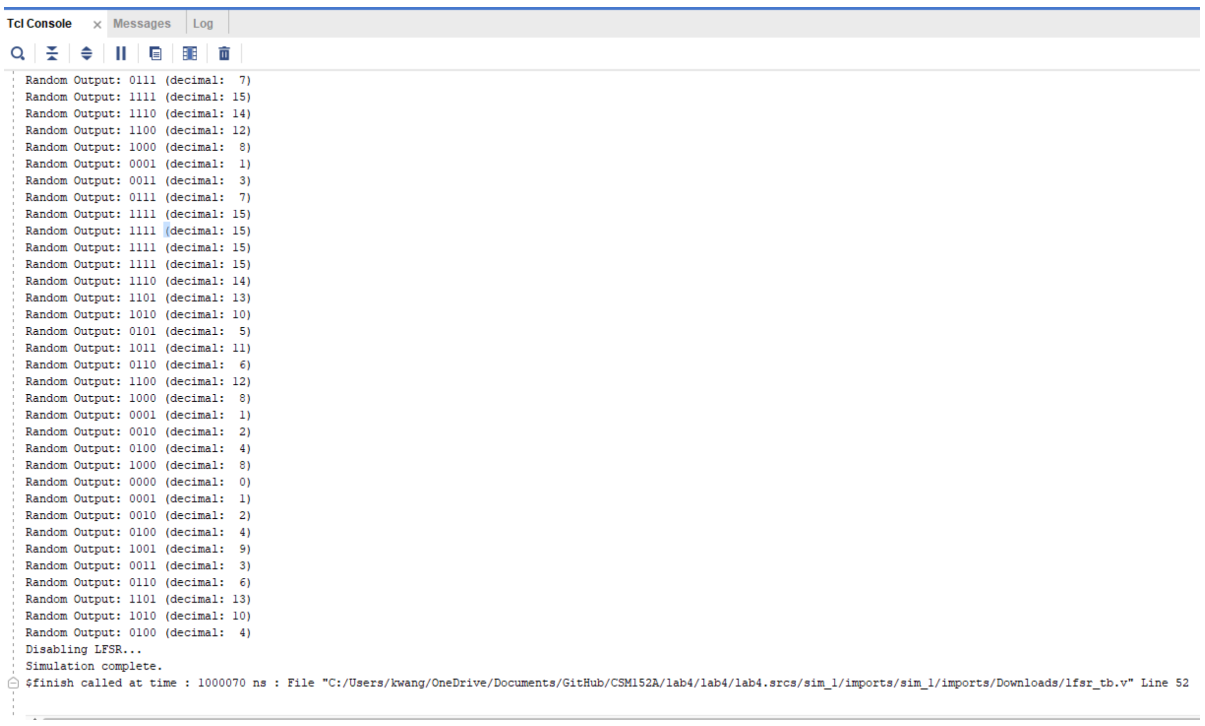


```
//helper module per switch that checks what/how the game should change for specific switch
module switch_edge(
    input switch,
    input switch_last,
    input led,
    output led_new,
    output is_negative,
    output change_point
    );

    assign change_point = switch ^ switch_last; //always have some point difference if switch flips
    assign is_negative = change_point && ~led; //indicated if point difference is positive or negative
    assign led_new = led && ~change_point; //if led is on and switch is flipped, turn off
endmodule
```

Simulation Documentation

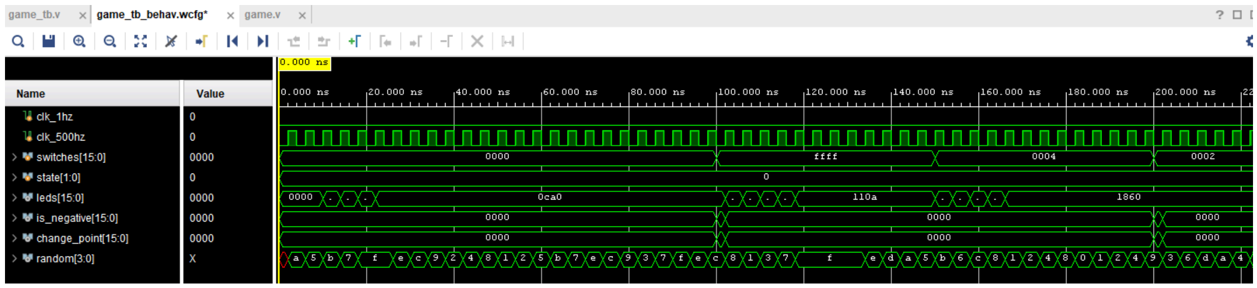
Using a testbench and the log, we tested our clock divider and pseudorandom to ensure that we were getting the expected timings and random numbers. This allowed us to adjust our LFSR random number generator until the results seemed “random enough”, since our first few attempts resulted in the output converging on zero rapidly or forming loops 5-30 numbers long.





```
Random Output: 0111 (decimal:  7)
Random Output: 1111 (decimal: 15)
Random Output: 1110 (decimal: 14)
Random Output: 1100 (decimal: 12)
Random Output: 1000 (decimal:  8)
Random Output: 0001 (decimal:  1)
Random Output: 0011 (decimal:  3)
Random Output: 0111 (decimal:  7)
Random Output: 1111 (decimal: 15)
Random Output: 1111 (decimal: 15)
Random Output: 1111 (decimal: 15)
Random Output: 1111 (decimal: 15)
Random Output: 1110 (decimal: 14)
Random Output: 1101 (decimal: 13)
Random Output: 1010 (decimal: 10)
Random Output: 0101 (decimal:  5)
Random Output: 1011 (decimal: 11)
Random Output: 0110 (decimal:  6)
Random Output: 1100 (decimal: 12)
Random Output: 1000 (decimal:  8)
Random Output: 0001 (decimal:  1)
Random Output: 0010 (decimal:  2)
Random Output: 0100 (decimal:  4)
Random Output: 1000 (decimal:  8)
Random Output: 0000 (decimal:  0)
Random Output: 0001 (decimal:  1)
Random Output: 0010 (decimal:  2)
Random Output: 0100 (decimal:  4)
Random Output: 1001 (decimal:  9)
Random Output: 0011 (decimal:  3)
Random Output: 0110 (decimal:  6)
Random Output: 1101 (decimal: 13)
Random Output: 1010 (decimal: 10)
Random Output: 0100 (decimal:  4)
Disabling LFSR...
Simulation complete.
$finish called at time : 1000070 ns : File "C:/Users/kwang/OneDrive/Documents/GitHub/CSM152A/lab4/lab4/lab4.srcs/sim_1/imports/sim_1/imports/Downloads/lfsr_tb.v" Line 52
```

Using a testbench, we also tested and debugged our main game logic. We were able to catch most logic errors at this point and test a bunch of edge cases without having to upload to the FPGA.



We were relatively confident in our hardware input handling input since we reuse our code from lab 3, so after simulating these modules we moved on to the top level module and did testing and debugging on the FPGA.

Conclusion

To sum up our game design, we achieved a unique recreation of Whack-a-Mole by utilizing a modular design by splitting up the work into a gameplay module, a state manager, and various helper functions to manage each of the hardware details. This was especially key for the button, switch, and LED implementations for modeling the whacking of the moles.

In this lab we learned a lot about implementing complex logic in modules, simulation, and especially debugging. We had quite a lot of difficulty with getting our combinatorial and sequential logic blocks to work together due to syntax and design issues, and struggled with getting modules to correctly feed into each other. I think one thing that could help this lab is providing resources for how to interpret Vivado error messages because we got some extremely cryptic errors where the only search result is somebody asking the same question on a forum and receiving no response. We wrote many testbenches and did a lot of unit testing to resolve our integration issues. Overall, we learned a lot in this lab and CS M152A overall. Thanks Qiming!