ECPE-174 Final Project: Simon Says

University of the Pacific: School of Engineering and Computer Science

Mari Anderson and Felix Pitts

December 10, 2024

Contents

Chapter 1

Project Overview and Objectives

1.1 Overview

The project involves creating a "Simon Says" memory game on a programmable board. This game challenges players to mimic a sequence of LEDs by toggling keys in the correct order. As rounds progress, the sequence becomes longer and more difficult, testing the player's memory and reaction time. The game integrates a visual (LCD Display) interface to enhance user engagement.

1.2 Objectives

1. Develop a Memory game
    - Implement a sequence-based gameplay mechanism where LEDs display an increasing pattern.
    - Require players to input the pattern via keys to progress through rounds.
2. Increase Challenge Progressively
    - Add a new LED to the sequence in each round, making it more challenging for players to recall and replicate.
3. Display Real Time Information
    - Use a LCD Display to provide critical feedback
        - Current round number
        - High score across multiple games
4. Support High Score Tracking
    - Ensure that high scores persist across game sessions, motivating players to achieve better results over time.
5. Test for Scalability
    - Design the game to support up to 100 rounds for testing purposes, even if impractical for real-time play.

Chapter 2

Design Process

2.1 System Design

Our system design began with the following diagram

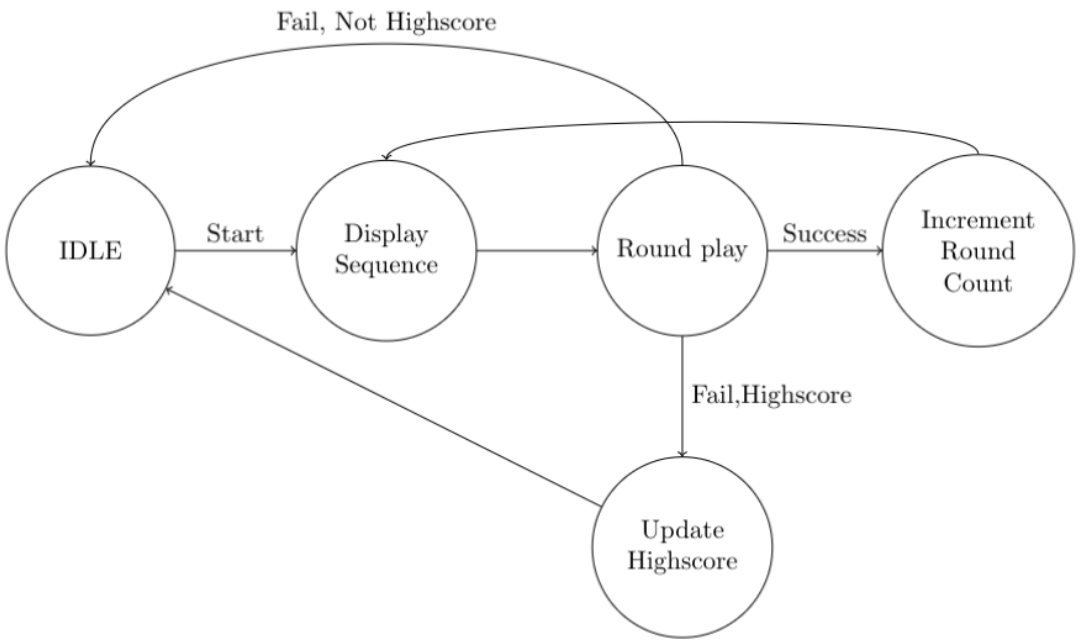


Figure 2.1: Diagram for the States of the Game

The diagram in Figure 1 was the basis for how the FSM that we use to run the game was designed. The Transitions that occur are the following

- **IDLE → Display Sequence:** When any button is pressed on the board
- **Display Sequence → Round Play:** Once the sequence is done displaying, the board accepts user input
- **Round Play → Increment Round Count:** If the user succeeds in replicating the input sequence displayed by the board, the round count increases

- **Increment Round Count → Display Sequence:** Once the round count has increased by one, the sequence is then displayed again but with 1 more term than the sequence displayed in the previous round
- **Round Play → IDLE:** When a player either makes a mistake or takes too long to input the sequence but does not get the high score, the game returns back to an IDLE state
- **Round Play → Update Highscore:** If the highscore was obtained, update the highscore at the end of the round before going back to IDLE
- **Update Highscore → IDLE:** After the highscore has been updated return to IDLE.

While the final FSM does not follow this exactly, it created a structure that made writing the System Verilog code much easier. Some of the differences between the diagram in Figure 1 and the final result would be that even if the highscore is not obtained, the FSM goes into a "Round Lost" state, and checks for the highscore in that state, instead of going to a state if the highscore is obtained. The implementation is explored further in Section 2.3

Another difference is that the round incrementing occurs in the Round Play State when the number of correct inputs matches the number of terms being displayed in the sequence.

```
if (subRound >= currRound) begin
    subRound = 8'd0;
    currRound = currRound + 8'd1;
    next_state = DISPLAY_SEQUENCE;
end else begin
    next_state = ROUND_PLAY;
end
```

Figure 2.2: Code Snippet from `ROUND_PLAY` state that handles Incrementing the Round Count

These differences in the final design did not hinder the functionality of the game, and by keeping the number fo states down to 4, we can represent our states with 2 bit values, and have no unused states.

2.2 Hardware Design

For hardware design we broke the design down into the following parts.

1. **LED Display:**
    - LEDs are used to visually display the game sequence that players need to replicate. Each LED corresponds to a switch for user input.
2. **Key Inputs:**
    - Keys serve as the primary user interface for entering the sequence. They are debounced to ensure accurate input capture. This replaced the initial concept design of using switches.
3. **Seven Segment Display**
    - Displays real-time game information, including the current round number and high score. This replaced the initially planned LCD display to ensure reliability and ease of integration.
4. **Programmable board**
    - Serves as the central processing unit, executing the state machine logic and controlling all hardware components.
5. **External Clock**
    - A clock divider generates game-specific timing signals, ensuring synchronized operation of game states and LED sequences.
6. **Finite State Machine (FSM):**
    - Governs the games behavior, ensuring logical transitions between the states of the game.
    - Takes care of the error handling, ensuring that when the incorrect button is pressed the game goes a failure state before going back to IDLE
7. **Random Sequence Generator:**
    - Creates unique sequences for each round, ensuring variability and challenge for the players

2.3 Finite State Machine Design

For the final FSM that we used for the board hardware there were 4 states. `IDLE`, `DISPLAY_SEQUENCE`, `ROUND_PLAY`, and `ROUND_LOST`. The roles that the states filled was the following

1. `IDLE`
    - Waits for any user input on the 4 buttons on the FPGA board to start the game
    - Keeps the game at an IDLE state when there is no Input
    - Sets the start signal to 0 so that the system can generate a random sequence when the game is not being played

```
IDLE:
begin
    currRound <= 8'd0;
    play_led <= 1'b0;
    start <= 1'b0;
    if (synced_keys != 4'b0000) begin
        next_state <= DISPLAY_SEQUENCE;
        currRound <= 8'd1;
    end else begin
        next_state <= IDLE;
    end
end
```

Figure 2.3: IDLE State System Verilog Code

2. `DISPLAY_SEQUENCE`
    - Starts the display at the first element of the sequence when coming from IDLE or ROUND PLAY
    - Lights up one of the LEDS depending on what the value of the sequence at that point is
    - When done displaying the sequence transitions into ROUND PLAY

```
DISPLAY_SEQUENCE:
begin
    start <= 1'b1;
    play_led <= 1'b0;
    if (prev_state == IDLE || prev_state == ROUND_PLAY) begin
        subRound <= 8'd0;
    end else begin
        subRound <= subRound;
    end
    if (subRound >= currRound )begin
        subRound <= 8'd0;
        next_state <= ROUND_PLAY;
    end else begin
        case (genned_sequence[subRound])
            2'b00: key_leds <= 4'b0001; //LEDG0
            2'b01: key_leds <= 4'b0010; //LEDG2
            2'b10: key_leds <= 4'b0100; //LEDG4
            2'b11: key_leds <= 4'b1000; //LEDG6
            default: key_leds <= 4'b0000;
        endcase
        seq_disp_count <= seq_disp_count + 8'd1;
        if (seq_disp_count == 5'b11111) begin
            subRound <= subRound + 8'd1;
            seq_disp_count <= 5'd0;
        end else begin
            subRound <= subRound;
        end
        next_state <= DISPLAY_SEQUENCE;
    end
end
```

Figure 2.4: `DISPLAY_SEQUENCE` State System Verilog Code

3. ROUND_PLAY

- Starts taking user input for the start of the sequence when the round begins
- Resets the round timer to 0 at the start of the round and then counts up the longer the round has been progressing
- If the input is wrong or the timer ran out, transitions to ROUND_LOST
- Goes to DISPLAY_SEQUENCE when the round is successfully completed

```
ROUND_PLAY:
begin
    key_leds = synced_keys;
    start = 1'b1;
    play_led = 1'b1;
    //Start of round, set subround counter to 0
    if (prev_state == DISPLAY_SEQUENCE) begin
        subRound = 8'd0;
        round_timer = 12'd0;
    end else begin
        subRound = subRound;
    end
    //If too much time in a round has been taken, lose
    if (round_timer == 12'hFFF) begin
        next_state = ROUND_LOST;
        round_timer = 12'd0;
    end else begin
        next_state = next_state;
        round_timer = round_timer + 12'd1;
    end
    //If Input, Check that input is correct
    if (synced_keys != 4'b0000) begin
        case (synced_keys)
            4'b0001: user_input = 2'b00;
            4'b0010: user_input = 2'b01;
            4'b0100: user_input = 2'b10;
            4'b1000: user_input = 2'b11;
            default: next_state = ROUND_PLAY;
        endcase
        //Compare against the current value in the sequence
        if (user_input == genned_sequence[subRound]) begin
            subRound = subRound + 8'd1;
            if (subRound >= currRound) begin
                subRound = 8'd0;
                currRound = currRound + 8'd1;
                next_state = DISPLAY_SEQUENCE;
            end else begin
                next_state = ROUND_PLAY;
            end
        end else begin
            next_state = ROUND_LOST;
        end
    end else begin
        //No Input, Do not increment subRound
        subRound = subRound;
    end
end
```

Figure 2.5: ROUND_PLAY State System Verilog Code

4. ROUND_LOST

   - Checks if the Highscore was beaten, if so updates the value stored.
   - Prepares the system to return back to IDLE

```
ROUND_LOST:
begin
    if (currRound > high_score) begin
        high_score <= currRound;
    end else begin
        high_score <= high_score;
    end
    key_leds <= 4'b0000;
    play_led <= 1'b0;
    next_state <= IDLE;
    currRound <= 8'd0;
    round_timer <= 12'd0;
end
```

Intel Quartus generated a state diagram for the FSM we created but due to the code being in an always @() block instead of an always_comb block the FSM Diagram generated is incorrect from what actually happens with the FSM.

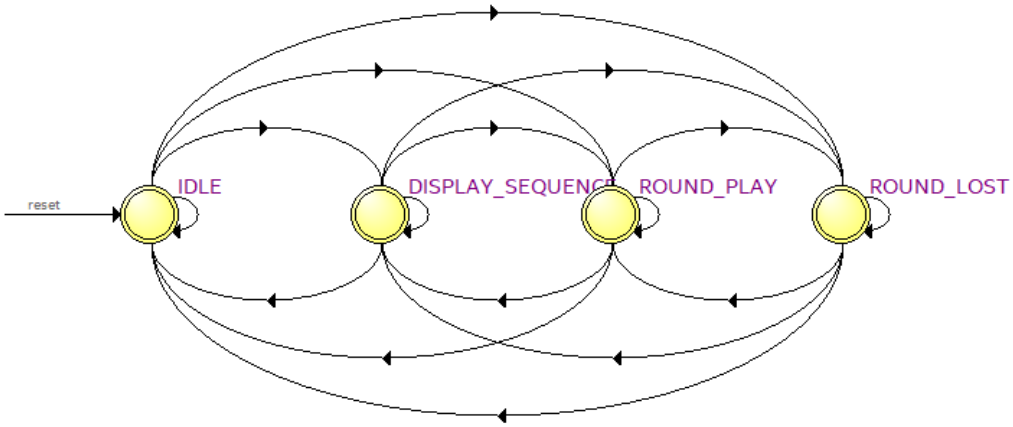


Figure 2.6: FSM Diagram Generated From Quartus

The reason an always @() block was used was that we had to use sequential logic for the ROUND_PLAY state to get it to function as intended. We did try to use an always_comb block, however I could not get it to compile despite many hours of debugging. The Full FSM Code, which was also the Top Level Module can be found in the Appendix

2.4 Random Sequence Generation

2.4.1 Seed Generation

For the random number generation we need a way to generate the input value for the random number generator to be able to work. The way we do this is by counting the number of clock cycles that have occurred on the hardware. In the case of the FPGA used for the game, the Terasic DE2-115, that clock is 50MHz, so it is fast enough to where no human could hope to manipulate the random number generation with any reliable consistency without the use of external hardware directly connected to the system.

```
module sequence_generation(
    input logic clk,
    output logic [31:0] seed
);

logic [31:0] a = 32'd0;
always (@posedge clk) begin
    seed <= a + 32'd1;
    a <= seed;
end

endmodule
```

Figure 2.7: Seed Generation System Verilog Code

Once the seed has been generated, we can then go create a pseudo random number.

2.4.2 Random Number Generation

For the Random Number Generation we used an XOR-Shift algorithm, as it was very straightforward to translate from C into System Verilog

```
#include <stdint.h>

struct xorshift32_state {
    uint32_t a;
};

/* The state must be initialized to non-zero */
uint32_t xorshift32(struct xorshift32_state *state)
{
        /* Algorithm "xor" from p. 4 of Marsaglia, "Xorshift RNGs" */
        uint32_t x = state->a;
        x ^= x << 13;
        x ^= x >> 17;
        x ^= x << 5;
        return state->a = x;
}
```

Figure 2.8: C Code for XOR-Shift RNG Generation (From Wikipedia)

How the algorithm in Figure 1 works is that it takes an unsigned 32-bit integer as an input, then performs

XOR operations with the initial value, and that value bit shifted by an arbitrary amount. For example the code in Figure 1 does the following steps

- set x to the input a
- set x to x XOR (x shifted left 13 bits)
- set x to x XOR (x shifted right 17 bits)
- set x to x XOR (x shifted left 5 bits)
- return x

Because this algorithm uses binary bit operations translating this to System Verilog is very straightforward.

```
module xorshift(
    input logic [31:0] a,
    output logic [31:0] x
);

logic [31:0] x0, x1, x2;
always_comb begin
    x0 <= a;
    if (x0 === 32'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx) begin //Checks if x0 isn't resolved
        x1 <= 32'd1;
    end else begin
        x1 <= x0 ^ (x0 << 13);
    end
    if (x1 === 32'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx)begin //Checks if x1 isn't resolved
        x2 <= 32'd1;
    end else begin
        x2 <= x1 ^ (x1 >> 17);
    end
    x <= x2 ^ (x2 << 5);
end

endmodule
```

Figure 2.9: XOR-Shift implementation in System Verilog

The main difference between Figure 1 and Figure 2 is that for Figure 2 we have to declare dedicated intermediate values due to the mechanics of System Verilog. With our modules for seed and RNG generation we can put them together to form a module that takes handles the random number generation.

```
module rng_generation(
    input logic clk,
    output logic [31:0] val
);

logic [31:0] seed;
seedgenerator seeding (.clk(clk), .seed(seed));
xorshift generator(.a(seed), .x(val));

endmodule
```

Figure 2.10: Random Number Generation Module System Verilog Code

2.4.3 Sequence Generation

```
module sequence_generation(
    input logic clk,
    input logic start,
    output logic [1:0] game_sequence [100]
);

logic [31:0] val;
int count = 32'd0;
logic [31:0] intermed_val;
rng_generation generator(.clk(clk), .val(val));

//Generate Sequence Here:
always @(posedge clk) begin
    if (start) begin
        count <= count;
        game_sequence <= game_sequence;
    end else begin
        if (count >= 32'd100) begin
            count <= 32'd0;
            intermed_val <= val % 4;
            game_sequence[count] <= intermed_val[1:0];
        end else begin
            count <= count + 32'd1;
            intermed_val <= val % 4;
            game_sequence[count] <= intermed_val[1:0];
        end
    end
end

endmodule
```

Figure 2.11: Sequence Generation System Verilog Code

Here the code in Figure 2.11 takes in the RNG value generated in Figure 2.10, and uses that to generate a sequence of 100 values between 0 and 3. The sequence moves throughout the array until the `start` signal is high, which is triggered when the round starts so that the sequence does not change on the user while they are playing the game.

Chapter 3

Testing Procedure

3.1 Seedgenerator Testbench

```
1  module seedgenerator_testbench();
2  logic clk = 1'b0;
3  logic [31:0] a;
4
5  seedgenerator seedGen (.clk(clk), .seed(a));
6
7  always begin
8      #10 clk = ~clk;
9  end
10
11 initial begin
12     $monitor("clk: ,  a: ", clk, a);
13 end
14
15 endmodule
```

Figure 3.1: Seed Generator Testbench System Verilog Code

The file `seedgenerator_testbench.sv` is designed to test the functionality of the seed generator module. Here's a breakdown of its purpose and components:

1. **Clock Signal Generation:**
   - A clock signal (clk) is toggled every 10 time units using an always block (`#10 clk = clk;`). This simulates a regular clock input to the seedgenerator module.
2. **Module Under Test (MUT):**
   - The seedgenerator module is instantiated with two ports:
     - clk: The clock signal generated in the testbench.
     - seed (mapped to a): An output signal from the seedgenerator module, expected to provide a generated seed value.
3. **Simulation Monitoring**
   - The `$monitor` task is used to continuously display the values of the clock (clk) and the generated seed (a) during simulation. This provides real-time insight into how the module responds to the clock input.

4. **Purpose**

   - The testbench evaluates how the seedgenerator module generates or updates the seed (a) in response to the clock signal. By observing the outputs, you can verify whether the module behaves as expected.

This testbench is straightforward, focusing on ensuring that the clock signal and module outputs function correctly in a basic simulation environment.

3.2 XORshift Testbench

```
module xorshift_testbench();
logic [31:0] a,x;
xorshift testMod(.a(a), .x(x));

initial begin
    int file;
    file = $fopen("./xor_test_output.txt", "w");
    a <= 32'd0;
    #150 $monitor("%b, %b", a, x);
    for (int i = 0; i < 32'd4294967295; i++) begin
        #5 a <= a + 32'd1;
        //for validating that the RNG follows a uniform distribution
        $fwrite(file, "%d\n, ", x % 4);
    end
    $fclose(file);
    $stop;
end
endmodule
```

Figure 3.2: System Verilog Code for the xorshift Test Bench

From there to reduce file size to make computing information on this data easier the output file is ran throughout the following python script:

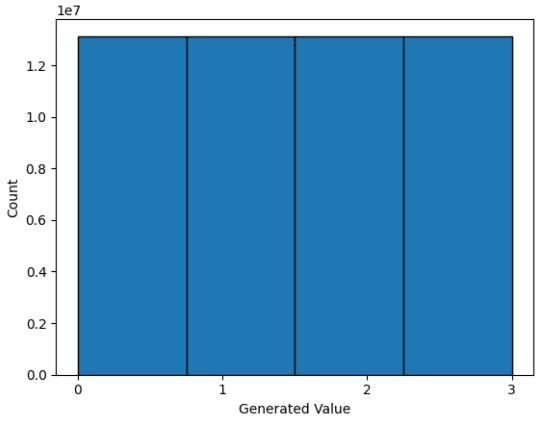
```
#Removes excess whitespace from files
import argparse

def main():
    parser = argparse.ArgumentParser(prog = "Questa Output Fix")
    parser.add_argument('--input',type=str,help="Filename/Path of the Input File")
    args = parser.parse_args()
    inFile = open(args.input, "r")
    outFile = open(args.input[0:(len(args.input) - 4)] + "_fixed.txt", "w")
    for line in inFile:
        newLine = line.replace(" ", "", 99)
        newLine = newLine.replace("\n", "")
        outFile.write(newLine)

main()
```

Figure 3.3: Python Script to Reduce the File Size of the Testbench Output

Questa adds a lot of whitespace into the output file, which adds no extra information but increases the file size dramatically, which is why we need the script in Figure 4. Running our output file through the script changed the file size from 682.7 MB to 105.0 MB, so a very large reduction in file size. Then to validate that the output is a uniform distribution the fixed output file is run through the following Python code in a Jupyter Notebook.

```
from matplotlib import pyplot as plt
from collections import Counter
dist = open("xor_test_output_fixed.txt", "r")
newdist = []
for line in dist:
    a = line.split(",")
    for val in a:
        newdist.append(int(val))
plt.hist(newdist,4, edgecolor= 'black')
plt.show()
counts = Counter(newdist)
print(counts)
```

Using that code after simulating around 52.5 million generated values creates the following plot:



From this plot we can see that we get a distribution that is uniform. Looking at the number of times each value (0,1,2,3) occurred we can see that the differences in how many times each number is generated is very small

```
0: 13128782, 1: 13128785, 2: 13128783, 3: 13128783
```

Figure 3.4: Number of Times each Generated Value Occurred in the Simulation

3.3 Sequence Generation Testbench

```
module sequence_testbench();

logic clk = 1'b0;
logic [1:0] game_sequence [100];
always begin
    #10 clk <= ~clk;
end
sequence_generation testGame (.clk(clk), .game_sequence(game_sequence));


initial begin
    int file;
    int count = 32'd0;
    #10000 file = $fopen("./test_sequences.csv", "w");
    for (int i = 0; i < 1000; i++) begin
        #25 if (count >= 32'd99) begin
                $fwrite(file, "%d, ", game_sequence[count]);
                $fwrite(file, "\n");
                count <= 32'd0;
            end else begin
                $fwrite(file, "%d, ", game_sequence[count]);
                count <= count + 32'd1;
            end
    end

    $stop;
end
endmodule
```

Figure 3.5: Sequence Generator Testbench System Verilog Code

The testbench for the `sequence generation` module (`sequence_generation`) is designed to verify that the sequence of game states is being generated and stored correctly in the `game_sequence` array. The module under test, `sequence_generation`, is instantiated with a clock (clk) and the output `sequence array` (`game_sequence`). The sequence output is stored and logged in a CSV file for analysis. The module generates 10, 100 value sequences and by storing them in a CSV file we were able to use a spreadsheet software to validate that the sequences were 100 values long, and that the wrap around worked. In summary, our testbench effectively verified the core functionality of the sequence generation module. We chose a straightforward approach of logging the output to a file and ensuring that the sequence generation was correct across multiple rounds. By focusing on key test cases, such as full sequence generation and boundary conditions, we ensured the robustness of the system while keeping the testbench design simple and efficient.

3.4 Top Module Testbench

The `Game_tb` testbench was designed to thoroughly test the Game module, which simulates a "Simon Says" game where players interact with the system using keys and observe outputs on LEDs and displays. The testbench was structured to cover several different scenarios of key presses to validate the correct operation of the game.

3.4.1 Testbench Design Approach

1. **Clock Generation**
   - A clock signal is generated with a period of 10 ns (100 MHz) using the forever construct. This ensures the system operates at a consistent frequency for all test cases.
2. **Stimulus**
   - A variety of test cases were written to simulate key presses and observe the corresponding outputs. This includes:
     - **No Key Pressed:** The system is initialized with no keys pressed (keys = 4'b0000), testing the default behavior of the game.
     - **Single Key Pressed:** A test case where only one key is pressed (keys = 4'b0001), verifying that individual key presses are correctly recognized.
     - **Multiple Keys Pressed:** Two keys are pressed (keys = 4'b1010), testing the system's ability to handle multiple inputs.
     - **All Keys Pressed:** A case where all keys are pressed (keys = 4'b1111), ensuring the system can process all inputs simultaneously.
3. **Coverage**
   - **Outputs Monitoring:** For each test, the testbench monitors both the LED outputs (`play_led` and `key_leds`) and the seven-segment display outputs (display0 to display7). This ensures the visual feedback from the game is correct based on the key inputs.
   - **Randomization:** The testbench does not utilize randomization as the main goal is to cover specific, deterministic key press cases. This approach is appropriate as it allows for a predictable evaluation of the system's response to particular input scenarios.
   - **Reset and Stability:** The design tests multiple key press scenarios in sequence, ensuring the system's stability and correctness after each change in input. The system's reaction to different input combinations (none, one, or many keys) is observed, covering normal operating conditions.
4. **Additional Considerations**
   - The testbench checks the functionality of both LEDs and displays to ensure all visual indicators work as expected for a range of key inputs.
   - The use of the `#10` delay between key changes ensures that the system has time to process the inputs and update the outputs, simulating a real-time game environment.

By writing tests that check individual and combined key presses, the testbench provides a comprehensive assessment of the Game module, verifying its ability to respond to expected input patterns and display the correct outputs.

Chapter 4

Results

We were unable to include memory into our project however we were still able to save the current high score of the game. We were also unsuccessful in getting the LCD screen to display any results or feedback as this was a challenging aspect of the final project. We instead resulted in using the Seven Segment display which was much easier as we were running out of time. The game was successful in working however in terms of randomizing the pattern of the game we usually got a consistent pattern due to the RNG only randomizing 4 values.

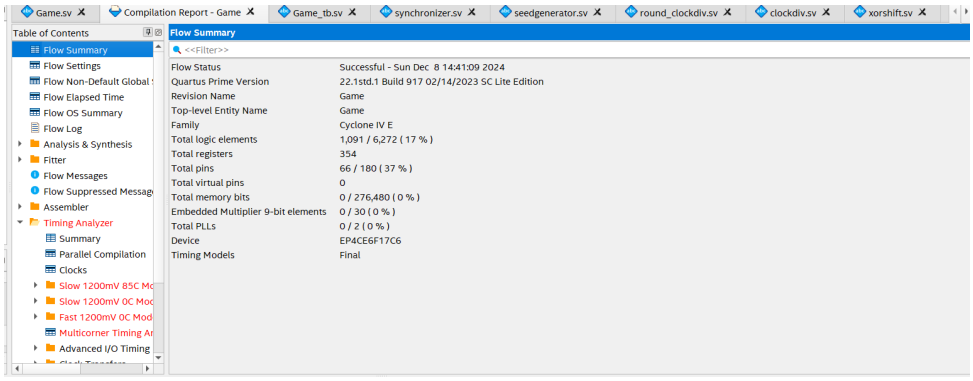


Figure 4.1: Intel Quartus Compilation Report

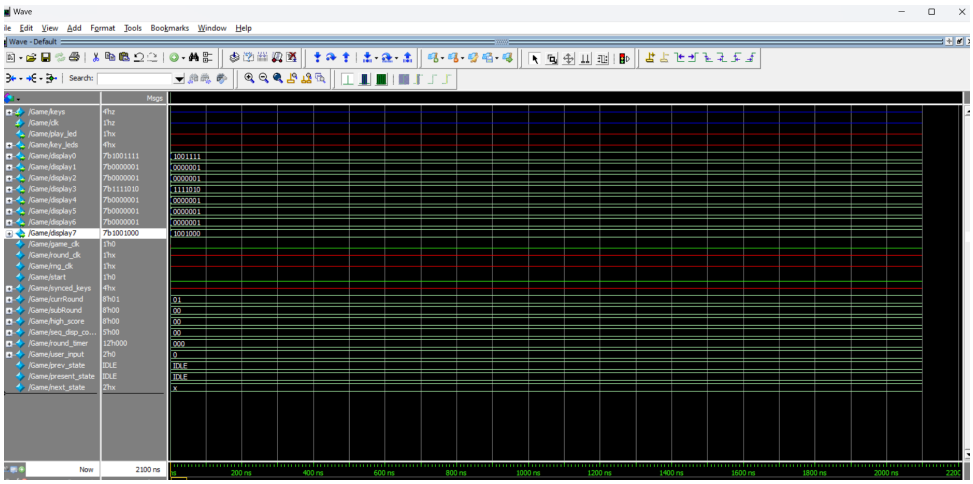


Figure 4.2: Top Level Waveform

Chapter 5

Analysis

The design operated successfully, meeting its primary objectives with some minor adjustments to the initial concept. The system was user-friendly and capable of supporting up to 100 rounds, though certain limitations were identified. For instance, user data could not be retained if the system was powered off, and there was no reset functionality unless the user lost the game. Despite these constraints, the game effectively randomized patterns on the board. However, the random number generator (RNG) had limited randomness due to the small number of keys used, which resulted in less variation over time.

From a performance perspective, the timing was sufficient for smooth gameplay, with acceptable latency and throughput to support user interactions. Space requirements were minimal, allowing the design to operate efficiently within the hardware constraints.

For future enhancements, we aim to include an LCD screen to improve user interaction and display additional game details. We also plan to implement a more advanced RNG to produce better randomization, particularly within smaller subsets, reducing sequential patterns. Additional features such as sound integration could make the game more immersive and engaging. A potential multiplayer mode would allow one user to create a pattern while the other attempts to replicate it, fostering competition and collaboration. This feature could also have educational benefits, enhancing pattern recognition and memory skills.

Feedback from the poster session highlighted the importance of incorporating these features to improve user experience and broaden the design's applications. These insights will guide our future iterations of the system.

Chapter 6

Task Breakdown

We collaboratively agreed on the initial concept of designing a "Simon Says" game and worked together to enhance the project by incorporating unique features to distinguish our work from other groups. Overall, our teamwork was effective, resulting in a functional and presentable final design.
Responsibilities:

- **Mari's Contributions**
  - Led the system design, focusing on key functionalities such as the finite state machine (FSM), random number generator (RNG), and sequence generation. Her work ensured the core game logic was functional and scalable.
- **Felix's Contributions**
  - Managed the written report, prepared the presentation materials, and worked on integrating the LCD display into the design. However, when the LCD display proved unsuccessful, we adapted the design to utilize a Seven-Segment Display, ensuring the game interface was operational for the final presentation.

Through effective collaboration and division of tasks, we were able to address challenges and present a fully functional design, showcasing our problem-solving and teamwork skills.

Chapter 7

Conclusion

The "Simon Says" game project demonstrated the integration of hardware and software systems to create an engaging and interactive memory-based challenge. We delivered a functional and scalable game by designing a progressively challenging gameplay mechanism, incorporating visual and textual feedback through LEDs and a Seven-Seg Display, and ensuring high-score tracking. Through this project, we gained hands-on experience in developing state machines, synchronizing hardware inputs, and managing persistence for user data. Additionally, we enhanced our problem-solving and debugging skills while working on real-time systems. This experience reinforced the importance of modular design and rigorous testing in achieving a reliable and user-friendly product. Overall, this project deepened our understanding of embedded systems and their potential for creating interactive applications.

Chapter 8

Links and References

Project Github `https://github.com/azeleaButterfly/ECPE174-Final-Project` XORshift RNG: `https://en.wikipedia.org/wiki/Xorshift`

Appendix A

Other Code

A.1 Clock Divider

```
/*************************************************************
 * SystemVerilog Masterclk clock divider
 *
 * Provides framework for clock divider code to step down
 *  the 50MHz internal clock
 * Set the XXX value in the define for the appropriate division
 *
 * Author: Elizabeth Basha
 * Date: 09/04/2013
 */

 module clockdiv(input logic iclk,
                                                output logic oclk);

        // define the parameter for number of clock edges to count
        const int HALF_OF_CLK_CYCLE_VALUE = 32'd781250; // 32Hz

        // internal variables for clock divider
        int count = 32'd0;
        logic clkstate = 1'b0;

        // generate clock signal
        always_ff @(posedge iclk)
                if(count == (HALF_OF_CLK_CYCLE_VALUE-1))
                begin
                        // we have seen half of a cycle, toggle the clock
                        count <= 32'd0;
                        clkstate <= ~clkstate;
                end else begin
                        count <= count + 32'd1;
                        clkstate <= clkstate;
                end

        assign oclk = clkstate;

 endmodule
```

A.2 Synchronizer Code

```
/*
Code From Umith Chandra, ECPE-174 TA
*/
module synchronizer(
input logic key, clk,
output logic ctrl
);
logic Q0,Q1,Q2;

always_ff @(posedge clk) begin  //synchronizer ff2
    Q0 <= key;
    Q1 <= Q0;
    Q2 <= Q1;
end

assign ctrl = (!Q1 && Q2);


endmodule
```

A.3 Full Top Level Module + FSM Code

Due to the width of the text in the file, a smaller font size had to be used

```
module game (
    input logic [3:0] keys,
    input logic clk,
    output logic play_led,
    output logic [3:0] key_leds,
    output logic [0:6] display0,
    output logic [0:6] display1,
    output logic [0:6] display2,
    output logic [0:6] display3,
    output logic [0:6] display4,
    output logic [0:6] display5,
    output logic [0:6] display6,
    output logic [0:6] display7
);

/*External Module Instantiation*/
sequence_generation sequence_gen (.clk(clk), .start(start), .game_sequence(genned_sequence));
clockdiv game_clock(.iclk(clk), .oclk(game_clk));
synchronizer key0_sync(.clk(game_clk), .key(keys[0]), .ctrl(synced_keys[0]));
synchronizer key1_sync(.clk(game_clk), .key(keys[1]), .ctrl(synced_keys[1]));
synchronizer key2_sync(.clk(game_clk), .key(keys[2]), .ctrl(synced_keys[2]));
synchronizer key3_sync(.clk(game_clk), .key(keys[3]), .ctrl(synced_keys[3]));
three_disp_seven_seg round_disp(.a(currRound), .display0(display0), .display1(display1), .display2(display2));
three_disp_seven_seg highscore_disp(.a(high_score), .display0(display4), .display1(display5), .display2(display6));


/*State Type and Variable Declaration */
typedef enum logic [1:0] {IDLE, DISPLAY_SEQUENCE, ROUND_PLAY, ROUND_LOST} State;
State prev_state = IDLE;
State present_state = IDLE;
State next_state;

/*Variable Declaration*/
logic [1:0] genned_sequence [100];
logic game_clk, round_clk, rng_clk;
logic start = 1'b0;
logic [3:0] synced_keys;
logic [7:0] currRound = 8'd1;
logic [7:0] subRound = 8'd0;
logic [7:0] high_score = 8'd0;
logic [4:0] seq_disp_count = 5'b00000; //How many times to display the led
logic [11:0] round_timer = 12'd0;
```

```
logic [1:0] user_input = 2'b00;
/*FSM Logic*/
always @(posedge game_clk) begin
    case (present_state)
        IDLE:
        begin
            currRound <= 8'd0;
            play_led <= 1'b0;
            start <= 1'b0;
            if (synced_keys != 4'b0000) begin
                next_state <= DISPLAY_SEQUENCE;
                currRound <= 8'd1;
            end else begin
                next_state <= IDLE;
            end
        end

                DISPLAY_SEQUENCE:
        begin
            start <= 1'b1;
            play_led <= 1'b0;
            if (prev_state == IDLE || prev_state == ROUND_PLAY) begin
                subRound <= 8'd0;

            end else begin
                subRound <= subRound;
            end
            if (subRound >= currRound )begin
                subRound <= 8'd0;
                next_state <= ROUND_PLAY;
            end else begin
                case (genned_sequence[subRound])
                    2'b00: key_leds <= 4'b0001; //LEDG0
                    2'b01: key_leds <= 4'b0010; //LEDG2
                    2'b10: key_leds <= 4'b0100; //LEDG4
                    2'b11: key_leds <= 4'b1000; //LEDG6
                    default: key_leds <= 4'b0000;
                endcase
                seq_disp_count <= seq_disp_count + 8'd1;
                if (seq_disp_count == 5'b11111) begin
                    subRound <= subRound + 8'd1;
                    seq_disp_count <= 5'd0;
                end else begin
                    subRound <= subRound;
                end
                next_state <= DISPLAY_SEQUENCE;
            end
        end

        ROUND_PLAY:
        begin
            key_leds = synced_keys;
            start = 1'b1;
            play_led = 1'b1;
            //Start of round, set subround counter to 0
            if (prev_state == DISPLAY_SEQUENCE) begin
                subRound = 8'd0;
                round_timer = 12'd0;
            end else begin
                subRound = subRound;
            end
            //If too much time in a round has been taken, lose
            if (round_timer == 12'hFFF) begin
                next_state = ROUND_LOST;
                round_timer = 12'd0;
            end else begin
                next_state = next_state;
                round_timer = round_timer + 12'd1;
            end
            //If Input, Check that input is correct
            if (synced_keys != 4'b0000) begin
                case (synced_keys)
                    4'b0001: user_input = 2'b00;
                    4'b0010: user_input = 2'b01;
                    4'b0100: user_input = 2'b10;
                    4'b1000: user_input = 2'b11;
                    default: next_state = ROUND_PLAY;
                endcase
                //Compare against the current value in the sequence
                if (user_input == genned_sequence[subRound]) begin
                    subRound = subRound + 8'd1;
                    if (subRound >= currRound) begin
```

```
                        subRound = 8'd0;
                        currRound = currRound + 8'd1;
                        next_state = DISPLAY_SEQUENCE;
                    end else begin
                        next_state = ROUND_PLAY;
                    end
                end else begin
                    next_state = ROUND_LOST;
                end
            end else begin
                //No Input, Do not increment subRound
                subRound = subRound;
            end

        end


        ROUND_LOST:
        begin
            if (currRound > high_score) begin
                high_score <= currRound;
            end else begin
                high_score <= high_score;
            end
            key_leds <= 4'b0000;
            play_led <= 1'b0;
            next_state <= IDLE;
            currRound <= 8'd0;
            round_timer <= 12'd0;
        end
                default:
        begin
            play_led <= 1'b0;
            key_leds <= 4'b0000;
            next_state <= IDLE;
            start <= 1'b0;
            subRound <= 8'd0;
            currRound <= 8'd0;
            round_timer <= 12'd0;
            high_score <= high_score;
        end
    endcase
end

always_ff @(negedge clk) begin
    prev_state <= present_state;
    present_state <= next_state;
end


always_comb begin
    display7 <= 7'b1001000; //Highscore H
    display3 <= 7'b1111010; //Round r
end

endmodule
```