Lab 8 Report

The Simon State Machine Part 2

Submitted to:

Inst: Gregg Chapman

GTA: Artun Sel

Created by:

Group #21

Ty Fredrick

Kevin O’Neill

Stephen Connair

ECE 2060

Section 9192 - Tues, 8 AM

The Ohio State University

Columbus, OH

Lab Date: 2 December 2025

Group 21 – Ty Fredrick, Kevin O’Neill, Stephen Connair
Instructor – Gregg Chapman, GTA – Artun Sel

The Simon State Machine Part 1
12/12/2025

Executive Summary

In this lab, the team expanded on the prior lab experience with Quartus Prime, ModelSim, the Altera DE2 board, and the Simon Box by integrating audio lights, and control logic into a completed Simon State Machine game. The team built on circuitry developed in previous labs, replacing the TEST_Controller with a provided System Controller VHDL file, which the team edited and implemented in the circuitry to coordinate the sequencing of the lights, audio output, and interaction with the user. This lab enhanced the team’s understanding of integrated digital systems, hierarchical design, and interaction between schematic components and VHDL-based state machines.

To build the complete Simon game, the team incorporated four major components into the existing CODEC: the Button_Debounce, RandomNumberGenerator, Audio synthesizer, and the System_Controller. The debounce circuit ensured stable button inputs, the random number generator produced the sequence of colors the player needed to replicate, and the audio block generated the audio tones for each button as well as an error tone for the failed game state. These modules were connected through the shared clock and incorporated into the System Controller, which controlled the game’s behavior. The controller displayed a sequence when started and monitored player input, determining a win or loss based on whether the user correctly repeated the light sequence pattern. After compiling and loading the design onto the DE2 board, the system performed as planned. The lights displayed a sequence, tones played through the speaker, and the controller responded appropriately to correct and incorrect button presses, demonstrating a complete Simon State game project.

The lab successfully integrated multiple hardware elements into a complete interactive system. This strengthened the team’s understanding of digital logic, state-machine control, and coordination of audio and visual outputs. Future improvements could include clearer documentation of signal naming, additional testing of user interactions, and potential enhancements and difficulty levels. The project demonstrated the team’s ability to combine VHDL-based logic with schematics to create a cohesive digital program.

Introduction

In this lab the team built on prior work with Quartus Prime, ModelSim, the Altera DE2 board, and the Simon Game Box by using the Simon Game Box in combination with a speaker to create a light sequence and audio response. The team accomplished this by working with existing circuitry, replacing the TEST_Controller with the System Controller VHDL file provided. The System Controller VHDL file was edited and implemented to create a fully functional Simon State Machine game.

Experimental Methodology

The team constructed the Simon game logic from a provided skeleton, called the CODEC, to which several logical elements were adjoined, to be discussed in sequence below. The elements crafted by the team themselves were linked to the existing structure by the AUD_DACLRCK output of the DE2_I2Sound symbol, which served as the clock input to all the elements, and the 16-bit output of the original Audio symbol was fed into the AUD_outR and AUD_outL inputs of the DE2_I2Sound symbol. The symbols added to the skeleton were four in number, each of which will be considered in their turn: the RandomNumberGenerator, the Button_Debounce, the Audio, and the System_Controller symbols.

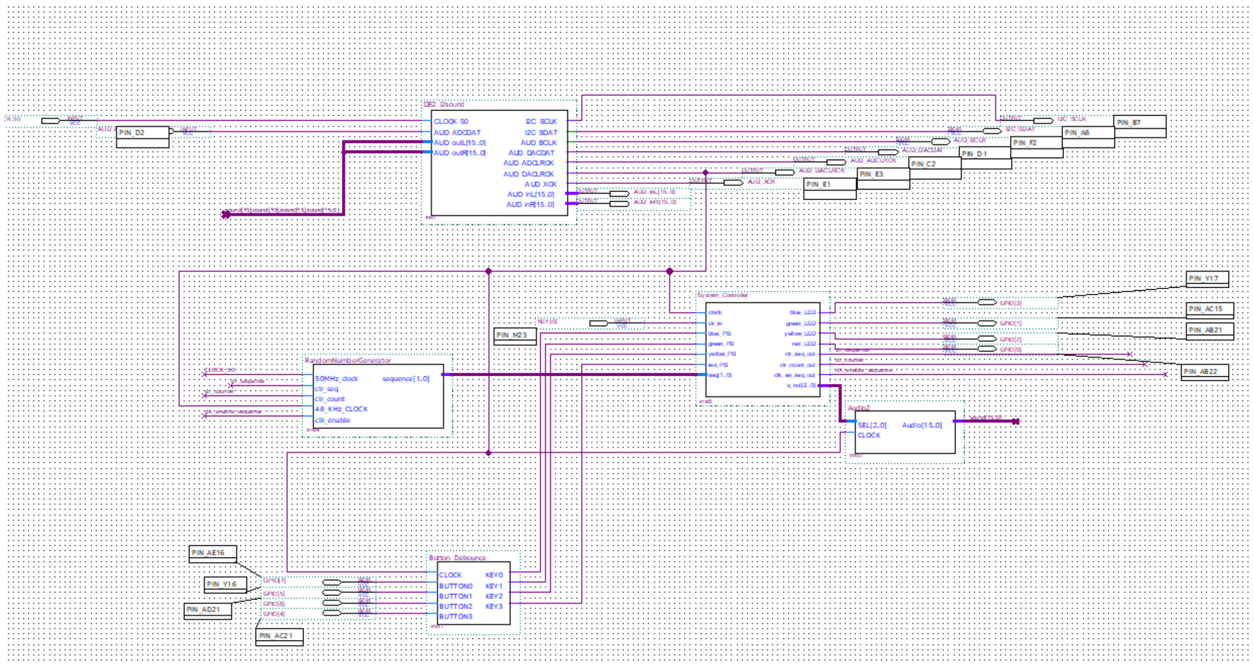


Figure 1: Top-Level Schematic with System_Controller

The Button Debounce

The team derived a Stretch Timer megafunction from the LPM_Counter template with the specifications of a q output bus of 16 bits, a clock enable input, a "clear" asynchronous input, the generation of a netlist, and the creation of a Quartus Prime symbol file. All other specifications were left in their default state. A similar process was undertaken for the creation of the stretch counter, derived from the LPM_Comparator archetype, with the data a and data b lines set to 16 bits, one of the comparanda to be a constant, namely 48000 as ordained in the lab document, the generation of a netlist, and the creation of a Quartus Prime symbol file. The two megafunctions were then added to the block schematic diagram of the button debounce circuit made in the previous lab, with the output of the Stretch Timer being fed into the Stretch Counter and the output of the Stretch Counter being fed into the "clear" asynchronous input of the Stretch Timer as well as each of the NAND gates.

The Random Number Generator

In the RandomNumber Generator file, a new form of the LPM_Counter archetype was designed, called Counter1, with a 16-bit output bus, which fed in to both the SevenSegment block and a random multiplexor derived from the LPM_MUX archetype with pipelining disallowed. The 16-bit bus was split into 8, 2-bit busses using the array notation, where each bus corresponded to two indices of the 16-bit array.

The select line for the Multiplexor was specified by a second LPM_Counter, called RandomCounter, with a 3-bit output bus width, a count enable, synchronous inputs, and a clear input.

The Audio Synthesizer

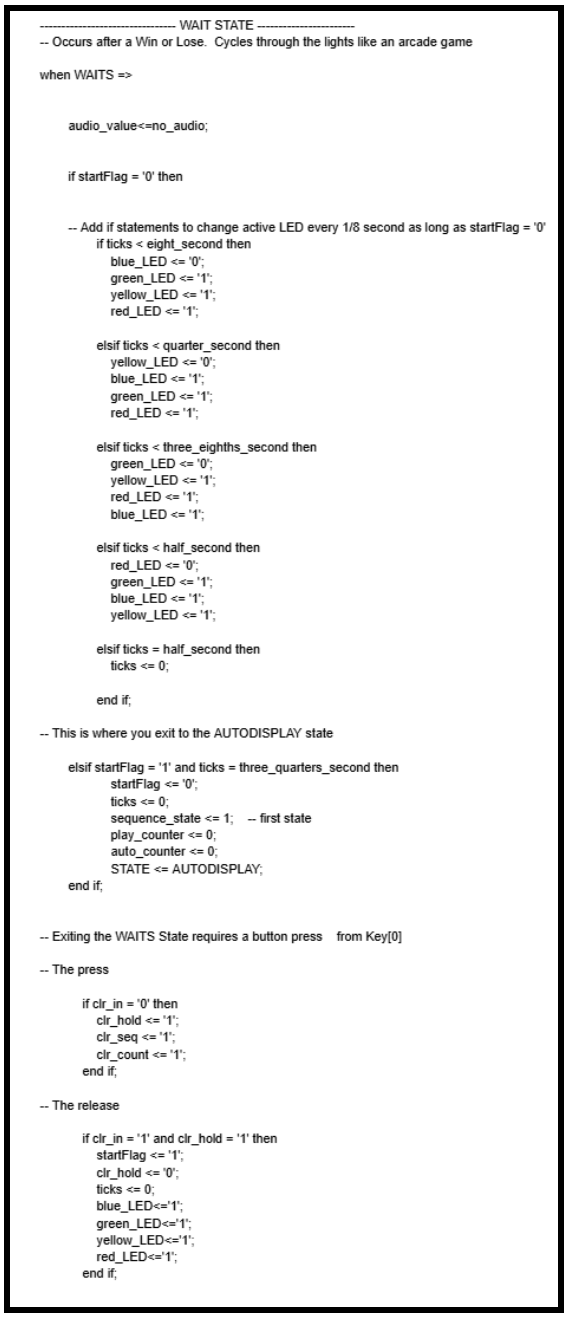
In order to create the audio synthesizer, the team used the provided sin_16bit.mif file and constructed an ROM symbol from the On-Chip Memory archetype, specifying an input with bus width 16 bits, and an initial ROM content as described by the sin_16bit.mif file, and with the In-System Memory Content Editor able to capture and update the contents of the ROM independently of the clock. This instance was named ROM1, and .cmp and .bsf files were generated from it, the .bsf being placed in the schematic. Hereafter, the lpm_dff1.vhd file was used to create an lpm_dff1.vhd symbol. Next, an LPM_ADD_SUB archetype was used to create an ADD_SUB symbol, with two 32-bit inputs and a 32-bit output. This output was connected to the data bus of the D Flip-Flop, whose q output was in turn fed into the data-b input of the ADD_SUB. Then, the ROM and the Flip-flop were connected to the same clock input. The other operand of the ADD_SUB, the data-a bus, was to be equal to the P-Value, specifying the frequency of sampling. As the desired frequency of the sine wave was 200Hz, the P-Value was calculated using the formula 2m*F/Fs, where m = 32 (the number of bits per address), Fs = 48KHz or 48000Hz, and F = 200Hz, thus producing 232*200480000, which is roughly equal to 17895697, which was incorporated as an LPM_Constant. Additionally, the address input to the ROM was specified as the final 8 bits of the feedback loop from the flip-flop to the Add_Sub, called s.

After this, a multiplexer was introduced in the Sine Wave schematic to provide a different sound for each potential user interaction - one for each button color in the case that the user selects the correct button, and a wrong tone in the case that the user selects the wrong button. A sixth input was added which connected to the ground so as to discontinue a sound being played. Because there were six inputs, a 3-bit input was needed to determine between them, which was provided by the q_out line from the System Controller.

The System Controller

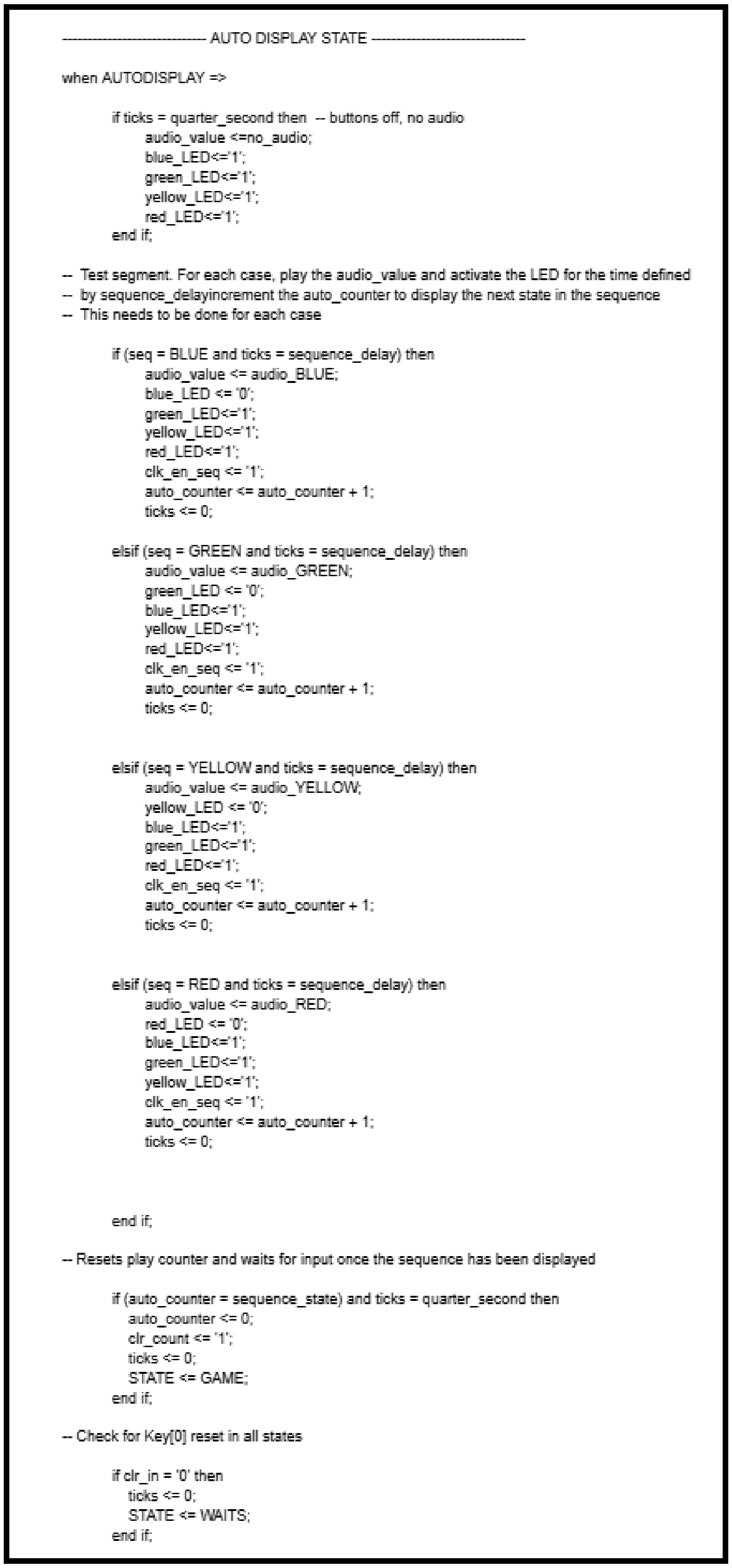
Rather than being constructed in a block-diagram file like the other components, the System Controller was coded using the VHDL language. The core of the System Controller is based on a case statement, analogous to the switch statements used in software languages such as Java and C, which examines the value of the variable STATE, which is initialized to WAITS. In the WAITS state (Figure 2), there are two paths: one where the startFlag is reset, in which a different button is illuminated for a stretch of ⅛ of a second at the intervals of [0, ⅛), [⅛, ¼), [¼, ⅜), and [⅜, ½) for a total period of half a second, based on the measurement of the ticks variable, with all the other buttons left dormant, causing a cyclical movement of light around the buttons. The other path is pursued when the startFlag is set, in which case the value of STATE is set to AUTODISPLAY (Figure 3), where a number of buttons specified by the sequence_state variable, whose identity is determined by the seq variable are displayed at intervals prescribed by the sequence_delay variable. Once the number of buttons which have been displayed (recorded in the auto_counter variable) is equal to sequence_state, the STATE is reassigned to GAME. In the GAME state, the button which the player has pressed is measured by the four inputs from the Button_Debounce symbol, and if a button of the color x has been pressed long enough after the previous press (specified by the lockout_delay variable), its x_LED value (an output line) is set to true, and the audio_value variable (also an output line) is set to audio_X, which cause the corresponding button on the DE2 board to illuminate and its tone to play. The identity of the pressed button is also saved to the variable button, which is compared to the seq variable, which holds the value of the button displayed by the system in the AUTODISPLAY state. If the two match, the play_count is incremented, which records the number of button presses the player has successfully replicated. When the play_count variable is equal to the sequence_state variable, i.e. the player has pressed as many buttons correctly as were displayed by the system, the STATE reverts to the AUTODISPLAY state unless the sequence_state is equal to the win_count, i.e. the system had displayed as many button presses as are requisite to win, in which case STATE is set to WIN. If, on the other hand, the seq and button variables do not match, i.e. the player has pressed a button out of sequence, STATE is set to LOSE. In the LOSE state (Figure 5), the audio_value is set to wrong_audio. In the WIN state (Figure 6), each button is illuminated in sequence, with each of their corresponding tones being played at the moment of illumination. All actions are synchronized based on the value of the ticks value, which increments with each falling edge of the clock (whenever clock = 1, which is when the clock is false, as low true logic was used).

Results



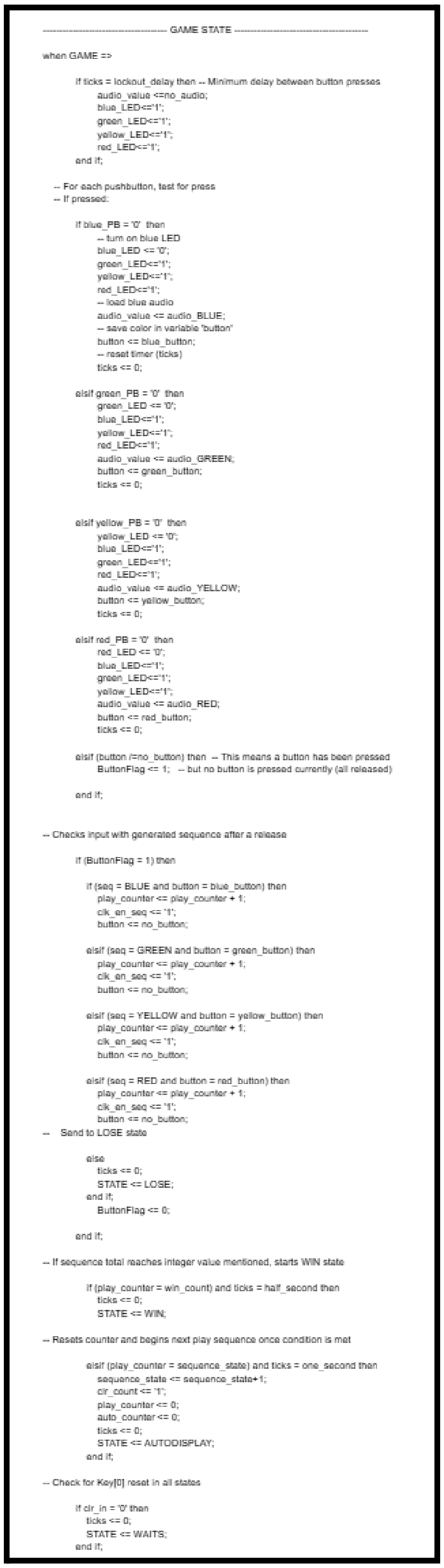
```
------------------------------- WAIT STATE ----------------------
-- Occurs after a Win or Lose.  Cycles through the lights like an arcade game

when WAITS =>


    audio_value<=no_audio;


    if startFlag = '0' then


    -- Add if statements to change active LED every 1/8 second as long as startFlag = '0'
        if ticks < eight_second then
            blue_LED <= '0';
            green_LED <= '1';
            yellow_LED <= '1';
            red_LED <= '1';

        elsif ticks < quarter_second then
            yellow_LED <= '0';
            blue_LED <= '1';
            green_LED <= '1';
            red_LED <= '1';

        elsif ticks < three_eighths_second then
            green_LED <= '0';
            yellow_LED <= '1';
            red_LED <= '1';
            blue_LED <= '1';

        elsif ticks < half_second then
            red_LED <= '0';
            green_LED <= '1';
            blue_LED <= '1';
            yellow_LED <= '1';

        elsif ticks = half_second then
            ticks <= 0;

        end if;

-- This is where you exit to the AUTODISPLAY state

    elsif startFlag = '1' and ticks = three_quarters_second then
            startFlag <= '0';
            ticks <= 0;
            sequence_state <= 1;    -- first state
            play_counter <= 0;
            auto_counter <= 0;
            STATE <= AUTODISPLAY;
    end if;


-- Exiting the WAITS State requires a button press    from Key[0]

-- The press

        if clr_in = '0' then
            clr_hold <= '1';
            clr_seq <= '1';
            clr_count <= '1';
        end if;

-- The release

        if clr_in = '1' and clr_hold = '1' then
            startFlag <= '1';
            clr_hold <= '0';
            ticks <= 0;
            blue_LED<='1';
            green_LED<='1';
            yellow_LED<='1';
            red_LED<='1';
        end if;
```

Figure 2: Wait state VHDL code



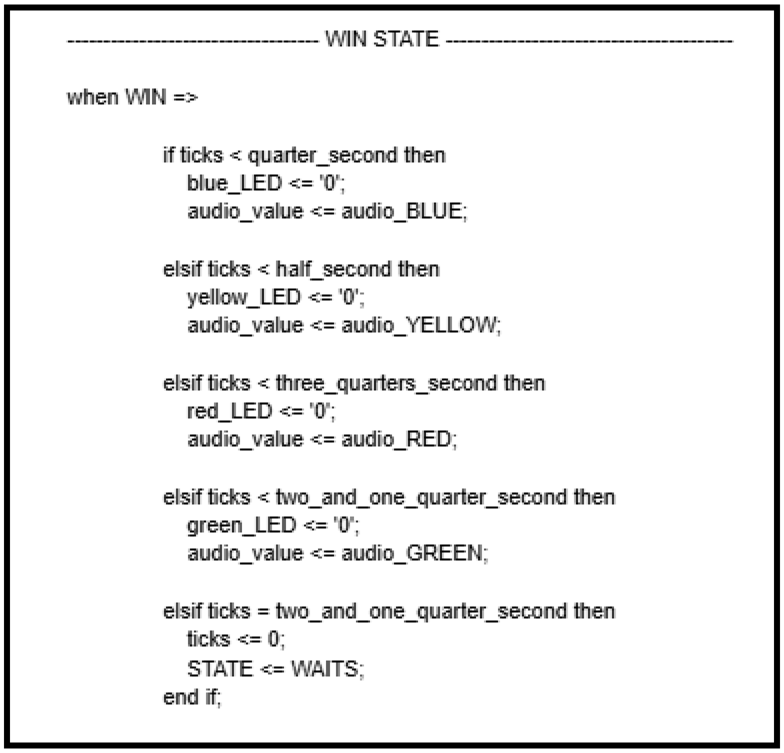
```
------------------------------ AUTO DISPLAY STATE --------------------------------

when AUTODISPLAY =>

          if ticks = quarter_second then  -- buttons off, no audio
                 audio_value <=no_audio;
                 blue_LED<='1';
                 green_LED<='1';
                 yellow_LED<='1';
                 red_LED<='1';
          end if;

--  Test segment. For each case, play the audio_value and activate the LED for the time defined
--  by sequence_delayincrement the auto_counter to display the next state in the sequence
--  This needs to be done for each case

          if (seq = BLUE and ticks = sequence_delay) then
                 audio_value <= audio_BLUE;
                 blue_LED <= '0';
                 green_LED<='1';
                 yellow_LED<='1';
                 red_LED<='1';
                 clk_en_seq <= '1';
                 auto_counter <= auto_counter + 1;
                 ticks <= 0;

          elsif (seq = GREEN and ticks = sequence_delay) then
                 audio_value <= audio_GREEN;
                 green_LED <= '0';
                 blue_LED<='1';
                 yellow_LED<='1';
                 red_LED<='1';
                 clk_en_seq <= '1';
                 auto_counter <= auto_counter + 1;
                 ticks <= 0;


          elsif (seq = YELLOW and ticks = sequence_delay) then
                 audio_value <= audio_YELLOW;
                 yellow_LED <= '0';
                 blue_LED<='1';
                 green_LED<='1';
                 red_LED<='1';
                 clk_en_seq <= '1';
                 auto_counter <= auto_counter + 1;
                 ticks <= 0;


          elsif (seq = RED and ticks = sequence_delay) then
                 audio_value <= audio_RED;
                 red_LED <= '0';
                 blue_LED<='1';
                 green_LED<='1';
                 yellow_LED<='1';
                 clk_en_seq <= '1';
                 auto_counter <= auto_counter + 1;
                 ticks <= 0;



          end if;

-- Resets play counter and waits for input once the sequence has been displayed

          if (auto_counter = sequence_state) and ticks = quarter_second then
             auto_counter <= 0;
             clr_count <= '1';
             ticks <= 0;
             STATE <= GAME;
          end if;

-- Check for Key[0] reset in all states

          if clr_in = '0' then
             ticks <= 0;
             STATE <= WAITS;
          end if;
```

Figure 3: Autodisplay state VHDL code



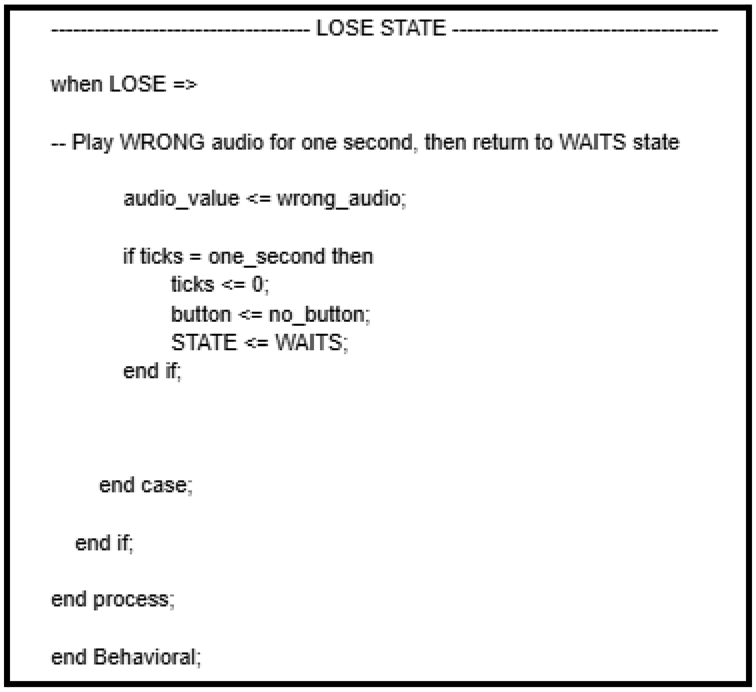
```
-------------------------------------- GAME STATE ----------------------------------------

when GAME =>

          if ticks = lockout_delay then -- Minimum delay between button presses
                audio_value <=no_audio;
                blue_LED<='1';
                green_LED<='1';
                yellow_LED<='1';
                red_LED<='1';
          end if;

   -- For each pushbutton, test for press
   -- If pressed:

          if blue_PB = '0'  then
                -- turn on blue LED
                blue_LED <= '0';
                green_LED<='1';
                yellow_LED<='1';
                red_LED<='1';
                -- load blue audio
                audio_value <= audio_BLUE;
                -- save color in variable 'button'
                button <= blue_button;
                -- reset timer (ticks)
                ticks <= 0;

          elsif green_PB = '0'  then
                green_LED <= '0';
                blue_LED<='1';
                yellow_LED<='1';
                red_LED<='1';
                audio_value <= audio_GREEN;
                button <= green_button;
                ticks <= 0;


          elsif yellow_PB = '0'  then
                yellow_LED <= '0';
                blue_LED<='1';
                green_LED<='1';
                red_LED<='1';
                audio_value <= audio_YELLOW;
                button <= yellow_button;
                ticks <= 0;

          elsif red_PB = '0'  then
                red_LED <= '0';
                blue_LED<='1';
                green_LED<='1';
                yellow_LED<='1';
                audio_value <= audio_RED;
                button <= red_button;
                ticks <= 0;

          elsif (button /=no_button) then  -- This means a button has been pressed
                ButtonFlag <= 1;   -- but no button is pressed currently (all released)

          end if;


-- Checks input with generated sequence after a release

          if (ButtonFlag = 1) then

             if (seq = BLUE and button = blue_button) then
                play_counter <= play_counter + 1;
                clk_en_seq <= '1';
                button <= no_button;

             elsif (seq = GREEN and button = green_button) then
                play_counter <= play_counter + 1;
                clk_en_seq <= '1';
                button <= no_button;

             elsif (seq = YELLOW and button = yellow_button) then
                play_counter <= play_counter + 1;
                clk_en_seq <= '1';
                button <= no_button;

             elsif (seq = RED and button = red_button) then
                play_counter <= play_counter + 1;
                clk_en_seq <= '1';
                button <= no_button;
--    Send to LOSE state

             else
                ticks <= 0;
                STATE <= LOSE;
             end if;
                ButtonFlag <= 0;

          end if;

-- If sequence total reaches integer value mentioned, starts WIN state

             if (play_counter = win_count) and ticks = half_second then
                ticks <= 0;
                STATE <= WIN;

-- Resets counter and begins next play sequence once condition is met

             elsif (play_counter = sequence_state) and ticks = one_second then
                sequence_state <= sequence_state+1;
                clr_count <= '1';
                play_counter <= 0;
                auto_counter <= 0;
                ticks <= 0;
                STATE <= AUTODISPLAY;
             end if;

-- Check for Key[0] reset in all states

          if clr_in = '0' then
             ticks <= 0;
             STATE <= WAITS;
          end if;
```

Figure 4: Game state VHDL code



```
---------------------------------- WIN STATE ---------------------------------------

when WIN =>

            if ticks < quarter_second then
               blue_LED <= '0';
               audio_value <= audio_BLUE;

            elsif ticks < half_second then
               yellow_LED <= '0';
               audio_value <= audio_YELLOW;

            elsif ticks < three_quarters_second then
               red_LED <= '0';
               audio_value <= audio_RED;

            elsif ticks < two_and_one_quarter_second then
               green_LED <= '0';
               audio_value <= audio_GREEN;

            elsif ticks = two_and_one_quarter_second then
               ticks <= 0;
               STATE <= WAITS;
            end if;
```

Figure 5: Win state VHDL code



```
---------------------------------- LOSE STATE ------------------------------------

when LOSE =>

-- Play WRONG audio for one second, then return to WAITS state

         audio_value <= wrong_audio;

         if ticks = one_second then
               ticks <= 0;
               button <= no_button;
               STATE <= WAITS;
         end if;



      end case;

   end if;

end process;

end Behavioral;
```

Figure 6: Lose state VHDL code

The team's results are expressed in Figures 1 to 6 where the CODEC schematic, with the new System_Controller block included, and the games's Wait state, Autodisplay state, Game state, Win state, and Lose State VHDL code.

Discussion

1. Name the 5 states and describe the function of each in one sentence.
    a. State 1: The wait state cycles through the button LEDs.
    b. State 2: The auto-display state shows the sequence of buttons that the player needs to press.
    c. State 3: The game state records the button that the player presses and checks if it is the correct one.
    d. State 4: The win state plays all the sounds at once.
    e. State 5: The lose state plays the wrong.
2. For each state, indicate which hardware blocks in CODEC.bdf are used.
    a. State 1: The wait state uses button debounce.
    b. State 2: The auto-display state uses the button debounce and the random generator and audio block.
    c. State 3: The game state uses the button debounce and the random generator and audio block.
    d. State 4: The win state uses the button debounce and the random generator and audio block.
    e. State 5: The lose state uses the button debounce and the random generator and audio block.
3. How would you change the code to increase the number of game states from 5 to 8?
    a. You must change the win_count variable 5 to 8 to increase the number of game states from 5 to 8.
4. How would you have to change the hardware to increase the number of game states from 8 to 16?
    a. Change the width of the output for counter from 16 bits to 32 bits and change the output of RandomCounter for 3 bit to 4 bit

Conclusion

In summary, this lab furthered the team's familiarity with ModelSim in the Quartus Prime development software, the Altera DE2 Board and the Simon Game Box. The team used these tools to complete the Simon State game consisting of the Simon Game Box, a speaker, and digital circuitry. The game consisted of a light sequence and audio response that was achieved by building on preexisting circuitry. This foundation with signal generation and integration will prove invaluable as the team continues their studies in circuit architecture.

Acknowledgements

Experiment Lead: Kevin O’Neill
Experiment Co-Leads: Ty Fredrick, Stephen Connair
Report Outline: Ty Fredrick
Report Draft: Stephen Connair
Report Revisions: Kevin O’Neill, Ty Fredrick

References

Wahlin, Leah. “Chapter 5. Writing Common Technical Documents.” *Fundamentals of Engineering Technical Communications.* Available: https://ohiostate.pressbooks.pub/feptechcomm/chapter/5-technical-documents/. [Accessed Oct. 1, 2025].

“Lab 01 – Using Quartus Prime and the DE2 Board” Available: https://u.osu.edu/ece2060labs/labs/lab-01/ [Accessed Sept. 23, 2025].

“Lab 02 – Introduction to ModelSim and VHDL” Available: https://u.osu.edu/ece2060labs/labs/lab-02/ [Accessed Oct. 10, 2025].

“Lab 03 –Encoders and Decoders” Available: https://u.osu.edu/ece2060labs/labs/lab-03-encoders-and-decoders/ [Accessed Oct. 20, 2025].

“Lab 04 –Latches and Flip-Flops” Available: https://u.osu.edu/ece2060labs/labs/lab-03/ [Accessed Oct. 28, 2025].

“Lab 05 –Counters” Available: https://u.osu.edu/ece2060labs/labs/lab-04/ [Accessed Nov. 3, 2025].

“Lab 06 – The Audio Synthesizer” Available: https://u.osu.edu/ece2060labs/labs/lab-05/ [Accessed Nov. 7, 2025].

“Lab 07 – The Simon State Machine Part 1” Available: https://u.osu.edu/ece2060labs/labs/lab-06/ [Accessed Nov. 18, 2025].

“Lab 08 – The Simon State Machine Part 2” Available: https://u.osu.edu/ece2060labs/labs/lab-07/ [Accessed Dec. 12, 2025].