Hardware Design and Lab - Final Project Report

”Simon Says Game”

National Tsing Hua University 國立清華大學

Team 37 Members:

1. Gabriela Natalie Hartono 徐青霞 (111062261)
2. Jasmine Davina 林佑銘 (111062361)
3. Charlene Shawn 溫俐妹 (111000268)

Instructor: 李濬屹

Date: 2024年1月8日

Table of Contents

Introduction

Our final project is a recreation—or rather a different take on the popular childhood game, “Simon Says” implemented on the Basys 3 FPGA board and programmed using Vivado with VHDL. The objective of the game is to remember and repeat a random pattern that will be displayed on both the LEDs and the monitor using buttons that the players need to press in sequence. *Figure 1* shows an example of the classic “Simon Says” game that can be found circulating in the market.

Figure 1 (Example of a classic “Simon Says” Game)[1]

In our version of “Simon Says”, the player interacts using the PS/2 keyboard buttons corresponding to four different colors: blue, green, yellow, and red. The pattern of the sequence will be displayed both through the Basys 3 FPGA board LEDs, and a monitor display implemented using VGA. Apart from visual aids, we also implement an audio feature that will play a certain note in correspondence to the buttons that are pressed by the player, as well as 2 different tunes that will play either when the player inputs the correct sequence, or when the player inputs the incorrect sequence and loses. There is no winning or losing in this game—per se since our implementation of the game simply keeps track of the player’s points, which will increment by one every time they input the correct sequence. If the player inputs an incorrect sequence, the game will immediately end, and the player can restart again. A detailed user flow chart can be observed in *Figure 2*.

[1] *Simon Says toy*. Toys R Us Canada. (n.d.). https://www.toysrus.ca/en/Simon-Game/3407A823.html

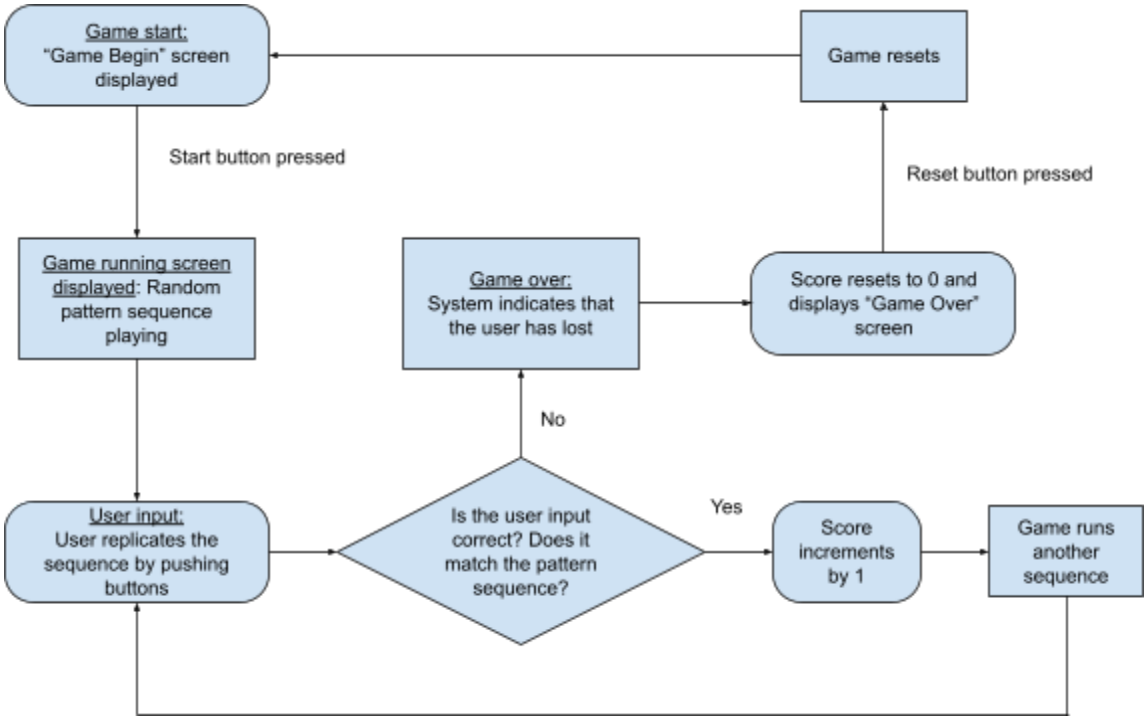

Figure 2 (User flow chart)

Motivations

The conventional handheld version of the "Simon Says" game inspired us to envision our rendition of the game that implements all the concepts we have learned throughout this course through VHDL programming using Vivado with the Basys 3 FPGA board. Our version of the game emphasizes and enhances the player's visual experience; going beyond the traditional blinking colored LEDs and focusing on creative display interfaces using VGA. Through this project, we push our hardware design and programming abilities, while exploring the intricacies of FPGA programming and interface integration.

System Specifications

1. Concept

   There are three main components for this program: the input, the control logic, and the display. It follows that our program will take in the player's button inputs and run the FPGA program logic to calculate its corresponding output. This output will then be forwarded to the VGA connector and finally displayed on the monitor for players to observe. *Figure 3* displays our "Simon Says" game program's design overview.

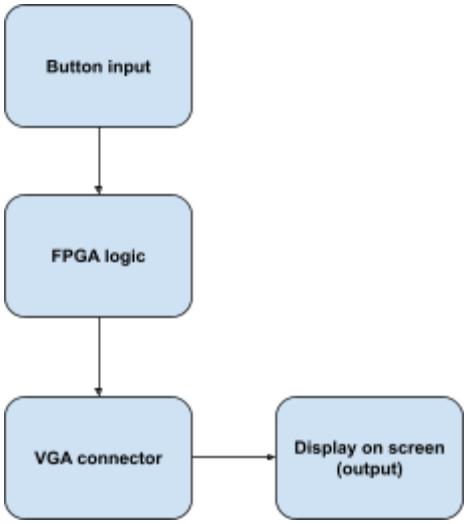

Figure 3 (Game design overview)

To aid in our implementation, we define several inputs and outputs that act as integral building blocks for our program and game design logic. **There are six main button inputs of the program**: four buttons for each color in the “Simon Says” game (blue, green, yellow, and red), as well as an additional 2 buttons for the “reset” and “start” signals. Additionally, the pattern sequence will also be displayed on the FPGA board using the four left-most LEDs. *Figure 4* displays the Basys 3 FPGA layout for our program’s I/O signals.

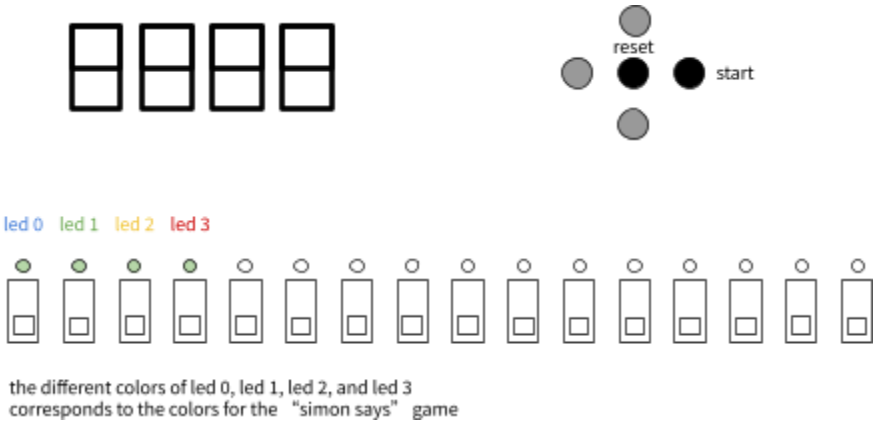

Figure 4 (FPGA board layout)

In addition to using the FPGA board, we are also to use a PS/2 keyboard for our four-button inputs. *Figure 5* shows an example layout of the button inputs using a PS/2 keyboard.

Figure 5 (PS/2 keyboard layout)

The display we used is a standard 640 x 480 pixels generated using a VGA connector. For imaging on the horizontal axis, we turn the video on from 0 to 640 pixels. Likewise, we turn the video on from 0 to 480 pixels for the vertical axis. To make the display, we also utilized the help of a clock divider that reduces the frequency from 100 MHz to 25 MHz, as well as 2 counter modules for both respective axes to count the pixels displayed. Finally, a pixel generator module is used to create our desired design for the display.

2. Implementation

   a. **`clock_divider`** module

      This module is used to take an input clock signal (**`clk`**) and generate an output clock signal (**`new_clk`**) that runs at a frequency divided by a specified factor (**`div_value`**), which is set to 1. In this module, we declare 2 registers used to hold the output clock signal (**`reg new_clk`**) and count clock cycles (**`reg count`**). On each rising edge of the input clock signal (**`always @ (posedge clk)`**), we check whether or not the count has reached div_value (**`if(count == div_value)`**). If so, it resets the count and toggles the output clock signal (**`new_clk <= ~new_clk`**). Otherwise, it will increment the count (**`count <= count + 1`**). This will eventually generate a clock divider that allows us to reduce the frequency of an input clock and generate a slower clock signal for our VGA display.

```
module clock_divider(clk, new_clk);
    input clk;
    output new_clk;
```

```
    parameter div_value = 1;
    //div_value = (100 / 2*5) - 1 = 9

    reg new_clk = 0;
    reg count = 0;

    // increment the counter
    always @(posedge clk) begin
        if (count == div_value) begin
            count <= 0;
            new_clk <= ~new_clk;
        end
        else begin
            count <= count + 1;
        end
    end
endmodule
```

Example of clock_divider module

b. **v_counter** and **h_counter** module

The **v_counter** module is used in tandem with the **h_counter** module to generate a counter that increments and resets itself when it reaches a certain value for our imaging display. While the **v_counter** module controls the imaging display for the vertical axis, the **h_counter** module controls the imaging display for the horizontal axis. Let's break down each counter module in detail.

The **v_counter** module takes in 2 inputs, a clock signal (**clk**) and an enable signal (**v_en**). It then outputs a vertical count signal (**v_count**) with the help of a 10-bit register to hold the count (**reg[9:0] v_count**). The logic behind this counter is fairly simple. Upon the positive edge of the **v_en** signal, the module checks if the **v_count** is less than 524. If not, it will continue incrementing the count by 1 until it reaches 524, which will reset its value back to zero.

The **h_counter** module works synchronously with the **v_counter** module. This module counts clock cycles on the positive edge of the clock signal (**clk**) and generates a trigger signal (**v_en**). It takes in just one input signal, which is the clock signal **clk,** and outputs two signals: a horizontal count signal (**h_count**) and a trigger signal (**v_en**). To hold both signals, two registers are declared, **reg[9:0] h_count** and **reg v_en**. On each positive edge of the clock signal, the module checks if **h_count** is less than 799. If it's less than 799, it will

continuously increment the counter by 1 and set **v_en** to 0. Once it reaches 799, it will set the value of **v_en** to 1 and reset the counter (**h_count**) back to 0. This **v_en** signal will then be outputted into our **v_counter** module as its **v_en** signal. *Figure 6* shows a block diagram of the two modules.

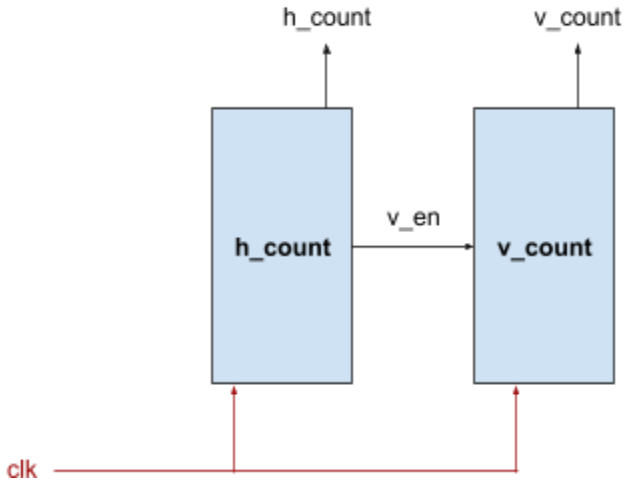

Figure 6 (Block diagram of h_counter and v_counter modules)

c. **vga_sync** module

This module generates synchronization signals and determines the active video area based on the horizontal and vertical counters (**h_count** and **v_count**) from the previous **h_counter** and **v_counter** modules. *Figure 7* shows a block diagram of the **vga_sync** module

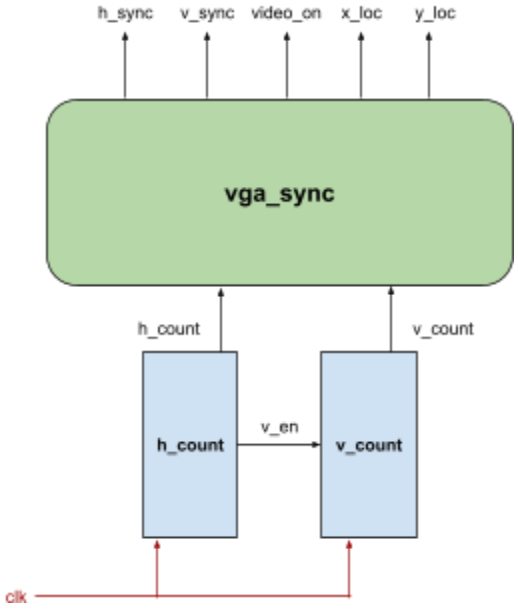

Figure 7 (Block diagram of vga_sync, h_counter, and v_counter modules)

Several parameters are defined in this module, such as **HD** (Horizontal Display), **HF** (Horizontal Front Porch), **HB** (Horizontal Back Porch), **HR** (Horizontal Retrace), **VD** (Vertical Display), **VF** (Vertical Front Porch), **VB** (Vertical Back Porch), and **VR** (Vertical Retrace) all with their respective values.

```
// Horizontal
parameter HD = 640; // Horizontal Display Area
parameter HF = 16; // Horizontal  Right Border
parameter HB = 48; // Horizontal  Left Border
parameter HR = 96; // Horizontal Retrace
// Vertical
parameter VD = 480; // Vertical Display Area
parameter VF = 33; // Vertical Bottom Border
parameter VB =  10; // Vertical Top Border
parameter VR =  2; // Vertical Retrace
```

Parameter initializations

video_enable checks if both h_count is less than HD and v_count is less than **VD**, signaling whether the current pixel is within the displayable region. **x_loc** and **y_loc** assign the current x and y pixel locations based on their corresponding counters. **h_sync** will be set to high during the horizontal display timing's front porch, back porch, and retrace periods. Likewise, **v_sync** will also be set to high during the vertical display timing's front porch, back porch, and retrace periods. Altogether, this module aims to synchronize the display by generating signals (**h_sync** and **v_sync**) and determining the active display area (**video_enable**) based on the input of the horizontal and vertical counters, which represent the pixel locations of our VGA display.

```
//Logic
assign video_enable = h_count < HD && v_count < VD; // if in the
displayable region then 1 else 0
assign x_pos = h_count; // x location same as h_count
assign y_pos = v_count; // y location same as v_count
assign h_sync = h_count < HD + HF | h_count >= HD + HF + HR; // 1
for left to right then 0 reset back to left
assign v_sync = v_count < VD + VF | v_count >= VD + VF + VR; // 1
for up to down then 0 reset back to up
```

Logic assignment

d. **Keyboard Decoder** module *(this module is reused from lab 5 basic)*

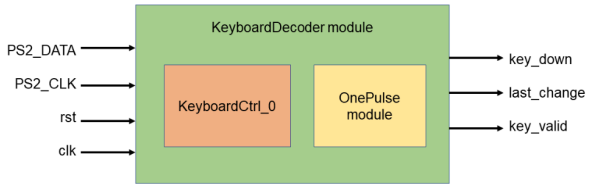

*Figure 8 (*Block Diagram of KeyboardDecoder Module)

This module is used to process the input signals from PS/2 keyboard. It decodes the signal, manages the key states and validity, and provides the signal of the key pressed or released. There are some parameters used, as shown below, to define state values for initializing, extending, and breaking signals.

```
parameter [1:0] INIT            = 2'b00;
parameter [1:0] WAIT_FOR_SIGNAL = 2'b01;
parameter [1:0] GET_SIGNAL_DOWN = 2'b10;
parameter [1:0] WAIT_RELEASE    = 2'b11;

parameter [7:0] IS_INIT         = 8'hAA;
parameter [7:0] IS_EXTEND       = 8'hE0;
parameter [7:0] IS_BREAK        = 8'hF0;
```

Parameter initialization

```
reg [9:0] key, next_key;         // key = {been_extend, been_break,
key_in}
reg [1:0] state, next_state;
reg been_ready, been_extend, been_break;
reg next_been_ready, next_been_extend, next_been_break;

wire [7:0] key_in;
wire is_extend;
wire is_break;
wire valid;
wire err;
wire [511:0] key_decode = 1 << last_change;
assign last_change = {key[9], key[7:0]};
```

Registers and Wires

The registers and wires above are used to store information about current state (**state** and **next state**), key data, extension and break signals, and the validity of the received signals. And last change output is assigned with the key register value of 9th bit concatenated with 7th to 0th bit.

We also need to call the OnePulse module to generate a single-pulse signal and **KeyboardCtrl_0** module from the repository for the PS/2 keyboard controller. Next, there are several always blocks that define the operation in this module. The first always block handles the logic for state transitions and stat-dependent assignments. It will initialize certain variables when resetted. When **rst** is asserted, it sets the **state** to **INIT** and initializes flags (**been_ready**, **been_extend**, **been_break**) and the **key** register. If no reset, it updates the **state**, flags, and the **key** register based on corresponding next-variables, as shown in the snippet below.

```
always @ (posedge clk, posedge rst) begin
        if (rst) begin
            state <= INIT;
            been_ready  <= 1'b0;
            been_extend <= 1'b0;
            been_break  <= 1'b0;
            key <= 10'b0_0_0000_0000;
        end else begin
            state <= next_state;
            been_ready  <= next_been_ready;
            been_extend <= next_been_extend;
            been_break  <= next_been_break;
            key <= next_key;
        end
    end
```

First always-block

The second always block determines the transition operations between states in this module (**INIT**, **WAIT_FOR_SIGNAL**, **GET_SIGNAL_DOWN**, **WAIT_RELEASE**) based on received signals and their validity, as shown below. This kind of logic is applied in the following four always blocks as well.

```
always @ (*) begin
        case (state)
            INIT:            next_state = (key_in == IS_INIT) ? WAIT_FOR_SIGNAL : INIT;
            WAIT_FOR_SIGNAL: next_state = (valid == 1'b0) ? WAIT_FOR_SIGNAL : GET_SIGNAL_DOWN;
            GET_SIGNAL_DOWN: next_state = WAIT_RELEASE;
            WAIT_RELEASE:    next_state = (valid == 1'b1) ? WAIT_RELEASE : WAIT_FOR_SIGNAL;
            default:         next_state = INIT;
        endcase
    end
```

second always-block

For the last always block, it manages the key validity and assigns **key_down** based on if-else condition. If there is a reset signal (**rst**), it resets **key_valid** and **key_down**. If the last change in key is detected and the one-pulse

signal (**pulse_been_ready**) is asserted, it sets **key_valid** and **updates key_down** based on the state of the key register.

```
always @ (posedge clk, posedge rst) begin
        if (rst) begin
            key_valid <= 1'b0;
            key_down <= 511'b0;
        end else if (key_decode[last_change] && pulse_been_ready) begin
            key_valid <= 1'b1;
            if (key[8] == 0) begin
                key_down <= key_down | key_decode;
            end else begin
                key_down <= key_down & (~key_decode);
            end
        end else begin
            key_valid <= 1'b0;
            key_down <= key_down;
        end
    end
```

Last always-block

e. **OnePulse** module *(this module is reused from lab 5 basic)*

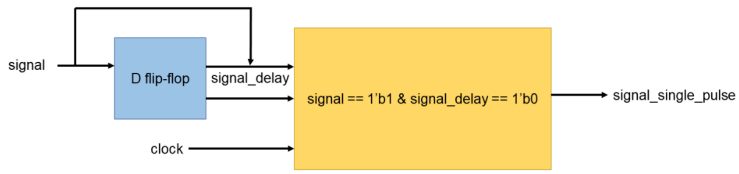

Figure 9 Block Diagram of OnePulse Module

OnePulse module is designed to generate a single-pulse signal of keyboard input. This module has **signal** and **clock** as input that triggers the pulse, **signal_single_pulse** as output to represent the generated single-pulse signal, and **signal_delay** as internal register to store the delayed value of the input signal. The operation on this module is defined within an always block that triggers on the positive edge of the clock signal. The algorithm of this module is defined in an if-else statement. It checks if the current value of the input signal is high (1) and the delayed value of the signal is low (0). If both conditions are fulfilled, it means there is a rising edge in the input signal, so **signal_single_pulse** is set to 1 (high) to generate the single-pulse signal;otherwise, it is set to 0. Then, we update the **signal_delay** register value with the current value of the input signal to compare the current and delayed values in the next clock cycle to detect rising edges.

f. **7-segment** module

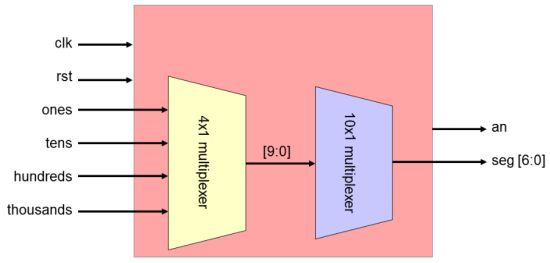

Figure 10 (Block Diagram of Seg7 Module)

This module is used to set the display of game score in the 7-segment of the FPGA. We use several parameters to ease coding processes and registers as shown below.

```
//segment pattern
    parameter zero  = 7'b000_0001;  // 0
    parameter one   = 7'b100_1111;  // 1
    parameter two   = 7'b001_0010;  // 2
    parameter three = 7'b000_0110;  // 3
    parameter four  = 7'b100_1100;  // 4
    parameter five  = 7'b010_0100;  // 5
    parameter six   = 7'b010_0000;  // 6
    parameter seven = 7'b000_1111;  // 7
    parameter eight = 7'b000_0000;  // 8
    parameter nine  = 7'b000_0100;  // 9

    reg [1:0] an_select;     // 2 bit counter for selecting each of 4 digits
    reg [16:0] an_timer;     // counter for digit refresh
```

Parameters and Registers initialization

There are two operations in this module. The first one is controlling the active digit of the 7-segment. For this operation, we always use an always block digit selection based on a timer. The always block is defined as a positive edge triggered by reset or clock. The values of **an_select** and **an_timer** are initialized to 0, same as when reset is asserted. If there is no reset signal, it depends on **an_timer** which is a counter for digit refresh. The refresh period is set to 4 milliseconds (ms) according to calculation (1/100MHz clock = 1/100.000.000 or 10ns(period of 100MHz) then 1/10ns multiplied by 100.000 = 1ms for each digit). When **an_timer** reaches 99.999, it resets to zero and **an_select** is incremented; otherwise, an_timer is incremented. Therefore, each 1ms, there is one digit activated.

```
always @(posedge clk_100MHz or posedge rst) begin
        if(rst) begin
            an_select <= 0;
```

```
            an_timer <= 0;
        end
        else begin
          // 1ms x 4 displays = 4ms refresh period, the period of 100MHz
clock is 10ns (1/100,000,000 seconds), /10ns x 100,000 = 1ms
            if(an_timer == 99_999) begin
                 an_timer <= 0;
                an_select <=  an_select + 1;
            end
            else begin
                an_timer <=  an_timer + 1;
            end
        end
    end
 // connecting an output based on digit select
    always @(an_select) begin
        case(an_select)
            2'b00 : an = 4'b1110;   // ones digit
            2'b01 : an = 4'b1101;   // tens digit
            2'b10 : an = 4'b1011;   // hundreds digit
            2'b11 : an = 4'b0111;   // thousands digit
        endcase
    end
```

Active digit controlling

Based on digit selection, we determine which digit is currently being displayed, either ones, tens, hundreds, or thousands.

```
    // segments based on which digit is selected and the value of each digit
    always @*
        case(an_select)
            //ones
            2'b00 : begin
                case(ones)
                    4'b0000 : seg = zero;
                    4'b0001 : seg = one;
                    4'b0010 : seg = two;
                    4'b0011 : seg = three;
                    4'b0100 : seg = four;
                    4'b0101 : seg = five;
                    4'b0110 : seg = six;
                    4'b0111 : seg = seven;
                    4'b1000 : seg = eight;
                    4'b1001 : seg = nine;
```

```
            endcase
        end
        //tens
        2'b01 : begin
            case(tens)
                4'b0000 : seg = zero;
                4'b0001 : seg = one;
                4'b0010 : seg = two;
                4'b0011 : seg = three;
                4'b0100 : seg = four;
                4'b0101 : seg = five;
                4'b0110 : seg = six;
                4'b0111 : seg = seven;
                4'b1000 : seg = eight;
                4'b1001 : seg = nine;
            endcase
        end
        //hundreds
        2'b10 : begin
            case(hundreds)
                4'b0000 : seg = zero;
                4'b0001 : seg = one;
                4'b0010 : seg = two;
                4'b0011 : seg = three;
                4'b0100 : seg = four;
                4'b0101 : seg = five;
                4'b0110 : seg = six;
                4'b0111 : seg = seven;
                4'b1000 : seg = eight;
                4'b1001 : seg = nine;
            endcase
        end
        //thousands
        2'b11 : begin
            case(thousands)
                4'b0000 : seg = zero;
                4'b0001 : seg = one;
                4'b0010 : seg = two;
                4'b0011 : seg = three;
                4'b0100 : seg = four;
                4'b0101 : seg = five;
                4'b0110 : seg = six;
                4'b0111 : seg = seven;
                4'b1000 : seg = eight;
                4'b1001 : seg = nine;
            endcase
        end
    endcase
```

- 7-segment display controlling

Next, based on the currently selected digit and the value of that digit, we determine the 7-segment pattern. We use nested case statements to handle each digit individually and assign the appropriate 7-segment pattern, as shown above.

g. **Simon** module

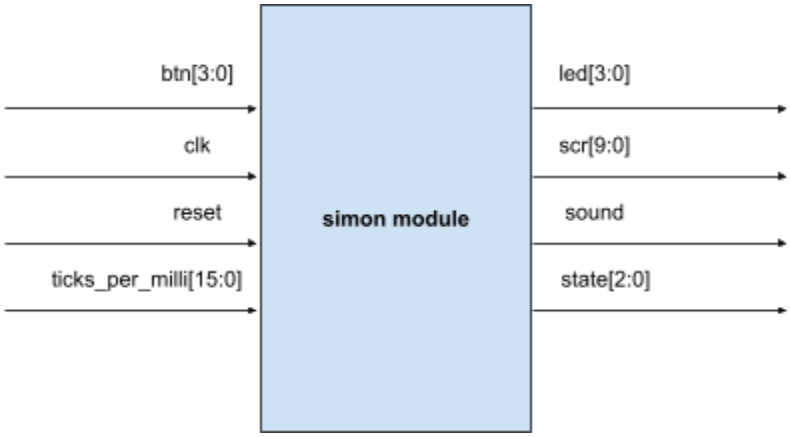

Figure 11 (Block Diagram of Simon Module)

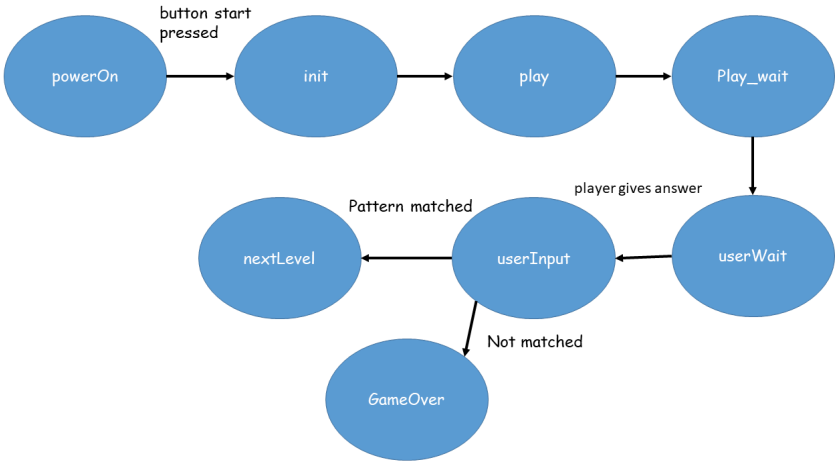

Figure 12 (State diagram of simon.v module's game logic)

Figure 12 shows the state transition logic behind our simon.v module in which we control most of the game logic. When the user runs the program, it will go to its default state (**`powerOn`**), in which we will display our "Game Start" scene and wait for the start button (**`btn_start`**) to be pressed. Once the start button is pressed, we transition to the init state where we initialize game parameters and sequences. After 500 ms, it will automatically transition to the **`Play`** state. In this state, we begin displaying our pattern sequence both through LEDs and VGA displays while simultaneously playing their corresponding sound frequencies and

transitions to the **PlayWait** state. In this state, we ought to reset all the values of the LEDs and sounds after 300 ms and check if the entire pattern has finished displaying. If yes, it will transition to **UserWait**; otherwise, it should increment the sequence counter and transition back to **Play**.

The **UserWait** state waits for the player's button inputs after the sequence is played and checks for valid key inputs which will go to **UserComp** if a valid key is pressed. Moving on, the **UserComp** state lights up the LEDs and sets the sound frequency based on the user input. In this state, we should also check for the correctness of the pattern, in which it will transition to one of these two states; **NextLevel** or **GameOver**. If there is a mismatch found in the pattern, it will transition to the **GameOver** state which resets the scores and waits for the reset button to be pressed to restart the game. Otherwise, if the player is successful, it will transition to the **NextLevel** state which will increment their score and transition to **Play** for the next round. We get the random sequence from the clock counter. Every cycle, the random variable will be added by 1. The final value will determine which button is selected. So according to when the user pressed the button and how long does the user need to finish pressing the sequence given, this module will generate the random sequence.

h. **Play_sound** module

```
else begin
      clk_cyc <= clk_cyc + frequency;
      if (clk_cyc >= (clk_per_second >> 1)) begin
        sound <= !sound;
        clk_cyc <= clk_cyc + frequency - (clk_per_second >> 1);
      end
end
```

This module is used to create the wave in a certain frequency to produce sounds that we hear. It will toggle a variable sound which is connected to the **pmod**. This variable will toggle when **clk_cyc**, which is incremented by the value of frequency every clock cycle, reaches or exceeds half of the clock cycles in one second. *Figure 13* shows the block diagram of this module.

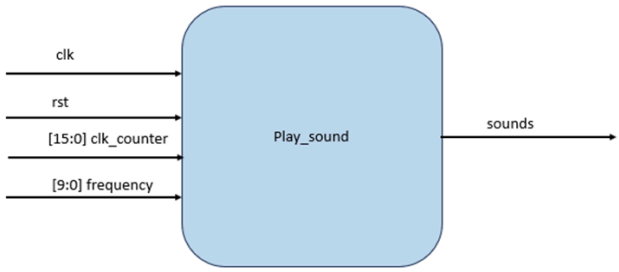

Figure 13 (Block diagram of Play_sound module)

i. **Digit** module

```
assign thousand = num/1000;
assign hundreds = (num%1000)/100;
assign tens = ((num%1000)%100)/10;
assign ones = (((num%1000)%100)%10);
```

The digit module is to help us output the score in the seven segment. The most left seven segments will be assigned thousands and the most right will represent ones.

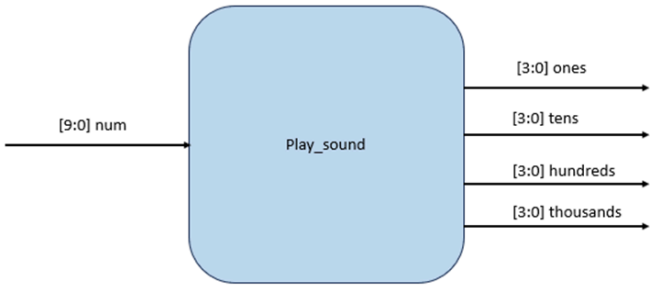

Figure 14 (Block diagram of Digit module0

The above picture shows the block diagram of the **digit** module.

j. **Pixel_gen** module

Since we are just going to display a simple picture, we use this module to adjust the color on each pixel coordinate. This module takes **clk**, current coordinates which are **pixel_y** and **pixel_y**, **LED0** to **LED3**, and **state** as an input. This module will output the RGB color. The block diagram can be seen in *Figure 15*.

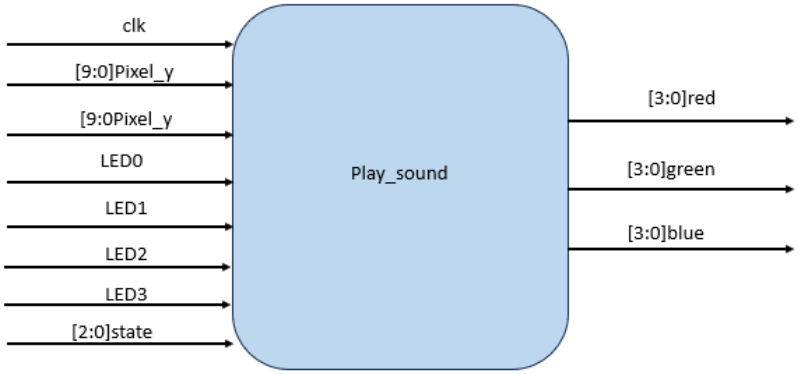

Figure 15 (Block diagram of Pixel_gen module)

We use state to determine which screen will be used. Each screen, which is begin, game over, and game working screen will have a series of if and else if to make the picture or letter we want. First we get the current coordinate of x and y. Then assign the condition inside an if and else if statement.

```
else if (
                // G
( pixel_x>=119 && pixel_x<=210 && pixel_y>=130 && pixel_y<=150 )||
( pixel_x>=119 && pixel_x<=139 && pixel_y>=130 && pixel_y<=230 )||
( pixel_x>=119 && pixel_x<=210 && pixel_y>=210 && pixel_y<=230 )||
( pixel_x>=190 && pixel_x<=210 && pixel_y>=160 && pixel_y<=230 )||
( pixel_x>=169 && pixel_x<=210 && pixel_y>=160 && pixel_y<=180 ))
                begin
                    red <= 4'hF;
                    blue <= 4'h0;
                    green <= 4'h0;
                end
```

Making letter G

The code snippet above is to make the letter G. To make a word, we make a lot of if and else if like the above. The color is arranged using 4 bit RGB color. In the game working screen, when a button is pressed then the box will look like it lights up. It is also being processed here. This module receives an input named **LED1** to **LED4** to give the signal which button is being pressed.

```
      case (LED1)
                          1'b0: begin
                              red <= 4'h0;
```

```
                    green <= 4'h6;
                    blue <= 4'h0;
                end
                1'b1: begin
                    red <= 4'h0;
                    green <= 4'hF;
                    blue <= 4'h0;
                end
            endcase
```

If button1 is being pressed then the signal **LED1** will be 1 therefore generating a color brighter than when it is 0. The same goes for all the buttons.

k. Top module

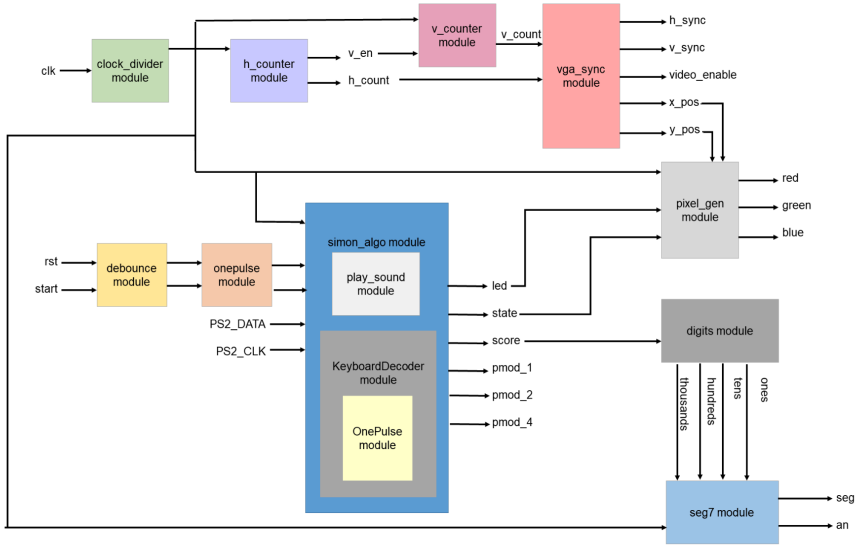

Figure 16 Block Diagram of Top Module

Top module is the top-level module in this game design. This module has several input and output ports for various signal, including clock (**clk**), reset (**reset**), start (**start**), PS/2 keyboard data (**PS2_DATA**) and clock (**PS2_CLK**), LED outputs (**LED0** to **LED3**), synchronization signal for VGA (**h_syn** and **v_sync**), RGB pixel signals (**red**, **green**, **blue**), 7-segment display (**seg** and **and**), and speaker signals (**pmod_1**, **pmod_2**, **pmod4**).

Beside that, there are several wires to connect each module called in this top-level module, as shown in Figure 16. In this module, we instantiate various modules, namely:

★ **clock_divider** module: to divide the input clock signal (**clk**) to generate a new clock signal (**new_clk**)

- ★ **debounce** and **onepulse** module: to eliminate noise and ensure stable button signals from reset and start from input button *(these modules are reused from the examples in the lecture notes)*

- ★ **simon _algo** module: to implement the simon says game algorithm, converting input from keyboard, controlling LED display, controlling speaker sound output, and counting the score of the game.

- ★ **h_counter** and **v_counter** module: to instantiate horizontal (**h_counter**) and vertical (**v_counter**) counter modules which are used for generating synchronization signals for VGA output.

- ★ **vga_syn** module: to generate synchronization signals for VGA output based on the horizontal and vertical counters.

- ★ **pixel_gen** module: to generate pixel colors based on the position coordinates (**x_pos** and **y_pos**) and the game state.

- ★ **digits** and **seg7_control** module: to handle score digits and control the 7-segment display based on the modified clock signal (**new_clk**), reset signal, and score of the game.

Experimental Results

To test our program, we did several runs for different versions of the programs; all with varying outcomes. **To aid our documentation, we have provided videos and images of the results, which can be accessed through the Google Drive links below.** *Table 1* provides a list of all the runs we did, as well as their corresponding results which we have attached accordingly. We also listed each run's outcome explanations and our thoughts on any alterations/ improvements that should be made to our program after each try.

No.	Date	Result (video/ image)	Outcome Explanation & Thoughts
1.	Jan 5, 2024	First Beta Testing of the Program	Outcome explanation: - There are still incorrect state transitions from the state in which the program should take in the player's inputs and the state in which the program will compare the inputs to the given sequence. - The outcome results in an unexpected output in which the LED in SW14 constantly lights up after every output - We concluded that this might be

			due to state transition mistakes, and incorrect debouncing of the button inputs Thoughts: - We speculated that we should try using the PS/2 keyboard buttons and altering the debounce module - Review and modify the state transition logic within the program
2.	Jan 7, 2024	Second Beta Testing of the Program	Outcome explanation: - We tried altering the state transition diagrams and used a different debounce module for our button inputs - Although the state transition is now correct, we encountered an issue where it's not outputting an entirely random sequence, instead constantly repeating the same sequence over and over Thoughts: - We speculated that the issue might lie in the random algorithm that is not correctly outputting a random sequence, and revision should be done for that module - We should try testing the correctness of the audio module and VGA display to catch any early errors - Instead of using the FPGA buttons, it might be better to integrate the PS/2 keyboard and use keyboard button inputs instead
3.	Jan 8, 2024	First Alpha Testing (without VGA, and audio Pmod)	Outcome explanation: - We added and tested the points feature shown in the seven-segment display. The outcome is successful as it increments with every successful pattern input - Revisited and modified the random algorithm and made corrections. The run shows that it correctly reads the user's input

			and displays a random pattern - Tested button inputs using a PS/2 keyboard and implementing the debounce module from Lab 5 (not shown in the video) Thoughts: - More testing should be done for the program, especially for the audio and VGA portion
4.	Jan 8, 2024	Second Alpha Testing (with VGA, audio Pmod, and keyboard) Attachment: Image of "Game Over" screen	Outcome explanation: - Tested the validity and correctness of the program with the VGA display logic (such as the blinking of the buttons) and audio Pmod, which should play a note corresponding to the inputs, and play 2 tunes which are shown to be successful - The "Game Over" display screen is still incorrect as some pixels overlap with one another Thoughts: - Revisit and fix the VGA display for the "Game Over" screen - Enhance and make improvements to the display to make the game more interesting
5.	Jan 9, 2024	Final Testing of Finished Product	Outcome explanation: - Mistakes in the "Game Over" screen have been fixed, as well as minor modifications made to the overall display such as pixel placements and redesigns. - No significant bugs or errors were found in the several trials that we did Thoughts: - If possible, more modifications to the display could be made to make it look more interesting

Table 1 (Experiment results log)

Conclusion

In conclusion, making this final project has been a challenging yet rewarding journey. Our project involved the integration of VHDL programming, hardware design, and interface implementation to create a unique and interesting twist on the classic Simon Says game. Throughout this game, we inevitably came across various obstacles and learning opportunities that only enriched our understanding of FPGA programming and system design.

We especially encountered a lot of issues in the beginning testing phases, with many bugs and errors and incorrect state transitions. Each testing phase we did played a crucial role in identifying and addressing issues we found in our program. As we progressed to the final testing of the program with the VGA, audio, PMOD, and keyboard integration, we also encountered several issues with our “Game Over” screen, requiring further refinement. In the end, we were able to polish our program just in time for our demo.

In reflection, we feel that this project has not only pushed our technical skills in FPGA programming and hardware design but also emphasized the importance of systematic testing and debugging in the development cycle, especially with a big project such as making a game. Looking forward, there is always room for improvement and refinement to our program, nonetheless, this project has been a valuable experience, providing us practical insights into hardware design and collaborative problem-solving.