ENGS 31 Summer 2016 Final Report

John Sullivan and Jennifer Jain

Simon Game

Abstract

The goal of our project was to build the Simon Game. The Simon Game challenges users to remember a randomly generated sequence of signals. A sequence of signals is displayed to the user using lights and sounds. The player then attempts to enter the sequence in the proper order. If the sequence is entered properly, the game advances and adds an additional signal to the sequence. The game is lost if an incorrect button is entered. We were able to build a working Simon Game that accomplishes these tasks. In addition, our Simon Game device keeps track of the current score and high score.

Table of Contents

1. Introduction

The problem that we solved was building a working Simon Game. To accomplish this task, we worked in steps and addressed several smaller challenges. It was necessary to create a form of memory to remember randomly generated sequences of values for the user to enter. With this memory, it had to be possible to both display all of the values in the sequence to the user and then compare those values to the ones entered. In order to display the sequence to the user, lights and sounds were incorporated into the design. A challenge with the lights and sounds was making them work at a speed that the player would be able to understand. In addition, the current and high score would have to be compared to determine if the high score should be updated. We addressed these challenges systematically and built a working Simon Game.

2. Design solution

2.1. Specifications

Our circuit displays sequences of signals to the user using lights and sounds. From this sequence, the circuit compares the user's entries to the entries stored in memory to determine if the user selected the sequences in the proper order. Our circuit has a start button input that is used to signal the game to begin and a clear button input that is used to reset the high score to zero. There are four button inputs that are used during the game so that the user may attempt to enter the correct sequence that was displayed. To display the sequence to the user there are four light outputs that designate which LED on the breadboard should be turned on, and a sound output which sends an audio signal to the speaker. These outputs are also displayed when the

user presses a button. For the audio, two additional outputs were necessary from our device to keep the audio output turned on and to adjust the decibel gain for sound that was sent to the speaker. Finally, there are outputs to the seven segment display that function to display the current and high scores. These update as the high and current scores change. For an illustration of these inputs and outputs, see the Appendix B, Figure 1.

2.2. Operating instructions

To prepare the Simon Game for use, the FPGA must be plugged into the USB port on the computer and all the Pmods should be connected to the FPGA as follows:

- Connect PmodBB to JC pins
- Connect PmodBTN to JXADC pins 1 through 7
- Connect PmodAMP2 to JA pins 1 through 7

The code must be taken through bitstream generation on Vivado and to start the game, the center button on the FPGA should be pressed to start the game. Each of the four buttons on the PmodBTN correspond to the corresponding LED's on the PmodBB. To clear the high score, the rightmost switch on the FPGA can be turned on and then off before starting the game again.

2.3. Theory of operation

Our circuit consists of multiple components, such as memory, counters and number generators. When the game is started, a random number is generated. This random number is loaded into the RAM at the current write address. Once that random number has been added, the write address is incremented so that the device is ready to add another value to the sequence, should the user get the next value correct. The contents of the RAM are displayed to the user using lights and sound, using the cycle counter to increment through all of the read addresses in

the RAM. Once this process is complete, the user is now able to enter what they believe to be the correct buttons in the correct order. The user's button presses are compared to those coming from the read address in the RAM.

If the user enters the correct button, a *correct* signal will be set, which will be seen by the controller in the check state. The device will continue and add another value to the sequence that the user is trying to remember. This process of display of the sequence and user entry will continue until the user presses the wrong button at some point while entering one of the sequences. If the user selects the wrong button, a losing sequence is displayed with the lights and sounds. Throughout the game, if the user ever starts to achieve a score that is greater than the high score, the high score will be changed to this new high score. The user has the option to reset the high score should they wish.

See Appendix B, Top Level Datapath for an illustration of the interaction between blocks in our datapath and with the controller. See Appendix C, Top Level State Machine for an illustrative view of how a user progresses through different states of our game. See Appendix B, Figure 3 to 6 to see annotated simulation diagrams that detail what is happening with the different signals in the circuitry when the game is played. The numbers for each of our entries below correspond to the number labels on the diagrams in Figures 3 to 6.

1. Controller

The controller has inputs and outputs. For an explanation of these inputs and outputs, see Section 2.1 Specifications. The controller also receives some signals from the datapath. Based on the inputs to the controller and some of the signals from the datapath, the controller functions as a state machine. The state diagram for this state machine is shown in Appendix C Top Level

State Machine. The inputs to the controller and inputs from the datapath will determine how the controller transitions between states. When the controller is in a certain state, it will output specific signals, such as signals for the LEDs to turn on or enable signals for the datapath. These specific outputs are indicated beneath the states in the state diagram in Appendix C. Through this interaction between the controller, user, and datapath, it is possible to play the Simon Game. In testing, such a setup makes it possible to determine the given state of the device at any time.

2. Dual Port RAM

We chose to use a dual port RAM, since we would need to be able to read and write from it and we wanted to deal with reading and writing separately. We used address A (*addra*) and input A (*dina*) from the RAM to handle writing data to the memory. The write address is chosen by the current score process. Writing to the RAM is always enabled (*ena*), since the write address is incremented just after data is loaded and the same address will not be returned to write to in the same play of the game. This current score is incremented every time a new random number is generated, such that for the next round it points at the correct address in memory to be written to.

We used address B (*addrb*) and output B (*doutb* or *rd_data*) from the RAM to handle reading data from the memory. The address to read from is controlled by one of two counters, the cycle counter and press counter, which are chosen between by a multiplexer. If the cycle counter is selected, the device increments through the addresses of the RAM and displays the data stored to the user. If the press counter is selected, the device increments through the same addresses each time a button is pressed, in order to compare the user press to the memory

contents. The data that is read from the RAM is compared with the button that the user enters to determine if it is correct, or the data is displayed to the user with lights and sounds.

3. Current score and address to score

When the Simon game is started for the first time, a random number is generated and then loaded to the RAM. It is loaded to the address specified by the current score process (*curr_addr*), which will point at the first address for the first run of the game. Once a value has been loaded to the RAM, the current score is immediately incremented (using *w_addr_incr*) so that it points at the proper address when a new random number has to be added for a new round. Should the user get a sequence correct, this process of loading and then incrementing the address is repeated. It is possible to derive the current score (*curr_score*) from the current address (*curr_addr*). The current score is set to zero for the first run of the game. After this first round, it is possible to find the current score by subtracting one from the current address, since the current address is incremented before the user can get the sequence correct.

4. Random number generator

The random number generator consists of a 2-bit counter (*rand_num*) that is constantly counting from 0 to 3 (4 numbers) and then back to 0. In this block, there is a register (*rand_num_out*) that when enabled (by *rand_en*) will load a random number from the counter. This random number is passed to the RAM (at *dina*). A new random number is generated and added to the RAM every time the user starts a new game or enters the proper sequence of buttons corresponding to the light display.

5. Cycle counter

The cycle counter is a counter that is clocked. It is only enabled to count if it is set to be enabled (*cycle_counter_en*) and it receives a slow tick (*slow_tick*). It counts until it reaches the address being pointed to write (*curr_addr*) and then outputs a terminal count to the controller (*tc_w_addr*). The reason for the slow tick is that it should not function directly on the fast clock that is controlling the rest of the device. If it did, the user would not be able to distinguish the different lights in the sequence because they would appear so quickly. Further, the lights and sounds should not be displayed directly one after the other, but rather there should be a delay in between. To solve this issue a signal, *alt*, is used, which alternates on each slow tick. This signal is used to determine whether to output lights and sound from the overall device.

The first output of the cycle counter is the cycle address (*cycle_addr*), which selects the address to read data from the RAM to output to the lights and sound. The second output is an enable signal for the lights and sound (*ls_cc*) which takes advantage of the *alt* signal, so that the lights and sound are only enabled every other clock cycle and the user can distinguish between individual lights and sounds. Finally, the cycle count is cleared (*clr_count*) every time a user gets the proper sequence (*Reset_Counts* in Appendix C Top Level State Machine) or when the user loses (*Reset_All* in Appendix C Top Level State Machine).

6. Press counter

The press counter is similar to the cycle counter. However, the press counter only counts whenever the press counter is enabled (*press_counter*_en) and the user presses a button (*btn_press*). It continues to increment every time that a button is pressed until it reaches the address being pointed to write (*curr_addr*), at which point it outputs a terminal count (*pc_tc*). It

outputs an address that will be used to read through the contents of the RAM. This makes sense, since every time a button is pressed, one would want to compare and then get a new address to read from the RAM for the next check. The press counter is cleared every time a user gets the proper sequence (*Reset_Counts* in Appendix C Top Level State Machine) or when the user loses (*Reset_All* in Appendix C Top Level State Machine).

7. Mux1

This multiplexer chooses which counter should be used to read data (*rd_data*) from the RAM. It will either use the press counter (*press_count*) or the cycle counter (*cycle_count*) depending on the current state. A signal (*rd_tog*) from the controller is used to choose between these two counters. This signal changes values based on whether the device is cycling through the sequence to show the user (*Cycle* state in Appendix C Top Level State Machine) or comparing the sequence to the user's button presses (*User and Check* states in Appendix C Top Level State Machine).

8. Button encode

This is a simple encoder. It takes the monopulsed button press (*btn_m*) which is 4-bits, and if only one of the 4-bits is set high, then it encodes it to two bits (for values 0-3), indicating which bit was high. If only one of the bits was high, then the register containing this button press (*curr_btn*) is updated with the new button press. Otherwise, this register is set to contain the same value that it did previously.

9. Button compare (*sc_comp*)

This compares the output from the RAM (*rd_data*) to the encoded button press (*curr_btn*). It is asynchronous, so the comparison is always happening, but its result will only be used when the device wants to check if the user entered the correct button (*Check* state in Appendix C Top Level State Machine). If it is correct or not in this state determines whether or not the game will continue and keep adding more values to the sequence, or if it will begin the procedure for the game being lost.

10. Lose sequence

We wanted a special way to indicate to the user if the incorrect button was entered and the game was lost. We decided to use a counter that would loop through all the buttons and associated sounds twice. Like the cycle counter, the lose counter only increments if it is enabled (*lose_en*) and it has received a slow tick (*slow_tick*). Also, like the cycle counter, it alternates between showing lights and displaying buttons and sounds (using signal, *alt2*). Internally, the counter outputs a terminal count once it has incremented through all of the lights and sounds twice. At this point, it sends a signal to the controller (*lose_done*), which indicates for it to move out of this state and on to the state to reset all the counts (*Reset_All* state in Appendix C Top Level State Machine).

11. Rester

At certain points in our design, it was necessary to have a process that implemented a delay using the slow tick (*slow_tick*). This occurs after a random number has been generated (*Gen* state in Appendix C Top Level State Machine) or after the user has released an incorrect button (*Lights_Off2* state) and before the losing sequence is displayed (*Lose* state). One slow tick

period will be elapse before the next state is entered (See Appendix K Figure 1.1 #6 and Figure 4.2 #1 to see this effect in action on the simulation diagrams).

12. Mux2

This multiplexer chooses between data for the lights and sound. Although data is sent from this multiplexer back to the controller to be used for lights and sounds, it does not necessarily mean that the lights and sounds will be displayed (Mux3 handles the lights and sounds enabling). The Mux2 output only contains which light or sound should be on if they are enabled. Mux2 receives output from the RAM (*rd_data*), from the current button being pressed (*curr_btn*), and from the losing sequence counter (*lose_count*). To choose between these inputs to this multiplexer, Mux2 has select bits (*ls_sel*), which it receives from the controller to determine which input signal to output back to the controller for the lights and sounds. These select bits changes depending on the current state (see *ls_sel* output in Appendix C Top Level State Machine).

13. Mux3

This multiplexer chooses between different processes for how the lights and sounds will be enabled. The output from this multiplexer determines whether or not the lights and sounds are enabled (see distinction explained in Mux2). Mux3 receives bits from the cycle counter (*ls_cc*), from a button press comparison (*btn_lt AND NOT(btn_off)*, which means to show lights for a pressed button), or from the losing sequence counter (*ls_lose*). Not that the signal, *btn_lt*, designates if the device is in a state in which lights and sound should be displayed if a button is pressed. Mux3 chooses between these bits using the same select bit signals from the controller that Mux2 uses to choose between its data (*ls_sel*). These select bits from the controller changes

depending on the current state of the diagram (see *ls_sel* output in Appendix C Top Level State Machine).

14. Lightbox

The lightbox is a multiplexer with an enable bit (using signal *ls_mux_en* from the controller). It is a 4 x 2 multiplexer, so it has 4 inputs and 2 outputs. Depending on the select bit (using signal *ls_mux* from the controller), the multiplexer chooses between four preset inputs, indicating to turn one of the four LED lights on. We designated the inputs to our multiplexer as high-true, so the output from it must be inverted before it reaches the LEDs.

15. Soundbox

We created a module to generate sound signals when given a specific frequency. This module will create a square wave with the frequency that it has been designated. We instantiated this module four different times to create sounds for the four different colored lights present in the game (*green, red, blue,* and *yellow*). These sound signals are output from the controller and into a MUX very similar to that used for the lightbox. The only difference between the two is that this soundbox multiplexer chooses between 1-bit sound signals as inputs rather than 4-bit light controls. It has both the same enable bit (*ls_mux_en*) and select bit (*ls_mux*) as the lightbox. For this reason, it will output sound at the same exact time that a light is shown.

16. High score

The high score (*high_score*) starts at 0 when the game is played the first time. The high score is updated only if the current score (*curr_score*) exceeds the high score. There is also an option by which the high score can be reset to 0 if the clear (*clr*) button is pressed while waiting

for start to be pressed (this is the *Idle* state, see *Reset_HS* state in Appendix C Top Level State Machine).

17. Score processor

We encountered an issue with how our current and high score were stored in the device. We stored these scores as 8-bit unsigned numbers (*curr_score* and *high_score*). This method causes an issue when exporting scores to the 7-segment display because the multiplexer for the seven segments takes in 4-bit inputs, one for each digit. To resolve the issue, we created a score processor by which the 8-bit unsigned number may be converted to two 4-bit standard logic vectors (this is the same as binary coded decimal).

The score processor checks whether or not the score is greater than 9 (that is, whether or not it is two digits). If the score is greater than 9, the score processor calculates the first digit by dividing by 10 and truncating (this is done by default). To get the second digit, the score processor takes the modulus 10 of the number to find the remainder when the number was divided by 10. If the score is not greater than 9, the score will not be processed. This method will give a two bit representation of the score that may be used by the 7-segment display to show the desired digit. We did not worry about the binary coded decimal (BCD) representation not containing enough digits to hold the unsigned score value since the highest score we have ever seen a user get is 15 and it will not be higher than 99, the max value for BCD.

18. Mux 7-segment display

The multiplexer for the seven segment display takes in 4-bit inputs for each of the 4 digits on the seven segment display. It also takes in 4-bits indicating if and where a decimal point should be placed. We had to convert the current score (*curr_score*) and high score (*high_score*)

to 8-bits so that they would comply with this module (see Score processor description). We put the high score on the first two digits of the display and the current score on the second two digits. We put a decimal point in between the current and high scores to separate them.

2.4. Construction and debugging

In constructing our design, we spent a large portion of time working on our initial state machine diagram and datapath. Even though we still ended up changing these two designs quite a bit, they provided a good foundation for us to start working on our device. From this point, we wrote code for the controller based on our state diagram. We adjusted our state diagram as we realized that more or less states were necessary or more outputs from the state machine needed to be communicated with the datapath. We spent the majority of our time working on this code for the datapath because it contains all of the inner workings of our device. The bulk of the code for our datapath is contained in the datapath VHDL file. Working with this datapath VHDL file, we were able to test specific components by sending them signals in the same way that the controller would send signals to the datapath. We produced simulations of the components on the computer and fixed any visible issues (see Appendix K for functioning simulation graphs).

We started by building the most complicated components first. These initial components included the RAM that is at the heart of our design, the counters that control the addresses for writing to and reading from the memory, and the comparators that would dictate how a player progresses through the Simon Game. Throughout this part of the project, we encountered issues with constructing the RAM. We initially tried using a single ported RAM. In the end we chose to use a dual ported RAM so that we could easily distinguish between reading from the memory and writing to the memory. With these components in place, we were able to focus on some of

the more tangential parts of our project, such as tracking the high score and outputting lights and sound.

Once we were able to test the components of our design by simulating them on the computer, we were ready to program the FPGA board using our VHDL code. Our design appeared to work fairly well at first, until we realized that it was only possible to play a few rounds of the game before losing. We ported the current button that the user was pressing and the value from the RAM that it was being compared against to ports on the FPGA board to view with the oscilloscope. With the outputs displayed on the oscilloscope, we quickly realized that our issue was that one of the buttons was not changing. We returned to our code and realized that the current button that the user pressed needed to be stored in a register, which is clocked. With this modification, our design worked on the FPGA board.

The last issue that we encountered was with finding a viable method to get the sound working. We considered a few possible methods of doing this, such as making a lookup table for a sine wave. The sine wave lookup table turned out to be more than was necessary for us to produce the sounds we desired. We only needed simple sounds and we were able to make these with a new module called soundbox. The soundbox has a generic constant input representative of the frequency at which we wanted the sound to be produced. With this generic constant input set, we were able to create a module that would output a square wave at a desired sound frequency. We were happy with this method in the end because it produces a very mechanical sound for each button press, true to the nature of the real Simon Game.

3. Justification and evaluation

Our design turned out to be simpler and more organized than our initial designs. We went through a series of implementation methods for each of the main components in our game, such as the random number generator, memory, and lights and sounds. Most of our design is based on binary button presses that other processes use. One random number generator implementation method was to generate a list of random numbers and store them. This method could have caused timing issues at the beginning, thus we only stored what we needed using user button pushes and a counter. Another benefit of our design is that we can easily separate the sound from the lights and from the button pushes, which resulted in cleaner states. Thus, it would be pretty simple to add a feature that allows the user to turn off the sound or lights.

There are still improvements that could be made to our design. Our datapath could be cleaned up a bit. There are many conversions of a single input or output from four bits to two bits that could possibly be minimized. We could have further simplified our design if we used a counter instead of a slow clock to run some processes that the user interacts with. Even though our slow clock is based on our fast clock, having a counter may reduce some possible implementation issues. If we had to do this project over again, besides simplifying our datapath even further, we might try to see how we could make a FSM and datapath that allows the user to choose the number of buttons to include in a Simon Game. If we were to use our current code, we would have to heavily restructure our design to make this multiple button feature work.

4. Conclusions

We initially proposed to create a classic Simon Game with many added features that would further challenge the user. We originally planned on adding a few more buttons, lights, and sounds so the user could choose to play with more than four options. We also wanted to let the user adjust the speed at which the sequence is played or allow the user to either disable lights or sound from the game. Ultimately, we added the high score and current score feature we originally proposed to our final Simon Game and added a button which allows the user to clear the high score display. We also added a sequence of lights and sound specifically for losing.

We recommend putting a lot of time and thought into the finite state machine and the datapath. We went through many designs and it paid off when we were debugging and writing code. Also, incorporating the additional features into separate designs during the design process may help with visualizing how that feature would fit in the simplest design.

5. Acknowledgements

We would like to thank Professor Hansen for teaching us what we needed to know to successfully complete this project and advising us through the development process. We would also like to thank Dave Picard for his valuable insights on our project and for obtaining the parts we needed. An additional thanks to all the TAs - Tyler A., Dan, Zach, and Tyler K. - who have spent their nights in Thayer and helped us with debugging and design ideas.

Both of us worked together on building the project to make sure each person knew how everything worked. Thus, both of us understood the design and code well. While writing the

report, John focused on the sections of the report that dealt more closely with the code and Jennifer focused on sections of the report were closely related to design. We both revised all parts of the report.

6. References

We referred to Professor Hansen's class lectures and used the debouncer and Mux 7-segment display code he had provided. We also used the clock divider code from lab 3 and lab 4 in our Simon Game code.

ENGS 31 Summer 2016 Final Report

John Sullivan and Jennifer Jain

Simon Game

APPENDICES

Table of Contents

Appendix A: Front Panel

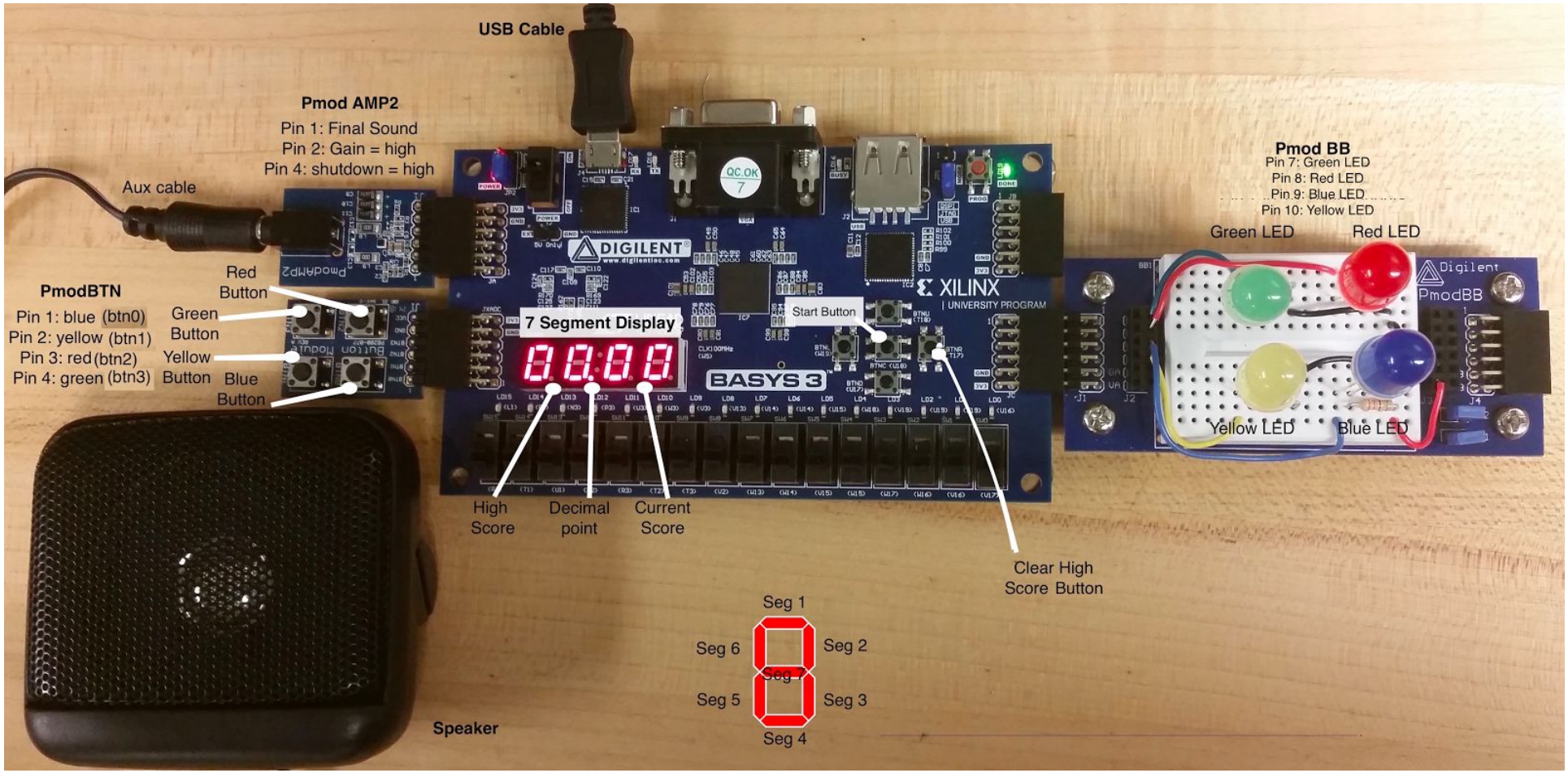


Figure 1: Front Panel of FPGA The input and output pins are labeled above. This diagram excludes the exact layout of the segments and anodes within the 7 segment display.

Appendix B: Top Level Block Diagram

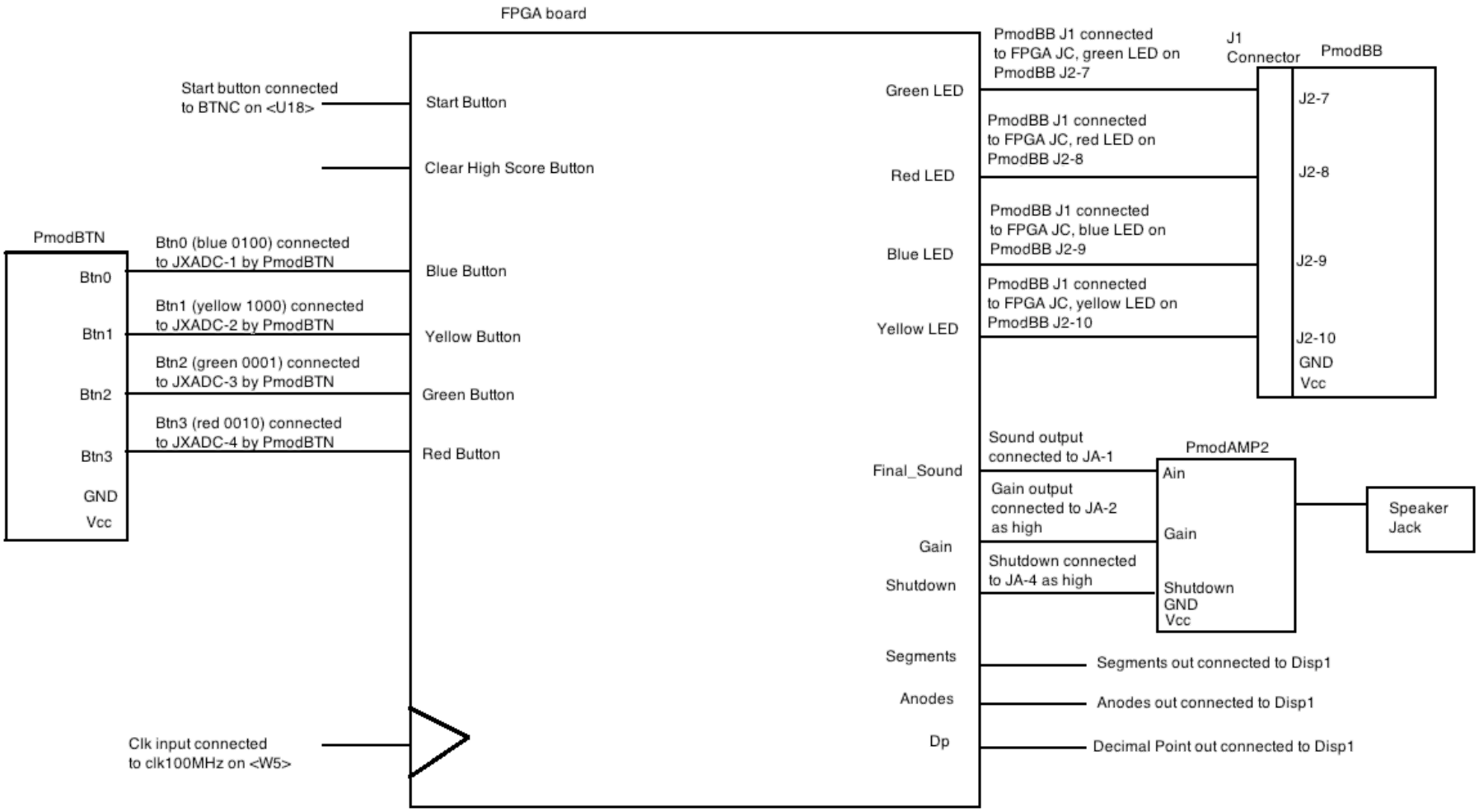


Figure 1: Top Level Block Diagram This diagram shows all of the main functional blocks and the associated ports. The FPGA takes in button presses as shown on the left and outputs a sound to the Pmod AMP2 and a light sequence to the PmodBB as well as BCD signals to the FPGA 7-segment display.

Appendix B: Top Level Datapath

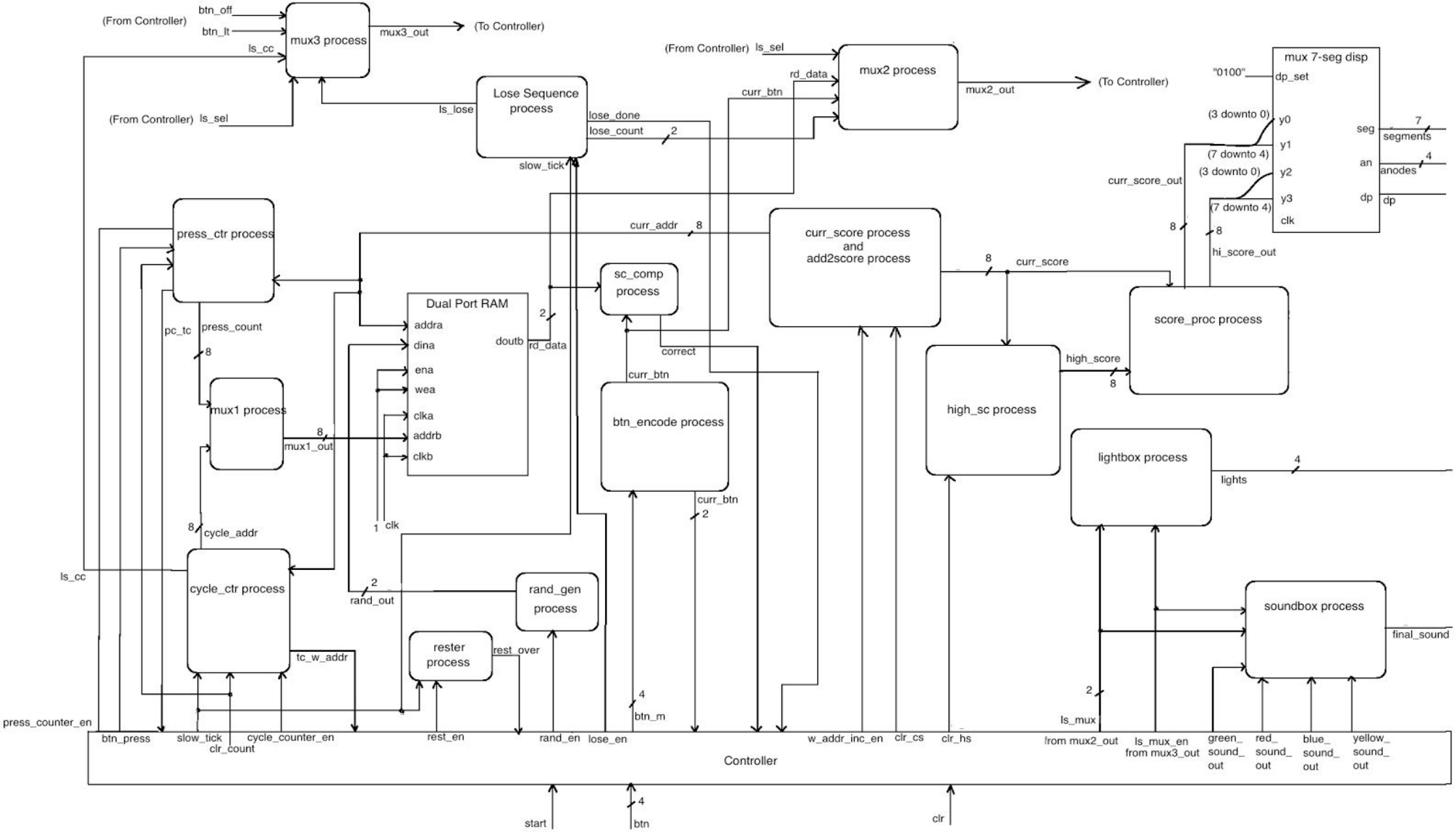


Figure 2: Top Level Datapath This diagram displays all the processes happening in the Simon Game. The internal logic of these processes are shown in the next few pages.

Appendix B: Component Level Datapaths

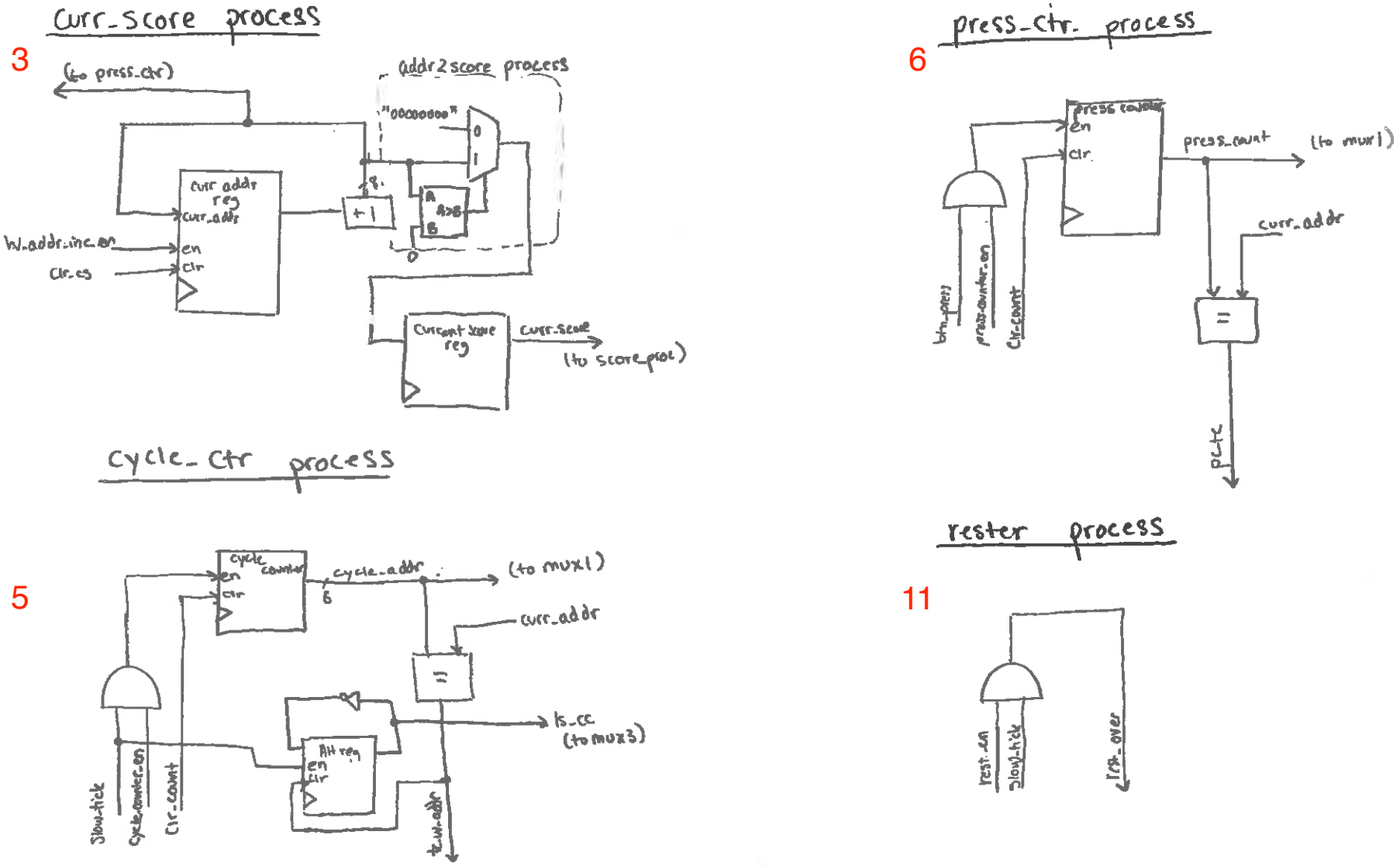


Figure 3:

This page shows internal component datapaths for [3] curr_score process, [6] press_ctr process, [5] cycle_ctr process and [11] rester process.

Appendix B: Component Level Datapaths

17

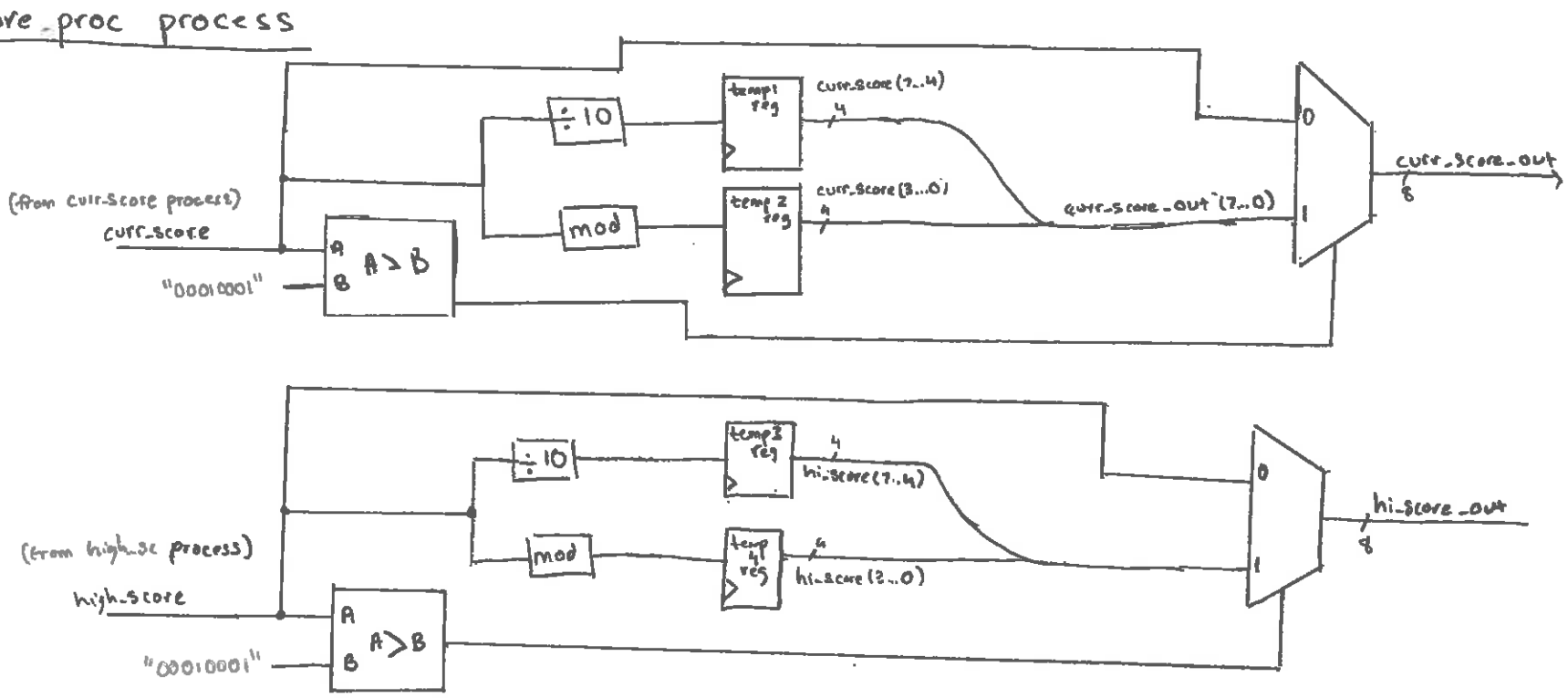


Figure 4:
This page shows the internal component datapath for the [17] score_proc process.

Appendix B: Component Level Datapaths

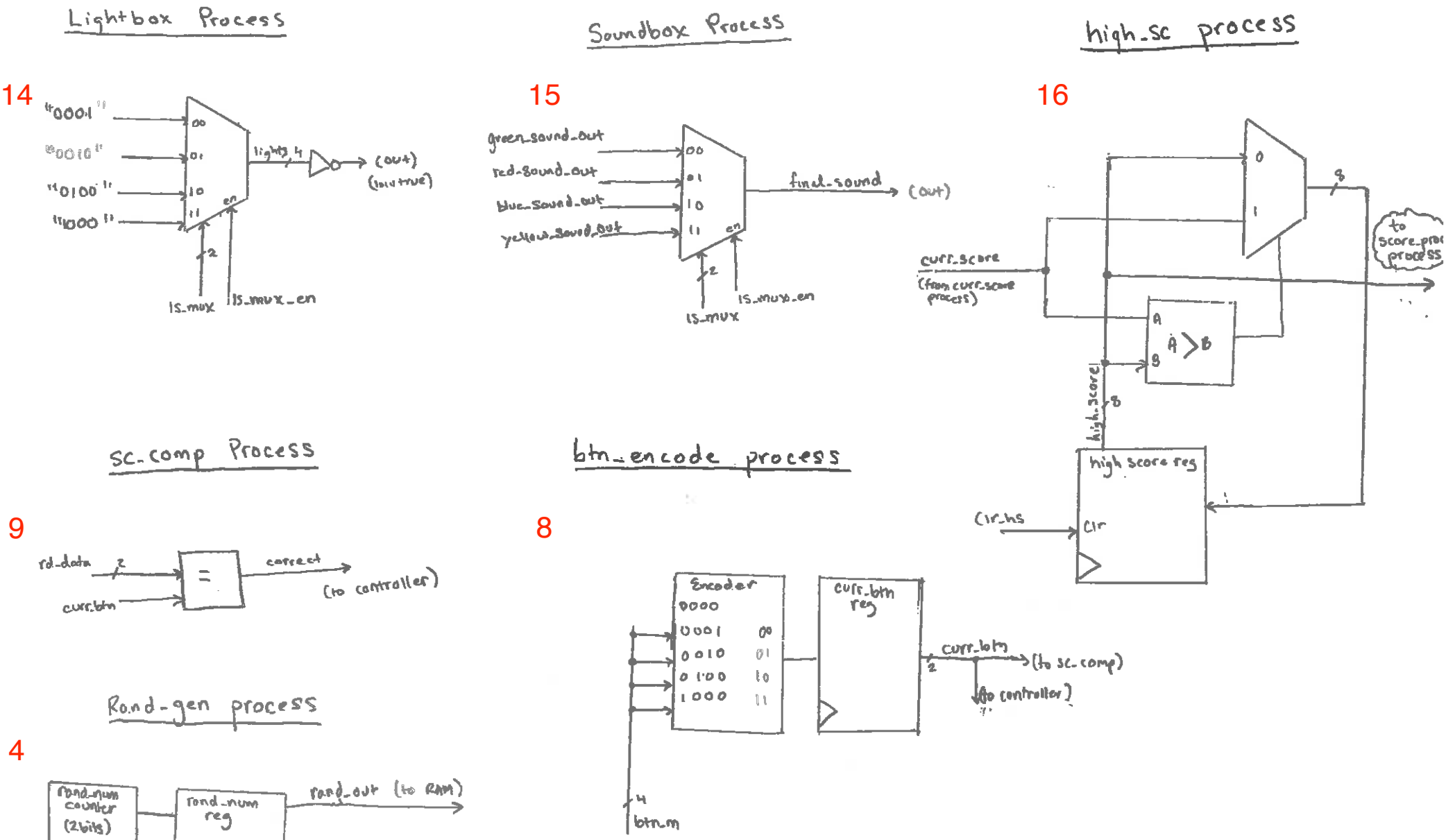
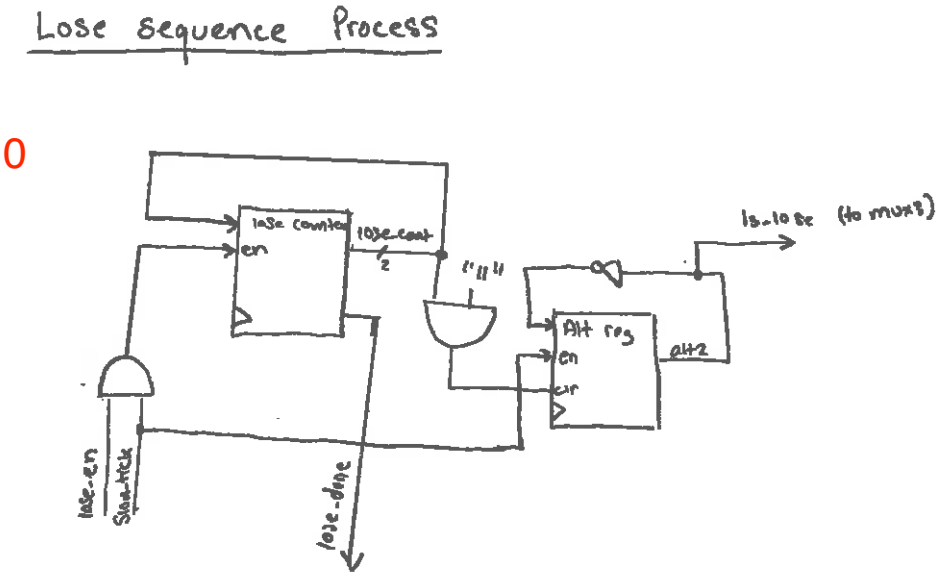
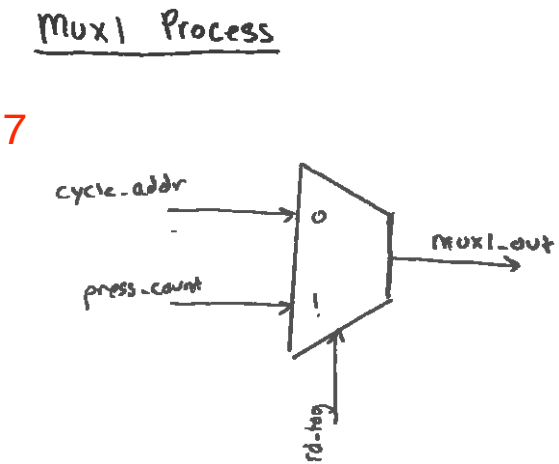
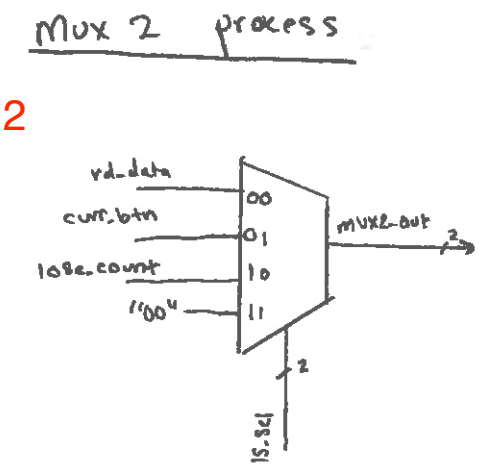


Figure 5:
This page shows internal component datapaths for [14] lightbox process, [15] soundbox process, [16] high_sc process, [9] sc_comp process, [8] btn_encode process and [4] rand_gen process.

Appendix B: Component Level Datapaths







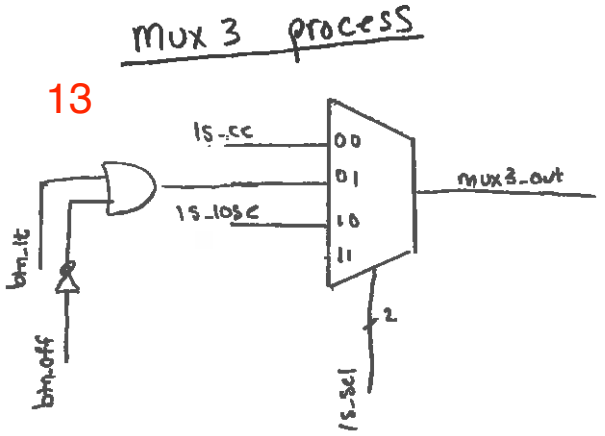


Figure 6:

This page shows internal component datapaths for [10] lose sequence process, [7] mux1 process, [12] mux2 process and [13] mux3 process.

Appendix C: Top Level State Machine

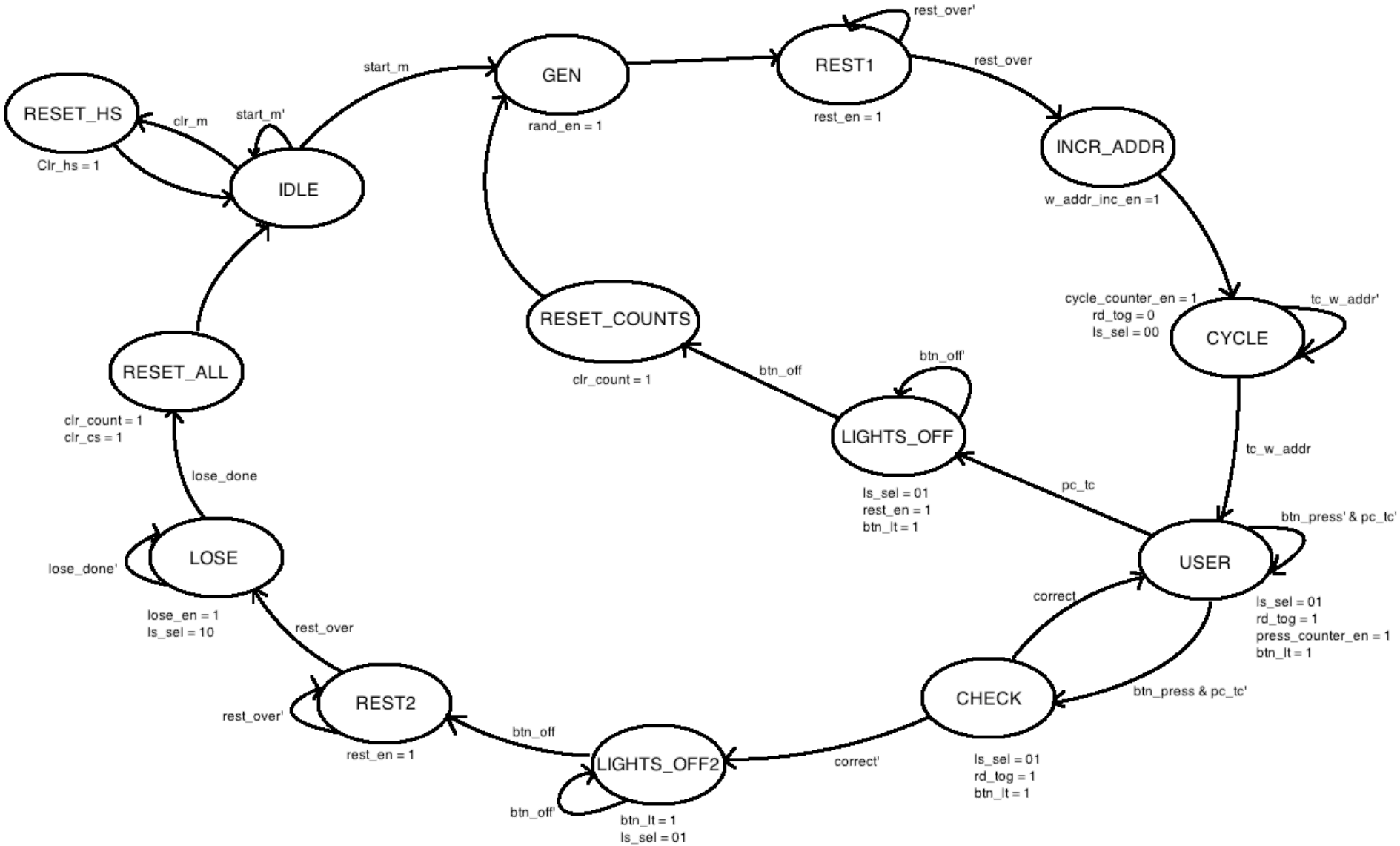


Figure 1: Top Level State Machine This state machine shows the function of the controller, which takes in button presses from the user and sends other signals to the datapath. State signals are initialized as zero and, unless defined as a value within the state, are set at zero.

Appendix C: Monopulse State Machine

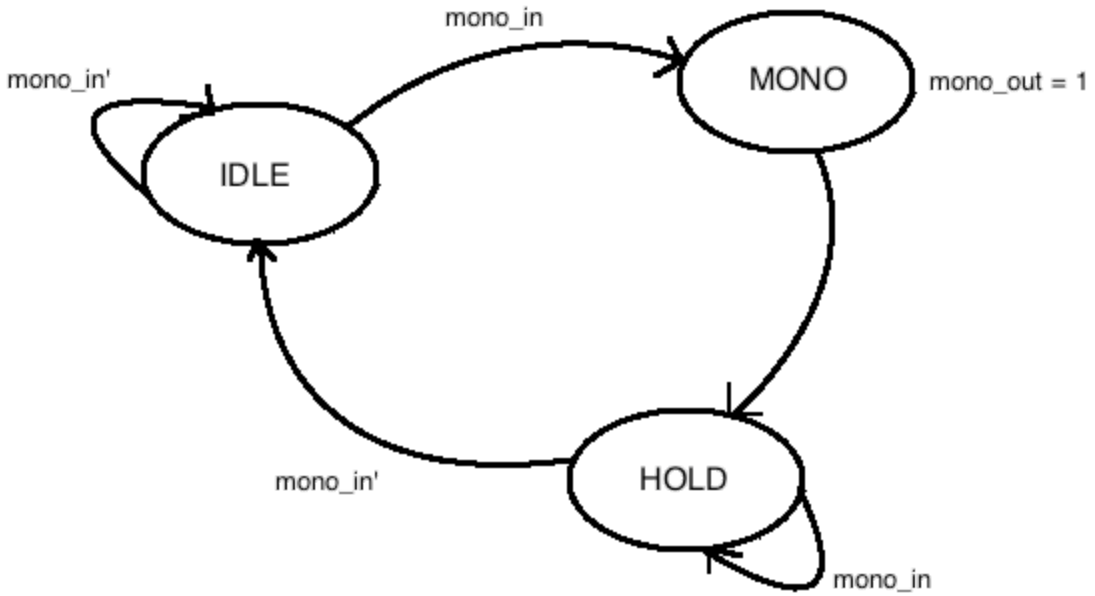
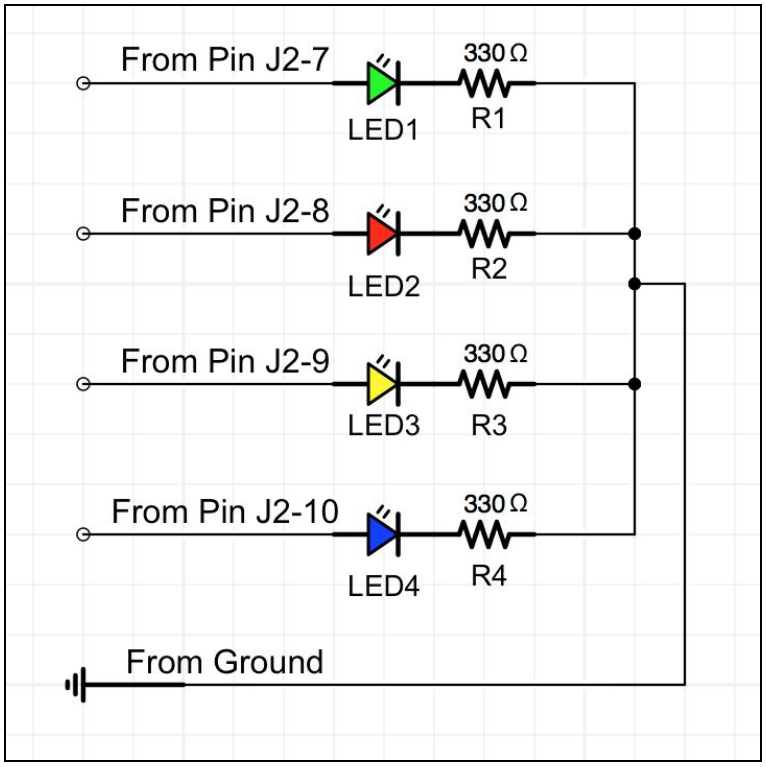


Figure 2: Monopulse State Machine This state machine controls button press signals. It only allows a signal to be sent the first time a button is pressed or held, then only sends another signal after the button has been released and pressed or held again.

Appendix D: Breadboard Schematic Diagram



Appendix E: Parts List

PARTS LIST		
Part Name	**Quantity**	**Description**
Basys 3	1	Digilent Basys 3 FPGA board
Pmod Connectors		
PmodBB	1	Digilent 12-pin connector, 170 tie point solderless breadboard
PmodAMP2	1	Digilent 6-pin Pmod port with GPIO interface, filterless high efficiency audio amplifier with mono speaker jack and gain
PmodBTN	1	Digilent 6-pin Pmod connector with GPIO interface and 4 buttons
Discrete Components		
Green LED	1	Any 4 differently colored LEDs that can be inserted into a breadboard would suffice
Blue LED	1	
Red LED	1	
Yellow LED	1	
330 Ohm Resistors	4	One resistor per LED used on the breadboard Pmod, needed to reduce current flow
Additional Components		
Speaker	1	Digilent Speaker with audio jack, any other speaker would also suffice

Appendix F: VHDL Code 1
Simon Game Top Level File

```
----------------------------------------------------------------------------------
-- Company: ENGS 31 16X
-- Engineers: John Sullivan and Jennifer Jain
--
-- Create Date: 08/10/2016 09:07:00 PM
-- Design Name: Simon Game Top
-- Module Name: simon_game_top - Behavioral
-- Project Name: Simon Game
-- Target Devices: Digilent Basys 3 board (Artix 7)
-- Tool versions: Vivado 2016.1
-- Description: Top-level file for the Simon Game project (includes controller)
--
-- Dependencies: Requires datapath, soundbox, monopulser, MAX6816
--                        mux7seg, and debouncer files to function
--
-- Revision:
-- Revision 0.01 - File created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity simon_game is
    port (
        -- INPUTS:
        btn : in std_logic_vector(3 downto 0);
        start, clr : in std_logic;
        clk : in std_logic;

        -- OUTPUTS:
        lights_out : out std_logic_vector(3 downto 0);
        final_sound : out std_logic;

        -- Multiplexed seven segment display:
        segments : out std_logic_vector(0 to 6);
        dp : out std_logic;
        anodes : out std_logic_vector(3 downto 0);

        -- Forced high outputs for audio:
        amp_shut, gain : out std_logic
    );
end simon_game;

architecture Behavioral of simon_game is

    -- CONSTANTS:

    -- Color sound frequencies (based on real Simon Game):
    constant GREEN_FREQ : integer := 415;
    constant RED_FREQ : integer := 310;
    constant BLUE_FREQ : integer := 209;
    constant YELLOW_FREQ : integer := 252;
```

```
    -- COMPONENTS:

    -- Multiplexed seven segment display (taken from lab 4) - Converts number to output for
       seven segment display
    component mux7seg is
        port (
            clk : in STD_LOGIC;
            y0, y1, y2, y3 : in STD_LOGIC_VECTOR (3 downto 0);
            dp_set : in std_logic_vector(3 downto 0);
            seg : out STD_LOGIC_VECTOR (0 to 6);
            dp : out std_logic;
            an : out STD_LOGIC_VECTOR (3 downto 0)
        );
    end component;

    -- Debouncer (by Professor Hansen) - Checks that button has truly been pressed
    component MAX6816 is
        port (
            clk : in STD_LOGIC;
            switch : in STD_LOGIC; -- switch input
        switch_db : out STD_LOGIC); -- debounced output
    end component;

    -- Datapath - Contains bulk of combinational logic for our device
    component datapath is
        port (-- INPUTS:
            clk, rand_en, w_addr_inc_en, cycle_counter_en, rd_tog, press_counter_en,
clr_count,
                clr_cs, lose_en, clr_hs, btn_press, btn_lt, btn_off, rest_en : in std_logic;
            curr_btn, ls_sel : in std_logic_vector(1 downto 0);

            -- OUTPUTS:
            tc_w_addr, correct, pc_tc, lose_done, ls_mux_en, rest_over : out std_logic;
            ls_mux : out std_logic_vector(1 downto 0);
            curr_score_out, hi_score_out : out std_logic_vector(7 downto 0)
        );
    end component;

    -- Monopulser - For monopulsing a button input
    component monopulser is
        port (
            clk, mono_in : in std_logic;
            mono_out : out std_logic
        );
    end component;

    -- Soundbox - Based on the desired sound frequency, produces a square wave to produce that
       sound
    component soundbox is
        generic (output_freq : integer);
        port (
            clk : in std_logic;
```

```
            sound_out : out std_logic
        );
    end component;


    -- SIGNALS:

    -- CONTROLLER (see state diagram for illustration):
    -- States:
    type state_type is (Idle, Gen, Load, Incr_Addr, Cycle, User, Check, Lights_Off,
Lights_Off_2,
        Lose, Reset_All, Comp_Score, Update_HS, Reset_Counts, Reset_HS, Rest1, Rest2);
    signal c_s, n_s : state_type; -- current and next state

    -- Transitions:
    signal tc_w_addr, btn_press, correct, pc_tc, btn_off, lose_done, rest_over : std_logic :=
'0';

    -- State outputs:
    signal rand_en, w_addr_inc_en, cycle_counter_en, rd_tog, press_counter_en, clr_count,
        clr_cs, lose_en, clr_hs, btn_lt, rest_en : std_logic := '0';

    -- 2-bit state outputs to datapath:
    signal ls_sel, curr_btn : std_logic_vector(1 downto 0) := "00";


    -- Lights/sounds select and enable from the datapath:
    signal ls_mux : std_logic_vector(1 downto 0) := "00";
    signal ls_mux_en : std_logic := '0';


    -- DEBOUNCER:
    signal btn_db : std_logic_vector(3 downto 0);
    signal start_db, clr_db : std_logic;

    -- MONOPULSER:
    signal btn_m : std_logic_vector(3 downto 0);
    signal start_m, clr_m : std_logic;

    -- SOUNDBOX:
    signal green_sound_out, red_sound_out, blue_sound_out, yellow_sound_out : std_logic := '0';

    -- LIGHTS:
    signal lights : std_logic_vector(3 downto 0) := "0000";

    -- SCORE DISPLAY:
    signal curr_score_out, hi_score_out : std_logic_vector(7 downto 0) := (others => '0');

begin
    -- PORT MAPS:

    -- Mux7seg port map (taken from lab 4) - Instantiating the multiplexed seven segment
       display
    display : mux7seg
```

```
port map(
clk => clk, -- runs off the master clock
y3 => hi_score_out(7 downto 4), -- most significant digit
y2 => hi_score_out(3 downto 0), -- map to actual signals
y1 => curr_score_out(7 downto 4), -- when you add the counters
y0 => curr_score_out(3 downto 0), -- least significant digit always zero
dp_set => "0100", -- which decimal points to turn on (high true)
seg => segments,
dp => dp,
an => anodes
);

-- Debouncer port maps (taken from lab 4) - Instatantiating the debouncers
db0 : MAX6816
port map(
clk => clk,
switch => btn(0),
switch_db => btn_db(0)
);

db1 : MAX6816
port map(
clk => clk,
switch => btn(1),
switch_db => btn_db(1)
);

db2 : MAX6816
port map(
clk => clk,
switch => btn(2),
switch_db => btn_db(2)
);

db3 : MAX6816
port map(
clk => clk,
switch => btn(3),
switch_db => btn_db(3)
);

db_start : MAX6816
port map(
clk => clk,
switch => start,
switch_db => start_db
);

db_clr : MAX6816
port map(
clk => clk,
switch => clr,
switch_db => clr_db
);
```

```
-- Monopulser port maps:
mp0 : monopulser
port map(
clk => clk,
mono_in => btn_db(0),
mono_out => btn_m(0)
);

mp1 : monopulser
port map(
clk => clk,
mono_in => btn_db(1),
mono_out => btn_m(1)
);

mp2 : monopulser
port map(
clk => clk,
mono_in => btn_db(2),
mono_out => btn_m(2)
);

mp3 : monopulser
port map(
clk => clk,
mono_in => btn_db(3),
mono_out => btn_m(3)
);

mp_start : monopulser
port map(
clk => clk,
mono_in => start_db,
mono_out => start_m
);

mp_clr : monopulser
port map(
clk => clk,
mono_in => clr_db,
mono_out => clr_m
);


-- Datapath port map:
dpath : datapath
port map(
clk => clk,
rand_en => rand_en,
w_addr_inc_en => w_addr_inc_en,
cycle_counter_en => cycle_counter_en,
rd_tog => rd_tog,
```

```
press_counter_en => press_counter_en,
clr_count => clr_count,
clr_cs => clr_cs,
lose_en => lose_en,
clr_hs => clr_hs,
btn_press => btn_press,
curr_btn => curr_btn,
ls_sel => ls_sel,
tc_w_addr => tc_w_addr,
correct => correct,
pc_tc => pc_tc,
btn_lt => btn_lt,
btn_off => btn_off,
lose_done => lose_done,
ls_mux_en => ls_mux_en,
ls_mux => ls_mux,
curr_score_out => curr_score_out,
hi_score_out => hi_score_out,
rest_en => rest_en,
rest_over => rest_over
);


-- Soundbox port maps:
green_sound : soundbox
generic map(output_freq => GREEN_FREQ)
port map(
sound_out => green_sound_out,
clk => clk
);

red_sound : soundbox
generic map(output_freq => RED_FREQ)
port map(
sound_out => red_sound_out,
clk => clk
);

blue_sound : soundbox
generic map(output_freq => BLUE_FREQ)
port map(
sound_out => blue_sound_out,
clk => clk
);

yellow_sound : soundbox
generic map(output_freq => YELLOW_FREQ)
port map(
sound_out => yellow_sound_out,
clk => clk
);


-- CONTROLLER:
```

```
    -- Process to update the state on new rising clock edges:
    state_update : process(clk)
    begin
        if rising_edge(clk) then
            c_s <= n_s;
        end if;
    end process state_update;

    -- Combinational logic for the controller to determine transitions between states and
       outputs from the states:
    -- (see state diagram illustration for more information)
    comb_logic : process(c_s, start_m, tc_w_addr, btn_press, clr_m, correct, pc_tc, btn_off,
lose_done, rest_over)
    begin
        -- Defaults:
        n_s <= c_s;

        rand_en <= '0';
        w_addr_inc_en <= '0';
        cycle_counter_en <= '0';
        rd_tog <= '0';
        press_counter_en <= '0';
        clr_count <= '0';
        clr_cs <= '0';
        lose_en <= '0';
        clr_hs <= '0';
        ls_sel <= "00";
        btn_lt <= '0';
        rest_en <= '0';

        -- State outputs and transitions:
        case c_s is
            when Idle =>

                if start_m = '1' then
                    n_s <= Gen;
                elsif clr_m = '1' then
                    n_s <= Reset_HS;
                end if;

            when Gen =>
                rand_en <= '1';

                n_s <= Rest1;

            when Incr_Addr =>
                w_addr_inc_en <= '1';

                n_s <= Cycle;

            when Cycle =>
                cycle_counter_en <= '1';
                rd_tog <= '0';
```

```
        ls_sel <= "00";

        if tc_w_addr = '1' then
            n_s <= User;
        end if;

    when User =>
        ls_sel <= "01";
        rd_tog <= '1';
        press_counter_en <= '1';
        btn_lt <= '1';

        if pc_tc = '1' then
            n_s <= Lights_Off;
        elsif btn_press = '1' and pc_tc = '0' then
            n_s <= Check;
        end if;

    when Check =>

        rd_tog <= '1';
        btn_lt <= '1';
        ls_sel <= "01";

        if correct = '1' then
            n_s <= User;
        else
            n_s <= Lights_Off_2;
        end if;

    when Lights_Off =>
        btn_lt <= '1';
        ls_sel <= "01";
        rest_en <= '1';
        if btn_off = '1' then
            n_s <= Reset_Counts;
        end if;

    when Rest1 =>
        rest_en <= '1';

        if rest_over = '1' then
            n_s <= Incr_Addr;
        end if;

    when Lose =>
        lose_en <= '1';
        ls_sel <= "10";

        if lose_done = '1' then
            n_s <= Reset_All;
        end if;

    when Lights_Off_2 =>
```

```
            btn_lt <= '1';
            ls_sel <= "01";

            if btn_off = '1' then
                n_s <= Rest2;
            end if;

        when Rest2 =>
            rest_en <= '1';

            if rest_over = '1' then
                n_s <= Lose;
            end if;

        when Reset_Counts =>
            clr_count <= '1';

            n_s <= Gen;

        when Reset_All =>

            clr_count <= '1';
            clr_cs <= '1';

            n_s <= Idle;

        when Reset_HS =>
            clr_hs <= '1';

            n_s <= Idle;

        when others => null;
    end case;
end process comb_logic;


-- PROCESSSES INVOLVING THE BUTTONS:

-- Encode the button from 4-bits to 2-bits:
btn_encode : process(clk)
begin
    if rising_edge(clk) then
        if btn_m = "0000" then
            curr_btn <= curr_btn;
        elsif btn_m = "0001" then
            curr_btn <= "00";
        elsif btn_m = "0010" then
            curr_btn <= "01";
        elsif btn_m = "0100" then
            curr_btn <= "10";
        elsif btn_m = "1000" then
            curr_btn <= "11";
        end if;
    end if;
```

```
    end process btn_encode;

    -- Signal when the button has been pressed:
    bp : process(btn_m)
    begin
        if btn_m /= "0000" then
            btn_press <= '1';
        else
            btn_press <= '0';
        end if;

    end process bp;

    -- Signal when button released:
    btn_release : process(btn_db)
    begin
        if btn_db = "0000" then
            btn_off <= '1';
        else btn_off <= '0';
        end if;
    end process btn_release;


    -- PROCESSES INVOVLING LIGHTS AND SOUND:

    -- Lightbox - When the light/sound enable is on, this will turn on the desired light:
    lightbox : process(ls_mux, ls_mux_en)
    begin
        if ls_mux_en = '1' then
            if ls_mux = "00" then
                lights <= "0001";
            elsif ls_mux = "01" then
                lights <= "0010";
            elsif ls_mux = "10" then
                lights <= "0100";
            elsif ls_mux = "11" then
                lights <= "1000";
            else lights <= "0000";
            end if;
        else lights <= "0000";
        end if;
    end process lightbox;

    lights_out <= not(lights); -- Since the LEDs are low true, the output from our
                               -- lights signal must be inverted for the final output

    -- Sounds - When the light/sound enable is on, this will turn on the desired osund:
    -- (The sound_out signals come from the soundboxes, which
    -- output a square wave for the desired sound frequency)
    sounds : process(ls_mux, ls_mux_en, green_sound_out, red_sound_out, blue_sound_out,
yellow_sound_out)
    begin
        if ls_mux_en = '1' then
            if ls_mux = "00" then
```

```
                final_sound <= green_sound_out;
            elsif ls_mux = "01" then
                final_sound <= red_sound_out;
            elsif ls_mux = "10" then
                final_sound <= blue_sound_out;
            elsif ls_mux = "11" then
                final_sound <= yellow_sound_out;
            else final_sound <= '0';
            end if;
        else
            final_sound <= '0';
        end if;
    end process;

    -- These are set to force specific inputs to the PmodAMP2 high as desired:
    amp_shut <= '1';   -- Keeps the PmodAMP2 on
    gain <= '1';       -- Sets a 6 dB gain for the audio
                       -- output as opposed to 12 dB
end Behavioral;
```

```
----------------------------------------------------------------------------------
-- Company: ENGS 31 16X
-- Engineers: John Sullivan and Jennifer Jain
--
-- Create Date: 08/12/2016 02:12:38 PM
-- Design Name: Datapath
-- Module Name: datapath - Behavioral
-- Project Name: Simon Game
-- Target Devices: Digilent Basys 3 board (Artix 7)
-- Tool versions: Vivado 2016.1
-- Description: Datapath for the Simon Game project,
--              consisting of bulk of combinational logic
--
-- Dependencies: Requires dual-ported block RAM from the IP
--               Catalog to have been properly generated
--
-- Revision:
-- Revision 0.01 - File created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity datapath is
    port (
        -- INPUTS:
        clk, rand_en, w_addr_inc_en, cycle_counter_en, rd_tog, press_counter_en, clr_count,
clr_cs, lose_en, clr_hs, btn_press, btn_lt, btn_off, rest_en : in std_logic;
        curr_btn, ls_sel : in std_logic_vector(1 downto 0);

        -- OUTPUTS:
        tc_w_addr, correct, pc_tc, lose_done, ls_mux_en, rest_over : out std_logic;
        ls_mux : out std_logic_vector(1 downto 0);
        curr_score_out, hi_score_out : out std_logic_vector(7 downto 0)
    );
end datapath;

architecture Behavioral of datapath is

    -- COMPONENTS:

    -- Simon RAM
    component simon_ram
        port (
            clka : in STD_LOGIC;
            ena : in STD_LOGIC;
            wea : in STD_LOGIC_VECTOR(0 downto 0);
            addra : in STD_LOGIC_VECTOR(7 downto 0);
            dina : in STD_LOGIC_VECTOR(1 downto 0);
            clkb : in STD_LOGIC;
            addrb : in STD_LOGIC_VECTOR(7 downto 0);
            doutb : out STD_LOGIC_VECTOR(1 downto 0)
```

```
    );
end component;


-- SIGNALS:

-- RAM:
signal rd_data : std_logic_vector(1 downto 0) := (others => '0');

-- 1. Clock Divider (modified from lab 4):
-- Signals for the clock divider, which divides the master clock
constant CDV : integer := 100E6/5; -- Basys3 board has 100 MHz clock
constant CLOCK_DIVIDER_VALUE : integer := CDV; -- use CDV for implementation; use 10 for
    simulation
signal clkdiv : integer := 0; -- the clock divider counter
signal slow_tick : std_logic := '0';

-- 2. Random Number Generator:
signal rand_num, rand_out : unsigned(1 downto 0) := "00";

-- 3. Current Score Address:
signal curr_addr : unsigned(7 downto 0) := (others => '0');

-- 4. Cycle Counter:
signal cycle_addr : unsigned(7 downto 0) := (others => '0');
signal fr_cc, sr_cc : std_logic := '1';
signal tc_read : std_logic := '0';
signal ls_cc : std_logic := '0';
signal alt : std_logic := '0';

-- 5. Press Counter:
signal press_count : unsigned(7 downto 0) := (others => '0');
signal pc_done : std_logic := '0';
signal first_run_pc : std_logic := '1';

-- 6. Mux1:
signal mux1_out : std_logic_vector(7 downto 0) := (others => '0');

-- 8. Lose sequence:
signal fr_lose : std_logic := '1';
signal lose_count : unsigned(1 downto 0) := "00";
signal lose_pass : unsigned(0 downto 0) := "0";
signal lose_read : std_logic := '0';
signal ls_lose : std_logic := '0';
signal alt2 : std_logic := '0';

-- 9. Mux2:
signal mux2_out : std_logic_vector(1 downto 0) := (others => '0');

-- 10. Mux3:
signal mux3_out : std_logic := '0';

-- 11. High score:
signal high_score : unsigned(7 downto 0) := (others => '0');
```

```
    -- 12. Address to Score:
    signal curr_score : unsigned(7 downto 0) := (others => '0');

    -- 13. Score Processor:
    signal temp1, temp2, temp3, temp4 : unsigned(7 downto 0) := (others => '0');

    -- 14. Rester:
    signal rest_tog : std_logic := '0';

begin
    -- PORT MAPS:

    -- Simon Ram:
    main_ram : simon_ram
    port map(
    clka => clk,
    ena => '1',
    wea => (others => '1'),
    addra => std_logic_vector(curr_addr),
    dina => std_logic_vector(rand_out),
    clkb => clk,
    addrb => mux1_out,
    doutb => rd_data
    );


    -- 1. Clock divider (modified from lab 4) - Generates a slow tick that may is used
    --                                          as an enable to slow certain processes
    Clock_divider : process(clk)
    begin
        if rising_edge(clk) then
            if clkdiv = CLOCK_DIVIDER_VALUE - 1 then     -- When the desired terminal count is
                clkdiv <= 0;                             -- reached, the slow_count is asserted
                slow_tick <= '1';                        -- for one clock cycle and clkdiv
                    reset to 0
            else
                clkdiv <= clkdiv + 1;
                slow_tick <= '0';
            end if;
        end if;
    end process Clock_divider;

    -- 2. Random Number Generator - Clocked counter; when user presses or gets correct
    --                              sequence, will cause rand_en to turn on and the counter
        number
    --                              to be stored; the stored value is effectively random
    rand_gen : process(clk)
    begin
        if rising_edge(clk) then
            rand_num <= rand_num + 1;    -- This is the counter

            if rand_en = '1' then        -- This will store the counter
                rand_out <= rand_num;    -- value when desired
```

```
        end if;
    end if;
end process rand_gen;

-- 3. Current Score Address - This starts at 0 when the user stats the game and increments
--                            everytime the user gets the correct sequence; it is used
--                            to keep track of the number of addresses in the ram for
--                            cycling through and comparison with user input
curr_score_addr : process(clk)
begin
    if rising_edge(clk) then
        if clr_cs = '1' then                -- This is cleared and incremented
            curr_addr <= (others => '0');   -- at the desired states on the diagram
        elsif w_addr_inc_en = '1' then
            curr_addr <= curr_addr + 1;
        end if;
    end if;
end process curr_score_addr;

-- 4. Cycle Counter - Using the slower tick signal, this will cycle through the contents
--                    of the RAM to display the sequence of buttons they should enter;
--                    it is also set to generate lights and sounds for the sequence with
--                    a rest in between each display so the user may distinguish the
    signals
cycle_ctr : process(clk)
begin
    if rising_edge(clk) then
        if slow_tick = '1' then
            if clr_count = '1' then                         -- Clears at desired state
                cycle_addr <= (others => '0');
            elsif cycle_counter_en = '1' then               -- This first run case is
                necessary,
                if fr_cc = '1' then                         -- as without it on the first
                    run,
                    fr_cc <= '0';                           -- the terminal count would be
                    ls_cc <= '1';                           -- asserted right away
                    alt <= '1';
                else
                    if alt = '0' then                       -- Then starts displaying and
                        ls_cc <= '1';                       -- incrementing through the
                        cycle_addr <= cycle_addr + 1;       -- RAM to display lights and
                            sound
                        alt <= '1';
                    else
                        ls_cc <= '0';                       -- There will be an
                            alternation between the
                        alt <= '0';                         -- if and else parts of this
                            hierarchy to
                    end if;                                 -- allow the user to see the
                    different lights

                    if cycle_addr = (curr_addr - 1) then    -- When finished cycling
                        through
                        tc_read <= '1';                     -- RAM, the counter is reset
```

```
                        ls_cc <= '0';
                        cycle_addr <= (others => '0');
                        fr_cc <= '1';
                        alt <= '0';
                    end if;
                end if;
            end if;
        end if;

        if tc_read = '1' then                                   -- The terminal count to
            transition out of the
            tc_read <= '0';                                     -- cycling state is set right
                on the clock edge so
        end if;                                                 -- that it may immediately
            come out without delay
    end if;
end process cycle_ctr;

tc_w_addr <= tc_read;

-- 5. Press Counter - Everytime the user enters a button in trying to get the sequence,
--                    the press count is incremented for comparison with the highest
--                    address in the RAM for that sequence (indicated by curr_addr)
press_ctr : process(clk)
begin
    if rising_edge(clk) then
        if clr_count = '1' then                 -- Clears at the desired state
            press_count <= (others => '0');
            pc_done <= '0';
            first_run_pc <= '1';
        elsif press_counter_en = '1' then       -- When enable, increments on the
            if press_count = (curr_addr) then   -- button press, unless terminal
                pc_done <= '1';                 -- count has been reached, in which
                press_count <= (others => '0'); -- case, pc_tc is asserted
            elsif btn_press = '1' then
                press_count <= press_count + 1;
            end if;
        else
            pc_done <= '0';
        end if;
    end if;
end process press_ctr;

pc_tc <= pc_done;

-- 6. Mux1 - Chooses whether the RAM should be reading the contents
--           to cycle through the sequence for display to the user
--           (using cycle_addr) or should be waiting for each user
--           press to increment the address (using press_count)
mux1 : process(rd_tog, cycle_addr, press_count)
begin
    if rd_tog = '0' then                            -- rd_tog designates what read
        mux1_out <= std_logic_vector(cycle_addr);   -- address will be used based
    else                                            -- on the current state (cycle
```

```
        mux1_out <= std_logic_vector(press_count);    -- address or press count address)
    end if;
end process mux1;

-- 7. Score Compare - Outputs correct when the user button press
--                    matches that output from the RAM
sc_comp : process(rd_data, curr_btn)
begin
    if rd_data = curr_btn then
        correct <= '1';
    else
        correct <= '0';
    end if;
end process sc_comp;

-- 8. Lose sequence - When enabled, runs clockwise through the buttons
--                    twice to indicate to the user that the wrong
--                    button has been entered
lose_sequence : process(clk)
begin
    if rising_edge(clk) then
        if slow_tick = '1' then
            if (lose_en = '1') then
                if fr_lose = '1' then                    -- This refers to just as the
                    fr_lose <= '0';                      -- lose sequence is entered;
                    ls_lose <= '1';                      -- it is necessary to display
                    alt2 <= '1';                         -- the desired light sequence
                else
                    if alt2 = '0' then
                        ls_lose <= '1';                  -- Increments around lights
                            until
                        lose_count <= lose_count + 1;    -- two clockwise passes around
                        alt2 <= '1';                     -- them have been completed

                        if lose_count = 3 then
                            if lose_pass = 1 then
                                lose_read <= '1';
                                lose_count <= "00";
                                lose_pass <= "0";
                                fr_lose <= '1';
                                ls_lose <= '0';
                                alt2 <= '0';
                            else
                                lose_pass <= lose_pass + 1;
                            end if;
                        end if;
                    else
                        ls_lose <= '0';
                        alt2 <= '0';
                    end if;
                end if;
            end if;
        end if;
```

```
        if lose_read = '1' then                                    -- Resets lose_read after
            immediately
            lose_read <= '0';                                      -- fafter asserted for
                one clock cycle
        end if;
    end if;
end process lose_sequence;

lose_done <= lose_read;

-- 9. Mux2 - Selects at a given moment where the output used for
--           the lights and sound should be coming from
mux2 : process(ls_sel, rd_data, curr_btn, lose_count)
begin
    if ls_sel = "00" then     -- from the RAM
        mux2_out <= rd_data;
    elsif ls_sel = "01" then  -- from the current button press
        mux2_out <= curr_btn;
    elsif ls_sel = "10" then  -- from the lose sequence
        mux2_out <= std_logic_vector(lose_count);
    else
        mux2_out <= "00";     -- nothing
    end if;
end process mux2;

ls_mux <= mux2_out;

-- 10. Mux3 - Selects at a given moment where the enable used for
--            the lights and sound should be coming from
mux3 : process(ls_sel, btn_off, btn_lt, ls_cc, ls_lose)
begin
    if ls_sel = "00" then     -- reperesents data from ram cycled through
        mux3_out <= ls_cc;
    elsif ls_sel = "01" then  -- represents user button press
        mux3_out <= not(btn_off) and btn_lt;
    elsif ls_sel = "10" then  -- from the lose sequence
        mux3_out <= ls_lose;
    else                      -- nothing
        mux3_out <= '0';
    end if;
end process mux3;

ls_mux_en <= mux3_out;

-- 11. High Score - Tracks the high score based on whether the user
--                  has achieved a score greater than that currently stored
--                  as the high score; able to be cleared when appropriate
high_sc : process(clk)
begin
    if rising_edge(clk) then
        if clr_hs = '1' then                -- Clears at the higher state
            high_score <= (others => '0');
        else
            if curr_score > high_score then  -- Sets high score to the current score if
```

```
                high_score <= curr_score;    -- the current score exceeds the high score
            end if;
        end if;
    end if;
end process high_sc;

-- 12. Address to Score - Converts the highest address being stored in
--                        on the RAM (indicated by curr_addr) to the
--                        current score for display on the board
addr2score : process(curr_addr)
begin
    if curr_addr > 0 then                   -- Subtracting one if current score
        curr_score <= curr_addr - 1;        -- greater than 0
    else curr_score <= (others => '0');     -- Otherwise sets it to 0
    end if;
end process addr2score;

-- 13. Score Processor - Since the score was being stored as an unsigned
--                       number, it must be converted to a two digit
--                       binary coded decimal (8-bits in total) for output
--                       to the mux7seg component and display on the board
score_proc : process(curr_score, high_score, temp1, temp2, temp3, temp4)
begin
    if curr_score > 9 then            -- If curr score is two digits
        temp1 <= curr_score / 10;     -- Gets first digit
        temp2 <= curr_score mod 10;   -- Gets second digit

        curr_score_out(7 downto 4) <= std_logic_vector(temp1(3 downto 0));   -- Stores
            first digit
        curr_score_out(3 downto 0) <= std_logic_vector(temp2(3 downto 0));   -- Stores
            second digit
    else
        curr_score_out(7 downto 4) <= (others => '0');                        -- First
            digit is 0
        curr_score_out(3 downto 0) <= std_logic_vector(curr_score(3 downto 0));  -- Get
            second digit

        temp1 <= (others => '0');    -- No need to use these
        temp2 <= (others => '0');
    end if;

    if high_score > 9 then            -- If high score is two digits
        temp3 <= high_score / 10;     -- Gets first digit
        temp4 <= high_score mod 10;   -- Gets second digit

        hi_score_out(7 downto 4) <= std_logic_vector(temp3(3 downto 0));   -- Stores first
            digit
        hi_score_out(3 downto 0) <= std_logic_vector(temp4(3 downto 0));   -- Stores
            second digit
    else
        hi_score_out(7 downto 4) <= (others => '0');                        -- First
            digit is 0
        hi_score_out(3 downto 0) <= std_logic_vector(high_score(3 downto 0));  -- Get
            second digit
```

```
            temp3 <= (others => '0');    -- No need to use these
            temp4 <= (others => '0');
        end if;
    end process;

    -- 14. Rester - At some points in our state diagram, it was necessary to wait
    --              momentarily, such that the states did not appear to be
    --              happening so quickly the user could not distinguish them;
    --              this process use the slow_tick to achieve this effect
    rester : process(clk)
    begin
        if rising_edge(clk) then
            if slow_tick = '1' then
                if rest_en = '1' then           -- If rest is enabled, it will wait one slow
                    if rest_tog = '0' then      -- tick cycle to exit the current rest state
                        rest_over <= '0';
                        rest_tog <= '1';
                    else
                        rest_over <= '1';
                        rest_tog <= '0';
                    end if;
                else rest_over <= '0';
                end if;
            end if;
        end if;
    end process;
end Behavioral;
```

```
----------------------------------------------------------------------------------
-- Company: ENGS 31 16X
-- Engineers: John Sullivan and Jennifer Jain
--
-- Create Date: 08/12/2016 02:39:50 PM
-- Design Name: Monopulser
-- Module Name: monopulser - Behavioral
-- Project Name: Simon Game
-- Target Devices: Digilent Basys 3 board (Artix 7)
-- Tool versions: Vivado 2016.1
-- Description: Takes a button input and asserts it for one clock cycle
--              as opposed to the full duration of the button press
--
-- Dependencies: Requires no extra files
--
-- Revision:
-- Revision 0.01 - File created
-- Additional Comments:
-- Monopulser has a one clock cycle delay from the input signal that it receives
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity monopulser is
    port (
        clk, mono_in : in std_logic;
        mono_out : out std_logic
    );
end monopulser;

architecture Behavioral of monopulser is
    type state_type_m is (Idle, Mono, Hold);   -- States
    signal cs_m, ns_m : state_type_m;          -- current state and next state
begin
    -- Process to update the state on new rising clock edges:
    state_update : process(clk)
    begin
        if rising_edge(clk) then
            cs_m <= ns_m;
        end if;
    end process state_update;

    -- Combinational logic for the controller to determine transitions between states and
       outputs from the states:
    comb_logic : process(cs_m, mono_in)
    begin
        ns_m <= cs_m;
        mono_out <= '0';

        case cs_m is
            when Idle =>                -- Wait for input
                if mono_in = '1' then
                    ns_m <= Mono;
                end if;
```

```
            when Mono =>                -- Assert for one clock cycle and
                mono_out <= '1';        -- then go immediately to hold

                ns_m <= Hold;
            when Hold =>                -- Stay in hold until button released
                if mono_in = '0' then
                    ns_m <= Idle;
                end if;
            when others => null;
        end case;
    end process comb_logic;

end Behavioral;
```

```
----------------------------------------------------------------------------------
-- Company: ENGS 31 16X
-- Engineers: John Sullivan and Jennifer Jain
--
-- Create Date: 08/20/2016 12:46:47 PM
-- Design Name: Soundbox
-- Module Name: soundbox - Behavioral
-- Project Name: Simon Game
-- Target Devices: Digilent Basys 3 board (Artix 7)
-- Tool versions: Vivado 2016.1
-- Description: Soundbox to generate square wave for desired sound frequencies
--
-- Dependencies: Requires no extra files
--
-- Revision:
-- Revision 0.01 - File created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
use ieee.math_real.all;

entity soundbox is
    generic (output_freq : integer := 1E3);
    port (
        clk : in std_logic;
        sound_out : out std_logic
    );
end soundbox;

architecture Behavioral of soundbox is
    -- Constants (taken from lab 3):
    constant CLOCK_FREQ : integer := 100E6;
    constant CLK_DIV_VAL : integer := CLOCK_FREQ / (2 * OUTPUT_FREQ);
    constant COUNT_LEN : integer := integer(ceil(log2(real(CLK_DIV_VAL - 1))));

    -- Signals (taken from lab 3):
    signal clkdiv : unsigned(COUNT_LEN - 1 downto 0) := (others => '0'); -- the clock divider
        counter
    signal sound : std_logic := '0';
begin
    -- Clock divider (taken from lab 3) - Based on a counter, a square wave is created for
    --                                    for the desired frequency passed to this module
    Clock_divider : process(clk)
    begin
        if rising_edge(clk) then
            if clkdiv = CLK_DIV_VAL - 1 then     -- When reached desired count,
                sound <= not(sound);             -- the sound signal is inverted
                clkdiv <= (others => '0');
            else
                clkdiv <= clkdiv + 1;            -- Counting
            end if;
```

```
        end if;
    end process Clock_divider;

    sound_out <= sound;
end Behavioral;
```

Appendix F: VHDL Code 5
MAX6816 (Debouncer)

```
----------------------------------------------------------------------------------
-- Company: Engs 31 / CoSc 56
-- Engineer: E.W. Hansen
--
-- Create Date: 15:06:54 07/25/2015
-- Design Name: Lab 4
-- Module Name: MAX6816 - Behavioral
-- Project Name:
-- Target Devices: Spartan 3E, Spartan 6
-- Tool versions: ISE 14.7
-- Description: VHDL model of the Maxim Integrated MAX6816 switch debouncer
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - file Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity MAX6816 is
    port (
        clk : in STD_LOGIC;
        switch : in STD_LOGIC; -- low-true switch input
        switch_db : out STD_LOGIC
    );
end MAX6816;

architecture Behavioral of MAX6816 is
    constant CLOCK_FREQUENCY : integer := 100E6; -- Basys 3 master clock
    constant DEBOUNCE_DELAY_MS : integer := 100; -- 10 msec debounce delay
    constant MAX_COUNT : integer := (CLOCK_FREQUENCY * DEBOUNCE_DELAY_MS) / 1E9;
    signal db_counter : integer := MAX_COUNT - 1;
    signal db_reg : std_logic := '0';
begin
    debouncer : process(clk, db_reg)
    begin
        if rising_edge(clk) then
            if switch = db_reg then
                db_counter <= MAX_COUNT - 1;
            else
                db_counter <= db_counter - 1;
            end if;

            if db_counter = 0 then
                db_reg <= switch;
            end if;
        end if;

        switch_db <= db_reg;
    end process debouncer;
```

```
end Behavioral;
```

```
----------------------------------------------------------------------------------
-- Company: Engs 31 16X
-- Engineer: E.W. Hansen
--
-- Create Date: 17:56:35 07/25/2008
-- Design Name:
-- Module Name: mux7seg - Behavioral
-- Project Name:
-- Target Devices: Digilent Basys 3 board (Artix 7)
-- Tool versions: Vivado 2016.1
-- Description: Multiplexed seven-segment decoder for the display on the Basys3
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - file Created
-- Revision 1.00 (07/17/2015) --- drop the clock divider, run on a 1000 Hz clock
-- Revision 2.00 (07/17/2016) --- put the clock divider back in
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;

entity mux7seg is
    port (
        clk : in STD_LOGIC; -- runs on a fast (100 MHz or so) clock
        y0, y1, y2, y3 : in STD_LOGIC_VECTOR (3 downto 0); -- digits
        dp_set : in std_logic_vector(3 downto 0); -- decimal points
        seg : out STD_LOGIC_VECTOR(0 to 6); -- segments (a...g)
        dp : out std_logic;
    an : out STD_LOGIC_VECTOR (3 downto 0) ); -- anodes
end mux7seg;

architecture Behavioral of mux7seg is
    constant NCLKDIV : integer := 18; -- 100 MHz / 2^18 = 381 Hz
    constant MAXCLKDIV : integer := 2 ** NCLKDIV - 1; -- max count of clock divider
    signal cdcount : unsigned(NCLKDIV - 1 downto 0); -- clock divider counter register
    signal CE : std_logic; -- clock enable

    signal adcount : unsigned(1 downto 0) := "00"; -- anode / mux selector count
    signal muxy : std_logic_vector(3 downto 0); -- mux output
    signal segh : std_logic_vector(0 to 6); -- segments (high true)

begin
    -- Clock divider sets the rate at which the display hops from one digit to the next. A
       larger value of
    -- MAXCLKDIV results in a slower clock-enable (CE)
    ClockDivider :
    process (clk)
begin
    if rising_edge(clk) then
        if cdcount < MAXCLKDIV then
            CE <= '0';
```

```
            cdcount <= cdcount + 1;
        else CE <= '1';
            cdcount <= (others => '0');
        end if;
    end if;
end process ClockDivider;

AnodeDriver :
process (clk, adcount)
begin
    if rising_edge(clk) then
        if CE = '1' then
            adcount <= adcount + 1;
        end if;
    end if;

    case adcount is
        when "00" => an <= "1110";
        when "01" => an <= "1101";
        when "10" => an <= "1011";
        when "11" => an <= "0111";
        when others => an <= "1111";
    end case;
end process AnodeDriver;

Multiplexer :
process (adcount, y0, y1, y2, y3, dp_set)
begin
    case adcount is
        when "00" => muxy <= y0;
            dp <= not(dp_set(0));
        when "01" => muxy <= y1;
            dp <= not(dp_set(1));
        when "10" => muxy <= y2;
            dp <= not(dp_set(2));
        when "11" => muxy <= y3;
            dp <= not(dp_set(3));
        when others => muxy <= x"0";
            dp <= '1';
    end case;
end process Multiplexer;

-- Seven segment decoder
with muxy select segh <=
    "1111110" when x"0", -- active-high definitions
    "0110000" when x"1",
    "1101101" when x"2",
    "1111001" when x"3",
    "0110011" when x"4",
    "1011011" when x"5",
    "1011111" when x"6",
    "1110000" when x"7",
    "1111111" when x"8",
    "1111011" when x"9",
```

```
    "1110111" when x"a",
    "0011111" when x"b",
    "1001110" when x"c",
    "0111101" when x"d",
    "1001111" when x"e",
    "1000111" when x"f",
    "0000000" when others;
seg <= not(segh); -- Convert to active-low

end Behavioral;
```

Appendix F: VHDL Code 7
Simon Game Top Level Testbench

```
----------------------------------------------------------------------------------
-- Company: ENGS 31 16X
-- Engineers: John Sullivan and Jennifer Jain
--
-- Create Date: 08/10/2016 09:12:40 PM
-- Design Name: Simon Game Top Testbench
-- Module Name: simon_game_top_tb - Behavioral
-- Project Name: Simon Game
-- Target Devices: Digilent Basys 3 board (Artix 7)
-- Tool versions: Vivado 2016.1
-- Description: Testbench to test the top-level file for Simon Game project
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - file Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity simon_game_top_tb is
end simon_game_top_tb;

architecture Behavioral of simon_game_top_tb is

    component simon_game is
        port (
            -- INPUTS:
            btn : in std_logic_vector(3 downto 0);
            start, clr : in std_logic;
            clk : in std_logic;

            -- OUTPUTS:
            lights_out : out std_logic_vector(3 downto 0);
            final_sound : out std_logic;

            -- Multiplexed seven segment display:
            segments : out std_logic_vector(0 to 6);
            dp : out std_logic;
            anodes : out std_logic_vector(3 downto 0);

            -- Forced high outputs for audio:
            amp_shut, gain : out std_logic
        );
    end component;

    --Inputs:
    signal clk100 : std_logic := '0';
    signal btn : std_logic_vector(3 downto 0) := "0000";
    signal start, clr : std_logic := '0';
```

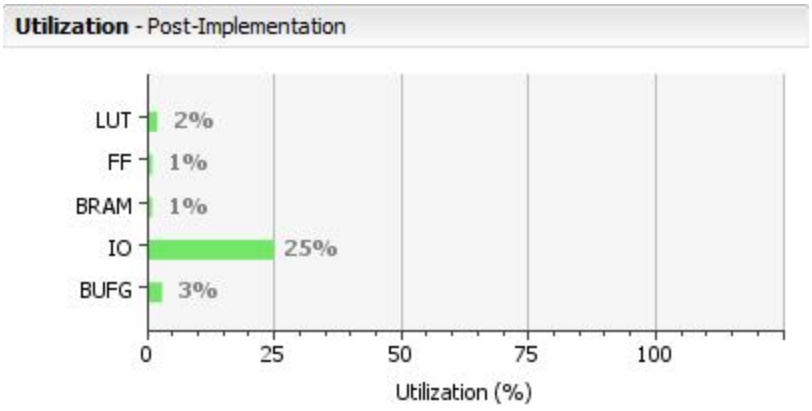
```
    -- Outputs:
    signal lights_out : std_logic_vector(3 downto 0);
    signal final_sound : std_logic;
    signal segments : std_logic_vector(0 to 6);
    signal dp : std_logic;
    signal anodes : std_logic_vector(3 downto 0);

    -- Clock period definitions (taken from lab 3):
    constant clk100_period : time := 10 ns;
    constant slow_clk_period : time := 20 * clk100_period;

begin
    -- Unit Under Test port map:
    uut : simon_game
    port map(
    clk => clk100,
    btn => btn,
    start => start,
    clr => clr,
    lights_out => lights_out,
    final_sound => final_sound,
    segments => segments,
    dp => dp,
    anodes => anodes
    );

    -- Clock process definitions (taken from lab 3)
    clk100_process : process
    begin
        clk100 <= '0';
        wait for clk100_period/2;
        clk100 <= '1';
        wait for clk100_period/2;
    end process;

    stim_proc : process
    begin
        wait for 20 ns + clk100_period/2;

        -- Start game

        start <= '1';     -- Start button pressed
        wait for 140 ns;
        start <= '0';     -- Start released
        wait for 500 ns;

        -- First run

        btn <= "0100";    -- Button 3 pressed (NOTE: Button numbers stated here go from least
        wait for 140 ns;  --                         siginificant bit to most significant bit)
        btn <= "0000";    -- Button 3 released
        wait for 1000 ns;

        -- Second run
```

```
        btn <= "0100";    -- Button 3 pressed
        wait for 140 ns;
        btn <= "0000";    -- Button 3 released
        wait for 140 ns;

        btn <= "0001";    -- Button 1 pressed
        wait for 140 ns;
        btn <= "0000";    -- Button 1 released
        wait for 1100 ns;

        -- Third run

        btn <= "0100";    -- Button 3 pressed
        wait for 140 ns;
        btn <= "0000";    -- Button 3 released
        wait for 140 ns;

        btn <= "0001";    -- Button 1 pressed
        wait for 140 ns;
        btn <= "0000";    -- Button 1 released
        wait for 140 ns;

        btn <= "0100";    -- Button 3 pressed
        wait for 140 ns;
        btn <= "0000";    -- Button 3 released
        wait for 1200 ns;

        -- Losing run

        btn <= "0100";    -- Button 3 pressed
        wait for 140 ns;
        btn <= "0000";    -- Button 3 released
        wait for 140 ns;

        btn <= "1000";    -- Button 4 pressed (incorrect)
        wait for 300 ns;
        btn <= "0000";    -- Button 4 released
        wait for 2600 ns;

        -- Clear high-score

        clr <= '1';
        wait for 140 ns;
        clr <= '0';

        wait for 10000 ns;
    end process;

end Behavioral;
```

```
# John Sullivan and Jennifer Jain
# Professor Hansen
# ENGS 31 16X
#
# XDC File for Simon Game Project

## Clock signal
set_property PACKAGE_PIN W5 [get_ports clk]
    set_property IOSTANDARD LVCMOS33 [get_ports clk]
    create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]


##7 segment display
set_property PACKAGE_PIN W7 [get_ports {segments[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {segments[0]}]
set_property PACKAGE_PIN W6 [get_ports {segments[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {segments[1]}]
set_property PACKAGE_PIN U8 [get_ports {segments[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {segments[2]}]
set_property PACKAGE_PIN V8 [get_ports {segments[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {segments[3]}]
set_property PACKAGE_PIN U5 [get_ports {segments[4]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {segments[4]}]
set_property PACKAGE_PIN V5 [get_ports {segments[5]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {segments[5]}]
set_property PACKAGE_PIN U7 [get_ports {segments[6]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {segments[6]}]

set_property PACKAGE_PIN V7 [get_ports dp]
    set_property IOSTANDARD LVCMOS33 [get_ports dp]

set_property PACKAGE_PIN U2 [get_ports {anodes[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {anodes[0]}]
set_property PACKAGE_PIN U4 [get_ports {anodes[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {anodes[1]}]
set_property PACKAGE_PIN V4 [get_ports {anodes[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {anodes[2]}]
set_property PACKAGE_PIN W4 [get_ports {anodes[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {anodes[3]}]


##Buttons
set_property PACKAGE_PIN U18 [get_ports start]
    set_property IOSTANDARD LVCMOS33 [get_ports start]
set_property PACKAGE_PIN T17 [get_ports clr]
    set_property IOSTANDARD LVCMOS33 [get_ports clr]


##Pmod Header JA
##Sch name = JA1
set_property PACKAGE_PIN J1 [get_ports {final_sound}]
    set_property IOSTANDARD LVCMOS33 [get_ports {final_sound}]
##Sch name = JA2
set_property PACKAGE_PIN L2 [get_ports {gain}]
```

```
    set_property IOSTANDARD LVCMOS33 [get_ports {gain}]
##Sch name = JA3
#set_property PACKAGE_PIN J2 [get_ports {JA[2]}]
    #set_property IOSTANDARD LVCMOS33 [get_ports {JA[2]}]
#Sch name = JA4
set_property PACKAGE_PIN G2 [get_ports {amp_shut}]
    set_property IOSTANDARD LVCMOS33 [get_ports {amp_shut}]


##Pmod Header JC
##Sch name = JC1
set_property PACKAGE_PIN K17 [get_ports {lights_out[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {lights_out[0]}]
##Sch name = JC2
set_property PACKAGE_PIN M18 [get_ports {lights_out[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {lights_out[1]}]
##Sch name = JC3
set_property PACKAGE_PIN N17 [get_ports {lights_out[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {lights_out[2]}]
##Sch name = JC4
set_property PACKAGE_PIN P18 [get_ports {lights_out[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {lights_out[3]}]


##Pmod Header JXADC
##Sch name = XA1_P
set_property PACKAGE_PIN J3 [get_ports {btn[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {btn[2]}]
#Sch name = XA2_P
set_property PACKAGE_PIN L3 [get_ports {btn[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {btn[3]}]
#Sch name = XA3_P
set_property PACKAGE_PIN M2 [get_ports {btn[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {btn[1]}]
#Sch name = XA4_P
set_property PACKAGE_PIN N2 [get_ports {btn[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {btn[0]}]


## These additional constraints are recommended by Digilent
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
set_property BITSTREAM.CONFIG.SPI_BUSWIDTH 4 [current_design]
set_property CONFIG_MODE SPIx4 [current_design]

set_property BITSTREAM.CONFIG.CONFIGRATE 33 [current_design]

set_property CONFIG_VOLTAGE 3.3 [current_design]
set_property CFGBVS VCCO [current_design]
```

Appendix G: Resource Utilization

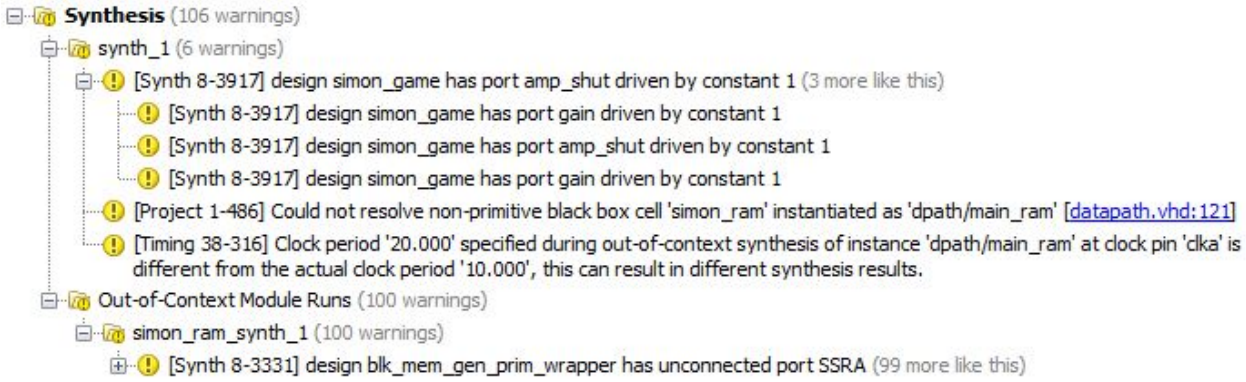


Resource	Utilization	Available	Utilization %
LUT	436	20800	2.096154
FF	393	41600	0.944712
BRAM	0.5	50	1
IO	26	106	24.5283
BUFG	1	32	3.125

Appendix H: Critical Timing

Worst Negative Slack (WNS): 3.98 ns

Appendix I: Analysis of Residual Warnings



We were able to fine-tune our design until we had no warnings of concern according to the Professor. We intentionally drove the amp_shut and gain ports with constants of 1, in order to interact properly with the PmodAMP2. Driving amp_shut high keeps the audio output on and driving gain high applies a 6 dB to the audio signal, rather than a 12 dB gain. According to the Professor, the black box warning regarding the RAM used in our Simon Game is generated at one point in the synthesis of our design and is fixed later. Thus, that warning is of no concern. According to the Professor, the [Timing 38-316] warning and the warnings generated by the out of context module run of the Simon RAM are also of no concern. We are also not concerned about the warnings from the out-of-context module runs since the ports have been intentionally unconnected because we do not use them.

Appendix J: Memory Map

RAM Memory Map			
# of locations	256		
# of bits per location	2		
Ports	**# of bits**	**Mapped To**	**Type: In/Out**
clka	1	clk	In
ena	1	1	In
wea	1	1	In
addra	8	curr_addr	In
dina	2	rand_out	In
clkb	1	clk	In
addrb	8	mux1_out	In
doutb	2	rd_data	Out
enb	always enabled		In

Appendix K: Legend of Signals

clk100 - The master clock for the device.

slow_tick - Enable bit used for processes that should not function at the high speed clock, such as the cycle counter and the lose sequence.

btn[3:0] - These are the main button inputs to the device.

btn_m[3:0] - The monopulsed version of the button input. It is monopulsed after the button has been debounced.

btn_press - Signal that becomes true just as a button is pressed. It is used to signal the transition into the *Check* state.

start - A button that signals to start the game. It is debounced and monopulsed before it is used.

clr - A button that signals to clear the high score when in the idle state. It is debounced and monopulsed before it is used.

lights_out[3:0] - Light outputs from the device. They go directly to the LEDs, which are low true.

final_sound - The sound that is emitted from the device when a sound is designated to play. Since sound frequencies have large frequencies, not much can be seen in the simulation.

segments[0:6] - Outputs to the segments of the seven segment display.

dp - Indicates where to place the decimal point on the seven segment display.

anodes[3:0] - Designates which of the lights should be turned on at a given point in time.

clk100period - The period of the master clock.

slow_clk_period - The period of the slow clock.

c_s - Indicates the current state of the device.

rand_en - Enables a random number to be stored as rand_out.

rand_num[1:0] - The value in a 2-bit counter.

rand_out[1:0] - Random number that is stored when rand_en is set.

curr_addr[7:0] - Current address for writing to the RAM. It is incremented in the *Incr_Addr* state, just after a random number has been generated.

cycle_addr[7:0] - The address to be read from the RAM to display with the lights and sounds.

tc_w_addr - The terminal count when the cycle address has incremented through all of the addresses with data for the current game in the RAM.

press_count[7:0] - The number of button presses for a specific sequence that the user has entered.

pc_tc - The terminal count when the press count has incremented through all of the addresses with data for the current game in the RAM.

mux1_out[7:0] - Selects which address should be read from the RAM using either the cycle count or the press count.

curr_btn[1:0] - Stores whichever button is pressed by the user.

rd_data[1:0] - This is the data output from the RAM. It is the same as *doutb* from the RAM module.

correct - Indicates if the button the user pressed (*curr_btn*) matches the data that is being read from the RAM (*rd_data*).

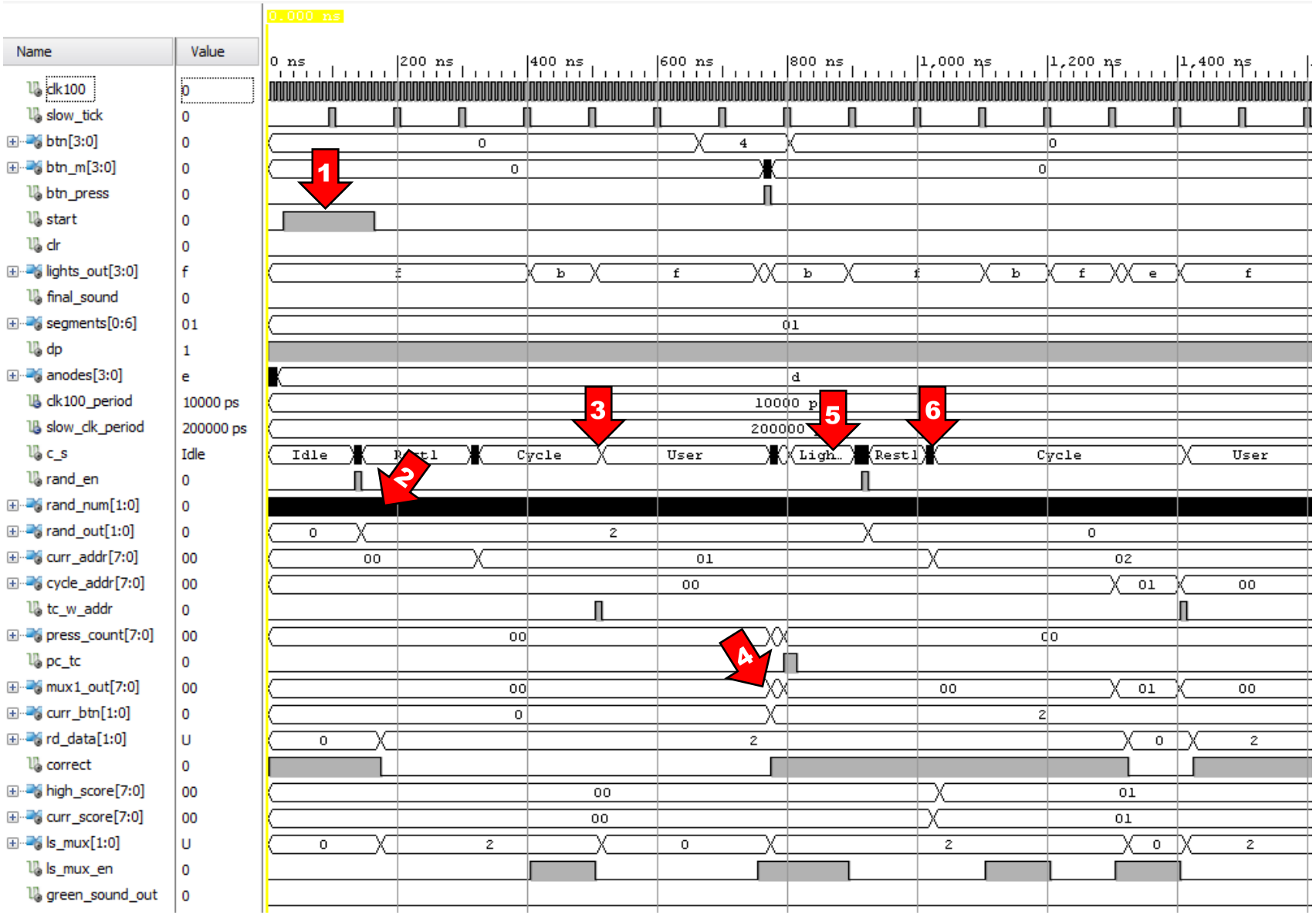
high_score[7:0] - The high score for the game (reset using the *clr* button).

curr_score[7:0] - The current score for the game (reset after each game and when game lost).

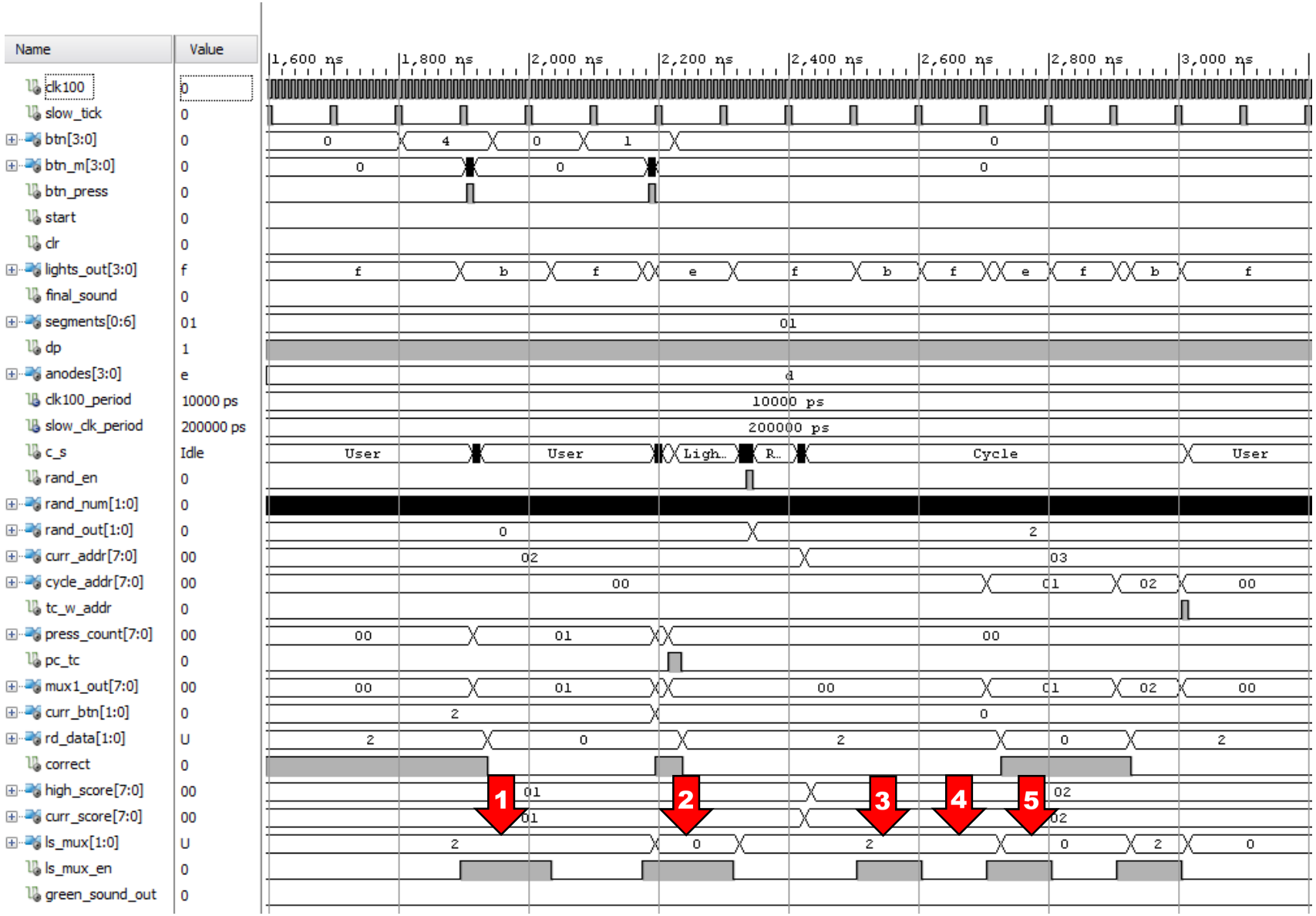
ls_mux[1:0] - Designates which lights and sounds should be outputted at a given point in the game (does not designate whether they should be turned or not).

ls_mux_en - Designates whether the lights and sound output by *ls_mux* should be displayed to the user.

green_sound_out - An example of a sound output signal. Since sound frequencies have large frequencies, not much can be seen in the simulation.

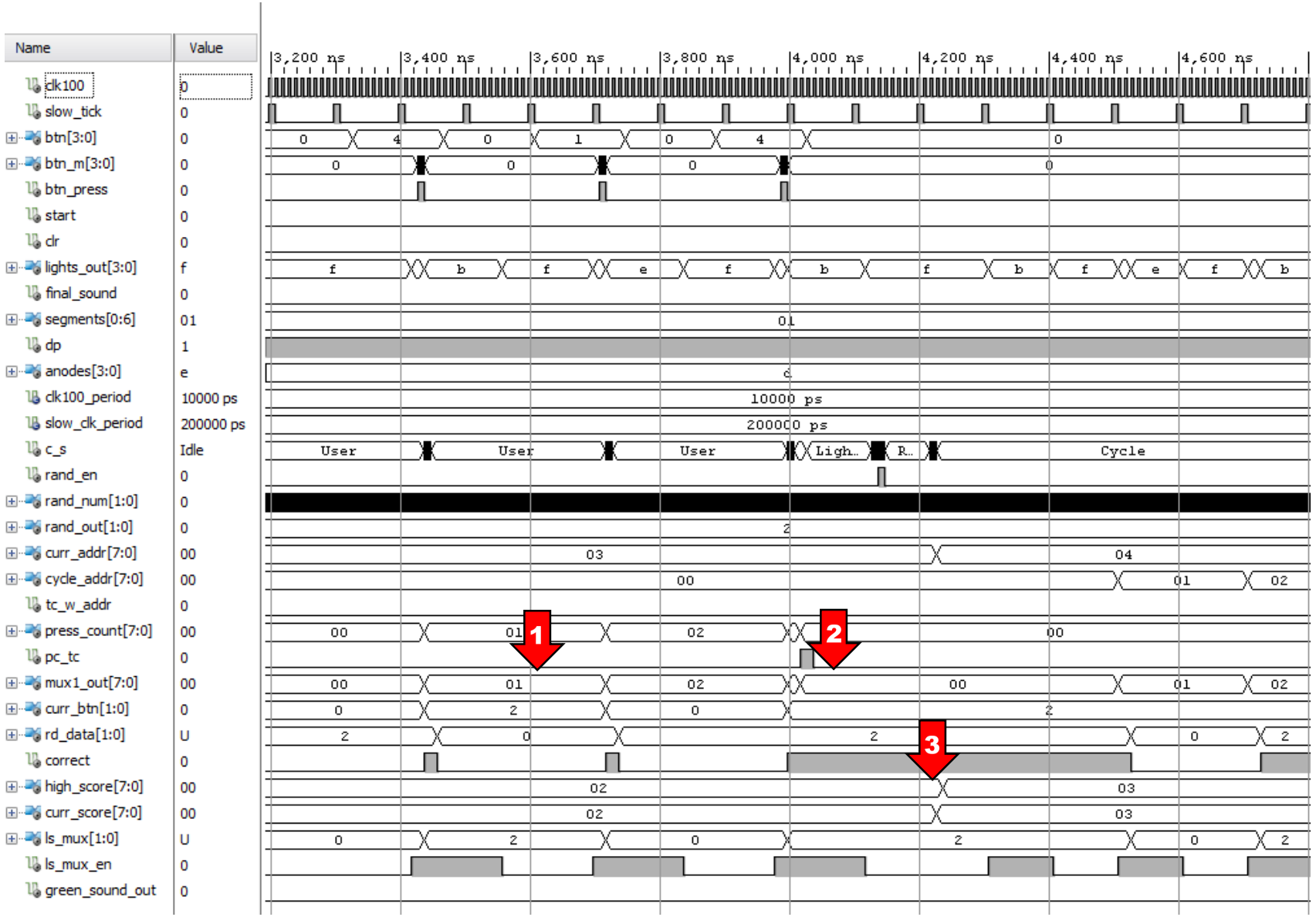


1. Start button pressed.
2. The random number generator is constantly generating numbers. When start button pressed or the sequence is properly entered, a new random number is generated.
3. The device cycles through the different lights and the user attempts to enter the correct button in the sequence.
4. The user enters the button (*curr_btn*) and the device checks that against the value in the RAM (*rd_data*). If they matched, *correct* is set high and the next state is entered.
5. In the *Lights_Off* state, the device waits for the user to release the button before advancing.
6. After another short rest, the next *Cycle* state is entered.

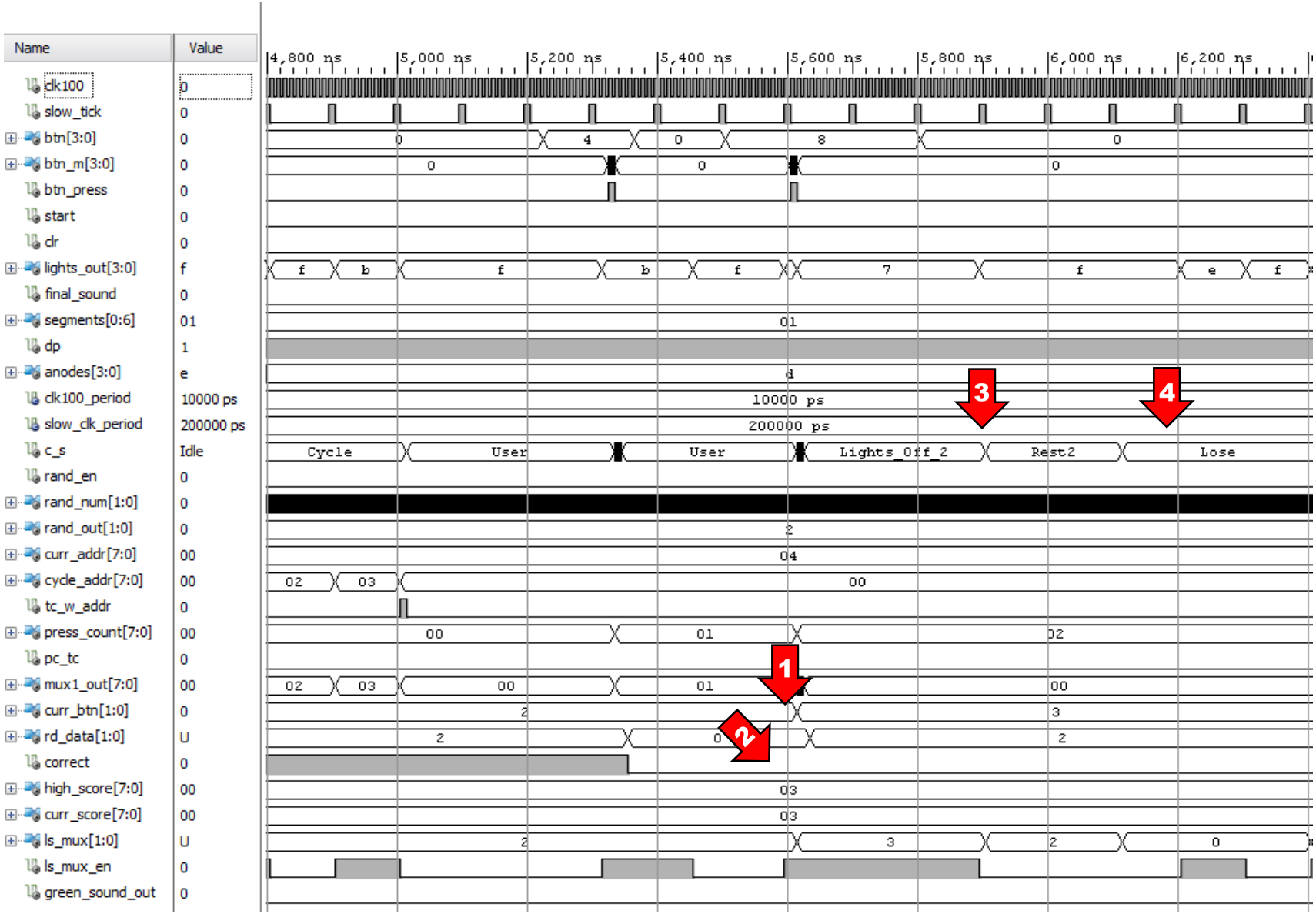


1. The light corresponding to the button that the user is pressing remains on for the duration that the user has held the button.
2. The user enters another correct button and the next sequence of lights is not displayed until this button is released.
3. This shows the process by which lights are cycled through to display to the user.
4. At this point, one can see that *ls_mux_en* is held off. This is to create a slight delay in between lights and sounds so that the user may process the changes.
5. The next button in sequence is selected and *ls_mux_en* is turned on to display the light and play the proper sound.

Appendix K: Figure 1.3 – Overall Simulation

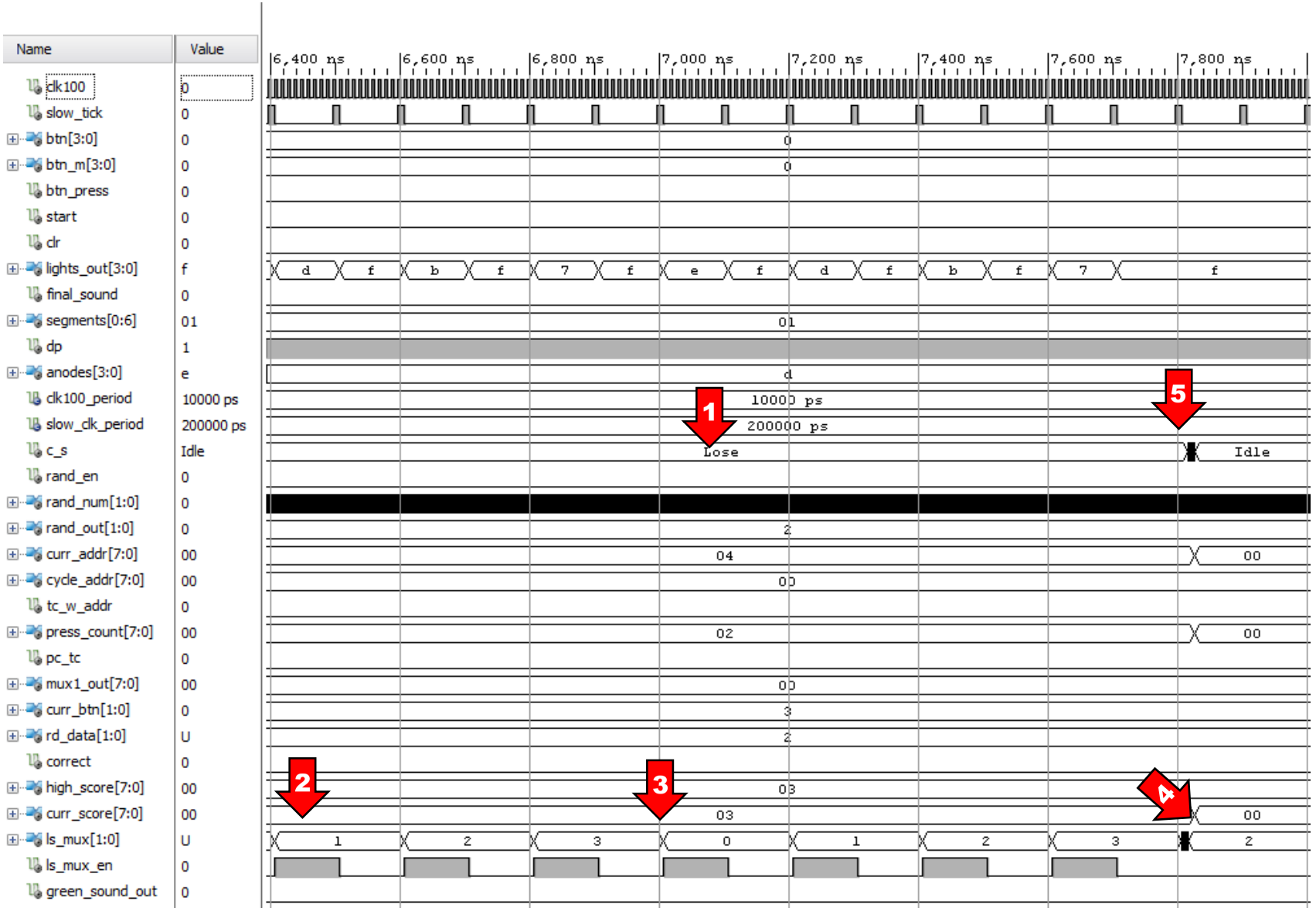


1. *mux1_out* is used to select the address which is read from the RAM
2. The mux used for *mux1_out* is choosing between *press_count* and *cycle_addr* depending on whether or not the user is entering input. Here, it switches to use *cycle_addr* to display the light sequence.
3. After the user has entered the correct sequence of buttons corresponding to the lights and just before the next sequence of lights is displayed, the current score and high score are updated. The high score is only updated if the current score is higher than the previous high score.

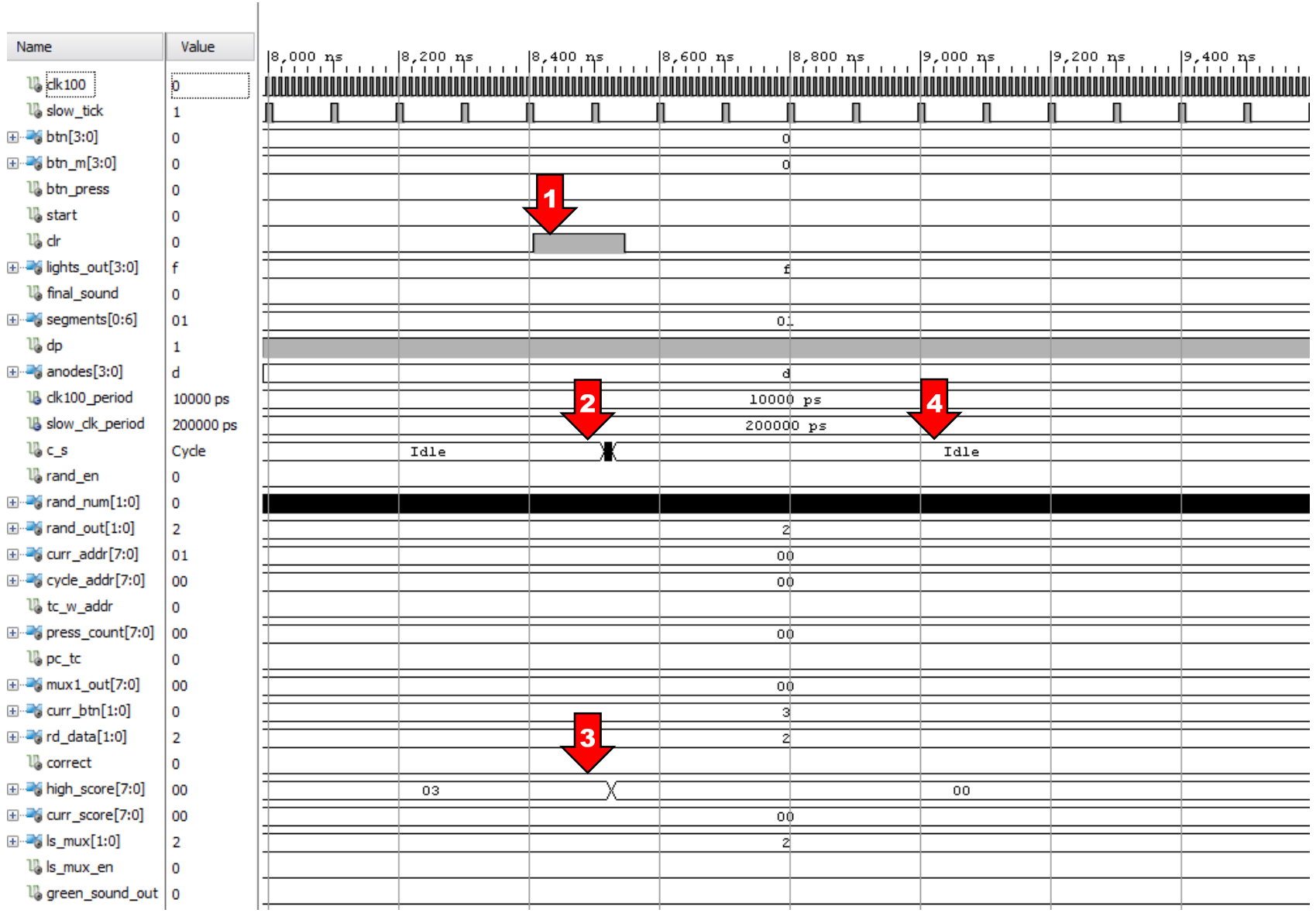


1. At this point, one can see that *curr_btn* and *rd_data* do not match.
2. The *correct* signal is not set high during the *Check* state. Since *correct* is set low, the device will initiate the lose process.
3. The device waits for the button to stop being held in the *Lights_Off_2* state.
4. In the *Lose* state, a losing sequence is displayed.

Appendix K: Figure 1.5 – Overall Simulation

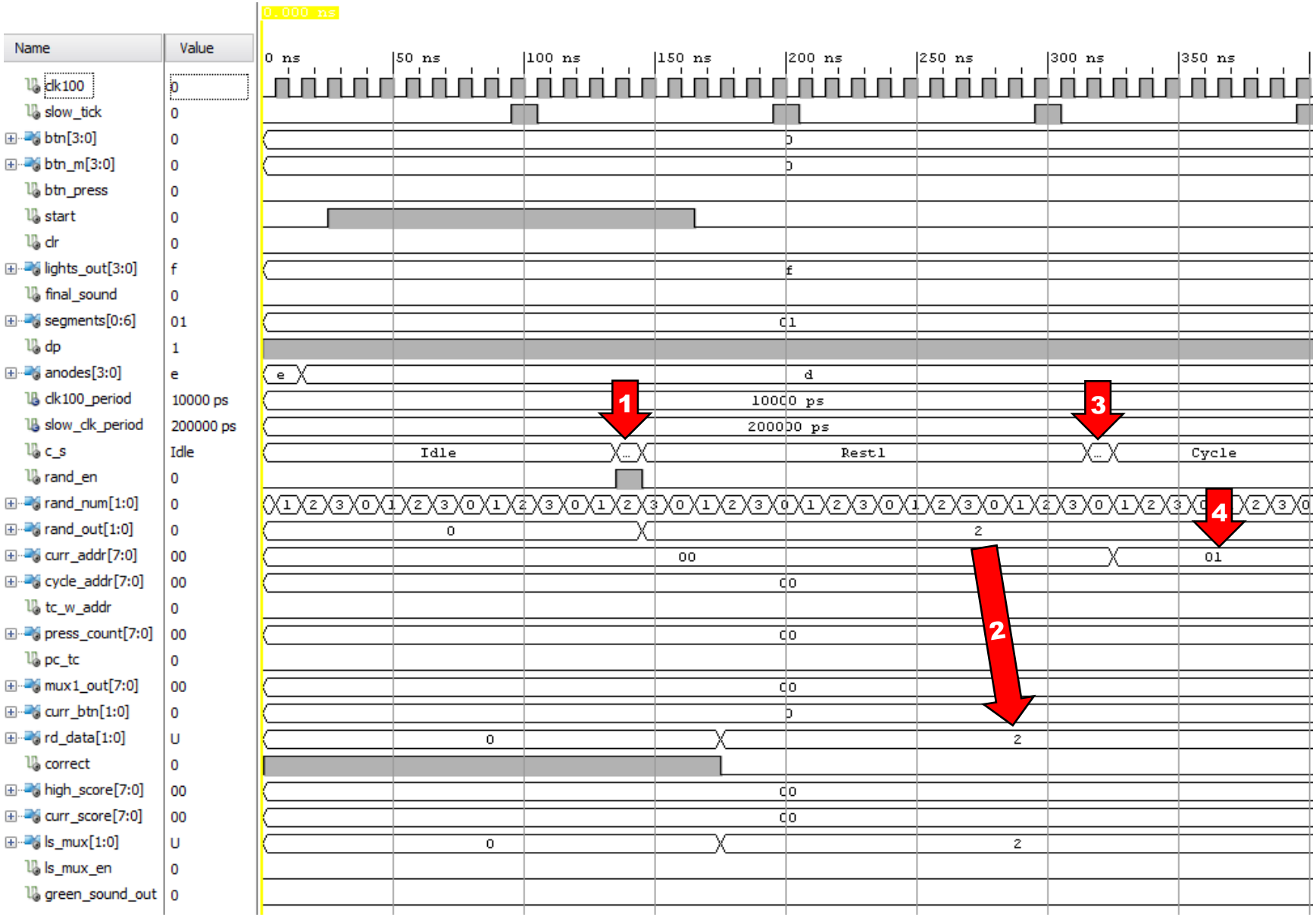


1. The device is in the *Lose* state.
2. As the device proceeds through the *Lose* state, all of the lights and sounds are cycled through twice, alternating between showing the lights and sounds and doing nothing so the user may differentiate.
3. The lose sequence enters the second cycle around the lights.
4. The *curr_score* is reset just before the device returns to the *Idle* state.
5. The device returns to the *Idle* state, awaiting either for the user to start the game again or to reset the high score.



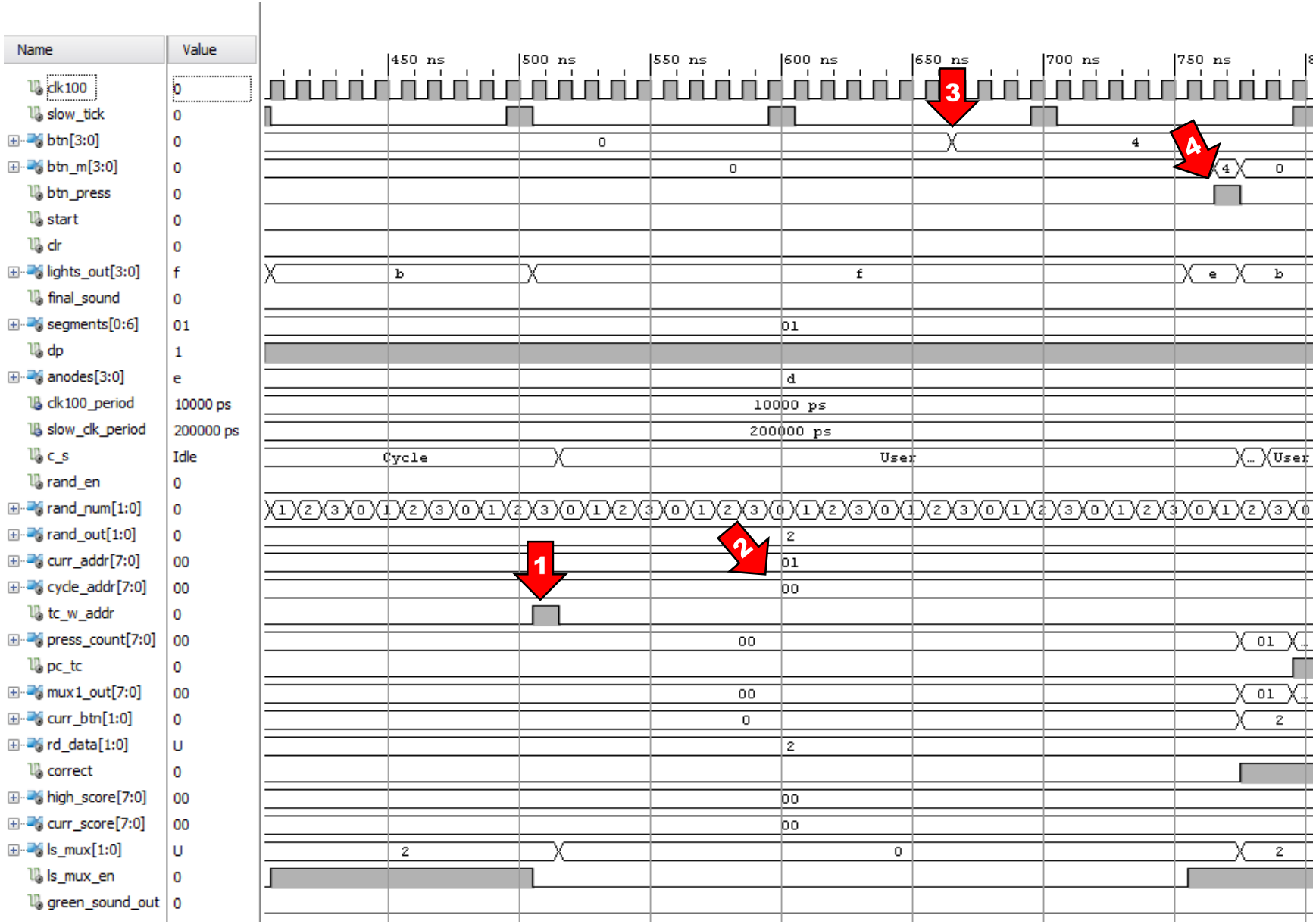
1. The user has selected to reset the high score.
2. The device enters the *Reset_HS* state.
3. The high score is reset.
4. The device returns to the *Idle* state, awaiting the next game.

Appendix K: Figure 2.1 – Magnification of Starting the Simon Game



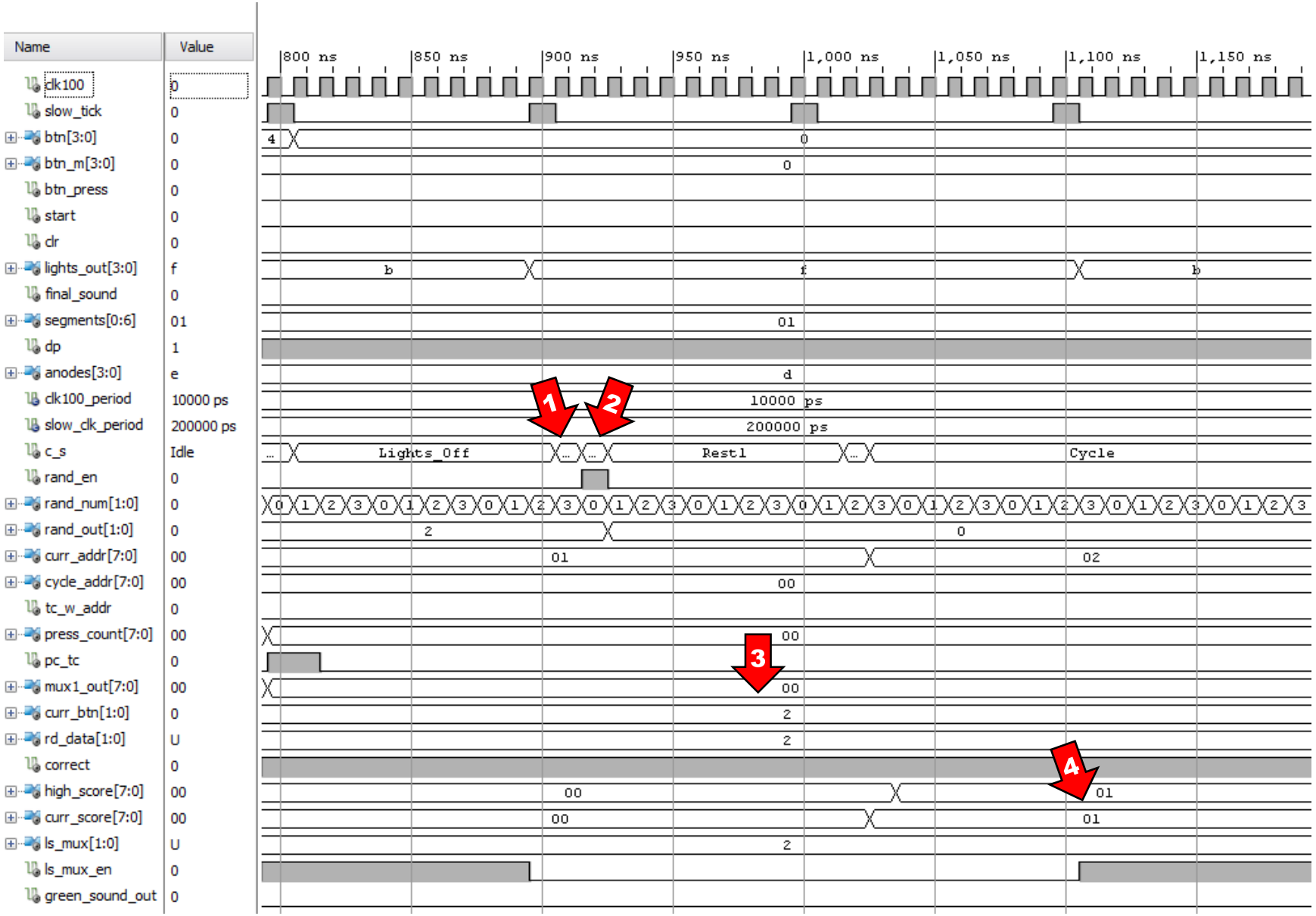
1. This is the *Generate* state. At this point, the *rand_en* is set and a random number is loaded.
2. This random number is loaded to the RAM, as shown by the *rd_data* reading out the same number after it has loaded. It takes two clock cycles to see the output that was loaded.
3. This is the state to increment the current address that will be written to.
4. The current address (*curr_addr*) has been incremented and the next time data is written to the RAM, it will be to the next address.

Appendix K: Figure 2.2 – Magnification of Starting the Simon Game

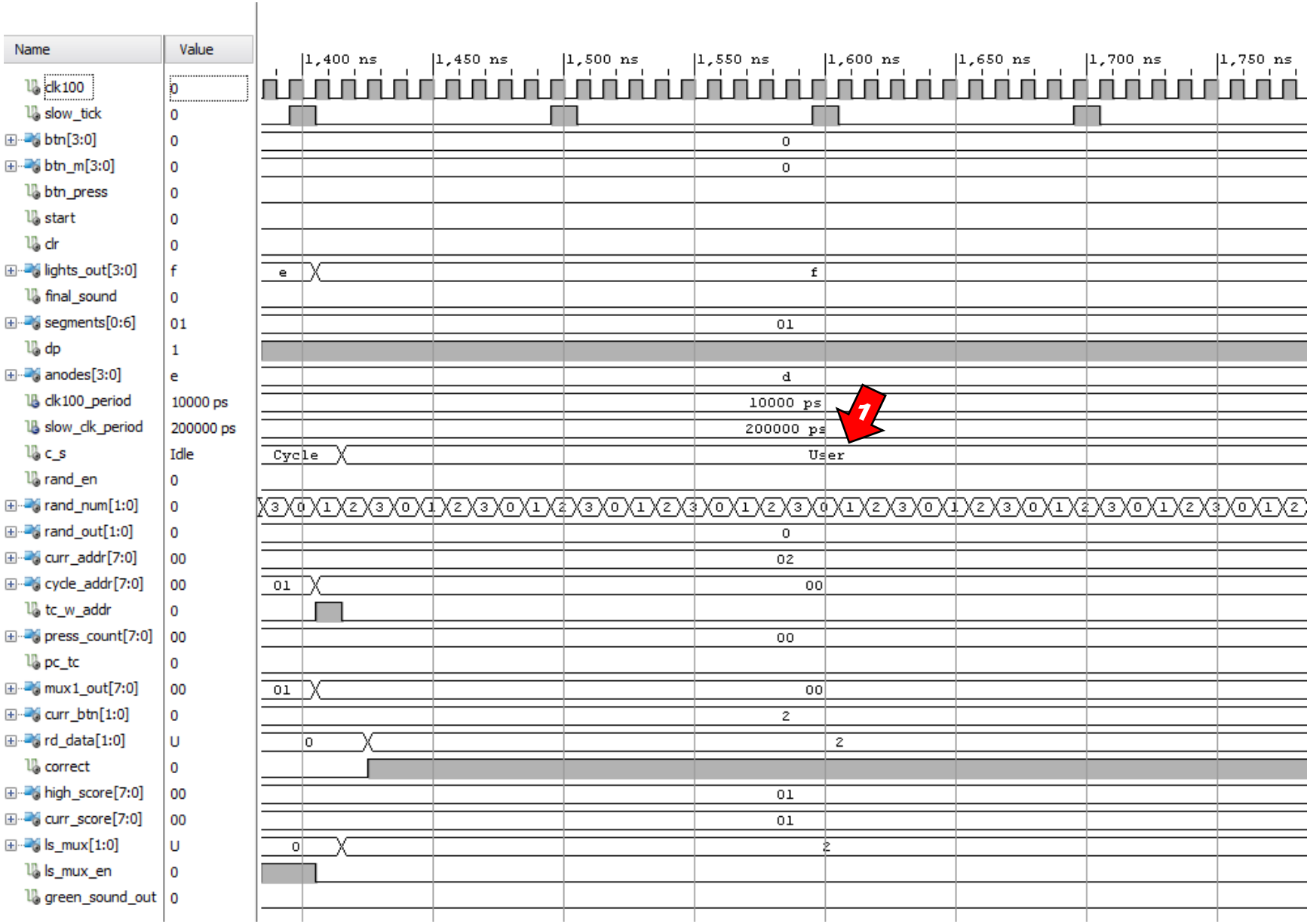


1. This signal (*tc_w_addr*) indicates that the cycle account (*cycle_addr*) has reached the current write address (*curr_addr*) minus 1. *curr_addr* is one ahead of the max, since it is ready to be written to next time.
2. The *cycle_addr* has not changed when progressing through the sequence, since there is only one light to show in the sequence.
3. The user has entered a new button.
4. After debouncing (checking that a button has been pressed), a monopulsed button signal is output.

Appendix K: Figure 2.3 – Magnification of Starting the Simon Game

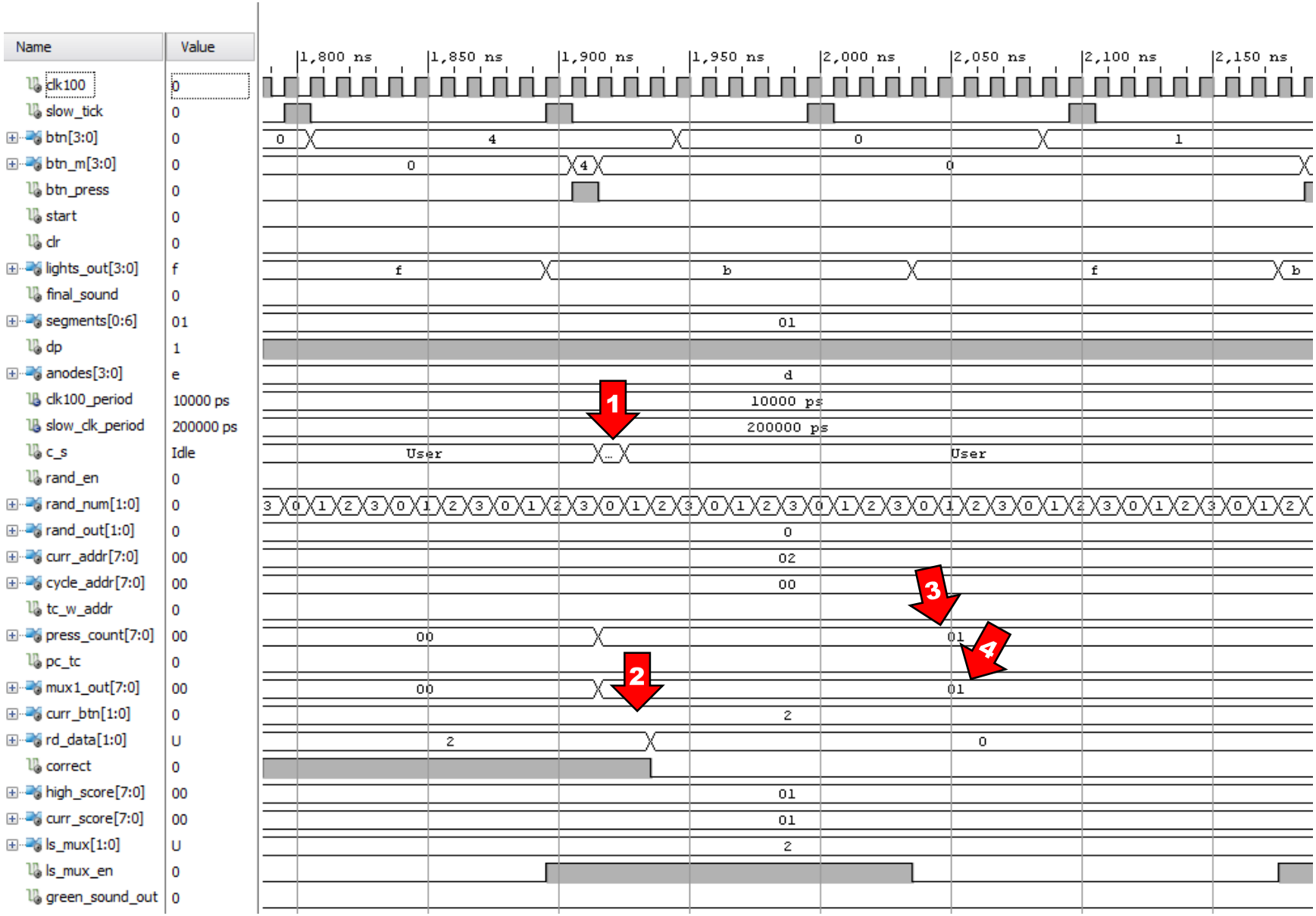


1. The cycle count and press count are reset.
2. A new random number is generated that will be added to the sequence by placing it in the next address indicated by *curr_addr*.
3. Since the cycle count and press count were not incremented (because only one address was being checked for the first run of the game), cycle count and press count do not change.
4. The current and high scores are incremented, since the user has completed one successful round of the game.



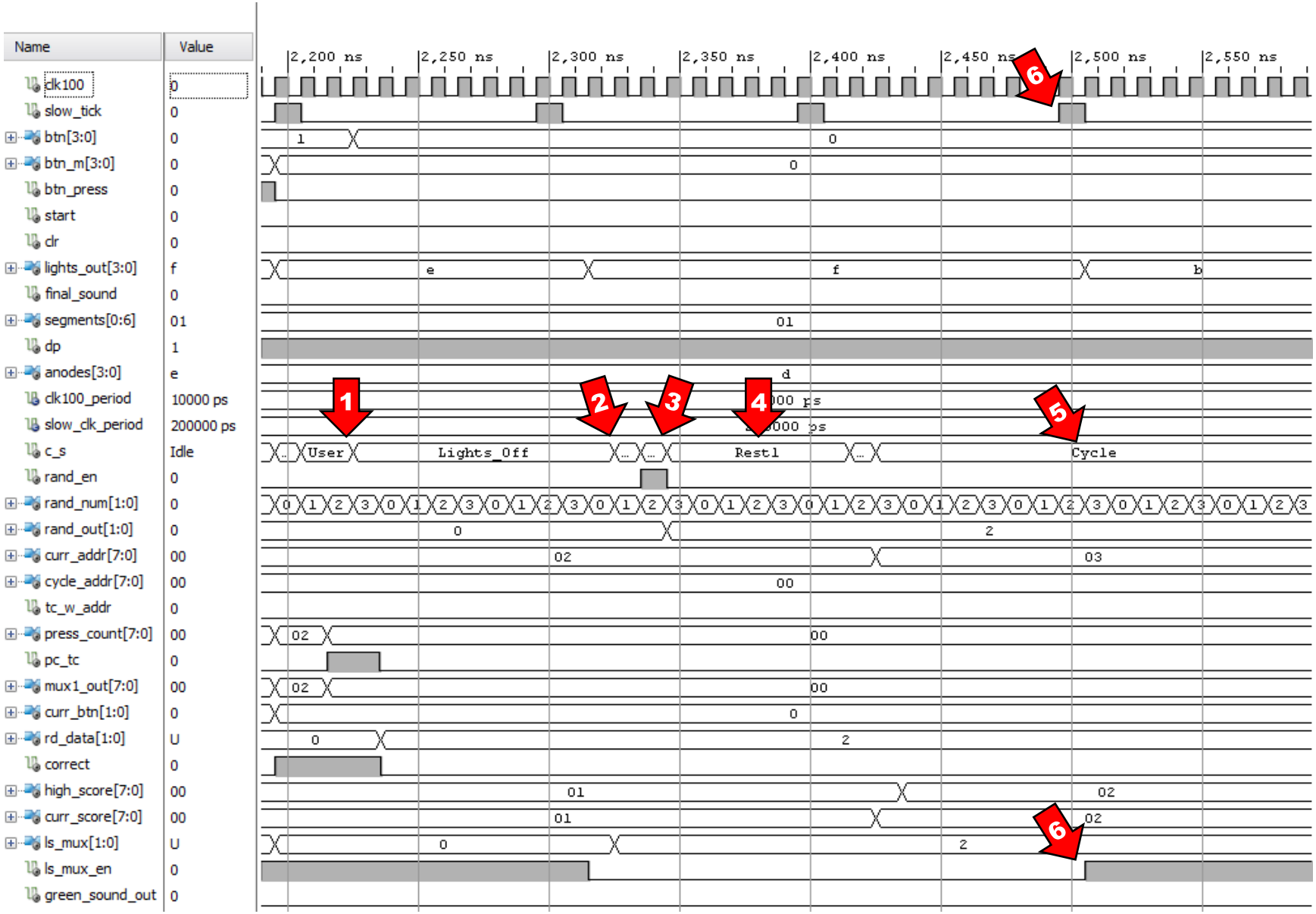
1. Waiting for a button press input from the user.

Appendix K: Figure 3.2 – Magnification of Middle of the Simon Game



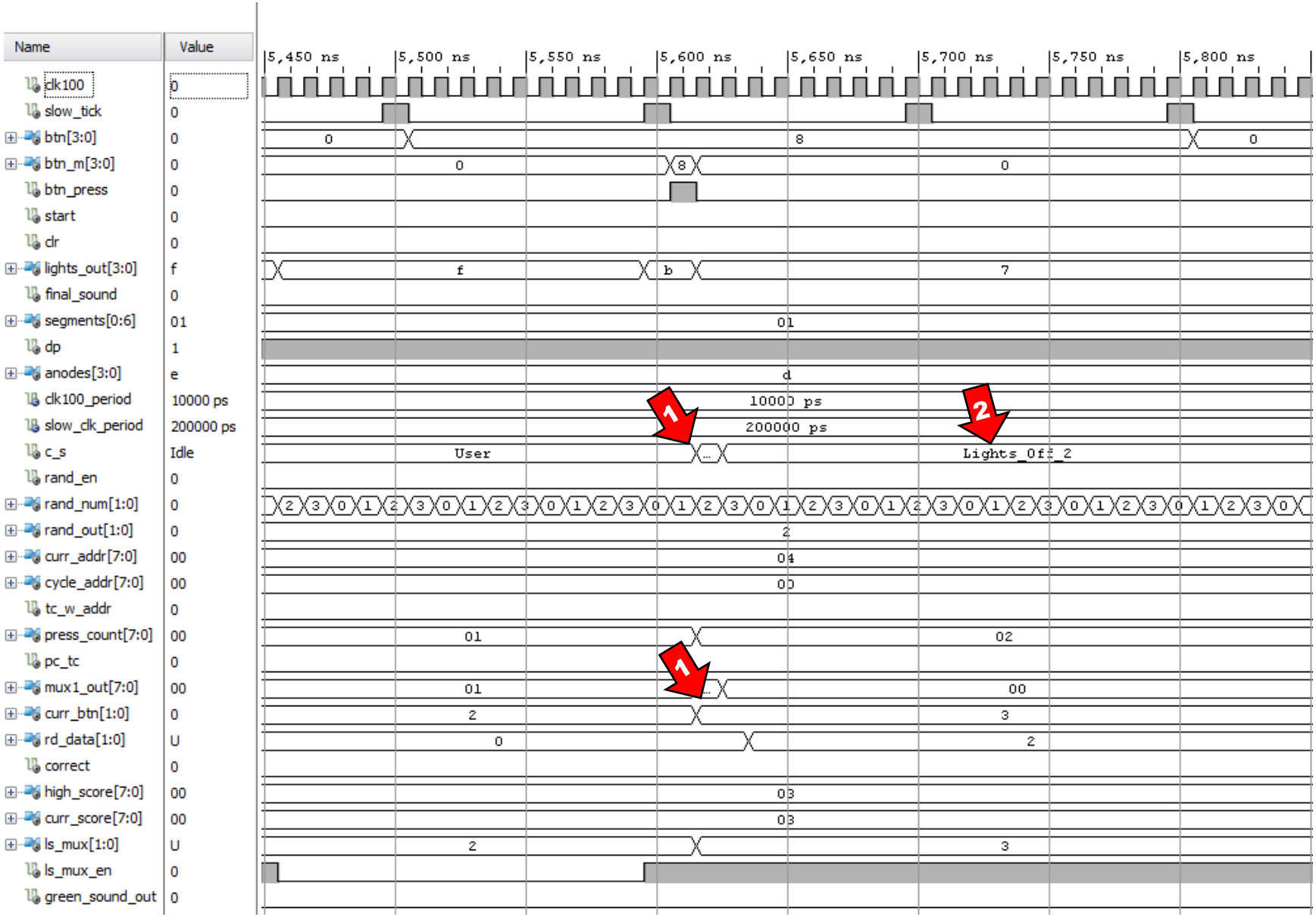
1. This is the *Check* state.
2. If *curr_btn* and *rd_data* match in the *Check* state, correct is set high. If *correct* is read as high in the *Check* state, the *User* state is returned to and further input from the user is awaited.
3. Since the user entered the *correct* button, the press count is also incremented.
4. With the press count incremented, the address for the device to check the next user press against is also incremented to the proper address.

Appendix K: Figure 3.3 – Magnification of Middle of the Simon Game



1. User entered the last button in the sequence correct, so the device is waiting in the *Lights_Off* state for it to be released.
2. The press count and *cycle_addr* count are reset for the next round of the game.
3. A new random number is generated and added to sequence.
4. There is a slight delay, so the user may distinguish their button press from the first signal of the *Cycle* state.
5. The *Cycle* state begins again.
6. When the first slow tick is recorded, a light/sound is output from the device. This is so that the lights and sounds are synchronous with the slow tick and all of the same duration.

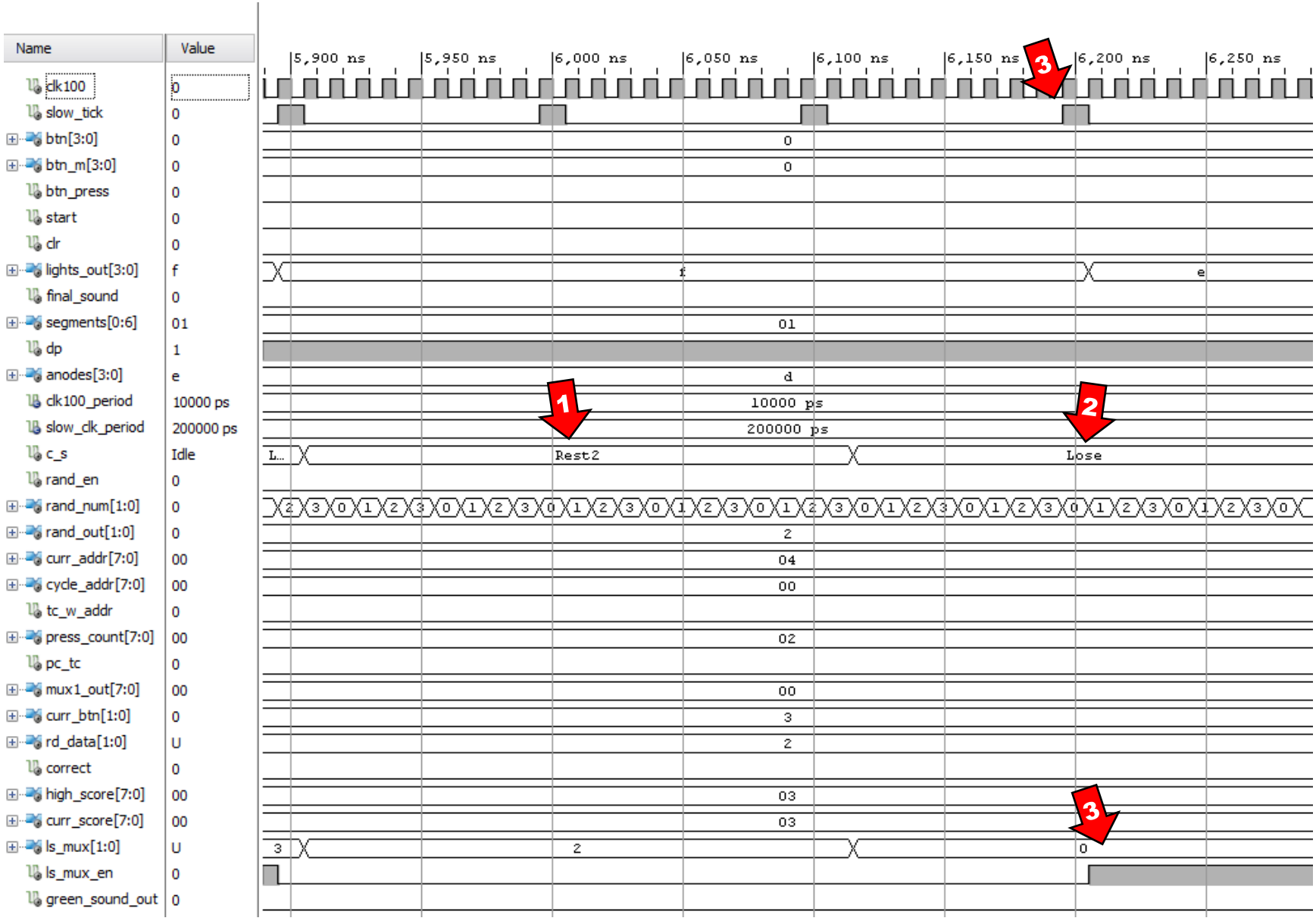
Appendix K: Figure 4.1 – Magnification of Losing the Simon Game



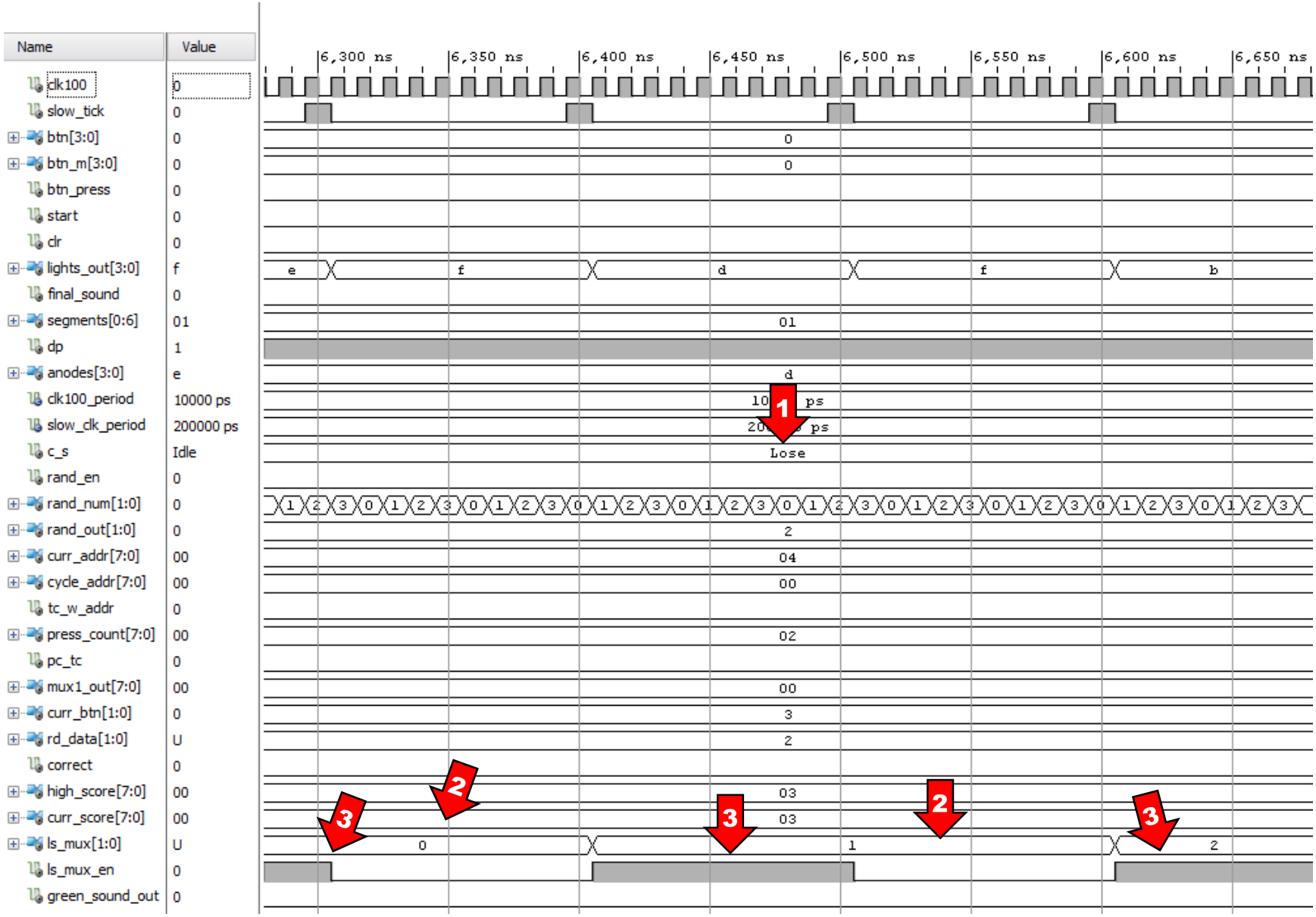
1. At this moment, the *correct* signal checks whether *curr_btn* (3) matches the data read out of the RAM, *rd_data* (0). This is not correct and correct is not set high. When the device exits the lose state, it goes to the *Lights_Off_2* state rather than the *User* state.

2. The *Lights_Off_2* state is held on for the duration that the button is held, so that the device does not enter a new state while the user is still pressing a button.

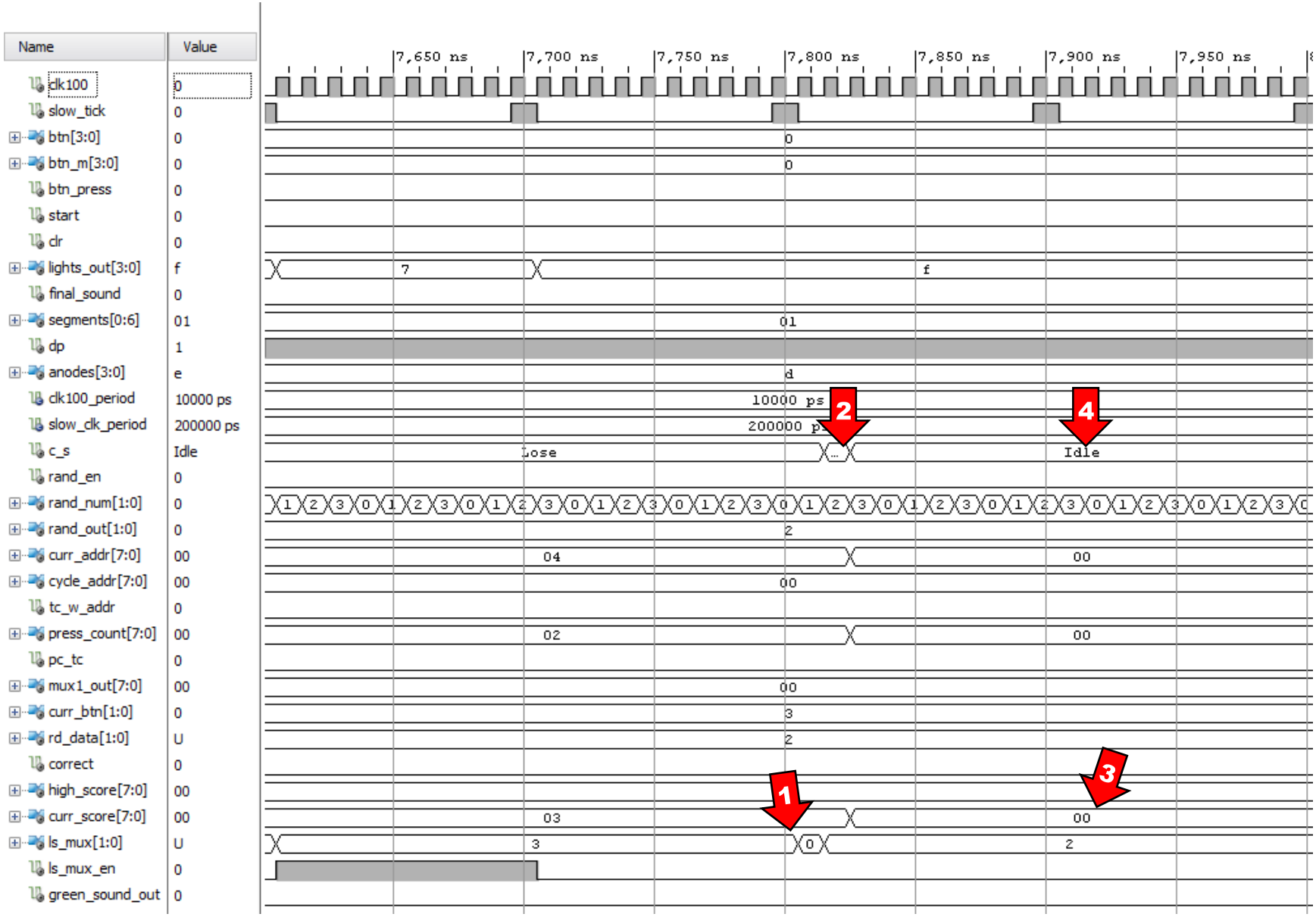
Appendix K: Figure 4.2 – Magnification of Losing the Simon Game



1. Rest delay just before continuing the lose sequence. This is so that after the user releases the button, there is a delay before the lose sequence begins and the user can tell the difference in lights and sounds.
2. The lose sequence commences.
3. On the first slow tick, the lights and sound are enabled. This is so that the lights and sound are on for the same duration.



1. In the lose sequence.
2. The *lose_count*, shown by the output from the *ls_mux* is incremented every two clock cycles.
3. However, at the same time the *ls_mux_en* enabling the lights and sound is set every other clock cycle, so that there is a slight delay between the lights and sounds being displayed.



1. The lose sequence is completed and the lose count, which is indicated by the output from *ls_mux*, is reset to 0.
2. The device enters the *Reset_Counts* state.
3. During the *Reset_Counts* state, the current score was reset to 0, so that during a new game the user can achieve a new score. However, the high score was not reset.
4. The device reenters the *Idle* state and awaits for the user to press the start button to commence with a new game.