ECED 4260– Final Project

Design of a Dice Game

Prepared by

Aethan Cubitt (B00877256)

Benjamin Jarrin (B00846633)

Justin Goguen (B00872480)

Submitted to

Dr. Yuan Ma
Dalhousie University
Halifax, Nova Scotia
12/03/2024

Table of Contents

Abstract

Three students in the ECED4260 class developed and implemented ‘Dice Game’ onto an De1-Soc Board. This paper summarizes the design behind each of the modules, delving into the flow of data and signals. Subsequently, the paper shows the simulation and testing of the ‘Dice Game’ both on the computer software ModelSim, as well as an overview of the testing done on the De1-Soc Board. The students successfully implemented Dice Game and made a second iteration to improve redundancy of signals within the game.

Introduction

The main task for this project was to design a CRAPS dice game in HDL and implement it on the DE1-SoC board. The primary goal was to achieve a functional implementation. The project requires designing and creating all the modules and entities, testing them separately, and then interfacing them in the top-level modules. After achieving a working design, the goal is to optimize the design for increased efficiency and to reduce redundancy.

The game needs to have an enter and reset input from the user which will be used to roll the dice and reset the game. The main outputs must be win and loss signals for the user to see the game results. Furthermore, two 7-segment displays should show the value of the dies.

After the game is implemented onto the DE1-SoC board, it will be able to work for all the tries the user desires, and the outputs should be clearly shown.

Design

To complete the design of the Dice Game project, a collection of modules, as well as the controller and data path needed to be created. The methods of these modules are described below.

Controller

A finite state machine is designed to act as the controller for the Dice Game. It is designed to take seven signals as an input, D711, D2312, D7, eq, enter, clock and reset and it has four outputs, which are loss, sp, roll and win.

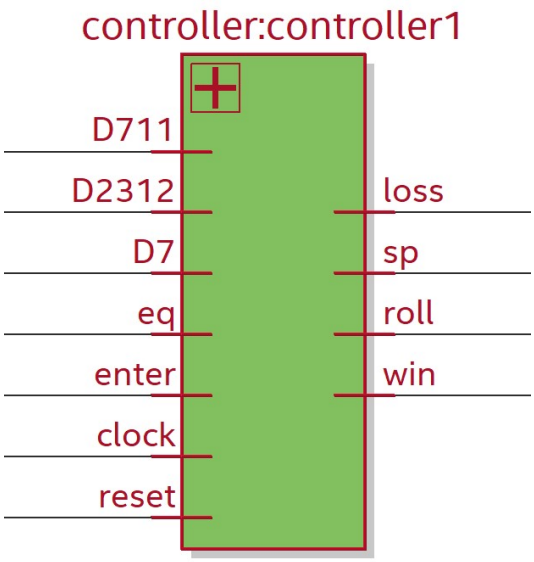


Figure 1: Controller module

The state transition diagram of the controller is shown below, there are six states which the machine can find itself in.

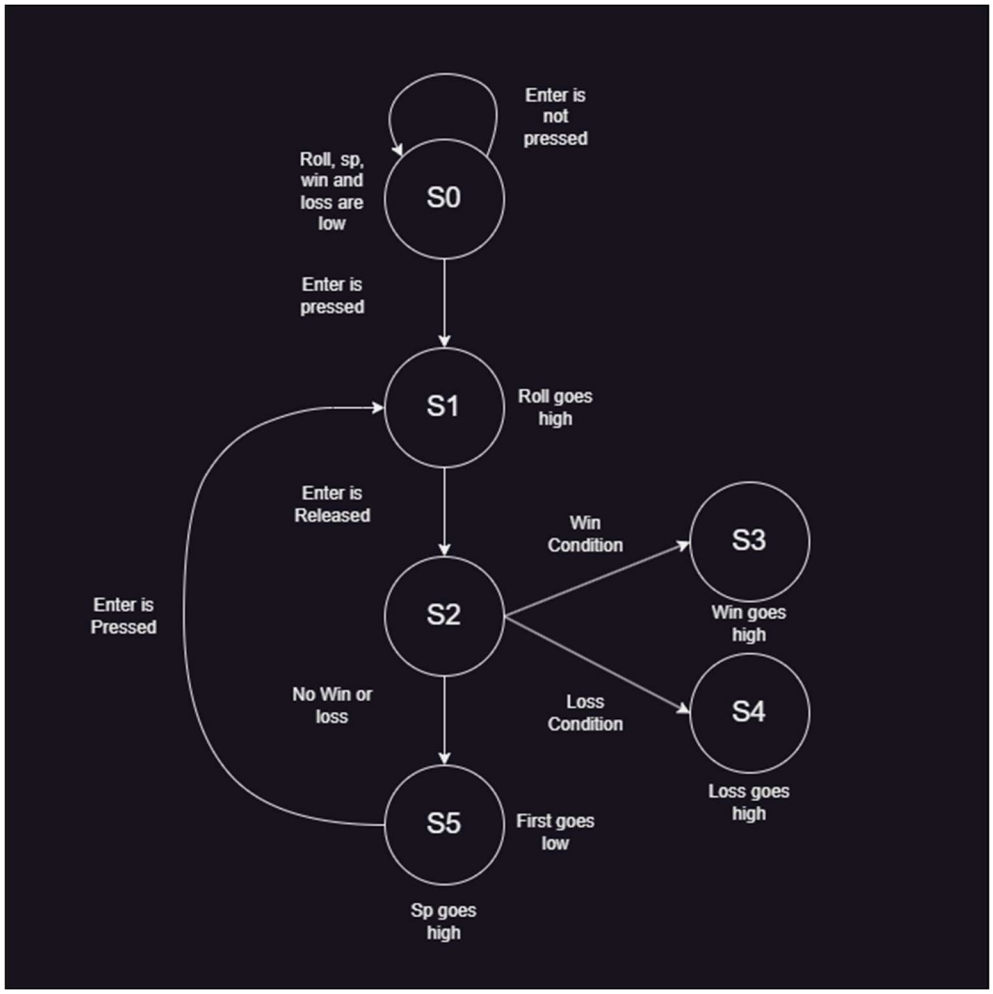


Figure 2: State transition diagram

The states of the finite state machines are explained below:

State 0: This is the initial state of the system, in this case the controller will have all its outputs low. An internal signal, called First is also set to high for this state. The First signal is used in the win and loss logic, as there are conditions which can trigger a win or a loss only on the first roll. The system will remain in this state if the reset signal is low, indicating that the reset button is pressed, or if the enter signal is high which indicated the enter button is not being pressed. The machine will transition into State 1 when the enter signal goes low, indicating the button is pressed.

State 1: In this state the roll signal is asserted high. The system will remain in this state until the enter signal returns to high, which indicates that the enter button is no longer being pressed. When the enter signal returns to high, the machine will enter State 2.

State 2: In this state the roll signal is set low. The machine is checking for any of the win or loss conditions. The signals which can trigger a win or loss condition are D7, D711, D2312 or eq. If the internal signal first is high and D711 is high or if first is low and eq is high then the machine will transition into State 3, which is the win condition. If the first signal is high and D2312 high or the first signal is low and D7 is high then the machine will transition into State 4 which is the loss condition. If none of these signals goes high, then the machine will transition into State 5 and if first is high, the SP signal will go high.

State 3: In this state, the system is in the win condition. The win signal is asserted high and it will remain in this state until the reset signal goes low.

State 4: In this state, the system is in the loss condition. The loss signal is asserted high and it will remain in this state until the reset signal goes low.

State 5: In this state, the machine is preparing for the next roll to occur. The first signal is asserted low in this state. The machine will remain in this state until the enter signal goes low again and it will transition into State 2. The first signal is asserted low in this state.

The state machine is designed to have a synchronous reset, which will return the machine to State 0.

Pseudo Random Number Generator

The design of the pseudo random number generator has two complicated components. The first is the design of the pseudo random numbers themselves followed by differing the timing of the two different clocks so that the rolls will be different sets of pairs.

The design of the pseudo random numbers originates with implementing an XOR gate between the 2nd and 1st outputs of the flops. This is a common implementation for pseudo random numbers. However, when implementing the XOR gate, the state machine reaches 111 which is not a dice roll. This can be fixed by XORing the output of the XOR with a 1 when the state 011 is appearing. This edit to the XOR produces the sequence 100, 001, 010, 101, 011, 110 ->100, or the decimal 4-1-2-5-3-6 repeating.

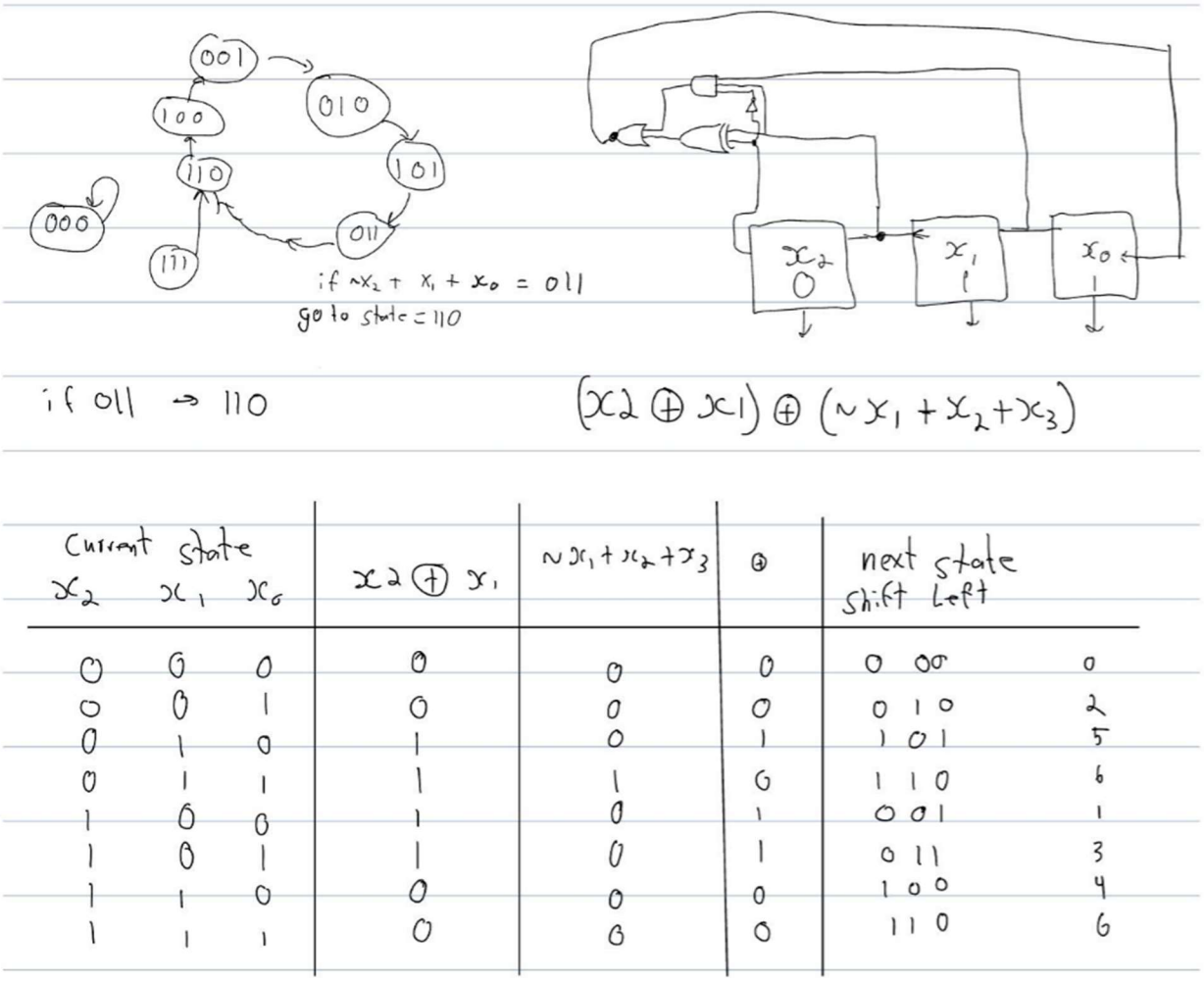


Figure 3: State diagram

The state diagram shows the next state with the applied logic gate functions applied. This circuit essentially skips the 111 state and hits all other states between 1-6 inclusive.

With the pseudo random number generator ready to be implemented in the code, the second component of design was to be able to output different pairs of random numbers using the same function. This was done by changing the frequency that the random number generator switches states. Of the two implemented dice, the first die switches at half the clock frequency, while the second die switches at a third of the clock frequency.

Comparator

The comparator has a simple job, with a single output and many inputs. The comparator needs to output if the two inputs are equal, through the change in state of the Eq signal. With the timing concerns out of the scope of the comparator, the comparator always outputs if the input numbers, a and b, are equivalent.

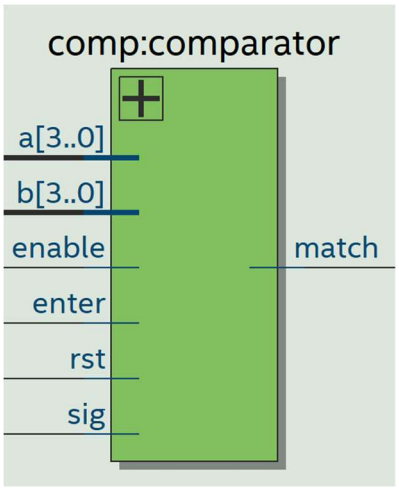


Figure 4: Comparator module

Adder

The adder is one of the components within the Datapath that allows us to add the results of the dies. This Verilog module is a three-bit adder shown in the figure below.

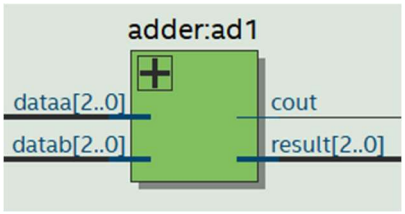


Figure 5: Adder module

As seen, the adder takes the three-bit final outputs from die 1 and die 2 as inputs. Then, it performs a three-bit addition and has a three-bit output plus a carry-out. These two form the output bits that the addition could result in. These sum and carry-out signals are used by other components in the data path. The adder module does not work with a clock signal; as long as there are inputs, it will add them.

Point Register

The point register is the component within the data path that stores the value of the point of the game. As shown in the image below, the point register module takes as inputs the main outputs of the adder (the sum and carry out), a set point, and reset signals from the controller.

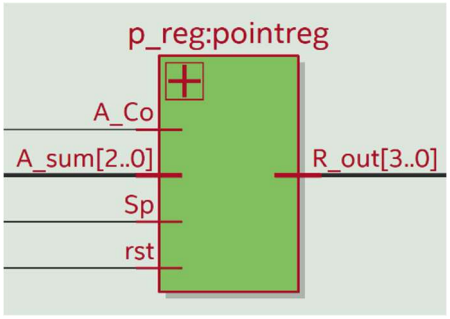


Figure 6: Point register module

After the first roll of the dies, if the player does not immediately win or lose, the controller will set the set point signal high. After detecting the positive edge of this set point signal, the register will store the adder values as the outputs. If the reset signal is set high, the point register will reset its values to zero which allows the component to be ready for a new game. The point register is important to store the value that will determine the win conditions for the next stages. The output of this module is used by the comparator.

After doing the main implementation of the game, it was identified that the reset signal was not needed. As a result, it was removed to improve the efficiency of the design.

Test Logic

The test logic module is located in the data path and is the one that sets signals high to determine the winning or losing conditions in the first round or the losing conditions in the subsequent rounds. This module can be seen in the figure below.

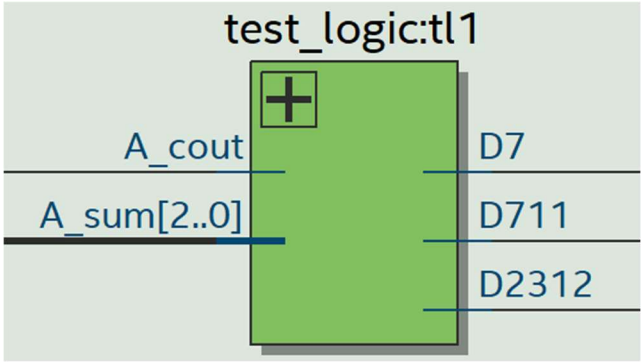


Figure 7: Test logic module

As seen, the test logic takes as inputs the main outputs of the adder: the sum and carry out to represent the full 4-bit sum of the dies. It then checks for the value of this input. If the value is seven, it will set the D7 and D711 signals high. If the value is 11 it will set the D711 signal high. If the values are 2, 3, or 12 it will set the D2312 signal high. These signals are used by the controller. In the first round, it checks the D711 signal for a win and the D2312 signal for a loss. In the next rounds, it will check the D7 signal for a loss.

Register Display

The register display is a module in the data path that allows us to display a 4-bit number into two 7-segment displays. This module can be seen in the figure below.

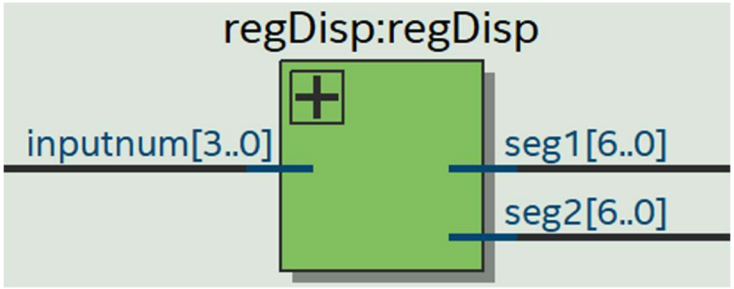


Figure 8: Reg display module

As seen, the input is a 4-bit number and there are two outputs each of which controls a 7-segment display. Internally, it isolates the tens digit by dividing the input by decimal ten and looking at the integer part. It then assigns the output to light the respective 7-segment display as 0 or 1. For the ones digit, it isolates the ones digit by using the modulo operator to calculate the remainder of the input divided by 10. Then, it assigns the output representing the digits from 0 to 9 in the corresponding 7-segment display. This module is used twice in the data path to output the values in the point register and in the adder, which allows the player to monitor the important parameters in the game.

Display

The display module in the data path allows a three-bit number to be displayed onto the 7-segment display. The module can be seen in the figure below.

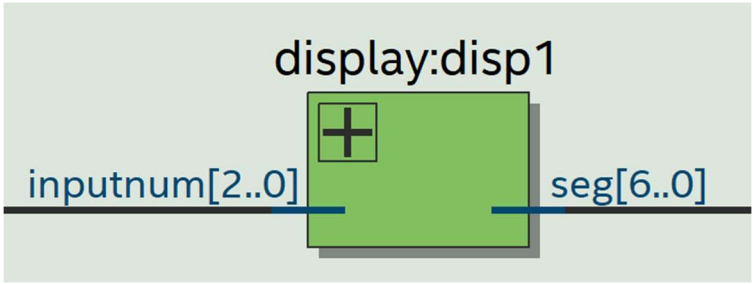


Figure 9: Display module

The theory behind the module is quite simple. There is a case statement, that converts the binary number to its respective representation on the 7-segment display. The numbers for the seven segment are active low, so there is a ~ operator to flip the bits offering the correct output on the display. The display changes on the change of the input num to change the display.

Datapath

The data path is a crucial component within the overall architecture of the Dice Game system. The data path is the part of the hardware that produces all the computation and processes data within the system. The data path needs to deal with the inputs between the data path and the controller. These inputs are then mapped to the different modules residing within the data path. This is shown below:

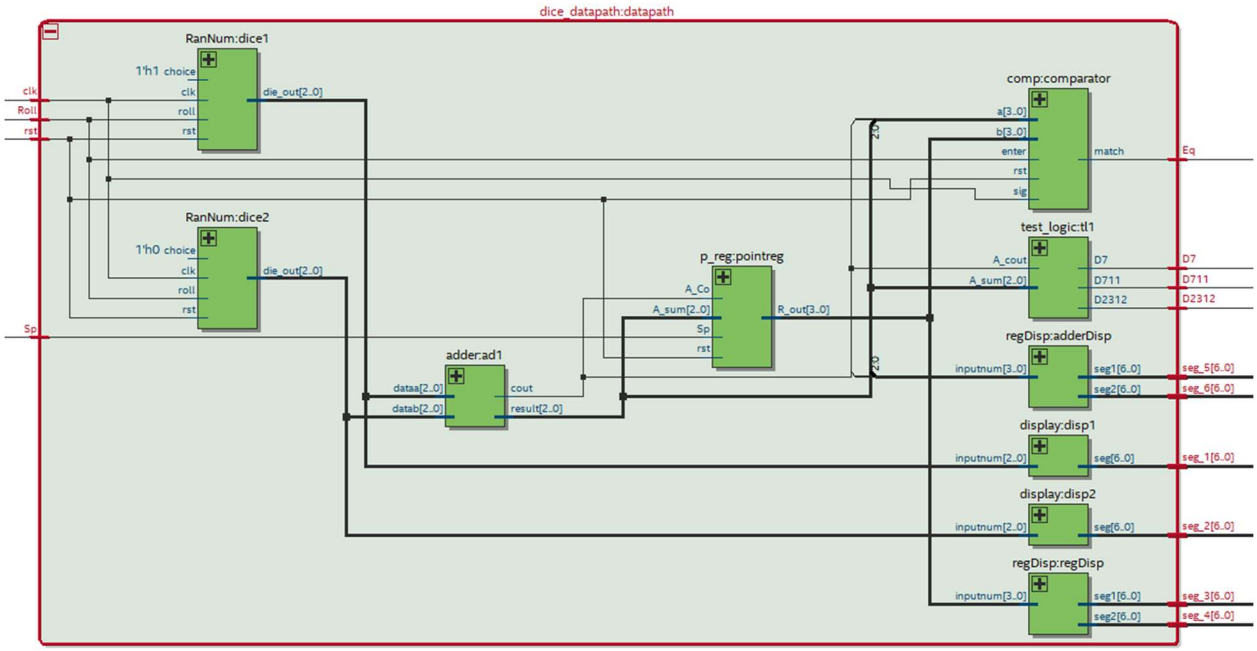


Figure 10: Datapath diagram

Within the data path there are three types of variables. The first two are input and output variables. The third is wires that interconnect modules that are not direct outputs of the data path.

There are five inputs to the data path, which include the following: first, Sp, clk, Roll, and rst. These control signals enact the timing on all the individual modules within the system.

There are 6 outputs from the data path, these include: Eq, D7, D711, D2312, seg_1[6:0], and seg_2[6:0]. These signals are only outputs of the system and are only changed from modules in the data path.

The last type of signal in the data path is a wire. A wire connects the output of one module to another module that is dependent on the signal. This wire needs to be created as it is a connection that needs to be made between modules that is not an input or output of the data path.

The data path is responsible for outputting the roll of the dice to the adder, as well as its output displays. The adder signal outputs its result to the point register as well as the comparator. The comparator outputs if the point register and the added number are equal through Eq. The test logic receives the added number to identify if said number is a 2, 3, 7, 11, or 12 which is output through different output signals. Within the random number generator function, a new clock was implemented which switched on a different frequency for the 1 or 0 input of the choice parameter. To prevent the numbers from being constantly updated to the adder and display a second always function was added to only show the output of the random number generator when the roll signal passes a negative edge, or the reset signal is activated.

Dice Game

With both the controller and the data path completed, they are combined to create the final entity. The roll signal and the sp signals which are outputs from the controller are mapped as inputs to the data path. The signals D7, D711, D2312 and which are output from the data path are mapped as inputs to the controller. Both blocks share a clock signal as well as the reset signal as inputs. The final entity takes the signal enter and reset as inputs and outputs a win and loss signal, as well as displaying the numbers which were rolled from both dice on two seven segment displays.

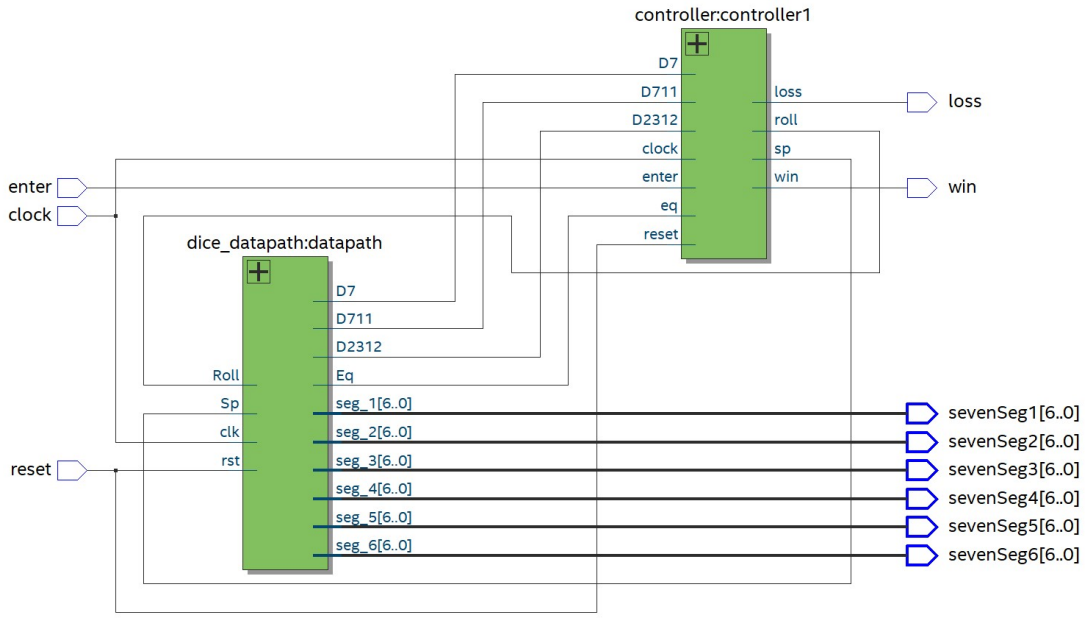


Figure 11: Dice game diagram

Simulation & Testing

Controller

The testing of the controller involves setting the conditions for the state machine to enter each state and confirming that each state transition is as expected. This test is done to ensure that once the data path is interfaced with the controller, it will function as expected with minimal debugging. The sequence of the game is mimicked in the test bench, where the enter signal is brought low to simulate a button press, and the win and loss signals from the data path are triggered depending on which outcome is desired. The game is to continue should none of these signals be triggered.

After the first roll, the conditions tested for are as follows: the D711 and D7 signal are brought high, indicating a win condition after the first roll. The desired outcome is that roll is set high one clock cycle after the enter signal is set low, and for roll to remain high until one clock cycle after enter returns to high. The D711 and D7 signal are set high in the next clock cycle, the controller should respond by setting the win signal high two clock cycles later.

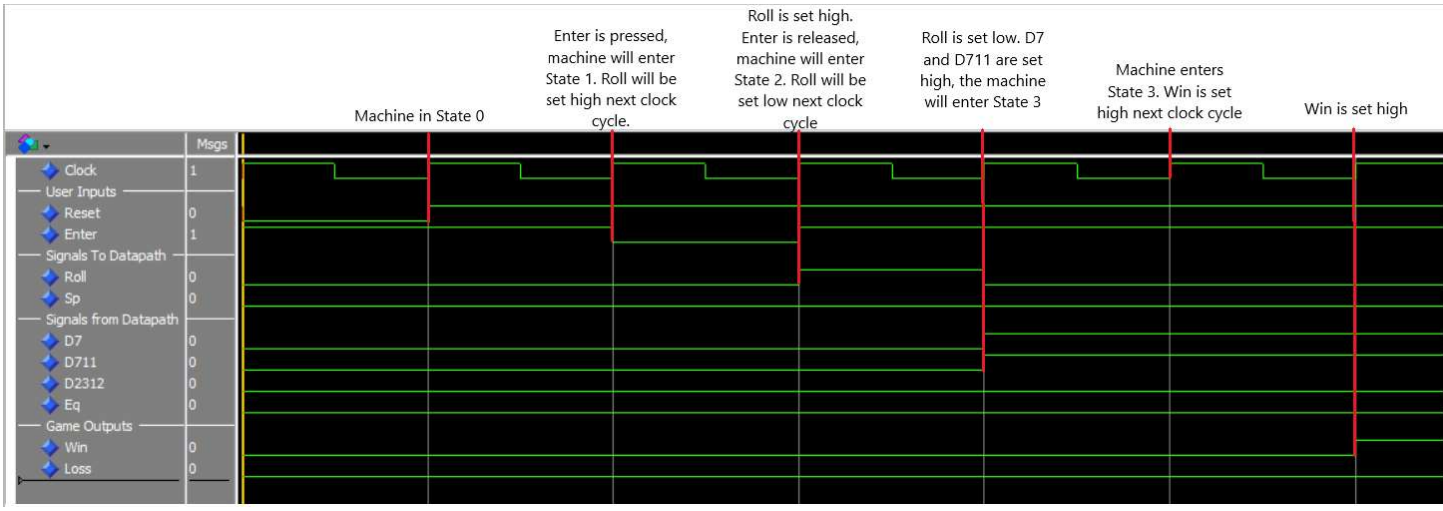


Figure 12: Controller waveforms

Next the D2312 signal is brought high after the first roll indicating a loss condition. The sequence will remain the same as the previous test, but after Roll is set low, the D2312 signal is set high instead. The controller should respond by asserting the loss signal high two clock cycles after D2312.

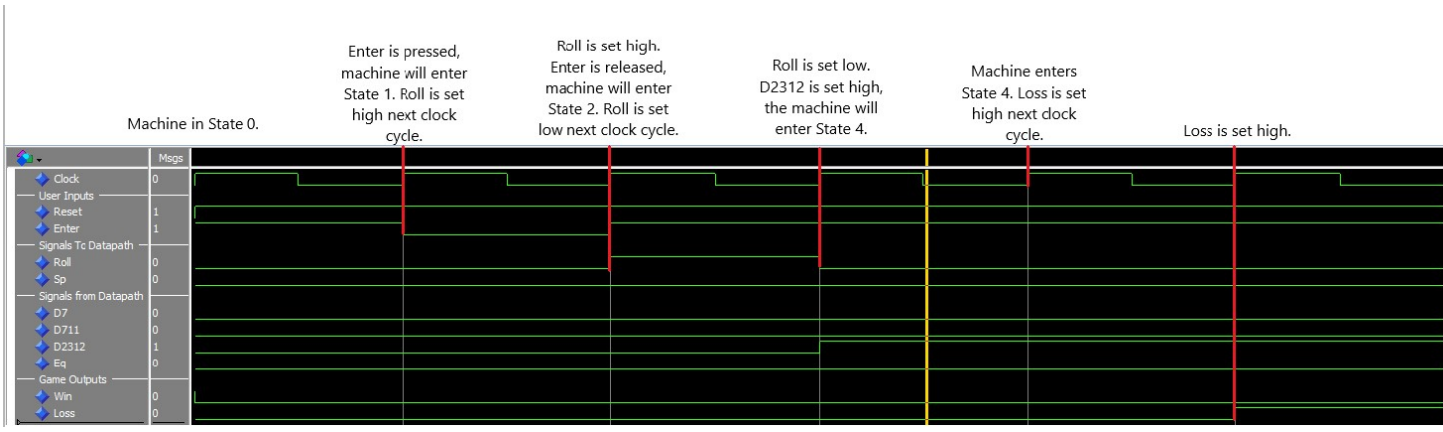


Figure 13: Controller waveforms

Next none of those signals will be brought high after the first roll, and the game will continue. After the second roll, the D7 and D711 are brought high indicating a loss condition. The expected sequence of the controller is the same as previous but D711 and D2312 are to remain low after the roll signal is brought low. The Sp signal should go high at the next clock cycle and it will remain high only for the one cycle. Then the enter button is pressed again and roll goes high one cycle later for as long as enter is pressed. Once roll is set low the D7 signal is set high and the controller will respond by setting the loss signal high two cycles later.

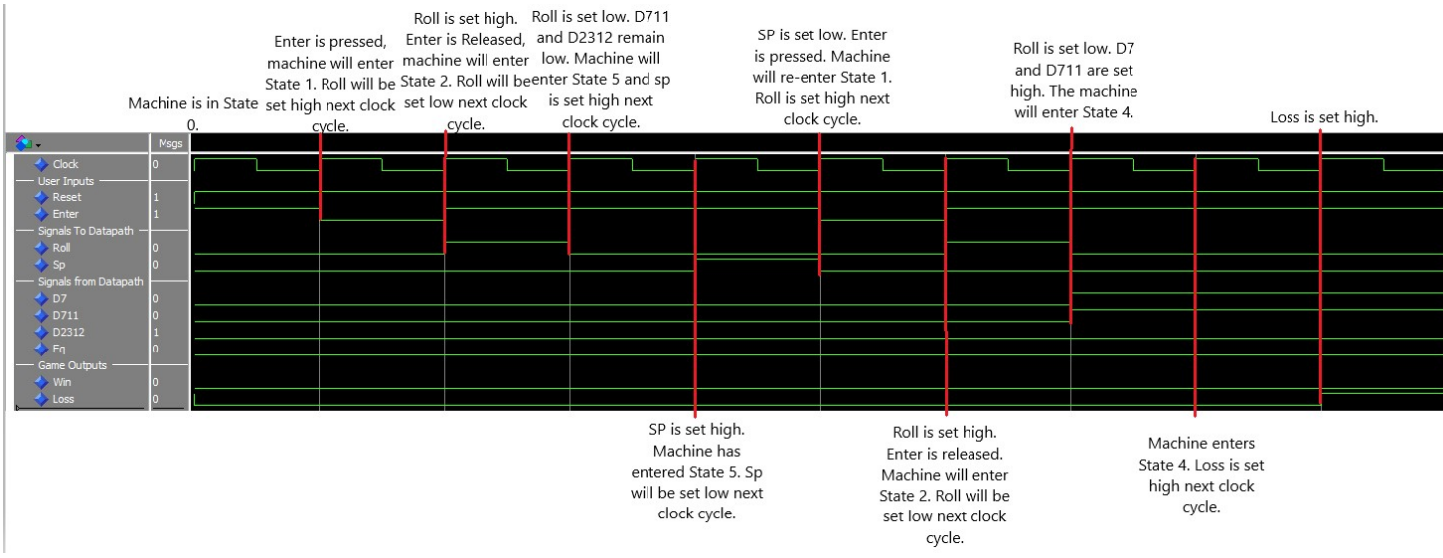


Figure 14: Controller waveforms

Lastly after a second roll, the no signals are brought high, and after the third roll the eq signal is brought high, indicating a win condition. This confirms the ability of the controller to continue a game indefinitely, until a win or loss condition does occur. The sequence is repeated same as the previous test, but after the second roll D7 and Eq both remain low. Enter is pressed again and roll is set high after the one clock cycle. After roll goes low, the

Eq signal is brought high and the controller responds by setting the win signal high after two clock cycles.

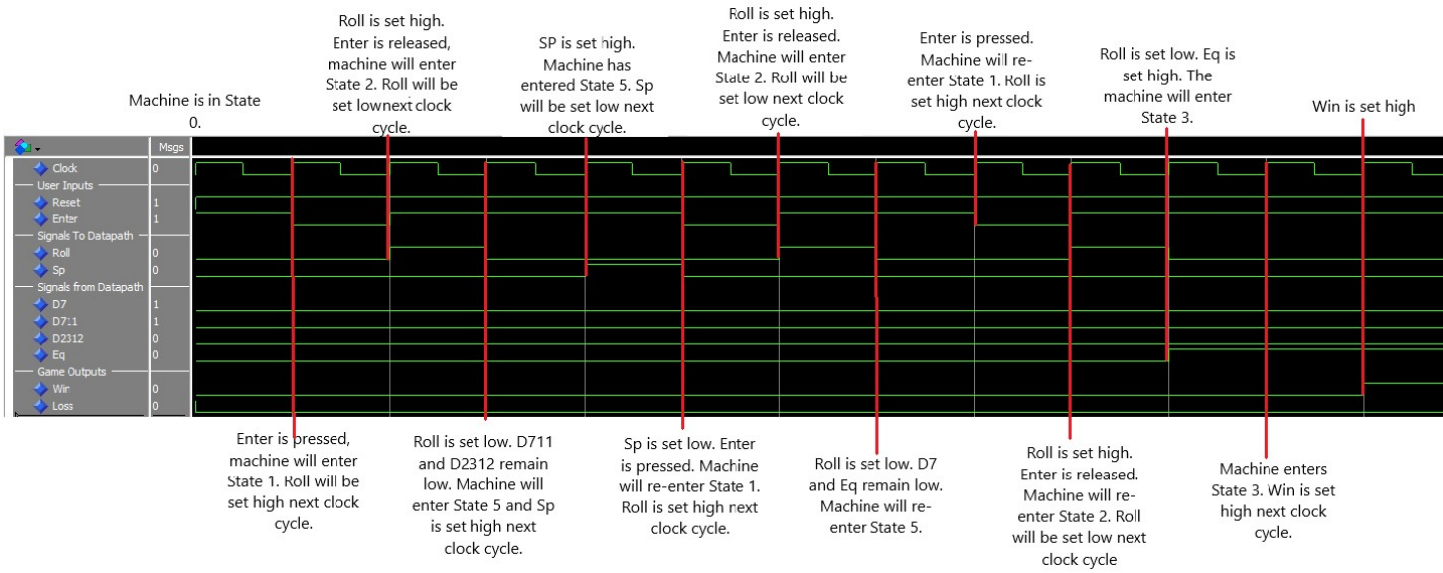


Figure 15: Controller waveforms

Adder:

To test the adder the strategy used is to test all the possible input combinations from 0+0 to 6+6 and see if the outputs give the correct sum values. This is a good test because by testing all the possible addition possibilities that the module will get from the dies, we can ensure there are no errors.

The inputs will be the possible values from the dies from 0 to 6 and each value will be added to each other at least once. A clock signal will be used to increment the input values using the clock for testing purposes. The outputs will be the 3-bit sum and the carry-out bit: both of these combined will form the 4-bit final value. For testing, an additional waveform showing the decimal sum value will be shown. It is expected that the outputs will be equal to the sum of the inputs.

After testing, the waveforms in the figure below have been obtained. As seen for all the different input cases, the outputs are equal to the sum of the inputs in both the binary and decimal cases. This shows how the module is working correctly in simulation and is giving out the desired outputs.

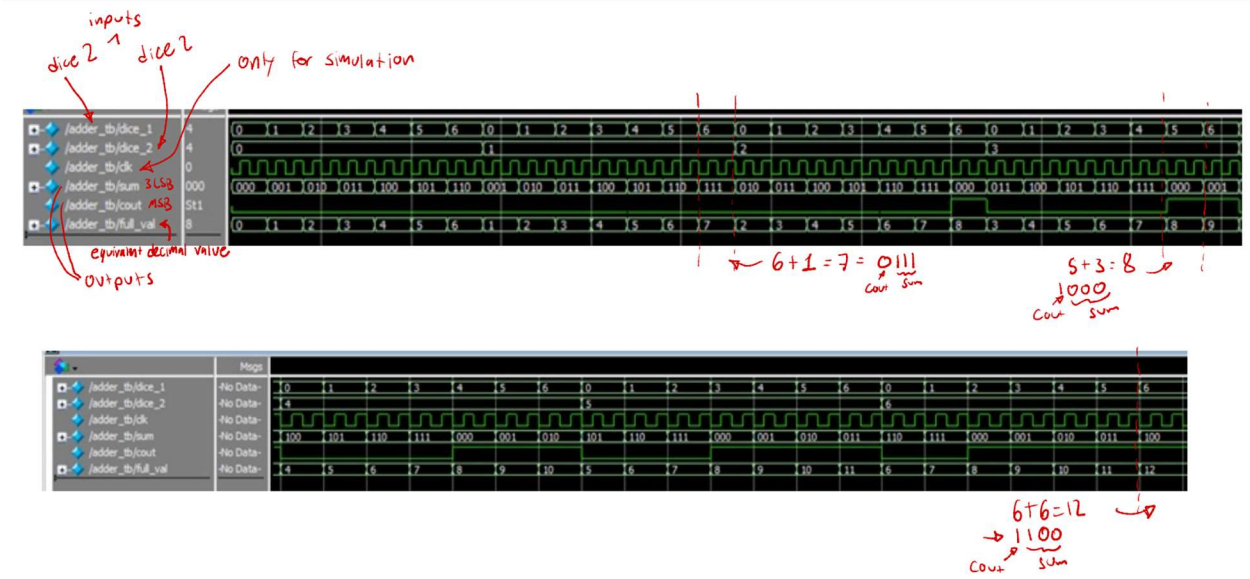


Figure 16: Adder waveforms

Point Register

To test the point register module, the testing sequence will simulate different values for the inputs this register could get. This means that values of 0 and 1 will be chosen for the carry-out input, different 3-bit combinations for the sum input will be selected, and a clock signal for the set point input will be used to see how the register gets updated following the enabling signal. The output will then be observed to see if the right values are chosen at the right moment. This is a good test because it will allow us to see whether the register stores the correct input values at the desired enabling time. The expected outputs should be a 4-bit value: the MSB should be equal to the carryout input and the three LSB should be equal to the sum input. The output values should only be updated at the positive edge of the set point signal.

After testing, the waveforms in the figure below were obtained. As seen, the output of the register is equal to the combination of the carry-out input as the MSB and the sum input as the 3 LSB. Also, it can be seen how the output value gets updated only at the positive edge of the set point signal, which is the expected behavior.

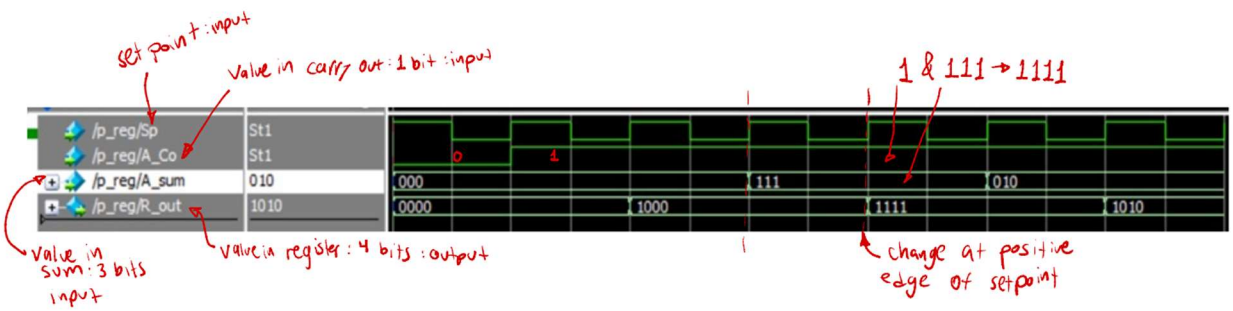


Figure 17: Point register waveforms

Test Logic

For the test logic module, the approach to test will involve simulating multiple inputs that would normally come from the adder module and seeing if the right output signals go high. The testing will involve showing a case in which each of the signals would go high and a case in which none should go high. This is a good test because we would be able to see the different input cases and whether the outputs are showing as desired. The expected behavior is that when the input is 7, both the D7 and D711 should go high. When the input is 11, the D711 output should go high. The inputs are 2, 3, or 12 the D2312 output signal should go high. For all the other inputs, all the outputs should be low.

After simulating, the waveforms in the figure below were obtained. As seen, the corresponding outputs go high if one of the inputs matches the value it is looking for. For all the other cases the outputs are low. This determines the successful behavior of the module.

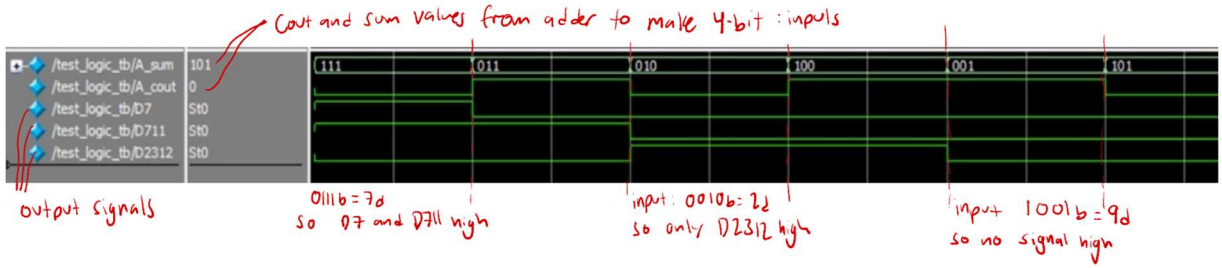


Figure 18: Test logic waveforms

Register Display

For the register display module, the testing approach will involve simulating all the possible 4-bit input cases and seeing that the outputs to the 7-segment displays light the correct numbers. This is a good test because by seeing all the possible input cases vs all the possible output cases we could determine whether the module is going to behave as expected. The inputs will be numbers from 0 to 15 and the expected outputs would be that

the output controlling the tens digit display shows zero for inputs from 0 to 9, and one for inputs of 10 to 15. The output controlling the one's display should output the sequence that is going to light the correct number.

After running the simulation, the waveforms in the figure below were obtained. As seen in all the cases, the 4-bit input value is getting the right output control signals toward the displays. The seven-segment display controlling the tens only outputs 0 or 1, and the seven-segment display controlling the ones correctly outputs the sequence that will light the digit. This shows how the module is behaving as predicted.

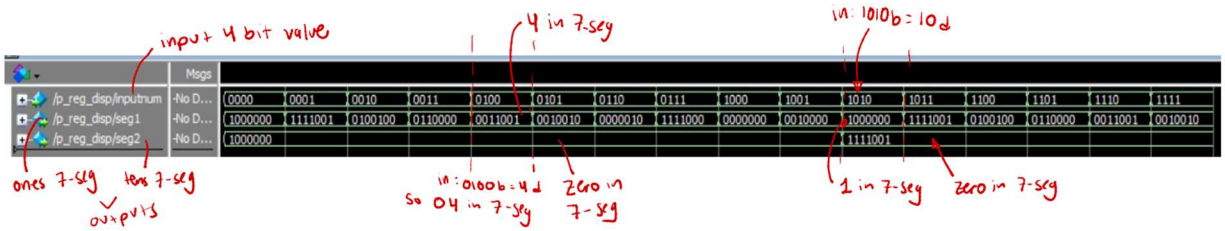


Figure 19: Reg display waveforms

Display

The seven segment display, that inputs a 3-bit number and outputs the segments for the 7-segment display are shown in the wave from below.

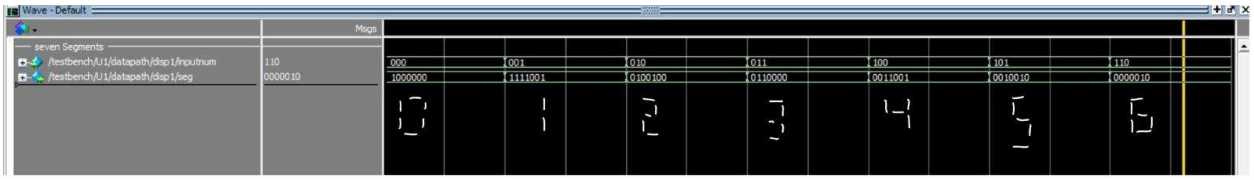


Figure 20: Display waveforms

In the first row, we can see the binary numerical input, and the output in the lower row correlates to the segments of the display. The annotations to the waveform show the output on the display.

Comparator

The comparator is a simple module. The always block is continuously comparing the two inputs, a being the result from the adder, and b being the value in the point register. When these two inputs are equal, the output variable match, which is traced to Eq, goes high. When the input variables are different, the output variable match is low.

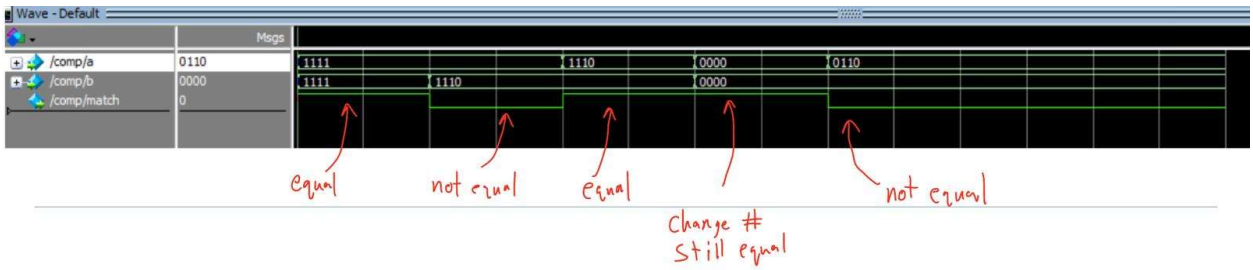


Figure 21: Comparator waveforms

It is also valuable to note that if a and b change but are still equal, the output variable match stays high.

Pseudo Random Number Generator

There are three significant conclusions from simulating and testing the random number generator.

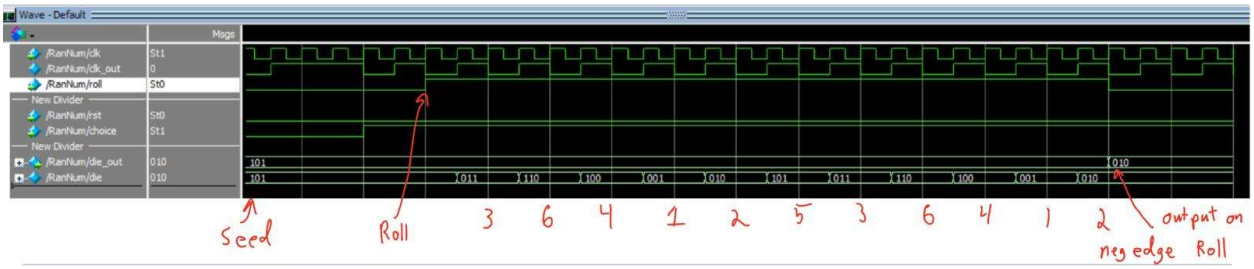


Figure 22: Pseudo Random Number Generator waveforms

The first comes from the figure above. The random number generator rolls the dice, for the duration of the button press rolling through the pseudo random numbers and stopping then outputting the roll at the end of roll being pushed. The die roll pseudo randomly cycles through the numbers 1 to 6 as a die should. The second two notable outcomes come from the wave form below.

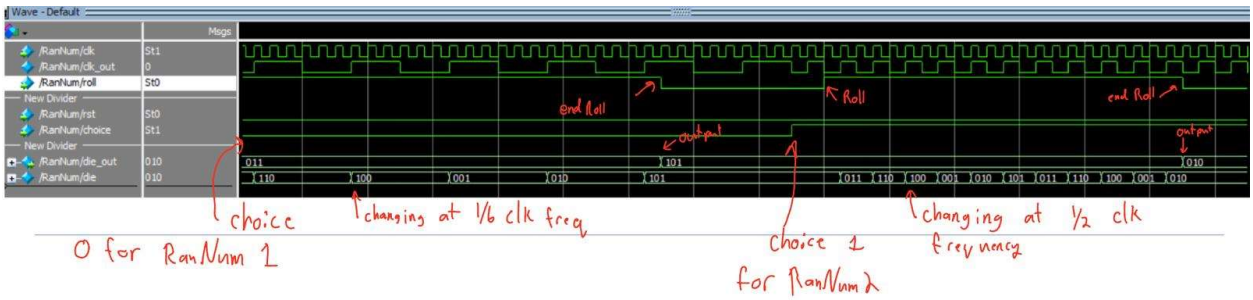


Figure 23: Pseudo Random Number Generator waveforms

The second notable outcome is the speed at which the dice rolls. Either dice rolls at ½ or 1/6th the speed of the clock, which is 50MHz, this means that a human cannot accurately get repeating dice outcomes. The last notable piece is that the two random number

generators will always output random pairs, or in other words the rolls will not come in the same pairs as if a 2 on die one would always give a 4 on die 2. The two different dice roll at different frequencies giving an output of always random dice rolls.

Data Path

The data path has numerous signals changing during the operation of the dice game. Starting with the signals at the top of the diagram shown below, the inputs come from the data path, and control what happens within the data path. Clock is consistently changing, and reset is active low, so initially the system is reset with a 0, and then a reset value of 1 starts normal operation.

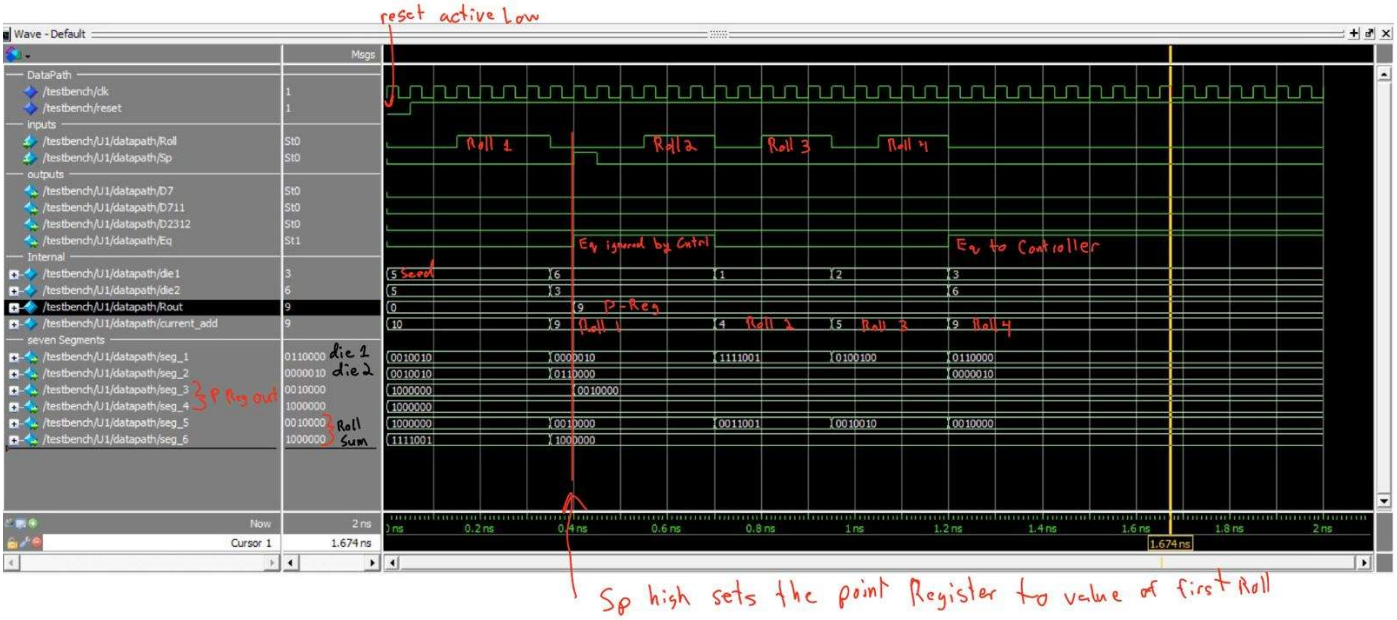


Figure 24: Datapath waveforms

The Roll input is given from the controller, given by the user. During roll, both dice are rolling, and upon Roll stopping, the negative edge triggers the update of roll from RanNum. Sp comes from the controller to set the first roll as the value in the point register. Eq is set high upon the first roll, as the point register and adder outputs are equal, however this is ignored by the controller. At the end of roll four, the value of the dice is equal to the point register, and the comparator sets the value of Eq to high, signifying a losing condition. There are three sets of outputs from the seven-segment display to help the user understand the game. The first two segments display output the roll of each individual dice. The Point register value is in the 3rd and 4th segment displays to show the user what the first roll was. Lastly the 5th and 6th segment displays show the sum of the current roll to help quickly sum the roll for the user.

Dice Game

The entire dice game is simulated.

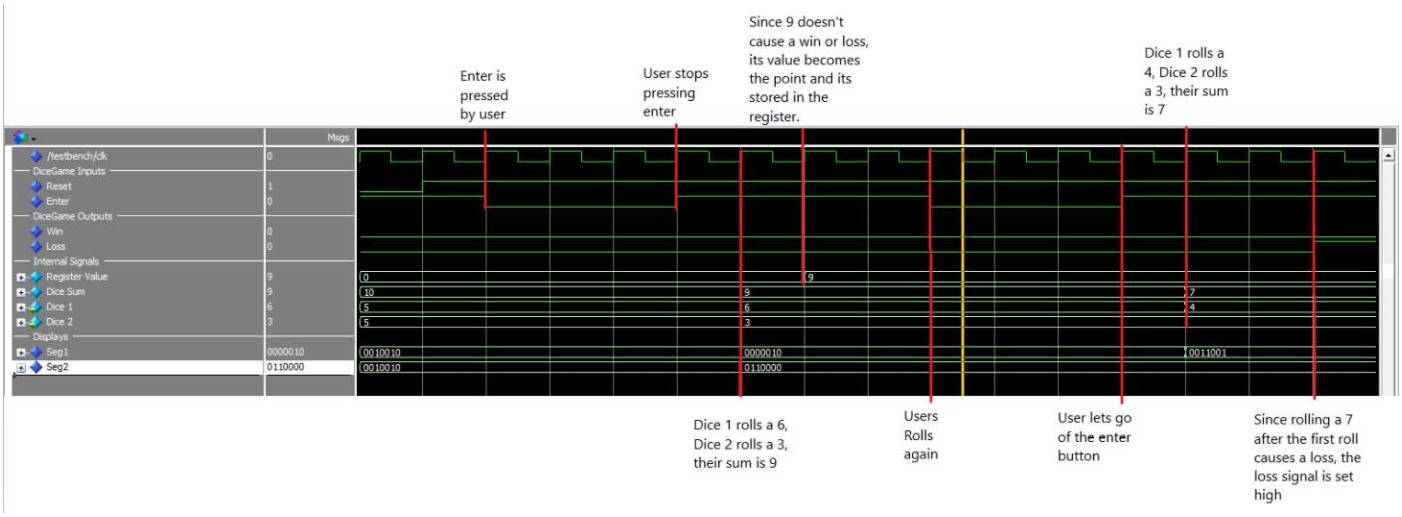


Figure 25: Dice game waveforms

In this round, the first roll results in a sum of 9. This number does not cause a win or a loss, and it becomes stored in the register as the point. The users then rolls again, which causes the dice to sum the number 7. Rolling a 7 after the first roll results in loss, two clock cycles later the loss signal is set high. This is exactly as expected, the rules of the game are followed.

Another round is played, after the first roll the sum of the dice is 6 and this value becomes the point. There are two more rolls which result in the number 12 and 5 being rolled after the 4th roll the number obtained is 6. The win signal is set high two clock cycles later. The game is functioning as expected.

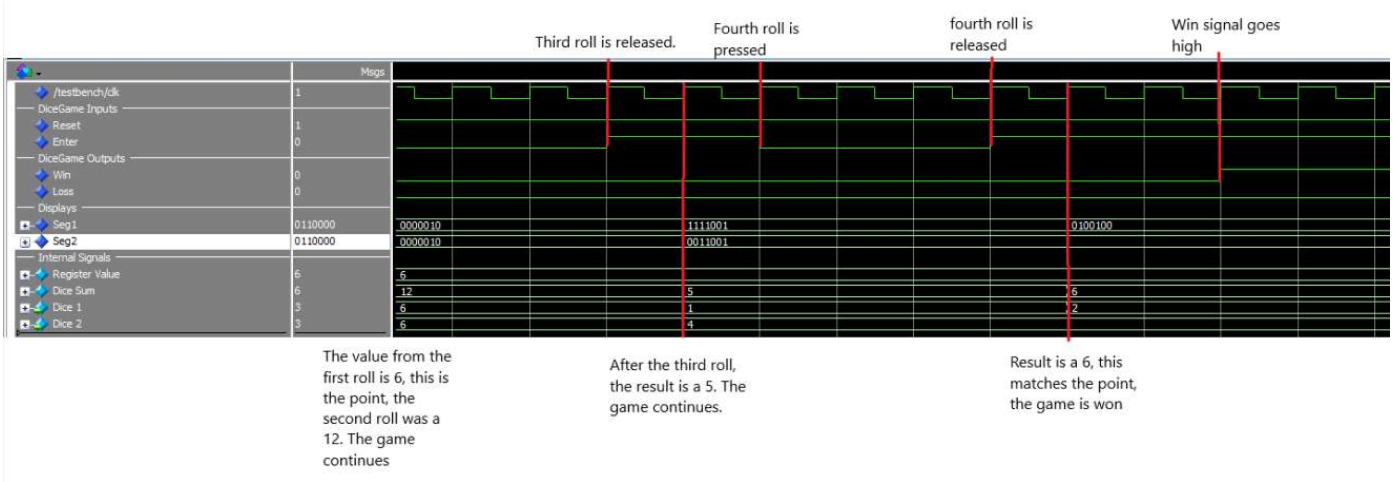


Figure 26: Dice game waveforms

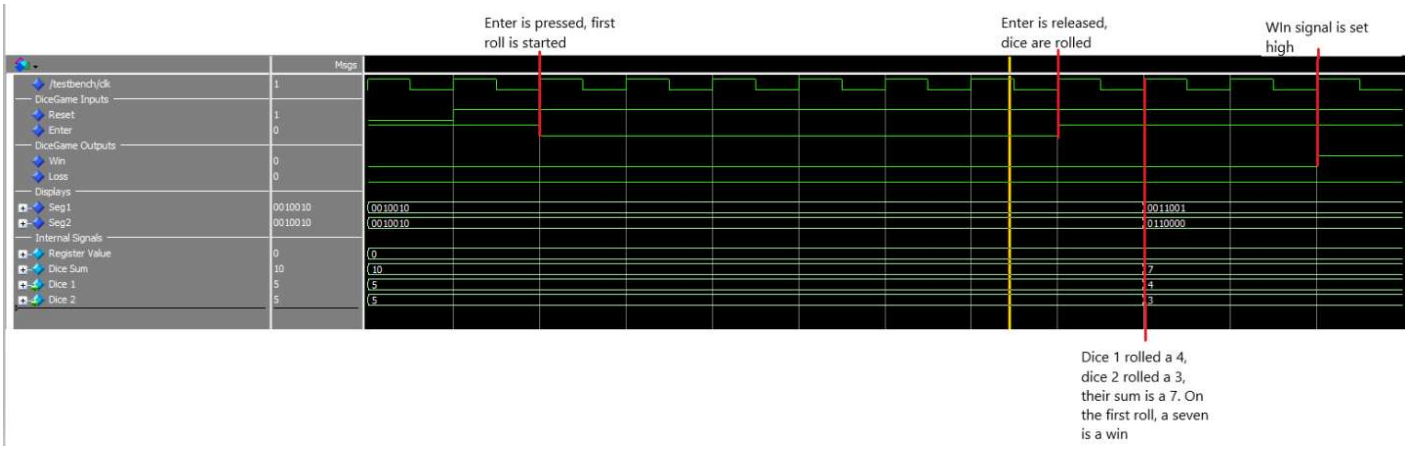


Figure 27: Dice game waveforms

After the first roll, the sum of the dice is a 7, which causes the game to be won. As expected, the win signal is set high.

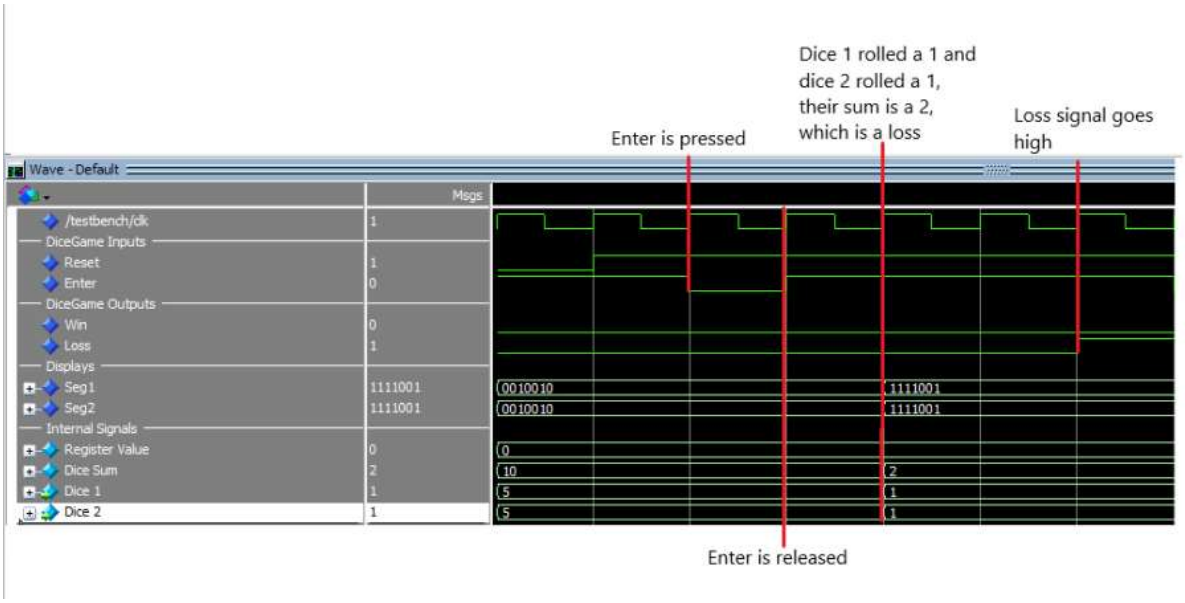


Figure 28: Dice game waveforms

The last test, after the first roll the sum of the dice is 2, which causes the game to be lost, as expected the loss signal goes high.

Discussion & Conclusion

The Dice Game incorporated different technical theories from the Digital IC course. The Controller utilized the students' knowledge of finite state machines and VHDL code. The controller communicated solely with the De1-Soc Board inputs and outputs, as well as with the data path. We used our knowledge of Verilog – especially from lab 2- to create a data path, which essentially organizes all the routing between the modules in the data path. Using the various modules within the data path, the logic for the dice game was created. The Dice Game works as we expected it to, with a design iteration to improve on the signals used from the first version.

After doing the implementation, it was found that it was possible to optimize the design by removing some signals and inputs that were not crucial to the operation of the game. For example, in the point register, the reset signal was removed. Furthermore, in the comparator module, the enable signal was removed.

When implementing the design, some issues appeared. One of them was that we experienced timing issues with the outputs of the comparator and the controller reading the signal at the right time. The issue was overcome by changing the state in which the controller sets the "first" flag low. It got moved from doing so in state 2 to doing so in state 5 to resolve the issue.

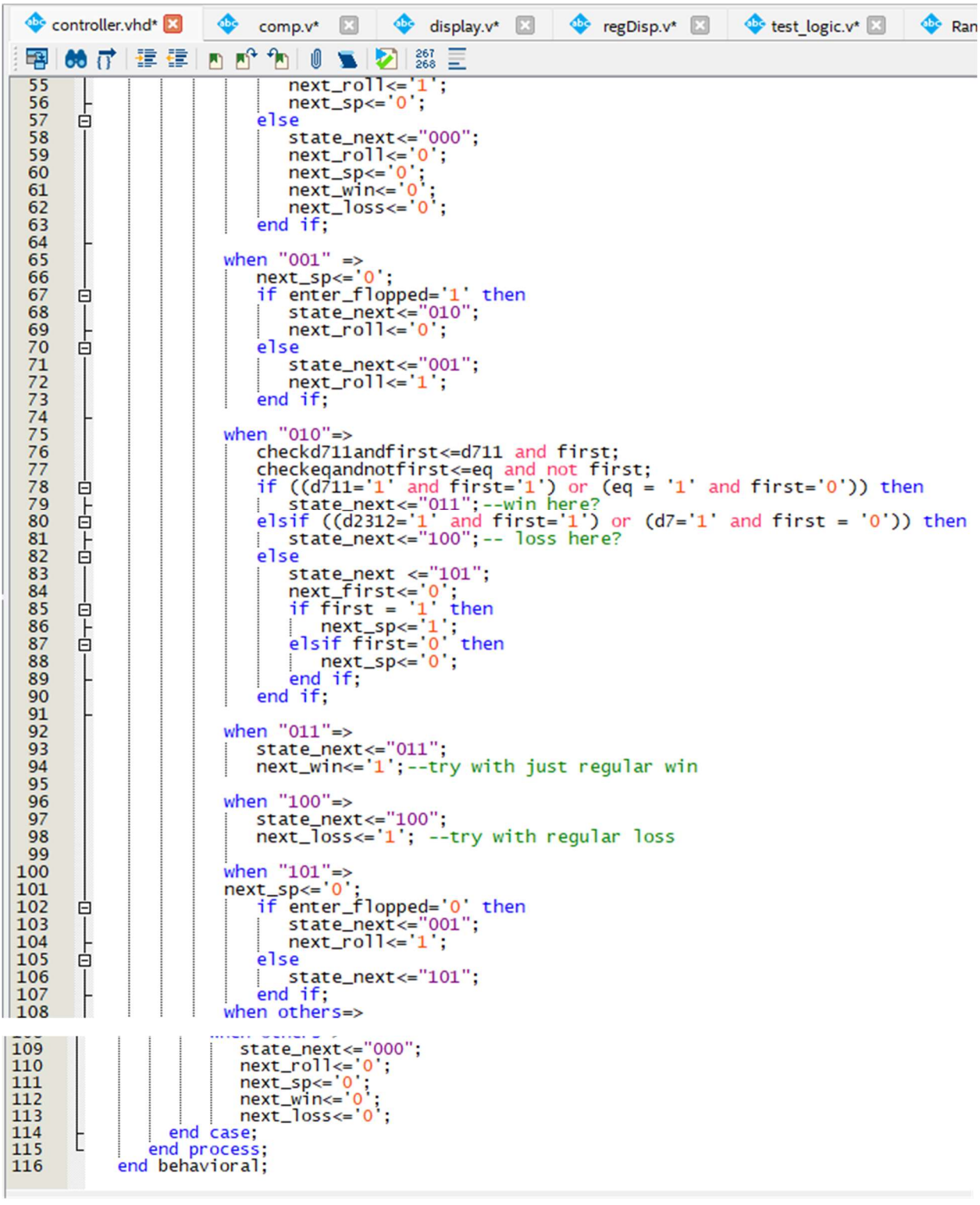
Another issue was that initially the comparator only XORed A and B. This was fixed by making this module only check if both of these values are equal. The difference in timing cause by this change allowed for proper operation of the datapath.

In future works, some additional considerations that can be implemented to optimize the design are using a Mealy machine for the controller instead of a Moore machine. This way, we would be able to decrease the number of states from six to four. Another consideration would be combining the outputs carry-out and sum outputs of the adder into a single bus to avoid having two different signals coming out of this module. Finally, the design could be optimized by getting rid of the D7 signal in the test logic and only using the D711 signal. This way, the controller can be modified to use only the latter signal for the win or loss conditions, by raising an internal flag that will let it determine if it is a winning or losing condition depending on the state the game is at.

Appendices

controller.vhd* | comp.v* | display.v* | regDisp.v* | test_logi

```
--Justin


library ieee;
use ieee.std_logic_1164.all;
entity controller is
   port(clock: in std_logic;
        enter: in std_logic;
        reset: in std_logic;
        D7: in std_logic;
        D711: in std_logic;
        D2312: in std_logic;
        eq: in std_logic;
        roll: out std_logic:='0';
        sp: out std_logic:='0';
        win: out std_logic:='0';
        loss: out std_logic:='0');

end entity;


architecture behavioral of controller is

signal state, state_next: std_logic_vector(2 downto 0):="000";
signal next_first,enter_flopped, first: std_logic:='1';
signal next_roll, next_sp,next_win, next_loss: std_logic:='0';
signal checkd711andfirst,checkeqandnotfirst: std_logic:='0';
begin
   process(clock, reset, first, D7, D711, eq, D2312, enter_flopped)
      begin
         if (reset='0') then
            state <= "000";
            next_roll<='0';
            next_sp<='0';
            next_win<='0';
            next_loss<='0';
            next_first<='1';

         elsif(clock'event and clock='1') then
            enter_flopped<=enter;
            state<=state_next;
            roll<=next_roll;
            sp<=next_sp;
            win<=next_win;
            loss<=next_loss;
            first<=next_first;


         end if;
         case state is
            when "000"=>
               if enter_flopped='0' then
                  state_next <="001";
```


controller.vhd*
comp.v*
display.v*
regDisp.v*
test_logic.v*
Ran
next_roll<='1';
next_sp<='0';
else
state_next<="000";
next_roll<='0';
next_sp<='0';
next_win<='0';
next_loss<='0';
end if;
when "001" =>
next_sp<='0';
if enter_flopped='1' then
state_next<="010";
next_roll<='0';
else
state_next<="001";
next_roll<='1';
end if;
when "010"=>
checkd711andfirst<=d711 and first;
checkeqandnotfirst<=eq and not first;
if ((d711='1' and first='1') or (eq = '1' and first='0')) then
state_next<="011";--win here?
elsif ((d2312='1' and first='1') or (d7='1' and first = '0')) then
state_next<="100";-- loss here?
else
state_next <="101";
next_first<='0';
if first = '1' then
next_sp<='1';
elsif first='0' then
next_sp<='0';
end if;
end if;
when "011"=>
state_next<="011";
next_win<='1';--try with just regular win
when "100"=>
state_next<="100";
next_loss<='1'; --try with regular loss
when "101"=>
next_sp<='0';
if enter_flopped='0' then
state_next<="001";
next_roll<='1';
else
state_next<="101";
end if;
when others=>
state_next<="000";
next_roll<='0';
next_sp<='0';
next_win<='0';
next_loss<='0';
end case;
end process;
end behavioral;

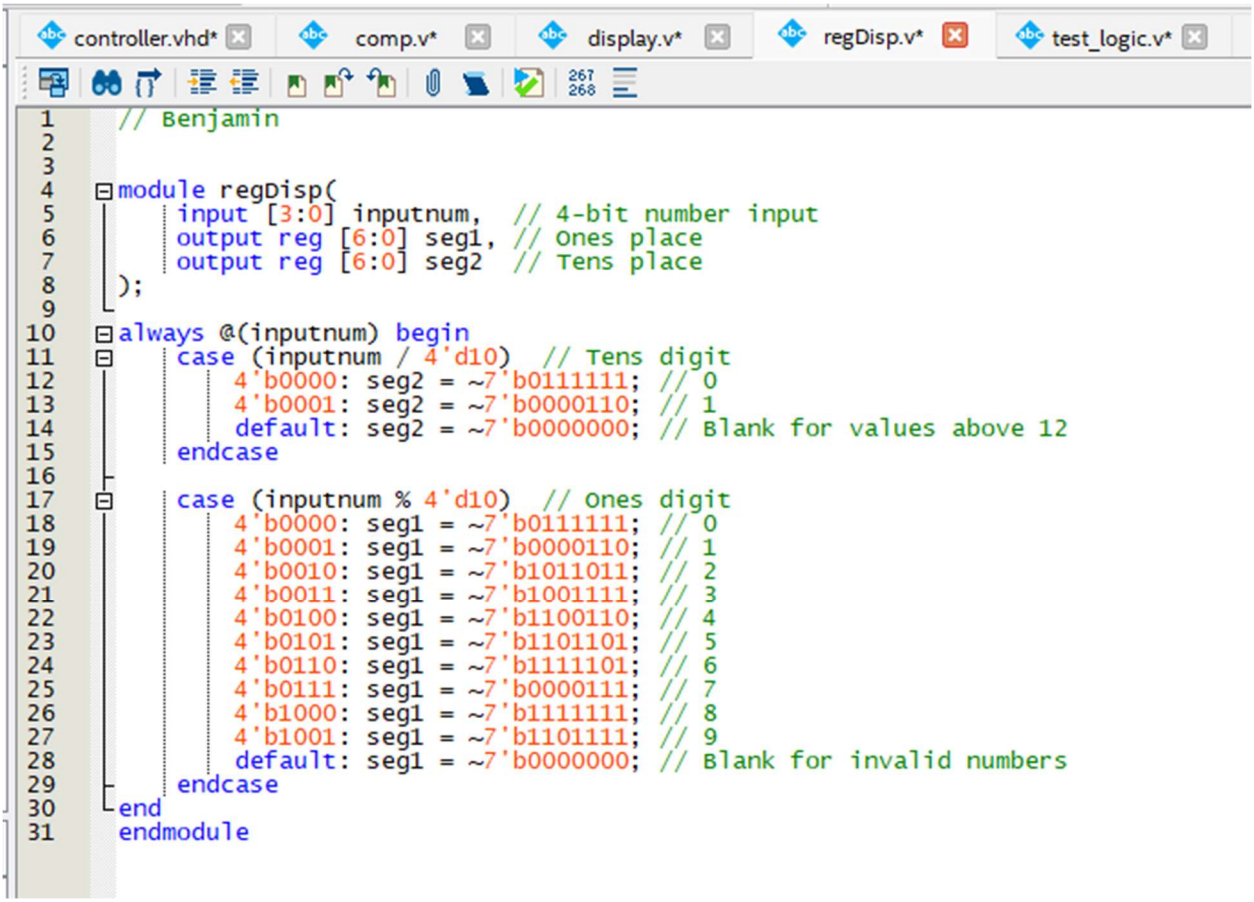
Data Path

controller.vhd* comp.v* display.v* regDisp.v*

```
// Aethan & Ben
module dice_datapath(
    input Roll,
    input Sp,
    input clk,
    input rst,
    output D7,
    output D711,
    output D2312,
    output Eq,
    output [6:0] seg_1, //dice one
    output [6:0] seg_2, //dice two
    output [6:0] seg_3, //p_reg
    output [6:0] seg_4, //p_reg
    output [6:0] seg_5, //adder
    output [6:0] seg_6  //adder
);

    wire [2:0] die1;
    wire [2:0] die2;


    wire [2:0] sum;
    wire Cout;
    wire [3:0] Rout;
    wire [3:0] current_add;

    assign current_add = {Cout, sum};

    adder ad1(
        .dataa(die1),
        .datab(die2),
        .result(sum),
        .cout(Cout)
    );

    test_logic tl1(
        .A_sum(sum),
        .A_cout(Cout),
        .D7(D7),
        .D711(D711),
        .D2312(D2312)
    );


    p_reg pointreg(
        .Sp(Sp),
        .A_Co(Cout),
        .A_sum(sum),
        .rst(rst),
        .R_out(Rout)
    );
```

```
54      display disp1(
55          .inputnum(die1),
56          .seg(seg_1)
57      );
58
59      display disp2(
60          .inputnum(die2),
61          .seg(seg_2)
62      );
63
64      RanNum dice1(
65          .clk(clk),
66          .roll(Roll),
67          .rst(rst),
68          .choice(1'b1),
69          .die_out(die1)
70      );
71
72      RanNum dice2(
73          .clk(clk),
74          .roll(Roll),
75          .rst(rst),
76          .choice(1'b0),
77          .die_out(die2)
78      );
79
80      comp comparator(
81          .a(current_add),
82          .b(Rout),
83          .sig(clk),
84          .rst(rst),
85          .match(Eq)
86      );
87
88      regDisp regDisp(
89          .inputnum(Rout),
90          .seg1(seg_3), //ones place
91          .seg2(seg_4) //10s place
92      );
93
94      regDisp adderDisp(
95          .inputnum(current_add),
96          .seg1(seg_5),
97          .seg2(seg_6)
98      );
99
100
101 endmodule
```

Register Display




```
1   // Benjamin
2
3
4   module regDisp(
5       input [3:0] inputnum,  // 4-bit number input
6       output reg [6:0] seg1, // Ones place
7       output reg [6:0] seg2  // Tens place
8   );
9
10  always @(inputnum) begin
11      case (inputnum / 4'd10)  // Tens digit
12          4'b0000: seg2 = ~7'b0111111; // 0
13          4'b0001: seg2 = ~7'b0000110; // 1
14          default: seg2 = ~7'b0000000; // Blank for values above 12
15      endcase
16
17      case (inputnum % 4'd10)  // Ones digit
18          4'b0000: seg1 = ~7'b0111111; // 0
19          4'b0001: seg1 = ~7'b0000110; // 1
20          4'b0010: seg1 = ~7'b1011011; // 2
21          4'b0011: seg1 = ~7'b1001111; // 3
22          4'b0100: seg1 = ~7'b1100110; // 4
23          4'b0101: seg1 = ~7'b1101101; // 5
24          4'b0110: seg1 = ~7'b1111101; // 6
25          4'b0111: seg1 = ~7'b0000111; // 7
26          4'b1000: seg1 = ~7'b1111111; // 8
27          4'b1001: seg1 = ~7'b1101111; // 9
28          default: seg1 = ~7'b0000000; // Blank for invalid numbers
29      endcase
30  end
31  endmodule
```

Test Logic

controller.vhd* comp.v* display.v* regDisp.v* test_log

```
1   //Benjamin
2
3   module test_logic (
4       input [2:0] A_sum,
5       input A_cout,
6       output reg D7,
7       output reg D711,
8       output reg D2312
9   );
10
11      always @(*) begin
12
13          case ({A_cout, A_sum})
14              4'b0111: begin  // Equal to 7
15                  D7 = 1;
16                  D711 = 1;
17                  D2312 = 0;
18              end
19              4'b1011: begin  // Equal to 11
20                  D7 = 0;
21                  D711 = 1;
22                  D2312 = 0;
23              end
24              4'b0010,        // Equal to 2
25              4'b0011,        // Equal to 3
26              4'b1100: begin  // Equal to 12
27                  D7 = 0;
28                  D711 = 0;
29                  D2312 = 1;
30              end
31              default: begin
32                  D7 = 0;
33                  D711 = 0;
34                  D2312 = 0;
35              end
36          endcase
37      end
38
39  endmodule
40
```

Pseudo Random Number Generator

controller.vhd* | comp.v* | display.v* | regDisp.v* | test_logic.v* | RanNum.v*

```
1   //Aethan Cubitt
2   module RanNum(
3       input clk,                  // clk
4       input roll,
5       input rst,
6       input choice,               // Choose frequency for clk 1 or 2
7       output reg [2:0] die_out    // Output die
8   );
9       wire feedback;
10      reg clk_out;
11      reg [1:0] counter;
12      reg [2:0] die;              // lfst register
13
14      // state logic equation for the lfsr
15      assign feedback = (die[2] ^ die[1]) ^ (~die[2] && die[1] && die[0]);
16
17      always @(posedge clk or negedge rst) begin
18          if (!rst) begin          //if rst active low
19              clk_out <= 0;
20              counter <= 0;
21          end else begin
22              if (choice) begin
23
24                  clk_out <= ~clk_out;      //for 1/2 clk frequency
25
26              end else begin
27
28                  if (counter == 2) begin      //for 1/6 clk frequenc
29                      clk_out <= ~clk_out;
30                      counter <= 0;
31                  end else begin
32                      counter <= counter + 1;
33                  end
34              end
35          end
36      end
37        //the actual changing the dice role on 1/2 or 1/6 clk frequency
38      always @(posedge clk_out or negedge rst) begin
39          if (!rst) begin
40              die <= 3'b101;          //reset
41          end else if (roll) begin
42              die <= {die[1:0], feedback}; ///the shift of the lfsr
43          end
44      end
45      // updates when roll is let go
46      always @(negedge roll or negedge rst) begin
47          if (!rst) begin
48              die_out <= 3'b101;     //rst
49          end else begin
50              die_out <= die;        // output current die
51          end
52      end
53  endmodule
```

Point Register

```
1   //Benjamin
2   module p_reg(Sp, A_Co, A_sum, rst, R_out);
3       input Sp;
4       input A_Co;
5       input [2:0] A_sum;
6       input rst; //add in datapath
7       output reg [3:0] R_out;
8
9
10      always @(posedge Sp or negedge rst) begin
11        if (~rst) begin
12            R_out <= 4'b0000;
13        end
14        else begin
15            R_out <= {A_Co, A_sum}; // Concatenate A_Co and A_sum into R_out
16        end
17    end
18  endmodule
19
20
21
```

Display

controller.vhd* | comp.v* | display.v* | regDisp.

```
1   //Aethan Cubitt
2
3   module display(
4     input [2:0] inputnum,
5     output reg [6:0] seg
6   );
7
8   always @(inputnum) begin
9   case (inputnum)
10    3'b000: seg =  ~7'b0111111;
11    3'b001: seg =  ~7'b0000110;
12    3'b010: seg =  ~7'b1011011;
13    3'b011: seg =  ~7'b1001111;
14    3'b100: seg =  ~7'b1100110;
15    3'b101: seg =  ~7'b1101101;
16    3'b110: seg =  ~7'b1111101;
17
18  endcase
19  end
20  endmodule
21
```

Comparator

controller.vhd* comp.v* display.v* regDisp.v*

```
1   //Aethan & Justin
2
3
4   module comp(
5       input [3:0] a,
6       input [3:0] b,
7       input sig,          // Clock signal
8       input rst,
9           output reg match   // Match output
10  );
11
12      always @(*) begin
13          if (!rst) begin
14              // Reset match to default value
15              match <= 1'b0;
16          end else begin
17              if (a == b)
18                  match <= 1'b1;
19              else
20                  match <= 1'b0;
21          end
22      end
23
24  endmodule
25
```

Adder

controller.vhd* comp.v* display.v* regDisp.v* test_logic.v*

```
1   //Benjamin
2   module adder (
3
4       input [2:0] dataa,     // 3-bit input dataa
5       input [2:0] datab,     // 3-bit input datab
6       output [2:0] result,   // 3-bit result output (sum)
7       output cout            // Carry-out
8   );
9
10      wire c1, c2;           // Intermediate carry wires
11
12      // Full adder for bit 0 with cin set to 0
13      assign {c1, result[0]} = dataa[0] + datab[0] + 1'b0;
14
15      // Full adder for bit 1
16      assign {c2, result[1]} = dataa[1] + datab[1] + c1;
17
18      // Full adder for bit 2
19      assign {cout, result[2]} = dataa[2] + datab[2] + c2;
20
21
22  endmodule
```

```
--Justin

library ieee;
use ieee.std_logic_1164.all;

entity DiceGame is
    port(clock: in std_logic;
         enter: in std_logic;
         reset: in std_logic;
         win: out std_logic:='0';
         loss: out std_logic:='0';
         sevenSeg1: out std_logic_vector(6 downto 0):="0000000"; --dice one
         sevenSeg2: out std_logic_vector(6 downto 0):="0000000"; --dice two
         sevenSeg3: out std_logic_vector(6 downto 0):="0000000"; --reg number in ones place
         sevenSeg4: out std_logic_vector(6 downto 0):="0000000"; --reg number in 10s place
         sevenSeg5: out std_logic_vector(6 downto 0):="0000000"; --adder number in ones place
         sevenSeg6: out std_logic_vector(6 downto 0):="0000000"--adder number in 10s place
         );
end entity;

architecture structural of DiceGame is

component controller is
    port(clock: in std_logic;
         enter: in std_logic;
         reset: in std_logic;
         D7: in std_logic;
         D711: in std_logic;
         D2312: in std_logic;
         eq: in std_logic;
         roll: out std_logic:='0';
         sp: out std_logic:='0';
         win: out std_logic:='0';
         loss: out std_logic:='0');
    end component;

component dice_datapath is
    port(Roll: in std_logic;
         sp: in std_logic;
         clk: in std_logic;
         rst: in std_logic;
         D7: out std_logic:='0';
         D711: out std_logic:='0';
         D2312: out std_logic:='0';
         eq: out std_logic:='0';
         seg_1: out std_logic_vector(6 downto 0):="0000000";
         seg_2: out std_logic_vector(6 downto 0):="0000000";
         seg_3: out std_logic_vector(6 downto 0):="0000000";
         seg_4: out std_logic_vector(6 downto 0):="0000000";
         seg_5: out std_logic_vector(6 downto 0):="0000000";
         seg_6: out std_logic_vector(6 downto 0):="0000000");
end component;
signal sp,roll,D7,D711,D2312,eq: std_logic:='0';

begin
controller1: controller port map(clock, enter, reset,D7,D711,D2312,eq,roll,sp,win,loss);
datapath: dice_datapath port map(roll, sp, clock,reset,D7,D711,D2312,eq,sevenSeg1,sevenSeg2,sevenSeg3, sevenSeg4,sevenSeg5, sevenSeg6);

end structural;
```