M152A Final Project – Boxed

Pierson Marks (304742143), Scott Lee (204955724)

June 6th, 2018

Introduction

In a culmination of the quarter, the final project tested our knowledge of learned skills and our ability to learn about new pieces of hardware. The overall objective was to create a fully operable game on the FGPA Board written in Verilog. By using push buttons, slider switches, and seven-segment display, the project pushed learned implementations to an incredibly complex level by following the guidelines specified in our project proposal. The grading was based on the correct implementation of all components and the overall logic of the game. Initially, the game logic seemed fairly straightforward, however, upon further inspection, there were dozens of extraneous cases we had to handle to ensure the game was intuitive and free of any bugs. The heavy implementation of multiple nested finite state machines, pulsing clocks (in addition to normal clocks), synthesizable ranged random number generation, intermediary register control, multiple digit display, and a binary adder all proved to be challenges in completing this project.

Overview

Our team decided to remaster the famous dice game **Boxed**. Boxed, initially invented over a thousand years ago, is a game of luck that requires dice - necessary for random number generation - and a method of scorekeeping (such as sticks, pebbles, etc.). A typical game normally composes of two people, however, we took it upon our implementation to allow the number of players allowed to be changed. A player in real-life could essentially play a one-player game with himself, however, then there would be no real competition. Thus, this extraneous case was sufficiently handled and explained

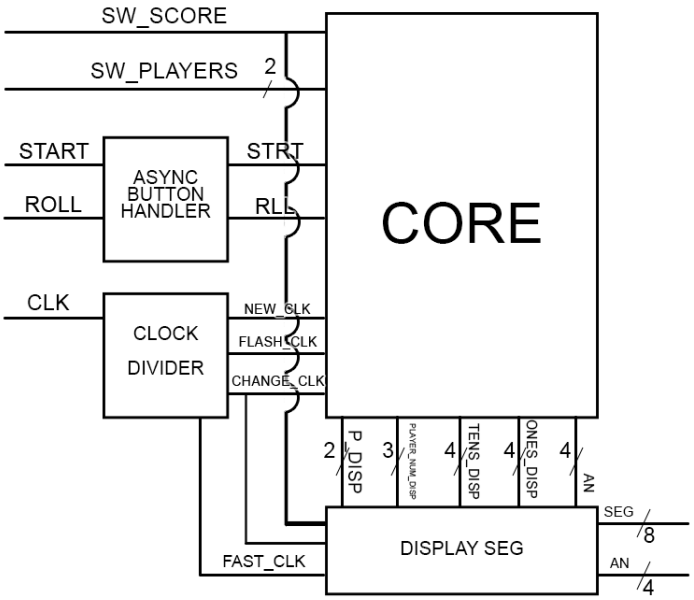
in the later of the report.

Boxed is played with a set of two dice. Each dice consists of the values between 1-6 on each of its faces and the sum of the two rolled dice is a value between 2-12. The rolled values and sum are displayed on the seven-segment display in a intuitive fashion. Additionally, the integrated push buttons are used to start/reset a game and roll the dice. The display will show dice rolling animations accordingly, giving the game a more natural feel, rather than values instantly displaying on roll.

The objective of the game is to be the first player to eliminate all the values of 2-12 contained in your box before the other players remove all their values from their boxes. The sum of the dice (displayed on the seven-segment) will be the removed value. The slider switches are used to designate the number of players to initialize the game with and to also switch from play mode to score mode. In score mode, the seven-segment display will show the number of values left in the current player's box. **See Appendix on the implementation of the seven-segment display and the respective encodings.**

Design

The core implementation of this game was much more tedious than expected. The additional multi-player selection functionality required careful thought and provided a good isolation of nested finite state machines. The design was rooted in that isolation. Although FSMs are not in their nature challenging, it became difficult to manage nested functionality beyond two or three levels. The following figure displays a high-level design diagram of the game's logic:



Each part of the rubric outlines a separate feature that will be a fundamental building block in the creation of the overall game. They are listed as Game Initialization, Roll Functionality, Dice Value Functionality, Reset Functionality, Scoreboard Functionality, and Game-Over Implementation.

Game Initialization & Reset/New Game Functionality

As it wouldn't make sense to play individually, a game can only begin with two or three players. A game is initialized by the press of the reset/start button on the five-button pad. The button implementation is similar to that of previous labs that allow for the down-press to occur at any time. The two left-most slider switches represent the number of players that game should be initialized with. **[00 = 0 players (obviously not allowed), 01 = 1 player (implementation does not allow), 10 = 2 players, 11 = 3 players].**

When the start button is pressed, the display will immediately show that it is in player mode (denoted by P), the first player's turn (1), and zeros in the tens and ones place of the seven-segment. If the game is started with player values that are not allowed (ie. one player), an error is displayed. The following diagrams show the two beginning states on initialization:

Error upon initializing with a player value that is not allowed.

Normal initialization of a game.

An additional restriction on game initialization was that the game must be in player-mode when the start button is pressed. If the game began in score-mode, it would lead to an unintuitive design and confusion. Because the first player's turn would not have gone yet, that would mean there are undefined score values. It would be similar to looking at the score of a football match when neither team has begun to play. It has no true meaning. Additionally, if in score-mode, the roll functionality is disabled, thus the player would have to switch to player mode to begin his turn and might believe the disabled roll functionality as a unintended bug.

Beyond restrictions, the initialization sets three registers to 11'b11111111111. Each 1 bit value represents the respective box value (2-12). The initialization to all 1's shows that all the player's values (2-12) are still in its box.

Player 1 Box:

2	3	4	5	6	7	8	9	10	11	12
1	1	1	1	1	1	1	1	1	1	1

The box initialization always resets three registers (boxes) regardless of the number of players in the game to simplify the logic. Furthermore, the seven-segment display is set to P1 00 to represent Player One's turn and that the dice have never been rolled yet.

We came across a few bugs in the initialization of the game. We wrongfully assumed the slider switches wouldn't be changed during the game. Our first implementation checked the number of players designated by the switches on each positive edge of the clock and then followed the respective

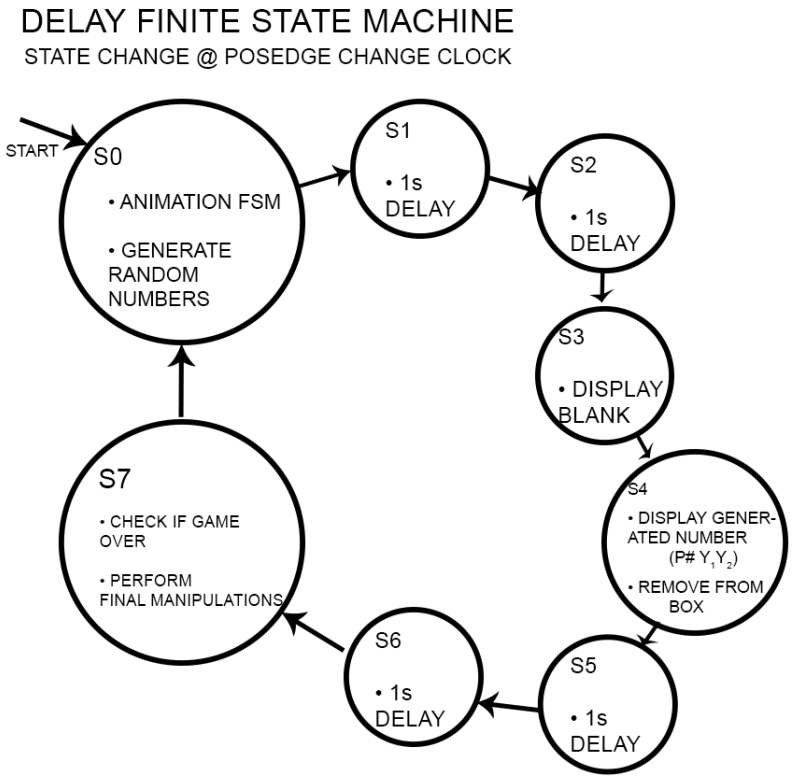
logic. For example, if there were only two players at the start of the game and in the middle of the game the switches changed to three players, the turns and boxes would then have confusing behavior. Technically, the implementation of our game would have allowed a third player to join mid-game (because all boxes are already initialized) but we decided to not allow this functionality. To fix this bug, we stored the number of players on the game start into a new register which would be then used in the game logic rather than the direct input of the slider switches.

Additionally, there were other values that were necessary to be initialized, such as registers storing button presses and the reset of the individual state machines - both of which will be further explained in their respective sections.

Roll Functionality

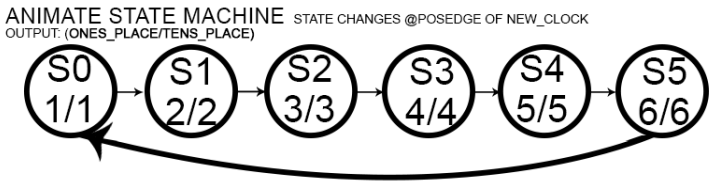
The roll functionality is based off of the button press technique from previous labs. On every positive edge of the main clock, if the roll button is pressed, a register named **rollPressed** is set. Initially, we struggled with how to check for a button press, start a state machine, and did not allow another roll until the next player's turn. The one-bit intermediate register, **rollPressed,** would change to true when the system detected that the roll button was pressed, however, would not change back to 0 when it was depressed. The rollPressed value constantly being true allowed for the iteration through a FSM and at the last state of a player's turn (also when the player's turn changes), the rollPressed register is reset to 0. This then allows for the new detection of a new roll button press for the next player. This sequential logic uses blocking assignment and describes one iteration of the state machine.

This is the state machine diagram:



Upon initialization, **player_turn_counter** is set to 0 for the first state. This state represents the first player's turn. The number of boxes left for that player is then calculated **(View Appendix)**.The player then has the option to either check his score (before roll) or to roll the two dice. Once the player rolls, **rollPressed** is set and the logic enters an additional nested state machine. At this initial state, the display animates, the dice obtain values, the values are displayed, the sum is calculated, and then the sum is displayed. This initial state is described in more detail in the following section.

The following is the state machine of the dice animation:



The changing of states was one of the more difficult aspects to get cor-

rect. As game designers, we wanted to be able to control the duration of how long events occurred. We initially created a normal clock, changed the frequency to a certain specified value (1Hz), and used that positive value to then increment the state machine once every second. The issue with this implementation was that the positive value was held for half a second, then went to negative, and back again. Because the positive value was magnitudes larger than the system clock, the state machine iterated through all its states extremely quickly, rather than staying at a desired state for a specified time. To fix this issue, we modified the clock to a pulse rather than toggle. **(See Appendix)**. The frequency of the pulse was 1Hz, thus each state remained for 1 second. In the next two states, nothing occurs. This acts as a delay to allow the values of the individual dice to be displayed for a few seconds before overriding the display with their sum. In the fourth and fifth states, the display is set to blank for one second and then the sum of the dice is displayed. This simulates a single flash effect. Also in this state, the player's box is updated and sets the value at the index[sum - 2] to zero. Purely setting the value to zero is part of the functionality. It does not matter if the value was already set to zero (meaning it was previously rolled).

To display the sum, we came across a unintended challenge. We could not pass the value to the display directly in one step. Taking the modulo of the sum and passing the respective values to the tens and ones display also did not work. We instead took the sum, ran it through a case statement of 2-12, and updated the ones and tens place display digits accordingly. The following figure shows a part of that implementation:

```
case(sum_of_dice) // display sum in tens_place and ones_place
        2:
            begin
            tens_place <= 0;
            ones_place <= 2;
            end
        3:
            begin
            tens_place <= 0;
            ones_place <= 3;
            end
        4:
            begin
            tens_place <= 0;
            ones_place <= 4;
            end
```

The next two states again act as delay to show the sum of the roll for a few seconds. After the delay, the state changes to its final important state.

The program checks to see if the player has won and the game is over. This functionality is also detailed in a later section. If that player has not won, the internal state machines are reset, allowing the same delay functionality for the next player. Additionally, the display updates to the next player number, zeros are displayed for the dice, and the player's turn increments to the next state. The internal nested states are now followed again for the next player and when all the players have completed their turns, the player turn resets to the first player's state, allowing for a new round to begin.

This concludes the largest aspect of the game.

Dice Value Functionality & Random Number Generation

This functionality seems to overlap with the roll implementation, however, the way of displaying the values is important. After roll occurs, the program handles the game logic and in this module, the values that are calculated are displayed in a way that is fundamentally clear to the user. Initially, the two individual values will be displayed but to make the game seem more realistic, the display will loop through a number generation animation. This required another finite state machine. On a new players turn, new register values (**randOkay** and **randOkayCounter**) are set to 0. The first, **randOkay**, checks to see if the random values should be displayed if its value is less than 5. The second, **randOkayCounter,** acts as the current state inside the value display FSM. An additional pulse (**new_clock**) was created that had a frequency of 5Hz. Thus, every state would change in 0.20s and meaning that five values will be displayed in a second. The total number of values displayed in this dice rolling animation are calculated by **randOkay (5) * [number_of_states_in randOkayCounter] (6) = 30.** 30 different values between 1-6 will be displayed in the six seconds of rolling. At first, we forgot to be thinking in terms of programming hardware leading to both the actual dice value and the animation being displayed simultaneously producing very strange results on the seven segment display. This was fixed when we an additional else clause was added to ensure the rolling animation was finished before showing/generating the actual values.

At the core of this game was random number generation. This was left to one of the final lab sessions because we were testing using strategically picked integer values. Upon viewing the Verilog documentation for ran-

dom number generation, we came across the system functions **$random, $urandom, $random_range**, and **$urandom_range**. These functions produced signed and unsigned values respectively and could also generate random values from a range. In our case we used **$urandom_range(1,6)** to generate a number between 1-6. On synthesizing our project, we found that our seven-segment display of ones and tens digits was not working. After narrowing down the issues, we found that the system built-in functions for random generation only work in SystemVerilog and are not synthesizable. Upon further research we found that random number generation is impossible when synthesizing logic based design. Nothing can be truly random. **See Appendix.**

Although random number generation was impossible, we decided to go with the next best which was pseudo-random number generation. Entire projects were built upon building pseudo-random generators for synthesis, however, all the projects were incredibly complex and beyond our understanding. In order to generate the numbers, we had two options - parse the noise signal from a button press or store an instantaneous value of the master clock. We ran into very technically advanced issues when trying to grab the noise data and generate a value between 1 and 6. Specifically the issue laid in that the 6 values were not a power of 2. Instead, we used the system clock method.

In our core module, we created two 3-bit registers called **random1** and **random2**. Inside the always block, the first thing the program does is increment/decrement our registers between 1-6. This gives us a value at any instance of 1, 2, 3, 4, 5, or 6 in those registers. After the rolling dice animation is completed, the program begins to assign **random1**'s value to the tens place, and directly after that assign **random2**'s value to ones place. Even though we're supposed to be thinking of the program in a hardware sense, there is an incredibly slight delay between the assignment of the two places. Because the system clock runs so incredibly quick, we're able to generate non-related random values. This works due to the imperfections in computing and the actual non-instantaneous execution of instructions (limited by computing power and the speed of electrons not being infinite). In a perfect world, there would be no fluctuation between the time it takes computing **random1** and **random2** thus always being a constant integer value apart. The following shows the implementation of the random number generation:

```
always@(posedge clk)
begin

    if(random1 < 6)
        random1 <= random1 + 1;
    else
        random1 <= 1;

    if(random2 > 0)
        random2 <= random2 - 1;
    else
        random2 <= 6;


            case(random1)
            1:
                tens_die <= 1;
            2:
                tens_die <= 2;
            3:
                tens_die <= 3;
            4:
                tens_die <= 4;
            5:
                tens_die <= 5;
            6:
                tens_die <= 6;
            7:
                tens_die <= 6;
            endcase

            tens_place <= tens_die;


        //Set Display to the Generated Values

        case(random2)
        0:
            ones_die <= 1;
        1:
            ones_die <= 1;
        2:
            ones_die <= 2;
        3:
            ones_die <= 3;
        4:
            ones_die <= 4;
        5:
            ones_die <= 5;
        6:
            ones_die <= 6;
        7:
            ones_die <= 6;
        endcase

        ones_place <= ones_die;
```

Score-mode Functionality

The score-mode functionality was controlled by a single slider switch on the far right of the board. When flipped up, the score of the current player was to be displayed. The mode that the game is in is denoted by the first seven segment display.

In normal game mode, the display shows the letter P, the player's number, and the dice. In score mode, the display shows the letter S, the player's number, and the values remaining in that player's box.

Game Mode - Player 1's Turn (P1 00)

Score Mode - Player 1's Score with 10 Values Remaining (S1 10)

At first, we nested the entire program checking if the switch was set to score mode or game mode. However, on testing the game, the outcome was unexpected. If the player flipped to score mode at the instant the dice were shown but before the sum was removed from the box, what should the score be? Also, we ran into the issue that the finite state machine would be interrupted and that players turn would be repeated when flipping to score mode before the next player's turn began.

To correctly implement this functionality, we allowed the player to check his score before the dice are rolled. This both avoids the interruption of the roll and state machine and is also more intuitive for the user to play. Because the value output to the display is a register, when switching back from score mode to play mode, we had to reset the display to zero. We checked to make sure the player hasn't rolled yet and then if true (meaning just switched out of score mode) we set the display to zero. The implementation of this mode was likely the simplest area of the program due to the value's remaining calculation being a part of a different aspect.

Game Over

The game is finished when one player is able to remove all the values from its box. That means in a perfect situation, a game can be over in 11 rounds, however, the likelihood of that is extremely rare. Due to the combination of the two dice, the numbers 2 and 12 are the most difficult to remove, whereas 7 is the easiest. After every roll, the game checks to see if that player has removed all its values from its box. A winning box is represented by all 0's in an 11-bit structure.

Player 1 Winning Box:

2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0

Instead of iterating through the entire box every roll, a conditional statement checks to see if the box is equal to zero. If there are any values still left, the pattern would have a 1 in it, thus being nonzero.

If the player wins, there will be a two state machine implemented , where the display will make clear that the game is now over. The display will change from the player number (P1 in this case) and the dice to EG P1. This stands for End Game - Player 1 Wins! The display will then iterate forever, flashing between the two states (blank and non-blank) until a new game is started.

State 1 - Blank (to simulate flashing)

State 0 - End Game Player 1 (EG P1)

Testing

The implementation of Boxed contains many finite state machines; some of them are nested FSMs, such as Delay FSM with Dice Animation FSM. Therefore, the testing of our implementation led to the appearance of many bugs - especially as we added new features and states.

The following are the important cases we looked for while physically testing our game. Due to the physical nature of our game, simulation was not a feasible option. For each major addition, we had to ensure both the proper synthesis of code and the correct logic on the board. Beginning at the top most state machine, the Dice Animation FSM looked very simple, however, there was a critical logic flaw in our initial implementation. Being nested in Delay FSM, we couldn't directly control the speed the dice rolling animation and occurred exponentially too fast. The solution was to add a new pulse clock with a frequency of 5Hz, making the state change only moderately

fast. The top state machine, Delay FSM, had problems as well. Initially, we directly used the values of "sw_players" and "btn_roll" for controlling Delay FSM, however, using "sw_players" would have triggered some bugs. As mentioned previously, the value of "sw_players" decides how many players were entered. The value of "sw_players" changing in the middle of the game would cause unintended behavior in the state transition. Therefore, we added an intermediate variable/reg that statically holds the number of player at the beginning of the game.

In addition, the use of "btn_roll" had the similar problem. Delay FSM is basically run by the signal of "btn_roll". As this FSM has seven states – taking at least takes seven seconds to traverse all transitions once – implies the roll button must be at least held for seven seconds to complete one turn. This was obviously not intuitive. Therefore, we again added another intermediate variable/reg to store the signal of "btn_roll" at the beginning of the game. Therefore, the Delay FSM can transition all the states without actually holding the button by the player. Then, at the end of the state, it resets the value of the intermediate reg to zero so that the next player can hit the button to complete his/her turn.

During physical debugging of our implementation on the FPGA Board, we had to go through hundreds, if not thousands, of synthesis and careful testing. To make sure every aspect performed as intended, we uploaded the program via USB. Once uploaded, we would set the desired number of players and press the start button.

We confirmed that an error is displayed when a game is started with unintended values for the number of players. The display output the error code (E - - -). Next, we tested that if good player values are set, nothing initially happens to determine if the number of players is correct. This is intended. No matter the input, the display should reset to unrolled dice for Player 1 (P1 00). For clarity, the rolling generation, dice animation, etc. are skipped in this portion of testing. Once the first player's turn is complete, the state machine should transition to the next player and display P2 00. This worked on the first try without any bugs. Depending on the number of players the game was initialized with, after all players had their turns, the display is reset to Player One and the process is repeated. It took a few times to make sure the correct registers were modified, however, they were simple fixes to make the player turns completely work.

The dice rolling animation was tested purely by looking to ensure the dice showed a rolling animation. It did as intended. However, a much harder aspect to test was the random number generation. The numbers seemed to be generated randomly, however, without an extremely large sample size to

model the probabilities of the numbers, it was very hard to determine if our random generation was accurate. For this game, however, it sufficed with the testing we tried.

Finally, the last aspect to test was determining the winner of a game. It took a while to get a winner due to the dice being random and turns taking a minimum amount of time. But within time, we were able to complete a game with a winner being determined. On win, the display will show an End Game code with the Player Number that won (EG P1). As intended, the display flashes forever continuously until a new game is started.

Conclusion

This project was very time consuming. It tested our abilities to work with new pieces of hardware, design a unique specification, and modify existing design knowledge to fit the project. We had many bugs in our code and fundamental flaws in our game that had to be reworked to fit a stable implementation on the FGPA Board and it took the most time to figure out how to make it work correctly, intuitively, and free of bugs.

It was very satisfying to see our ability to take a old physical game and transform it to a digital version with visuals, number generation, and buttons/switches. As we designed our specification, it seems as if our project was able to correctly achieve all the milestones listed in the revised submission. Initially, our specification did not have dice rolling animations but we updated our project to increase its difficulty with the addition of those aspects. Overall, the project was very rewarding.

Appendix

Seven Segment Display Module Codes

The seven-segment display was implemented in the same manner as in the previous lab. A very fast clock iterated through the four displays, allowing the update of each display at an extremely fast rate, invisible to the human eye. The following is a table of display encodings that were used:

Display Value	8-bit Encoding Sequence
Zero (0)	8'b11000000
One (1)	8'b11111001
Two (2)	8'b10100100
Three (3)	8'b10110000
Four (4)	8'b10011001
Five (5)	8'b10010010
Six (6)	8'b10000010
Seven (7)	8'b11111000
Eight (8)	8'b10000000
Nine (9)	8'b10011000
Letter P	8'b10001100
Letter S	8'b10010010
Letter E	8'b10000110
Letter G	8'b11000010
Dash	8'b10111111
Blank	8'b11111111

Each bit represents a segment of the display. 1's represent the segment to be off where 0 is on. The first is the decimal/dot, followed by the center dash, then around the segments in a clockwise rotation.

Calculation of Number of Values Remaining

To minimize the number of registers necessary in our program, we created a single 11-bit register that represented each player's remaining values. The array index represented the number of the box minus two. For example, the value at index [3] would represent whether or not the player's five has been removed. This technique was helpful because to detect if a player has any values left (has not won yet) their box would have a value greater than zero. Only when all the register's values are set to zero will the entire register be equal to zero.

We ran into trouble because we forgot how to implement the number of boxes remaining when the user flips to score mode. The long register value could not be displayed and even if it was, that would still be unintuitive and require the player to count the numbers of ones. Initially to solve this problem we created another register, initialized it to 11 at the beginning of the game, and decremented it by 1 at each removal from a player's box. This wasn't efficient and duplicated information that was already available through the 11-bit register. Another reason why we abandoned this approach

was due to occurrences of this code segment. Because it decremented the value by one inside the FSM, we ended up with a bug that decremented the value extremely quickly - forcing the term to always be zero - an unintended result. We decided to try a new method.

Instead of keeping track of the two different values, the number of remaining values was calculated at each clock cycle. We know the register must have 11 elements, so we summed up the value at each index which gave the total number of ones in that register. This is equivalent to the number of values remaining. The figure below shows the code:

```
player_1_box_left = player_1_box[0] + player_1_box[1] + player_1_box[2] +
    player_1_box[3] + player_1_box[4] + player_1_box[5] + player_1_box[6] +
    player_1_box[7] + player_1_box[8]  + player_1_box[9] + player_1_box[10];
```

Another method of calculation would have been to create a counter using shifts and OR, however, this approach is much more simple.

Pulse Clock

It was necessary to create a pulsing clock rather than a normal toggle clock for this project. A normal clock cycles between the value of 1 and 0 at the frequency specified. A pulse clock must remain at zero except for one single instant that the value pulses to one. Creating a 1Hz frequency pulse clock would mean in the cycle of one second, the clock will be positive for one instant and negative for the remainder of the clock cycle.

The following shows the code that represents the pulse clock:

```
always @ (posedge master_clk) begin
  if (change_count == 100000000) begin
    change_count <= 0;
    change_clk <= 1;
  end
  else begin
    change_clk <= 0;
    change_count <= change_count + 1;
  end
end
```

The difference of this clock lies in the non-blocking assignment of change_clk. In a normal clock, on true the value would be toggled (set to !change_clk). Instead, on true the value is set to 1, but on else it remains as zero.

True Dice Probability

Because our logic cannot have truly random dice values, the probabilities of rolling certain values are not entirely accurate. In a true dice game with two

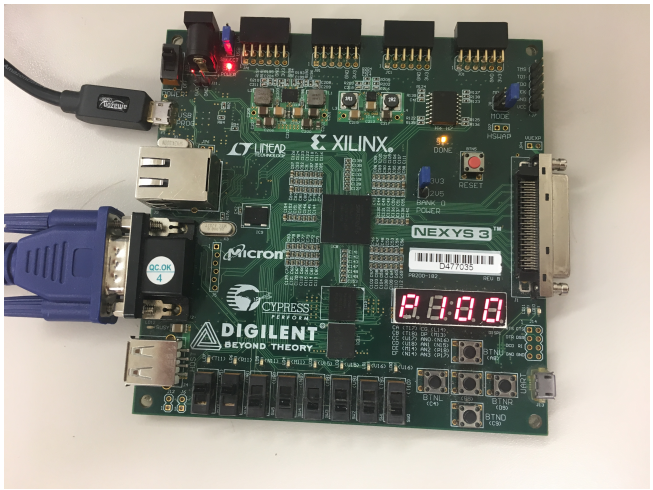
dice being rolled, we calculated the probabilities to be the total ways a value can be create divided by the total combinations of the two dice (36). For example, 2 and 12 can only be created in one way, by rolling double ones or double sixes. However, 3 and 11 can be rolled in two ways, were each dice can either be a one/two or five/six. Seven is the most likely outcome.

The following is a table of the calculated dice sum probabilities:

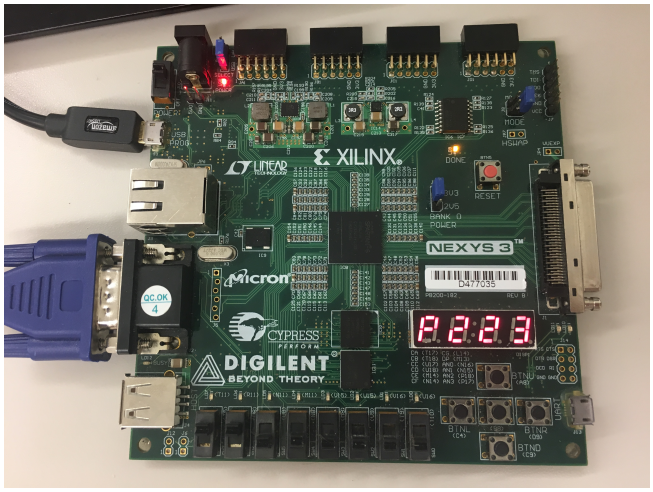
Sum	Combinations	%
2	1	2.78%
3	2	5.56%
4	3	8.33%
5	4	11.11%
6	5	13.89%
7	6	16.67%
8	5	13.89%
9	4	11.11%
10	3	8.33%
11	2	5.56%
12	1	2.78%

Pictures of Physical Board

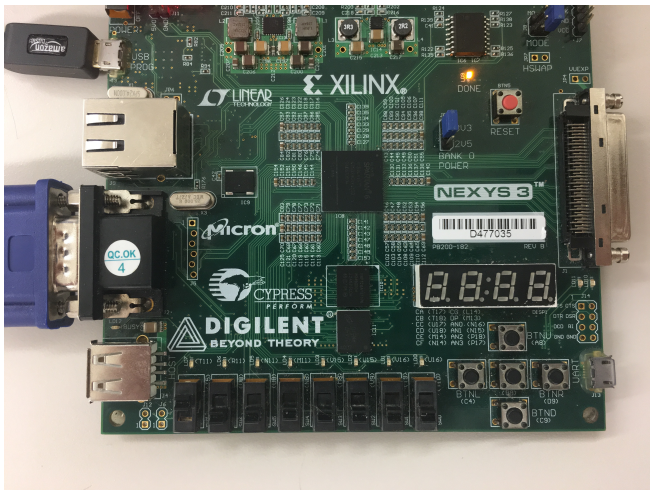
Directly after new game initialization displaying P1 00:



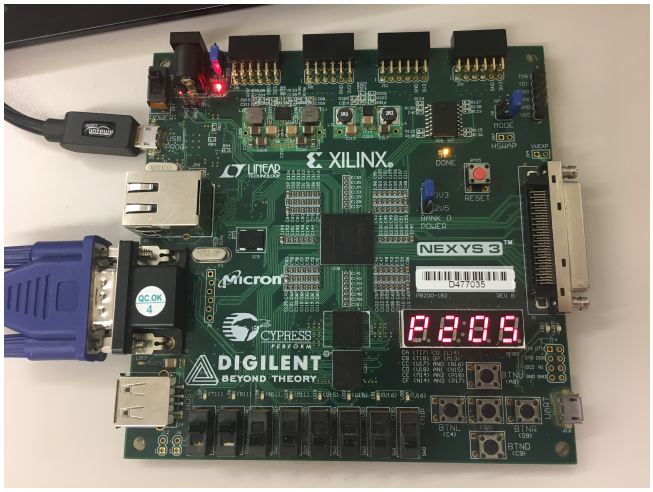
Display after Player Two rolls a 2 and a 3 on their dice:



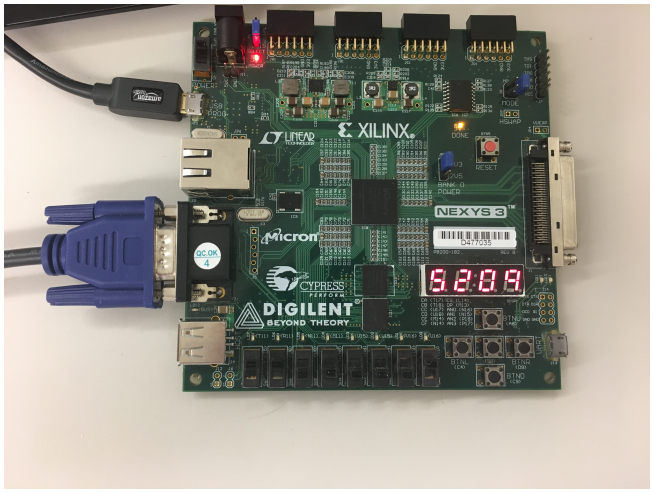
Display in the middle of a flash (blank):



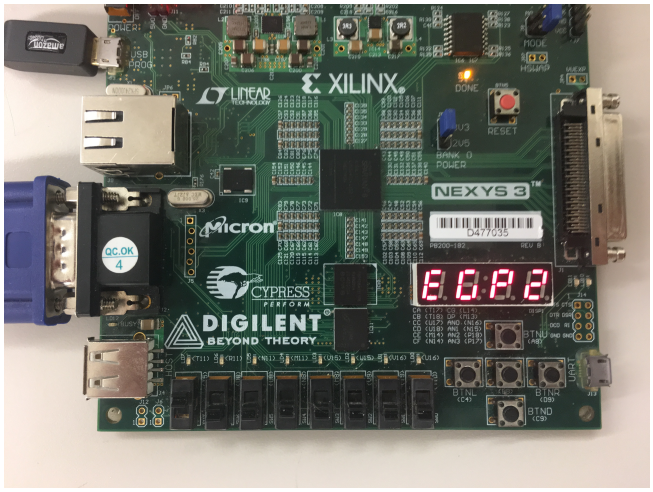
Display show the sum of Player Two's roll of 2 and 3 equalling 5:



Display in Score Mode showing Player Two's Score with 9 values remaining in their box:



Display showing the End Game Screen with Player Two Winning:


XILINX
LINEAR TECHNOLOGY
DONE
RESET
MODE
NEXYS 3
Micron
CYPRESS
DIGILENT
BEYOND THEORY
QC.OK
4