CSM152A
Team 4 (Justin and Krish)
Date: Mar 17, 2024

Lab Report 4: Poker implementation on Basys3

Introduction:

For this project, we created a 2-player Texas Hold'em poker game interface for the Basys3 FPGA module, drawing inspiration from the exciting atmosphere of Las Vegas casino slot machines. Our aim was to recreate the thrill of playing slots, but with a poker twist: using the capabilities of the Basys3 hardware.

To achieve this, we implemented custom card mapping strategies and hand checking functions to ensure smooth gameplay. Through careful experimentation and refinement, we fine-tuned our Verilog implementation to allow both players to draw two random cards from a digital deck. We designed the game interface to display three community cards chosen at random while keeping each player's hand hidden from their opponent, adding an element of strategy and anticipation to the game. Due to limitations in of the basys3 module (size limitations), we modified our game to work with 3 community cards instead of 5 and just a single round, however with better compiling systems and more time, we could have also easily extended it to the 5 community card version.

Players have the freedom to place bets(by entering their bets in binary using the switches), fold their hands if they feel they're at a disadvantage(using the flick of switch 15), and ultimately determine the winner based on the hand rankings of the players currently in the game. We also have added features, such as using the leftmost button to toggle through the current players wallet amount, their total bet for the round, and the total value of the pot. At the end of each round, our hand determining system evaluates the hands of both players, considering factors like flushes and straights to declare a winner. If both players have the same hand ranking(such as a straight or a flush), our code carries out winner determination by determining which player has the highest cards and assigning the winner to them.

Overall, our project offers an engaging and immersive poker experience tailored for the Basys3 platform, providing players with the excitement of casino gaming in a digital setting.

Design:

Our design is made up of five modules, basys3, uart, uart_fifo, uart_top, and poker. The first four modules were based off of the modules we were given in Lab 1. The basys3 module is used for managing the inputs and outputs of our device while the 3 UART modules are used for printing out characters to the terminal. The last module, poker, contains our implementation for a simplified version of Texas hold'em style. Players can play the game by using the center button to advance through the game. First, UART is used to show to the players the 3 community cards for that round as well as the 2 cards in their hands. Next, the betting phase starts and each player is forced to make a bet of 5 dollars. During this phase the terminal will instead display either 1P or 2P to signify whose betting turn it is. During their turn, players can use the left button to toggle the 7-segment display to show different betting information. In order, the 7-segment display will show: the current player's money, player 1's current bet, player 2's current bet, and the pot. The player can also add on to their current bet by setting the switches on the board, treating each as the bits of a binary number. Finally, a player can choose to fold by setting switch 15 in which case they will automatically lose that round. If a player does not match the bet of the other player, their betting turn will repeat until they either fold or place an equal or greater bet. The betting phase will repeat and continue to alternate between the player's turns until both of their bets match. When this is the case, the winner of the hand is determined using the rules of poker. The winner is then displayed on the terminal and an led corresponding to that player is lit up. Finally, the winning player has the pot added to their money and a new round starts.

UART

To display the cards we modified the given UART code from Lab 1, mainly modifying uart_top. First, we changed uart_top to instead receive 3 bytes of data, where each byte would represent a card. For each byte, the upper nibble would represent the value of the card and the lower nibble would represent its suit. We also modified uart_top to read the output to the screen as follows. Step 1, using a function (fnCardValueToName), convert the uppermost nibble of our 3 byte input to a character representing a corresponding card value. Step 2, left shift the input by 4 and read the current uppermost nibble using a different function (fnNib2ASCII) to convert it to a character representing a corresponding card suit. Step 3, output a space. Then, leftshift the input again and repeat these 3 steps 2 more times. Finally, print a newline and a carriage return. We also have uart_top output differently in a few cases. First, if a read nibble is the value 1111, then output a space. This allows us to print out fewer cards in a line, which we use to print out the 2 cards in each player's hand. Also, in the suit and card value functions, there are extra cases which map bit sequences to the characters 'P' and '1' which are used for displaying the current player on the screen. Finally, if the 3rd byte is not all ones (ie. when UART must display 3 cards), then before printing out the cards, print out several newlines in order to clear the screen. This is mainly used to prevent players from seeing each other's cards between turns.

```
function [7:0] fnCardValueToName;
    input [3:0] card_val;
    begin
      case (card_val)
        4'b0000: fnCardValueToName = "2";
        4'b0001: fnCardValueToName = "3";
        4'b0010: fnCardValueToName = "4";
        4'b0011: fnCardValueToName = "5";
        4'b0100: fnCardValueToName = "6";
        4'b0101: fnCardValueToName = "7";
        4'b0110: fnCardValueToName = "8";
        4'b0111: fnCardValueToName = "9";
        4'b1000: fnCardValueToName = "T"; //10
        4'b1001: fnCardValueToName = "J"; //Jack
        4'b1010: fnCardValueToName = "Q"; //Queen
        4'b1011: fnCardValueToName = "K"; //King
        4'b1100: fnCardValueToName = "A"; //Ace
        4'b1110: fnCardValueToName = "1";
        4'b1111:fnCardValueToName = " ";
        default: fnCardValueToName = "?"; //Invalid Value
      endcase
    end
  endfunction
```

Function for mapping bits to character for card value

```
function [7:0] fnNib2ASCII;
    input [3:0] suit;
    begin
      case (suit)
        4'b0000: fnNib2ASCII = "H"; //Hearts
        4'b0001: fnNib2ASCII = "D"; //Diamonds
        4'b0010: fnNib2ASCII = "C"; //Clubs
        4'b0011: fnNib2ASCII = "S"; //Spades
        4'b1110: fnNib2ASCII = "P"; //Player
        4'b1111: fnNib2ASCII = " ";
        default: fnNib2ASCII = "?"; //Invalid Value
      endcase // case (suit)
    end
  endfunction
```

Function for mapping bits to character for suit

7-Segment Display
For our 7-segment display, we mostly reused code from Lab 3. In this design, we have our display take an integer outputted by our poker module (display_value), which it will then display using base 10 representation. This was achieved by using a different set of formulas to set our values for LED_BCD which can be seen in the code below.

```
always @* begin
      case(clk_display)
         2'b00: begin
            an = 4'b0111;
            LED_BCD = (display_value / 1000) % 10;
         end
         2'b01: begin
            an = 4'b1011;
            LED_BCD = (display_value / 100) % 10;
         end
         2'b10: begin
            an = 4'b1101;
            LED_BCD = (display_value / 10) % 10;
         end
         2'b11: begin
            an = 4'b1110;
            LED_BCD = display_value % 10;
         end
      endcase
   end
```

Code for setting LED_BCD for 7-seg

Random Generation of Cards
In order to randomly generate cards for the game, we first created a simple hash function (rand) which would generate an integer using its last output as its input, starting with 0 as its input. To guarantee that this function only returned values in the range of the cards in a deck (0-52) we first set the uppermost bit of the integer to 0, effectively ensuring it was positive, and then took the modulo of the value with 52. However, we also wanted to ensure that in a given round, we do not generate duplicate cards. To do this, we created another function called randcard. This function makes use of a 52-bit register called card_array which represents where each bit represents a card, if that bit is high the card is already taken. The randcard function starts by calling rand to get a random card value storing it in an integer called card. Using card as an index, the next 5 bits after card in card_array are stored in a 5 bit register called card_check. If the next 5

bits reach the end of the array, it wraps around back to the beginning and stores those bits instead. Next the function checks if the value of the bit in card_array at the index of card is high. If it is not, the function simply returns the value of card. If it is, then the value of card must be updated to new value which has not been generated yet. This is done by checking the values stored in the card_check register. Multiple if else statements are used to check the values of each bit in card_check in order. When a bit with a value of 0 is found, then card is updated to the new value represented by that index in check_card and that value is returned. For example, if check_card[1] is found to be 0, then card is set to (card + 2) % 52. If all of the bits in check_card are 1s, then card is simply incremented by 6 with the formula (card + 6) % 52 and returned, which can be done because we are only generating at most 7 cards in a round. This guarantees that we generate random numbers representing each card in a deck without any repeats.

```
integer s = 0;
    function integer rand;
        input integer max;
    begin
        s = (s + 36) * 253;
        s = s ^ ((s * 76) >> 13);
        s = (s + 49) * 17;
        s[31] = 0;
        rand = s % max;
    end
    endfunction
```

Code for hash function

```
if (card_array[card]) begin
          if (!card_check[0])
              card = (card + 1) % 52;
          else if (!card_check[1])
              card = (card + 2) % 52;
          else if (!card_check[2])
              card = (card + 3) % 52;
          else if (!card_check[3])
               card = (card + 4) % 52;
          else if (!card_check[4])
               card = (card + 5) % 52;
          else
              card = (card + 6) % 52;
        end
```

Code for setting “card” to a unique value

Card Convert

As mentioned previously, our implementation has UART read cards a a single byte with the upper 4 bits representing the value and the lower 4 bits representing the suit. In order to convert between our integer representation for a card and the byte value we need for display purposes, we created a function called cardConvert. For this function, we store the integer representation for a card in an input called card. We find the value of the card by calculating card % 13. Similarly we find the suit of the card by calculating card / 13. These two values are stored in two integers, value and suit, respectively. Finally, to get our byte representation, we concatenate the last 4 bits of value with the last 4 bits of suit. We also account for a couple special cases in this function as well. If the function receives -1 as its input, then it will output 11111111. In our uart_top module, this would be represented by printing out spaces. If the function receives a -2 or a -3, then it will output the corresponding bit sequence for displaying “1P” or “2P” respectively.

```
function [7:0] cardConvert;
        input integer card;
        integer value, suit;
        begin
        if (card == -1)
            cardConvert = 8'b11111111;
        else if (card == -2)
            cardConvert = 8'b11101110;
        else if (card == -3)
            cardConvert = 8'b00001110;
        else begin
            value = card % 13;
            suit = card / 13;
            cardConvert = {value[3:0], suit[3:0]};
        end

        end
    endfunction
```

Function for converting cards from integer representation to byte representation

Display Value

Our poker module controls what value is shown on the 7-segment display, which is controlled by altering the value of the integer display_value. First, we figure out what information the display should show using the integer display_state. To control display_state, we have it increment its value by 1 whenever the left button is pressed (represented by the variable display_toggle). We also modulo the display_state by 4 so that it only has a range from 0-3 and wraps around. Finally, we also update display_state if the center button is pressed (represented with the variable valid). If valid is true, then display_state will be set back to 0. As the center button is also used for advancing through turns, this ensures that the display is reset every time a new turn starts. To update display_value, we have its value set in a case statement based on the value of display_state. If display_state is 0, then display_value will be set to the corresponding player's money depending on the player variable. If display_state is 1, then display_value is set to player 1's total bet. If it is currently player 1's betting turn however, it is instead set to their total bet plus the current bet they have added on (represented by the variable temp). If display_state is 2, then a similar case happens to case 1, but now player 2's betting information is used. Finally, if display_state is 3, display_value is set to the value of the pot.

```
case (display_state)
        0: begin
         if (player != -1)
             display_value = player == 0 ? money_p1 : money_p2;
         else
             display_value = 0;
        end
        1: display_value = player == 0 ? temp: p1_total_bet;
        2: display_value = player == 1 ? temp: p2_total_bet;
        3: display_value = pot;
    endcase
```
Code for determining display_value

The Game
In order to manage the different turns and phases of the poker game, we created an integer called rndStart which would store what state the current round was in. We then put our code for updating the game in an Always @ (posedge valid) block, which would cause the game to update everytime the center button was pressed (represented by valid). At the end of this block, we increment rndStart, so that whenever the button is pressed, the next state of the game will occur. At the beginning of the block, we check if rndStart is equal to 10 and if so, we set it back to 0 and reset several other variables. This is used to reset our variables at the beginning of a new round. After this, we check if rndStart is 0. If so, then begin the process of choosing new cards for the round. First we set card_array to 0 so that any cards chosen before can now be chosen again (think reshuffling the deck). Then, we set the variables p10, p11, p20, p21, c1, c2, and c3 to random cards using the randCard function. These variables are used to represent the cards of player 1, player 2, and the community pile.

```
if (rndStart == 10) begin
            rndStart = 0;
            pot = 0;
            p1_score = 20;
            p2_score = 20;
            card_array = 0;
            winner = -1;
            initialize = 0;
            _led[3:0] <= 4'b0000;
    end

    if(rndStart == 0 && initialize == 0) begin

      card_array = 0;

      p10 = randcard(s);
      p11 = randcard(s);
      p20 = randcard(s);
      p21 = randcard(s);
      c1 = randcard(s);
      c2 = randcard(s);
      c3 = randcard(s);

      initialize = 1;
    end
```

Code for initial checks done when a round restarts, values are reset and cards are shuffled

Following these initial checks, we now perform the main logic for each phase of the game. Which phase is currently occurring is decided by a case statement using rndStart. The logic for the phases of the game are as follows:

0) The values of currp1, currp2, and currp3 are set equal to c1, c2, and c3. The currp variables are used to determine what data is sent to UART. In this case, this will cause UART to display the community cards first. The variable player is also set to -1, signifying it is neither player's betting turn.

1) The values of currp1, currp2, and currp3 are set equal to p10, p11, and -1. This will cause UART to print out the two cards player 1 has in their hand and a empty space at the end. Player is set to -1.

2) Same as 0. This will cause the community cards to display again and clear the screen in preparation for player 2's turn.

3) The values of currp1, currp2, and currp3 are set equal to p20, p21, and -1. This will cause player 2's hand to be shown. Player is set to -1.

4) The values of currp1, currp2, and currp3 are set equal to -1, -1, and -1. This will cause an empty line to be printed. This is because this phase is the preparation for the betting sequence. In this phase, the players total bets, money, and the pot are updated so that each player has placed a forced bet of 5. Player is set to -1.

5) The values of currp1, currp2, and currp3 are set equal to -2, -2, and -2. This will print "1P 1P 1P" to the screen, signifying it is player 1's betting turn. On this phase, player is set to 0 to instruct the 7-segment display to show player 1's info. Also in a separate Always @* block the value of player 1's bet is set to be the same as the value represented by the flipped switches. This allows the player's bet to update in real time.

```
always @* begin
    if (player == 0) begin
      bet_player1 = _sw[15:0];
      temp = p1_total_bet + bet_player1[14:0];
    end else if (player == 1) begin
      bet_player2 = _sw[15:0];
      temp = p2_total_bet + bet_player2[14:0];
```

Code for setting the bet of the current player

6) Player 1's bet is checked. The 15th bit of bet_player1 is checked to see if the player set the 15th switch signifying a fold. If this is the case, the pot is added to player 2's money, an led representing that player 2 has 1 is set, "2P 2P 2P" is displayed on the terminal, and rndStart is set to 9 to restart the round. Next we check if player 1's money is lower than their bet. If so, rndStart is decremented by 2, which will cause phase 5 to start again. Otherwise, we add player 1's bet to the pot, subtract their bet from there money, and add their bet to their total bet. Finally, we check if player 1's total bet is less than player 2's total bet. If so we decrement rndStart by 2 to restart phase 5.

```
if (bet_player1[15] == 1) begin
                money_p2 = money_p2 + pot;
                _led[3:0] <= 4'b0010;
                currp1 = -3;
                currp2 = -3;//displays p2
                currp3 = -3;
```

```
                    rndStart = 9;
                end
                else if(money_p1 < bet_player1)begin
                    rndStart = rndStart - 2;
                end
                else begin

                    pot = pot + bet_player1;
                    money_p1 = money_p1 - bet_player1;
                    p1_total_bet = p1_total_bet + bet_player1;
                    if (p1_total_bet < p2_total_bet)begin
                        rndStart = rndStart - 2;
                    end

                end
```
Code for player 1's betting turn

7) "2P 2P 2P" is displayed to signify player 2's betting turn and player is set to 1. Also, player 2's bet is set during this phase

8) Betting logic similar to phase 6 is performed, but using player 2's betting values. This phase also checks if player 2's bet amount was higher than player 1's and if so, the rndStart is set to 4 so that phase 5 repeats and player 1 must bet again.

9) During this phase the winner determination is performed and the integer winner is set to signify who won the hand. The winning player is then displayed on the screen and the pot is added to their money.

After this phase, rndStart is incremented one more time and the initial check at the beginning of the block will reset rndStart to 0 when the button is pressed again.

Outside of the block, currp1, currp2, and currp3 are passed also through the cardConvert function and concatenated. This is then stored inside a variable called playerout which is then passed to UART to display the values.

Winner Determination
Winner determination was done using several hand calculation functions, which took the player’s cards and the community cards, and would give an output based on whether the condition was met, and the ranking of the highest card in that sequence. By default, if a set of 5 cards does not meet the criteria for a ranking, the output is set to 20. If it is positive for it, it will instead return the highest card out of them (or the card value, for instance if the cards are 5C 4H 5D 5S AS, the function 3 of a kind would return 3 (which is the value mapping to a card labeled 5).

An example of how the three-of-a-kind function works is as follows: it first sorts all of the cards based on their value using a bubble sort algorithm. This essentially places all the cards with similar values together. The function returns the value of the card if the first three, middle three, or last three cards have the same value. Else, it would return a 20.

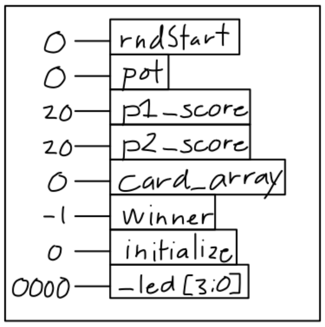
Another function would be flush. This calculates if a flush exists by dividing the card numbers by 13, which would give the suit, and check if all of the values are equal. If they are, it returns the value of the highest card, else returns a 20).

There are a set of 10 nested if blocks that are used, and each of these uses a separate function to determine the “score” of the player. It first checks whether a royal flush exists. If both are negative for it (equal 20), then it does another check to see if a straight flush exists. If it doesn’t, it would check if the player has 4 of a kind. It keeps going deeper and deeper until it finds something of value (returns the highest card number as the score in the last loop).

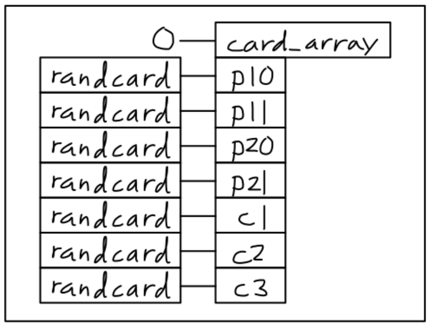
Based on these scores, the winner is determined. If a player has a higher score than the other (excluding values of 20) then that player is the winner. If one of the players has a 20, then the other player is automatically assigned the winner. If both have the same non-20 score, whichever player has the highest card wins the round. Once the winning player is determined, the LED’s light up. LED1 for player 2 , and LED0 for player 1.

Posedge valid

if rndStart==10:

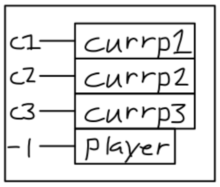


if rnd Start ==0 & Initialize==0

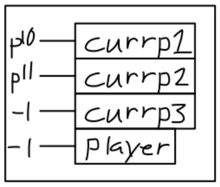


Case(rnd Start)

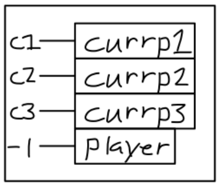
0:



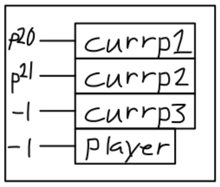
1:



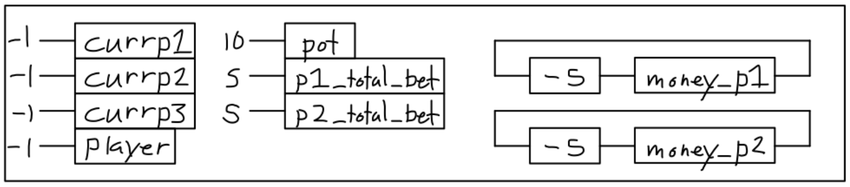
2:



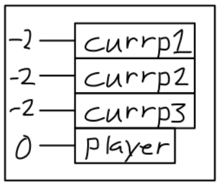
3:



4:

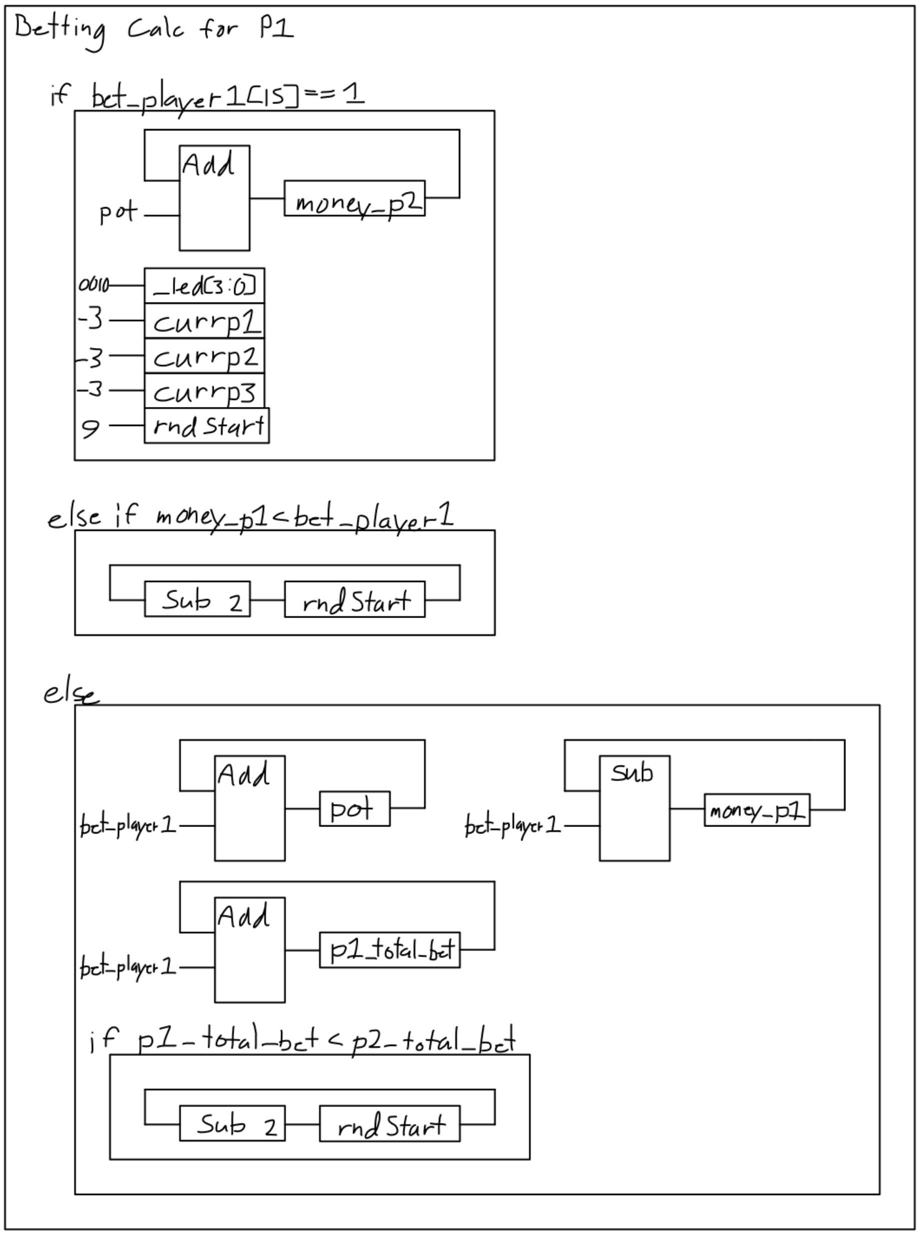


5:

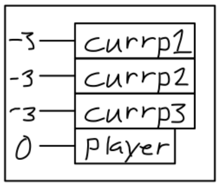


Displays 1P on UART

6:



7:



Displays 2P on UART

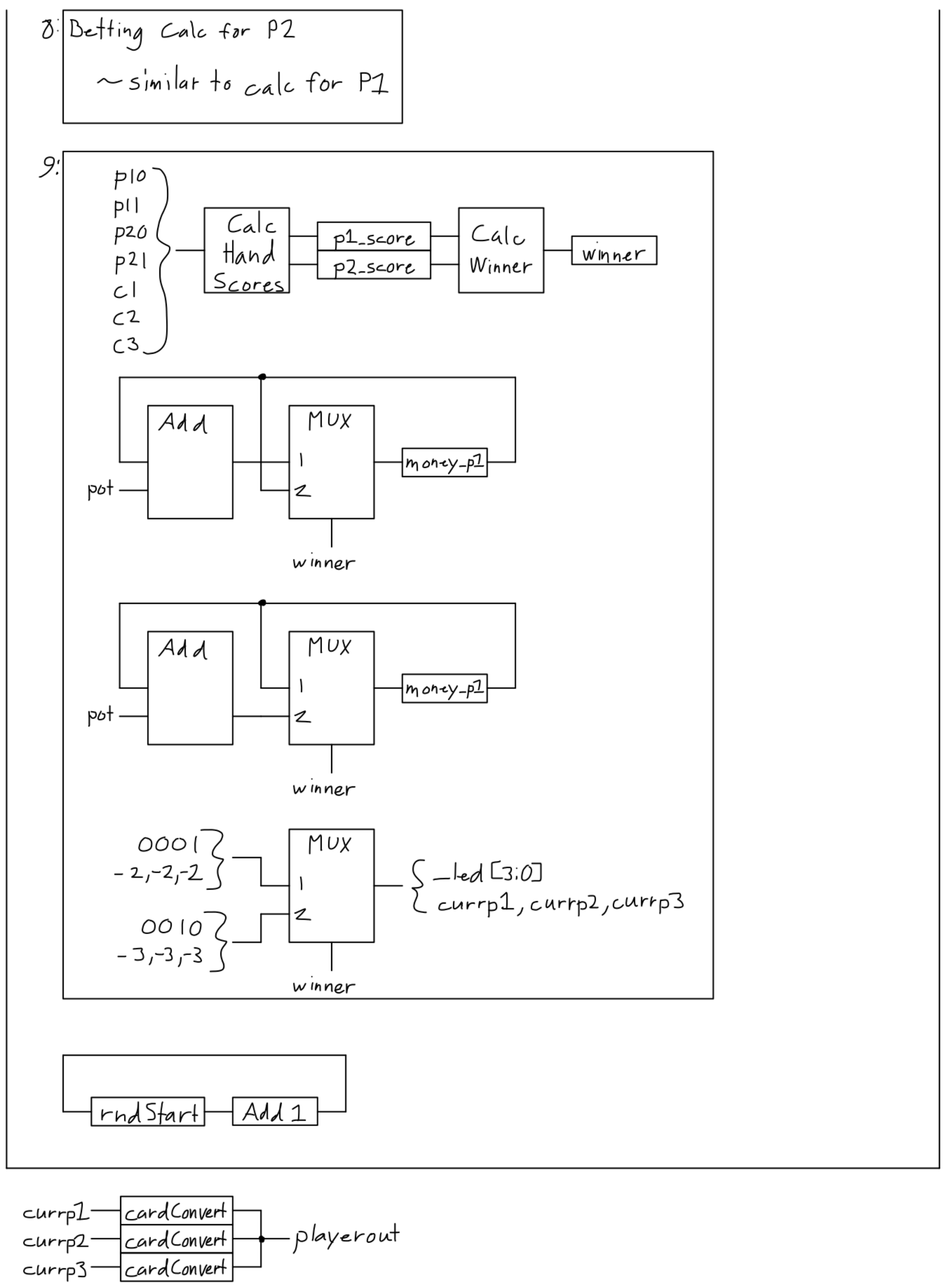


Figure 1. Logical design diagram for managing the turns of the game

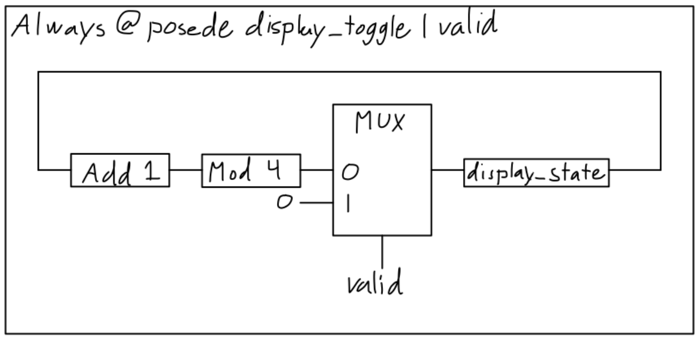


Figure 2. Logical design diagram for incrementing display_toggle

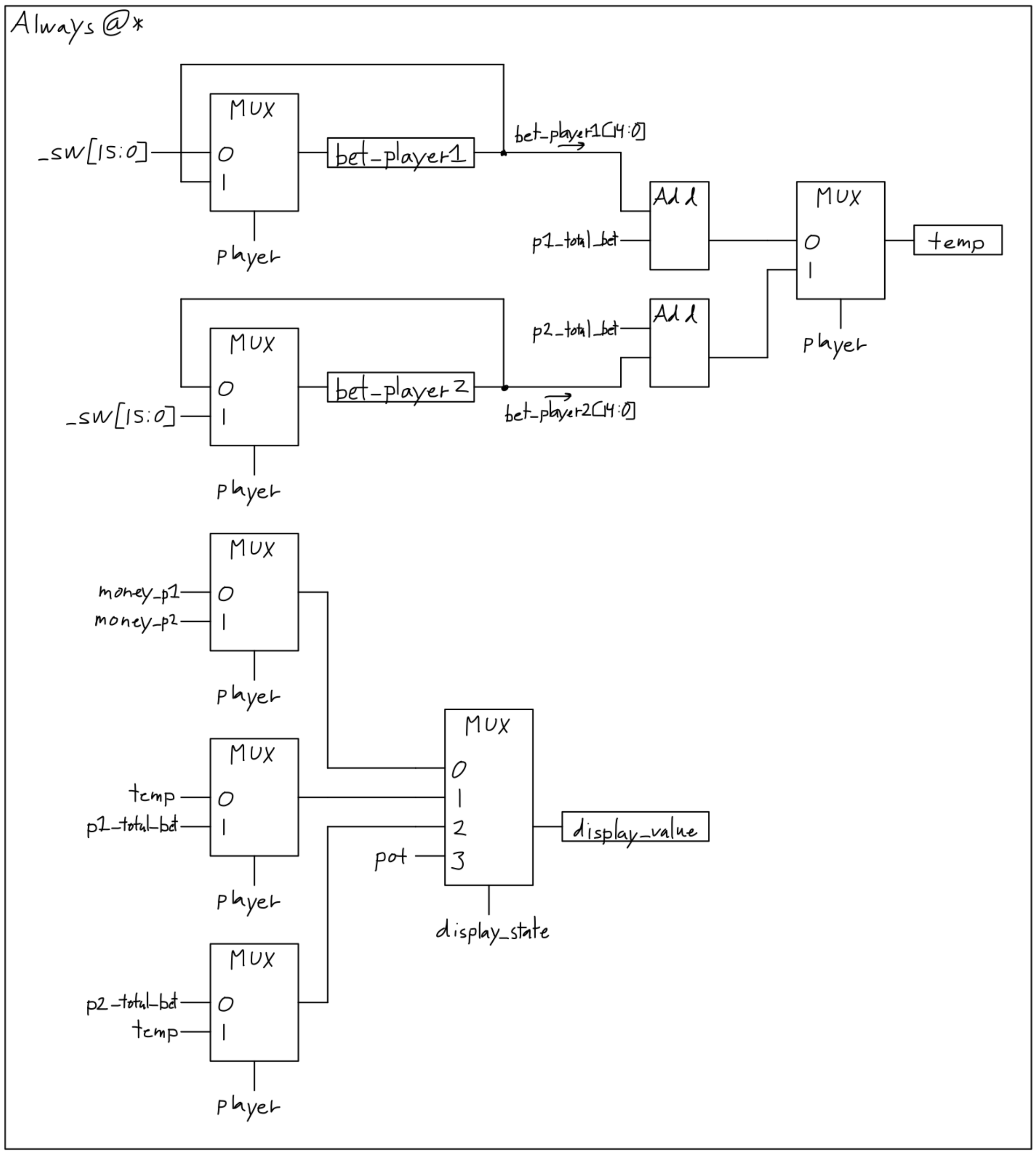


Figure 3. Logical design diagram for setting value shown on 7-segment display

Simulation and test cases:

For simulation and test case purposes, we came up with a range of test cases. As our synthesis was taking upwards of 3 minutes, we relied on the random card generating function to provide outputs, and determined the winning strategy based on our own poker experiences. Using this method, we were able to test out our functions for two of a kind, two pair, full house, flush, three of a kind, and high-card-number. However, as royal flush, straight flush were rare to come by, we manually assigned player1, player2 and the three community cards to see if the correct winner was determined. This was done using our custom mappings, where the value of the card was determined as (cardinteger % 13 + 2), and suit determined as cardinteger / 13. By using this mapping strategy, we chose our test cases such as

P1 : AH KH
P2: 3D 6C
Community cards: QH TH JH

This should give a royal flush for p1, and led to player 1 getting assigned the winner the pot value incremented to p1's wallet amount.

P1: 3D 6C
P2 : 10H 5H
Community cards: QH TH JH

The above card combinations should give a flush to p2, and the correct winner for this round was determined correctly by the system.

For the betting system, we came up with edge cases for different rounds, such as the player betting an amount larger than what they had in their wallet. In this scenario, the player was prompted to add a valid bet again, and this looping occurred till a valid bet was placed.

```
P1 bet : 10
P2 bet : 105  //asked to bet again
P2 bet : 101 //asked to bet again
P2 bet : 15   //accepted
P1 bet : 5      //accepted
```

Another test case that we carried out for the betting system was if the player bet a lower amount than what the other player bet in the previous round. In this case, the player was prompted to bet more, and if a value was bet, that would be incremented to the pot. We also constantly checked the display values to see if the values for the player's holdings, pot money, etc were supposed to change and if they changed by the correct amount:

```
P1 holdings : 95
P2 holdings : 95
P1 bet : 10
P2 bet : 7  //asked to bet again
P1 holdings: 85
P2 holdings: 88
P2 bet : 101 //asked to bet again
P2 bet : 8      //accepted
P1 bet : 5       //accepted
```

We also tested if our randomization algorithm was working correctly, by spamming card generation and seeing if there were any repetitions or patterns that were visible in the cards that were being output, however we weren't able to find any.

Issues and Bugs Encountered:
Like every other verilog project, we faced an enormous amount of bugs while trying to get our code to compile and run.

One of the most frustrating issues that we encountered was from our random card generating function. Somehow, our card values kept changing even after displaying the same cards again and again, and we couldn't figure out the reasoning for it. Sometimes, the cards would be off by a single value or suit, and in other scenarios it would display garbage values. When manually assigning our cards to player out, we weren't running into the issue. Through intense debugging and multiple hours of waiting for the code to compile, we figured out that the issue was pertaining to the assignment of playerout, which was not an atomic function. Thus, we first assigned the outputs of the individual cards to a playerout placeholder in an always @ block, and this was then used to update the playerout.

Printing out our outputs was also a monumental task, as we were essentially working with bits. We thus opted to use lab1's logic for sending outputs to UART, however due to the complexity of the code and the difference in the way outputs were being processed, we spend 2 labs trying to get the outputs to work as intended. We changed the way UART was handling the outputs(using states), made large modifications to them and added more states. However, there were many times when we were getting garbage values due to an incorrect nibble size, an incorrect mapping of output, misaligned bitshifts, etc.

Another issue that we ran into was creating random seeds for the random card generating function. This was a challenge to fix as we couldn’t confirm our outputs due to the above issue. We initially thought of using clock value to generate the seeds based on when the button was clicked(giving completely random values), however we instead went ahead with a hashing function to randomize card generation.

Due to the enormous compile time of the code, the project took a lot more time to debug and resolve issues. However, despite all of these issues that we faced, we were able to generate a working implementation of poker on our basys3 module successfully

Conclusion:

In conclusion, despite all the hardships, issues, meticulous debugging, and a uncomfortably high compile time, we were able to develop a user friendly, fully functional two player poker game on a basys3 module with extra features such as invalid output validation, random card generation, player hand ranking determination, ability to view different amounts on the digital display etc. Despite the frustrations and setbacks, our perseverance paid off in the end, resulting in a robust and enjoyable poker game implementation. We're proud of what we've accomplished and grateful for the opportunity to apply our knowledge and skills to a real-world project.