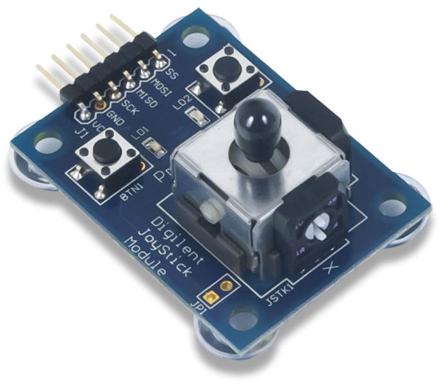
Members: Rahul Mallick
Eric Choi
Sahiti Gabrani

CS M152A Project: Slot Machine

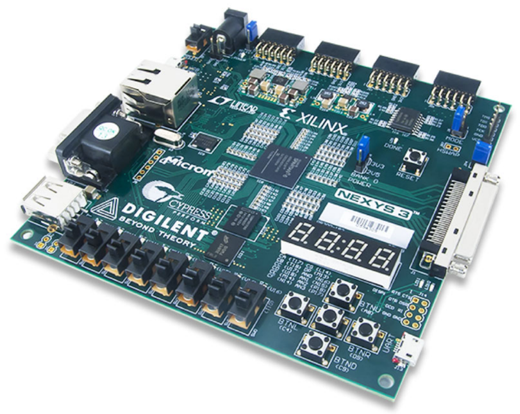
Group 8

Introduction and Requirement

For this project, we created a slot machine using a Pmod JSTK Digilent Joystick Module, a Pmodkypd 16-button Keypad, and a Nexys 3 Spartan-6 FPGA Trainer Board. The user should be able to bet the desired amount (ensuring compliance with the restrictions) in the slot machine game using the 16-button Keypad. This value should be stored and later used to calculate the amount the player has won or lost. The player should be able to simulate a slot machine pull by moving the Digilent Joystick Module (the joystick) in any direction. To ensure the joystick has been pulled, the first 3 LED Lights on the Nexys Spartan Board will flash. The Seven-Segment Display on the Nexys FPGA Board should be used to display the random number which has been generated after the joystick has been pulled. The Seven Segment Display should also be used to display the total amount of money the player has at the start of the game, and after the random number generation (updated according to the slot machine rules). The game logic should be sound, the total amount of money should increment by a factor of the amount they bet if the random number simulator gives “777”, if not it should decrement by a factor of the amount bet. To ensure the slot machine is practical and realistic, the following table shows the technical modules and features successfully implemented:

Technical Module	Explanation
Pmod JSTK Digilent Joystick Module 	We used the Digilent Joystick Module to simulate a slot machine "pull". In a technical aspect, we provided randomness everytime the FPGA board receives a signal that the joystick has been moved. We did not necessarily care about what direction the joystick moves at, but just consider an action as a pull whenever the joystick moves at any direction. This also triggered the LED Lights on the Nexys Spartan Board to flash indicating that the joystick has been pulled and the random number is being generated.
Pmod KYPD 16-button Keypad	We used the 16-button Keypad in order to allow the player to bet in the slot machine

Members: Rahul Mallick
Eric Choi
Sahiti Gabrani

	game. The number has to be between 1 and 9 (included) and the player has to pull the joystick after they input the number to confirm their selection.
Nexys 3 Spartan-6 FPGA Trainer Board (Seven-Segment Display) 	We used the seven-segment display embedded within the Nexys FPGA board to showcase how much money the player has, the slot machine random value after the pull, and the updated amount of money after the player won/lost.

Gameplay

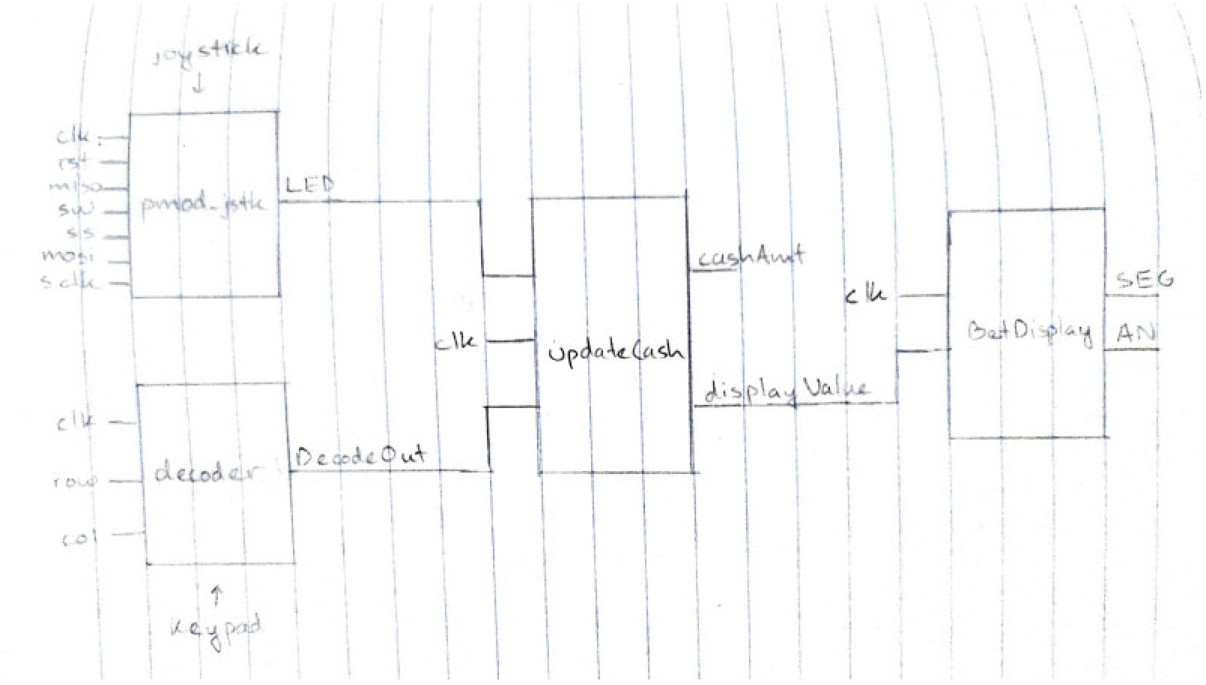
Our gameplay is as follows:

1. The user is prompted with the amount of money they currently have in reserve. In our case, we have it set that the player starts off with $1500.
2. The user is then able to input the amount of money they want to bet between $1 to $9.
3. After they have inputted the money they want to bet, they pull the joystick down to initiate the random number generation.
4. A random number will appear after a couple of seconds, in which one of two scenarios occur:

   a. If the random number is 777, then the user wins a random multiple of the amount they bet.
   b. Otherwise, the user loses a random multiple of the amount they bet.
5. After they see the random number, the user is then prompted with a new amount of money that they currently have, and the gameplay repeats.

Design Description

The following schematic demonstrates a high level overview of our project Slot Machine. The following sections and paragraphs will be explaining each module in greater detail.



Pmod Joystick Modules	
Respective Module Names	PmodJSTK_Demo.v, PmodJSTK.v, spiCtrl.v, spiMode0.v, ClkDiv_66_67kHz.v, ClkDiv_5Hz.v, ssdCtrl.v, Binary_To_BCD.v
The Pmod Joystick communicates with the FPGA board, with a clock speed of 1 MHz, through the SPI communication protocol. In other words, the Pmod Joystick sends 5 byte chunks of information each information exchange sequence; 4 of the 5 bytes correspond to the x and y joystick coordinate data while the last byte communicates about button clicks that	

occur on the joystick module. To be more technical, the Nexys3 board reads MISO inputs during MISO posedges and the Pmod Joystick reads MOSI inputs during MOSI posedges, ultimately allowing the transfer of data.

In order to effectively use the Pmod Joystick, we had to be aware of all the Pmod Joystick inputs/pins as well. Reading the documentation and adapting some sample code that the manufacturers provided, we connected the Pmod Joystick to the modules with the joystick's pin values (in PmodJSTK.v) as so:

1. MISO - Master in slave out
2. MOSI - Master out slave in
3. CLK - 100Mhz onboard clock
4. SCLK - Serial clock

With all this set up, we were able to read data from the Pmod Joystick in binary form and convert this 10 bit representation to a packed binary coded decimal. With this converted value, we could integrate the joystick in our game by evaluating how far the joystick turned towards a certain direction. For our game, because we were trying to replicate a slot machine, we wanted to replicate the "pull" sensation through the joystick. To make it as authentic as possible, we only allow the player to pull in one direction. Within the code, we evaluated whether or not the converted value exceeded a certain threshold; if the joystick reported that it exceeded the threshold, we would tell the FPGA board to blink 3 of the FPGA's LEDs to notify the player that they had successfully committed a pull in the game. The code below demonstrates how we incorporated the joystick data and LEDs:

```
always @(*) begin
   if (bcdData[11:8] >= 4'h7) begin
      LED <= 3'b111;
      brrOutput <= 1'b1;
   end
   else begin
      brrOutput <= 1'b0;
   end
end
```

As the code above demonstrates, if the hundredths place of our bcdData exceeds 7, implying that the joystick is pulled in a magnitude greater than 700, we set the LED output embedded within the FPGA board to all 1s.

Pmod Keypad Module	
Respective Module Name	Decoder.v
The Pmod Keypad Module (formally called PmodKYPD) is a keypad with 16 buttons that can	

Members: Rahul Mallick
Eric Choi
Sahiti Gabrani

integrate with a FPGA board through a 12-pin Pmod port. With a 100 Mhz clock, the keypad decoder evaluates what row and column key got clicked by checking which row and column is set to "high"; by examining the column and row signals, the voltage divider circuit implemented within the Pmod Keypad enables us to determine the pressed key.

Depending on what key got pressed, the decoder is able to translate the pressed key to its four bit representation. In turn, the FPGA board reads in these four bit representations, allowing the game to understand how much the user wants to bet at a specific turn. The following table demonstrates the translation that the Decoder.v module used:

Key Pressed	Output 4 Bit Representation
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Game Mechanic Module	
Respective Module Name	updateCash.v

The updateCash.v module serves as a connecting element, linking all the modules and enforcing the game mechanics of the Slot Machine game. The inputs of this module are the 100 Mhz clock, 4-bit betVal (bet amount received from the Pmod Keypad), and 3-bit LED values. The outputs of this module are a 11-bit cashAmt, 11-bit displayValue, and 1-bit rst (reset) register.

Members: Rahul Mallick
Eric Choi
Sahiti Gabrani

Looking at the gameplay details we placed above, we wanted the player to follow a specific sequence of actions in order to "pull" a slot machine. The ordered steps that a user has to follow is first press a single digit on the keypad (player telling the slot machine how much the player wants to bet) and then physically move the joystick to a specific direction (player "pulling" the slot machine handle indicates to the slot machine that the player confirmed their bet). In the game's end, the game has to display the amount of money the player has (through the 7-segment display), receive the bet amount, receive the joystick pull information, display the number that the slot machine "randomly" generates, and then display the amount of money that the user now has in a specified order. We needed to enforce this specific sequence of actions within the game module. In order to do so, we used two 1-bit registers (booleans) to save the game "state" that a particular player is on; the two 1-bit registers are called hasBet and hasPulled. The code sections below demonstrates the game logic that we implemented with these two 1-bit registers; we added comments next to each code block to specify what each code block does in terms of the game logic:

Game Logic Code Blocks	
Code Block	Explanation

`if (hasPulled == 1'b1 && led != 3'b111) begin`
`    hasPulled = 1'b0;`
`    if (receivedPull == 777) begin`
`        cashAmt = cashAmt + betVal;`
`    end else begin`
`        if (cashAmt - betVal >= 0) begin`
`            cashAmt = cashAmt - betVal;`
`        end`
`    end`
`end`
	This code block gets executed if the player has bet (pressed a key), "pulled" the joystick, and has let go of the joystick. In this stage, the game computes how much the player wins/loses. As the gameplay details says, the player only wins money if the pull is exactly 777; otherwise, the player loses the amount that the player bet. With this code block, the hasPulled boolean is reset to false, effectively restarting the game cycle.

`else if (hasPulled == 1'b1) begin`
`        displayValue <= receivedPull;`
`        hasBet = 1'b0;`
`end`
	This code block gets executed if the player has bet, "pulled" the joystick, but hasn't let go of the joystick. In this stage, the game simply displays the value that the slot machine randomly generates. This code block stops getting executed in the game cycle once the player lets go of the joystick. The hasBet boolean is reset to false in this block, signifying that once the player lets go of the joystick (look at the code

	block above), the whole game will reset back to its initial state.
else if (hasBet == 1'b1 && led == 3'b111) begin hasPulled = 1'b1; end	This code block gets executed if the player has bet and if the joystick gets "pulled". This code block is primarily here to set the hasPulled boolean, which in turn, tells the game logic that the player has "pulled" the slot machine. We use the LED input as an indicator on whether or not the player moves the joystick above our noted threshold.
else if (betVal == 4'h1 || betVal == 4'h2 || betVal == 4'h3 || betVal == 4'h4 || betVal == 4'h5 || betVal == 4'h6 || betVal == 4'h7 || betVal == 4'h8 || betVal == 4'h9) begin hasBet = 1'b1; end	This code block gets executed if the player has pressed a certain key on the Pmod keypad. Similar to the code block above, this code block is primarily here to set the hasBet boolean, which in turn, tells the game logic that the player has bet. If the player presses one of the digits on the keypad (other than 0), we indicate that as a valid bet.
else begin receivedPull = receivedPull + 1; if(receivedPull == 1000) begin receivedPull = 0; end displayValue <= cashAmt; end	This code block gets executed if the game is in its initial state (no bets and no pulls). Here, the module simply displays the amount of money the player has. Since this code block will receive the most attention in terms of clock cycles, we implemented the random number generator here. Essentially, we iterate through 1 to 999 in quick fashion (100 Mhz clock); this allows us to get a "random" value and slot machine "pull".

With these game logic elements, we are able to replicate a slot machine and allow the player to interact with the FPGA board as if the board is a slot machine game.

Seven-Segment Display Module

Members: Rahul Mallick
Eric Choi
Sahiti Gabrani

Respective Module Name	BetDisplay.v

The BetDisplay.v module functions as a module that facilitates the conversion of raw decimal values into the corresponding anode and cathode values for the seven-segment display. The inputs of this module are CLK (100Mhz clock) and 11-bit cash value (raw decimal value) and the outputs of the module are 4-bit AN (anode values) and 7-bit SEG (cathode values).

In order to translate the raw 4 digits of the cash value into separate digits, we separate and translate the number by "modding" the value (shown in the code block below).

```
always @(CNT[1], CNT[0]) begin
  if (CNT == 2'b00) begin
    digit = cash - ((cash / 10) * 10);
  end
  if (CNT == 2'b01) begin
    digit = (cash / 10) - ((cash / 100) * 10);
  end
  if (CNT == 2'b10) begin
    digit = (cash / 100) - ((cash / 1000) * 10);
  end
  if (CNT == 2'b011) begin
    digit = (cash / 1000) - ((cash / 10000) * 10);
  end
end
```

The module also uses a 2-bit counter to iterate through the four digits that need to be illuminated and a 1khz clock divider to refresh and display values to the built-in seven-segment display. An additional decoder is utilized to convert the value of each digit into its corresponding cathode pattern. Moreover, depending on the digit position being evaluated, the module also translates the digit's position into its respective anode pattern.

Modularizing the seven-segment display module was a must, because we utilize the seven-segment display for a wide variety of game states (amount of money the player has, the number the slot machine lands on, etc.).

User Constraint File

Respective Module Name	Nexys3_Master.ucf

To ensure accurate mapping, placement, and timing of inputs and outputs, the slot machine game necessitates precise instructions to direct the FPGA. By utilizing the representative LOC values, the various inputs (such as buttons, joystick inputs, keypad inputs) and the output components (such as the seven-segment display and LEDs) are appropriately mapped within the file. This enables the slot machine game to effectively implement our design and function as a typical slot machine.

Simulation Documentation

Our testing efforts involved both uploading our code to the Nexys3 FPGA Board as well as utilizing testbenches to troubleshoot our problems in simulation. We realized early on that constantly uploading our code onto the FPGA board to test would be time consuming due to the amount of time required to compile our Verilog code before every run. As a result, we resorted to using testbenches to debug problems involving our clock signals, register data, etc. Examining the waveforms allowed us to get a better understanding of whether or not our expected behavior was being exhibited at certain times.

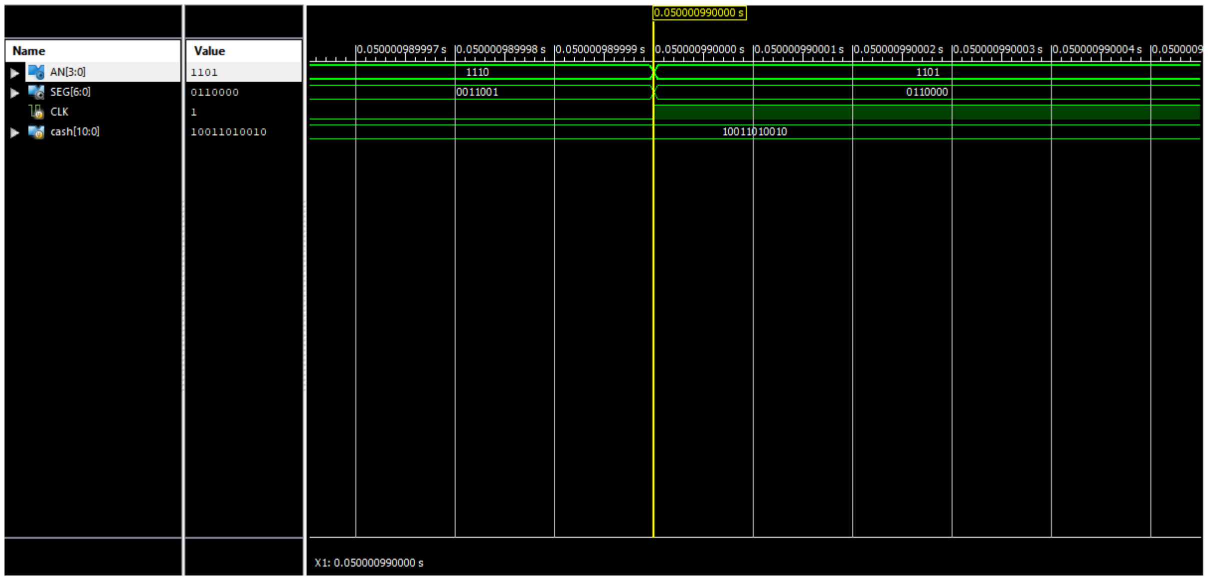
When we first started out with the project, we wanted to ensure that the joystick and keypad input that we would be using were functional. As a result, we utilized the demo code from the manufacturer's site for both pieces of hardware so that it would be easier for us to interface with them. For the keypad, we made sure that whatever value that the user pressed was being displayed onto the seven-segment display. Later on, we would change the implementation such that the value being inputted is not displayed, but rather is used in calculations for the new betting amount. When it came to testing the joystick, we had to modify the demo code from the manufacturer slightly. The demo code outputs an integer between 0 and 1050 when the joystick is pulled, corresponding to the degree to which the joystick is pulled in the y direction. We changed this slightly such that when the value is above a certain threshold (i.e. 700), we would light up certain LEDs. We tested to make sure that this implementation worked on the hardware. Afterwards, we moved on with the actual game logic of our project.

To make it easier for us to see how our gameplay logic worked, we started off with the BetDisplay Verilog module, which is a variation of the display module from the pmod_kypd demo code from the manufacturer. However, the difference was we would want to display the cash as well as the random number that would be generated from our game. This was a relatively easy fix, as we had to adapt the demo code such that it extracted the thousands, hundreds, tens, and ones digit of whatever value we wanted to display. We were running into issues with getting the correct value being displayed, since we couldn't directly use the modulo operator. After playing around with how to extract these digits in the simulation view, we managed to get it to

Members: Rahul Mallick
Eric Choi
Sahiti Gabrani

work for 1, 2, 3 and four digit numbers for our various test cases. The image below reveals the moment when the value for one of the digits is being displayed on the seven segment display.

Bet Display Testbench



Name	Value
AN[3:0]	1101
SEG[6:0]	0110000
CLK	1
cash[10:0]	10011010010

Next up was the actual game logic itself. We separated the game logic into different stages of gameplay: 1) when the user hasn't inputted any money yet 2) when the user has inputted money but hasn't pulled the lever 3) when the user has inputted money and has pulled the lever and 4) when the user has released the lever. In stage 1, the user should see the total amount of money they have. In stage 2, the user should still see the amount of money they have, but in the background, the amount of money they have betted should be stored in a register. In stage 3, a random number should be outputted on the seven segment display. And in stage 4, the new amount of money should be displayed whether a "777" has appeared on the seven segment display.

Since we had these four stages for us to work with, we worked on one before moving on to the next. We utilized the testbench to make sure that we were getting the right values getting stored in the right registers before we went on to upload our code on the FPGA. One of the main issues that we had was dealing with the joystick input, and getting it to display the right value on the seven segment display when the joystick is pulled. We tried different methods of trying to utilize the LED signal that would turn on when the joystick was being pulled. Primarily, we tried to generate a random number when the LED was on, but this was causing problems for us when we

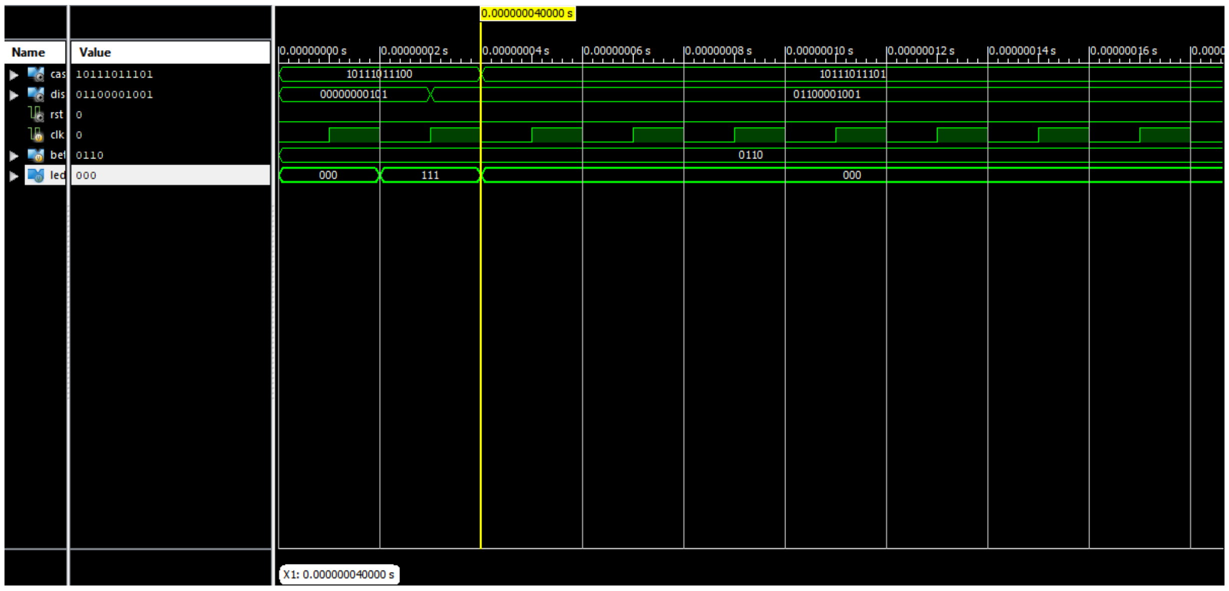
Members: Rahul Mallick
Eric Choi
Sahiti Gabrani

wanted to go back to displaying the total amount of money the user had. After much trial and error, we realized a simple and elegant solution: display the current amount of money after the LED is turned off (when the joystick is let go). This means looking for the moment when LED goes from 111 to 000. In turn, this implied that we would have to hold the joystick to generate a random value, which was a sacrifice we had to make in order to finish on time.

Another issue we had to grapple with was when to use blocking and nonblocking assignments. Using the blocking assignment (=) means that the instruction is being done sequentially, while non-blocking (<=) means the instruction is being done simultaneously. We were initially using "<=" to assign to registers whenever we wanted to change the value of certain variables. However, we realized later on this was causing problems for us in displaying the wrong values at certain times. This is due to the sequential nature of our game, in which we had to use the blocking operator to assign our variables.

The image below shows one of our simulation testing efforts, in which we were making sure that lever/joystick input was being recognized properly by our Verilog code. The moment that the led register switches back to 000, that is when we update the cash in our reserve.

Update Cash Testing



Name	Value
cas	10111011101
dis	01100001001
rst	0
clk	0
bet	0110
led	000

Members: Rahul Mallick
Eric Choi
Sahiti Gabrani

Conclusion

To summarize, starting out for the project, our focus was on getting the Pmod JSTK Digilent Joystick Module and the Pmodkypd 16-button Keypad to individually work with the Nexys 3 Spartan-6 FPGA Trainer Board. Afterwards, we combined the two components to work together and display the correct values on the Seven Segment Display according to the slot machine rules. For the game logic itself, we divided it into distinct stages based on the progress of gameplay: 1) prior to the user entering any money, 2) after the user has entered money but before pulling the lever, 3) after the user has entered money and pulled the lever, and 4) after the user has released the lever. This enabled us to work incrementally and get each component to work correctly. Since we had never worked with the Keypad or Joystick Module before, it took us longer than expected to set them up and make sure they receive input from the user, reading the documentation for these and looking at past StackOverflow questions helped us. In our BetDisplay Verilog module, it took us a few trials and errors until we figured out how to correctly display four digit numbers (by extracting each digit individually) onto our Seven Segment Display. Our biggest issue was generating and displaying the random number which we were able to resolve by holding the joystick to generate a random value, and displaying the updated amount after the player lets go. We also learnt when to use blocking and nonblocking assignments, depending on if we want the instruction to execute sequentially or simultaneously. Through this project, we learned how to work with the 16-button Keypad and the Joystick Module, and increased our proficiency working with the Nexys FPGA Board. We have also realized it’s a lot easier to work on your project if you set small manageable goals and build up from there, essentially breaking down your project into components.