**Lab Report 4**

**Air Hockey**

CS M152A

Lab 5

Nina Maller and Huiying Li

Student id: 604955977 (Nina),   
304958595 (Huiying)

**Introduction and Requirements**

Fort his lab, we decided to make a ping pong game, where each player can move up, down, right and left. After defining this, we realized that we are actually making an air hockey game, except that the paddles are rectangular and not round.

The air hockey game supports 2 players, where each one can control the motion of the paddle with a PMOD joystick. In the game, each player has to hit the ball (the puck) so it will enter the opponent’s goal. Each player can only move in their own half of the board, and the goal is located on the opponent’s half. The purpose of air hockey is to win as many round as you can, with a limit of 9 rounds per player.

The air hockey game has two modes, a game mode, and a score mode. In the game mode, the players can move their joysticks, reset the game, or pause the game. After one of the players won the round, the game automatically switches to the score mode, where both scores are displayed. The person who is using the left joystick (port JA) can see his score by looking at the left number, while the person using the right joystick (port JD) can see her score by looking at the right number. Further, they can find who won the previous round by looking at the colors of the numbers. The score of the player who won is displayed in green, while the score of the person who lost the previous rounds is displayed in red. While on the score mode, the players can adjust the difficulty of the next round by flipping the difficulty switch up in order to make the ball, as well as the players, go faster. They can also flip it back down to return to the slow, default difficulty. Further, the players can also reset the game at this mode, which will result in an automatic switch to the game mode at the default difficulty, and will reset both scores to 0. Lastly, while on the score mode, players can press the continue button, which will start the next round at the difficulty they chose. This loop will continue until one of the player will receive a score of 9, or if they choose to reset.

We built this lab upon the ideas introduced in labs 1 through 3. In lab 3, we learned how to design and implement a stopwatch, which taught us a lot of valuable skills we used in this lab. We already knew how to connect switches and buttons using the debouncer and changing the UCF file. We also knew how to properly design the modules by dividing the task to many small modules, in order to make debugging easier. However, in this lab we also had to think about the idea by ourselves, and this time, we weren’t given the names and the instructions on how to implement each module. Doing this by ourselves was challenging and taught us a lot.

Furthermore, we learned from the stopwatch lab how to read and make sense out of manual pages. This was very helpful in this lab as we had to figure out how to use the VGA display, and the PMOD joysticks. The background information we had to learn before starting the lab can be seen on the next page.

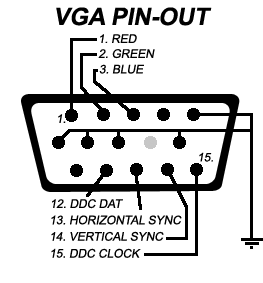
By completing this lab, we are able to think of an idea, design, implement, and debug the program by ourselves using the Nexys™3 Spartan-6 FPGA Board, a skill we are sure will be very valuable in the future.

**Background information:**

**VGA:**

Video Graphics Array (VGA) is a display standard that provides 640 X 480 resolution color display screens with a refresh rate of 60 Hz and 16 colors at a time.

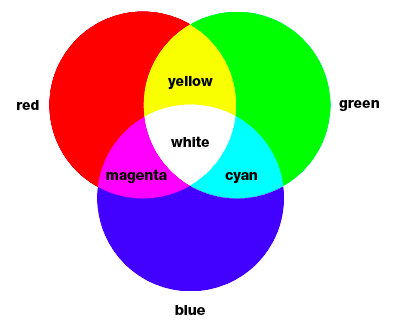
VGA pin functions:



The VGA requires 5 signals in order to display a picture: the red, green, and blue signals, as well as the horizontal and vertical synchronization. By specifying which color should be at each point, we can create an image on screen (as we will explore in the VGA module in the design part of this lab).

The monitor displays a line by line, from top to bottom. It starts from the top left corner, so the positive horizontal is on the right, and the positive vertical count is at the bottom.

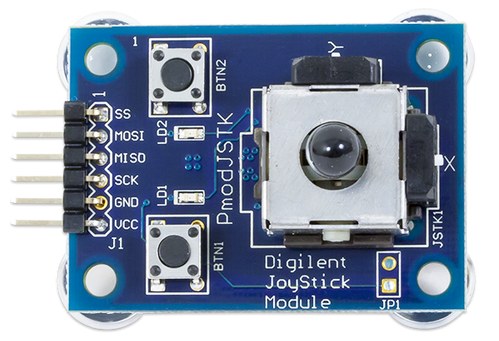
Colors:

For this project, we are going to use 8 bit colors. This means that the red and green values are represented with 3 bits, and the blue color is represented with 2 bits. Using these three colors, the following photo shows all the combination of color we can use:

Black can be achieved by having inputting 0 as all colors. Therefore, 8 colors can be achieved. In the project, we used 5 colors: black, white, blue, cyan, and magenta.

**PMOD Joystick:**

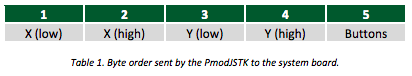
The PMOD joysticks allows the user to interact with the program. In our case, the PMOD joystick will be user to control the paddles. Using the way is it shown in the photo below, by

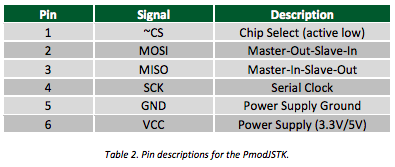


moving the joystick up, down, left, and right, the paddle will move in the same direction as the finger of the player controlling the joystick (notice that the joystick must be held as shown in the photo, where the connectors are at the bottom.

The joystick communicates with the board in 5 byte chunks. The first 4 bytes represent two 10-bit values for the X and Y coordinates, while the 5th byte represents the three buttons. For the purposes of the air hockey game, we only need the first 4 bytes. For the x direction: the lower 8 bits of this 10-bit value will arrive as MSB in the first byte and the remaining two MSB of the 10-bit value will arrive as the last two bits in the second byte. Similarly for the Y coordinate with the 3rd and 4th bytes.

The table below was taken from the PMOD joystick manual, and represents the overall order to bits.



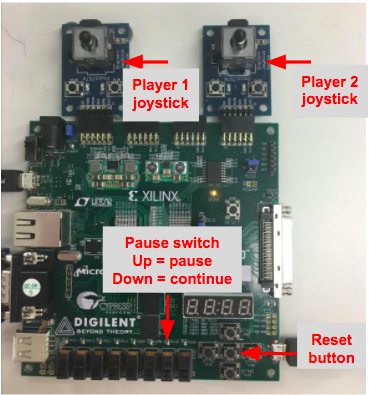
Further, the table below describes all the pins used for the joystick in order to communicate with the FPGA board

**Design Description**

We designed the game to have two modes, a game mode, and score mode. Each mode has buttons that associate with it, that are meaningless in the other mode. Below are the available choices for the user for each mode.

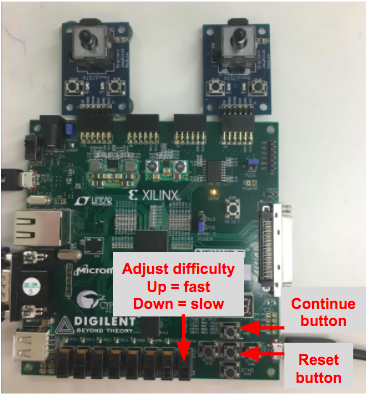
**Game mode:**

In the photo below, it is seen how the user can control the game. The game supports two players, where each one can play using their own joystick. At any time, the players can pause the game by lifting the switch up, and then lowering it down to continue from the same point they left off. They can also reset the game, where both final scores would reset to 0, and the game will start again from the default, easy mode.

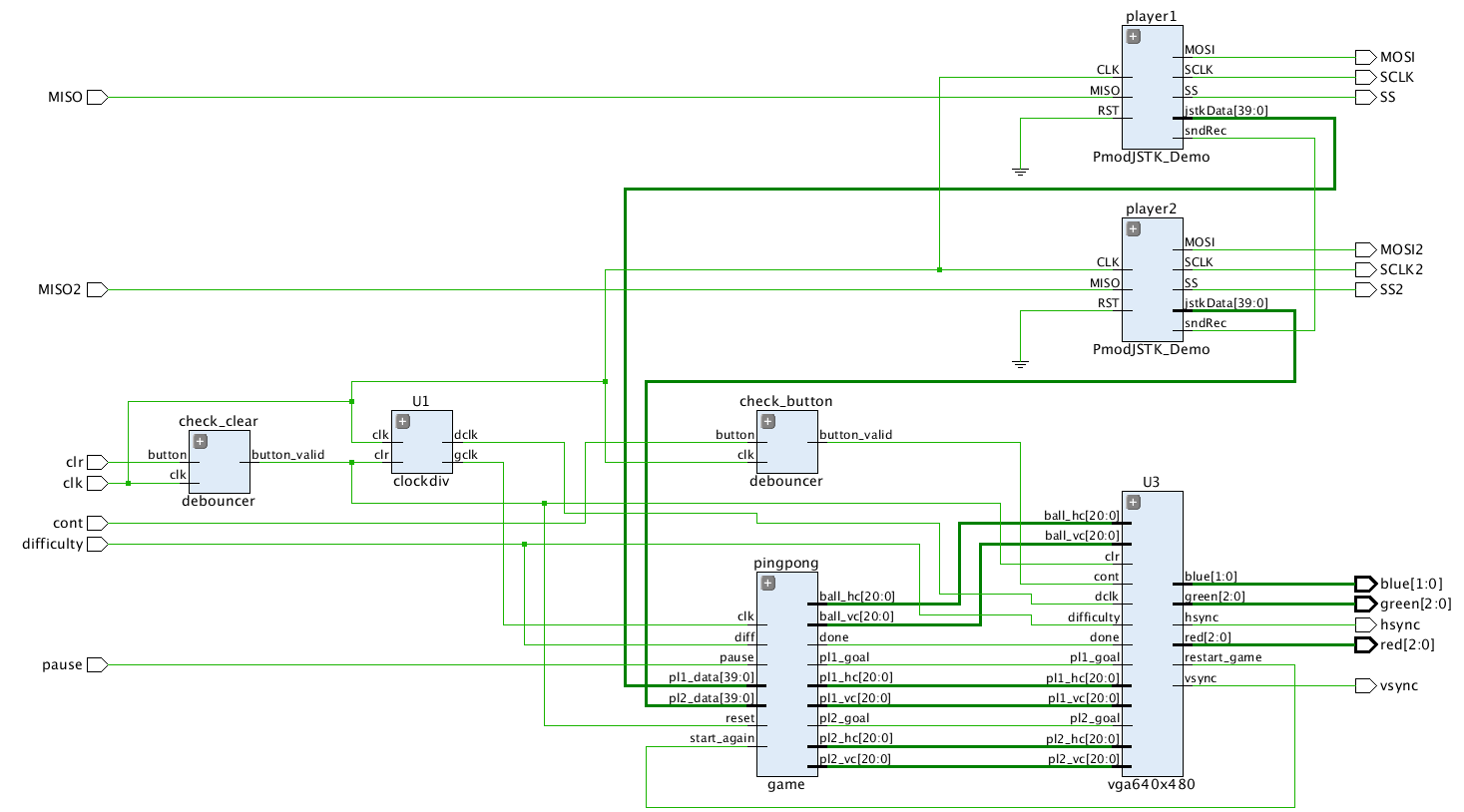


**Score mode:**

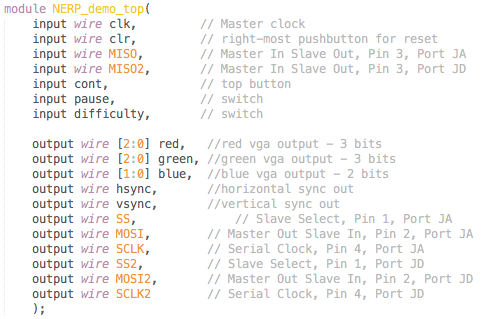
In the score mode, the screen displays the total score of each player, and indicating with a green color which player won the last round (while the other player’s score is displayed in red). While in this mode, the user has the choices shown in the photo below. The user can adjust the difficulty by flipping the switch up in order to make the ball as well as the players go faster, or leave it down. Switching the switch in game mode does not do anything. Further, the users can press continue, in order to continue to the next round, so that the game will keep track of the total scores. If the user clicks on the continue button while in game mode, nothing will happen. They can also reset at any time, in which the game will go to immediately to game mode, and both scores will be reset to 0. Notice that reset is the only button that can be chosen regardless of the mode. See photo of available options on the page below.



In order to design the air hockey game, we broke down the design to many modules, and then connected them in the top module.



In order to explain how the design is connected, we will start from the top module NERP\_demo\_top. The name of the module comes from the demo we used, which will be discussed later in the design (more accurately, we used 2 demos, one for the VGA and one for the joystick. Each demo will be discussed under the corresponding module).

Below is a photo of all inputs and output of the top module.

The input of the game are: the master clock, the reset button (clr), 2 wires that output the movement of the joysticks, which are connected to ports JA and JD, a continue button, a pause switch, and a difficulty switch. For the outputs, we have 5 wires which are required for the VGA display (3 colors and horizontal and vertical distance of each pixel, more information under the VGA module), and 5 output wires for the joystick control.

After defining the inputs and outputs of the top module, we started defining the rest of the modules, adding inputs and output as they became necessary.

Here is an overview of all modules and what they do. A detailed explanation of modules follows shortly after the overview.

Modules:

PmodJSTK\_Demo - handles connection with the PMOD joystick.

Game - has all the code for the game portion

Clockdiv - divides the master clock in order to support the VGA module used.

Debouncer - used to make sure the buttons to continue and reset are valid.

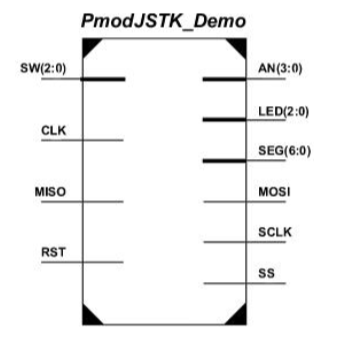
Vga640x480 - displays both modes on the screen using the VGA cable.

Then, we defines 2 instances of the PmodJSTK\_Demo modules, one instance for player 1, and one for player 2. We also defined 2 instances of the deboncer, one for each button. The rest are defined once.

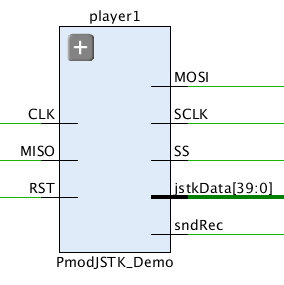
Below is an explanation of each module.

**PmodJSTK\_Demo**

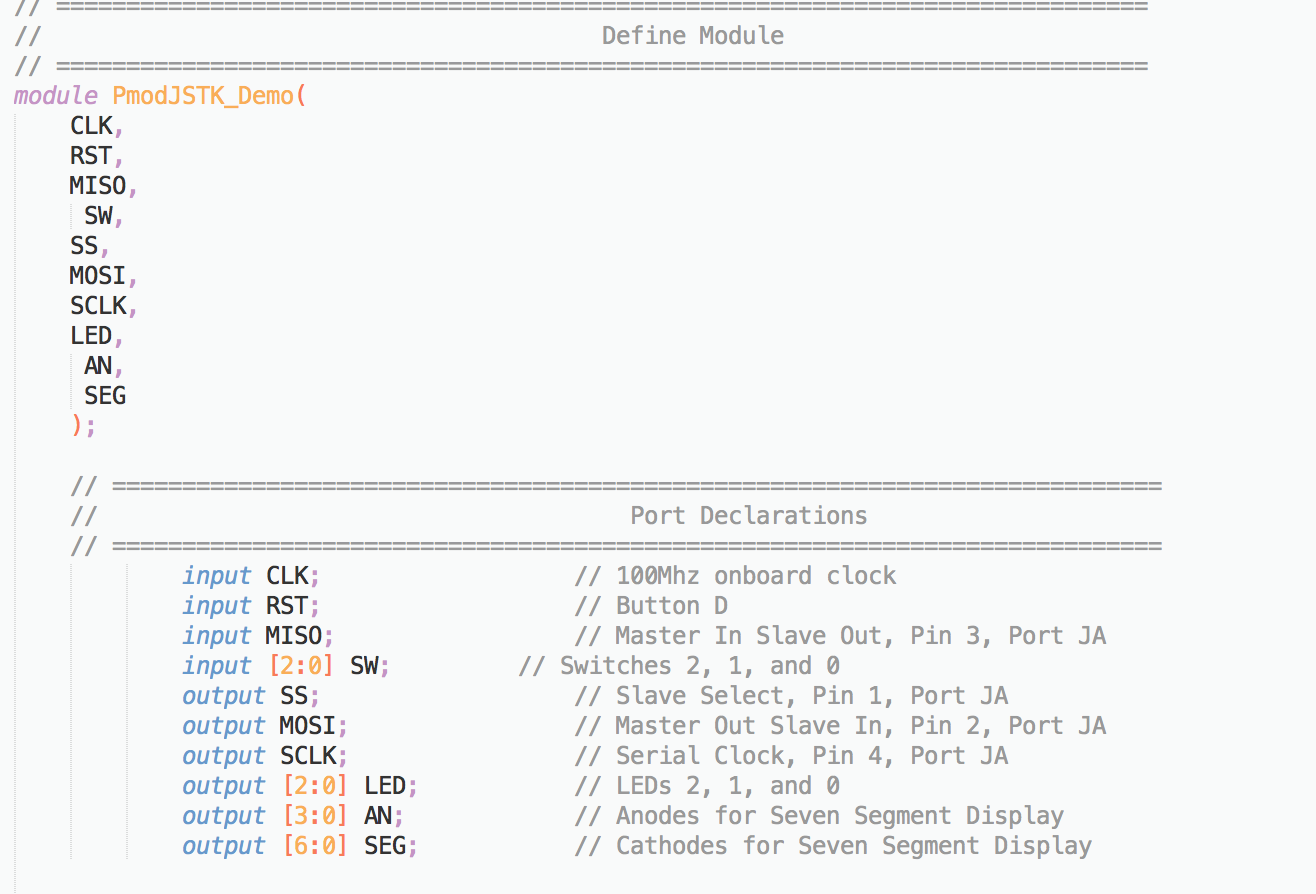
This module is responsible for the communications with the PMOD joystick. Our base for this module is the PmodJSTK\_Demo.

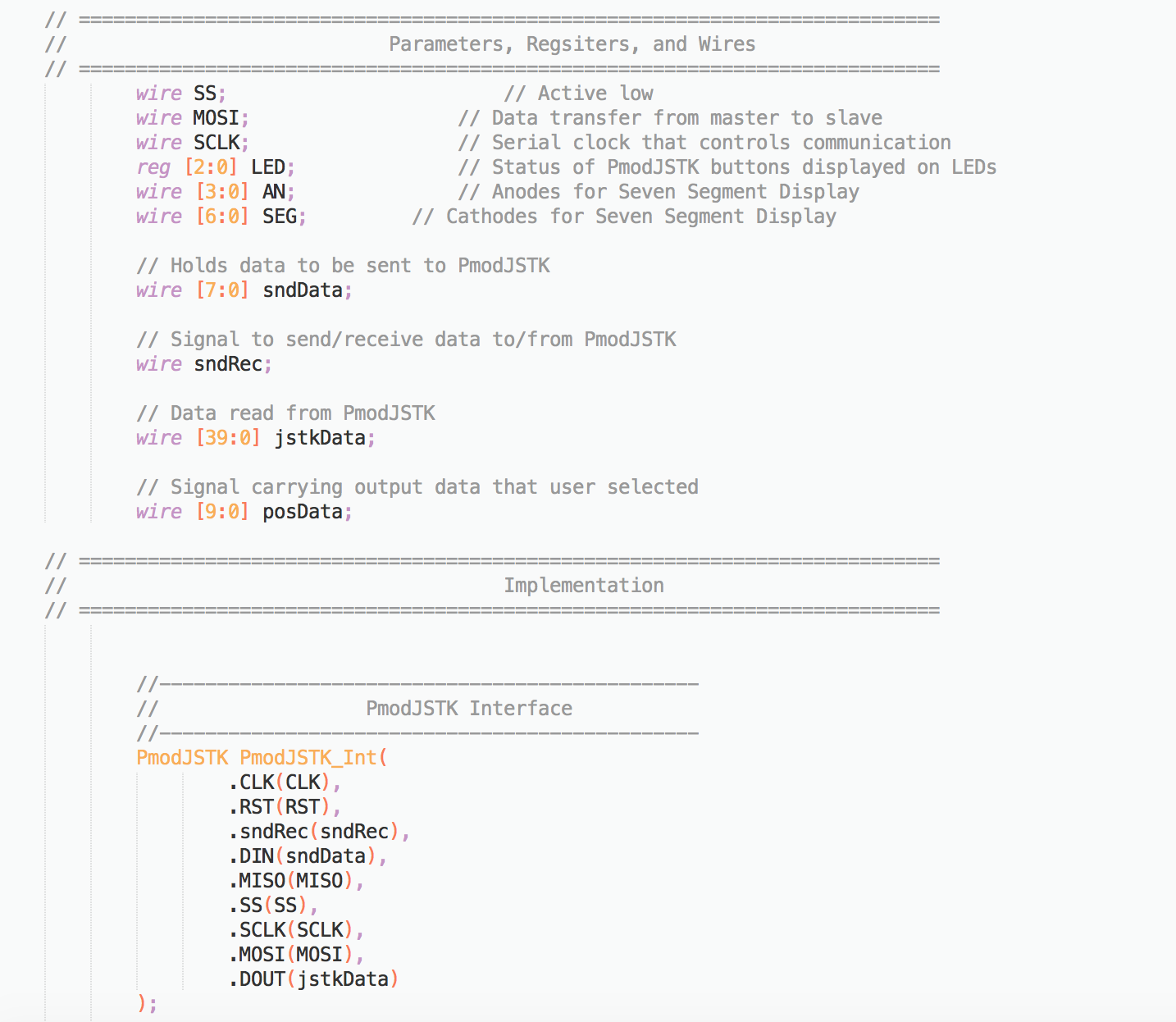


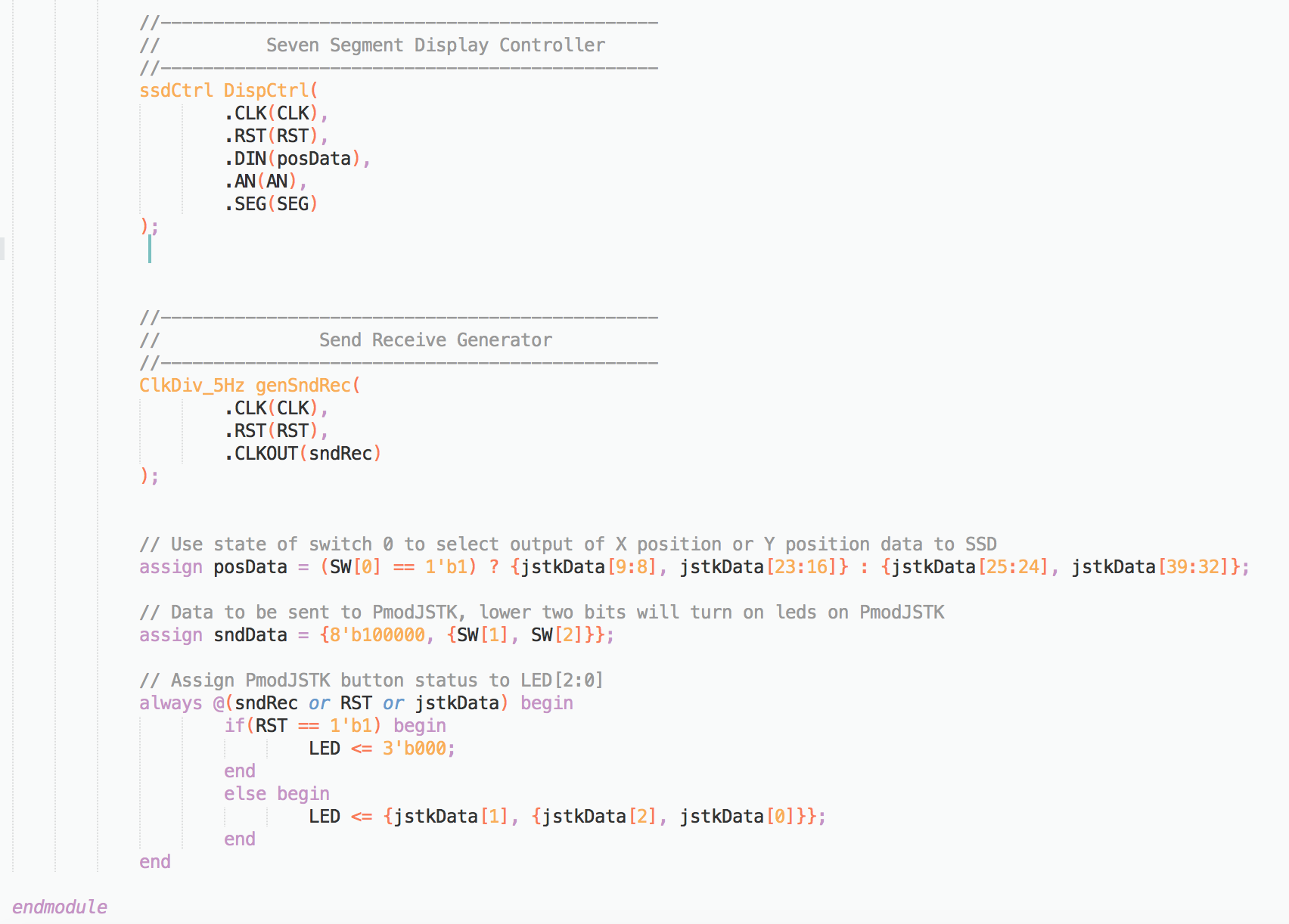
Below is a photo showing the joystick module we wrote. The jstkData is 40 bits, 5 bytes, but we only used the first 4 bytes.



Here is the code from the demo:



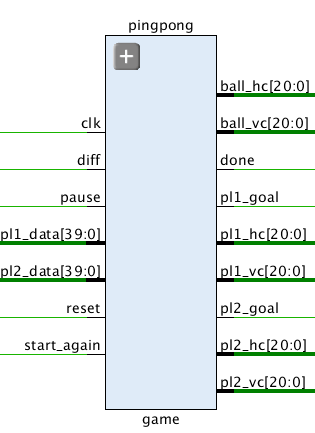




Since we don’t need SW, LED, AN, SEG, so we delete them. We don’t use seven segment display controller, so we also comment it out. In our project, we use two dimension instead of choosing only one dimension, so we don’t need posData to check X-dimension and Y-dimension. Because we don’t use LED, we comment out the always block. We just need to assign sndData = {8'b100000, 0};

**Game module**

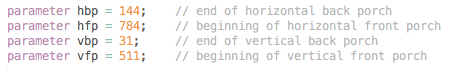
In the game module, we handle everything that happens while in game mode. This includes handling the movement of the ball and the players, handling all collisions, handling the case when one of the users wins, react to pause, and adjust parameter based on user’s choice. We will elaborate on each one.



The photo below shows all the inputs and outputs of the game module, and the comments by each one indicate what they are.



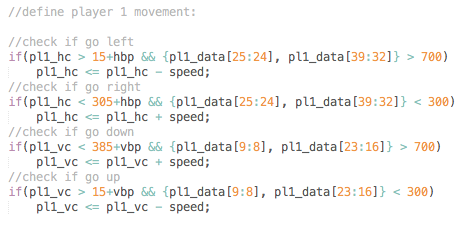
Also, we defined useful parameters, so we only measured distance from the back porches:



Descriptions of the design below.

**Movement of the players:**

In this part, the input was the output from the 2 joysticks. How it works\_\_\_\_\_\_\_\_\_\_\_\_. Using this, we enabled the user to move up, down, right, and left, as long as they dont cross their boundaries. The speed of each user is defined by the difficulty, either slow or fast. The implementation of the design can in the photo below.



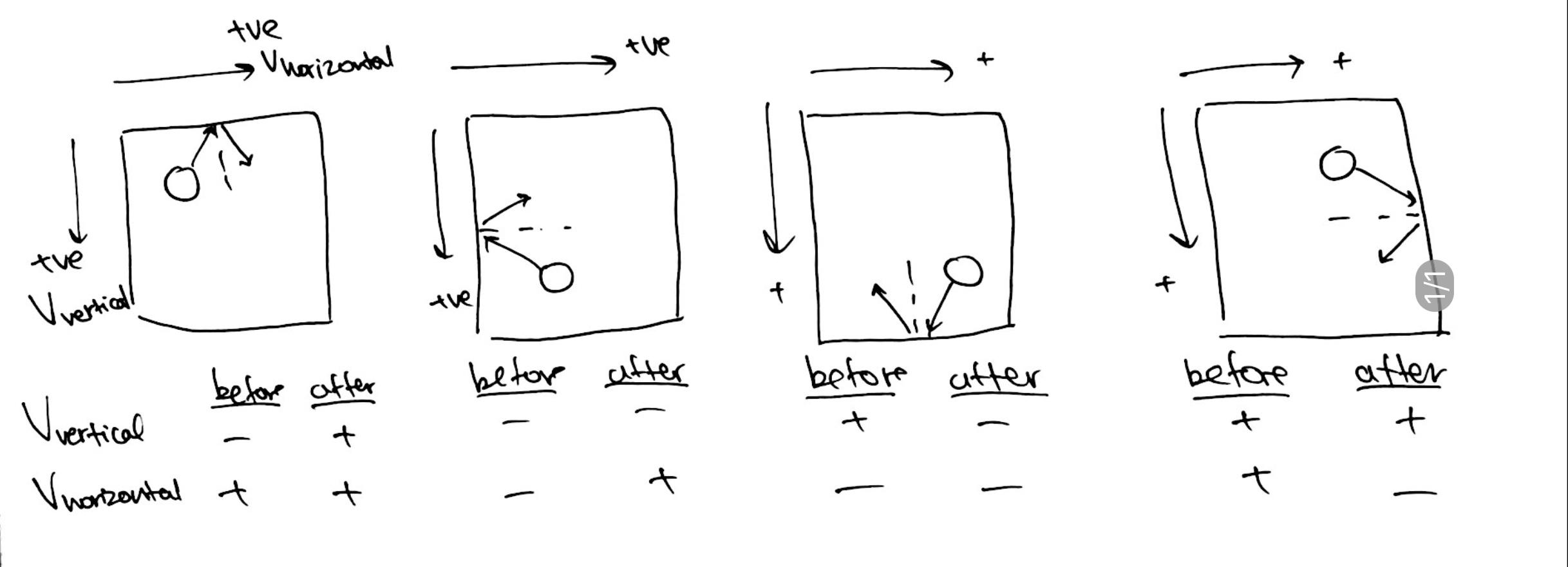
The code for player 2 is very similar, except for the horizontal boundaries. Regarding movement boundaries, player 1 is allowed to move only in the left half of the board, while player 2 is allowed to traverse only on the right half of the board (not including the middle boundary and the side boundaries). For each player, we check that the bits are greater than 700 or less than 300, which makes the movement of the players not very sensitive (since we don’t want players to touch lightly the joystick and therefore move).

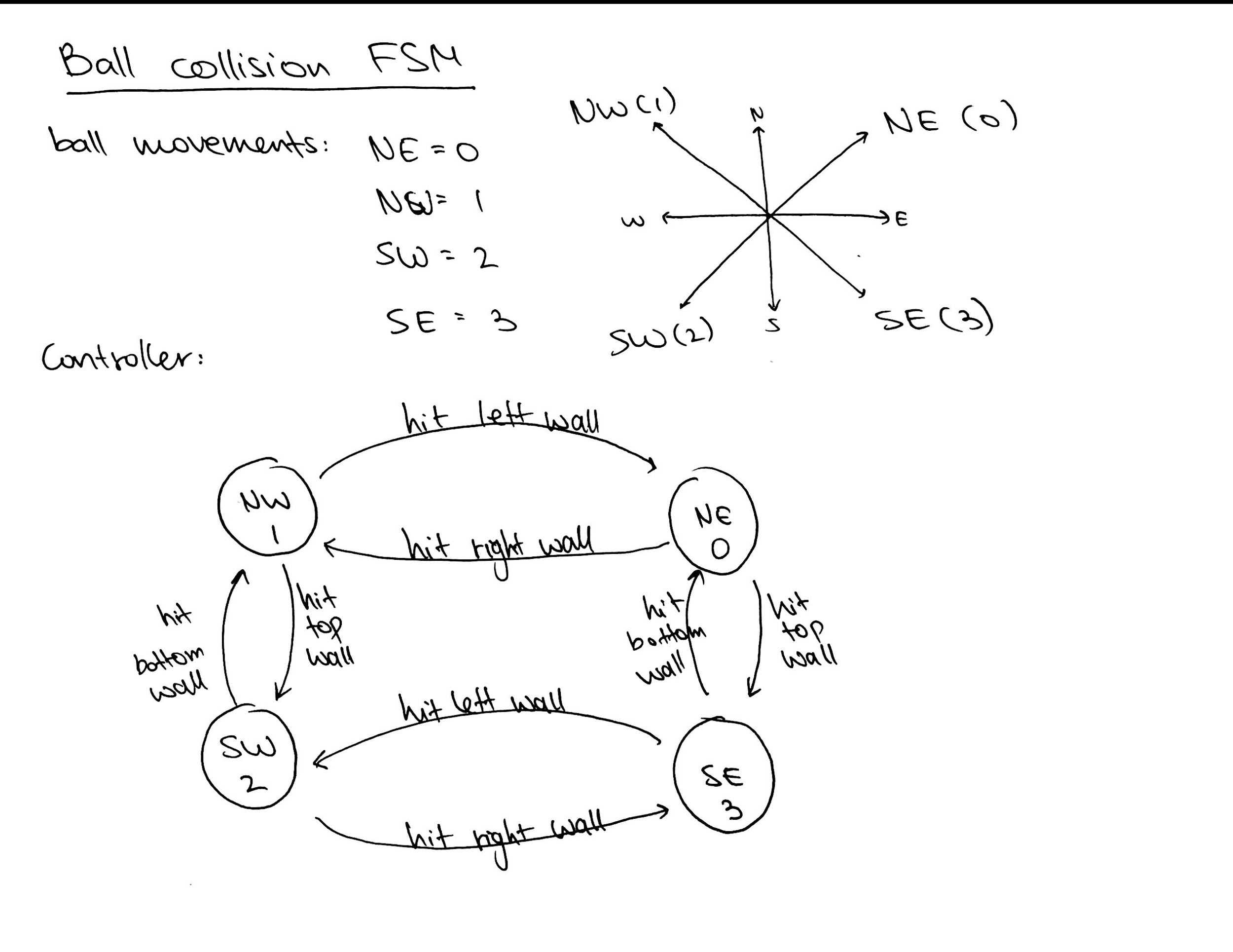
**Movement of the ball:**

The ball, unlike the players, can move around the entire board, only restricted by the boundaries of the board. Since there is no input for the ball, we just add the speed of the ball to its vertical and horizontal speed.

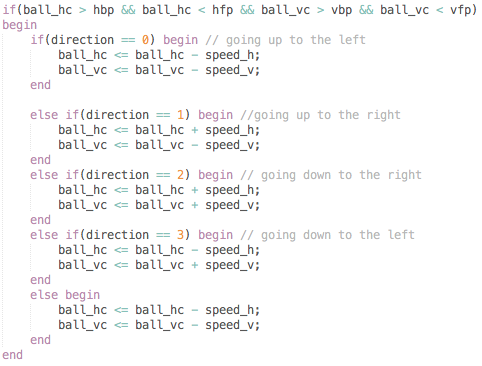
**Collisions with the ball:**

In this case, there are two types of collisions: collisions of the ball with the wall, and collisions of the ball with the players. However, since the collision with the left wall is handled the same as a collision with the right player, and similarly for the right wall and the right player. When designing this part, we analyzed what happens to the speed of the ball when it hits a wall. The photo of the 4 types of collisions can be seen below.

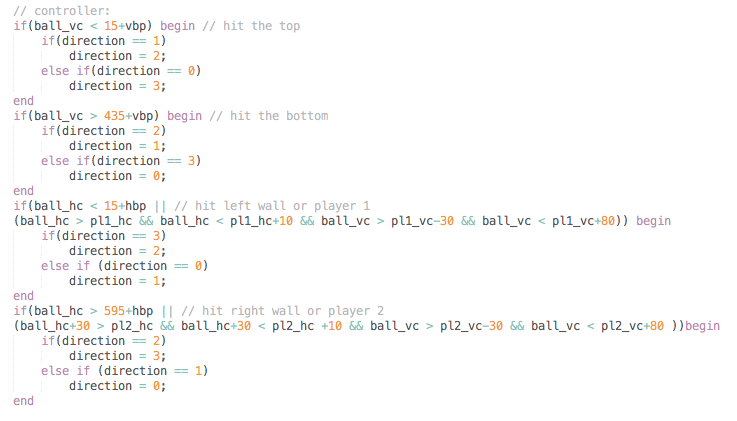
We observe that the only thing that is changing whenever the ball hits one of the walls is either the horizontal speed, or the vertical speed. We simply wrote few if statements, and multiplied the correct speed by -1. However, even after declaring the registers as signed, it was not allowed. Too much debugging time led us to change our design into an FSM which controls the direction of the ball. The ball can go to 4 directions: north-east, north-west, south-east, and south-west. It’s next direction is determined by which wall the ball hit (the input), as well as at which direction it was going (the state). Therefore, since the next state is determined by the current state and the input, this is a Mealy finite state machine.



Here is the code of how we implemented the controller of the FSM (photo on next page).



Now that we had the controller, we had to add the movement of the ball based on the direction. This was few if statements that depending on in which direction the ball is moving, would add or subtract the speed of the ball from the horizontal or vertical distance, according to the photo that analyzes the collisions with the ball. This the implementation:



**A user wins**

When the ball reached on of the goals, it means that the player who own the target loses, and the other player wins. In order to check that, we simply had to check the coordinates of the ball to see if it intersect with the target. Once one of the players lose, the current round must end.

This is the code we used:



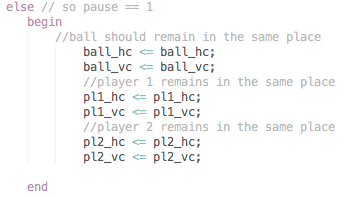
Notice that the vertical distance for both goals is identical. For player number 1, we must check that the horizontal coordinates are less than the horizontal back porch + 15. However, for the second player, we got the number 594+hbp by taking the distance from the 2 horizontal porches which is 640 pixels, then subtracting 30 for boundaries from each side, and then subtracting the radius of the ball, since the location of the ball is measured from its top left corner. We also subtract 1 from each to make sure the ball’s movement is not restricted.

Once either pl1\_goal is 1 or pl2\_goal is 1, we must tell the VGA module that the round is done, so it will switch windows. Notice that we reset the speed of the ball to equal zero, this is because otherwise the ball will keep going and potentially hit a goal again.

**Pause**

While on game mode, the users can use a switch to pause the game. Once the switch is high, the game is paused. The game resumes from the point it was left at when the user continues the game by setting the switch to low.

The implementation is simple, as it only makes sure the coordinates remain the same.

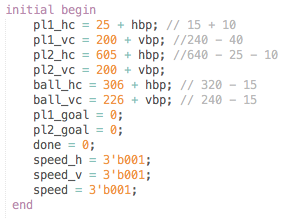


By getting the pause switch as an input only to the game mode, we make sure that the switch has no functionality in the score mode.

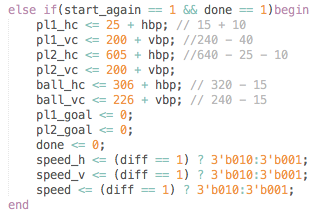
**Adjusting parameters:**

Whenever the game starts, some parameters must be adjusted. There are three cases we handled.

The first one is when the program starts. In this case, the players and the ball are starting from their starting locations in the vertical middle, the score are set to zero, and the speed is slow regardless of the choice (since user can only adjust difficulty in the score mode, but they still didn't get there).



The second case is when the user presses continue from the score mode. Then, we have to adjust the difficulty according to the user’s choice.



This happens inside an always block, therefore it is sequential logic so we need to assign by <=.

In this case, notice that speed\_h, speed\_v, and speed are all adjusted based on diff, the difficulty. Speed\_h and speed\_v are for the ball, whereas speed is for the players; they all have the same speed. We also check that done == 1 because we cant change start\_again (since it is an input from the VGA module). Therefore, without done == 1, the players and the ball would remain in the same place since the condition would always be true. However, since we are esetting done to 0, this would only happen once at the beginning of the game. We chose to check done because it has to be high from the previous game, and we can reset it.

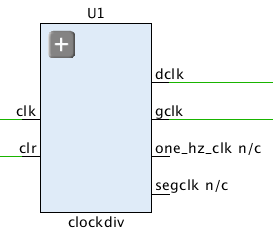
In the third case, we check if reset == 1 and assign exactly like in the initial block. This is because when they reset, the program acts as if they just opened it, so the difficulty cannot be adjusted.

**Communication with VGA module:**

The game module communicates with the VGA module for 2 reasons. The first is that they have to know when to stop game and start score mode, and vice versa. The second reason is that some buttons have a meaning depending on the user’s mode.

There are 2 inputs that game module receives from the VGA module. This is the variables start\_again and diff. Start again represents the continue button that can be pressed only in the score mode, and therefore tell the game to start again. Diff indicates the difficulty that the user has chosen. This guarantees that switching to difficult or pressing the button continue while on game mode will have no effect.

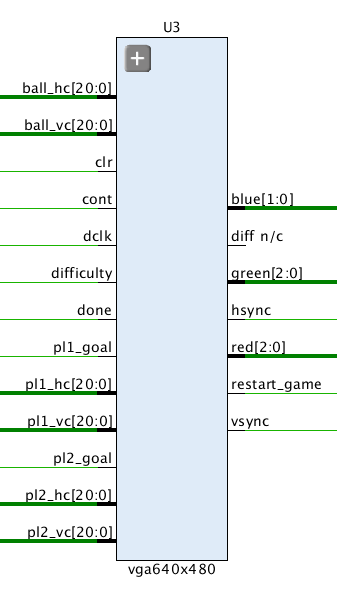
**Clockdiv module:**



In this module, we generate two lower clocks which are dclk and gclk. In our code, we have four clock, but we only use dclk and gclk in our project. The segclk is from the demo we use, and we didn’t comment it out.

**VGA module:**

In order to make this module, we used the ideas we saw in the NERP\_demo demo. Below are the inputs and output of the module.



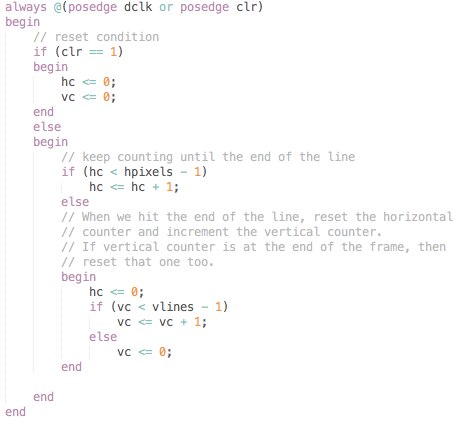


In order to make this module, we used the demo for the VGA provided in the the lab instructions. We left the input and output parameters as they were, and we add many more.

For the inputs of the module, we have the display clock, a clear button (clr), the coordinates of all objects on the board (which are outputs of the game module), done bit indicating game mode is done, a bit to indicate who won the previous round, a button continue from the user, and a difficulty switch.

The outputs are: the difficulty which goes to the game module, and therestart\_game bit which goes to the game module and lets it know to start a new round. Then, in order to display, we are outputting horizontal and vertical coordinates as well as the color at that coordinate.

For this module, there is one block of code that we left from the demo:

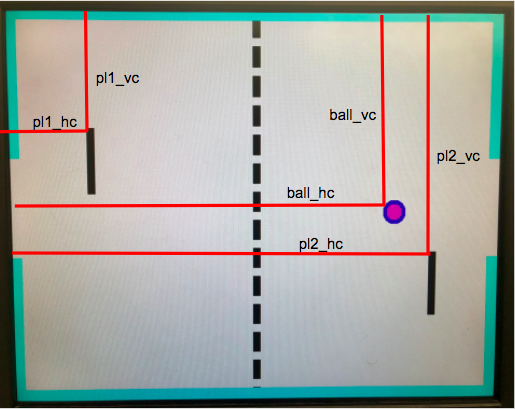


What this block does is defining two values, hc and vc, and keep incrementing and resetting them based on the line and column that are display.

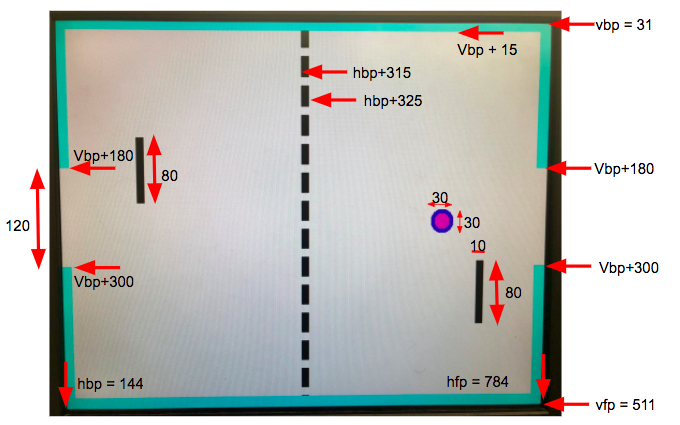
This module is divided into 2 parts, based on the value of the variable in\_game\_mode. If it equals 1, we are in game mode and we are displaying the game, otherwise, we display the scores.

**In\_game\_mode == 1**

In the large block of code, we define everything that happens in the game. The photo below represents how the distances are given to the module as inputs from the game module. All object's distance are represented from the horizontal/vertical back porch, up to the top left point of the object.



Further, the photo below represents how the objects are defined (horizontal and vertical distances), and all the associated variables.



In order to make the above photos, we wrote all the code inside an always block.



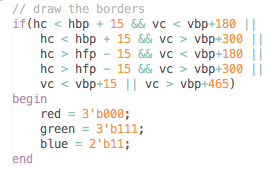
Which makes sure that the display changes every time one of the variables change.

First, we check that we are in the vertical and horizontal video range by:



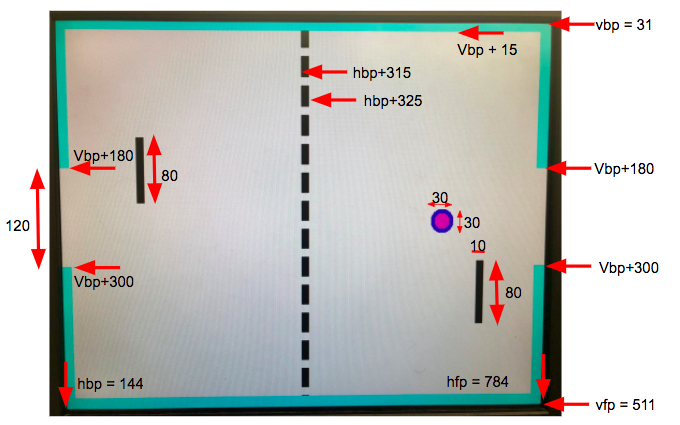
This works earlier in the module we defined the horizontal and vertical front and back porches, which define the display range.

Then, we continue to draw the blue borders on the sides of the board.

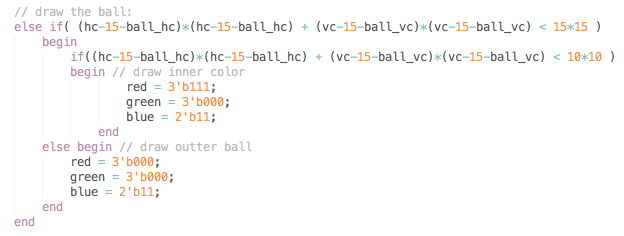


This block of code checks few things. First, if at any time the vertical coordinate is at the top 15 pixels, or the bottom 15, then it will be colored, which creates the top and bottom borders. The side borders are essentially the same since they change if at any time the horizontal coordinates are the left 15 or the right 15 pixels, but they also check to make sure that the goals are not colored.

After making the borders, we write an else if statement to draw the ball. This means that if we are not draw the border, check if we need to draw the ball. It makes sense because the ball never overlaps with the border. Below is a photo of the ball.



This is the code to draw the ball:

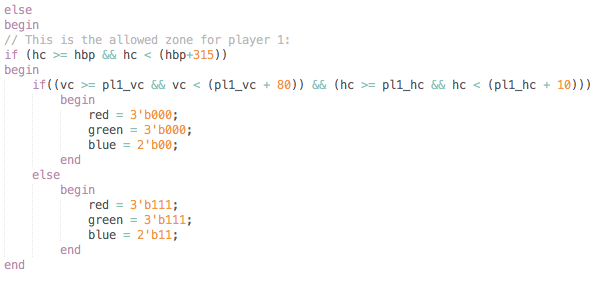


In order to draw the ball, the following condition must be met:

(

Which gives us the formula used for the inner and outer ball. For the outer ball, the radius is equal to 15, and the inner pink ball has a radius that is equal to 10.

Then, we continue to draw the left part of the board, where the left player is. We once eagain use else if statement to indicate that if we are not drawing the ball, then we should check if we can draw the left player.



First, we check that we are in the left half of the board. Then, if we are where the pl1\_vc and pl1\_hc are, we start drawing the left player. Since it is only a black rectangle, we only check that if the vertical coordinated start from where the player is and end 80 pixels below that point (positive 80 since the front porch is at the bottom), and that the horizontal coordinates start from where the player is and end 10 pixels to the right. This is consistent with the game, that takes into account distances from the top left corner of the object.

Next, we once again do an else if statement which covers the middle 10 pixels of the board, and draws the middle border.

A photo of the code is on the next page.



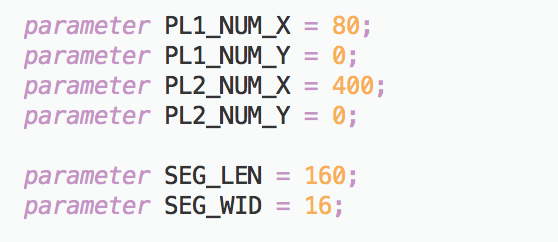
In order to draw the middle border, we make everything black except for the white spaces that are defined by the vertical coordinates.

Similarly to the left side the board, we make the right side where the right player is. We are not adding the code to this report because it is very similar to the left half, only different in the horizontal constraints.

**In\_game\_mode == 0**

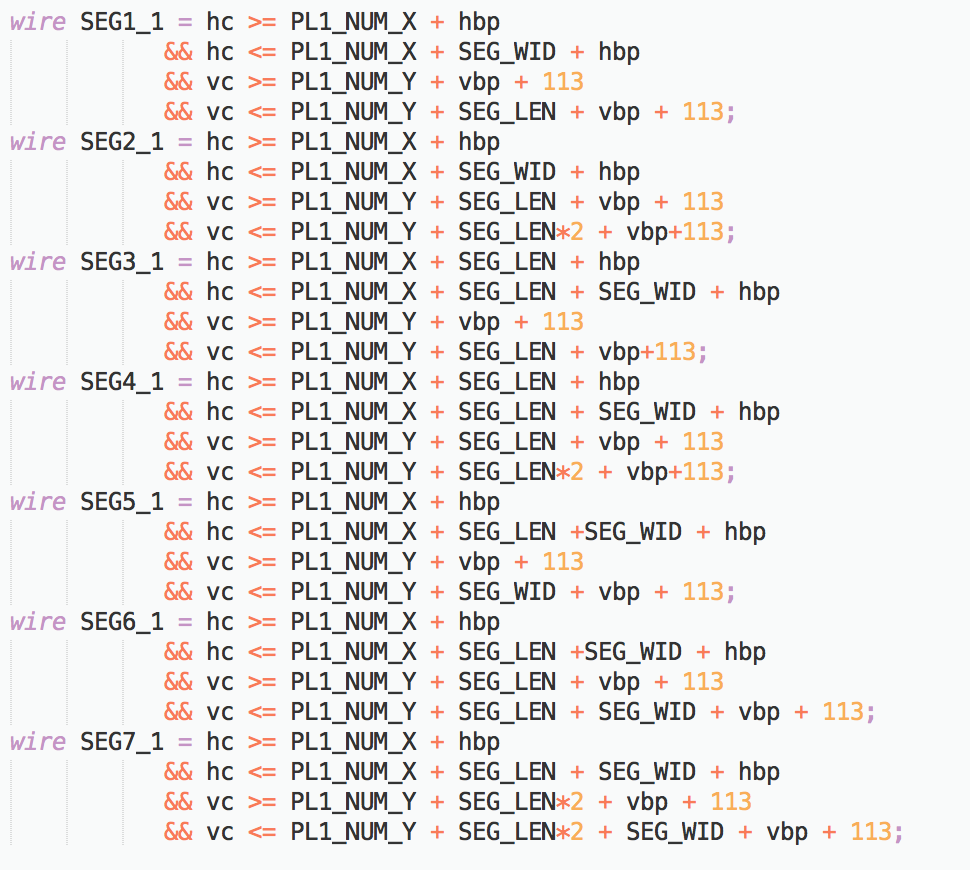
When in\_game\_mode == 0, we are in score mode. In this mode, we need to display the final scores of player one and player two. At the same time, we display the color of winner’s score as green and the color of loser’s score as red. Firstly, we wanted to display a number on the window, so we separate the number into seven segment. We define the position of player one’s score, player two’s score, segment’s length and segment’s width.

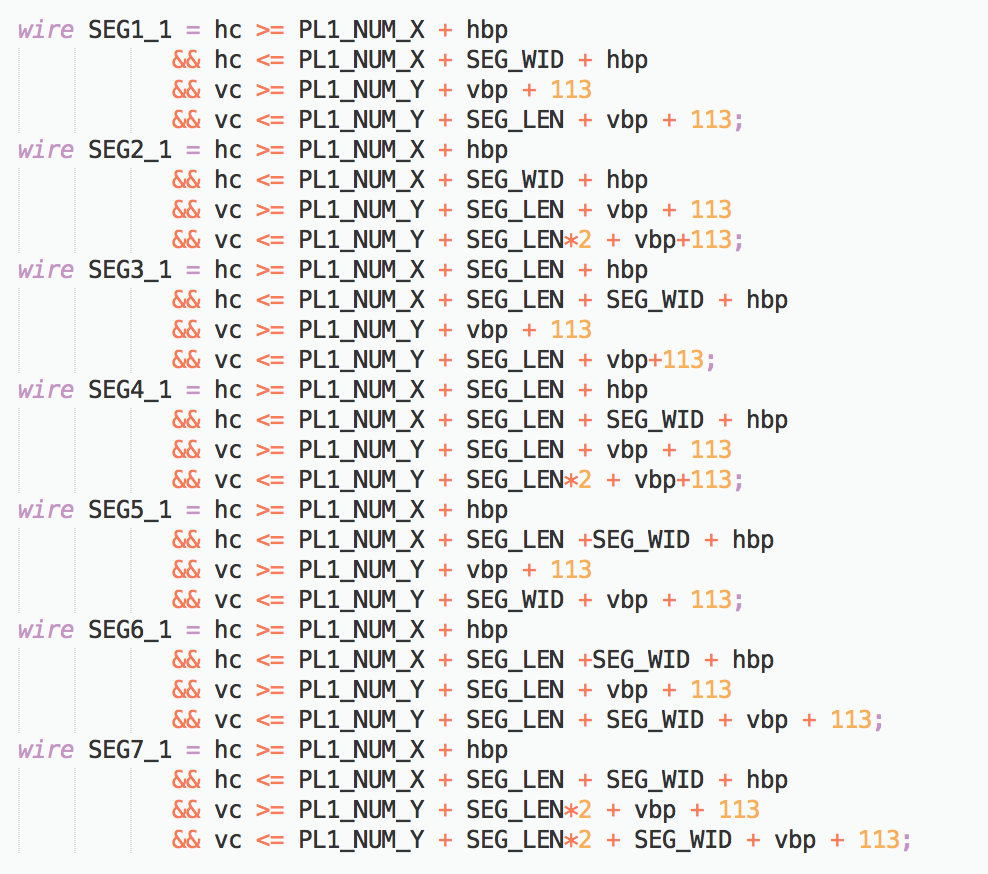
The following snapshot is the code how we define these parameters.



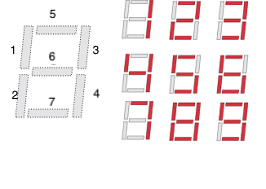
We define the location where we draw these segments in the next step. Since we want to make each segment starts drawing X and Y at the same point, in other words, seg\_one\_startpoint\_X = seg\_one\_startpoint\_Y. From the former definition, hbp is 144 and vbp is 31, and the difference between hbp and vbp is 113. So when we define the range of vc, we add 113 at the end to make sure X direction of segment startpoint and Y direction of segment startpoint start at the same point.

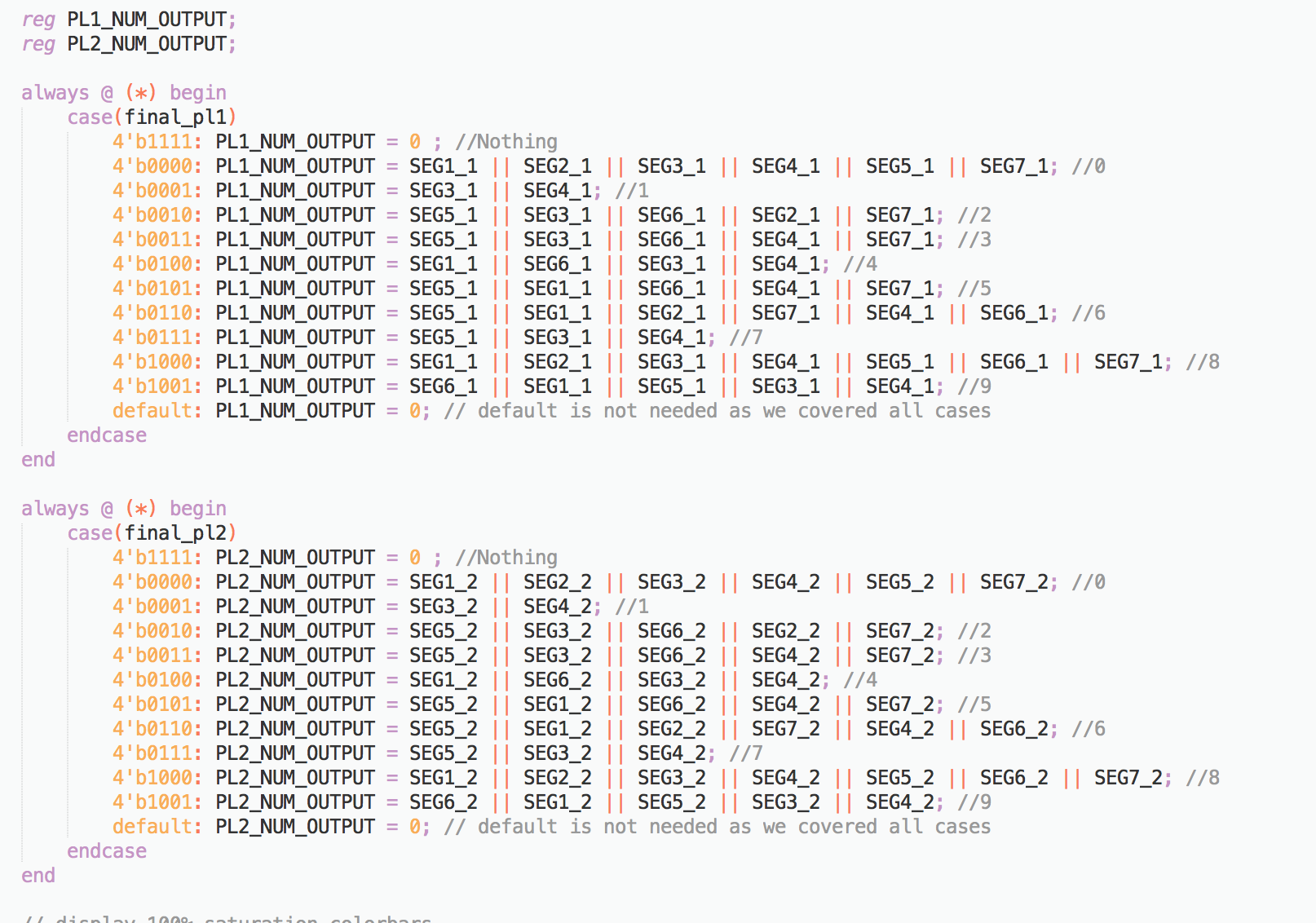
The following snapshot is the code how we define the position of segments.

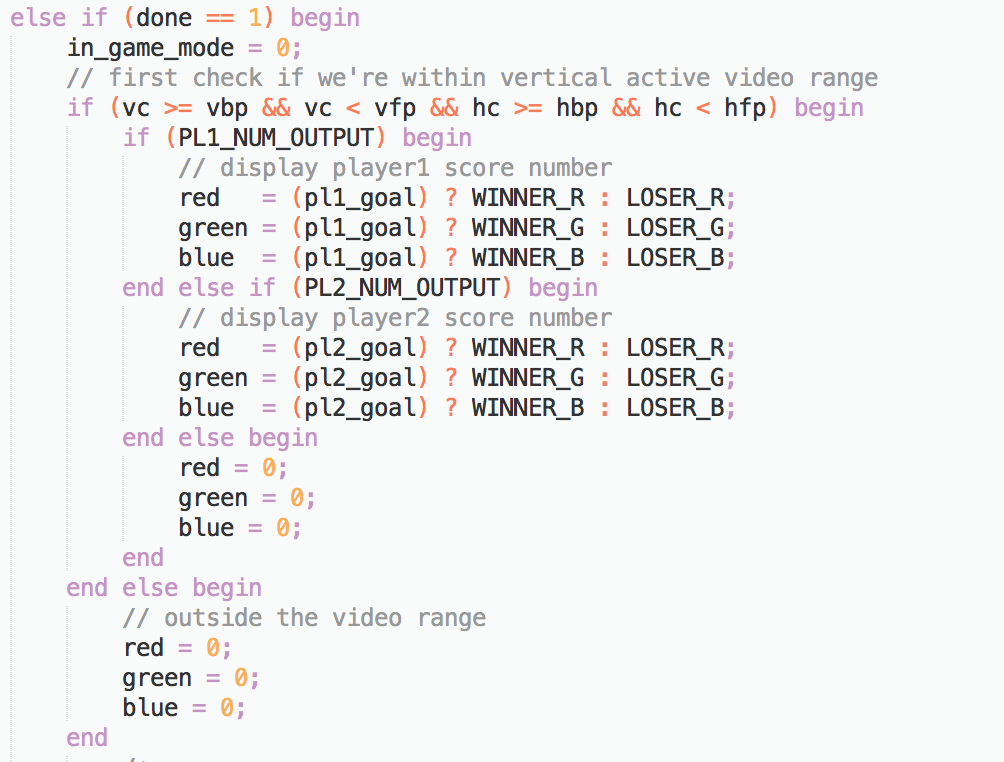




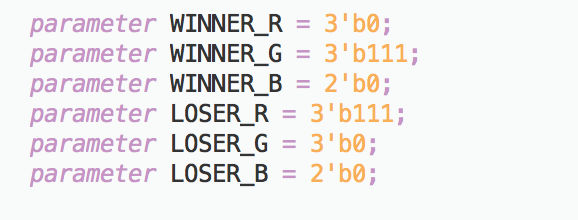
When we want to display number 0, segment 1, segment 2, segment 3, segment 4, segment 5, and segment 7 should be drawn which means when hc and vc in the range of segment 1 or segment or segment or segment 4 or segment 5 or segment 7, we draw out the number. When we want to display number 2, segment 3 and segment 4 should be drawn which means when hc and vc in the range of segment 3 or segment 4, we draw out the number. We use the same method to display number 2 to 9. Firstly, we use case to check the final score, and then there is a reg call PL1\_NUM\_OUTPUT to check if the corresponding segment is in the range of the area we should color, if yes, then PL1\_NUM\_OUTPUT=1. In the big always block, when done is 1, and in the score mode, if PL1\_NUM\_OUTPUT is 1, we draw the number on the window.

The following snapshot is the code how we display number.



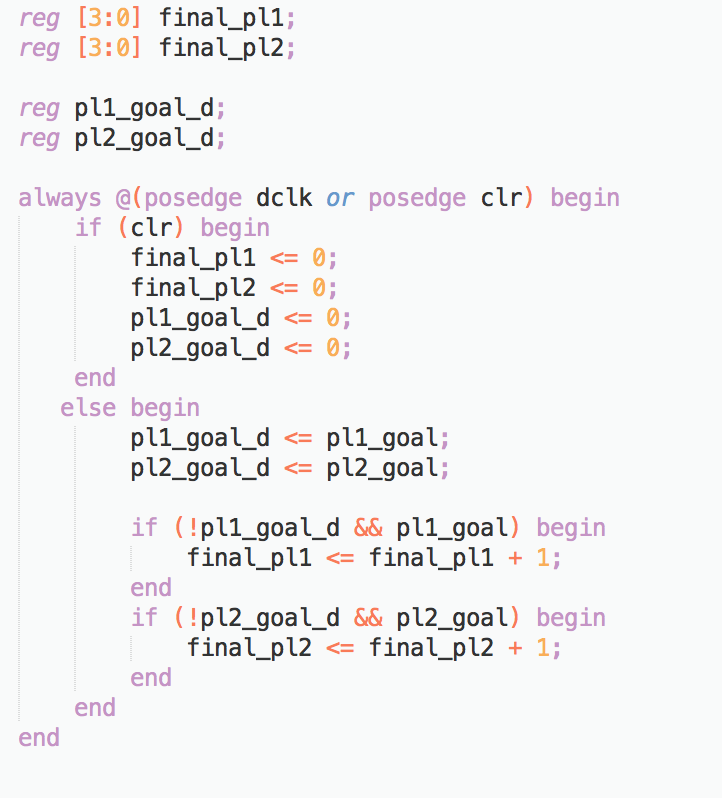


When the player win the game, we want to display the score as color green. When the player lose the game, we want to display the score as color red. We define the winner color and loser color as parameters at the beginning, and then when we draw the number, we check if the player wins the game first. If the player wins the game, then the number displays green color. Otherwise, the number displays red color.



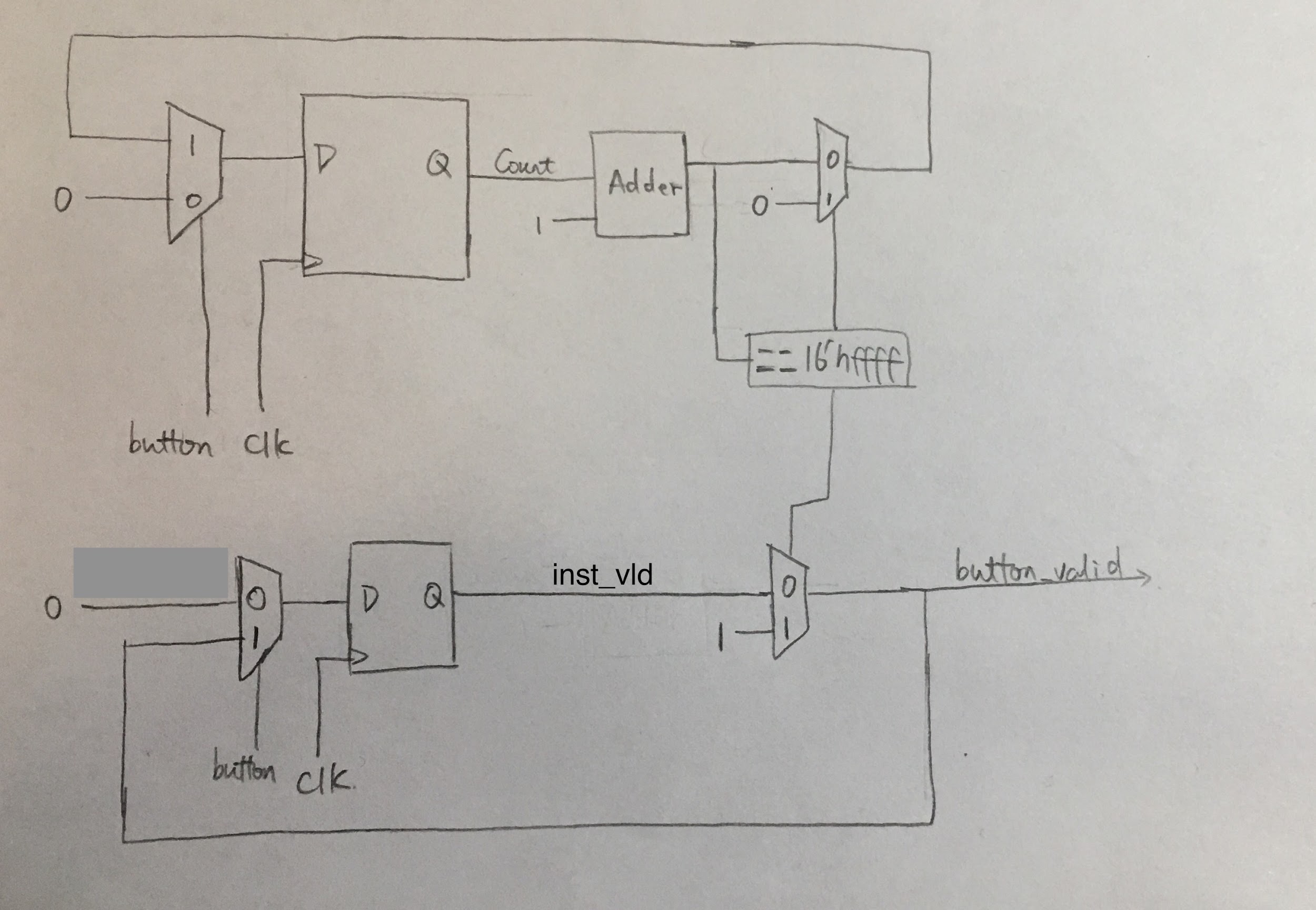
Since we only have pl1\_goal and pl2\_goal as inputs, we also need to write another always block to calculate the final scores.

The following snapshot is the code how we calculate final scores.



**Debouncer**

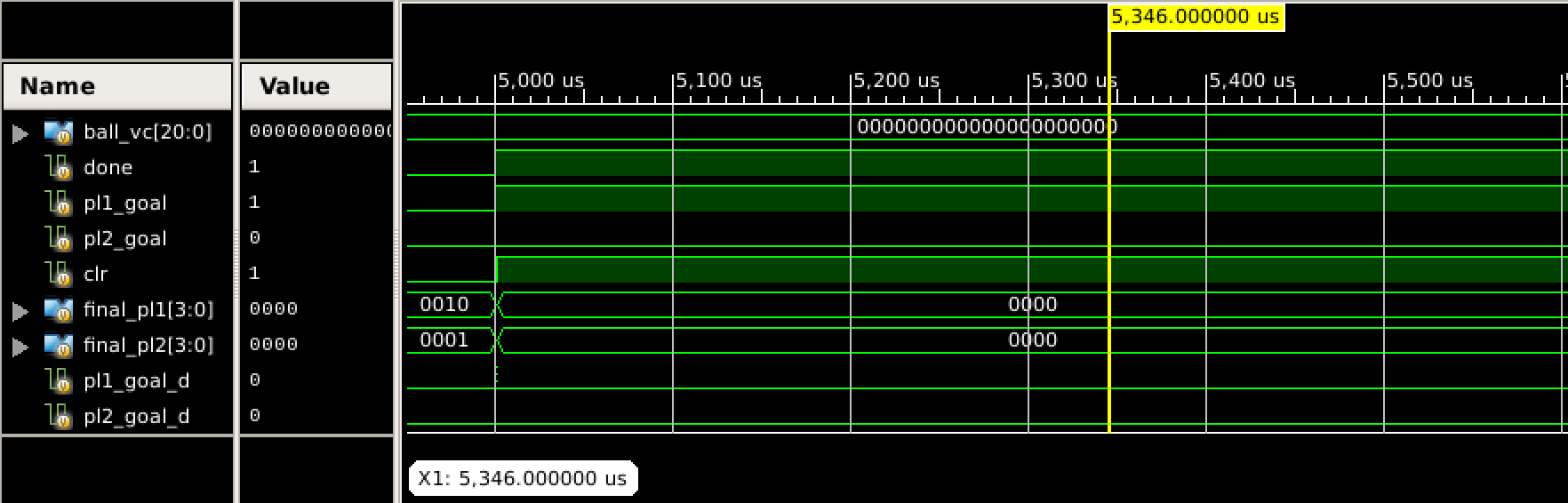
The debouncer module is used for checking if the input button is valid. When a button is pressed, before the button pressed signal becomes stable, it will generate noise. So we need the debouncer to separate the noise from the stable button pressed signal. We assume that when the button is pressed for a long enough time period, we consider this is an valid button pressed signal. We use a 16 bit counter and the master clock as the input to check the if the button pressed for a long time, when the counter hits the maximum value, debouncer module will return the valid signal for the input button. Therefore, when pressing the buttons in our stopwatch, it is important to give them a firm press. A diagram of the debouncer can be seen on the next page.



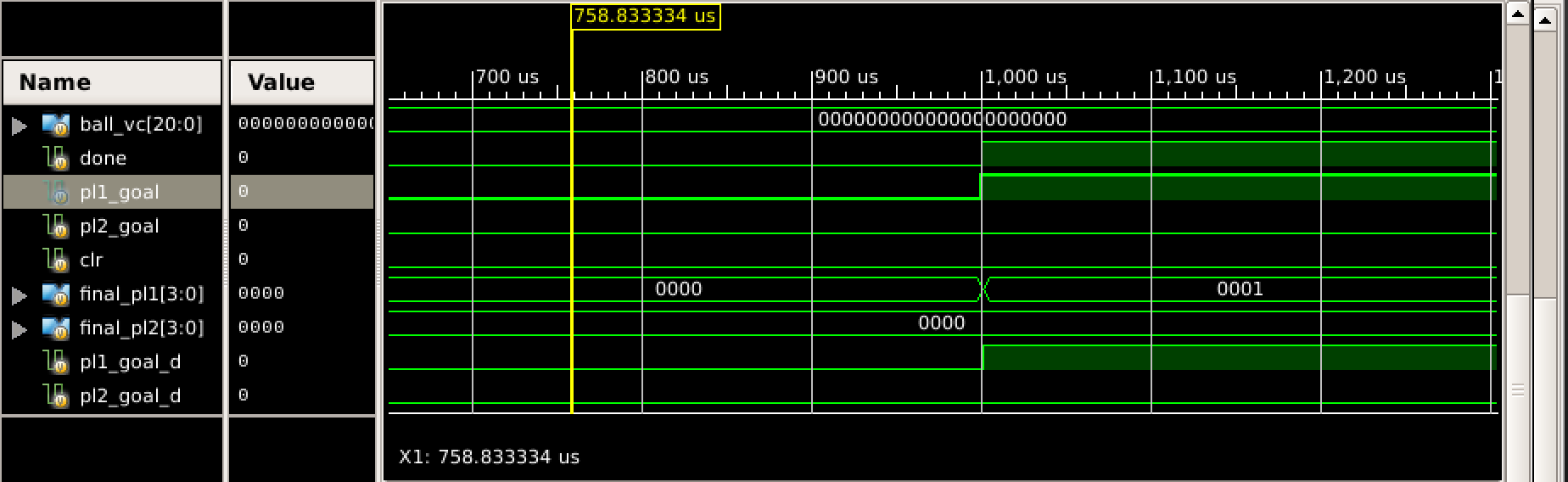
**Simulation Documentation**

**vga640\*480 module:**

|  |  |
| --- | --- |
| Test Case |  |
| clr=1 | When clr =1,pl1\_goal\_d=0, pl2\_goal\_d=0, final\_pl1=0, final\_pl2=0. |
| clr=0 | When clk=0, pl1\_goal\_d and pl2\_goal\_d will store the value of pl1\_goal and  Pl2\_goal, final\_pl1 and final\_pl2 will accumulate the score. |

****

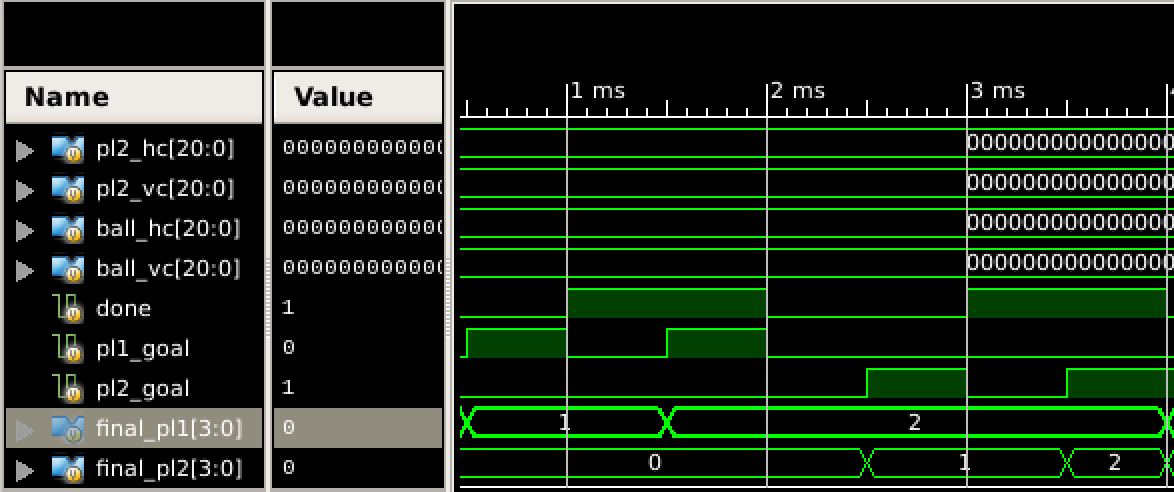
In this waveform, we see that when clr =1, pl1\_goal\_d, pl2\_goal\_d, final\_pl1 and final\_pl2 will be reset to 0.

****

In this waveform, we see that when clr =0; final\_pl1 and final\_pl2 accumulate the score.

In order to make the final score increment correctly, we set some test cases for the accumulation of final score.

|  |  |
| --- | --- |
| Test Case |  |
| At t=0.5ms, pl1\_goal=1, pl2\_goal=0 | final\_pl1 =1, final\_pl2=0 |
| At t=1.5ms, pl1\_goal=1, pl2\_goal=0 | final\_pl1=2, final\_pl2=0 |
| At t=2.5ms, pl1\_goal=0, pl2\_goal=1 | final\_pl1=2, final\_pl2=1 |
| At t=3.5ms, pl1\_goal=0, pl2\_goal=1 | final\_pl1=2, final\_pl2=2 |

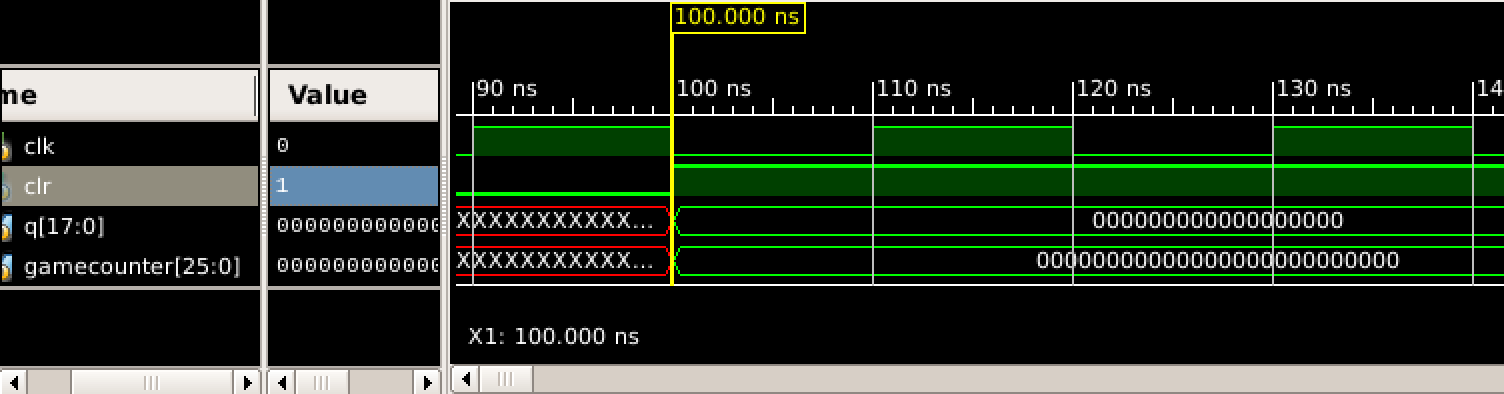


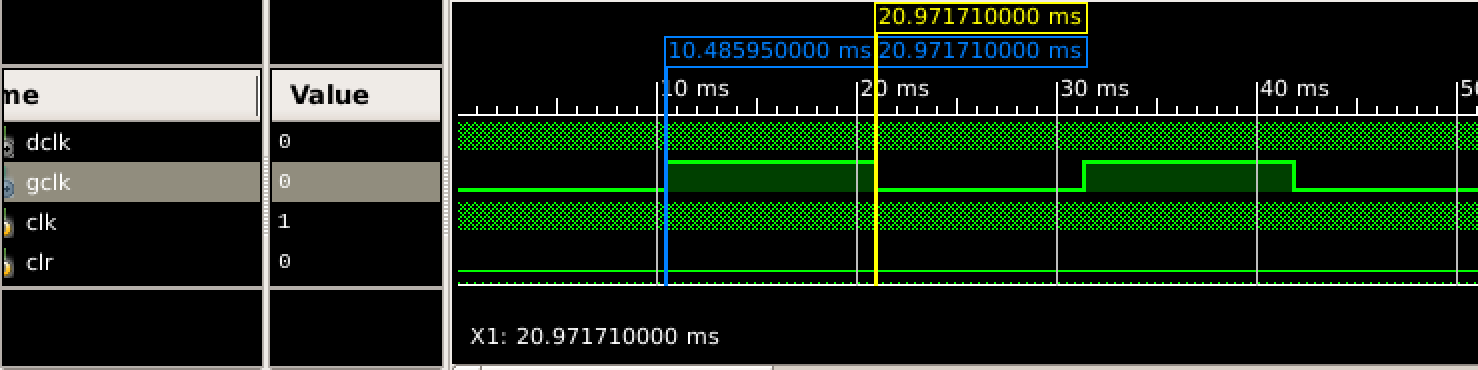
In this waveform, we can see that final\_pl1 and final\_pl2 increment correctly.

**clockdiv module:**

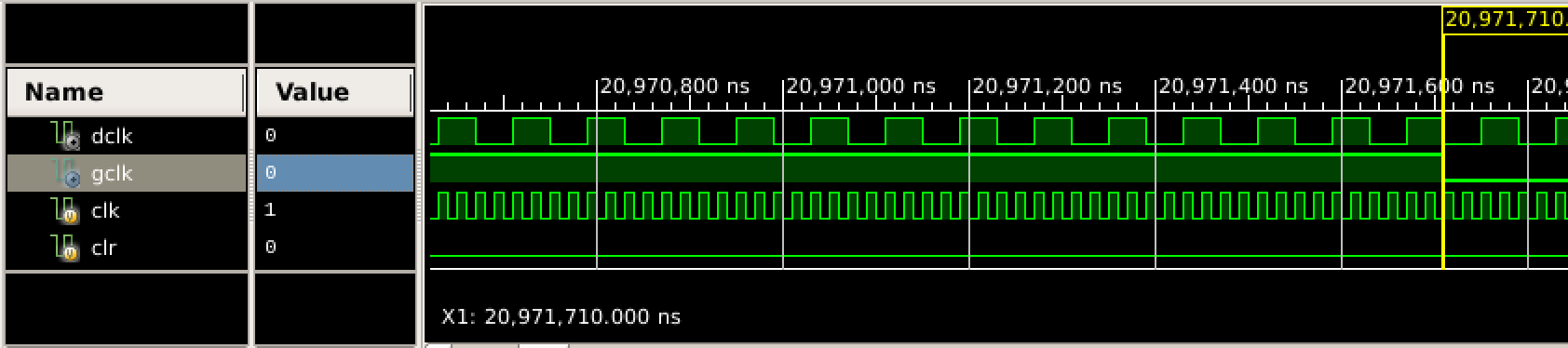
|  |  |
| --- | --- |
| Test Case |  |
| clr =0 | the counter of gclk and dclk will be reset to 0 |
| clr =1 | gclk and dclk can be generated correctly |

In this waveform, we see that at t=100 ns, the counter of gclk which is gamecounter and the counter of dclk which is q are reset to 0.





In the following two waveforms, we can check that dclk and gclk are generated correctly.



**Debouncer module:**

In our project, we used the same debouncer as lab3, so we didn’t create a test bench for the debouncer, since we knew that it is working.

**Joystick module:**

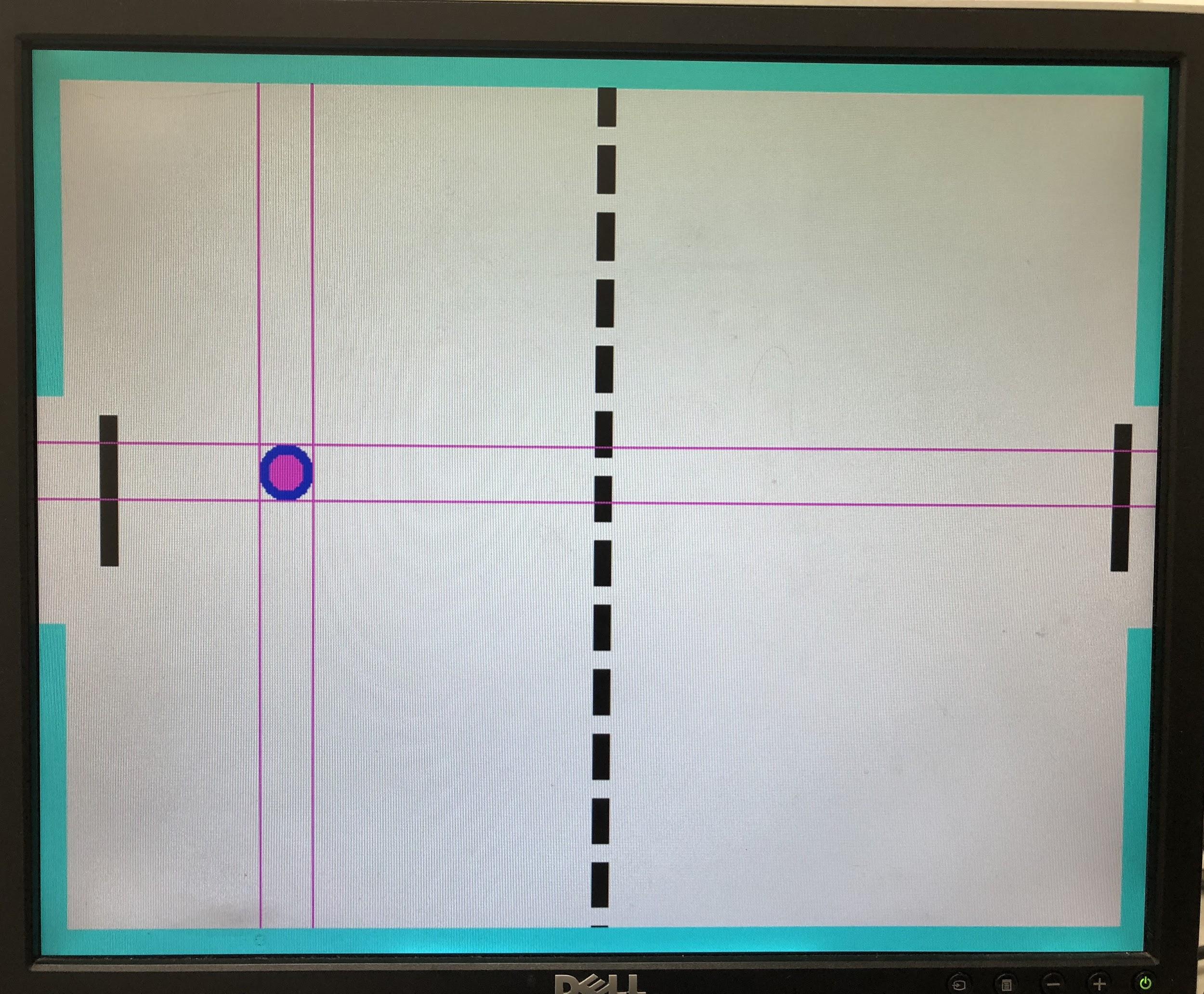
For the joystick, we used the demo which is provided on the project specifications. We made some changes in order to make the program work for two joysticks and two dimension. We then built the program, and played with the joystick to make sure that it is working. Since it did, there was not need to write a test bench for this module.

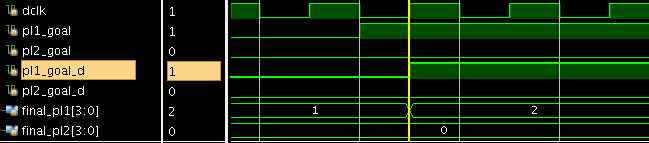
**Game module:**

For this module, we spent a lot of time debugging. However, we found that executing the code and trying to play the game was sufficient in order to debug the code for the game.

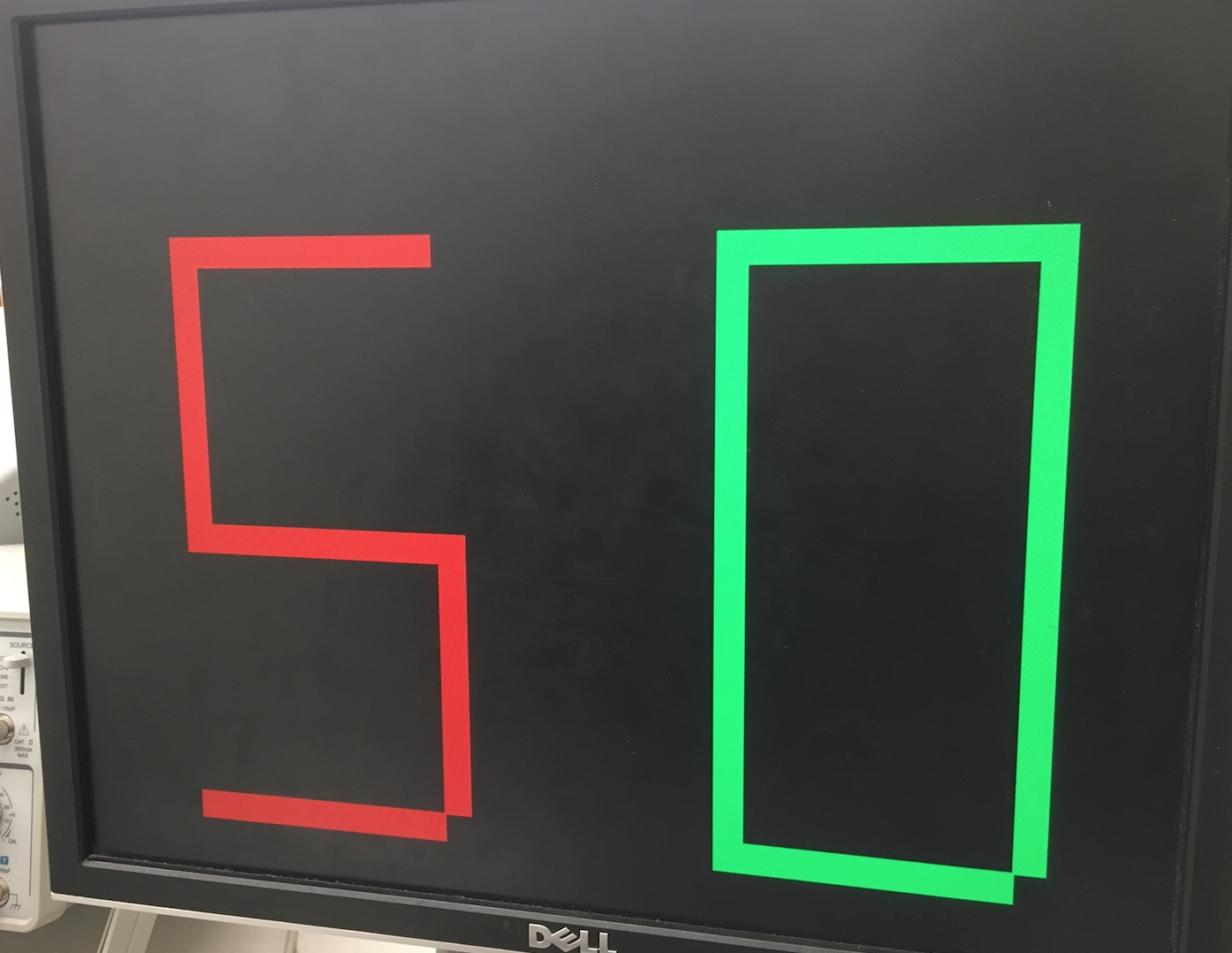
Whenever we found an error, we would immediately go back to the code and fix it. The disadvantage of this way of debugging is that we lost a lot of time on building the project, since the computers are very slow. Even if we made change only to one line, we still had to wait a long time for the project to finish synthesizing.

Here are some of the bugs we encountered in the project:

* After we drew out the boundaries and the middle line, we wanted to check if the rectangles (paddles of each player), can cross the middle line. We made sure that the rectangle could not cross the middle line, but we found out that when the rectangle reached the middle line, it couldn’t go back. We also found the same issue as the rectangle reached the left and right boundaries, the rectangle got stuck there. We checked the code, and we found that the condition we set up for movement was not correct. We checked if the players coordinates are less than some number, and then we let it go one more pixel. However, because of that, it could not go back. After that, we realized that we don’t have to check all the border when a play moves, but only one border per direction. For example if the player wants to move to the left, there is not reason to check where she is on the right side of the board, but only that she doesn’t go too fat to the left. This fixed the issue and the square did not get stuck anymore.
* Ball didn’t bounce, we could not simply multiply the vertical or horizontal speed by -1. We had to create a finite state machine as we described in Design Description (game module - ball collision FSM).
* Since we have game mode and score mode, we want to create two separate vga.v, one is used for displaying game window and another one is used for display score window at the beginning. But we found out that we could not simply connect another VGA window whenever done == 1. Therefore, we used the same VGA window. If done was 0, we would be in game mode, otherwise, we switched to score mode.
* We were not able to connect the second joystick. Sometimes, it worked, and sometimes it did not. But whenever the game started, the second joystick would immediately went to the top left corner. We found that there are 8 inputs, and we were connecting to the bottom 4 instead of the top 4. Therefore, the input for the second joystick was 0, so it went up and to the right. Once we realized that, we reconnected the joysticks and they worked perfectly.
* At first, we wanted to connect a new game signal to vga640\*480 module, and then passed this signal from game module to game module. When we configured the top module, it had error. So we passed a new game signal which is clr in vga file and reset in game file to both module directly, and then it worked.
* In game module, when done is 1, we need to change mode from game mode to score mode. At first, we just checked when pl1\_goal or pl2\_goal is 1, then we passed 1 to done, but we didn’t make the ball to stop. Therefore, when the screen showed scores of player1 and player2, the scores would increase automatically because the ball may hit the goal again even we on the score window. We reset speed to 0 so the score incremented while on score mode.
* At the beginning of the lab, we drew the ball as a square because it was easy. Later on, we wanted to change it to a ball. It clearly did not work as intended because we received an output that did not make sense. So, I drew contour line where where the box is and compared it to where the ball is. At first, I accidentally added hbp and hfp in the equation, and it drew the ball on the bottom left of where the square is. Then, I kept debugging and changing the code until finally it resulted in the ball inside the contour lines. This can be seen in the photo below. After that, I simply removed the lines.  
  
* After we connected the goal signals to the vga module and tried to display the player scores, we found we had a “9” and a “0” displayed on the screen, which are supposed to be both zeros. By a closer examination, we discovered that the “9” was a result of fast switching from 0 to 9, instead of a steady state on the screen. That means our score accumulation logic does not work as desired. Since one of the goal signals is always kept high when one player scores, we need to adjust our logic to detect the rising edge of the goal signal and only increment the score when we identify a rising edge. So we buffer the input goal signals by one cycle and detect rising edges by looking at the current value and the delayed value. With this change, the scores can be increment correctly, which is demonstrated in the following simulation snapshot.

****

* The last thing we added to the project is difficulty switch, we wanted to make sure the difficulty switch only work in score mode and don’t work in game mode. At first we passed the the difficulty to both module, but then difficulty switch worked in both module. And then we change the code to the following snapshot.When vga module pass start\_again signal to game module and done is 1, then we can check the difficulty. And then based on difficulty, we choose the speed.
* Even we created a test bench for vga640\*480 module, but we didn’t know if the number was displayed at the correct location. When we ran the top module, we found out the only half of the number was displayed on the screen because we set PL1\_NUM\_X = 80; PL1\_NUM\_Y = 70; PL2\_NUM\_X = 400; PL2\_NUM\_Y = 310; We changed the PL1\_NUM\_Y and PL2\_NUM\_Y to 0, and then the screen could display the whole number.
* When we displayed the number, we also found out that the number had defect. We decided to make Segment 5, Segment 6 and Segment 7 one SGE\_WID longer, so we add the SGE\_WID to the second hc which is check the right ending. The following picture is the number with defect.



**Conclusion**

In the air hockey lab, we had to design and implement the air hockey game, with little to no guidance. This was very challenging as it required from us to think about how to design the game, and then how to write the code efficiently and finally, to debug it. The final result of the lab was a air hockey game, where the players can control their movement with a joystick, and are able to move in all directions. Once a player wins the round, the game automatically changes to a score mode, where the player can see their scores as well as who won the previous round.

The game lab took the lessons learned in the previous few labs and built upon them. For this lab, there was much more code writing, debugging, and manual reading. However, this lab also added new concepts as we had to incorporate new ideas, such as the joystick and the VGA.

One of the difficult parts of the lab was writing the lab specifications before we understood how much time learning about the VGA and the joystick would take. When we designed the project proposal, we just thought about good ideas that we would like to implement because we thought they will be fun. However, because of that, we did not have a good estimation of time. Further, after giving each part of the project a certain amount of points we would get for that part, we realized that some parts are much more difficult and deserve more points. Also, we forgot in the project proposal to add that a big part of the game is handling collision, and we should have added points for that part.

Because we chose a lot of things to implement in our project, we were very time pressured. We think that the main challenge in the project was to work as efficiently as possible because there were not enough office hours to come to in order to work on the project. Therefore, we had to code as much as we can before the lab, and then only implement it in the lab.

The points above brings us to the next difficulty we encountered, debugging. A lot of the debugging in the VGA module was about building the program and seeing what the output is. Therefore, it was very hard to write code for that module at home. Since the lab was a game, we found that the most convenient way to test everything is to just play the game and take notes of the imperfections, and then debug those.

By going to almost every office hour we could go to, and going to many other labs, we were able to finish the project (on the last office hour given a day before the project was due). Therefore, we are very proud that we have finished everything we had on the project proposal.

However, if we had more time to work on the lab, there are few things we would like to change. First, we would want to change the graphics of the game. When looking at other groups, we saw that some had python scripts used to translate the object into an image for the VGA module. We thought that was a good idea, and it could have made our project more attractive. Also, writing a python script is probably easier than trying to draw objects by ourselves. Further, currently the ball is always going in a constant speed, and no matter where it hits the the paddle, the velocity is negated but the speed remains the same. The game can be more sophisticated. First, acceleration can be incorporated, so that the ball would go faster after a collision, and then slow down. This way, if the players hit the ball closer to the border, the ball would go faster and keep accelerating, making the game more realistic and challenging. Furthermore, there should be a difference if the ball hits the middle of the paddle or the sides. If it hits the middle, the returning speed should be higher than the speed if it would have hit the sides.

Although there are many improvements that can be done to the game, using the time given to us, we did the best we could and we are very proud of the final result. The air hockey is a good game that allows the user to interact with the paddle, and play against another user. We were able to fix all bugs that we could find, and we can say that we believe that the final result is bug free, since we tested a lot. We got feedback back from students saying our game does not work because the ball can go through the paddle from the back. However, this is not a bug as our project simply does not support this function. It could, and it would require a small change in the ball FSM in the game module, but we decided that given the allowed time this project, we don’t have to implement that for the purposes of the project. However, this is just one other thing that we would like to add if we had more time.

Overall, the main difficulty was time. We had a lot of fun making this lab, and there are a lot of things that we would like to add, but we just needed more time. Nonetheless, we feel that the first three labs prepared us very well for this lab, where we made out own game. We feel that this was a very good way to end the class, and we learned a lot about the FPGA board and verilog. We are now confident we can make our own projects on the verilog.