EE 314 Digital Electronics Laboratory 2018-2019 Spring Pinball Project Report

Alkım Berke Bozkurt 2231520, Kaan Caner 2231553
Electrical and Electronics Engineering
Middle East Technical University

***Abstract*—This document is the project report for the EE 314 term project. We discusss the general structure of the project, explain the fundamental principles of our modules and elaborate on their interaction.**

***Index Terms*—FPGA, Digital Design, Game Design, Verilog, VGA**

I. Introduction

The goal of the EE314 Term Project is to implement a Pinball game on the FPGA which is to be displayed on a monitor through VGA. The completion of this project requires multiple different tasks. First, we need to implement a VGA driver which will be elaborated in Sec. II. The VGA driver provides the necessary signals to the monitor to display the desired image. For this task it needs to provide RGB outputs for every active pixel and provide synchronization signals in accordance with the timing specifications of our choice of display. The images to be observed in the game require the implementation of fundamental objects as explained in Sec. III. There will be circular objects (ball, penalty and circular targets), hexagons (scattering targets), vertical, horizontal and diagonal lines for the arena, flipper and the necessary displays. The interaction between these objects must be defined according to the game physics in Sec. IV. The ball will collide with the arena, circular, scattering targets and the flipper according to simple physical rules. Furthermore, gravity will be implemented to have more realistic motion. With the proper interconnection of all these subparts we will obtain a Pinball game which can be played using the buttons on the FPGA.

The fundamental structure of our project can be understood from the state diagram given in Fig. 1.

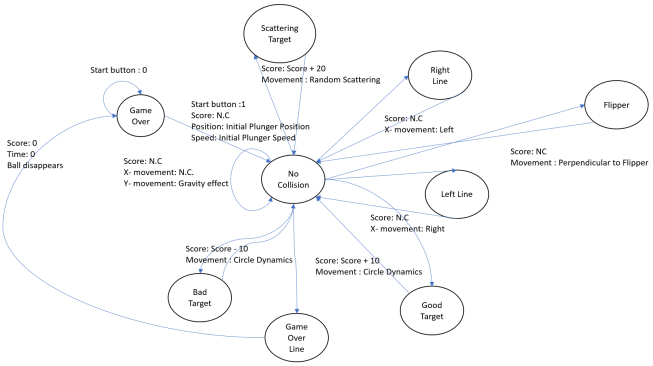

Fig. 1: State diagram of the project.

II. VGA Driver

The VGA Driver Module is perhaps the most essential part of the project design, as it is the module that drives almost every other module and provides the basis for how we display images on the screen. The most important considerations while giving VGA input to a screen are the timing specifications. These timing specifications are intrinsically tied to the output screen resolution and screen refresh rate. We have chosen to use a resolution of 640 x 480 pixels at a screen refresh rate of 60 frames per second. This requires that we use the timing specifications that can be seen in Fig. 2 at a pixel clock of 25MHz.

As we can see from Fig. 2, our screen is composed of lines. The screen is swept across these lines, pixel by pixel, and the RGB values of those pixels are seen as output on the screen if said pixel is an active pixel. Since our pixel clock is 25MHz, each pixel is "active" for 40ns before the next pixel is activated. As we can see, VGA signals have two phases, drawing pixels and the blanking interval. This means that while our active screen, i.e. what we will see on our display is 640 x 480 pixels, our actual "screen" is larger, and we should consider this while designing the VGA Driver Module. The horizontal blanking interval is comprised of regions called the front porch, the sync and the back porch which are 16, 96 and 48 pixels long respectively. The front porch and back porch regions separate the horizontal sync

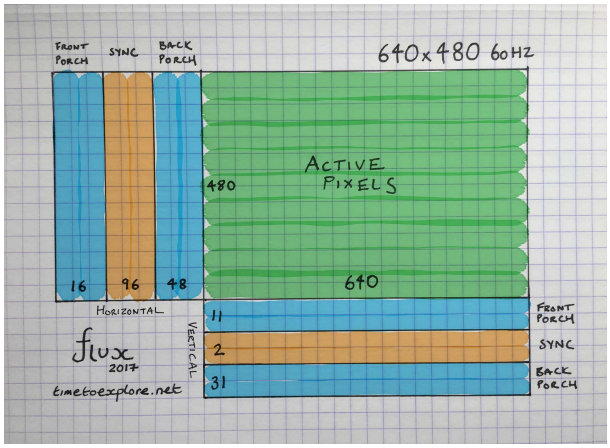

Fig. 2: Timing specifications of the VGA driver [1].

signal from the pixel drawing region. The active low horizontal sync signal determines the end of a line. Each line can be seen to be 800 pixels total. Once the horizontal sync signal is received, the screen moves on to the next line. Similar to this, the vertical blanking interval is comprised of the front porch, sync and back porch regions, which are 11, 2 and 31 lines long respectively. The vertical front porch and back porch regions separate the vertical sync signal from the pixel drawing region. The active low vertical sync signal determines the end of the screen (or the end of the frame). Each frame can be seen to be 525 lines total. At the end of each frame, we have one clock tick where animation, i.e. the updating of object positions, takes place. In order to properly display an image using VGA, we must use 6 of the output pins on the 15 pin D-sub VGA connector, those being the 3 pins for 8 bit RGB outputs, the 2 pins for the horizontal and vertical sync signals and the 1 pin for the VGA clock signal, which is 25MHz according to our timing specifications. We generate the RGB outputs and the VGA clock signal in the top module, so the VGA Driver Module only increments pixel and line values in accordance with the timing specifications described above in order to generate the horizontal and vertical sync signals and the pixel positions of the active pixels as outputs.

The correct operation of our VGA driver can be observed from our simulation results in Figs. 3, 4, 5, 6, 7. These results include the instants when the hysnc,vsnyc and animate signals are activated and deactivated.

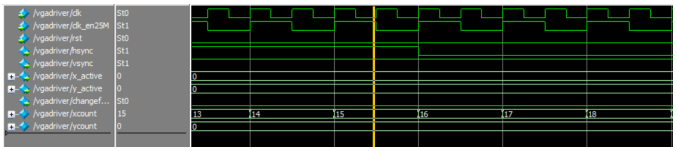

Fig. 3: Time instant when the active low hsync signal is activated (negative edge). This corresponds to xcount = 16.

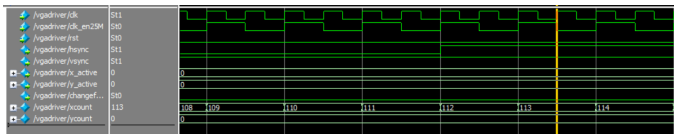

Fig. 4: Time instant when the active low hsync signal is deactivated (positive edge). This corresponds to xcount = 112.

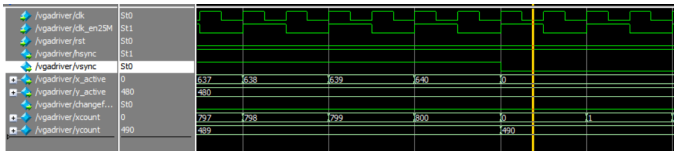

Fig. 5: Time instant when the active low vsync signal is activated (negative edge). This corresponds to ycount = 490.

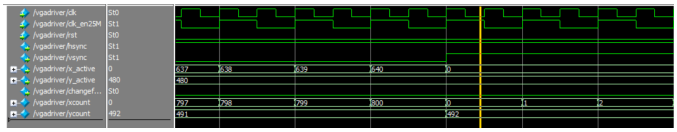

Fig. 6: Time instant when the active low xsync signal is deactivated (positive edge). This corresponds to xcount = 492.

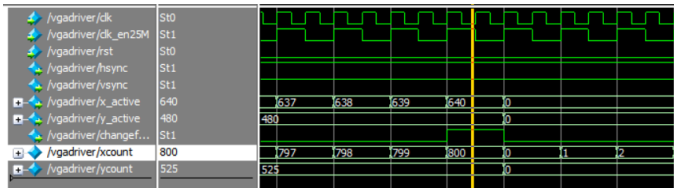

Fig. 7: Time instant when the animate signal is activated for a single clock cycle. This corresponds to the instant when xcount = 800 and ycount = 640.

III. Objects

The game requires the implementation of multiple geometrical objects. The ball, circular targets and penalty targets are formed of circles. The scattering targets are hexagons. Furthermore, the implementation of the arena and displays requires the generation of a plunger, horizontal, vertical and diagonal lines. A flipper which pivots around a fixed point must also be designed in order to enable user interaction.

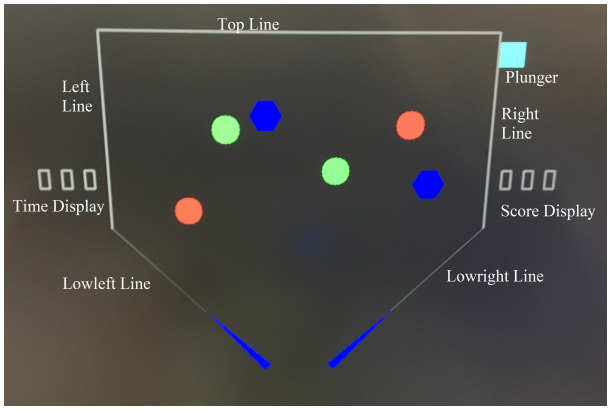

Fig. 8: Photo of all the generated objects displayed on the monitor.

A. Horizontal Line

One of the fundamental building blocks needed to design the game is the generation of horizontal lines. The module should be able to generate a horizontal line at any position, at any length and at any width we desire. To have this functionality the module needs to take the desired horizontal starting and stopping points for the line, the desired vertical position and width of the line as inputs. The module should also take the current active pixel from the vga driver module, test whether it is on the generated horizontal line and give the result of this test as output. In order for the active pixel to be on the

generated horizontal line, it must satisfy the two conditions shown below in Eq. (1) and Eq. (2).

$$x_{\text{start}} \leq x_{\text{active}} \leq x_{\text{stop}} \tag{1}$$

$$y_{\text{position}} - \text{width} \leq y_{\text{active}} \leq y_{\text{position}} + \text{width} \tag{2}$$

Simply, the tested active pixel is on the horizontal line if the x_{active} is between the beginning and ending of the line and y_{active} is within the width of the line. If the active pixel satisfies these conditions, the module will output 1, otherwise it will output 0. This will allow us to display the horizontal line on the screen by using the output of this module in the top module.

B. Vertical Line

It is of fundamental importance to generate vertical lines. This is very similar to horizontal line generation. Since the module must generate a vertical line at any position, with any length and width, the module must take the horizontal position, the vertical starting and stopping locations and the width of the line as inputs. Likewise the current active pixel values will be input to it for testing. For the active pixel to be on the generated vertical line it must satisfy the two conditions shown below in Eqs. (3) and (4) :

$$y_{\text{start}} \leq y_{\text{active}} \leq y_{\text{stop}} \tag{3}$$

$$x_{\text{position}} - \text{width} \leq x_{\text{active}} \leq x_{\text{position}} + \text{width} \tag{4}$$

C. Diagonal Lines

As part of our arena design, we require the generation of diagonal lines. Two kinds of diagonal lines, with a slope of 1 ($45°$) and slope of -1 ($-45°$) need to be generated. The ratio of the horizontal and vertical distances from any point on the lines to another point on the lines is 1 in magnitude, so testing of a pixel is simple. The module should take the starting and stopping positions of the diagonal line and test active pixel position. Our logic and conventions are best described using Fig. 9.

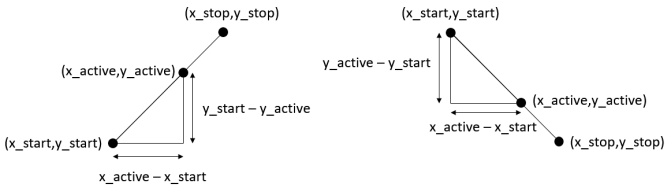

Fig. 9: Schematic representation of diagonal line generation for lines with slope of 1 and -1.

We always choose the leftmost point of the diagonal line as our (x_{start} ,y_{start}) point. After this, we choose our (x_{stop} ,y_{stop}) point depending on the length and slope we desire. Horizontal distance is always calculated as ($x_{\text{active}} - x_{\text{start}}$) and the vertical distance is calculated as ($y_{\text{active}} - y_{\text{start}}$) or ($y_{\text{start}} - y_{\text{active}}$) depending on the slope of the line. Then we check if the horizontal distance is equal to the vertical distance and whether the active pixel is within the square defined by the starting and stopping points of the line.

D. Plunger

The plunger will provide the initial speed and position to the ball. At the start of each game, a button input should be provided by the user to give the ball its initial speed and position and start the game. We simply run another instance of the horizontal line generation module by setting a desirable position, length and width. Through this method we have generated the rectangular plunger object visible in the top right corner of Fig. 8. We have chosen to place the plunger object outside of the borders of the playing field as it should not interact with the ball during gameplay.

E. Score and Time Displays

We need two displays, one to show the current score of the player and one to show the elapsed time. The the operating logic behind the module is very similar to a real world three-digit seven segment display. We display both score and time (in seconds) on three digits as it has a sufficient decimal upper limit of “999”. This number corresponds to around 16 minutes of play time or about 50 consecutive collisions with a scattering target which gives the highest score value of 20 points. Considering that there are targets that also reduce the score of the player, limiting the display to three digits is a reasonable decision. There are two main parts to the module: Firstly, the module must convert the binary data input to a three digit BCD number, and secondly the BCD number created must be displayed on three digits in a desired location.

A 10-bit data input is given to the module, which we should convert to three BCD digits. We know the decimal equivalent of each binary digit, so we merely sum all of the ones digits of the decimal equivalents of the binary digits that are “1” to get the ones digit of the converted BCD number, which is the ones digit of the sum. The maximum sum for the ones digits of the decimal equivalents of the binary digits is 43, which corresponds to a maximum carry of 4 into the tens digit of the BCD number. The tens digit of the BCD number is the sum of all of the tens digits of the decimal equivalents of the binary digits that are “1” plus the carry over coming from the ones digit. This results in a maximum sum of 22 for the tens digit of the BCD number, which corresponds to a maximum carry of 2 into the hundreds digit of the BCD number. Lastly, we generate the hundreds digit of the BCD number by summing all of the hundreds digits of the decimal equivalents of the binary digits that are “1” and the carry from the tens digit. This results in a maximum sum of 10 for the hundreds digit, in which case we cannot display the number and all three BCD digits are set to “9”; otherwise the BCD digits remain as is. An example for the binary to BCD conversion is given in Table. I.

For the display, we simply use instances of the horizontal line and vertical line generator modules in order to form digits similar to seven segment display. A total of 21 modules are used for three digits, with each digit using 7 modules, i.e. 3 vertical line generators and 4 horizontal line generators. All of these “segments” are placed in the appropriate locations using the placement functionality of the vertical and horizontal line generators. We map each BCD value of each digit to the

Decimal Number	613									
Binary Input	1	0	0	1	1	0	0	1	0	1
Decimal Equivalent	512	256	128	64	32	16	8	4	2	1
Ones Digit Sum	1 + 4 + 2 +4 + 2 = 13									
Tens Digit Sum	1 + 3 + 6 + 1 = 11									
Hundreds Digit Sum	1 + 5 = 6									
BCD Number	6			1			3			

TABLE I: BCD to binary conversion algorithm is represented for a binary input corresponding to decimal 613.

appropriate cases where some of the segments are either on (logic 1) or off (logic 0). Then we simply AND the values of these seven segments with the outputs of the seven modules for each digit to get our output. We run two instances of the seven segment display module, one for displaying the elapsed time and one for displaying the score, both of which we generate and update in the top module.

F. Circle

Whether a point is within a circle is computed by calculation of its Euclidian distance to the center of the circle and comparing this distance to its radius as follows

$$(x_{\text{active}} - x_{\text{center}})^2 + (y_{\text{active}} - y_{\text{center}})^2 \leq \text{radius}^2 \qquad (5)$$

All the points whose distance to the center is smaller than the radius is considered to be a part of the circle. When observing the circles on VGA we can see that the circles aren't smooth and are pixelated around the circumference. This is due to the fact that we have a discrete array of integer points for the VGA and the points around the actual geometric circumference correspond to non-integers.

Therefore to define a circle we need its center position (x_{center}, y_{center}) and its radius. The candidate point has the position (x_{active}, y_{active}).

G. Hexagon

We concluded that implementation of the hexagon through a bitmap or by defining multiple regions separately would be difficult and sought a method to define a hexagon through utilizing Euclidian distance, which is easy to calculate on the FPGA. For this purpose we used a result from solid state physics which states that the primitive cell of a triangular Bravais lattice is an equilateral hexagon [2]. The primitive cell of a lattice point is the area which is closest to it than any other lattice point. Although the Bravais lattice is ideally an infinite lattice, we can use 7 lattice points to compute the hexagon as seen in Fig. 10. We compute the distance of a candidate point to all the lattice points and conclude that the candidate point is within the hexagon if the distance to the central lattice point is smaller than the distance to all the other lattice points.

The distance calculation is done in the same manner as Eq. (5) but this time it is compared to the distance to other points rather than the radius. Through such a mathematical definition we obtain flexibility in defining the hexagon. We can obtain any hexagon through defining its (x_{center}, y_{center}) and a parameter specifying its size by defining the distance to other points. For this calculation a problem occurs since we need to utilize $\sqrt{3}$ in our calculations but FPGA doesn't support

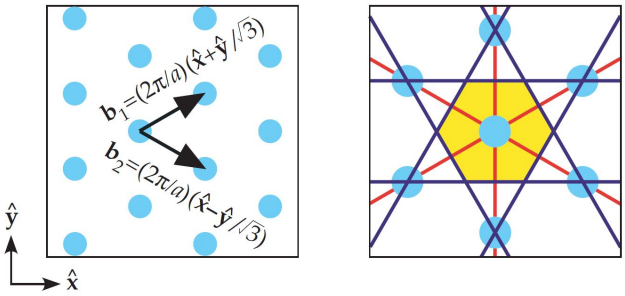

Fig. 10: The geometrical representation of the hexagon primitive cell with the Bravais lattice on the left and the primitive cell obtained from 7 lattice points on the right [3].

floating point numbers. We solve this issue by multiplying every distance by 100 to increase precision and use 173 as an approximation to $100 \times \sqrt{3}$.

H. Flipper

The flipper is an object which pivots around a fixed point defined as (x_{center},y_{center}) as seen in Fig. 11. Since the flipper pivots between $-45°$ and $45°$ with respect to the horizontal line θ varies between $0°$ and $90°$. We use θ to primarily keep track of the position of the flipper. When the button is not pressed, the flipper is at its lowest position and θ is $0°$. When the button is pushed we start incrementing θ by $5°$ every animation cycle until it reaches $90°$ which is the top position of the flipper. The flipper starts dropping whenever the button is no longer pressed and stops at $0°$ which is the bottom position.

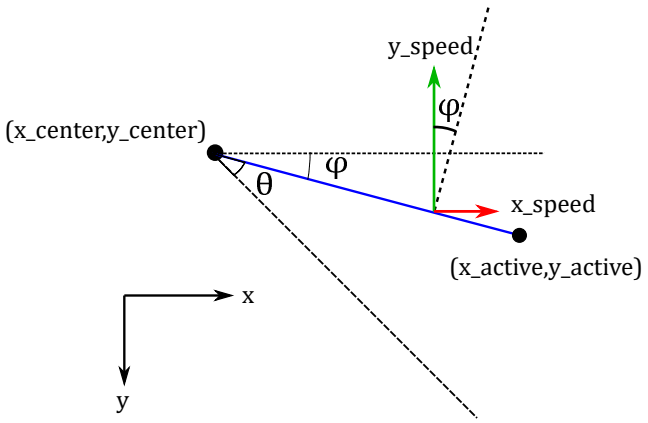

Fig. 11: The schematic representation of the left flipper. Two angles θ and φ are used to represent the position of the flipper with respect to fixed lines.

In order to evaluate whether a candidate point is within the flipper we use multiple conditions. First of all we use a condition relating to the x_{active} of the candidate point, for the left flipper the candidate point must lie to the right of the pivot point which results in $x_{\text{active}} \geq x_{\text{center}}$. Furthermore we have a magnitude condition which relates the distance from the candidate point to the pivot point, similar to Eq. (5)

$$(x_{\text{active}} - x_{\text{center}})^2 + (y_{\text{active}} - y_{\text{center}})^2 \leq \text{length}^2 \tag{6}$$

where length is the length of the flipper.

Another condition follows from the angle of the flipper. For these calculations we use φ which is equal to $45° - \theta$. We know that for θ between $0°$ and $45°$ the flipper is lower than the pivot point which corresponds to $y_{\text{active}} \geq y_{\text{center}}$ and when $\theta > 45°$ the flipper is higher, corresponding to $y_{\text{active}} \leq y_{\text{center}}$. First we create a lookup table for the tangents of possible values of φ with $5°$ increments. Utilizing these values we create the angle condition for $\theta \leq 45°$ as

$$0.9\,tan(\varphi) \leq \frac{y_{\text{active}} - y_{\text{center}}}{x_{\text{active}} - x_{\text{center}}} \leq 1.1\,tan(\varphi) \tag{7}$$

The condition for $\theta \geq 45°$ follows analogously to Eq. (7) where we use $y_{\text{center}} - y_{\text{active}}$ rather than $y_{\text{active}} - y_{\text{center}}$. The values 0.9 and 1.1 are used to create an error margin for the angle condition since calculations utilizing tangent values don't result in exact values corresponding to fixed pixels due to number precision. An advantage of these margins is that the resulting flipper widens as we move further from the pivot point and we only need to use this single module to define a thick flipper. We conclude that a candidate point is within the flipper whenever all the conditions are simultaneously satisfied.

Furthermore we take the output x_{speed} and y_{speed} from our flipper module since the angle of the ball after collision is dependent only on the position of the flipper. Since we need the ball to reach the top of the arena regardless of the position of the flipper we fix y_{speed} to a fixed value sufficient for the ball to hit the top (8 in our case). In order to set the motion of the ball orthogonal to the flipper we use simple geometry visible in Fig. 11 and set $x_{\text{speed}} = tan(\varphi) \times y_{\text{speed}}$.

The operation of the right flipper follows analogously. For the module we input x_{center} , y_{center} , length of the flipper and we obtain an output indicating if a candidate point is within the flipper and x_{speed} , y_{speed} which will be utilized in the interaction and game physics further on.

IV. Game Physics

The game physics is related to the motion of the ball and its interaction with the objects. We define the motion of the ball through four variables x_{dir} (1 for right 0 for left), y_{dir} (1 for down 0 for up), x_{speed} and y_{speed} . The position of the ball is defined through its x_{center} and y_{center} which is input to a circle module. The position is updated every animation frame using the variables as follows

$$x_{\text{center}} = \begin{cases} x_{\text{center}} + x_{\text{speed}}, & \text{if } x_{\text{dir}} = 1 \\ x_{\text{center}} - x_{\text{speed}}, & \text{if } x_{\text{dir}} = 0 \end{cases} \tag{8}$$

$$y_{\text{center}} = \begin{cases} y_{\text{center}} + y_{\text{speed}}, & \text{if } y_{\text{dir}} = 1 \\ y_{\text{center}} - y_{\text{speed}}, & \text{if } y_{\text{dir}} = 0 \end{cases} \tag{9}$$

Without any collision (i.e interaction between ball and other objects) the variables are only updated with respect to the gravity present in the game. In the case of a collision the variables are updated according to the rules of physical interaction which in turn causes the ball to move accordingly.

A. Gravity

When the ball is roaming freely it must be affected by gravity, it must slow down as it is going up and it must speed up as it is going down. The gravity ensures that the ball will eventually reach the flippers. The gravity doesn't effect horizontal motion hence x_{dir} and x_{speed} won't be updated. We will only update y_{dir} and y_{speed} . To do this we define a constant acceleration α depending on the direction of motion as

$$y_{\text{speed}} = \begin{cases} y_{\text{speed}} + \alpha, & \text{if } y_{\text{dir}} = 1 \\ y_{\text{speed}} - \alpha, & \text{if } y_{\text{dir}} = 0 \end{cases} \tag{10}$$

We set α to be equal to 1 at every 4 animation frame to prevent extreme acceleration which causes very high speeds on the game. Furthermore to control the speed of the game we set a maximum limit to y_{speed} as 10, if it reaches 10 while going down it won't further accelerate. Furthermore if the ball reaches sufficiently low speeds while going up it must change direction and start going down. To attain this if y_{speed} = 1 and y_{dir} = 0, we set y_{dir} = 1 at the next time acceleration is applied.

B. Collision Check

We understand that a collision has occured if a pixel (x_{active} ,y_{active}) is both within the ball and another object. This condition enables us to understand which collision happened and update our values accordingly. However, it is a common occurrence for multiple pixels to be shared between colliding objects and, possibly due to low initial speeds, the collision to last multiple frames. In such cases we want the speed and the score to be updated only a single time. Hence we defined the variables "collision" and "previous collision". "collision" checks if a collision has happened within the frame and "previous collision" checks if there was a collision in the previous frame. We only update the speed and the score if both are zero, i.e there wasn't a collision in the previous frame and there hasn't been a collision in the current frame.

C. Circle Collision

The circle collision dynamics are represented in Fig. 12. The dynamics are dependent on the relative positions of the ball and the circular target. The basic principle is that the speed vector after collision should be the negative of the vector from the center of the ball to the center of the target. This is implemented in a conditional manner where the collision dynamics depend upon the respective positions. The dynamics are summarized in Table. II which is for horizontal dynamics and in Table. III which is for vertical dynamics.

Although we directly subtract the center positions in the tables, in our implementation we divided these numbers by

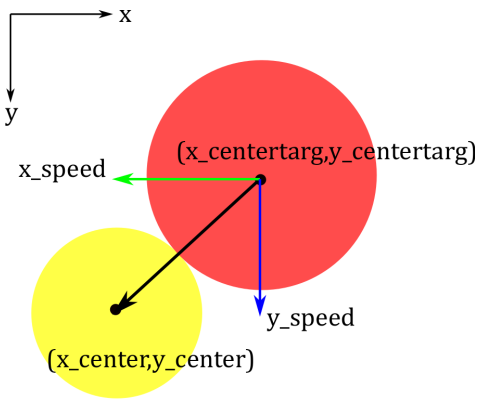

Fig. 12: The schematic representation of the circle collision dynamics. $x_{\text{centertarg}}$ and $y_{\text{centertarg}}$ denotes the center point of the circular target the ball collides with.

	$x_{\text{center}} \leq x_{\text{centertarg}}$	$x_{\text{center}} > x_{\text{centertarg}}$
x_{dir}	0	1
x_{speed}	$x_{\text{centertarg}}$ - x_{center}	x_{center} - $x_{\text{centertarg}}$

TABLE II: Collision dynamics according to the respective x positions of the ball and the circular target.

	$y_{\text{center}} \leq y_{\text{centertarg}}$	$y_{\text{center}} > y_{\text{centertarg}}$
y_{dir}	0	1
y_{speed}	$y_{\text{centertarg}}$ - y_{center}	y_{center} - $y_{\text{centertarg}}$

TABLE III: Collision dynamics according to the respective y positions of the ball and the circular target.

right shifting to obtain lower speed values, otherwise as these distances commonly exceed our speed limits we would have the ball moving at the max speed which is undesirable. We observed that dividing 4 (corresponding to 2 bit right shift) results in acceptable speeds.

D. Random Scattering

In order to perform random scattering, we have decided to utilize random numbers generated from a 17-bit linear feedback shift register. This is a simple shift register, which has been given feedback by using an XOR gate on its 17th and 14th inputs [4]. It is this feedback bit that is random, all other bits are simply shifted to the left and the feedback bit is concatenated at the LSB position. The 17-bit LFSR is capable of generating $(2^{17} - 1)$ random bits before repeating itself, so the values it generates are sufficiently random for our application. We utilize arbitrarily chosen groups of these 17-bits in order to generate a random x_{speed} (3-bits), y_{speed} (3-bits), x_{dir} (1-bit) and y_{dir} (1-bit) each time the ball collides with any of the scattering targets. However, it is not desirable for any of the speeds to be zero, as this may cause issues, so the x_{speed} and y_{speed} values have been set to arbitrary non-zero values in the case that either of them are zero. The generated x_{speed} and y_{speed} values are then extended to 10-bits to match the size of the speeds we use in the top module.

E. Arena Collision

The arena consists of multiple lines which have their own collision dynamics. The lines which form the arena are rightline, leftline, topline, lowleftline (diagonal) and lowrightline (diagonal) as seen in Fig. 8.

1) Rightline: In order for the ball to collide with the rightline it must be moving to the right with $x_{\text{dir}} = 1$, after collision it simply reverses its x_{dir} moving left and we set $x_{\text{dir}} = 0$.

2) Leftline: Leftline dynamics is simply the reverse of the rightline dynamics. After collision we set $x_{\text{dir}} = 1$.

3) Topline: For the ball to collide with the topline it must be moving upwards with $y_{\text{dir}} = 0$. After collision it changes y_{dir} and moves downwards with $y_{\text{dir}} = 1$.

4) Lowleftline: Upon collision with the lowleftline we expect the ball to move upwards and rightwards, hence we set $x_{\text{dir}} = 1$ and $y_{\text{dir}} = 0$. However it isn't sufficient to simply change the directions, for correct reflection from the diagonal line the speeds must also be altered. This is done by considering the reflection as being symmetric with respect to the normal of the lowleft line which corresponds to simply swapping x_{speed} and y_{speed} . Thus we set x_{speed} = y_{speed} and y_{speed} = x_{speed} .

5) Lowrightline: After colliding with the lowrightline the ball should move left and upwards, thus we set $x_{\text{dir}} = 1$ and $y_{\text{dir}} = 0$. Similar to the case with the lowleftline we also swap the speeds to obtain correct motion angles after collision.

V. Top Module

The top module is where the interaction of the submodules, the inputs and the outputs are defined. Our top module has four inputs, 2 push buttons to move the flippers, a push button to activate the plunger and start moving the ball and a switch to reset the game. It also has 6 outputs, horizontal sync and vertical sync signals which provide timing synchronization with the VGA, three 8 bit RGB outputs and a 25MHz clock signal.

The positions of the stationary targets, arena, time and score displays are assigned in the top module. To display the targets, flipper and the ball we use the most significant bits of the RGB outputs. The second most significant bit is used for the arena and the displays. Within the object modules, we test whether a test pixel is within a specific object, this output is used to make logical assignments to the RGB bits according to the desired colors of the objects. Hence for every test pixel we have an RGB output defined according to the object it is a part of.

The updating of the position with respect to the game physics defined in Sec. IV is also accomplished in the top module. The (x_{center} , y_{center}) position of the moving ball is updated and given as an input to the ball object module. According to the collisions the score is also updated here. We take care to update it only once per collision. Furthermore, we don't let the score become negative after hitting penalty targets; it is set to 0 if it hits a penalty target and its present score is less than 15.

In order to update the time we utilize the animate signal from the VGA driver. The animate signal has a frequency of 60 Hz, hence we update the time (in seconds) when a counter which is triggered by the animate signal reaches 60.

VI. Conclusion

We have successfully implemented the game on the FPGA and demonstrated its proper operation. The timing specifications required for establishing connection with the monitor through VGA was an especially unique challenge as we were introduced to different conventions for interconnecting various equipment. We were required to generate very specific signals in order to meet the strict timing and design requirements that were necessary to drive the VGA display. As we were working directly on the hardware the complexity of the design was entirely under our control. Being able to exactly assert the operations to be completed in a single clock cycle presented us with unique opportunities and timing relationships. Furthermore, the stringent requirements associated with the FPGA led us to search for and come up with innovative solutions. As we couldn't implement division on the FPGA we had to resort to right shifts whose accuracy was significantly limited, also we used unique tricks to deal with problems that required the use of floating numbers. We observed that certain calculations were more suited to the FPGA and tried to perform our computations in a way that maximized the usage of such calculations. For example, the primitive cell hexagon solution heavily relies upon multiplication and comparisons, which the FPGA is very proficient in. In conclusion, this has been a very challenging yet fruitful project that has taught us much as engineers.

References

[1] W. Green, "Fpga vga graphics in verilog part 1," Dec 2017. [Online]. Available: https://timetoexplore.net/blog/arty-fpga-vga-verilog-01

[2] C. Kittel, P. McEuen, and P. McEuen, *Introduction to solid state physics*. Wiley New York, 1976, vol. 8.

[3] J. D. Joannopoulos, S. G. Johnson, J. N. Winn, and R. D. Meade, *Photonic Crystals: Molding the Flow of Light (Second Edition)*, 2nd ed. Princeton University Press, 2008.

[4] P. Alfke, "Efficient shift registers, lfsr counters, and long pseudorandom sequence generators," Jul 1996. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp052.pdf