A Game, Fake Quidditch, in Verilog HDL

Furkan Şentürk
Electrical and Electronics Engineering
Middle East Technical University
Ankara, Turkey
e216724@metu.edu.tr

Güray Özgür
Electrical and Electronics Engineering
Middle East Technical University
Ankara, Turkey
e216705@metu.edu.tr

***Abstract*—This document is a final report of a project, which is an implementation in Verilog HDL. In this implementation, aim is to make a game that two players can play on FPGA. Furthermore, FPGA VGA graphics are used to display the game in a screen. Explanations and testbench simulations of all the modules are presented in this report.**

***Index Terms*—Verilog HDL, 5CSEMA5F31C6, Quartus, Digital Circuits Labarotory, EE314, METU**

I. Introduction

In this project, we will write a Verilog HDL code that allows us to play Fake Quidditch by using FPGA programmed by this code and a monitor to show us the game screen. Fake Quidditch is a game inspired by Harry Potter's quidditch. However, it has some differences from the quidditch. It will last 180 seconds and will be played by two teams, each has two players. One of the players in the team will move only horizontally, while other one will move vertically. Apart from that, there will be two balls; main ball and bludger. The main ball will be moved by players by hitting it. The bludger will move semi-randomly, i.e. it will be moved by players' hit, however, the scattering resulting from this collusion will be random, whereas the main ball scatterings will be based on the basic physical principles. When bludger collides one of the players, this player will be inactive for 10 seconds. There are three goals for each team placed at a fixed position on the game field. When main ball reaches one of the goals, if the main ball can be encircled by the goal properly, it will be score for the other team. After, each score, players, main ball and bludger will return their initial positions. There will be an information field apart from the game field. In this field, there will be a score board, a counter showing how much time left, and 4 counters which will indicate how much time left for a player being inactive. Lastly, players cannot pass thru each other, since they are rigid bodies.

In order to create such a code, we will first divide all parts that should be included code module by module. After each module is completed, we will combine them by using a top module. The works each module will be responsible are selected by us. The above explanation of the Fake Quidditch is given specifications to us. The other properties of the game which are determined by us will be introduced in the Modules section of this report at where they are determined.

II. Modules

We have created 11 different modules in this project; one top module to combine the submodules, three submodules to control the VGA, and seven submodules that determines what happens in the game.

A. Top Module

In this module, we are connecting every single module to each other. There are inputs that taken from outside, i.e. buttons or switches, and outputs which connected to VGA port. In addition, there are inputs and outputs that are taken form a submodule and given another one which are provided by wires.

B. Collision Detection Module

In this module, we determine whether the objects collide with each other or not, and the velocities of main ball and bludger. There are thirteen inputs; *clk*, *reset*, *restart*, *player1_y*, *player4_y*, *player2_x*, *player3_x*, *random_x*, *random_y*, *mainball_x*, *mainball_y*, *bludger_x*, *bludger_y* and five outputs; *x_shift*, *y_shift*, *bludger_x_shift*, *bludger_y_shift*, *crush*.

Clock signal, *clk*, is the 50 MHz signal that FPGA generates. Collision detection module gets it and dividing it by 250 000, it generates output four hundred times in every second. If *reset* input is high or *restart* input is low, then all players returns their initial positions. Random inputs are taken from random number generator modules. They are used to find the velocity of the bludger. Each one of them determines the velocity in the corresponding axis. Other remaining inputs are giving us the center points of the objects. We will use them to make calculations respectively. There are fourteen different collisions between two objects. Although it seems that 4 bit register is enough to remember them, that is not the case, since one object can collide with multiple objects at the same time. Therefore, we define a 14 bit register to carry this information between modules. Each bit of the crush corresponds different objects collision. The collision took place between objects shown by each bit of crush is given in Table I.

TABLE I
CRUSH BIT INFORMATION

Crush register	Collisions took place
crush[0]=1	Player 1 - Main Ball
crush[1]=1	Player 2 - Main Ball
crush[2]=1	Player 3 - Main Ball
crush[3]=1	Player 4 - Main Ball
crush[4]=1	Player 1 - Bludger
crush[5]=1	Player 2 - Bludger
crush[6]=1	Player 3 - Bludger
crush[7]=1	Player 4 - Bludger
crush[8]=1	Main Ball - Bludger
crush[9]=1	Player 1 - Player 4
crush[10]=1	Player 1 - Player 2
crush[11]=1	Player 3 - Player 4
crush[12]=1	Player 1 - Player 3
crush[13]=1	Player 2 - Player 4

In order to find whether a collusion occurs or not, we will use Equation 1. The distance value is selected different for different cases. Since we know the radii of all objects, we will select the distance according to them. For example, while checking collusion between a player and the bludger, since the radii for them are 20 and 10, we will select the distance d^2 at the Equation 1 as 961.

$$d^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2 \qquad (1)$$

For bludger's velocity, we will use the numbers that are generated in the random number generator modules. To do the random scattering correctly, we should consider that, the bludger do not pass through other objects and field boundaries, therefore, we are not completely free to determine bludger's velocity. Therefore for each object we will use the velocities determined in the Figure 1 for scattering the objects. In this figure as you can see we arrange each velocity vector such that the bludger goes to the outside of the objects it collide. For scattering field boundaries, we will use the same logic. The Figure 2 shows the direction of the velocity vectors at each boundary.

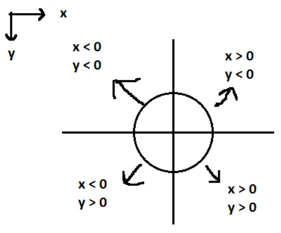

Fig. 1. Velocity directions for objects

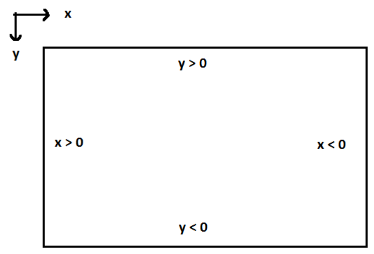

Fig. 2. Velocity directions for field boundaries

For main ball, we cannot obtain a proper way to reflect main ball from players while keeping principle of physics since division in Verilog HDL is problematic. We try many ways to solve this problem like writing a divider module or expanding/shifting the fraction. Therefore, at this stage of the process we adjust the main ball's movement based on the same logic we used for bludger.However, we only used the direction determination method of the bludger which is shown in the Figure 1. After that, we just give speed of 1 in each direction.

C. Player Module

In this module, we determine the movements of the players. Players movements are restricted by either other objects or the field boundary. There are twelve inputs; *clk*, *reset*, *restart*, *crush*, *up1*, *down1*, *left2*, *right2*, *left3*, *right3*, *up4*, *down4* and eight outputs; *player1_y*, *player2_x*, *player3_x*, *player4_y*, *count1*, *count2*, *count3*, *count4*.

Clock signal, *clk*, is the 50 MHz signal that FPGA generates. Player module gets it and dividing it by 250 000, it generates output four hundred times in every second. If *reset* input is high or *restart* input is low, then all players returns their initial positions. *up1*, *down1*, *left2*, *right2*, *left3*, *right3*, *up4*, *down4* inputs give us whether the buttons or switches are active or not, which will be the **move** command. Collusion information between the players are determined through *crush* input. For each player, we check all possible collusion cases. It makes eight cases for Player 1 and Player 4 and four cases for Player 2 and Player 3, total of twenty four cases. The possible collisions between the players are stored in *crush* input and *crush*'s bit information is given in Table II.

TABLE II
COLLISION CASES

Players	P1	P2	P3	P4
P1		10	12	9
P2	10			13
P3	12			11
P4	9	13	11	

*$crush[\cdot]$=1 is indicated.

Inside each case, we will check the move command for the player and if it is acceptable, the player will move. Note that, in each collusion case, we will check also whether the players reach the field boundary or not. If so, then we will check move commands' validity for these situations as well. For obtaining a more simple code, we will leave unused cases and unacceptable move commands empty. That is, for example it is not possible for Player 1 to collide Player 2 and Player 3 at the same time. Therefore, we will not write anything for this case, or if Player 1 collides with Player 2 or Player 3, we do not have to check if the player one will stay in field boundary or not due to the positions of our players. Player 1 and Player 4 will change their y axis positions while Player 2 and Player 3 will change their positions in x axis. The unchanged positions of the players are determined by us. After reaching an acceptable move command, we will add one to the position of the player receiving the command so that it can move one bit at a time. These positions are the outputs; *player1_y*, *player2_x*, *player3_x*, *player4_y*. There will be a counter for each player which will count 10 seconds if the bludger and a player collide, and this player will wait for the punishment to end. The crush bit that keeps the information about collision happened is given in the collusion detection module, Table I. To adjust this counting process for punishment of each player, we will assign four different outputs, i.e. *count1*, *count2*, *count3*, *count4* in this unit. Since the bludger will leave the player that it crushed immediately, when the collusion occurs, we will assign the corresponding counter to 2000. Then we will decrease it by 1 in each time. Since we calculate players positions 200 times at a second, after 10 second the counter will be 0, which will enable player to move. Then we will give outputs *count1*, *count2*, *count3*, *count4* to the VGA draw module to show the remaining punishment time.

D. Main Ball Module

In this module, we will determine the main ball's movement. There are five inputs; *clk*, *reset*, *restart*, *x_shift* and *y_shift*, and two outputs; *mainball_x* and *mainball_y*. Clock signal, *clk*, is the 50 MHz signal that FPGA generates. Main ball module gets it and dividing it by ten million, it generates output five times in every second. If *reset* input is high or *restart* input is low, then main ball returns its initial position. To find its next position, we take the inputs *x_shift*, *y_shift* and add them to old positions of the main ball and get the new positions, which are *mainball_x* and *mainball_y*. In a way, *x_shift* and *y_shift* are the velocity components of the main ball, and they are determined by collision detector module.

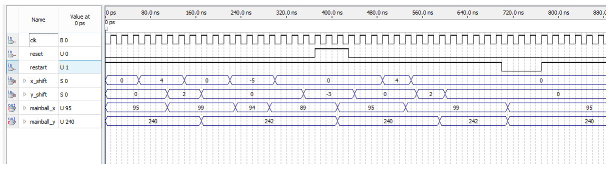

Fig. 3. Main Ball's position with different velocities

In the Figure 3, mainball's position is shown. As we can see it's x and y coordinates changes at every clock when we change x_shift and y_shift, velocity of the ball. If we get reset or restart active, then the position return its original location.

E. Random Number Generator Modules

In this modules, we generate a four bit random number, since we need a randomness in movement of bludger. There is only one input, *clk*, and only one output *number*. Fİrst, we create a register, *rand*, and assign it initially a random number, we determine. This will be our seed. By selecting random four of its bits, we create two new registers by bit-wise xor operation. We add those two bits to the tail of *rand*, and choose our random generated number as random four bits of *rand*. By changing the seed we can create a different random number generator and that is what we used to have a randomness.

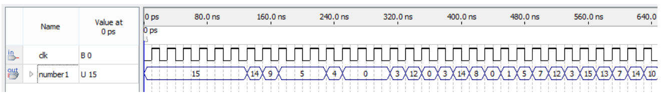

Fig. 4. Random Number Generator

In the Figure 4, we can see it generates different numbers at different times. By consruting with two different seeds we can obtain two different random number at each time.

F. Bludger Module

In this module, we will determine the bludger's movement. To do it, we will consider its place on the screen and with respect to other things. Then by adding (or substracting) some quantities that from bludger's present position, we will find its next position. There are six inputs; *clk*, *reset*, *restart*, *crush*, *bludger_x_shift* and *bludger_y_shift*, and two outputs; *bludger_x* and *bludger_y*.

Clock signal, *clk*, is the 50 MHz signal that FPGA generates. Bludger module gets it and dividing it by ten million, it generates output five times in every second. If *reset* input is high or *restart* input is low, then bludger returns its initial position. Crush input determines whether the bludger collide with any objects or not, and it stops for a while if it collides with main ball. *bludger_x_shift* and *bludger_y_shift* determines the velocity of the bludger. By adding these two to the present place of the bludger, we can find next position of the bludger, which is determined by outputs *bludger_x* and *bludger_y*.

G. Score Detection Module

In this module, we will detect whether any team gets a score or not. We will determine this by checking main ball's position. If it enters a goal, we will give the score to the team that gets the point. There are four inputs; *clk*, *restart*, *mainball_x* and *mainball_y* and three outputs; *reset*, *score1*, *score2* in this module.

Clock signal, *clk*, is the 50 MHz signal that FPGA generates. Score detection module gets it and by dividing it with ten million, we check main ball's position 5 times in each second. *restart* is generated by timeleft module. If *restart* is high, it means that the game is started again so, we reset the scores. *mainball_x* and *mainball_y* is generated by main ball module which gives us where the center of the main ball in the screen. Then, by comparing these positions distance with each goal's center we detect when there is any score change. To find the distance between two center, we use Equation 1 for each case.

reset is generated in this module as output. If any team gets a point, the balls and the players will return their initial positions. It provides us to increase the scores ones at a time. if we do not define such a parameter, there will be multiple point increasing when the main ball enters one of the goals and moving in it. *score1* and *score2* are generated in this module as outputs. They give us the current scores of each team. To detect whether a team gets a point, each goal is named, and the corresponding places of the goals are shown in the Figure 5. If main ball is in one of the goals 11, 12 or 13, it means that Team 1 gets a point so, we add one to *score1*. If main ball is in one of the goals 21, 22 or 23, it means that Team 2 gets a point we add one to the *score2*.

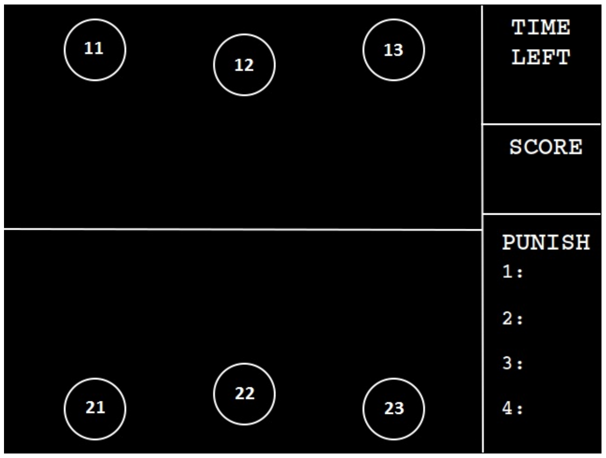

Fig. 5. Position of the goals in the game field

H. Time Left Module

In this module, we will determine time left before game ends. To do it, we will use a counter however, since we should show it in the screen later, we will do this by arranging three different digits; *left2*, *left1*, and *left0* at each time to show remaining time. Therefore, there are three inputs; *clk*, *reset*, *restart* and three outputs; *left2*, *left1*, *left0* in this module.

Clock signal,*clk* , is the 50 MHz signal that FPGA generates. Time left module gets it and dividing it by 50 million, it generates output once in every second.*reset* is generated by score detection module and when it is high, time left module waits for a second without producing a new output combination.*restart* is taken as input and if it is high, the outputs returns their initial values. *left2*, *left1*, *left0* are produced such that the combination of them gives us the remaining time. In order to get a count down starting from 180, we initially assign each digit of this number to the outputs. Then by time we change them. If *left1* and *left0* are 0 then, they become 9 and *left2* is decreased to 0. If *left0* is 0 and *left1* is nonzero, then *left0* becomes 9 and *left1* decreases by 1. If all of them are 0, then the time left stops counting.

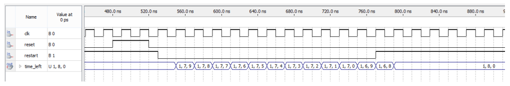

Fig. 6. Time Counter

In Figure 6, three different digits counts simultaneously which creates a general counter that counts down from 180. Moreover, every time restart being active , the counter restarts itself and starts again.

I. VGA Clock Generator Module

In this module, we generate a 25 MHz clock to run the VGA, since it works with that frequency. There is only one input; *clk*, and only one output; *VGA_clk*. Clock signal, taken from FPGA and *VGA_clk* is given as output. In the Figure 7 below, we can see the generated VGA_clk has 25MHz frequency.

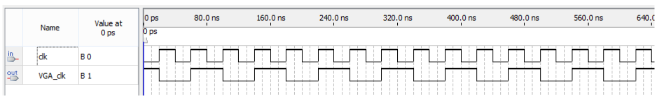

Fig. 7. Waveform of generated VGA clock

J. VGA Controller Module

VGA is used to display the game, i.e. to deliver video signals from FPGA in a monitor. A 640x480 image, a frame containing 480 lines which are made up of 640 pixels, should be delivered. The monitor displays the image starting from its first pixel of the first line and then scans all the pixels at that line. In each line, the first pixel is the leftmost one and the first line is the top of the screen. Other than generating the image buffer, we need to adjust two synchronization signals, which are *HSync* and *VSync*, horizontal and vertical synchronizations. These signal are there to determine end of the line and frame. Therefore, there are two inputs; *clk*, *reset*, and five outputs; *HSync*, *VSync*, *display*, *X*, *Y* in this module. We can imagine our screen is the green region in Figure 8, and there are some blank spaces, which do not correspond to an image. In yellow regions, *HSync*, and *VSync* goes high, and our other outputs, *X* and *Y*, determines our location and gives an output of *X*=0 and *Y*=0 when they are at the top left corner of green region. Furthermore, *display* becomes high for active pixels.

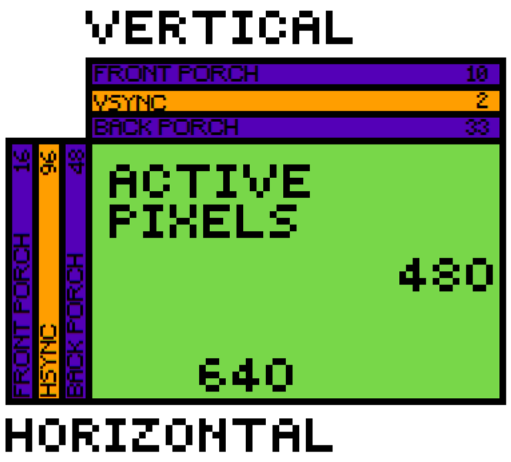

Fig. 8. VGA timings

K. VGA Draw Module

In this module, we begin to draw the screen. First, there will be a background which is fixed, and it will be passive. That is, if there is not any other object placed at the position, the screen will give the background. We draw the background by hand and then by using MATLAB, we create a *.mem* file that consists of the colour codes of each pixel. By reading this *.mem* file in Verilog HDL, we can introduce the background to our code. The background image is given in the Figure 9.

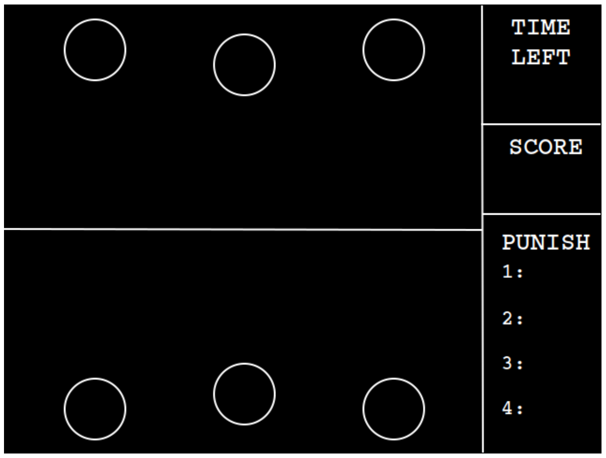

Fig. 9. Background image used in the game

If there is an object in the screen at a position, it will have priority, and displayed in the screen. We will get the positions of all the objects that moves around and will draw them in the screen. We will also get all counters' output and they will also be shown in the screen. To show the number in the screen we will define their places at first. Then, according to the inputs coming the numbers will appear. There are twenty-one inputs; *clk*, *display*, *x*, *y*, *mainball_x*, *mainball_y*, *bludger_x*, *bludger_y*, *player1_y*, *player2_x*, *player3_x*, *player4_y*, *score1*, *score2*, *left0*, *left1*, *left2*, *count1*, *count2*, *count3*, *count4* and three outputs; *red*, *green*, *blue*.

Clock signal, *clk* is a 25 MHz signal which is generated in VGA clock generator module. *x* and *y* are taken from VGA controller module. They are used to define which pixel we are in. Display is also taken from the same module. We use it to understand whether colours should be displayed or not. *mainball_x*, *mainball_y*, *bludger_x*, *bludger_y*, *player1_y*, *player2_x*, *player3_x*, *player4_y* used to detect the center position of the objects. After we find the center positions, by using Equation 1, which was the distance equation, a unique colour code to the pixels that will be turned on for objects. *score1*, *score2*, *left0*, *left1*, *left2*, *count1*, *count2*, *count3*, *count4* are taken as inputs and they will show us one decimal digit that will be displayed on the screen. In order to show a digit on the screen, we get help of the logic of seven segment display module. We put seven different segments to the screen for each digit. Then, by activating different segments for different numbers, we will arrange the numbers that should be shown in the screen. After that, for each number we will assign a white colour. We assign each segment a place in the screen such that there will be a consistence between the background and numbers. We already show which number shows which counter by writing them on the background. The initial screen of the game is shown in Figure 10 and initial positions of all players, counters and balls are shown in the same figure as well.

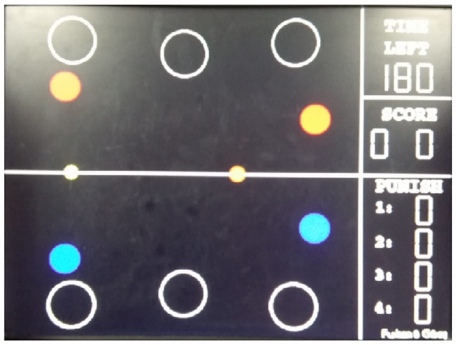

Fig. 10. Initial screen appearing in the monitor

VGA needs only three connections to obtain a colour, these are R, G, and B. Therefore, the outputs red, green, and blue are adjusted according to the displayed objects. Table III shows the colour codes for each object.

TABLE III
COLOUR CODES FOR THE OBJECTS

	Red	Green	Blue
Player 1	0	0	255
Player 2	0	0	255
Player 3	255	0	0
Player 4	255	0	0
Main Ball	255	255	0
Bludger	210	105	30

According to the this table we arrange the colours for objects and other things that takes place in the screen like words by assigning to red, green and blue corresponding quantities.

III. Conclusion

In this project, we write a Verilog HDL code and by using an FPGA and a screen, we try to obtain a game, Fake Quidditch. While writing this game, we create submodules that each one has its own duty. By making required connections between them, we allow submodules to interchange information. After that, we connect every submodule to each other with a top module. As a result of that, we obtain a more simple tasks for each module and the observability of each modules is increased, i.e. we can understand the operations of each module and the code portions that responsible for an operation in the submodule. While writing the Verilog code, we should consider all possible effects of discrepancies like delays of a circuitry part since what we are doing is connecting circuit elements basically. Moreover, while we are operating a module, we should consider what other modules did since we get inputs or give outputs to some other modules. Therefore, we should consider the clock accordance in order not to get an unwanted result. All in all, we implement the code we write to the FPGA and play the game we create.

References

[1] EE314 DIGITAL ELECTRONICS LABORATORY SPRING 2018-2019 TERM PROJECT, METU, Academic.

[2] Readler B. " Verilog by Example: A Concise Introduction for FPGA Design", Full Arc Press 2011

[3] Chu, Pong P.(FPGA Prototyping by Verilog Examples (Chu/FPGA) —— VGA Controller I: Graphic, 2008, pp 309-340.

[4] Terasic Technologies Inc, Altera University Program, De1-SoC User Manual, 2014