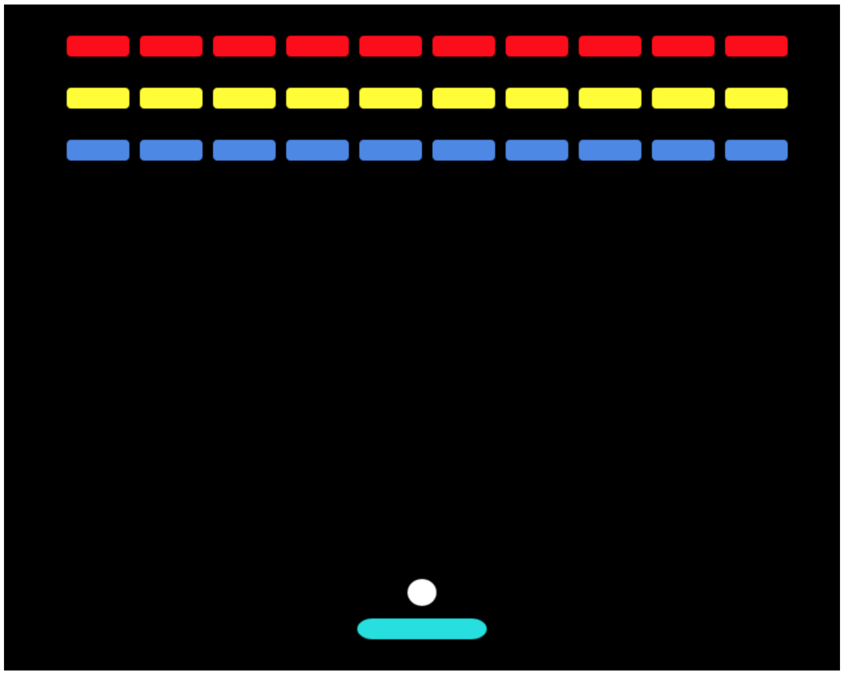
Majid Zare, SID #: 1864882
Leona Burk, SID #: 1742363
Due: 5/21/2020

Lab 6: Breakout Game

- ## <u>Design Procedure</u>

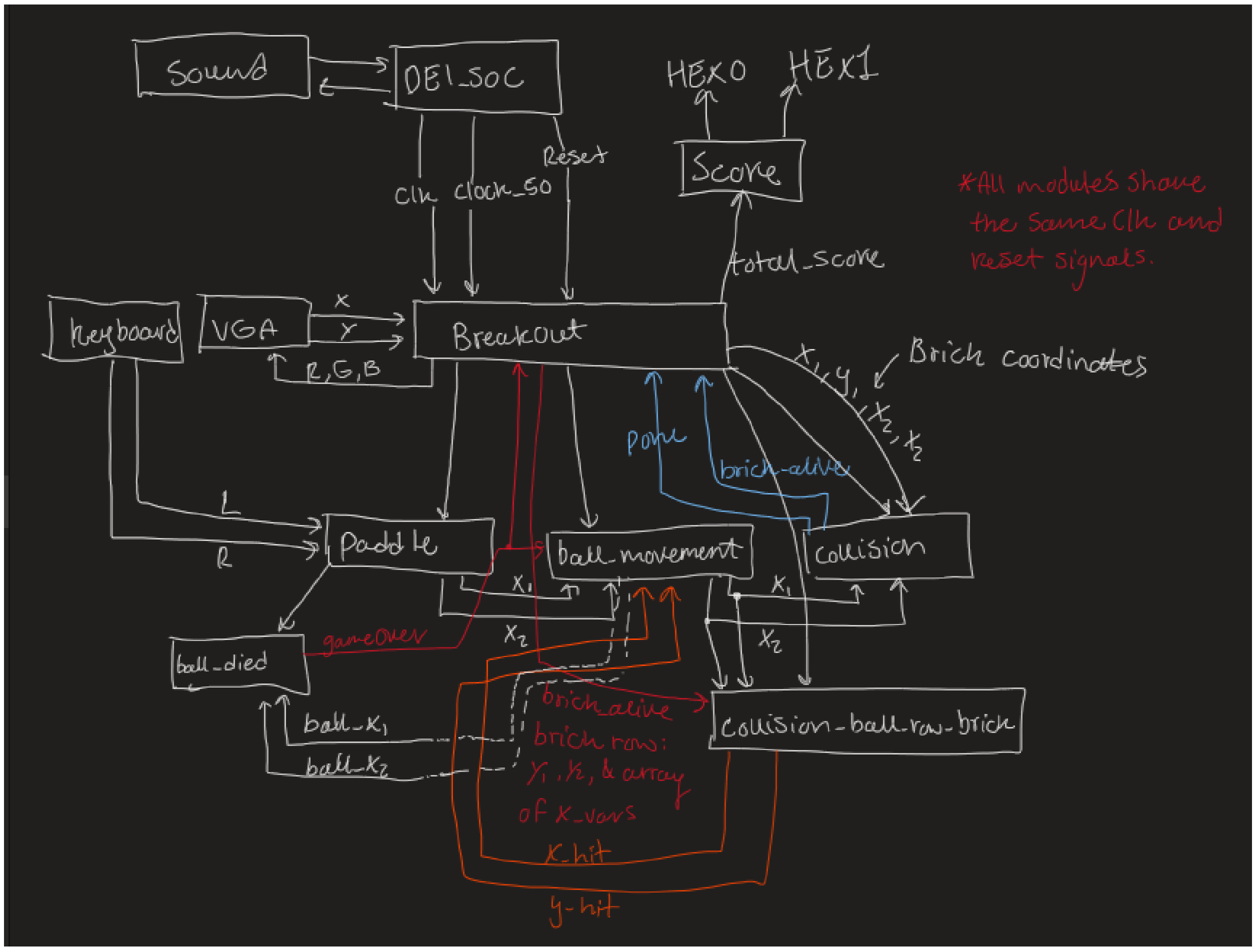
We implemented the game Breakout after the classic Atari game released in 1976.



This version of Breakout that we implemented, is a single player game that has one level with 50 bricks, a paddle, and a ball. The goal is to destroy every brick displayed in the game with the ball. The difficult part of this game is keeping the ball from hitting the bottom of the screen. The ball must only ricochet off the paddle. Usings the left and right keys on the keyboard, you can move the paddle to where you think the ball will fall next. The score of the game is kept on the HEX0, and HEX1 of the DE1_SoC board. For every brick destroyed, the player is awarded one point, and there is a total of 50 points available to be awarded. If the ball ever touches the bottom of the screen, the game is over, and the player is presented with a red screen. Likewise, when the player destroys all 50 blocks, they are presented with a fully green screen, and a score of 50.

This game was implemented within multiple stages. First, we figured out how to display the bricks onto the screen, then the paddle with moving capabilities, and finally the ball movement and collision with the blocks and paddle.

Block Diagram of whole system:



VGA module:

The VGA drivers were the most important module provided by the instructor. The driver routinely scans over all pixels of the screen and outputs it's current x, and y values. The driver takes in as an input R,G,B values that dictate what combination of colors is shown at every pixel. Each color is an 8-bit logic. When the logic equals 8'b0, there is no color shown. As the 8-bit value increases, the corresponding color increases in intensity. Combinations of R,G,B values can be used to create many different colors.

Breakout module:

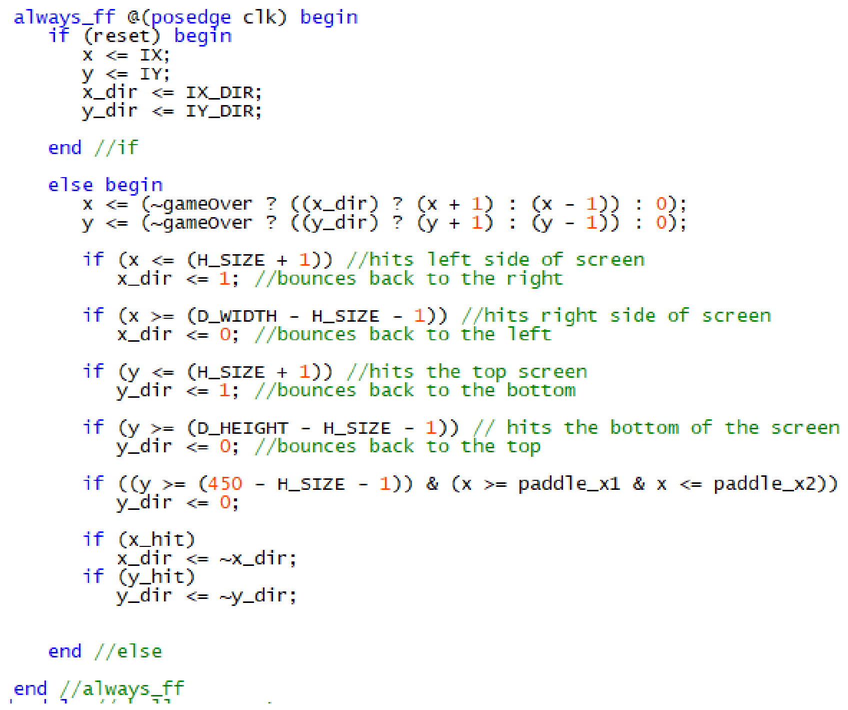
This is the top-level module that instantiates all of the game logic. In this module we instantiated all x, and y values of the 50 bricks with respect to the 640x480 aspect ratio. Then, to display a brick, we created 50 brick boolean variables. When the x, and y outputted by the VGA module was within the range of each brick's coordinates, the brick variable would evaluate to true.

```
// brick initialization
genvar i;
   generate
      for (i = 0; i < 10; i++) begin : BlockInitial
         assign brick[0][i] = ((x > x1[i]) & (y >  y_var[0]) & (x < x2[i]) & (y < y_var[1]) & brick_alive[0][i] & ~gameOver);
         assign brick[1][i] = ((x > x1[i]) & (y >  y_var[1]) & (x < x2[i]) & (y < y_var[2]) & brick_alive[1][i] & ~gameOver);
         assign brick[2][i] = ((x > x1[i]) & (y >  y_var[2]) & (x < x2[i]) & (y < y_var[3]) & brick_alive[2][i] & ~gameOver);
         assign brick[3][i] = ((x > x1[i]) & (y >  y_var[3]) & (x < x2[i]) & (y < y_var[4]) & brick_alive[3][i] & ~gameOver);
         assign brick[4][i] = ((x > x1[i]) & (y >  y_var[4]) & (x < x2[i]) & (y < y_var[5]) & brick_alive[4][i] & ~gameOver);
      end // BlockInitial
   endgenerate
```

Then, we devised a way to be able to toggle the bricks on and off using a variable named “brick_alive”. Each brick has its own brick alive variable, and will evaluate to false when the ball collides with a brick.

Generate statements were an integral piece of this project, because many tasks and initializations required over 50 lines of code, however through generate for loops, we lessened each to around 10 lines of code. As depicted above, we used a for loop to initialize all 50 elements in our 2D brick array. Each row has 10 bricks, and there are a total of 5 rows, which all adds up to 50 bricks. For loops were also used to initialize the x and y positions for each brick in an automated way.

Paddle_movement:
This module controls where the paddle is displayed on the screen with respect to the user controls. This simple task was implemented by hard coding an initial position for the paddle, and adjusting that position using user-inputs passed in as “L” or “R”. Whenever the user inputs ‘L’, the paddle is shifted left 10 pixels, and vise-versa when ‘R’ is inputted. The paddle stops moving once it hits the border of the 640x480 screen.
Ball_movement module:
The ball_movement module is arguably one of the most important piece of the breakout game. We had to implement a way to move the ball around, and make it bounce appropriately off objects such as the bricks, paddle, and the screen borders (walls). To achieve this task, we used variables named y_dir, and x_dir to indicate the vertical and horizontal directions of the ball respectively.



```
always_ff @(posedge clk) begin
   if (reset) begin
      x <= IX;
      y <= IY;
      x_dir <= IX_DIR;
      y_dir <= IY_DIR;

   end //if

   else begin
      x <= (~gameover ? ((x_dir) ? (x + 1) : (x - 1)) : 0);
      y <= (~gameover ? ((y_dir) ? (y + 1) : (y - 1)) : 0);

      if (x <= (H_SIZE + 1)) //hits left side of screen
         x_dir <= 1; //bounces back to the right

      if (x >= (D_WIDTH - H_SIZE - 1)) //hits right side of screen
         x_dir <= 0; //bounces back to the left

      if (y <= (H_SIZE + 1)) //hits the top screen
         y_dir <= 1; //bounces back to the bottom

      if (y >= (D_HEIGHT - H_SIZE - 1)) // hits the bottom of the screen
         y_dir <= 0; //bounces back to the top

      if ((y >= (450 - H_SIZE - 1)) & (x >= paddle_x1 & x <= paddle_x2))
         y_dir <= 0;

      if (x_hit)
         x_dir <= ~x_dir;
      if (y_hit)
         y_dir <= ~y_dir;


   end //else

end //always_ff
```

Variables used:

IX_DIR >> Initial horizontal movement (parameter)

IY_DIR >> Initial vertical movement (parameter)

IX >> Initial x coordinate of the ball

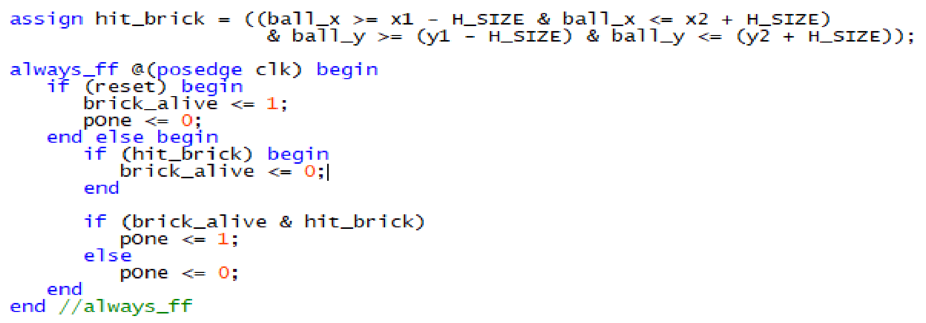
IY >> Initial y coordinate of the ball

H_SIZE >> Half the size of the ball in pixels
As shown in the picture above, when x_dir evaluates to true, the ball will move to the right, and left if it is false. Similarly, when y_dir evaluates to true, the ball will move up, and down otherwise.

In this module, we also handled collision of the ball with the paddle so that the ball will bounce when they collide. Lastly, to change the ball's direction when it collides with bricks, the module takes in the boolean variables x_hit, and y_hit which indicate where the ball hits a brick, and the ball bounces accordingly.

We chose this type of setup for the ball dynamics after lots of trial and error, because this method provided the most efficient way to handle the ball movement, and ball collision with objects all together.

Collision module:

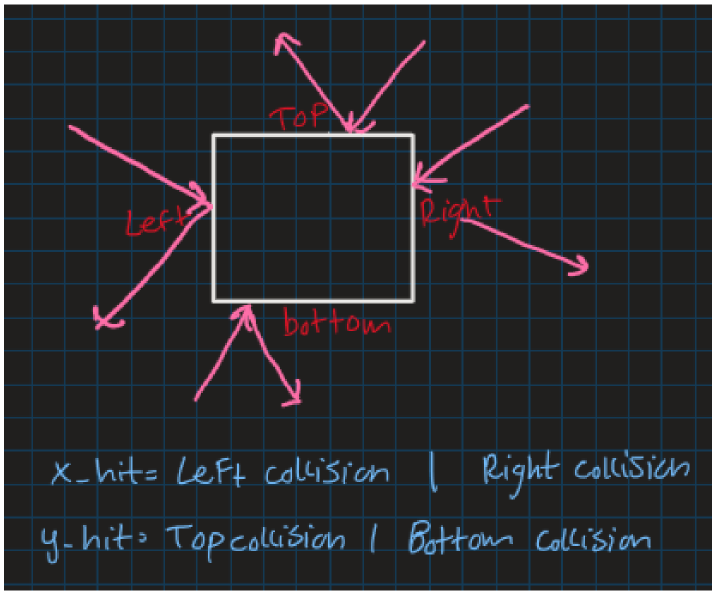


```
assign hit_brick = ((ball_x >= x1 - H_SIZE & ball_x <= x2 + H_SIZE)
                    & ball_y >= (y1 - H_SIZE) & ball_y <= (y2 + H_SIZE));

always_ff @(posedge clk) begin
    if (reset) begin
        brick_alive <= 1;
        pone <= 0;
    end else begin
        if (hit_brick) begin
            brick_alive <= 0;
        end

        if (brick_alive & hit_brick)
            pone <= 1;
        else
            pone <= 0;
    end
end //always_ff
```

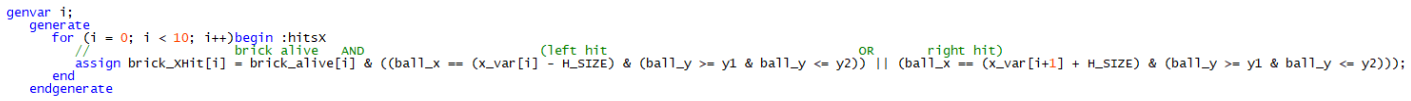
We handled collision with bricks in two modules called collision and collision_ball_row. The reason this was done was to divide the task of "breaking" a brick, and the ball's collision with each brick. The collision module also handled the incrementation of the player's score when collision occurred.

Within the breakout module, a generate for loop was used to instantiate this module 50 times, because the collision module is created to handle a brick at a time. As depicted in the picture above, the ball_x and ball_y coordinates are taken in along with the x1,y1,x2,y2 of each brick to determine when a collision occurs. When that happens, the 'brick_alive' variables corresponding to that brick will be set to false, making that brick disappear from the screen. Because of this decision, that means 'brick_alive' will also be instantiated within the collision module, because a variable cannot be assigned in two different places.

Collision_ball_row_brick module:
After handling the "breaking" of a brick after it collides with the ball, we had to handle the balls bouncing dynamics during such a collision. To handle this case, we implemented a module that takes in ball_x, ball_y, and an entire row of brick x, and y values, along with their corresponding brick_alive variables. Any time a brick is hit within its respective row, the module decides how the ball should bounce by returning x_hit and y_hit.


Top
Left
Right
bottom
x_hit= Left collision | Right collision
y-hit= Top collision | Bottom collision

In the figure above, we can see some potential ways a ball can collide with a brick. As we can see, the ball bounces differently based on which side of the brick it collides with. When the ball collides with either the left or right side of a brick, its x_dir will become its opposite value, and every time the ball collides with the top or bottom sides of a brick, the y_dir will also become its opposite value.



```
genvar i;
   generate
      for (i = 0; i < 10; i++)begin :hitsx
         //                   brick alive   AND                        (left hit                                 OR       right hit)
         assign brick_xHit[i] = brick_alive[i] & ((ball_x == (x_var[i] - H_SIZE) & (ball_y >= y1 & ball_y <= y2)) || (ball_x == (x_var[i+1] + H_SIZE) & (ball_y >= y1 & ball_y <= y2)));
      end
   endgenerate
```

This module was also difficult to implement, because it required very precise mathematical expression to determine where the ball collided with a brick. Using this module along with collision, we were able to make a brick “break” when it collided with a ball.

Keyboard:

The keyboard drivers were provided by the instructor. Given the choice of two keyboard interfaces, we used the simpler keyboard_press_driver, because it allowed us to create a simple state machine to determine when the user presses the left arrow or right arrow on their keyboard.

```
always_comb begin
   case (ps)
      Idle: begin
         if (makeBreak & ((outCode == 8'h61) || (outCode == 8'h6B)))
            ns = Left;
         else if (makeBreak & ((outCode == 8'h6A) || (outCode == 8'h74)))
            ns = Right;
         else
            ns = Idle;
      end

      Left: begin
         if (~makeBreak)
            ns = Idle;
         else
            ns = Left;

      end

      Right: begin
         if (~makeBreak)
            ns = Idle;
         else
            ns = Right;

      end
   endcase
end

assign L = (ps == Left);
assign R = (ps == Right);
```

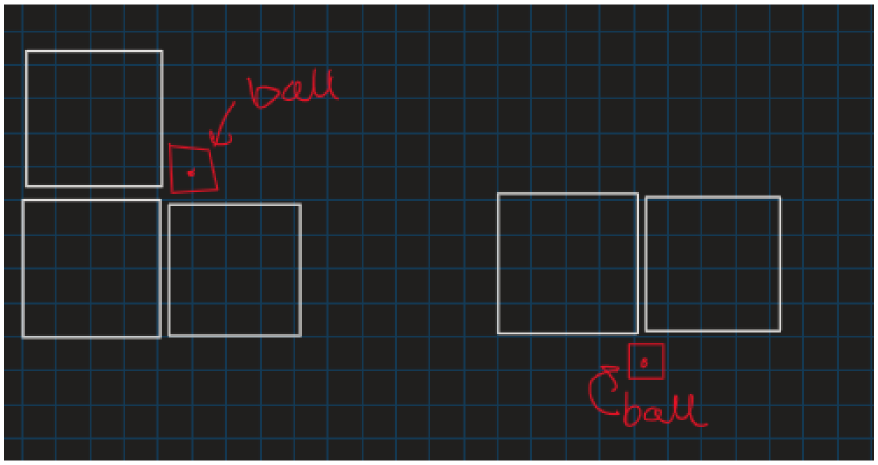
The driver provides us with an outCode which describes which key is pressed, and a makeBreak signal which is 1 or 0 when the user presses a key and releases it. We needed each key press to evaluate to only one action, so we created this state machine to register each press as one 'L' or 'R' signal to the paddles.

One issue we faced while testing this module was that the keyboard suddenly changed from scan code set 2 to scan code set 3. As a fail-safe feature, we used the outCodes from set 2 or set 3 to register a left or right arrow key input. Using this module, we can now use the keyboard as a user-input interface that can be used with most PS2 keyboards that use Scan codes corresponding to set 2 or set 3.

HEXSTATES:

The score of the game was displayed on the HEX0 and HEX1 displays, because it would have been more complex to display the score on the screen, not to mention the lack of screen real estate. One issue one might face when carrying out the task of displaying an incrementable score on the HEX displays is that it would be very slow to map each number to a HEX representation. That would be easy for numbers from 0-9, but gets more unpleasant once there are scores as high as 50, or even 100 and up. As a result, I used a previously implemented interface in 271 to handle an incrementing representation of score. This interface allows for the score to be incremented from 9 to 10, or 19 to 20, and so on. This interface includes two modules called HEXStates0 and HEXStates12. The first module is a state machine that has 10 states starting from 0-9, and it represents the ones place of a number. The second module is a similar state machine with 10 states from 0-9, and it controls then tens, hundreds, thousands, and so on place. Every single HEXStates12 module represents one place within a number. Together, these two modules interact with each other to create the appropriate representation of an incrementing score. Each module answers to one HEX display. In this case, HEXStates0 communicates with HEX0 and HEXStates12 communicates with HEX1. There is only one HEXStates12 instantiated within the score module, because we only need scores in the ten's place, or rather scores up to 50.

Score:



A ball can break up to 2 or 3 bricks at once. In the picture above, you can see that when the ball collides at the border of two or three blocks, they will all “break” due to the way the game is implemented. This occurrence is rare, and it rewards the player well if they manage to achieve this exact scenario.
The score module takes in the variable ‘total_score’, which represents the amount of blocks destroyed at one instance. Each broken block corresponds to one point.

```
always_comb begin
   case(total_score)
      2'b01: begin
               pOne = 1'b1;
               pTwo = 1'b0;
               pThree = 1'b0;
             end
      2'b10: begin
               pOne = 1'b0;
               pTwo = 1'b1;
               pThree = 1'b0;
             end
      2'b11: begin
               pOne = 1'b0;
               pTwo = 1'b0;
               pThree = 1'b1;
             end
      default: begin
                  pOne = 1'b0;
                  pTwo = 1'b0;
                  pThree = 1'b0;
               end
   endcase
end
```

As we can see in the case statement above, signals plus one, plus two, and plus three are assigned based on the value of total_score. These plus signals are then sent to HEXStates0 to tell the score counter to increment by the appropriate amount. I implemented this case statement, and used total_score, because previously, when two or more blocks broke at once it would count as one point, but through this method I can get the total amount of broken blocks at an instance and reward the player for it.

- <u>**Results**</u>

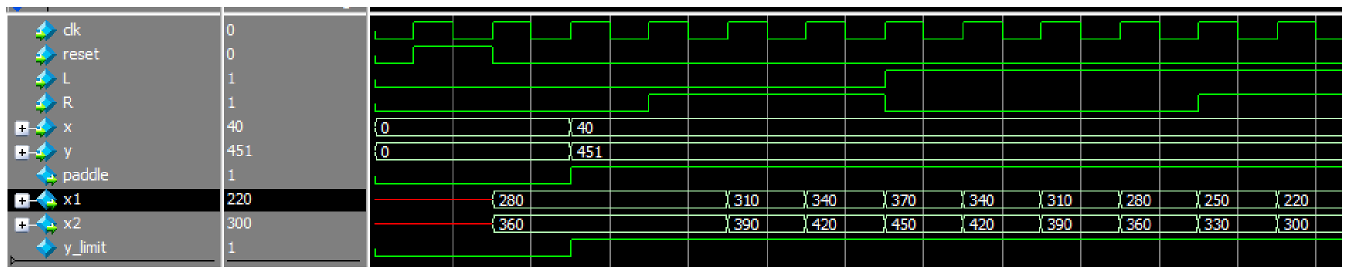
Overview of the Project

- Overall the Breakout Game runs smoothly and you can play the game until you destroy all 50 blocks. The whole project is called from the DE1_SoC and once uploaded to the board you can play the game. The process for creating this game was to first create the static elements of the game. Once that was done we focused on getting the paddle and the ball moving. Then we worked on making the ball bounce off of the border and each of the bricks, causing them to disappear. Lastly we implemented the score system for the game and added in background music.

Breakout:

The breakout module was tested on the monitor, because it was impossible to simulate the VGA frame_buffer, and get any conclusive results without testing the code on the monitor. The instantiation of all the blocks was tested on the monitor. First, we tried to display only one block, then we moved on to two blocks, a row, two rows, and finally five rows.

Paddle_movement:



The paddle movement module was tested on modelsim first for simple functionality, and field tested by testing whether the paddle would not move to the right if it was touching the right border of the screen, and similarly for the left side of the screen. In the waveform screenshot, we can see that the paddle's initial position at start up is x1 = 280, x2 = 360, y1 = 450, and y2 = 460. The y1 and y2 values were implicitly used in the code, as they will never change as the paddle moves left and right. In the waveform, the paddle is given 3 right signals, starting from the front. We can see that the paddle moves 30 pixels with every R input. Likewise, when given 3 L signals back to back, you can see that the paddle moves left 30 pixels with each input.

The case of both R and L being pressed at the same time was very unlikely, because it is unrealistic to expect the user to be able to press both keys within the same clock cycle.

Ball_movement:

The ball movement module was tested thoroughly on the monitor, because it was also impossible to test using modelsim. First we tested whether the ball would show up on the screen without moving. Then we tried to move the ball right infinitely, then we tried to move it right. Then we implemented the ball bouncing off the left and right wall logic, and tested it on the monitor. Luckily, we only had to test this once, because it worked flawlessly on the first try. Likewise happened when testing the balls bouncing off the top screen border.

After simple bouncing dynamics were tested, we then tried to simply bounce the ball off the paddle. This task also worked very seamlessly, because we drew out all of these mathematical comparisons on paper first, and thought about them thoroughly.

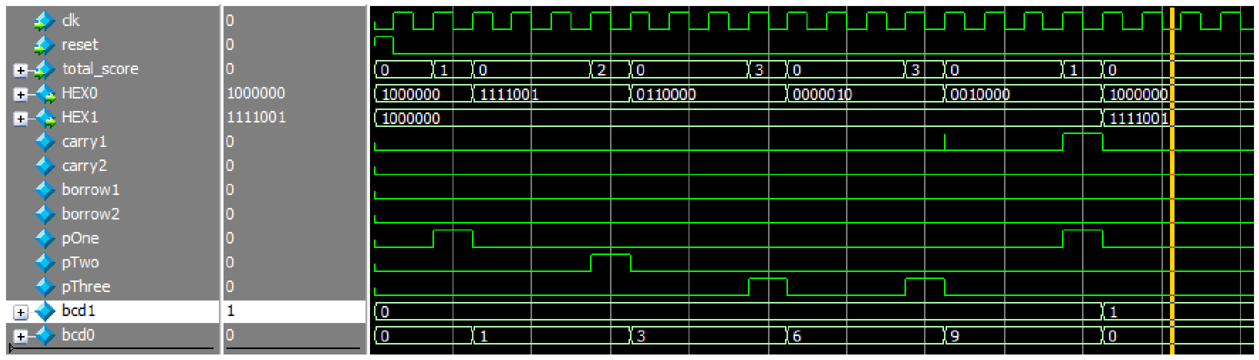
Collision:

The collision module handles the brick_alive variables that decide whether a brick should be displayed on the screen or not, and it also handles the score incrementation when a brick is "broken" by the ball. This module was impossible to test using modelsim, because we had no way of simulating the VGA frame_buffer since it cycled very fast. I simply worked out the mathematical expression necessary to detect a collision of the ball with a brick, and tested it on the monitor. First I implemented the ball just colliding with a brick from the bottom side, then I implemented the top side, and finally I added in collision with the left and right sides. With each addition, there were lots of compilations of the project, so that we could test it on the monitor.

Collision_ball_row_brick:

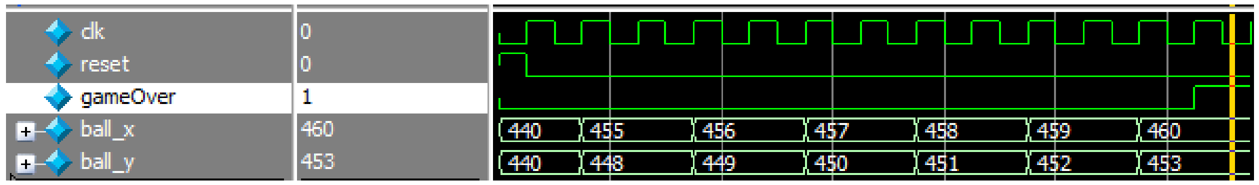
This module was quite difficult to implement, because we had to detect where the ball hit a brick so that we could know how the ball bouncing dynamics would react with such a collision. We could also not test this module using modelsim, because it is impossible to simulate the VGA frame_buffer. Instead, we tested how the ball would bounce just off the bottom side of a brick, then the top, and finally, the sides. We tested each special case on the monitor before moving onto the next.

Score Testbench:



Implementing the score was a relatively forward process, as I have used this style of score-keeping in my final project for EE 271. I used HEXStates0 and HEXStates12 from that class, and the score module, and seg7. These modules were extensively tested for the final project in EE 371, so I did not spend any time testing HEXStates0, and HEXStates12. The only module from this system I tested was the score, because it showed whether the score system worked or not. The score can be read by interpreting bcd1 as the tens place and bcd0 as the ones place. As we can see, when total_score = 1, the score is incremented by one, and when total_score = 3, score is incremented by 3. When the score reaches 9, and total_score = 1 once more, we can see that the bcd1, and bcd0 values are both updated synchronously. Going from 9 to 10, bcd1 changes to 1, and bcd0 changes to 0. We can also see that the pOne, pTwo, and pThree signals work flawlessly, and they go on at appropriate times. Furthermore, this testbench also shows the internals of the HEXStates0 and HEXStates12 modules. When it is ready for the tens place to increment, the carry1 signal of the HEXStates0 module goes high to tell the HEXStates12 module to increment once.

Ball Died Testbench:

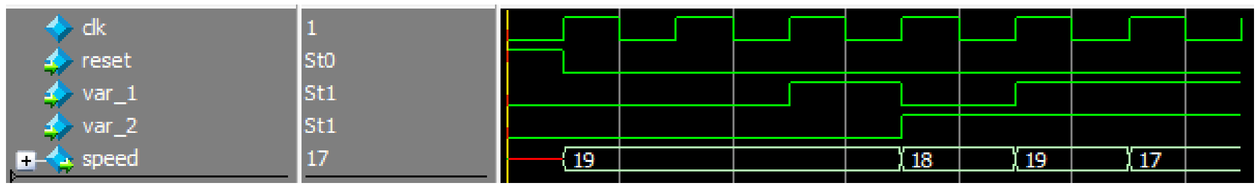


The testing of gameover was very simple. This module checks for the position of the ball using its x and y values, and makes gameOver true when the ball misses the paddle, or said in another way, when it surpasses the y coordinates of the paddle. This results in the game being over which tells the system to display a red screen to the user indicating that they have lost the game. As we can see in the waveform, since the paddles lowest y coordinate is 460, once the ball crosses 460 the gameOver signal comes on meaning that the ball missed the paddle.

HEXStates0 and HEXStates12:

These modules were thoroughly tested last quarter, and did not need further diagnostics due to being used extensively in live programs over many trials.

Difficulty:



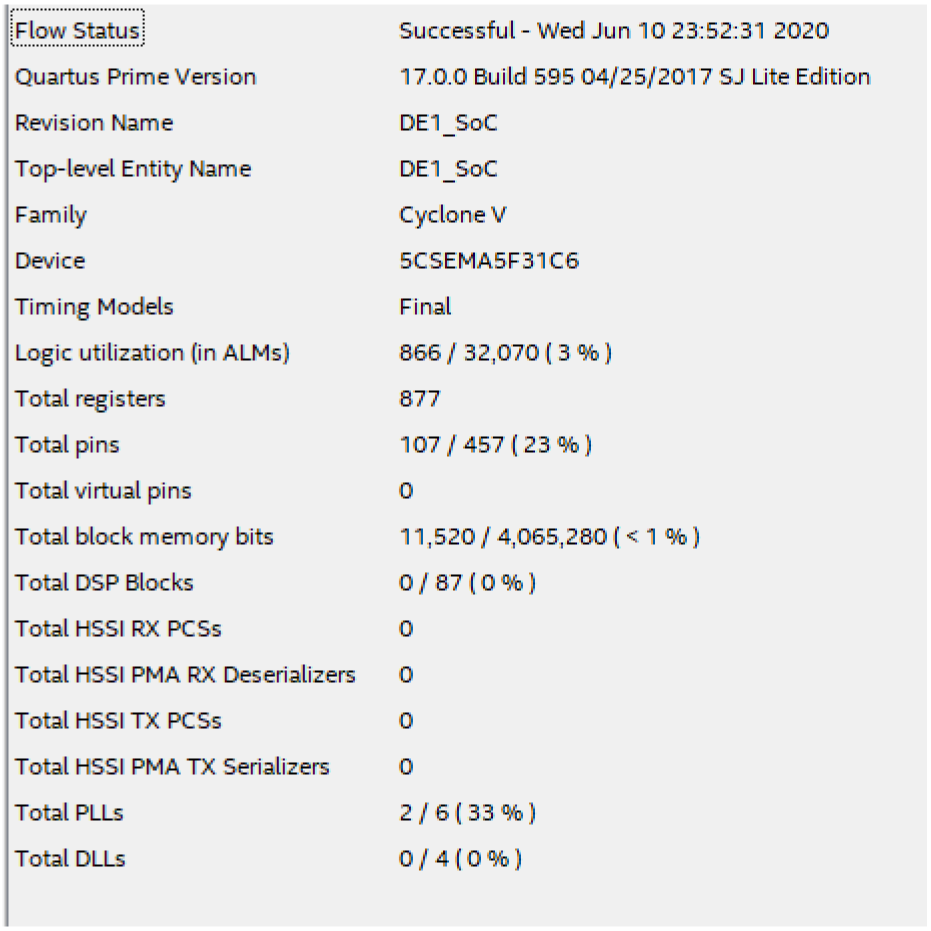
The difficulty module controls the speed of the ball. The speed of the ball is controlled by the user using the var_1 and var_2 variables which are connected to SW[0] and SW[1] respectively. When None of these switches are toggled, the module outputs the default speed of the clock_divider, which is 19. When only SW[0] is toggled, the speed is lowered down to 18, and when both SW[0] and SW[1] are toggled, the speed decreases down to 18. As we can see in the waveform, this exact behavior takes place.

Sound:

The sound drivers and logic was used from lab 5. Since this system was thoroughly tested during lab5, we did not bother to test any of the elements using testbenches. We tested these modules after we adapted them to our project by playing some music, and seeing if they are output to the speaker. This worked on the first try.

Keyboard:

The keyboard drivers were not tested as we didn't know how, and because they were provided by the instructor. The keyboard top level module however was tested using the HEX displays of the DE1_SoC board. We know that each key has a unique make code, and we also know that there is a makeBreak signal. We connected the make code to the HEX displays, and we displayed the HEX values of the key provided by the PS2 keyboard. Through this test we found out which scan code set our keyboard used. Likewise, we connected the makeBreak signal to the LEDR[9], and verified that the makeBreak was working properly as we pushed and released the key. We tested the signals that were output by the keyboard top level module “L” and “R”, by connecting them to the paddle, and seeing if it moved once with a single key press. Once that was verified, we had successfully tested the keyboard top level module.



Flow Status	Successful - Wed Jun 10 23:52:31 2020
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	DE1_SoC
Top-level Entity Name	DE1_SoC
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	866 / 32,070 (3 %)
Total registers	877
Total pins	107 / 457 (23 %)
Total virtual pins	0
Total block memory bits	11,520 / 4,065,280 (< 1 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	2 / 6 (33 %)
Total DLLs	0 / 4 (0 %)

- **Experience Report**

  Issues: The initial issues that we had with this lab was figuring out how to get graphics to appear on the VGA monitor. The driver we were provided with was different than the one we used for linedrawer. We had to look up a FPGA graphics tutorial to understand the logic behind creating a square on the screen. After that it was easy to create all the elements on the screen. The next task that proved difficult was the ball movement and bounce off the bricks upon collision. This took a while to understand because each individual block needed to be seen by the ball at all times, so they understood that its x or y direction needed to change.

  Tips and Tricks: A trick that we knew about before was the generate statement. It functions similar to a for loop and allows you to perform multiple tasks at once. Instead of calling every module individually we discovered that we could use generate statements to call all the logic of the bricks without cluttering our code. We accomplished this by storing the outputs of each module call in an array or 2D array which allowed us to keep track of each individual call's output.

Partner work summary: We put in equal effort into all parts of this lab. Initially we decided to implement the game Tetris, but after research into how to animate squares through the VGA, we figured out that the game breakout made more sense. We decided to change our game and rewrite a new proposal detailing the features of the breakout game we wanted to implement. We meet up through zoom to work on each part of the game together. Additional work that was done by our own was to clean up code or code modules that we were the most familiar with. First we calculated the dimensions of each game element, then made both the paddle and the ball move. Next we worked on all the ways the ball collided with game pieces. Then all the external outputs such as the score and the sound were implemented last.

Feedback: This lab was difficult to complete and took a lot of hours. The reason why we didn't get overwhelmed was because we would work in 3-4 hour intervals each day over the 2 weeks. This lab would have been easier to start if the VGA driver provided to us had a more detailed explanation of how it worked. Instead we had to seek outside sources to understand the basics of how the code worked before we implemented it for our Breakout game. Because the driver was difficult from the linedrawer lab, referring back to the lab wasn't helpful. The other problem that we encountered was the audio, we decided to have background arcade music for our game and unfortunately couldn't figure out how to implement anything further with the information provided to us.

Estimated total time: 50 hours of testing, and development + brainstorming.
Difficulty: Hard

Project video

https://drive.google.com/file/d/1cIJjaEUw8DzCNWlmfwkVHgRa61dTYlJX/view?usp=sharing

Demo Video

https://drive.google.com/file/d/1gL8194jqigkFkuEVFNLQLLOTgaAWnThU/view?usp=sharing