Group Report
ECE 532: Duck Hunt Simulation Game

Group 7
Aranyak Mishra
Goutham Palaniappan
Yi Fan Shao

Contents

List of Figures

1. OVERVIEW

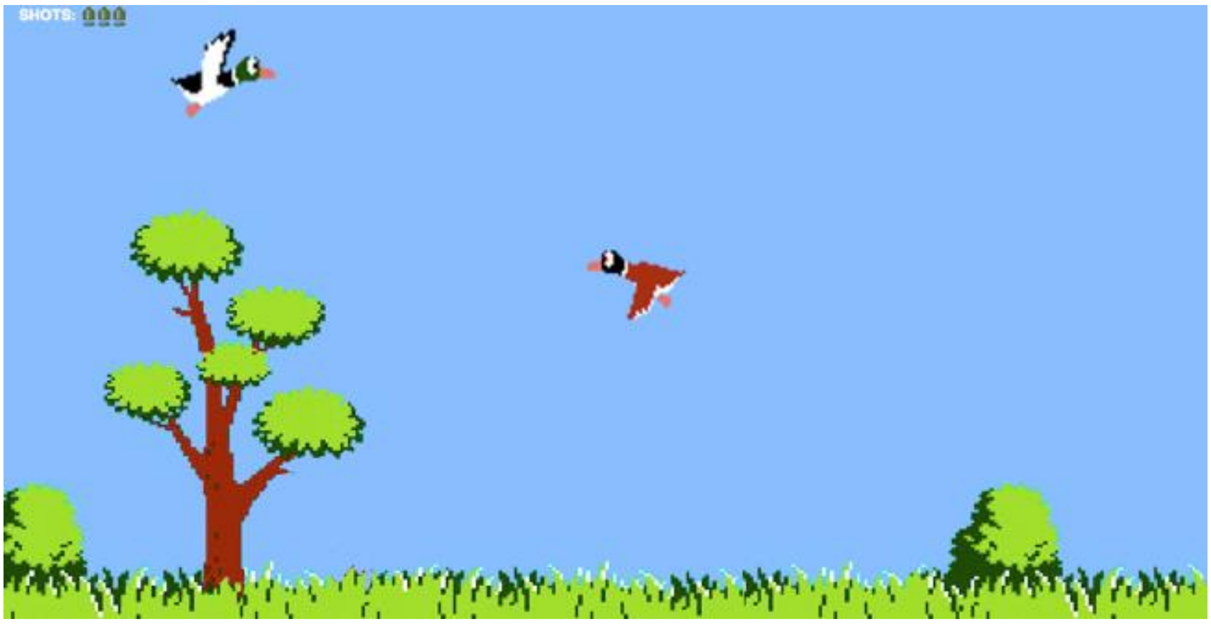


Figure 1: Example of a classical Duck Hunt Game

In this project for ECE 532 Digital Design course, an FPGA based interpretation of the classic NES game Duck Hunt is implemented. Duck Hunt was introduced by Nintendo for its various consoles like chiefly the NES and Arcade. In the classic version, players use a zapper gun to shoot ducks that appear on the screen. It was implemented as a single player game but recently was modified into a 2-player version for the Wii U console, also requiring zapper guns.

In the game play, the player sees a grassland kind of display where ducks rise and try to fly out to the boundary. The ducks change their direction randomly in mid-flight a number of times, depending on the level of difficulty. The ducks also move slower or faster according to the difficulty level. The player wins points on shooting a duck by pointing at it with the zapper gun. And upon reaching a maximum score graduates into the next level.

In the designed product, the need to use a zapper gun has been eliminated. This is a significant benefit as it removes the need for another peripheral just to play the game. With the use of a camera and hardware processing (common resources on most devices), this game can be played with virtually any hand held object. Basically, the player can now use any object to become the gun pointer and be able to `shoot` the moving targets. Additionally, implementing on hardware makes the game much smoother and scalable on different displays. The modular design presented in this project can be adapted to develop a lot of similar light-gun shooter games (emulation of other classic games like Clay shooting and Hogan`s Alley).

Two modifications have hence been implemented in this project that differentiate it from the classical version. The player can use any object as long as it can be calibrated as a stable tracker(the player can see the calibrated marker or gun point on the screen, check its stability before starting the game). Secondly, the gameplay is limited to three levels only and to accommodate this a new scoring strategy has been implemented. Upon missing the ducks(targets moving out of the screen or escaping) the score is decremented. If at any time, the score reaches zero, the player loses the game.

1.1 Motivation and background

The main motivation behind undertaking this idea was to find a simple yet interesting use of the camera module with the FPGA development board. This application uses the camera module and exposes the designers to its functionalities and their limits. It can safely be judged as an image processing-based application running on an FPGA. Besides that, this idea involved working with a lot of other functionalities like memory, video and audio on the development board.

1.2 Goal

The goal of this project is develop a video game that can be controlled with object movement. Precisely, use digital signal processing on video feed to track the movement of gun(hand) along with moving target to hit the object on screen.

1.3 System Block Diagram

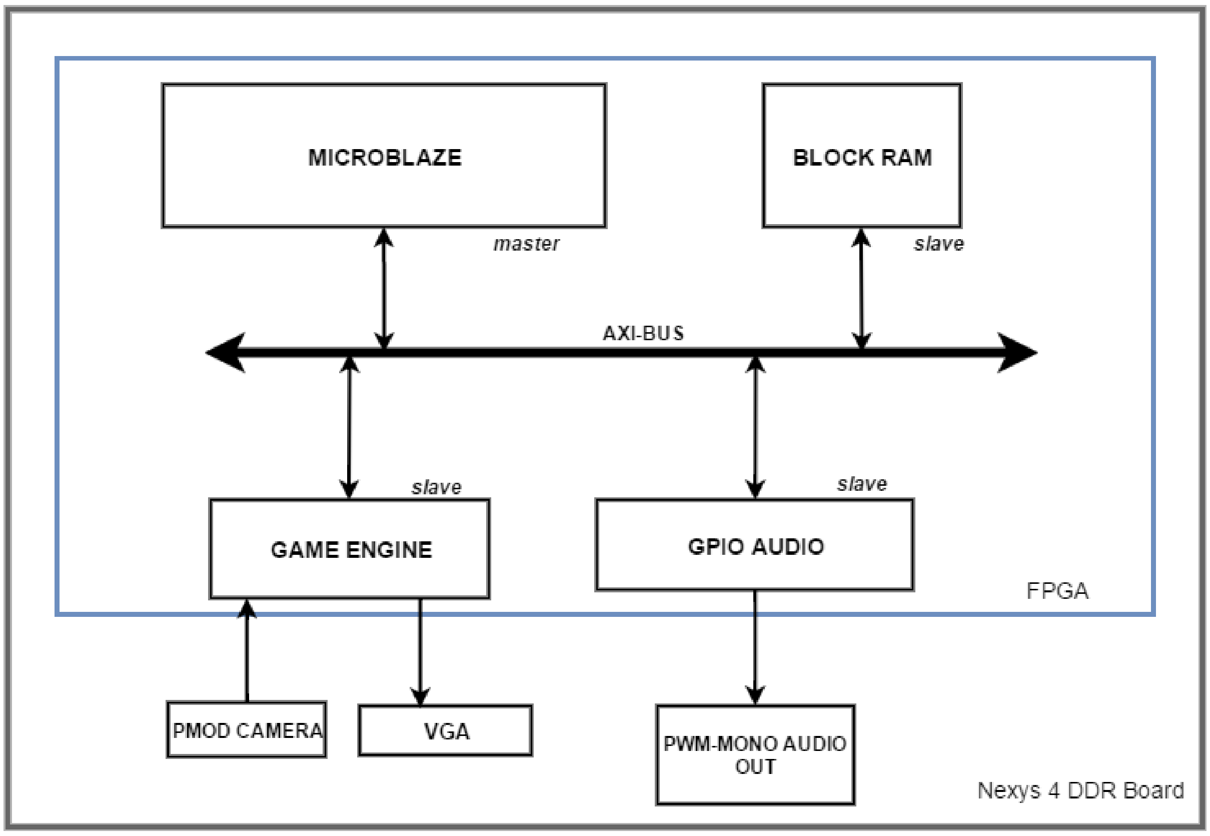


Figure 2: Overall System Block Diagram

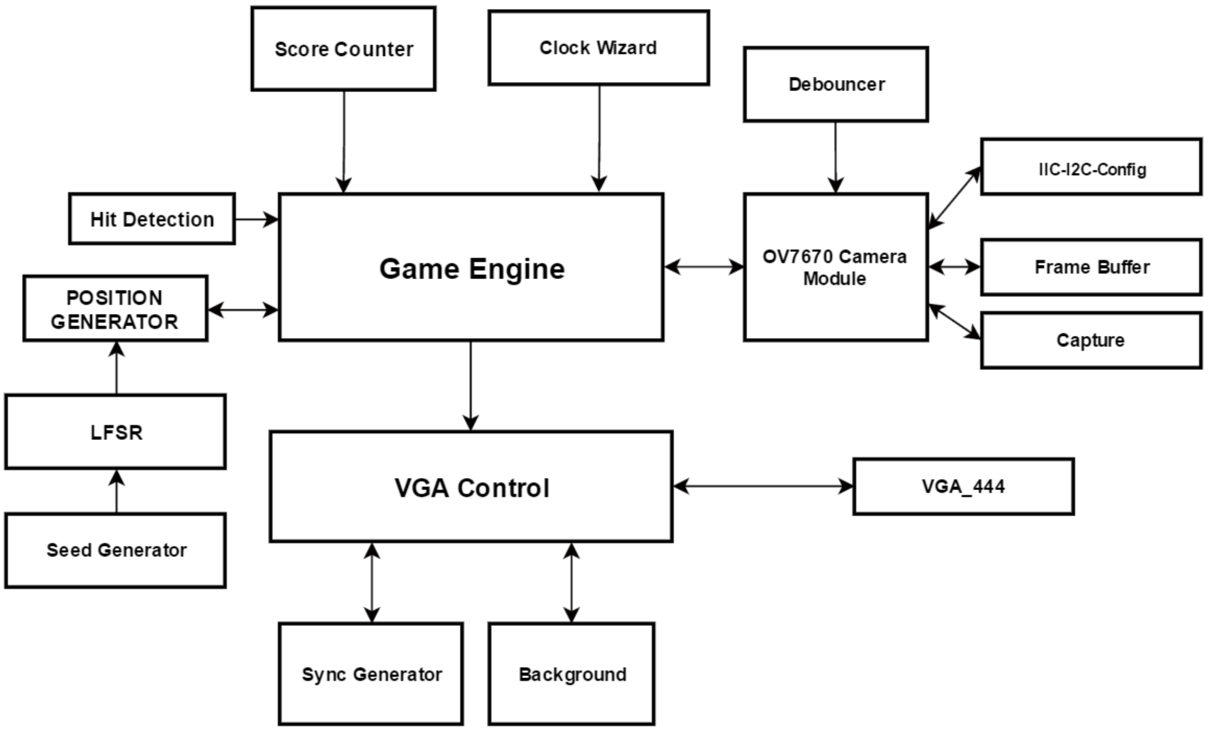


Figure 3: Block Diagram for the Duck Hunt Game Engine

The Game Engine interfaces with the seven segment displays for the score counter, the OV7670 Camera Module for video input, and the VGA controller for displaying the game on a monitor.

1.4 Brief Description of IPs and modules used

Name of IP/module	Created by	Description
ov7670_top	**Custom**	Built upon the provided ov7670_top module to instantiate all the gameplay modules like position generator, capture module and control the communication between these modules.
ov7670_capture	**Custom**	Built upon the provided ov7670_capture module, but customised to handle processing during calibration and tracking
audio_ip	**Custom**	Not integrated into the main project. This is an AXI Lite slave IP which gets 32bit PCM data from the MicroBlaze, which in turn fetches this data from Block RAM. This IP connects directly to PWM_module
pwm_module	**Custom**	Coverts 8 bit PCM values to PWM audio output
GPIO	**Xilinx**	Executes write instructions to the board`s PWM mono out
MIcroBlaze	**Xilinx**	Soft processor with 64Kb local memory for instruction and data. Not integrated.
Block RAM Controller	**Xilinx**	AXI interface to communicate with the local BRAM
Block Mem Generator	**Xilinx**	Frame buffer in the ov7670_top game engine module, used mainly in displaying the camera feed in calibration mode.
UART	**Xilinx**	AXI Lite based UART controller to establish connection between board and SDK
Clock Wizard	**Xilinx**	It takes system clock of 100 MHz to generate clock frequencies of 25 MHz and 50 MHz for camera/capture module and 108 MHz for game engine and VGA module.
Hit_detect	**Custom**	It takes x and y coordinates of target and marker as input and sets output signal hit high when hit is detected
positionGen	**Custom**	Generates the target pseudorandomly
sevenseg	**Custom**	It takes Hit count as input and displays on the 7-

		segment display
vga_ctrl	**Custom**	Generates H-sync and V-sync signals and controls what gets displayed on the monitor
sound	**Custom**	It takes in hit signal as input, plays background tune when hit signal is low and plays beep when hit signal is high

Figure 4: Table with brief description of modules used

2. OUTCOME

2.1 Capture

The capture module(ov7670_capture) implements the calibration and tracking modes for the game. The module reads pixel data directly from the camera feed before writing the feed into the frame buffer (block mem). It processes this feed in two different modes. In calibration mode, it displays the raw camera feed onto the screen with a square `calibration box` where the user can place the object to be used as marker. Upon changing the mode to gameplay mode, the capture module communicates the location of the calibrated object to the main game engine module(ov7670_top).

2.2 Game Engine

We were successful in implementing the initial goal of developing a video game that can be controlled with object movement.
The game engine controls the game-play. It receives coordinates from the capture module, draws the marker and generates the target. It decides when the hit is made and when the miss happens. The game engine updates the seven-segment display score counter upon a hit or a miss and ends the game when the score reaches the limit or drops below 0.

2.3 Audio

The gameplay in the original version of the product is accompanied by a characteristic audio. A similar implementation was attempted. In this project, two separate methods were experimented and one of them was integrated into the final project. The integrated implementation is described below. It is strictly hardware based, where the notes are created with hardcoded values written in verilog modules.

The tune generator created has 3 basic modules. The '**get_fullnote**' module contains the music notes for the tune. It takes an address as input and gives the corresponding note as output. The '**divide-by12**' takes the last 6 bits of the note, stores the least significant 2 bits as remainder and divides the rest to get the octave.

The remainder is corresponds to the exact note in that octave. The main '**music'** module checks this remainder and stores the corresponding count value of the note in the clock divider register.
The counter then begins to run and it at the end of the count limit, the mono audio output pin is set high.
The main music module also has a beep sound generated by a separate counter and is enabled when it gets a hit signal for the hit detection module.
Part of the code is taken from a sample audio project found online and edited separately to add the hit sound along.

Apart from the tune generator described above, a more complex and design-directed implementation is also included in the code base. This involves using the microBlaze soft processor IP to read data from a Block RAM and convert those PCM data into corresponding PWM output stream at a proper sampling frequency. Included in the code base, there are two approaches for this implementation. In both the approaches, the XMD tool in the SDK is used to write the audio data (8 bit PCM values) onto the Block RAM.

In one approach, the MicroBlaze reads 32-bit data from the BRAM in each call and then processes each 8 bit chunk sequentially. A C code runs an algorithm to converts each 8 bit chunk into a PWM stream which is directly written onto the GPIO slave.

In the second approach, a separate slave ip (audio_ip) is used alongside the MicroBlaze and the BRAM. The MicroBlaze sends 32-bit PCM data to the slave audio_ip which uses the PWM module to write PWm data to the board's mono out. The C code in the previous approach is used in the slave ip.

In both these implementations involving the soft processor, the correct tune for the audio data was not recreated. Hence they were not integrated into the final product, but were significant experiments that provide potential for future work.

2.4 Improvements and Future Work

As we have discovered during our time spent working on this project, the camera calibration is not yet robust enough and is easily interfered with background colours. The marker is also not stable enough due to recognition sensitivities. There may be potential improvements on the capture algorithm that can be done to stabilize the marker position which will in turn improve playability of the game.

3. DETAILED MODULE DESCRIPTIONS

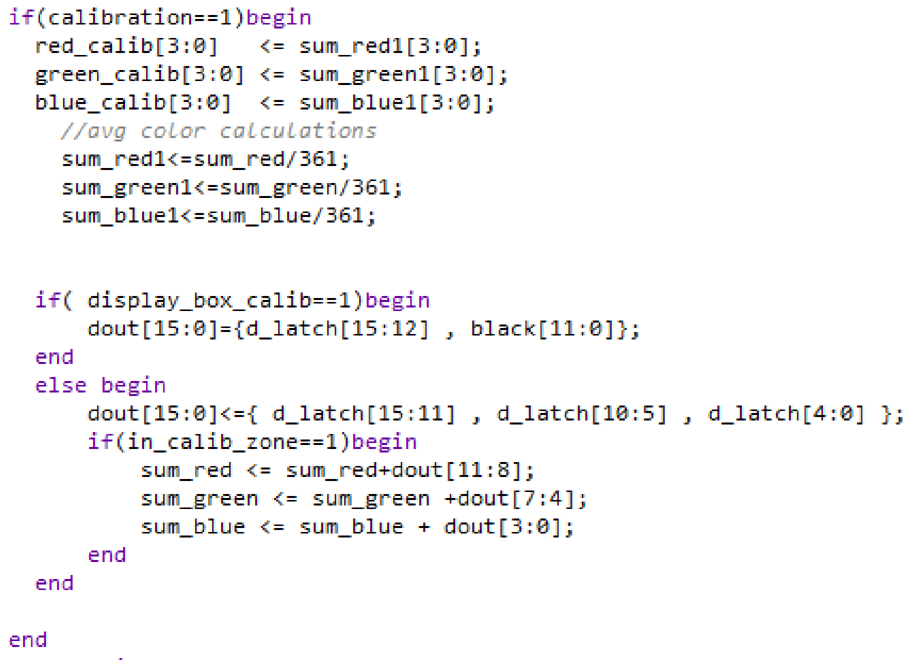
3.1 Main Game Engine

The main game engine is the main module of the project. A state machine consisting of six states determines the procession of the game. Modules such as position generator and hit detection are also instantiated inside this module. The state machine has seven states: Initial, Fetch, Fetch_Wait, Move, Hit, Miss, and Game_end. Depending on the status of the start button input and the status of the game score. Reset initializes the game state back to Initial and the game reaches Game_end once the score exceeds 15 or falls below 0.

3.2 Capture Module

The capture module is built on the base pmod camera module provided in the course. The provided capture module writes camera feed into a block RAM buffer. The processing of the feed happens before the data is written into the buffer. This module is customised to process the input stream of pixel data in two different modes. These modes are calibration and gameplay. The mode itself is an input to this module from the main game engine.

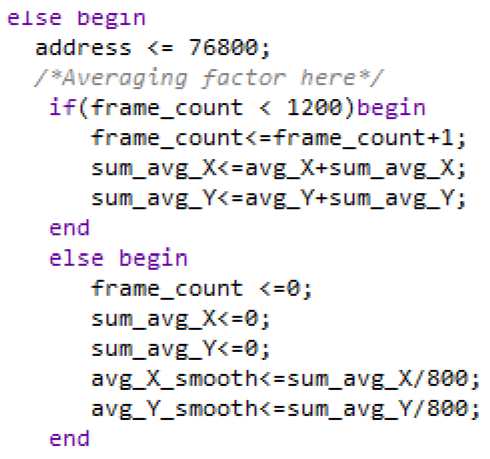
In **calibration mode**, the raw camera feed is written into the frame buffer with a square drawn around the center. These frames (320x280 pixels) are put through the vga444 module instantiated in the game engine to the display. Within the stream of each frame, the pixels within the square are processed to calculate the average RGB values within the square. It is intended for the user to put the desired object (tracker) in front of the camera such that the object fills up most of the area within the displayed square. The module continues calculating the calibration data in each frame until the user changes the mode to gameplay through the board button.



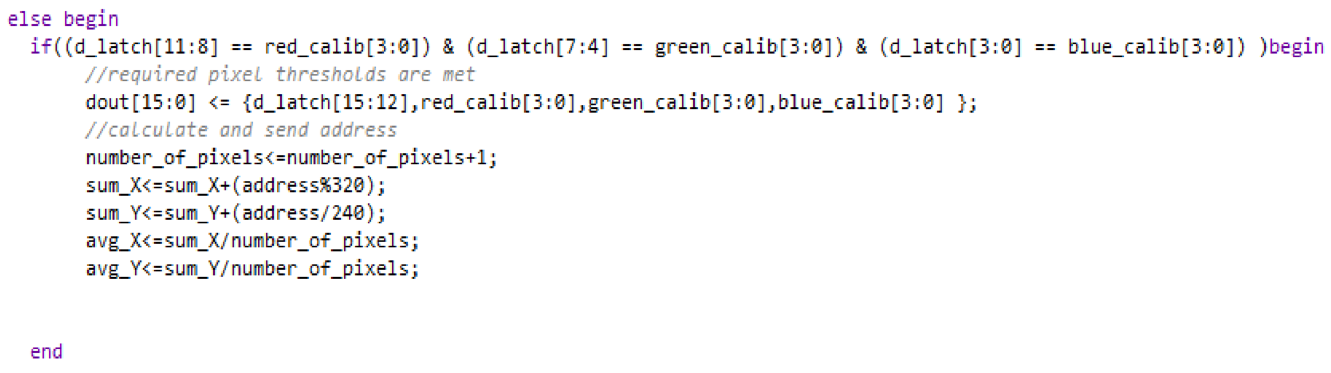
```
if(calibration==1)begin
  red_calib[3:0]   <= sum_red1[3:0];
  green_calib[3:0] <= sum_green1[3:0];
  blue_calib[3:0]  <= sum_blue1[3:0];
    //avg color calculations
    sum_red1<=sum_red/361;
    sum_green1<=sum_green/361;
    sum_blue1<=sum_blue/361;


  if( display_box_calib==1)begin
      dout[15:0]={d_latch[15:12] , black[11:0]};
  end
  else begin
      dout[15:0]<={ d_latch[15:11] , d_latch[10:5] , d_latch[4:0] };
      if(in_calib_zone==1)begin
          sum_red <= sum_red+dout[11:8];
          sum_green <= sum_green +dout[7:4];
          sum_blue <= sum_blue + dout[3:0];
      end
  end

end
```

Figure 5: Code Snapshot of the Calibration Logic in OV7670_capture

In **gameplay or tracking mode**, this module finds the average X and Y location of all pixels that have the calibrated RGB values. It follows the same condition as calibration mode, but instead of finding the average RGB values of all the pixels in a given frame whose location falls within the square, it computes the average location (or address) of all the pixels with the calibrated RGB values. After computing the average X, Y location in each frame, the tracker location is updated with a running average over 1000 (parameterizable) frames. This is the basic smoothening of the tracker's X, Y location. The data in avg_X_smooth and avg_Y_smooth is then channelled to the main game engine, where it is scaled to relative location for a 1280x1024 pixels display.



```
else begin
  address <= 76800;
  /*Averaging factor here*/
   if(frame_count < 1200)begin
      frame_count<=frame_count+1;
      sum_avg_X<=avg_X+sum_avg_X;
      sum_avg_Y<=avg_Y+sum_avg_Y;
   end
   else begin
      frame_count <=0;
      sum_avg_X<=0;
      sum_avg_Y<=0;
      avg_X_smooth<=sum_avg_X/800;
      avg_Y_smooth<=sum_avg_Y/800;
   end
```



```
else begin
  if((d_latch[11:8] == red_calib[3:0]) & (d_latch[7:4] == green_calib[3:0]) & (d_latch[3:0] == blue_calib[3:0]) )begin
      //required pixel thresholds are met
      dout[15:0] <= {d_latch[15:12],red_calib[3:0],green_calib[3:0],blue_calib[3:0] };
      //calculate and send address
      number_of_pixels<=number_of_pixels+1;
      sum_X<=sum_X+(address%320);
      sum_Y<=sum_Y+(address/240);
      avg_X<=sum_X/number_of_pixels;
      avg_Y<=sum_Y/number_of_pixels;


  end
```

Figure 6: Code for Smoothening logic

In both the calibration and tracking mode, the addresses are essentially used to calculate the X and Y location of a given pixel. As the frame in the feed is 320x280, each frame consists of 76800 pixels which are addressed starting from 0. By using proper operation on a given address, X and Y location for any given pixel can be found. In the module, the calculated X and Y location are relative to 320x280 pixels feed, and hence they need to be scaled up for 1280x1024 display.

3.3 Position Generator

The position generator is responsible for generating targets. We have chosen to implement a Fibonacci Linear Feedback Shift Register (LFSR) to generate pseudo-random numbers. Due to the limits of LFSR's pseudo-randomness, we have also instantiated a seed generator to initialize the LFSR. The seed generator is a counter with the same bit width as the LFSR. The counter is incremented every clock cycle and the seed is assigned to the LFSR when the user presses a button on the FPGA board. Because the counter is counting at a very fast pace, the seed that is assigned to the LFSR will likely be different every time the user presses the button.

3.4 Hit Detection:

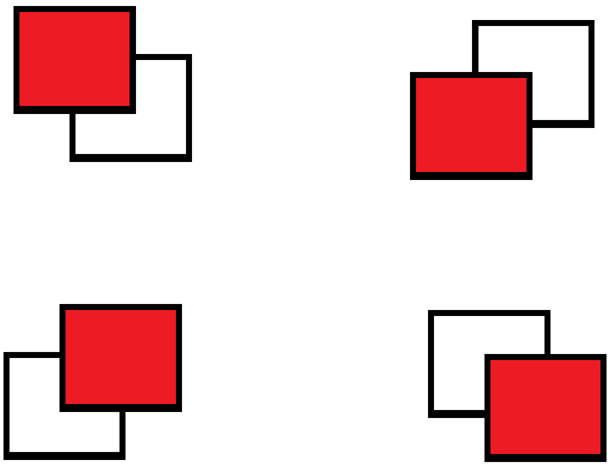
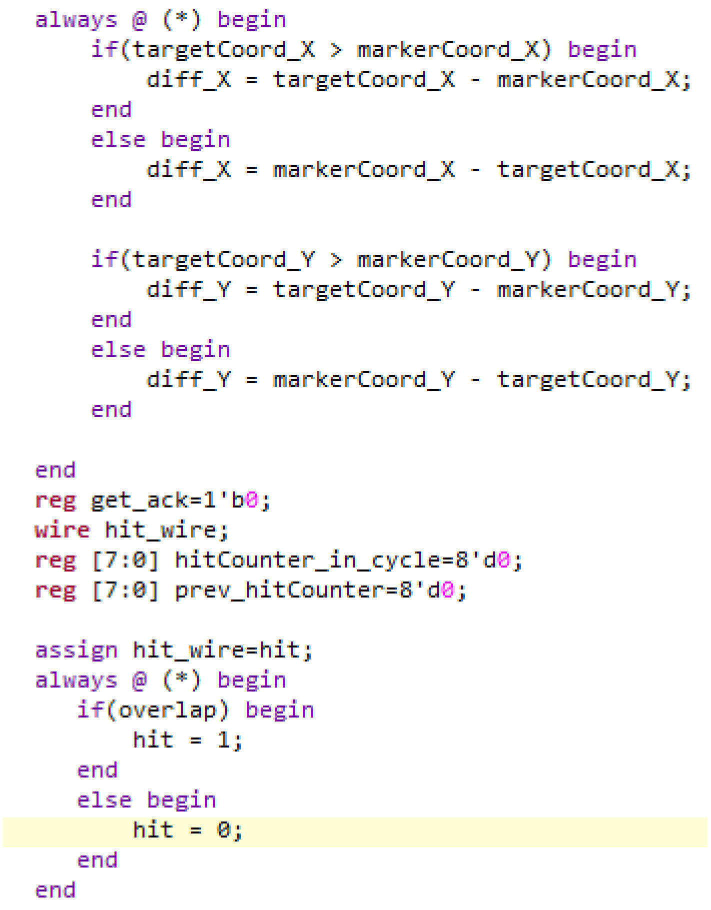


Figure 7: Overlaps detected for HIT

The hit detection module takes the x and y pixel coordinates of both target and marker as inputs along with clock and reset signals, and gives a hit signal output depending on the four basic overlaps as shown in figure. Two difference signals DIFF_X and DIFF_Y are calculated by subtracting the starting x and y coordinates of the target and the marker. This value is compared with a threshold set at 50 pixel length and a hit signal is set high when DIFF_ X and DIFF_Y fall within the threshold.



```
always @ (*) begin
    if(targetCoord_X > markerCoord_X) begin
        diff_X = targetCoord_X - markerCoord_X;
    end
    else begin
        diff_X = markerCoord_X - targetCoord_X;
    end

    if(targetCoord_Y > markerCoord_Y) begin
        diff_Y = targetCoord_Y - markerCoord_Y;
    end
    else begin
        diff_Y = markerCoord_Y - targetCoord_Y;
    end

end
reg get_ack=1'b0;
wire hit_wire;
reg [7:0] hitCounter_in_cycle=8'd0;
reg [7:0] prev_hitCounter=8'd0;

assign hit_wire=hit;
always @ (*) begin
   if(overlap) begin
       hit = 1;
   end
   else begin
       hit = 0;
   end
end
```

Figure 8: Code for hit detection

3.5 Score Logic

The score for the game is displayed on the 7 segment displays. This module receives the hit-count as input along with clock and reset, and sends corresponding output signal to the anode and cathode of the 7-segment display pins.
The hit count is first sent to BCD converter (separate module) which converts the binary value to a decimal value. The tens and the ones part of the count is sent back to score module.
The 100 MHz system clock is divided to the correct refresh rate of the displays and the anodes are continuously switched at this frequency with cathodes being ones or tens depending on the anode enabled.
The cathode value for each number is specified in a switch based combinational logic with default value being set to 0.

4. GAMEPLAY

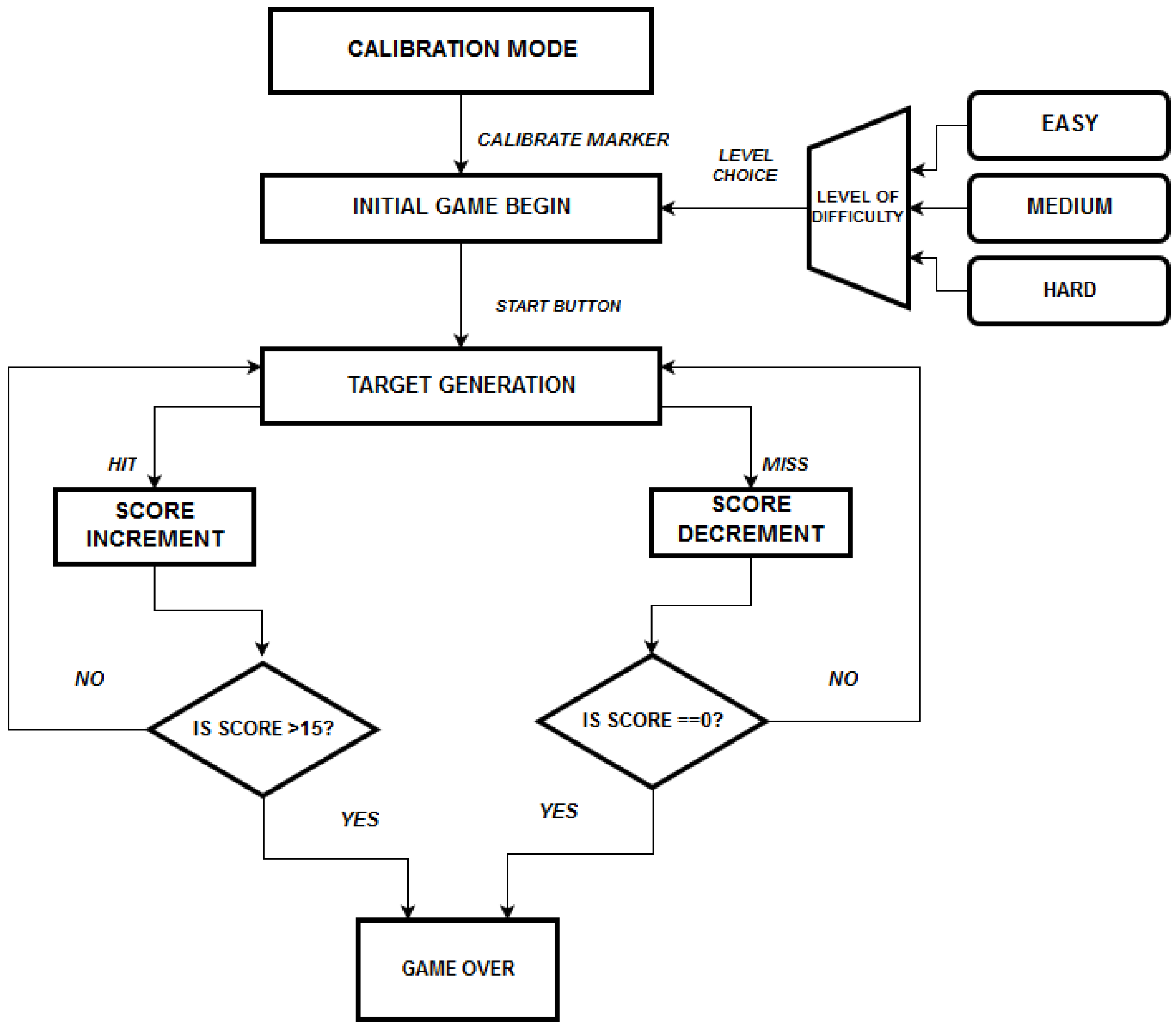


FIGURE 9: **The flowchart of the Game-Play.**

Game modes:

Calibration:

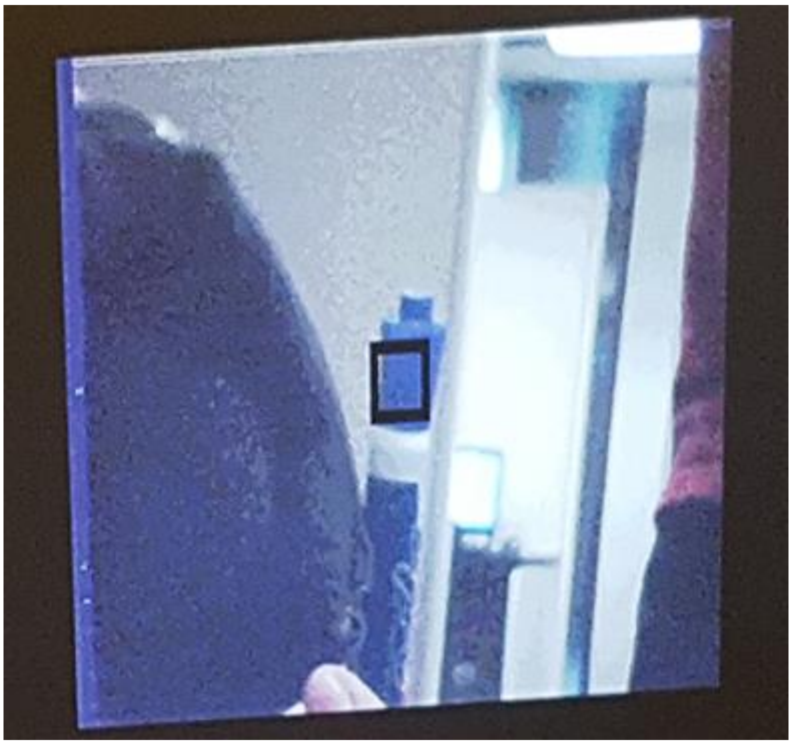


FIGURE 10: Calibration of a marker cap.

The game begins with calibration stage. During this stage, the player is asked to place any object that he wishes to play with (or even his hand) within a square box indicated in the camera feed and the calibration button is pressed to capture the average pixel value of the image within the box.

Game begin:

The player is asked to choose level of difficulty- easy, medium, hard. The corresponding switch is set high and depending on the level choice the speed of the target movement is modified in the code.

Target Generation and play mode:

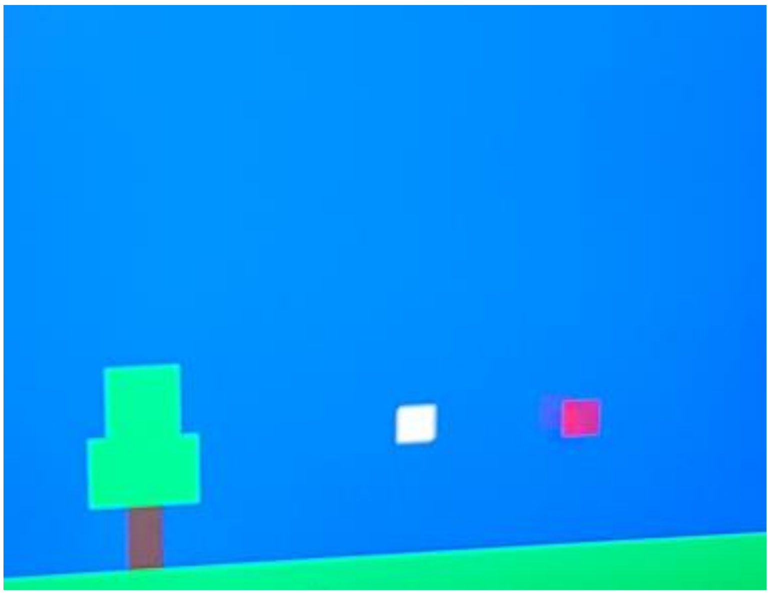


Figure 11: Marker in red and target in white

The target is then generated from the bottom of screen with random starting points and moves upward.
The path of the target can also change at anytime because the seed for the next move is randomized.
The target can go out of bound to the left, right or the top of the screen. Once, the target goes out of bound, the score is decremented as it is counted a miss. If the player manages to hit the target before it goes out of bound, the score is incremented.
If the score reaches 15, then the game ends and the player has successfully completed the corresponding level. He can then choose to play the next level of difficulty.
If the player misses frequently and his score reaches 0 because of the decrement for each miss, the game ends and the player must restart at the same level.

5. Appendix:

Milestones

Week 1	• Implement the audio module and play basic sounds • Develop algorithm to process static images for object recognition
Week 2	• Basic audio and beep generation • Camera feed displayed on the monitor • Basic animation
Week 3	• Frame pixel processing in input stream • Integrated camera feed an animation • Tune generation
Week 4	• Audio successfully written to DDR through tcl script • Calibration to detect and track any distinctly colored object
Week 5	• Slave IP experiments for audio from DDR • Random target generation
Week 6	• Calibration module integrated • Hit detection and hard coded background • Score and 7 segment display
Week 7	• Game engine implemented with calibration, tracking, smoothing • Target generation improved • Audio experiments with master AXI and MicroBlaze-GPIO
Week 8	• Demo

Figure 12: Table with milestones achieved