ECE 385

Spring 2015

Final Project

FlappyCopter in SystemVerilog

Name: Alvin Wu, Kevin Ly
Section: ABM (11:30am-2:20pm)
TA's: Bilal Gabula, Shuo Li

May 7, 2015

Table of Contents

List of Figures

1 Introduction

This is the documentation and design philosophy of our final project called: *FlappyCopter* using an Altera DE2 FPGA, a keyboard, and a VGA monitor. A fusion between the classic games Helicopter and Flappybird. FlappyCopter involves the player taking control of the helicopter and avoiding the obstacles with Flappybird-esque scenery in the background. The reason we chose this project was we wanted to do a project that we were able to complete within the timeframe and a project that allowed us to add features to make it our own.

2 FlappyCopter Setup Description and Instructions

FlappyCopter, the game, starts off with a start screen with instructions on how to play, the title of the game as well as who created it. Once the player presses the "w" key, the game starts and the helicopter will start to ascend. When the "w" key is not pressed, the helicopter naturally descends down the screen due to gravity. The goal of the player is to avoid as many obstacles as possible with the helicopter without crashing into the obstacles or flying out of bounds. There are two kinds of obstacles. Double obstacles which are a variant of the Flappybird pipes where the player must fly through the two blocks or find some other way to avoid them. The second type of obstacles is the single block where the player simply needs to maneuver around it. Once a player crashes into a block or flies out of bounds, the game will end and a gameover screen will appear. In order to play again, the player will hit the reset button on the FPGA.

3 Description of Circuit and Modules

This circuit is composed of several files called modules. Through ports, we are able to allow individual modules to communicate with its environment through inputs and outputs. The description of each individual module that was used as well as its purpose can be found below:

3.1 lab8_usb (Top Level) Module

The lab8_usb module is our top level entity. Essentially what it does is it defines the components like the Ball, Cloud and gamestate modules as well as the behaviors between these modules. In this module ports are declared to be the input and outputs of each individual module, allowing communication between that modules and other modules.

3.2 Ball (Helicopter) Module

The ball module is the module for our main player character. In our game, the ball is the helicopter which the player controls. The inputs of this module

include the reset, frame_clk, rdy, and key. The outputs include BallX, BallY, BallS, BallW, BallH, and a gameover signal. The internal logic signals include Ball_X_Pos, Ball_X_Motion, Ball_Y_Pos, Ball_Y_Motion, Ball_Size, Ball_W, Ball_H, and gg. We begin this module by setting some key parameters such as the center X and Y to 100 and 240 respectively, the start point of the helicopter. We then set the parameters for the whole game screen, the min and max X and Y of the screen, which is 480x640. Finally, we have two parameters called Ball_Y_Step and Ball_Y_Jump. Step is essentially our "gravity" constant, so how fast it will fall. Jump is our flying parameter which determines how fast the helicopter will ascend. The choices of 8 and -6 respectively were found through testing the game multiple times and tuning the value to what felt right difficulty wise. The width and height of our helicopter is finally set to 86 by 26 which was the size of the rectangle that encapsulated the helicopter.

The always_ff block deals with the reset state and start screen state. If reset is pressed, the helicopter will be reset to its original position. Until the start button triggers the game, it will be stuck in the !rdy state where the helicopter does not move. Otherwise, the game will run where the helicopter can move up and down depending on whether the key ẅïs pressed or not. This was hex "1a". Hex "cc" was used for testing purposes. Upon collision, the gameover flag will be set which tells the helicopter and game to freeze. Otherwise Ball_X_Pos and Ball_Y_Pos will constantly be updated. Finally, the internal variables are set to the outputs which go up to the top level design.

Once we completed this, we focused on implementing the 128 bit software encryption program in C, which can be seen in figure 1. In order to do this, we needed to implement a main, KeyExpansion, RotWord, SubWord, SubBytes, AddRoundKey, ShiftRows and MixColumns function. Inside our main function, we first started by asking for the user to input 32 characters, which we later converted to a 16 bit array. Similarly, we also did this for the cipher key input. Once that was done, we called key expansion, which was needed before we could start the AES algorithm. Once this was done, we followed the AES algorithm shown below of looping through 9 rounds of: SubBytes, ShiftRows, MixColumns and AddRoundKey. Once these 9 rounds were finished, we did one more round of just SubBytes, ShiftRows and AddRoundKey. Once this final round of encryption was done, we stored the 16bit array back into a 32 bit array, which we printed to the screen.

3.3 Cloud Module

The cloud module takes in reset and frame clk. In this module, all reset does is it would reset the position of the clouds back to its default starting position. If reset is not pressed then we randomly generate the y position of the cloud (default x being 840) depending on whether or not it has reached the end of the screen (X Position is less than 1) by getting the random value from the random number generator and multiplying it by two. Since we dont want the clouds to

appear too low, we checked if the randomly generated value was pass a certain value that we would deem too low on the y axis of the screen. If it was too low, then we just used the randomly generated value instead of offsetting it by a factor of two. Otherwise we would just have the cloud continue moving across the screen. The size of the clouds were chosen to be 80x40, which is the size of the 2D array created to map the pixel colors in the sprite table and it was a size that we felt was perfect for a cloud.

No matter what the state is in, we chose to make the clouds move across the screen since we felt it would give a better overall appearance to the game if clouds moved across the screen with a speed of 6 even when the user was not playing. Since we wanted to have two clouds, we had the module output the X, Y,Width and Height of two clouds.

3.4 Hot Air Balloon Module

The Hot Air Balloon Module is very similar to the cloud module. This module takes in as input reset, frame_clk, rdy, collision and a rand_out value. Reset is important because when the reset key is pressed, the hot air balloon is set to it's original starting position and it stays at that position until the player plays the game. The size of the module was determined to be 58x63, which is the size of the 2D array that was created to hold the pixel colors.

Similar to the cloud module, once the hot air balloon reaches the end of the screen, this module checks whether or not the random value multiplied by two is greater than 300 (which we deem to be too low on the screen). If it is then we just use the random value as the new y position with the new x position being 840. If it isn't then we use the random value multiplied by two as the new Y axis value. The reason we multiply the random value by two is to get a wider range of values that we could use. If the hot air balloon hasn't reached the end of the screen, then the sprite keeps moving towards the left of the screen with a x_step of -4, which is slower than the obstacle speed. This module outputs X, Y, Width and Height of the hot air balloon, which is needed by the color mapper to draw the sprite.

3.5 Obstacle Modules

3.5.1 Obstacle Behavior and Design

The design philosophy behind our obstacles in general was to have 3 obstacle modules which would be recycled in a queue like matter. There would be 1 double block obstacle and 2 single block obstacles. This would give the illusion of many different obstacles constantly flying across the screen because of our recycling of these 3 blocks as well as the random number generation. The blocks were rectangular in nature because they were suppose to simulate wall like objects in which the helicopter might crash into. The speed as well as the

size of the obstacles was fine tuned through testing to provide a good level of difficulty. The queue like implementation of our blocks was implemented by setting the obstacles a little further than the maximum size of the screen after it reach the far left hand side of the screen.

3.5.2 Obstacle Module

The obstacle module represents a double block obstacle (as opposed to a single block obstacle) in our game. Its inputs include reset, frame_clk, rdy, key, rand_out, ballx, bally, and balls. It outputs ObsX, ObsY, ObsW, ObsH, ObsX_bottom, ObsY_bottom, ObsW_bottom, ObsH_bottom, and gameover.

The internal logic variables include registers for the x,y, motion, width, and height information for both a top and bottom block. A register called Pipe_gap and gap also exist to determine the gap between the two obstacles. Finally, a variable called gg determines whether or not the game is over or not. The parameters for the obstacles include the X and Y min and max of the screen as well as Obs_X_Step which represents the speed of the block set to -7. The size of the block was determined to be 15x60 and the initial pipe_gap is 280 pixels apart.

The always_ff block begins with the reset case where the blocks get set to an initialized position on the right side of the screen. If the game has not started yet (player has not pressed the start button) the obstacles will not move. Once the key is pressed however, the obstacle will move from right to left across the screen. To make things more difficult, we implemented the random number generator. Therefore, when the left most side of the block reaches the left side of the screen, we recycled the block to the block queue at the right side of the screen with a randomly generated a new gap value so the two obstacles would always be different when it comes again. This was achieved through our instantiation of the random number generator on the top level and passing in rand_out as our random value (the seed is set in the reset state).

Otherwise, this module also takes in the ball x, y and size for collision detection. Essentially how collision detection works is the it checks to see if the x and y position of the ball plus the size is within the boundaries of the obstacle. If yes, the gg flag is set to tell the game to freeze. Our collision detection method will be explained more in a later section.

3.5.3 Obstype1 and Obstype2 Modules

The obstype1 module(and obstype2) is very similar to our two block obstacle, however, this module only has one block sliding across. The different obstacle types increase the difficulty of the game because the player has different obstacles to maneuver around. The main differences is that this module does not hold information about the bottom block. Also, when the single block reaches the

left most side of the screen, it is recycled to the block queue at the right side of the screen, but rather than a random gap being generated, a random y position within the bounds is generated. To do this, we set Obs_Y_Pos to 2*rand_out plus an offset of 100.

3.6 Gamestate Module

The gamestate module is our state machine as shown in Figure 1 for the different states of the game. The inputs of this module are reset, frame_clk, gameover, gameover1, gameover2, gameoverball, key. The outputs include rdy, lost. This state machine has 3 main states: the start state, playing state, and the gameover state.

The state machine uses a 3 always state machine and starts with the always_ff block which tells the state machine to go to the next_state unless reset is pressed where it is initialized to the start state. The player is stuck in the start state until they press "w" (hex xCC was used for testing purposes on different keyboard). Once this button is pressed, the player moves onto the playing state. This sends the rdy signal to all modules telling them to start moving. The player is in this state until one of the gameover flags is set high. If any of the flags are triggered, the game moves into the game over state which freezes the game until the user manually resets the game to play again.

Figure 1: Gamestate State Machine: Key W press to move to Playing, Gameover Flag on to move to Gameover and Reset to move back to Start State

3.7 SpriteTable Module

This modules takes in a clock as a input and it outputs all the sprites that were used in the game, which are: Helicopter, Clouds, Balloon, Blimp, FlappyCopter (title screen), StartScreen (text that shows created by and instructions, and GameOver text. This module stores all the sprites that we used in a 2D array. Each individual sprite would store Height x Width pixels, where each pixel contains a binary number that represents a specific color that is mapped by the ColorMapper. For example, we would represent a helicopter with a 26x86 array of numbers between 0-5. 0 would represent the background color, 1 would represent the color black and so on. Every clock cycle, one of these pixels are chosen to be drawn by the ColorMapper module. The sprite table and sprite for the helicopter can be seen below in Figures 2 and 3.

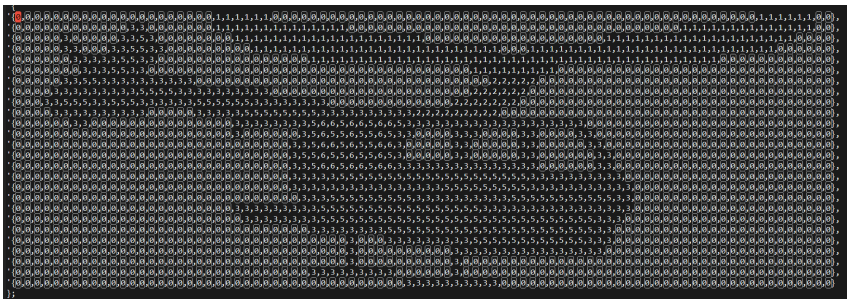

Figure 2: Helicopter SpriteTable: 0 represents background color (skyblue), 1 and 3 represent black and so on, 2 dark grey, 5 is red and 6 is white

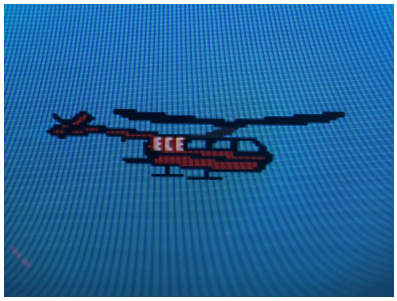

Figure 3: Helicopter Sprite

3.8 Color Mapper Module

This module performs object or shape rendering and coloring. It takes in as input, the x and y position and the 2D array of all of our sprites and it outputs RGB signals to the monitor. Inside this module there are two essential processes for each sprite. The first one determines if the pixel that is currently being drawn is greater than the leftmost position of the sprite and less than the rightmost position of the sprite. It also checks if the pixel that is currently being drawn on the y axis is greater than the position of the topmost position of the sprite and less than the bottommost position of the sprite. If it doesn?t meet these conditions then we would set the sprite to 0 so it doesn?t draw it in the RGB_Display. If it does meet these conditions, we would pass in the specific color value that corresponds to the pixel. We determine what the parameters are into the double array by passing in DrawY-(topmost y position of sprite) into the first array and DrawX-(leftmost x position of sprite) into the second array. So to access the helicopter pixel for example we would do:

helicopter[DrawX - BallY][DrawX - BallX]

The second essential process in this module is the RGB_Display process which checks the 2D array from the SpriteTable and it assigns it a specific RGB value based on what the value at that pixel is. For example, when the value at that pixel is 1 for the helicopter, we set the RGB values to Hex 0, which represents the color black. We chose to map each number by different colors for different sprites because it was easier than creating a different value for each color that we wanted to use. So 1 for the hot air balloon represents orange, not black.

For the start screen, game playing screen and gameover screen we have different sprites that are shown. For example, in the start screen we have the title of the game, the game creators, instructions, helicopter and clouds. In order to only get these sprites to only show in the start screen, we pass in a rdy variable, which is only on when the key ?w? is pressed. For the sprites that don?t show up on the start screen like the obstacles or balloon we check to make sure that it only displays when rdy is on, which is checked in the first process of the module. When the player is playing we want the text to disappear so we also use the rdy variable and the text is only displayed when either rdy is off or gameover is on or off depending on which text we want to show.

3.9 VGA Controller Module

This module was given to us in lab 8 and it is responsible for synchronizing the VGA monitor with the FPGA clock. Essentially it updates the current pixel to be drawn, running through all 640x480 pixels in separate clock cycles. This process can be seen in the figure 5 below.

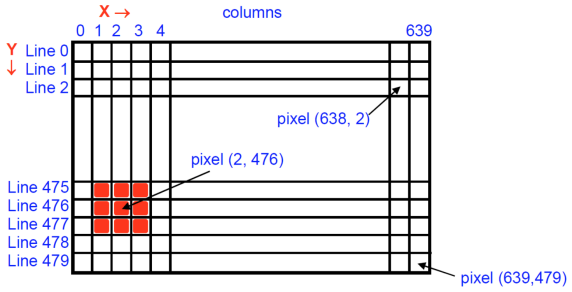

Figure 4: VGA Controller

3.10 Randtop Module

The randtop module is the top level module for the random generator (randgen being the code for the actual random number generation). This module is instantiated in the top level of our game and its inputs are a clock, clk enable, load enable, seed, and rand. The clock is the main clock input. Clk enable we always

set to 1 so the random number generator could run. For the seed, we picked the number 100 at random. At the restart state we toggled load enable high so the seed could be loaded into the module. After this, the random number generator generates random numbers by a shift registering left and appending the feedback bit which is computed through a series of XOR operations. This random number output is fed through to the obstacle modules/background clouds when they hit the end of the screen so their position is randomized when it appears again.

4 Circuit Operation

4.1 Collision Detection

Collision detection was implemented in our game through a rectangular box method. In this method, the helicopter was represented by a rectangular box that was considered its hit box. If this rectangle overlapped with any of the obstacles, the game over flag was set high. Most of the collision detection was implemented in the obstacles themselves. The balls x, y, and size were passed into all the obstacles so we could check in the obstacle modules themselves whether or not the rectangular hit box of the ball touched the obstacles. However, in ball.sv itself, we implemented ceiling and ground collision detection as well by checking the helicopter y position. If this y position is greater than the max height or less than 0, the game over flag was also set.

Inside the obstacle modules, we derived an algorithm that would check whether or not certain parts of the helicopter collided with the obstacle. The first one checked whether or not the top of the helicopter (rotor blades) hit the obstacle. This was done with this formula:

```
(Obs_X_Pos - Obs_Width <= ballx + 86 &&
  Obs_X_Pos - Obs_Width >= ballx &&
Obs_Y_Pos - Obs_Height <= bally + 26 &&
          Obs_Y_Pos - Obs_Height >= bally)
```

Afterwards, we checked if the bottom of the helicopter collided with the obstacle with the formula below. Essentially ballx and bally keep track of the leftmost and topmost position of the sprite. 86 and 26 are the width and height of the sprite respectively. Therefore we check if the bottom of the helicopter is greater than the obstacle's topmost position and less than the obstacles bottommost position.

```
(Obs_X_Pos - Obs_Width <= ballx + 86 &&
  Obs_X_Pos - Obs_Width >= ballx &&
Obs_Y_Pos + Obs_Height <= bally + 26 &&
          Obs_Y_Pos + Obs_Height >= bally)
```

Finally for any extraneous cases, we checked if the position of the ball was within the position of the obstacles.

((ballx >= Obs_X_Pos - Obs_Width + Obs_X_Step) &&
(ballx <= Obs_X_Pos + Obs_Width + Obs_X_Step) &&
(bally >= Obs_Y_Pos - Obs_Height) &&
(bally <= Obs_Y_Pos + Obs_Height))

5 Diagrams

5.1 Block Diagram

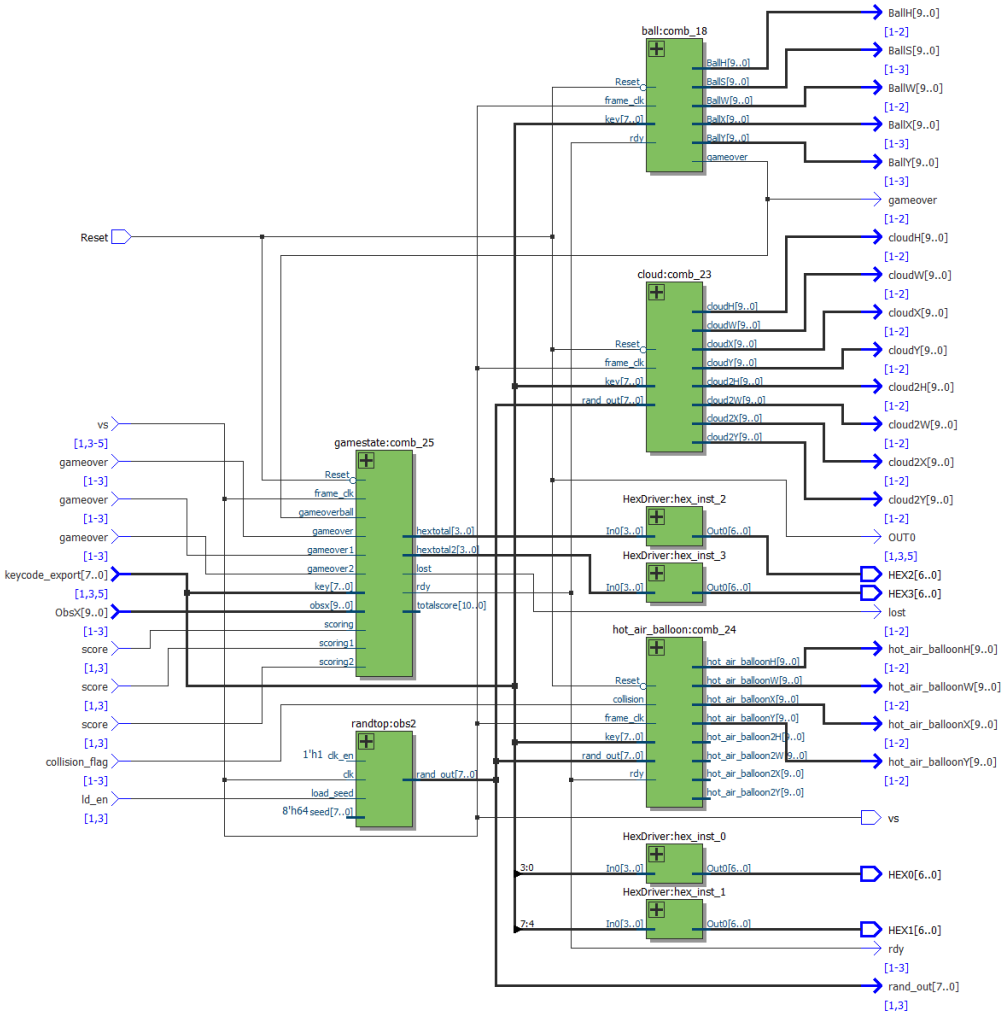

Figure 5: Block Diagram

5.2 Top Level Entity

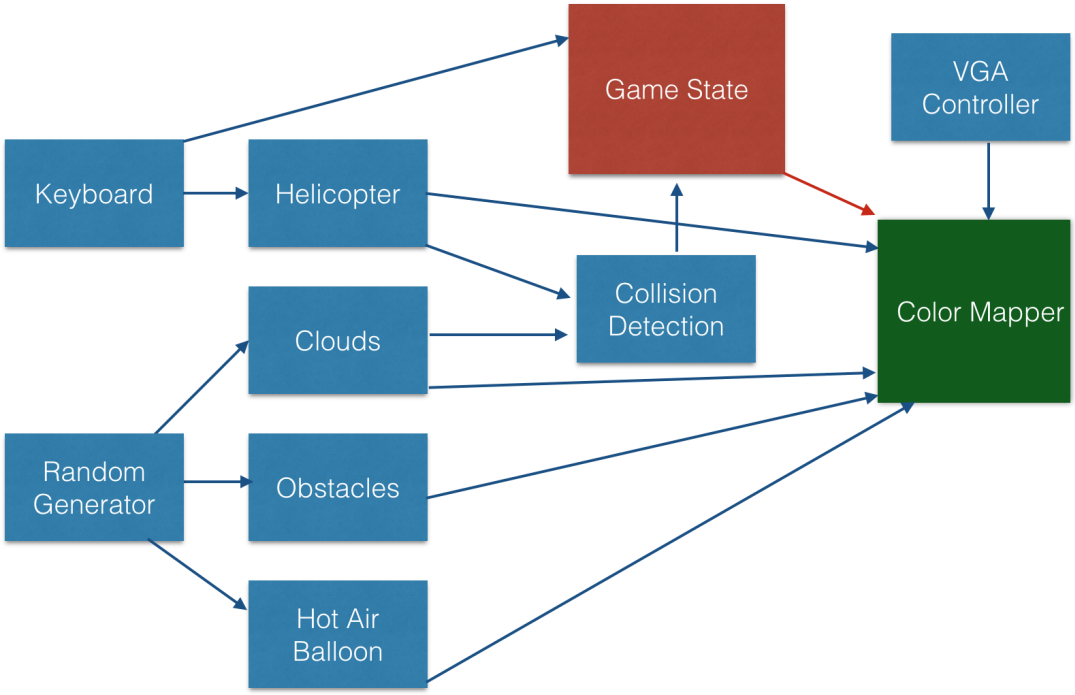

Figure 6: Top Level Diagram

6 Conclusion

6.1 Statistics

Design Statistics Table:

6.2 Final Statement

In the end there were plenty of features that we still want to add but did not have time to implement. For example, we wanted to implement scoring on the screen but we couldn't get it to only increment once after the helicopter passes the obstacle. Another feature that we would have liked to add was different levels. We had a good idea of how to do it, which was to have the user press the number for level of difficulty they wanted and based on that difficulty, we would have changed the speed of the game to make it faster and for more obstacles to

LUT	
DSP	
Memory (BRAM	
Flip-Flop	
Frequency	Mhz
Static Power	mW
Dynamic Power	mW
Total Power	mW

appear at a time.

Although we were not able to add all of these features, we believe that our project turned out well and we were able to take the things we learned throughout the semester the build a project that we were proud of. In the end, we were able to get sprites to work and the general baseline of the game, which we aimed to do from the start of the project.