FPGA Flappy Bird

David Whisler & Sai Nimmagadda

ECE350L

January 19, 2017

1 Introduction

Here we have chosen to implement the cult classic game Flappy Bird by Dong Nguyen on a FPGA. The game logic was implemented in assembly code run on a 5-stage pipelined MIPS-like processor built in Verilog, together with dedicated VGA controller hardware and 7 segment display outputs. Players of the game use a single push button to control the bird and avoid obstacles in the form of pipes moving from right to left with varying gap locations. All graphics were transmitted via VGA to appear on a standard computer monitor. A player's score was continually tracked and displayed on three seven segment displays. As an added "easter egg", players have the ability to use a switch to change the character to an image of Professor Board.

2 Input and Output

2.1 User Input

Only a single button is needed to control the bird's position on the screen, which causes the bird to jump a fixed height. Pressing the button multiple times causes the bird to move higher on the screen, and the bird is continuously falling when not jumping. Because we are sampling the button state so frequently (50 Mhz), debouncing was deemed unecessary and a missed button press was never observed during gameplay. Additionally, a single LED on the DE2 board lights up when the jump button is asserted. This is was used mainly for debugging purposes and visual confirmation that a press was recorded.

As an added bonus, a slide switch can be toggled to change the the bird to an image of Dr. Board (Flappy Board!). This can be toggled at any time during the game, and the character will instantaneously switch while the game continues.

2.2 Output

The majority of the game's output is displayed on the computer screen. The processor updates the positions of each sprite on the screen (pipes and the bird) abstracted as X and Y coordinates, and then these coordinates are passed to dedicated VGA controller hardware to make the sprites appear on the screen. This process is described in more detail in the Additional Modifications section.

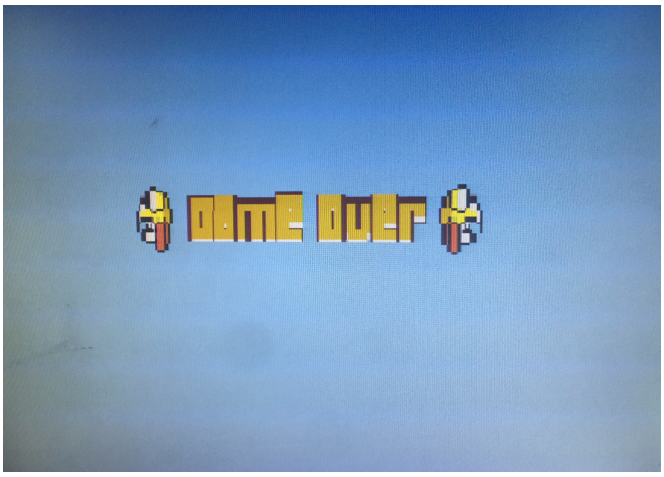

Figure 1: Game Over Screen

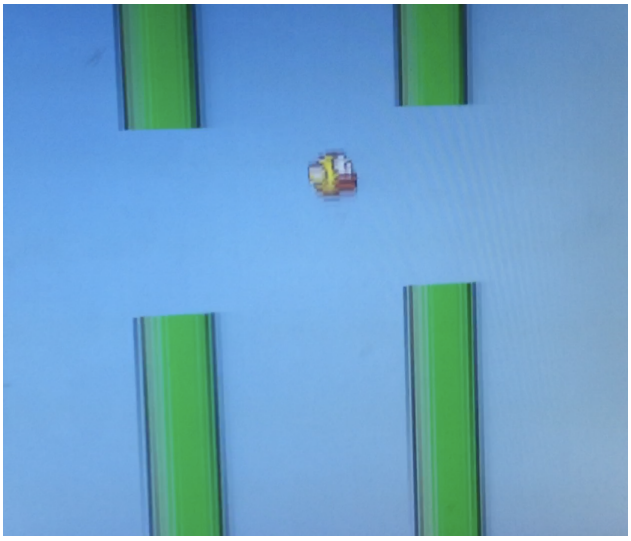

Figure 2: Playing Game with Bird Sprite

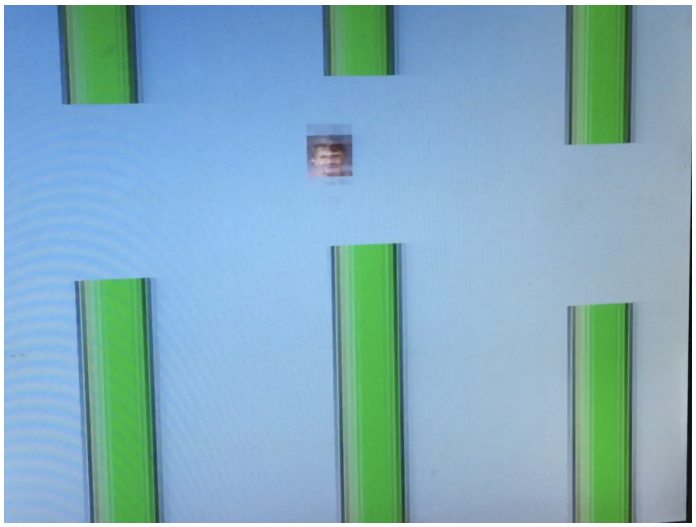

Figure 3: Playing Game with Board Sprite

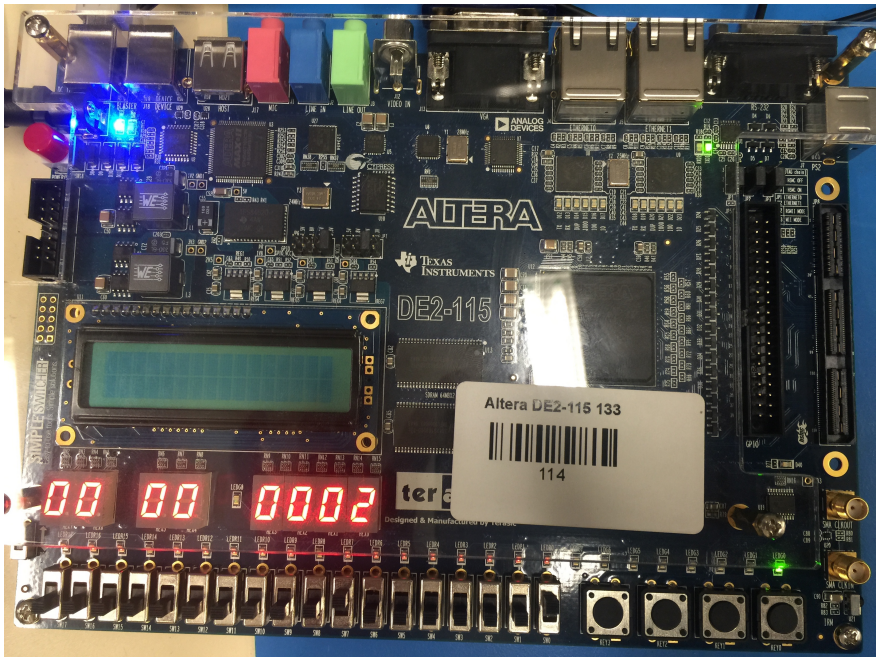

Figure 4: Seven-Segment Display of Score

3 Assembly Program and Additional Code

The assembly language program controls the processor, and implements the main game logic. The game loop first moves all the pipe coordinates to the left, moves the bird down, checks for collisions, and then calls an animation delay routine of 15,000 noop instructions. This synchronizes the processor with the VGA controller hardware to keep the animation speed reasonable. Then, the game score is incremented, and the jump button is checked. If the button is not pressed, the program jumps back to the beginning of the main loop, but if it is pressed, the program jumps into a different main loop that calls the same routines with the sole difference that it makes the bird move up instead of down. A flow chart outlining the high-level assembly program is included below.

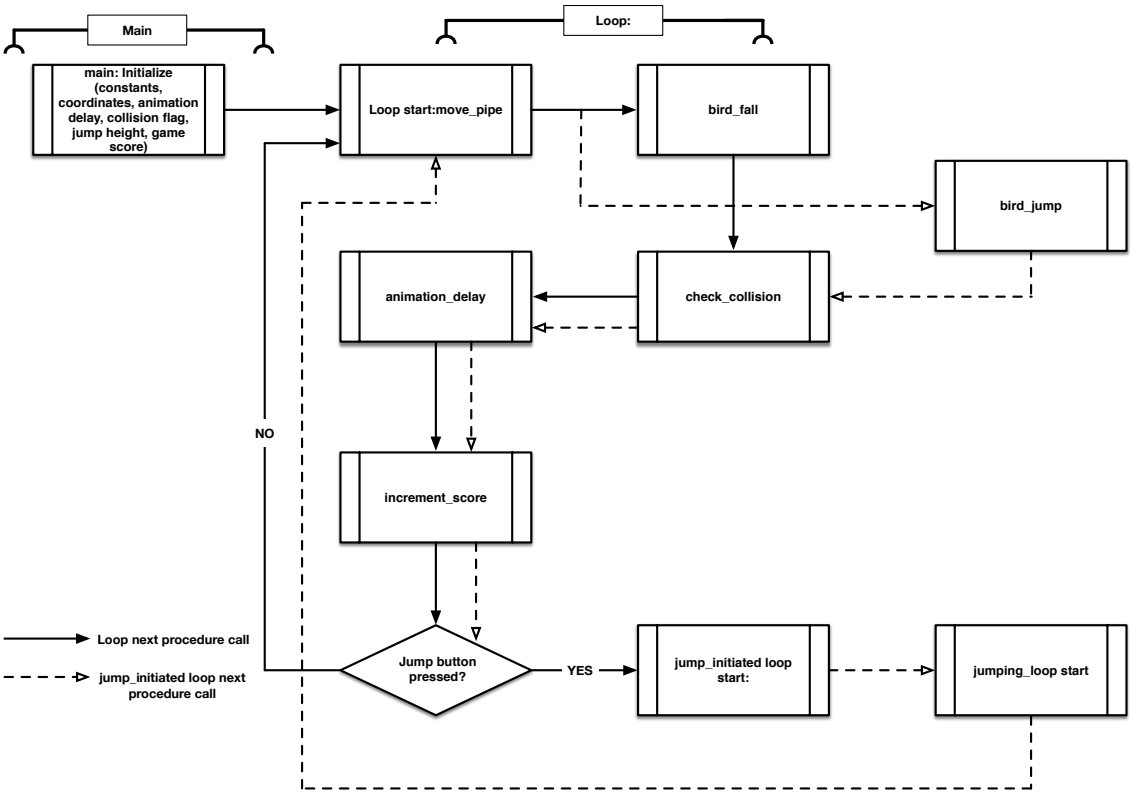

Figure 5: Assembly Software Flowchart

4 Specifications

The main specification for this project was a 50 MHz clock speed for the processor. The processor speed needed to be fast enough to perform all of the required calculations in between each frame. An additional specification was the frequency of the button press checks - this needed to be fast enough to avoid missing user input, and was implemented at 50 MHz (the same speed as the processor).

5 Processor and Additional Modifications

5.1 Processor

A number of additional inputs and outputs were added to the processor in order to implement Flappy Bird. See Table 1 below for an overview of the register assignments.

Table 1: Register Mappings

Register	Function
$r1	Bird Y coordinate (initialized to 240)
$r2	Pipe 1 X coordinate (initialized to 71)
$r3	Pipe 1 Y coordinate (initialized to 200)
$r4	Pipe 2 X coordinate (initialized to 261)
$r5	Pipe 2 Y coordinate (initialized to 350)
$r6	Delay counter (used for VGA timing)
$r7	Delay counter maximum (set to 15,000)
$r8	Collision flag
$r9	Jump height of bird (set to 12)
$r10	Game over flag (initially zero)
$r11	Game score
$r12	Jump height counter
$r13	Screen lowest limit (set to 450)
$r14	Pipe left limit
$r15	Bird X left limit (initialized to 320)
$r16	Pipe 1 right side
$r17	Pipe 2 right side
$r18	Pipe 3 Right side
$r19	Pipe 3 X position
$r20	Pipe 3 Y position
$r21	Random Y position

5.1.1 Additional Outputs

The x and y coordinates of each pipe center as well as the y coordinate of the bird were taken as 32 bit outputs for the processor, coming directly from the register file. In addition, the game score was assigned to an output, and a gameover flag bit was taken as the 0th bit of register 10. For reserved registers used as game objects, the output wires of these registers were assigned to additional outputs within the regfile as follows:

```
assign bird_y = readOut[1];
assign pipe1_x = readOut[2];
assign pipe1_y = readOut[3];
assign pipe2_x = readOut[4];
assign pipe2_y = readOut[5];
assign pipe3_x = readOut[19];
assign pipe3_y = readOut[20];
assign gameover_flag_long = readOut[10];
assign game_score = readOut[11];
```

5.1.2 Additional Inputs

In addition to the outputs from the register file, there were some additional inputs to the register file. In order to record button presses and to act on a button press, the STATUS register was modified to hold a zero-extended version of the button input wire. From this, a `bex` command could be used to act conditionally on a button press, in the case of our game, a jump for the bird. Also, a collision flag from dedicated collision detection hardware in the VGA controller was zero-extended and assigned to register 8 for use in the processor assembly program. A similar process was used to poll for randomly generated pipe Y coordinates from the LFSR so that new pipes could be updated randomly on the right side of the screen as the game progressed.

5.2 VGA Controller

While the processor dealt with sprite locations at an abstracted level of X and Y coordinates, the VGA controller handled the task of rendering them at the specific locations passed to it from the processor. To achieve this, the base VGA controller given in the skeleton was modified. Instead of having a single ROM module to hold the image data, this was split into a ROM module to hold static image data with all the necessary sprites and images in a pre-initialized MIF file, and a RAM module that would hold a dynamic game screen, initially with the default background color at every pixel location. For a given sprite on every frame, the VGA controller reads the necessary pixel data from the static image file, and writes it to the correct location in the dynamic image file.

In order to do this, the address generation logic was heavily modified. The dynamic_ADDR register holds the current location in the dynamic game screen file, which increments by one for every pixel on the screen. If the dynamic_ADDR enters a pixel range where the VGA controller needs to render some of a given sprite's pixels, the VGA controller loads the last saved location in the static image file of the sprite being drawn, and keeps incrementing the static_ADDR register, which holds the current location in the static image file. This process may switch between many different sprites as the entire game screen is drawn row by row, and dedicated registers hold the current locations in the static data file for each sprite. Conditional logic was also used to draw a gameover screen if the gameover flag was set, or to use the Board sprite in place of the bird sprite with the slide switch input.

The static image data MIF file itself was generated using a custom script written in Python. All the sprite prototype images were placed in a 256 level .bmp file, and the Python script read the pixel data and converted it to RGB values at each pixel. Then, a color palette was made from these values, and each pixel was mapped to the correct memory location in the color palette to mimic the VGA architecture proveded with image data and image index. A text file was written automatically with the correct memory width and depth with the .mif extension to be used by the Quartus software. A copy of the Python script is included in the assignment submission.

In addition to the necessary logic to draw objects on the screen, the logic for collision detection can be found inside of the the VGA controller. The collision detection module within the VGA controller takes in 16 inputs for each of the four boundaries of the bird and pipes on the screen (left, right, top, bottom). By using behavioral inequalities and logical operators, overlaps between object boundaries were detected. If a collision occurred, the the collision flag was set to 1. This output of the VGA controller was passed as an input to the processor and stored in a 32 bit register ($r8)

5.3 Skeleton

Two additions were made to the top-level skeleton. One of these additions was outputs for three of the seven-segment displays on the DE2 board to display the current game score. A module was written to convert a decimal number to seven segment display outputs using behavioral Verilog.

The second addition added to the skeleton was a 32 bit linear feedback shift register (LFSR) to generate random pipe Y coordinates continuously. A LFSR generates a pseudo-random sequence of numbers by tapping some of its internal bits with XOR gates, and assigning this calculated bit back to itself, creating feedback. This pseudo-random number was converted to the proper scale for Y coordinates by scaling it with a modulus function, and shifting it with an adder.

6 Testing

Testing of the game was relatively simple. The FPGA was programmed with the compiled project, and then the VGA output on the DE2 board was used to visually confirm the game's functionality. Buttons inputs were debugged using some of the LED outputs on the DE2 board.

7 Challenges

During the development of the game, a number of challenges were faced. The biggest challenge was how to actually draw graphics on the monitor via VGA. Because graphics are a major part of Flappy Bird, we wanted to mimic the original game's graphics to the best of our ability. In order to avoid manually editing values in the mif files, the process was expedited via a Python script to generate the necessary files (see section 5.2 and attached script to project).

Additionally, when first trying to draw images on the screen, we encountered instances where nothing would be drawn, or a frozen screen. This was due to inconsistent timings and synchronization between the VGA controller (which draws the objects) and the processor (which updates the locations of the objects). In order to overcome this, we had to first fix our usage of blocking and non-blocking assignments within always blocks in the VGA controller. Additionally, a counter variable was created in the VGA controller to add the correct delay to every drawing procedure. This allowed for synchronization between updating an object's position and the drawing of that object.

Lastly, we initially struggled with how to generate locations for incoming pipes. At first pass, pipe locations were hard coded to a set of predefined coordinates. To increase the difficulty and entertainment value of the game, we chose to implement a pseudo-random number generator in the form of a LFSR. (described above in section 5). The challenge in this was how to generate random n-bit numbers that would be within the range of coordinates on the screen. This was overcome by scaling and shifting the 32-bit output from the LFSR using modulus and an adder to ensure its value was within the given height of the screen.