Flappy Bird Redux

Moon Zhu
Faculty of Engineering
University of Auckland
Auckland, NZ
xzhu664@aucklanduni.ac.nz

William Brunton
Faculty of Engineering
University of Auckland
Auckland, NZ
wbru608@aucklanduni.ac.nz

Sandeep Manilal
Faculty of Engineering
University of Auckland
Auckland, NZ
sman840@aucklanduni.ac.nz

***Abstract*— We were tasked with implementing a recreation (with some added mechanics) of the hit mobile game Flappy Bird on a Cyclone V FPGA dev board. We were required to use most of the peripheral and I/O capabilities of the board. We divided the game into subsystems that could be created individually and then linked together, transferring data around by way of a top-level module. The game is displayed via compositing of sprites and text; ROMs are used to store image and font data. Some information is displayed on the dev board itself via seven-segment displays. The player can collect various powerups to assist in reaching a high score. Targeted optimisation of various modules reduced Logic Element usage, though potential maximum frequency improvements lie in reducing critical path length. Future additions to the game could include more sprite variation, an enhanced control scheme, and sound and more advanced visual effects techniques.**

Keywords—Flappy Bird, VHDL, FPGA, Altera, VGA

I. Introduction

This report details the design and implementation of the Flappy Bird game created by Group 4 (MOSFET Gaming) using VHDL on the supplied Altera DE0-CV FPGA board. The project aimed to replicate the gameplay experience of the original Flappy Bird using an FPGA board, in which the player controls a bird moving horizontally across a scrolling background.

The objective is to fly through the vertical gaps between pipes which are generated and scroll past continuously. The location of the gap between the upper and lower pipe is randomly determined. The pipes are uniformly spaced horizontally. The bird initially has three lives and loses one each time it collides with a pipe. Losing all health results in a game over.

The bird can move up when the wings flap (controlled using a PS/2 mouse). If the bird is not flapping, it will free-fall towards the ground. The bird is dead if it hits the ground, resulting in an instant game over.

To create an increasing level of difficulty as the game progresses, the obstacles in the game slowly speed up.

There are randomly generated powerups which can boost life or provide other benefits, such as widening the pipe gap or disabling collision for a few seconds.

A training mode is accessible via DIP switch, disabling collision damage to assist players in honing their skills.

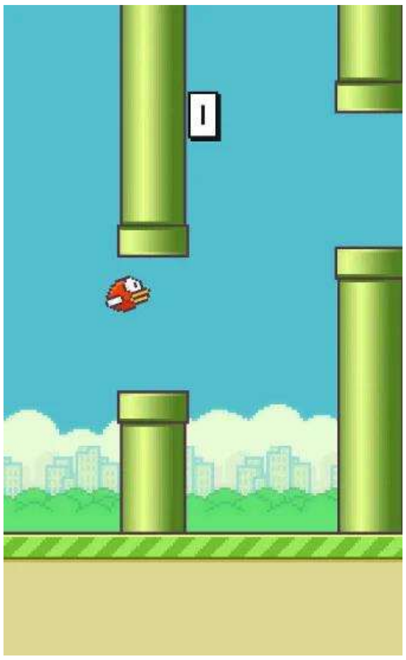


Figure 1: Screengrab from the original Flappy Bird game.

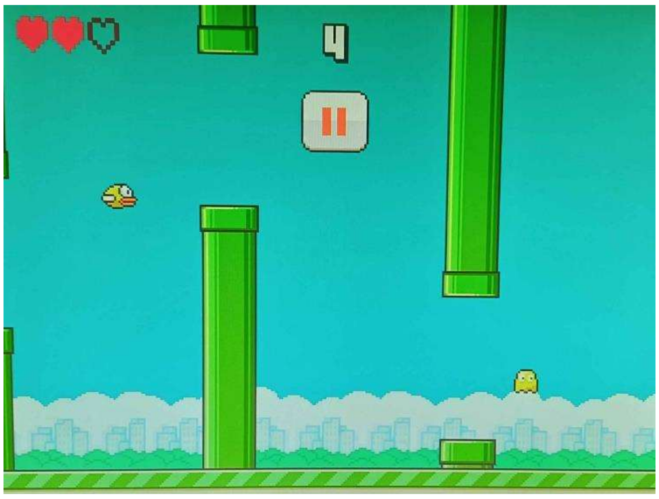


Figure 2: Our recreation, with added features such as health and powerups.

II. System Design Overview

The game implementation is highly modular and comprises several components:

- Flappy Bird module: Top-level module of the system. Controls gameplay via state machine.
- Graphics controller: Handles all output to the VGA port.

- Bird controller: Calculates and updates the bird's position on-screen.
- Pipe controller: Positions pipes and moves them across the screen.
- Collision controller: Checks and reports the bird's overlap with pipes, powerups, and the ground.
- Score controller: Updates the player's score and calculates live game difficulty.
- Powerup controller: Creates and moves powerups.
- Random generator: Constantly generates a 16-bit random number.
- Mouse controller: Wraps the mouse communication component.
- Util package: Useful constants and type definitions.
- Sprite package: Generated package holding sprite constants required to display from the sprite ROM.

III. Detailed System Design

A. Flappy Bird module

Being the top-level module, all the controller components are instantiated in this component. This module also serves as the 'host' for all the signals passed between subcomponents or used in the state machine. The state machine, which manages the gameplay loop, resides in a process in this module and is clocked at 60Hz.

B. Graphics Controller

The graphics controller generates a VGA signal to display game elements on the screen. It uses a raster-based rendering approach to draw sprites and backgrounds.

Image and font data are stored in two ROMS – font in an 8-bit, 1024-word ROM and sprites in a 12-bit, 32,768-word ROM. The font ROM stores an entire row of each character in every word, as it only supports one-bit data. The sprite ROM stores a single pixel in each word, as it supports 12-bit RGB (though complete black is reserved as a transparent pixel). The sprite ROM is set up with two address and data ports ('A' and 'B'), allowing for two pixels to be retrieved simultaneously. This is critical for the rendering process, as some sprites have transparent pixels that must correctly show the colour of any sprites behind them in the Z-plane. The 'B' port is used for background sprites, and the 'A' port is used for foreground sprites. The single port of the font ROM is used for UI text and is rendered in front of both sprite layers.

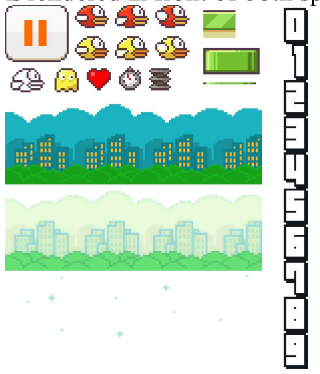


Figure 3: The sprites used in-game.

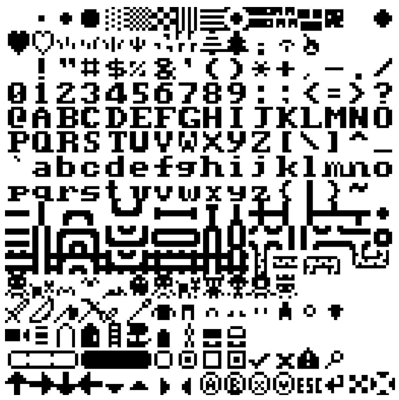


Figure 4: The custom font used in-game. Note the filled and unfilled heart glyphs in the top left, used to represent health in the UI.

The internal VGA Sync component sweeps across the screen and outputs the coordinates for the current pixel. The controller uses the following procedure to calculate the colour (note that any consequent address writes to the same port will override the prior write):

1. The colour is set to the background colour.
2. If the current pixel is within the background, the position inside the background sprite is found and the appropriate address is sent to the 'B' port. This is repeated for the 'stars' sprite (if the game is in night mode), and each pipe's head and body.
3. If the powerup sprite is active and the current pixel is inside it, the address is calculated and sent to the 'A' port. This is repeated for the bird (handling animations and a red tint if the bird's just lost health), ground, score, pause icon (if the game is paused), and active powerup (if the user has a powerup active).
4. The "Click to Start" or "Click to Restart" text is rendered. This involves checking whether the pixel is inside the string, finding the character in the string at the current pixel, extracting the row and column of the character, and assigning the appropriate signals for the character ROM to output the font bit for the character. This is repeated for the "TRAINING" text, and the health display (which uses characters 16 and 17 for filled and empty hearts respectively).
5. If the pixel was in a layer 'B' sprite, and the returned colour was not black, the current pixel is set to that colour. This is repeated for layer 'A' and the text layer (where the colour is assigned by each text component). The ordering of this step results in correctly calculated layering of sprites.
6. The computed pixel is assigned to the VGA red, green, and blue output ports, to be converted into an analogue signal via resistors.

Due to this lengthy process, the Graphics controller requires a significant number of input ports and is by far the most complex component of the game.

A significant amount of optimisation was continuously applied as the controller was being written. Both available ports of the 'altsyncram' component were used to access sprite data. Modulo operations were only performed with powers of two (resulting in the compiler simplifying to a bitwise 'AND'), which was achieved by carefully crafting the sprites to be of specific sizes. This also ensured that no DSP blocks were required for large multiplications, as these were optimised away by the compiler to bit shifts. These optimisations ensured that the Logic Elements used by the component were kept to a minimum, with a final count of about 1,200. This also helped keep the critical logic paths of the rendering pipeline short.

C. Bird Controller

User input from buttons and switches is processed to control the bird's movement and interact with the game. The bird's Y velocity is stored and computed internally, to achieve the effect of gravity on the bird.
Mouse input is processed to make the bird fly up when the left click button is pressed. An internal memory signal is used to ensure the click is only registered on the first frame it is detected, and not continuously.

D. Pipe Controller

The game processes an array of three pipes internally. The gap between the top and bottom pipes is fixed and constant, though can be expanded if the user has a Spring powerup. Pipes are internally stored as a single point onscreen, positioned in the centre of the pipe's gap. The graphics and collision controllers use this alongside various constants to create the effect of two separate 'halves' of pipe on the top and bottom.
The pipe controller manages each pipe's horizontal movement, shifting it some number of pixels (computed via the current difficulty) to the left at 60Hz. When a pipe disappears from the left edge of the screen, it is immediately repositioned off the right edge, and given a new Y position.
To generate these randomised Y positions, the pipe controller utilises the random number generator. The 7 lower bits of the random number are added to a precalculated constant, resulting in a random offset centred vertically on-screen.
These mechanics combined with careful initial positioning of the pipes creates the illusion of an infinite number of evenly spaced pipes.

E. Collision Controller.

The collision controller module monitors the position of every object on the screen with a collision box, and outputs an enumerated signal to specify the type of collision for each frame. All collision boxes are rectangles, which makes calculating overlaps easy but ignores transparent pixels in sprites – the bird's collision box has corners that the bird's sprite does not. There are four types of collision:

- None: No collision in the current frame. This is used to reset the collision type.
- Pipe: A collision with a pipe. This type will trigger damage, if the player does not have any of the invincibility mechanics applied that frame.
- Powerup: A collision with the powerup. This will trigger the powerup's effect.
- Ground: A collision with the ground. This will immediately result in a game over unless the player has the Ghost powerup or is in Training Mode.

F. Score Controller

The Score controller keeps track of each pipe's current and last position. If the pipe's current position is behind the bird and last position ahead, the score is increased by one. This method only works up to a certain pipe speed, as a fast enough pipe would wrap around before the controller counted it, but the game's current speed system does not reach a high enough value for this to happen.
This module also handles difficulty, which is calculated as such:

$$\text{difficulty} = \min\left(9, \frac{score}{10} + 1\right)$$

The difficulty is capped at 9 to stop the pipe speed increasing indefinitely.
To implement this difficulty, pipe speed is calculated by adding 3 to the difficulty value; the result is how many pixels the pipe will move over the next *two* frames, including the remainder on the 2nd frame if the result is odd. This allows for half-pixel-per-second speed increments, allowing for difficulty/speed increases that do not get immediately out of hand.

G. Powerup Controller

Controls the spawning and movement of powerups based on the random number generator's output. There are four types of power ups:

- Health: Adds one to the player's health. Can only generate if the player is not at maximum health.
- Slow: Multiplies the speed of pipes and powerups by 0.8.
- Ghost: Disables player collision.
- Spring: Widens the gap between pipes by 20 pixels.

The latter 3 powerups are active for 5 seconds before they run out.
Each frame, if the on-screen powerup is inactive there is a 1 in 128 chance for a generation attempt to be made. If this succeeds, the horizontal position of each pipe is checked. If there is determined to be no overlap, generation will proceed. A randomly chosen type will be used – if the player is at full health and the type chosen is the Health powerup then the type will be replaced by either a Ghost or Spring powerup.
The powerup is then assigned a random Y position and moves across the screen exactly like a pipe. Upon reaching the end or being touched by the player, it is set to inactive for the generation process to begin once more.

H. Random Generator

This component generates a 16-bit random integer signal between 0 and 65535 using a linear feedback shift register (LFSR). The signal is used to randomize the position of the pipe gaps, the spawning of the power-ups and the type of power up generated. The underlying mechanics for the random number generator are as follows:
A 16-bit internal signal is set to an initial seed. Each game frame, the bits at positions 15, 13, 12, and 10 are XORed and then concatenated with the lower 15 bits of the signal to produce a pseudo random 16-bit number. This is then converted to an integer, which is output as a signal. The randomization of obstacles and powerups provides variability and improves gameplay.

I. Mouse Controller

This component simply contains the Mouse subcomponent, providing it with a 25MHz clock and passing through the mouse clock and data ports.

J. Utils Package

This package contains constants used for various purposes throughout the game. Some custom types are also defined for ease of passing related data between modules – examples being a position type containing x and y components, and a powerup type containing all relevant data for the on-screen powerup.

K. Sprites package

This package is programmatically generated by a Python script, alongside the ROM initialisation MIF file. It contains various constants required to operate the ROM – namely sprite dimensions and memory offsets, and the address width required to store all the pixels.

L. BCD to Seven Segment

Instances of this component are used to handle displaying the current score and bird health on four of the seven-segment displays on the FPGA board.

M. High-level Block Diagram and State Machine

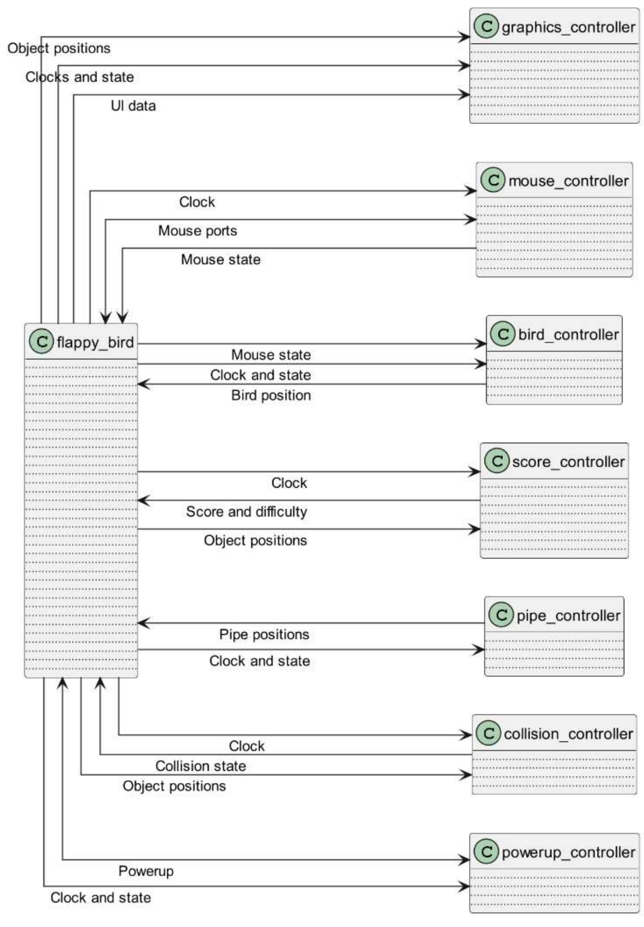


Figure 5: Block Diagram showing the system modules and their connectivity with the top-level module.

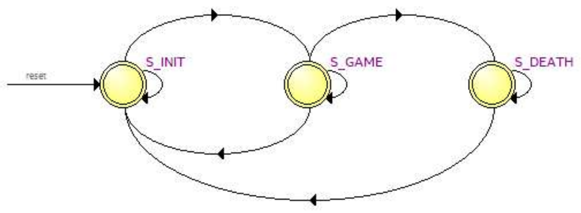
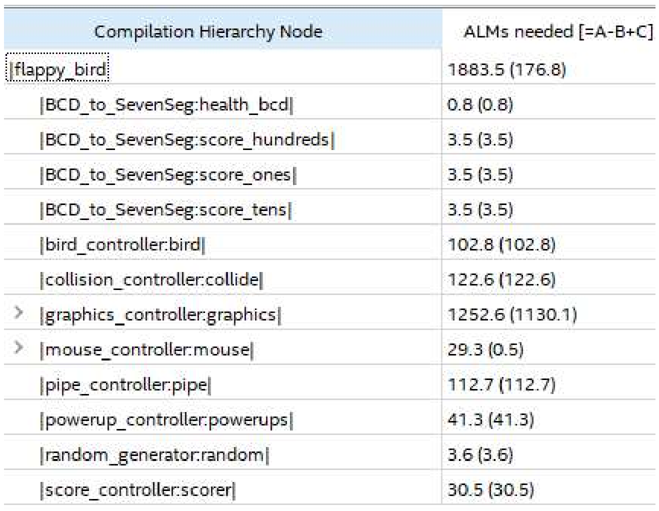


Figure 6: State Machine of the core gameplay loop.

As previously stated, the game's state machine resides in the Flappy Bird top-level module. The initial state is S_INIT. Clicking the left mouse button will advance to S_GAME, where the main game is run; dying will advance to S_DEATH, where clicking again will set the state to S_INIT and then immediately S_GAME. Pressing key 0 on the dev board will set the state to S_INIT at any time, as will toggling the Training Mode DIP switch whilst in the S_GAME state (to avoid abuse of Training Mode).

Aside from maintaining the gameplay state, this process handles events like the player losing health or getting powerups, and the difficulty-controlled movement speed of various in-game objects.

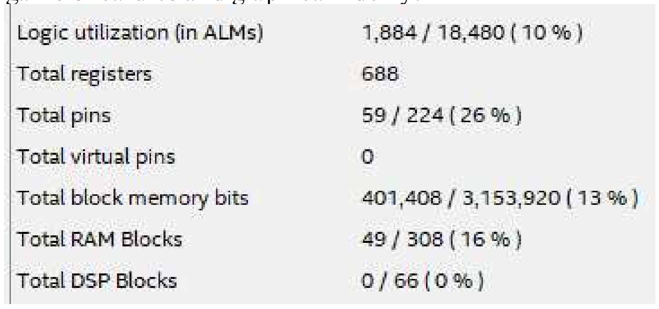
N. Resource Usage



Compilation Hierarchy Node	ALMs needed [=A-B+C]
|flappy_bird	1883.5 (176.8)
|BCD_to_SevenSeg:health_bcd|	0.8 (0.8)
|BCD_to_SevenSeg:score_hundreds|	3.5 (3.5)
|BCD_to_SevenSeg:score_ones|	3.5 (3.5)
|BCD_to_SevenSeg:score_tens|	3.5 (3.5)
|bird_controller:bird|	102.8 (102.8)
|collision_controller:collide|	122.6 (122.6)
> |graphics_controller:graphics|	1252.6 (1130.1)
> |mouse_controller:mouse|	29.3 (0.5)
|pipe_controller:pipe|	112.7 (112.7)
|powerup_controller:powerups|	41.3 (41.3)
|random_generator:random|	3.6 (3.6)
|score_controller:scorer|	30.5 (30.5)

Figure 7: Logic Element breakdown by entity. The Graphics Controller uses the majority of required L.E.s.

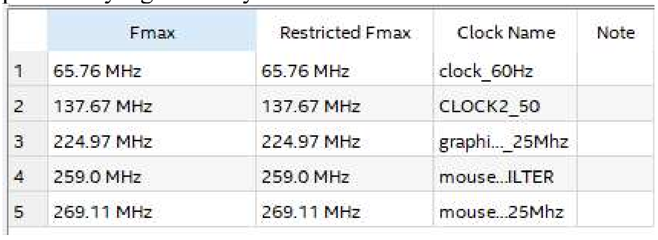
Time spent optimising Logic Element usage of the game resulted in a resource efficient synthesis relative to the game's features and graphical fidelity.



Logic utilization (in ALMs)	1,884 / 18,480 (10 %)
Total registers	688
Total pins	59 / 224 (26 %)
Total virtual pins	0
Total block memory bits	401,408 / 3,153,920 (13 %)
Total RAM Blocks	49 / 308 (16 %)
Total DSP Blocks	0 / 66 (0 %)

Figure 8: Physical FPGA resource usage. Note that not a single DSP block has been allocated!

A substantial number of memory bits are used. This is due to the size of the sprite ROM, as it requires a word per pixel for every sprite used. For convenience, the ROM reserves the smallest power of two that can hold all the sprite data – reducing this to a perfect fit would reduce memory bit usage, potentially significantly.



	Fmax	Restricted Fmax	Clock Name	Note
1	65.76 MHz	65.76 MHz	clock_60Hz	
2	137.67 MHz	137.67 MHz	CLOCK2_50	
3	224.97 MHz	224.97 MHz	graphi..._25Mhz	
4	259.0 MHz	259.0 MHz	mouse...ILTER	
5	269.11 MHz	269.11 MHz	mouse...25Mhz	

Figure 9: Maximum clock frequencies.

The most important maximum frequency is that of the 60Hz clock. At 65MHz, it is already many magnitudes higher than the clock is run at but has potential for improvement. Analysis of the game's logic to find critical paths would prove useful in improving this frequency.

IV. Ideas for Future Improvements

A. Varying sprites

A possibility for improvement would be variations in sprites for elements like the bird, background, and pipes. These could be unlocked upon reaching certain scores in-game, to provide gameplay incentives.

B. Bird movement

Richer interaction with the mouse, such as allowing control via movement, are potential future additions. However, the effect this would have on game balance and difficulty should be considered.

C. Additional Features

Sound and visual effects such as a score increase sound or the bird rotating as it flies upwards and downwards would add to the gameplay experience.*Block Diagram created with PlantUML.*