

# COMPSYS 305 Mini Project Team 31

Jenny Nguyen  
Department of Electrical, Computer  
and Software Engineering  
The University of Auckland  
Auckland, New Zealand  
jugn310@aucklanduni.ac.nz

Shriya Singh  
Department of Electrical, Computer  
and Software Engineering  
The University of Auckland  
Auckland, New Zealand  
sisn751@aucklanduni.ac.nz

**Abstract**— Flappy Bird is a side scrolling game where the players are tasked with maneuvering a bird through small gaps between pipes. This report outlines our system design and implementation, which includes aspects such as the finite state machine (FSM) that handles the game modes and details on collision detection and object display. Design decisions details cover the use of linear feedback shift registers (LFSR) for pseudo-random number generations. This is along with possible future improvements for our game such as refining collision detection and other graphical features to enhance visual design and gameplay experience.

**Keywords**—Flappy Bird, FPGA, LFSR, FSM, VGA, Game

## I. SYSTEM DESIGN

### A. Design Specifications

The Flappy Bird game is implemented on an Altera Cyclone V FPGA and is included with the DE0-CV development board. A PS/2 mouse interfaces with the FPGA to allow user control of the bird. The FPGA generates a 640x480 pixel Video Graphics Array (VGA) output signal for displaying the game.

### B. Game Description

Flappy Bird is a rendition of the classic side-scroller video game where the player navigates the bird to fly between the gaps in the pipes, avoiding collisions.

In our design of this classic game, the bird is influenced by a gravitational force, causing it to fall towards the ground when no user input is detected. To keep the bird airborne and navigate it through the gaps in the pipes, the user must click the mouse connected via the PS/2 interface. The bird interacts with different objects, such as pipes and coins.

The pipes serve as obstacles that the players must navigate the bird through. The pipes appear from the right side of the screen and move horizontally towards the left. Each pipe protrudes from both the top and bottom of the screen, creating a vertical gap between the upper and lower segments. The location of the gap the bird must fly through is not fixed, and it is generated randomly using a Linear Feedback Shift Register (LFSR); this ensures that the game remains challenging and unpredictable.

As the bird navigates through the gaps in the pipe, players also aim to collect coins to boost their score more rapidly. Each coin collected adds 2 points to the score, while successfully passing through a pipe adds 1 point. The LFSR also determines the coin's location, contributing to the game's challenging dynamics.

Depending on the player's chosen mode, the speed at which the bird encounters the objects may also increase over time, creating a more significant challenge for the player.

### C. Game Modes

Our Flappy Bird game begins at the menu screen; the player can select their preferred game mode using either KEY2 or KEY3 push buttons on the DE0-CV board. These buttons start the game in Training Mode or Normal Mode, respectively. In both modes, the game starts with the bird hovering in the middle of the screen, and no obstacles are visible.



Figure 1: Menu Screen

Initially, only the bird is visible. With the first left click, the game begins, revealing the obstacles. The bird continues to fly in the vertical direction, as described before. This continues until the player reaches the first obstacle, requiring them to guide the bird through the gap in the pipes.



Figure 2: Bird Hover Before Game Start

In TRAINING Mode, the bird has three lives, displayed at the top right of the screen. The lives decrease each time the bird collides with the pipes until the bird has no remaining lives left, in which case the game is over. If it collides with the ground, the game is over, regardless of the player's number of lives.



Figure 3: Training Mode

In NORMAL Mode, the user has a traditional flappy bird experience. The player has a single life, and the game ends if a collision occurs between the pipes or the ground.



Figure 4: Game Over Screen in Normal Mode

In both game modes, the score increments each time the bird successfully passes through the gap in the pipes and collides with the coins. Pseudo-random pipe generation is applied in both modes, allowing the player to train in an environment replicating the NORMAL Mode.

NORMAL Mode has three levels, which increase proportionally to the score accrued. Every 10 points, the Level of the game increases (up to Level 3), and the speed of the pipes and coins increase constantly to create difficulty. The game level is shown at the bottom left of the screen.

In TRAINING Mode, the Level is stagnant at Level 1 throughout the whole game and is not affected by the score. This feature allows the player to acclimatize themselves to the gameplay.

Both modes also allow players to change their game background color between Black, White, Red, Green, Blue, Yellow, Magenta, and Cyan using the SW0, SW1, and SW2 switches on the FPGA Board. They also have the option to add clouds to their background using the SW9 switch on the

FPGA Board. These allow for a more immersive experience for the player.

#### D. Finite State Machine

The finite state machine, as shown, controls the different states of the objects in our Flappy Bird game. It is designed as a Moore-type finite state machine, as the output depends on only the current state value. The machine oversees the status of the bird and the number of lives it has if it is in TRAINING Mode.

The four states the game recognizes are as follows:

1. **Start Game:** This state represents the game's starting screen, where the title of the game, game modes, and corresponding keys are displayed. It is reached at the start of the game and when the game is reset after the bird's death.
2. **Training Mode and Normal Mode:** Upon reaching either of these states, the bird will hover in the air, awaiting the mouse's left click. The games will then respectively start and run until either collision with the ground or until the bird's lives run out.
3. **Game Over:** All bird movement stops, and a game over screen is shown to the user. They will have to go back to the Start Game screen to start again using the reset key on the board.

#### State Transitions:

Following are the State Transitions of the four states in our game, along with a description of how part of the logic is handled in individual component files (under "External Checking"):

1. **Start Game:**
  - a) **Set:** If the current state is Start Game or if the current state is Game Over and the player presses the push button KEY2.
  - b) **Next:** If the player presses the push button KEY2, the state proceeds to the Training Mode State; if KEY3 is pressed, the state proceeds to the Normal Mode State.
  - c) **External Checking:** This state is checked by the "menu\_on" signal, which determines if the menu screen is shown at the start of the game.
2. **Training Mode**
  - a) **Set:** If the current mode is set to training by pressing KEY2.
  - b) **Next:** If the signal "life" is set to zero, i.e., no remains lives for the player, or the signal "ground\_collision" is set to one, i.e., the bird has collided with the ground.
  - c) **External Checking:** The signal "ball\_on" checks this state, which displays the ball. This is also checked by the signals "score\_on," "level\_on," and "hearts\_on," which displays the score, level, and hearts, respectively.

### 3. Normal Mode:

- Set: If the current mode is set to normal by pressing KEY3.
- Next: If the signal “collision” is set to one, i.e., the player collides with an obstacle, or the signal “ground\_collision” is set to one, i.e., the bird has collided with the ground.
- External Checking: The signal “ball\_on” checks this state, which displays the ball. The signals “score\_on” and “level\_on” also check this, which display the score and level, respectively. This is also checked by the signal “pipe-x\_motion” to determine the speed of the pipes according to the state. They allow the pipes to adjust their speed to the score in Normal Mode.

### 4. Game Over:

- Set: If the mode is training and “life” is set to zero or “ground\_collision” is set to one. If the mode is Normal, “collision” or “ground\_collision” is set to one.
- Next: If the player presses KEY2, it will proceed to the next state, Start Game.
- External Checking: The signal “game\_over\_on” checks this, determining if the “game over” screen is displayed.

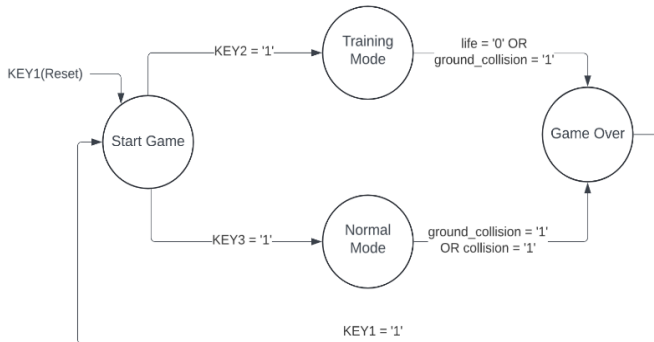


Figure 5: Finite State Machine

**Inputs:** The state machine takes in essential inputs from the bird and pipe if it collides with the ground or pipes. These signals are “ground\_collision” and “collision” respectively. These inputs decide whether to end the game, keep it going, or decrease the lives. The inputs also include two push buttons (One for setting and menu navigation (KEY2) and 1 for menu navigation only (KEY3)).

**Outputs:** The state machine has three outputs. These outputs are the “lives,” “state\_out,” and “reset\_out.” The output “lives” is used as an input to the hearts component to determine the number of hearts to display. “state\_out” is given to components such as Level and Score so they can read the state and determine their output as described above.

The output “reset\_out” is also read externally by the “game\_over” to reset the game and change its state to Game Start.

## II. IMPLEMENTATION

### A. Object Display

The objects were displayed by cascading multiplexers and employing select bits for each object to prioritize signals. This approach allowed us to layer the objects in priority order in which we would like the display displayed. In Flappy Bird, each object was implemented as its unique entity within the top-level entity; it had two critical signals needed for our multiplexer: ‘object\_on’ and ‘object\_rgb.’ The ‘on’ signal acted like a select bit that determined whether a component was displayed on the screen, while the 3-bit RGB (Red, Green, Blue) signal specified the object’s color. These two signals, along with an additional input form the three inputs of our multiplexer:

- select\_bit: Derived from the current object “on” signal.
- rgb\_in\_1: RGB signal of the current object
- rgb\_in\_0: RGB signal of the previous component, connected to the previous ‘rgb\_out’ (the output of the previous multiplexer)

These inputs work together to create a common output called the “rgb\_out,” which is cascaded into the following component as the “rgb\_in\_0” input. These are stacked according to their priority of display.

### B. Collision Detection

The VGA component implemented our collisions in the game, both obstacle and gift, by checking if the location of the bird’s x and y pixels overlapped with the obstacle’s pixels. This overlap was detected and passed as a signal to the Finite State Machine (FSM).

The three collision types we implemented were as follows:

- “collision”- This signal indicated bird collision with the pipes
- “coin\_collision”- This signal indicated bird collision with the coins
- “ground\_collision” indicates a bird collision with the ground.

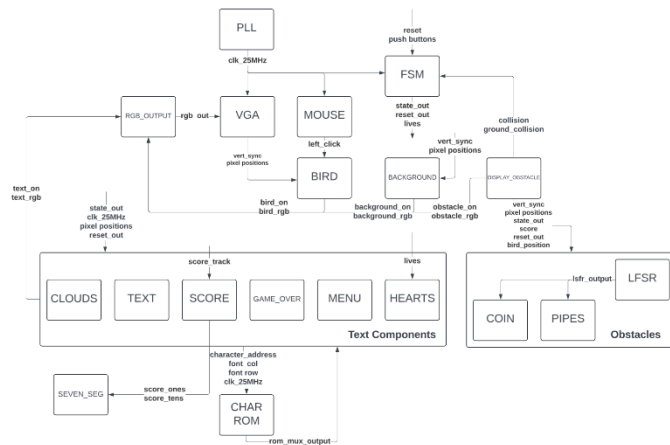
In NORMAL Mode, the collision signal resulted in an instant game over, whereas in TRAINING Mode, it caused a life to be deducted.

We encountered an issue where a single collision would often deplete all lives in TRAINING Mode. This was likely due to the collision lasting multiple frames and being detected multiple times. We implemented a collision flag to prevent the user from losing multiple lives upon a single collision. This flag was activated when the bird initially contacted an obstacle and remained active until the bird moved into an area without obstacles.

### C. Removing the object upon collision

Our coins are objects that are removed upon collision. This implementation also used the VGA component at its core as well as the “coin\_on” signal. This signal was previously explained in Section A of Implementation as the “on” signal for the component.

Like the pipes, a signal is passed to the FSM if the pixels of the bird and coin overlap. A flag was also used to make sure only a single signal was being passed to the FSM. This flag (“collision\_flag”) was used as a condition when determining if the coin should be visible on the display. This allowed the freedom of not displaying the coin any longer when the user collides with the coin.



*Figure 6: High Level Block Diagram*

### III. DESIGN DECISION AND TRADE-OFFS

### A. Screen Edge Handling

The obstacles in the game do not disappear smoothly off the left edge of the screen. This is because we have chosen to use an unsigned bit for the x-positions of objects with motion, such as the pipes and coins. If we were to employ a signed bit, we would have the option to check for negative numbers to represent the area beyond the screen edge. However, a decision was made not to implement it as signed bits are more resource-intensive than unsigned bits. Additional logic is also required to handle the signed bits during arithmetic operations. The lack of smooth disappearance of objects off the screen edge was deemed non-critical to the overall game-play experience. Thus, the decision to use unsigned bits simplified the code logic.

### B. Linear Feedback Register

Our design used an 8-bit Galois-type LFSR to generate pseudo-random numbers for the y-positions of the pipe gaps and coin heights.

We have chosen to implement the Galois-type LSFR, where the taps are placed between the registers, instead of the Fibonacci-type LSFR, where the taps are placed within feedback paths. This is because the additional logic elements needed on the feedback path increase the design's maximum propagation delay, negatively impacting the maximum clock frequency. By choosing to use the Galois-type LSFR, we can minimize any propagation delay,

allowing for higher clock frequencies and avoiding performance constrictions. Our LSFR follows the lecture example, where taps are placed at 1,2,3,7, which gives us a maximum sequence length of 255 bits.

We have chosen an 8-bit LSFR as it gives us 255 possible numbers for the y-position of the pipe gap height and coin heights. We thought this was appropriate as the screen only has a height of 480, which gives us a wide range of options for the object's height. Choosing a smaller bit size would limit the available range of heights, while choosing a larger one would introduce the risk of the heights exceeding the screen boundaries. To ensure the pipes and coins are positioned at appropriate heights for gameplay, we have applied suitable offsets to ensure this in our implementation.

### C. Graphics

Our game implements an updated MIF file and custom-created 2-D Arrays to display our characters and objects. Our alphabet and numerical characters are all implemented through the MIF file. It handles our hearts and clouds displayed in the game by customizing an 8 x 8 sized version of both components. Our Bird and Coin are both customized 2-D arrays that are defined in their respective files and then displayed according to the boundaries specified.

We decided to use these custom 2-D arrays due to a time restraint. Initially, we planned to map detailed sprites to enhance the game's visual appearance; however, due to unforeseen circumstances, only two members could work on the game. So, as a matter of priority, we decided to prioritize core game development and instead used custom 2-D arrays and updated the MIF file accordingly.

#### IV. RESOURCE USAGE AND PERFORMANCE

### A. Maximum Operation Frequency

The timing analysis report generated by Quartus in *Figure 7* shows the maximum clock frequency that a component can output within our project. The lowest clock frequency comes from our PLL component, which generates a clock frequency of 84.67MHz. We aimed to achieve a 25MHz signal from the PLL to drive the 60Hz refresh rate for the 640x480 VGA monitor. This is comfortably within the established limitation.

Table of Contents

- Flow Summary
- Flow Settings
- Flow Non-Default Global Settings
- Flow Elapsed Time
- Flow OS Summary
- Flow Log
- Analysis & Synthesis
- Fitter
  - Flow Messages
  - Flow Suppressed Messages
- Assembler
- Timing Analyzer
  - Summary
  - Parallel Compilation
  - Clocks
- Slow 1100mV 85C Model
  - Fmax Summary
  - Timing Closure Recommendation
  - Setup Summary
  - Hold Summary
  - Recovery Summary
  - Removal Summary
  - Minimum Pulse Width Summary
  - Metastability Summary

Slow 1100mV 85C Model Fmax Summary

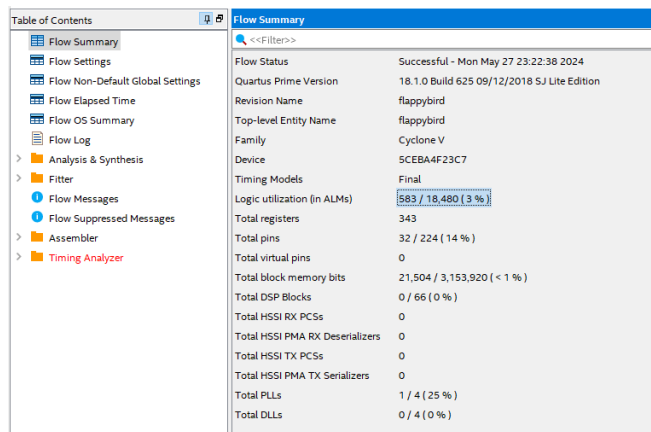
<<Filter>>		
Fmax	Restricted Fmax	Clock Name
1 84.67 MHz	84.67 MHz	inst3 pll_inst altera_pll general[0] gpll-PLL_OUTPUT_COUNTER divclk
2 128.73 MHz	128.73 MHz	VGA_SYNC inst1 vert_sync_out
3 267.67 MHz	267.67 MHz	MOUSE inst2 MOUSE_CLK_FILTER
4 273.82 MHz	273.82 MHz	display_obstacle inst8 pipes pipes2 fusr_clk
5 335.35 MHz	335.35 MHz	display_obstacle inst8 coin.shiny_coin fusr_clk
6 346.62 MHz	346.62 MHz	display_obstacle inst8 pipes pipes1 fusr_clk

Figure 7: Maximum Clock Frequency Report



## B. Logic Elements & Registers

The report generated by Quartus in Figure 8 shows the usage of our logic element in the project. We have used 583 logic elements and 343 registers in the gate-level description of the project. Within our design, we tried to minimize wasted resources by assigning and ensuring that the bit widths were allocated appropriate space. We also limited the integer sizes where possible. Some sections exhibit repetition and redundancy, such as files with similar logic or sharing signals. We may explore utilizing functions to refactor our code in the future to eliminate unnecessary repetition in code logic and shared signals. This will reduce the usage of our logic element overall within the project.



The screenshot shows the 'Flow Summary' window in Quartus. The left sidebar contains a 'Table of Contents' with various categories like Flow Summary, Flow Settings, Flow Non-Default Global Settings, Flow Elapsed Time, Flow OS Summary, Flow Log, Analysis & Synthesis, Fitter, Flow Messages, Flow Suppressed Messages, Assembler, and Timing Analyzer. The main panel displays the 'Flow Summary' table with the following data:

Flow Status	Successful - Mon May 27 23:22:36 2024
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	flappybird
Top-level Entity Name	flappybird
Family	Cyclone V
Device	5CEBA4F23C7
Timing Models	Final
Logic utilization (in ALMs)	583 / 18,480 ( 3 % )
Total registers	343
Total pins	32 / 224 ( 14 % )
Total virtual pins	0
Total block memory bits	21,504 / 3,153,920 ( < 1 % )
Total DSP Blocks	0 / 66 ( 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	1 / 4 ( 25 % )
Total DLLs	0 / 4 ( 0 % )

Figure 8: Quartus Resource Usage Report

## C. Future Improvements

We discovered a known bug in the game through testing.

Our bird is asymmetrical, meaning the outer square boundary defined can cause the player to “invisibly” crash into the obstacles. This happens in cases where the defined boundary of the bird that is not visible collides with any slight edges of the obstacles. This has not affected our gameplay primarily, as the player is more likely to collide with a larger area of their bird. This is an improvement we would like to make to allow the player to correctly visualize the boundaries of the bird instead of assuming a square boundary while playing.

An improvement we would like to make is implementing a pause feature in our game so the player can pause the game if needed. We would also like to increase the RGB channels to accommodate more colours in our game (currently only supports eight colours).

We would also like to incorporate a different variety of gifts. Currently, we offer a coin that increases the score by 2 points. However, this is not a very useful gift if the players are training. Therefore, we would like to add power-ups such as lives that the player can use both in training and normal mode, which allows the player an extra chance in the game, regardless of game mode.

## ACKNOWLEDGMENTS

We would like to give a big thank you to our lecturers Dr. Morteza Biglari-Abhari and Dr. Maryam Hemmati and also our teaching assistants Dhruv Joshi, Ehsan Behravan, Robert Sefaj and Ross Porter.

## REFERENCES

The coursebook materials from COMPSYS 305 by Dr. Morteza Biglari-Abhari and Dr. Maryam Hemmati