GROUP 25

Digital Systems Design
Mini project Report

Jonathan Browne - jbro682
Riley Fromont - rfro377

Abstract

This report details our design of a Flappy Bird styled game coded in VHDL and implemented on an FPGA development board. The game plays similarly to the original Flappy Bird where the player must constantly adjust the vertical position of their spaceship through 'bouncing' up to avoid colliding with obstacles or falling below the bottom of the screen. The game is based around four states/screens, these include the: Main Menu, Training, Normal and Game Over screens. The main menu allows the player to select a game mode and start a game, the training mode is a simpler easier game mode that does not increase in difficulty, the normal mode is the full game with increased difficulty and a score system and the game over screen is displayed when the user dies and shows the user their overall score. The system is designed in separate blocks which simplifies the code making more maintainable and improves performance. A majority of the system resources are used by the ship and text blocks which manage the fuel, ship movement and collision logic and text/character positioning. Overall, the system could be improved with better variable/signal types saving registers and additional features could be added or existing features improved to make gameplay more enjoyable and engaging.

Introduction

We were tasked to design our own version of the Flappy Bird game and implement it using VHDL on an FPGA development board. The game is played on a screen through a VGA connection and takes user inputs from a PS/2 Mouse and switches and push-buttons on the development board. The game is based on a constant horizontal scrolling motion where the player can adjust their vertical position to avoid obstacles and pick-up collectibles that move across the screen.

Additional Game Features:

We chose to implement a Linear Feedback Shift Register to act as a pseudo-random number generator for object and obstacles heights.

Custom background and spaceship image using MIF files and the DE0 board Read-Only Memory.

An acceleration system that constantly increases the downwards speed by a constant acceleration value to model the effects of gravity. By clicking the players provides an upwards speed that is then subject to this 'deceleration' giving the 'bouncing' or parabolic gravitational effect

Game Rules and Features:

Colliding with an 'Asteroid' or falling below the bottom of the screen results in instant game over; this differs from the specified life-system in that you effectively only have one life. We decided to do this to increase the overall difficulty of the game.

The player clicks jump upwards a small amount, the player can click multiple times or hold left-click to jump upwards further.
The player can only jump whilst their fuel is not empty (equal to 0), fuel can be replenished with fuel pick-up boxes.

Score is incremented by 1 for each gap passed through, additional score can be gained from score pick-up boxes; this gives the player the opportunity to take the risk of hitting an obstacle to gain more score from a pick-up.

The game starts in the main menu where the player can select between two game modes: Training and Normal, normal includes all previous mentioned features. In training the player can still die however the difficulty (scroll speed) does not increase and there is no fuel or score.

This gives the player the opportunity to practice with controlling the spaceship and passing through gaps without worrying about fuel or the game getting more difficult.

When the player loses they are transitioned to a Game Over screen where they can see their score, they can transition back from this screen to the main menu and play another game in either mode.

User I/O:

Mouse: Left-click will provide a small acceleration upwards (a ‘bounce’).

Push Button: Button2 allows the player to start the game or get back to the main menu from game over.

VGA display: The VGA display is used to display the game on the screen. This is done by passing the RGB values to the associated pins.

DIP Switch: Switch0 can be used to select which gamemode the player wants to play (‘Up or 1’ for training, ‘Down or 0’ for normal).

How To Play:

The game plays like Flappy Bird where the player controls a spaceship and must keep it from falling out of the map whilst simultaneously avoiding hitting Asteroids (pipes) by navigating through gaps between them.

The player can see their fuel in the form of a bar at the top of the screen that constantly decreases as the game is played. Fuel pick-ups appear as red boxes and the player must collide with these to refill the bar.

The primary aim of the game is to get the highest score; this is done by passing through as many gaps as possible where each gap passed through will award 1 point. In addition to this there are randomly generated score pick-up boxes that will award 2 points if the player collides with them. As the game progresses it gets more and more difficult as the speed that the spaceship moves across the screen increases (the asteroids/pipes move faster to the left: this is based on the number of pipes the player has travelled through

System FSM:

Outputs:
MENU: State = “00”
TRAINING: State = “01”
NORMAL: State = “10”
GAMEOVER: State = “11”

Transitions:
MENU => TRAINING If PB0 = ‘0’ and SW0 = ‘1’
MENU => NORMAL If PB0 = ‘0’ and SW0 = ‘0’
MENU => MENU If PB0 = ‘1’
TRAINING => GAMEOVER If Ship_Alive = 0
TRAINING => TRAINING If Ship_Alive = 1
NORMAL => GAMEOVER If Ship_Alive = 0
NORMAL => NORMAL If Ship_Alive = 1
GAMEOVER => MENU If PB0 = ‘0’
GAMEOVER => GAMEOVER If PB0 = ‘1’

The system FSM only outputs a signal representing the current State to other system blocks, these blocks then use this signal to determine which operations/logic to perform. The button is active low, so ‘0’ indicates it has been pressed.

Note: A diagram of the system FSM is included in the appendices.

Implemented System Blocks:

FSM - System control block to determine what state the game is in, takes in several status signals from other blocks as well as inputs from the user-input devices (button and switch).

Display - The primary display module for setting rgb values and determining what to display at a particular position on the screen.

Pipe - Controls the movement and position of an obstacle that scrolls across the screen; two of these blocks are implemented for the two pipes that scroll across the screen.

Objects - Controls the movement and position of the pick-up/collectible boxes that scroll across the screen: this module determines when the place these objects based on fuel (for fuel pick-up) and a random number (for score pick-up).

Ship - Controls the movement and position of the spaceship as well as whether the player has collided with an obstacle or pick-up; this module outputs signals indicating whether the ship is alive or not and if the player has collided with a pick-up. Also determines the players score based off pipe positions and score box collisions.

Text - Determines where to position lines of text on the screen, contains numerous loops that determine if the current pixel column/row is within a character position and which character it is and outputs a signal to the Display module when a pixel should be a character pixel.

SevenSegConverter - Converts the players score to a seven segment representations which is outputted to the board seven segment displays.

LFSR - Generates a sequence of 'semi-random' numbers, several of these blocks are used to randomise pipe Y-positions and score/fuel box positions.

Background_rom - Used to read the background mif file into memory as well as to read pixel values to be able to put the background image on the screen.

Ship_rom - Used to read the ship mif file into memory as well as read pixel values to be able to put the ship image on the screen.

Note: A High-Level block diagram of the system is included in the appendices.

Resource Consumption:

- Total Logic Element Usage: 2051
- Total Combinational Functions: 1994
- Logic Registers: 590
- Used Pins: 47
- Memory Usage: 246,786 bits
- PLLs: 1

Most combination functions were used in the Text and Ship blocks as these blocks contained most of the game logic. The text block contained numerous loops to iterate for every text line to determine individual character positions and which character to select; this required 577 of the total combinational functions or around 30%. The ship block contained several nested if statements for collision logic as well as the score incrementation and fuel management logic; this required 475 of the total combinations functions or around 24%.

Logic registers were used throughout the design however the ship and objects blocks used 288 of the total 590 used. This was because these blocks contained a significant number of position and status variables requiring registers for storage, as well as this there were several variables that used 32-bit integers that could have been substituted for smaller vectors to save on register usage.

All the memory usage can be attributed to the mif files that were used for the background/ship images and the characters. In total 48% of the development boards memory was used with about 93% of this being the background image; this is because the background is a 160 x 120 pixel image where each pixel is represented by a 12-bit RGB value using a significant amount of memory.

Timing analysis:

- Maximum Frequency: 95.95 MHz
- Critical Path: 10.4 ns
- Operating Frequency: 25 MHz

With a maximum operating frequency of 95.95 MHz the longest datapath in the system (critical path) is 10.4 ns (this would include setup and hold times). However, given we are using a VGA to display the game on the screen the system needs to operate at 25 MHz (this is the standard for VGA signals at 60 Hz). The operating frequency (of 25 MHz) is derived from a system PLL that is set to divide the FPGA development boards 50 MHz clock by 2 to give the required 25 MHz clock.

Conclusions:

Overall, the game has been designed to be similar to Flappy Bird in both design and complexity. The game is based around an FSM with four states, these being: MENU, TRAINING, NORMAL and GAMEOVER, where the user uses a DIP switch to determine which game mode to play and the button to start the game or to transition from game over back to the main menu.

The primary aim of the game is to get the highest possible score which can only be done in normal mode, as this is where all the game features are present and getting higher scores is more challenging and engaging. In comparison to normal mode training mode is a simplified/easier version where the game difficulty doesn't increase over the play-time and there is no fuel.

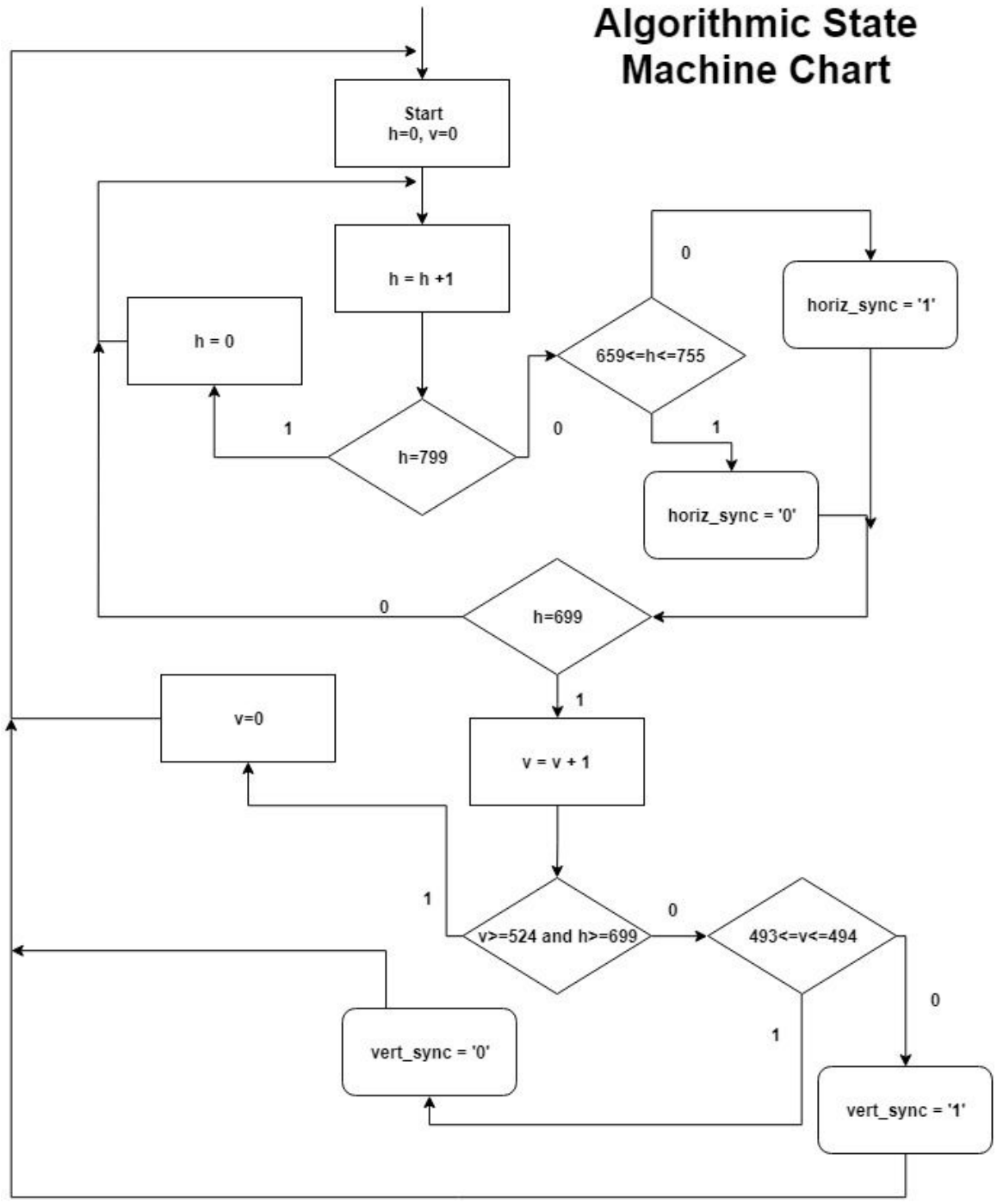
For system design, components and game logic was separated out into a number of blocks to both simplify the VHDL coding process as well as improve system performance. A majority of the systems resources were used by two of these blocks; Ship and Text, which were the two blocks that contained the most logic, variables and signals. To improve resource usage and performance we would optimise variable types to be as small as possible as there are some cases where we are using integers or 10-bit vectors where we could use 2 or 4 bit vectors. Overall however our designs maximum operating frequency, of 96 MHz, is significantly greater than our set operating frequency of 25 MHz meaning our system can deal with delays.

Given the limited time scale of the project we were not able to refine or implement all the features in the game. If we had additional time we would add more variety in the form of alternate power-ups to the game, improve the obstacles with less pipe-like obstacles and more random asteroid themed obstacles. We would also improve the difficulty/speed system to not be entirely based on travelling through pipes and Improve the games graphics by refining the existing images and mif files as well introduce a texture for the pipes.

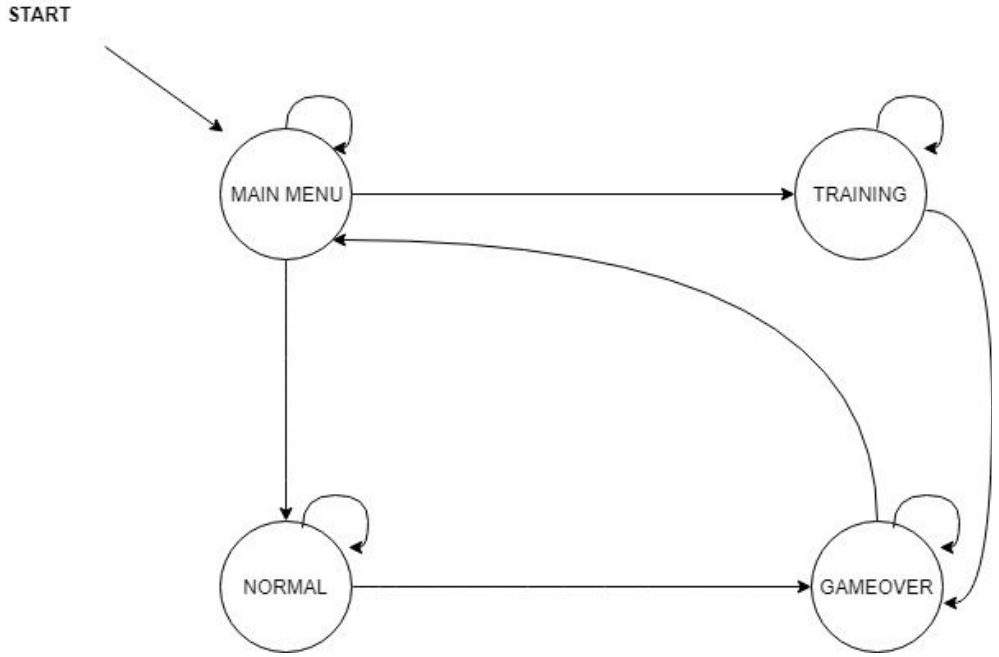
References:

-Mif file creation: https://stackoverflow.com/questions/29862161/extracting-memory-initialization-file-mif-from-a-bmp-photo
-Graphic Assets: https://opengameart.org/content/32-x-32-spaceships By BizmasterStudios
-DE0 User Manual v 1.3 Altera University Program (terasic.com)
-Quartus Timing Analyzer Quick-Start Tutorial: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_tq_tutorial.pdf

Appendix


Algorithmic State Machine Chart
Start
h=0, v=0
h = h +1
h = 0
h=799
659<=h<=755
horiz_sync = '1'
horiz_sync = '0'
h=699
v=0
v = v + 1
v>=524 and h>=699
493<=v<=494
vert_sync = '0'
vert_sync = '1'
0
1

System FSM



System block diagram