ECE241
Road to Heaven

1.0 Introduction

The game *"Road to Heaven"* is realized by successful integration of Verilog coding, FPGA board, VGA display, sound sensor and pixel graphic design. Below is a brief description of the game.

A character is standing at the right side of the screen, and cars are coming from the left side of the screen towards the character. In order to continue the game, the player must ensure that as soon as the car is about to hit the character, the player makes a loud enough sound to be received by the sound sensor, and thus instruct the character to make a jump. Else the character is going to be hit by the car and the game will be over.

The motivation of the game is to provide an easy, non-addicting platform for stressed engineering students to relieve stress. The graphic design of the game gives out a very cute and funny vibe.

2.0 Design

The design is discussed in detail in this section.

2.1 Top-Module (Verilog code shown in Appendix A)

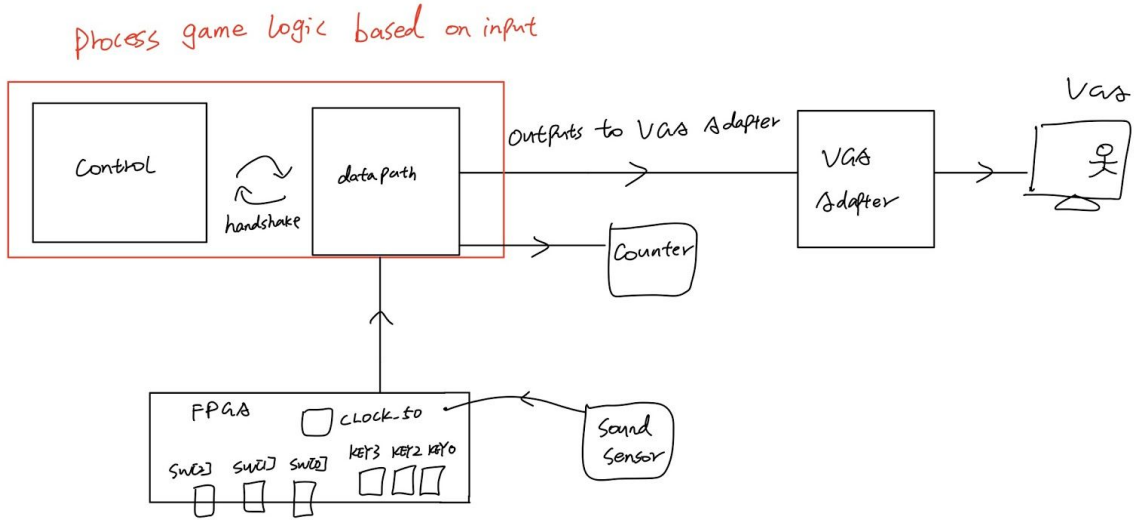

Figure 1: Top module block diagram

The player could:

1. Select difficulty using SW[2:0] (it can be changed any time during the game)
2. Press KEY[0] to start the game
3. Press KEY[2] to restart the game when game is over (the car hits the character)
4. Shout "jump" to the sound sensor to control the character

The above inputs along with CLOCK_50 are sent to the Game Logic Processor, as indicated by the red box in figure 1, to control the game. The Processor then outputs the corresponding image to the VGA adapter every 1/60 second, which then plots it to the screen(colour & x,y position are produced by datapath). There is also a counter that will

increment on each successful jump. If game is over, the counter will be reset to 0. The purpose of the counter is to record the score of the player.

2.2 Control (Verilog code shown in Appendix B)

The Control module gives “orders” to the datapath. It consists of 9 stages. The content of “order” is different in each stage. In general, the transition between stages happens when Control receives a “done” signal from datapath with one exceptional case. Below is a brief explanation of the stages and the orders they give:

Stage	Output Signal	Transition Condition	Explanation
reset	None	When KEY[0] (game_start) or KEY[2](restart) is pressed	Initial stage: set all output signals to 0 and wait for user input
drawBackground	go_draw_background	When it is done	Draw the game background
drawCar	go_draw_car	When it is done	Draw the car image from the “mif” file
drawPerson	go_draw_person	When it is done	Draw the person image from the “mif” file
waitDraw	go_divide	When it is done	Make the rate divider count down from a specific number, as set by SW[2:0]
clearScreen	go_draw_background	When it is done	Draw the game background (to erase car and person images)
updateCar	go_update_car	If “enable” is 1, it will transit to “updatePerson”. Otherwise it will transit to “drawCar”	Update the car location (to make it seem as moving)
updatePerson	go_update_person	None	Update the person location (to make it seem as jumping)
gameOver	go_draw_gameOver	When KEY[2] is pressed (the user decides to restart the game)	**NOTE: this stage will only be visited if game is over** Draw the gameOver image from the “mif” file

2.3 Datapath (Verilog code shown in Appendix C)
The datapath consists of 7 different modules. Each of them will be discussed below.

(1) drawBackground
This module works only if it receives the “go_drawBackground” or “go_clear_screen” signal. At each positive clock edge, it will increment x (max = 160) and y (max = 120), which is the pixel location to plot on the screen. Then it inputs the x,y locations into an address translator (a sample of address translator can be found in Appendix D) to produce an address. Then it will read from the background.mif at the specific address to get the colour we want to output (a ROM that is initialized with background.mif is required). When the process is done, a “done_drawBackground” signal is outputted to the Control module to indicate that a state transition can now be made.

(2) drawCar
This module works only if it receives the “go_drawCar” signal.
This module is very similar to “drawBackground” module except that

(a) The output “x” is added with “xShift” from “updateCar” module to make the car seem as moving. It is then subtracted by 40 to make the car drawn from outside of the left end of screen.
(b) The limit of “xAddress” is 40 and the limit of “yAddress” is determined by “carType” from “updateCar” module, since different cars have different height.
(c) The output colour is also determined by “carType”.
(d) For each type of car, a unique “yShift” is assigned to shift the car to the road on the background image.

(3) drawPerson
This module works only if it receives the “go_draw_person” signal.
This module is very similar to “drawBackground” module except that

(a) The output “x” is added with 120 to make the person standing at the right end of the road. The output “y” is added with “yShift” from “updatePerson” module to make the person seem as jumping.
(b) There are two sprites regarding the person: jumping and standing. When yShift is close to 50 (close to the road), colour from “stand.mif” would be output. In all other cases colour from “jump.mif” would be output.

(4) waitDraw
This module works only if it receives the “go_divide” signal.
At every positive clock edge, Q will count down 1 unit from the defined limit (limit is defined by difficulty, which is SW[2:0]). When Q reaches 0, a “done_divide” signal will be output and Q will be reset to limit.

(5) updatePerson
This module works only if it receives the “go_update_person” signal.
“y” is initially set to yMax (on the road). At each positive edge of clock, y is subtracted by one. When it reaches yMin (top of screen), it counts back up to yMax.
Our goal is to make the person jump from the road, reaches the top and falls back down to the road when the player shouts “jump”. So the x,y locations of the person have to be updated multiple times during the process. In order to achieve this goal, I declared an output register named “enable”, which is set to 0 initially. If the user shouts “jump”, “enable” will be set to 1. When the counter counted a full cycle (count down to yMin and count back up to

yMax), “enable” will be set back to 0. This signal will determine whether the transition should happen from “updateCar” to “updatePerson” or from “updateCar” to “drawCar”.

(6) updateCar
This module works only if it receives the “go_update_car” signal.
“x” is initially set to xMin (left end of screen). At each positive edge of clock, x is incremented by 1. When it reaches xMax, it is set back to xMin. Also, “carType” got incremented, indicating to “drawCar” module that a different car should now be drawn.

(7) drawGameOver
This module works only if it receives the “go_draw_gameOver” signal.
This module works identically the same as “drawBackground” module. The only exception is that the colour is output from “gameover.mif” instead of “background.mif”.

2.4 Game Over Condition
The game is regarded as over when the x position of car equals x position of person, and y position of car equals y position of person.

3.0 Report on Success

The overall game is successful since it is highly playable and amusing. But there was one problem throughout the project - the flashing issue of the images on VGA. Originally I used a resolution of 640x480, but I found out that the image is flashing so badly and it ruined the playability. One possible reason is that my code requires the FPGA to do a lot of drawing, but it failed to do so due to hardware limitations. If I wanted to fix this problem I would have to completely change my FSMs and therefore restart the project from the beginning. Since I do not have the time, I changed the resolution to 160x120 and hoped that it will fix the problem because the FPGA now could draw less. The result was both comforting and frustrating - the image was still flashing, but to a much lesser content.

4.0 What would I do differently

If I were to start the project all over again, I would use a completely different logic when I am implementing my control and datapath modules. Instead of drawing the entire image at one stage, it would just draw 1 pixel (so the drawing of the entire image requires thousands of cycles). Instead of drawing the images first and then drawing the background (serves as an eraser), it will draw only once, using the correct colour (car, person or background). I believe this way I could make the FPGA draw to a minimum content, and therefore fix the flashing problem.