Computational Logic – Final Project – Master the Pipes

Brian Fu 付博 3170300210

Richard Dong 董传琪 3170300264

Parsa Alamzadeh 帕萨 3170300180 (Leader)

Hansoo Yoon 3170300179

1) Introduction

In Master the pipes, we use a screen and keyboard to emulate the simple arcade games of old, creating a “rotate the blocks” puzzle style game, where the point is to rotate various tubes to create viable path through the “maze”.

This is best demonstrated by the “PicRoad” flash game, except instead of trying to create a cycle like they have, we merely try and create a path for the water to travel from one point to another.

We used the Xilinx ISE, Verilog, the SWORD board (containing a Kintex 7 FPGA), and may optionally also use a VGA monitor to create and run our project).

2) Functional Overview

For selection of the blocks in the game we used an external keyboard. Using WASD user can choose the selection area and then by pressing the space button the user can rotate the blocks. Rotations happen in a clockwise fashion meaning that if you have a 90 degrees pipe which connects the incoming pipe from left and connecting it to the bottom block by pressing space you rotate the pipe clockwise, meaning that the new state would connect the top module/pipe to the right module/pipe. Here is a demonstration of the rotation. Given the left state, when space is pressed it turns into the right state.

The screen is divided into 24 sections, the inner 4 by 4 grid contains our pipe/module and the top left contains our starting point and bottom right contains our ending point. User’s job is to connect these two points as if there is water running through the pipes.

The general view on our display, which can be any 640 x 480 VGA connected screen, comprises of a grid of tubes, as shown below:

Upon successfully making a path from the “Start” indicator to the “End” indicator, the screen will display the winning screen shown as below:

3) Structural Design

In our game, like mentioned in our introduction, the goal is to connect a series of pipes in order to allow a theoretical flow of water to travel from “Start” to “Finish”.

To do this, for each box we take an 80 by 80 image, and convert it into RGB format, with each color taking 4 bits, resulting in a 12 bits output for each pixel; This entails that we use a 80 x 80 = 6400 bit array, such that each element is 12 bits long, with each element being 1 pixel of the image.

We only need to convert the images for all the sorts of tubes we use, of which there are only a small set amount, detailed below:

(The last image represents a blocked module meaning that there is no input and output)

In total, there are 4 different tube styles, with 2 orientations for the horizonal pipe, 4 orientations for the 90 degrees one and the cross and the blocked on do not change since their other orientations are the same as their initial state.

Since the FPGA on SWORD requires 4 bits for each color we had to convert each pixel’s RGB value into a 12 bits representation. To do so, we used a simple python script which calculated square root of each pixel’s color’s value and took its floor value and then converted into a hexadecimal value then repeated that process for the other values and eventually we had 3 hexadecimal values. Then we stitched them together and formed a 12bits value and then recorded that value for each pixel. The code used for the conversion is shown as below:

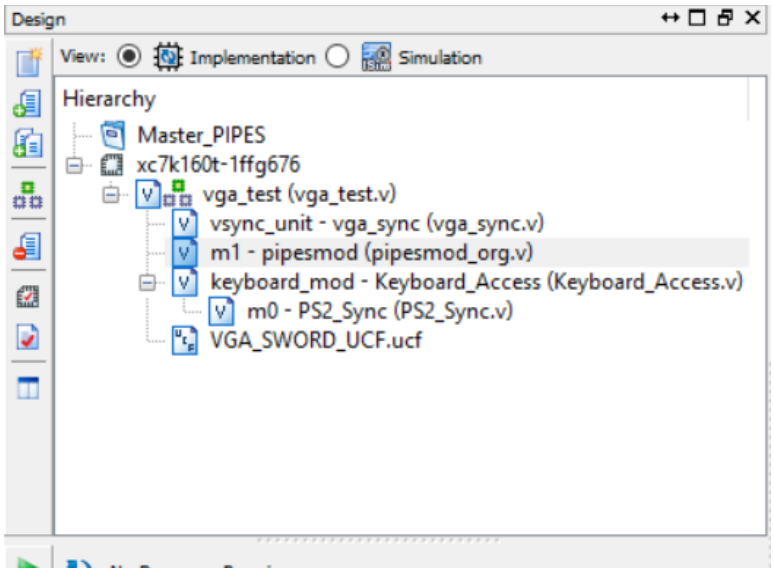
```
import numpy
import cv2 as cv
from math import floor
```

```
import glob
j = 0
k = 0
for images in glob.glob("folder/*.png"):
    print(images)
    a = cv.imread(images)
    R = a[:,:,0].flatten()
    G = a[:,:,1].flatten()
    B = a[:,:,2].flatten()
    for i in range (len(R)):
        R[i] = floor(R[i] ** (0.5))
        G[i] = floor(G[i] ** (0.5))
        B[i] = floor(B[i] ** (0.5))
    val = []
    for i in range (len(R)):
        val.append("12'h"+str(hex(R[i]).split('x')[-1])+str(hex(G[i]).split('x')[-1])+str(hex(B[i]).split('x')[-1]))
    for i in range (len(val)):
        with open("folder/"+str(images.split("/")[1].split(".")[0])+".txt", "a") as openfile:
            openfile.write(val[i])
            openfile.write(", ")
            if (i % 80==0 and i != 0):
                openfile.write("\n")

            j+=1
```

For our game platform, there are 16 boxes configured in a 4 by 4 grid in total on the display (that are manipulatable), each with one of the above tubes contained within, surrounded by whitespace and the “Start” & ”END” signs.

4) Detailed Description

Project Structure:



Top Module "VGA_test":

Top module VGA_test contains all of the modules used for this game. Each of the modules is described individually later in this report. The main purpose of this top module is to draw the game and get the inputs and pixel values from other modules and output the pixel values to the FPGA. This module contains vsync_unit, a module to sync the x and y values for vga, pipesmod which gets the RGB value of each pixel according to the location of the "electron gun" or our current pixel, keyboard_mod which gets the inputs from the keyboard. Our first series of if statements decides the selection, which is used for rotation of the pipes. KEY_BUTTON[0], KEY_BUTTON[1], KEY_BUTTON[2], KEY_BUTTON[3] and KEY_BUTTON[4] are indicating whether W, S, A, D and space are pressed respectfully. If any of the selection buttons are pressed then we move the selected grid accordingly, then if space button is pressed according to current state of the pipe we perform a rotation. For example, if the current selected module has a state of 0 then if space is pressed we change its state to 1. We have a register which stores the state of each block and we change it according to the actions of the user if needed. Then the next series of if statements take care of the mode selection which is fed into the pipesmod which outputs the appropriate pixel value. Once the user achieves the intended states for each block the user wins and then we update the screen with the winning screen which again is outputted by the pipesmod.

```
module vga_test(
    output wire[7:0]Segment,
   output wire [3:0]AN,
    input clk,
    input reset_n,
    output vga_h_sync,
    output vga_v_sync,
    output [3:0] vga_blue,
   output [3:0] vga_green,
   output [3:0] vga_red,
  inout [4:0]BTN_X,
  inout [3:0]BTN_Y,
  input [15:0]SW,

  //input CLK,    //board clock signal
  input PS2_CLK,    //keyboard clock signal
  input PS2_DATA    //keyboard data signal
   );



  reg [15:0] buttons;
  reg [3:0] red, blue, green;
  wire video_on;
  wire [9:0] x_loc, y_loc;
  reg [9:0] p_x, p_y;
  reg [31:0] clk_div;
  wire [11:0]out;
  always @(posedge clk) clk_div = clk_div + 1;
    vga_sync vsync_unit
    (.vga_clk(clk_div[1]), .reset(1'b0), .hsync(vga_h_sync), .vsync(vga_v_sync),
    .video_on(video_on),
    .pixel_x(x_loc), .pixel_y(y_loc));

  reg [3:0] mode;


  pipesmod m1 (p_x, p_y,mode, out);

//  reg [3:0]states[0:17] = {72'b0};
  reg [3:0]states[0:17] = {4'd8, 4'd0, 4'd7, 4'd1, 4'd3, 4'd4, 4'd6, 4'd1, 4'd2, 4'd0, 4'd2, 4'd7,
  4'd1, 4'd7, 4'd6, 4'd3, 4'd1, 4'd9};
  reg wasReady;

  wire [71:0]out_state;
  reg [3:0] location = 4'b0;
  wire [5:0] KEY_BUTTON;

  Keyboard_Access keyboard_mod (PS2_DATA,
    PS2_CLK,
    KEY_BUTTON,
    clk
  );



    always @(posedge clk)begin
    if (KEY_BUTTON[0]) location <= location - 4;
    if (KEY_BUTTON[1]) location <= location + 4;
    if (KEY_BUTTON[2] && location % 4 != 0) location <= location - 1;
    if (KEY_BUTTON[3] && location % 4 != 3) location <= location + 1;
    if (KEY_BUTTON[4])
        case (states[location+1])
          4'b0000: states[location+1] <= 4'b0001;
          4'b0001: states[location+1] <= 4'b0000;
          4'b0010: states[location+1] <= 4'b0101;
          4'b0011: states[location+1] <= 4'b0100;
          4'b0100: states[location+1] <= 4'b0010;
          4'b0101: states[location+1] <= 4'b0011;
```

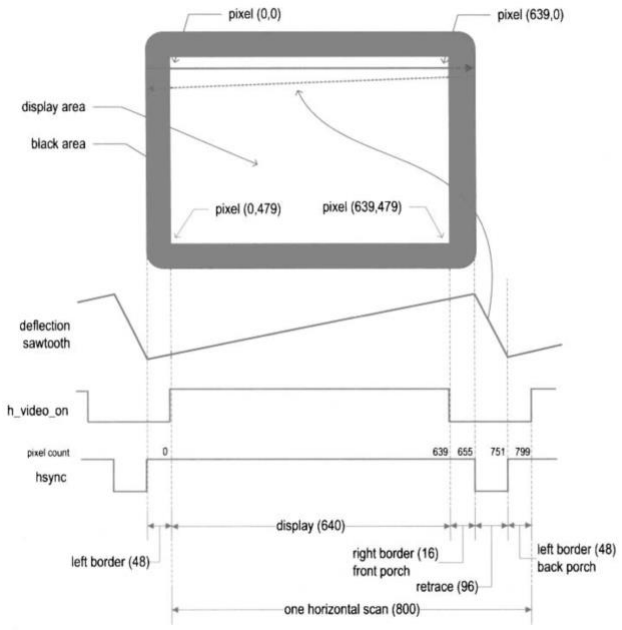
```
    endcase
    p_x <= x_loc;
    p_y <= y_loc;
    if (states[1] == 4'b0001 && states[5] == 4'b0011 && states[7] == 4'b0 && states[8] == 4'b0010
      && states[12] == 4'b0001 && states[16] == 4'b0001) // winning state
      mode <= 4'd13;
    else if (p_y < 80 && p_x > 160 && p_x<240 )
      mode <= states[0];
    else if (p_y > 400 && (p_x > 400 && p_x < 480) )
      mode <= states[17];
    else if (p_y >80 && p_y<400 && p_x>160 && p_x < 480) begin
      if (p_y > 80 && p_y < 160)begin // Y
        if (p_x > 160 && p_x < 240) mode <= states[1];
        else if (p_x> 240 && p_x < 320) mode <= states[2];
        else if (p_x > 320 && p_x < 400) mode <= states[3];
        else if (p_x> 400 && p_x < 480) mode<= states[4];
      end else if (p_y > 160 && p_y < 240) begin //Y
        if(p_x > 160 && p_x < 240) mode <= states[5];
        else if (p_x> 240 && p_x < 320) mode<= states[6];
        else if (p_x > 320 && p_x < 400)mode<= states[7];
        else if (p_x > 400 && p_x < 480)mode<= states[8];

    end else if (p_y > 240 && p_y < 320) begin //Y
        if(p_x > 160 && p_x < 240) mode <= states[9];
        else if (p_x> 240 && p_x < 320)mode <= states[10];
        else if (p_x > 320 && p_x < 400)mode<= states[11];
        else if (p_x > 400 && p_x < 480)mode<= states[12];
      end else if (p_y > 320 && p_y < 400) begin //Y
        if(p_x > 160 && p_x < 240)mode <= states[13];
        else if (p_x> 240 && p_x < 320)mode <= states[14];
        else if (p_x > 320 && p_x < 400)mode<= states[15];
        else if (p_x > 400 && p_x < 480)mode<= states[16];
      end
    end
    else
//      if (p_x<160) mode<=4'd11;
//      else if (p_x>480) mode<=4'd12;
//      else
        mode <= 4'd10; // blank


  end



  assign vga_red = (video_on) ? out[11:8] : 4'b0;
  assign vga_blue = (video_on) ? out[3:0] : 4'b0;
  assign vga_green = (video_on) ? out[7:4] : 4'b0;
endmodule
```

VGA_sync:

This module is in charge of syncing the screen. The inputs are vga_clk, reset and the outputs are hsync, vsync, video_on, pixel_x and pixel_y. As it was described in the textbook, the screen has borders on the left and right as well as top and bottom. The reason we have a 25MHz clock as an input is that we have a height of 800 and width of 525 (including the borders and retracing) and since we want to have a 60Hz refresh rate (optimal for human's eyes) we have the following:

$$800 \times 525 \times 60 = 25{,}440{,}000$$

This means for every second we need to process 25,440,000 pixels per second, and our 25MHz clock approximates that. Instead of maintaining the 60Hz refresh rate per second, we have 59.52Hz which is fairly close and undistinguishable for our eyes. Then we have hsync and vsync which are our outputs. The hsync is our output signal for this module and its job is to be *high* when the “electron gun” is not retracing. Same thing applies to the vsync. Then we have video_on, which takes care of the viewable region of the screen. This means, while the “electron gun” is within the range of 48 and 688 for the x value and for y value which its within 33 and 518 it is high, else it’s low. The value of 48 and 33 correspond to the left and up borders respectfully. The following diagram show this clearly for the x signal:



Then we use this to define or x and y coordinates on the display. This is later used for our pixel values to be displayed on the screen. Below is our VGA_Sync module:

```
module vga_sync(
    input vga_clk, // 25 MHZ
    input reset,
    output reg hsync,
    output reg vsync,
    output video_on,
    output reg [9:0] pixel_x,
    output reg [9:0] pixel_y
    );

  // sync counters
  reg [9:0] h_count;
  reg [9:0] v_count;
  wire [9:0] row;
  wire [9:0] col;
  // output buffer
  wire v_sync, h_sync;
  // status
  wire h_end, v_end, pixel_tick;

  always @(posedge vga_clk) begin
    if (reset) h_count <= 10'h0;
    else if(h_count == 10'd799)
      h_count <= 10'h0;
    else h_count <= h_count + 10'h1;

    pixel_x = col;
    pixel_y = row;
    hsync = h_sync;
    vsync = v_sync;
  end

  always @(posedge vga_clk or posedge reset) begin
    if(reset) v_count <= 10'h0;
    else if (h_count == 10'd799) begin
      if(v_count == 10'd524) v_count <= 10'h0;
      else v_count <= v_count + 10'h1;
    end
  end
  // Y value needs to lose 7
  // X lose 11
  assign row = v_count - 10'd35; // pixel y value
  assign col = h_count - 10'd143; // pixel x value
  assign h_sync = (h_count > 10'd95); // 96->799, is enabled
  assign v_sync = (v_count > 10'd1); // 2->524
  assign video_on = (
    h_count > 10'd142 &&
    h_count < 10'd783 && // 640 pix
    v_count > 10'd34 &&
    v_count < 10'd515); // 480 lins
endmodule
```

Pipesmod:

As we discussed previously the pipesmodule is where we grab our RGB values. This module has x, y and mod as inputs and out as an output. x and y are the coordinates of the pixel that is going to be drawn on the screen and mode describes which block is going to be drawn. Mode has 14 states, 8 reserved for pipes, 9 and 10 for starting point, 11 and 12 for backgrounds, 13 for winning screen and 14 for a blank white pixel. Once we have the x and y values and its mode, then we grab the appropriate pixel value which is calculated by:

$$6400 \times mode \times ((x\%80) + (y\%80) \times 80)$$

Since we stored all the values in a big array, this provides us with correct pixel value, because we have divided the whole screen to grids of 80 by 80 pixels. Once we have the appropriate pixel location we set the out wire to be the same value as our pixel value. If mode is 13 we use the array specified for the winning screen and apply the same logic.

```
21  module pipesmod(input wire [9:0] x,
22                  input wire [9:0] y,
23                  input wire [3:0]mode,  // 0-7 are pipes, 8 and 9 are the starting and ending points pixel val
24                  output wire [11:0]out
25                  );
26     reg [11:0]data[0:63999] = {
27  // pipe zero horizontal straight pipe
```

(RGB image bit data omitted)

```
4711     reg [11:0] pix_value; // the output pixel value according to the position and mode of the input
4712     reg [9:0]x_loc, y_loc; // x and y values
4713     reg [16:0]pix_val; // index of the pixel needed for the the output
4714     reg [11:0] D_in; // temporary reg
4715     always @(*)begin
4716        if (mode == 4'b1010)
4717           D_in<=12'hfff;
4718        else if (mode == 13)begin
4719           pix_val <= x + y * 640;
4720           D_in <= data[{pix_val}];
4721        end else begin
4722        pix_val <= 6400 * mode +((x%80) + ((y%80) * 80));
4723        D_in <= data[{pix_val}];
4724     end
4725  // else begin
4726  //    if (mode==11) begin
4727  //          pix_val <= 6400*2 * (mode%10-1) +((x%160) + ((y%80) * 80));
4728  //          D_in <= background[{pix_val}];
4729  //    end
4730  //    else begin
4731  //          pix_val <= 6400*2 * (mode%10-1) +((x%160) + ((y%80) * 80));
4732  //          D_in <= background[{pix_val}];
4733  //    end
4734  //end
4735     end
4736     assign out = D_in;
4737
4738  endmodule
```

Keyboard_Access:

Takes in keyboard presses and converts them to they can be used by our program as controls.

```
21  module Keyboard_Access(
22      input ps2d,
23      input ps2c,
24      output reg [5:0] KEY_BUTTON,
25      input clk
26  //  input [71:0] in_states,
27  //  output [71:0] out_states,
28      );
29
30     reg [7:0] num_clk;
31     reg new_clk;
32     always @(posedge clk)begin
33        if (num_clk != 8'hff)
34           num_clk <= num_clk + 1;
35        else
36           num_clk <= 8'h0;
37           new_clk = ~new_clk;
38     end
39
40     reg isArrow;
41     reg isRelease;
42     wire ready;
43     wire [7:0] dout;
44     wire [7:0] prev;
45     reg [7:0] key_code;
46     //wire data;
47     //assign data = ~isRelease;
48
49     initial begin
50        KEY_BUTTON = 6'b0;
51        isArrow = 1'b0;
52        isRelease = 1'b0;
53     end
```

```
53     end
54
55     PS2_Sync m0(
56        .ps2d(ps2d),
57        .ps2c(ps2c),
58        .clk(new_clk),
59  //    .ready(ready),
60        .dout(dout),
61        .pout(prev));
62
63     always @(posedge new_clk) begin
64        if(dout == 8'h1d) // W
65           if(prev == 8'hf0) // release code
66              KEY_BUTTON[0] = 1'b0;
67           else KEY_BUTTON[0] = 1'b1;
68        else if(dout == 8'h1b) // S
69           if(prev == 8'hf0) // release code
70              KEY_BUTTON[1] = 1'b0;
71           else KEY_BUTTON[1] = 1'b1;
72        else if(dout == 8'h1c) // A
73           if(prev == 8'hf0) // release code
74              KEY_BUTTON[2] = 1'b0;
75           else KEY_BUTTON[2] = 1'b1;
76        else if(dout == 8'h23) // D
77           if(prev == 8'hf0) // release code
78              KEY_BUTTON[3] = 1'b0;
79           else KEY_BUTTON[3] = 1'b1;
80        else if(dout == 8'h29) // SPACE
81           if(prev == 8'hf0) // release code
82              KEY_BUTTON[4] = 1'b0;
83           else KEY_BUTTON[4] = 1'b1;
84        else if(dout == 8'h2d) // R
85           if(prev == 8'hf0) // release code
86              KEY_BUTTON[5] = 1'b0;
```

```
73           if(prev == 8'hf0) // release code
74              KEY_BUTTON[2] = 1'b0;
75           else KEY_BUTTON[2] = 1'b1;
76        else if(dout == 8'h23) // D
77           if(prev == 8'hf0) // release code
78              KEY_BUTTON[3] = 1'b0;
79           else KEY_BUTTON[3] = 1'b1;
80        else if(dout == 8'h29) // SPACE
81           if(prev == 8'hf0) // release code
82              KEY_BUTTON[4] = 1'b0;
83           else KEY_BUTTON[4] = 1'b1;
84        else if(dout == 8'h2d) // R
85           if(prev == 8'hf0) // release code
86              KEY_BUTTON[5] = 1'b0;
87           else KEY_BUTTON[5] = 1'b1;
88     end
89
90
91  endmodule
```

The ps2 is a device that inputs data in a serial stream. This device has its own clock signal where the following edge of the clock indicates that the ps2 data is valid and can be retrieved. The transmission begins with a start bit, followed by 11 data bits. The timing of our clock is one-fourth of the boards clock, which is 12.5MHz.

In the keyboard module, we have a matrix of keys and an embedded microcontroller that monitors the inputs (when a key is pressed) and scans it accordingly. The keyboard consists of 3 types of keys. One, is the standard pressing down of a key, the make code of the key is transmitted. When a key is held down continuously, the make code is transmitted repeatedly. Lastly, when the key is released, the type break code of the key is transmitted.

Each key in of the keyboard has two hexadecimal values which represents its value, this code is represented by 8bits or 1 byte which is part of the packet transmitted by the keyboard along with its clock. These codes can be conveyed by one packet when transmitted. However, the extended keys (special purpose keys such as shift, back space) have 2 to 4 bytes. But in our project, we only used WASD and space keys. Now we know that we need a scan code to monitor the arrival of the received packets from the keyboard, and then display it. Upon pressing the key, the code keeps listening for packets from the keyboard and shifts our existing register. Suppose you press a key then our dout (key press) equals to h'1D (hexadecimal value of W) which is given encoded in the packet sent by the keyboard, but when you hold it, our prev will equal 8'hf0. This way the code can recognize when the user is hold a key or constantly pressing it by comparing the current state with its previous state. Once we know which key is pressed we set its corresponding value in Key_button to one meaning that that key was pressed. The keys are represented as W, S, A, D, space and R. WASD are used for moving our selected section of our 4X4 grid and space key is used for rotating our selected clockwise and R was originally planned to be used for reset button.

PS2_Sync:

Makes the keyboard inputs from Keyboard_Access we receive work with the clock.

Our Ps2 sync module primarily focuses on the timing of data transfer from the keyboard with PS2clock. As mentioned before, the negative edge of the clock is used as the reference point to retrieve data. The input data is stored in an 8-bit register that we use to hold. With each cycle of the clock we get the corresponding value and then set, just like bit shifting. We also have a filter reg which is used for filtering our input from the keyboard. Once we set the data in the ps2_sync we pass the data into Keyboard_Access where we process the inputs and then pass them into our top module.

```
20  //////////////////////////////////////////////////////////////////
21  module PS2_Sync(
22      input ps2d,
23      input ps2c,
24      input clk,
25      output [7:0] dout,
26      output [7:0] pout
27      );
28
29     reg [7:0] filter_reg;
30     reg ps2c_reg;
31     wire [7:0] filter_next;
32     reg [7:0] data, curr, prev;
33     reg [4:0] count;
34
35     assign dout = curr;
36     assign pout = prev;
37     assign filter_next = {filter_reg[6:0],ps2c};
38
39     initial begin
40        count = 4'b1;
41        filter_reg = 8'b0;
42        data = 8'b0;
43        prev = 8'b0;
44        //ready = 1'b0;
45     end
46
47
48     always @(posedge clk) begin
49        filter_reg = filter_next;
50        ps2c_reg = (filter_reg == 8'b0)? 1'b0 :
51           (filter_reg == 8'b1111_1111)? 1'b1 : ps2c_reg;
52     end
```

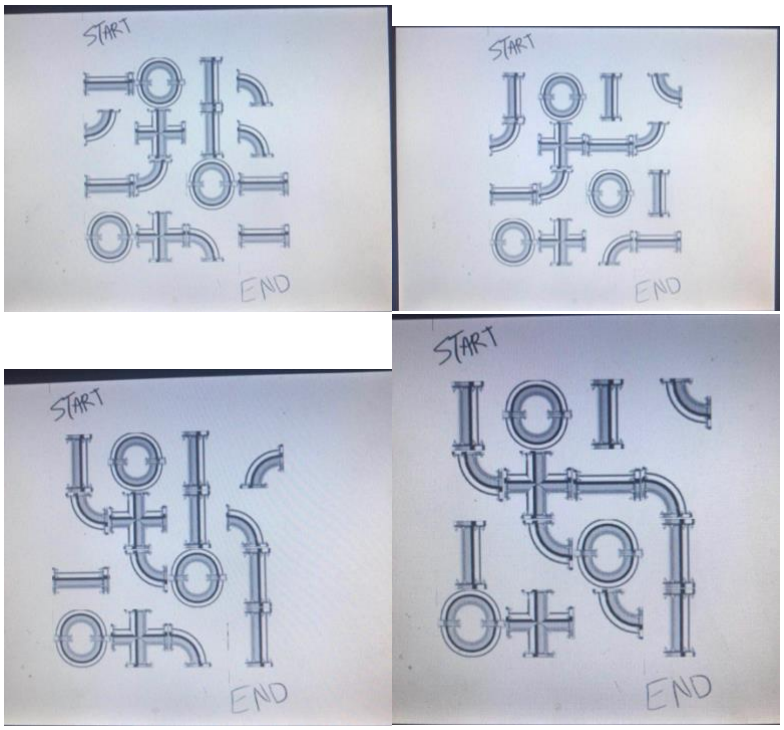
```
45     end
46
47
48     always @(posedge clk) begin
49        filter_reg = filter_next;
50        ps2c_reg = (filter_reg == 8'b0)? 1'b0 :
51           (filter_reg == 8'b1111_1111)? 1'b1 : ps2c_reg;
52     end
53
54     always @(negedge ps2c_reg) begin
55        case(count)
56           1: begin curr <= 8'h0; prev <= data; end//first bit
57           2: data[0] <= ps2d;
58           3: data[1] <= ps2d;
59           4: data[2] <= ps2d;
60           5: data[3] <= ps2d;
61           6: data[4] <= ps2d;
62           7: data[5] <= ps2d;
63           8: data[6] <= ps2d;
64           9: data[7] <= ps2d;
65           10: curr <= data; // Parity bit, we assume it's correct :3
66           11: ; // Ending bit, we prepare ready again
67        endcase
68        if (count < 4'd11) count <= count+1'b1;
69        else count <= 1'b1;
70     end
71
72  endmodule
73
```

Implementation Constraints:

```
NET "clk"                LOC = AC18      | IOSTANDARD = LVCMOS18 ;
NET "reset_n"              LOC = W13       | IOSTANDARD = LVCMOS18 ;


NET "PS2_DATA" LOC = M19 | IOSTANDARD = LVCMOS33 | PULLUP;
NET "PS2_CLK"  LOC = N18 | IOSTANDARD = LVCMOS33 | PULLUP;


NET "btn_x[0]"           LOC = V17       | IOSTANDARD = LVCMOS18 ;
NET "btn_x[1]"           LOC = W18       | IOSTANDARD = LVCMOS18 ;
NET "btn_x[2]"           LOC = W19       | IOSTANDARD = LVCMOS18 ;
NET "btn_x[3]"           LOC = W15       | IOSTANDARD = LVCMOS18 ;
NET "btn_x[4]"           LOC = W16       | IOSTANDARD = LVCMOS18 ;
NET "btn_y[0]"           LOC = V18       | IOSTANDARD = LVCMOS18 ;
NET "btn_y[1]"           LOC = V19       | IOSTANDARD = LVCMOS18 ;
NET "btn_y[2]"           LOC = V14       | IOSTANDARD = LVCMOS18 ;
NET "btn_y[3]"           LOC = W14       | IOSTANDARD = LVCMOS18 ;

NET "SW[0]"          LOC = AA10      | IOSTANDARD = LVCMOS15 ;
NET "SW[1]"          LOC = AB10      | IOSTANDARD = LVCMOS15 ;
NET "SW[2]"          LOC = AA13      | IOSTANDARD = LVCMOS15 ;
NET "SW[3]"          LOC = AA12      | IOSTANDARD = LVCMOS15 ;
NET "SW[4]"          LOC = Y13       | IOSTANDARD = LVCMOS15 ;
NET "SW[5]"          LOC = Y12       | IOSTANDARD = LVCMOS15 ;
NET "SW[6]"          LOC = AD11      | IOSTANDARD = LVCMOS15 ;
NET "SW[7]"          LOC = AD10      | IOSTANDARD = LVCMOS15 ;
NET "SW[8]"          LOC = AE10      | IOSTANDARD = LVCMOS15 ;
NET "SW[9]"          LOC = AE12      | IOSTANDARD = LVCMOS15 ;
NET "SW[10]"         LOC = AF12      | IOSTANDARD = LVCMOS15 ;
NET "SW[11]"         LOC = AE8       | IOSTANDARD = LVCMOS15 ;
```

5) Sample Case – Game Playthrough

Here is what would be displayed through our VGA output in one complete playthrough of the game, and as you can see, we start with a jumble of unaligned and unconnected tubes, connecting them together one by one until we create a path from “Start” to “Finish”:



6) Discussion and Revision

One of our biggest mistakes, which we spent the longest time fixing, was the "RGB bitstring formatting problem"; Initially we supposed that the FPGA used here supports 8 bits RGB values for colour like most of the other FPGA. There wasn't an English documentation for VGA module which made our job way harder. After days of trying to get to work we were told that the board support 12 bits colours. Further, when we tried to use switch cases for our pipe module we encountered a bug with the Xilinx software, which kept crashing the computer by consuming all the available ram on the computer and freezing. This forced us to use a massive array which had all the modes stored in it with the size of 64,000 elements with each element having a size of 12 bits. Due to this problem had to wait tens of minutes for synthesizing and generating the bitstream. We had plans to add the backgrounds to the game and a starting screen but due to this constraint it was not feasible. The final project took over 40 minutes to synthesize, and over 10 minutes to generate the bitstream.

There were few difficulties with both buttons and keyboards due to the timings. There was no viable tutorial for the buttons which we initially intended to use, and after trying to figure it out on our own, we ended up with switching over to PS2 keyboard. Although there was a document explaining everything and following every step of it, we still had some difficulties with timing of the keyboard, meaning that if the key was held for a certain amount of time the program would register it as multiple key presses. This is a bit problematic if the user holds a key for longer than intended, but overall the project was a success despite the difficulties we faced.

We wish there was more English documentations on the FPGA, and more resources available to us.

7) Labour division

Names	Duty Percentage
Parsa (Leader)	27.5%
Richard	27.5%
Brian	22.5%
Hansoo	22.5%