CPE223 Digital System Design

Final-project : Flappy Box

Technical Report

Group 7 Section 32B

Supakorn	Srisawas	62070503449
Chatchapon	Sukitporn-udom	62070503455
Natthasit	Chunhaworawong	62070503456
Narapathra	Morakrant	62070503464

Submission date

25 September 2020

Instructors

Suthathip Manee

Prapong Prechaprapranwong

TABLE OF CONTENTS

1. OBJECTIVE(s)

The objective of the project is to create a video game using the knowledge of Verilog and other theories that were studied in class by implementing the program on Basys3 board to display on a computer monitor via VGA port. The expected features of the program including score showing on both the monitor and the 7-segment display, being able to change character skins, themes, and game modes.

2. EQUIPMENT

The equipment used in this project consists of the main board for the program and as the input device, a connector, and the output device, including:

- Basys3 Artrix-7 FPGA Board (Serial Number : DACAB80)
- A computer monitor with VGA port (Samsung LCD 22Inch)
- A VGA to VGA cable

2.1 Basys3 Artix-7 FPGA Board

The Basys3 is a FPGA development board with Xilinx Artirx-7 FPGA architecture designed for Vivado Design Suite. It offers a lot of ports and peripherals such as 16 user switches and 4-digit 7-segment display. There are 4 types of connectors available on the board, one of them is 12-bit VGA connecter which is applied in the project. The board is used to implement the main program using Verilog as VHDL.

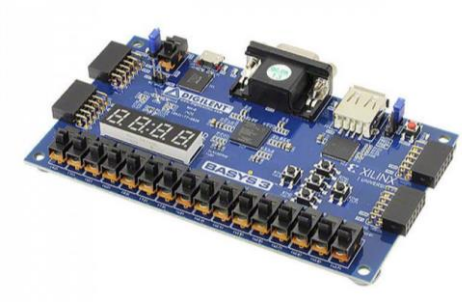

Fig.1. Basys3 Artix-7 FPGA Board.

2.2 Computer monitor

The computer monitor is a device used for showing information in pictorial form. The computer monitor is used as an output to display the game for the project. It is connected to the board via VGA port using VGA cable. Any type of monitor is acceptable for the project as long as it has VGA port. In this experiment, we used a Samsung LCD monitor with the size of 22 inch for testing.

Fig.2. Computer monitor.

2.3 VGA connector and cable

VGA stands for Video Graphics Array, first introduced with IBM® PS/2 computer in 1987. A VGA connector is a standard connector for transferring video signals, in this project the VGA to VGA cable is used to transfer the video signal from Basys3 to computer monitor. A VGA cable carries 5 main signals, the RGBHV (red, green, blue, horizontal sync, and vertical sync.) The red, green, and blue wires carry the color of each pixel. A horizontal sync is to specify the time taken to scan through a row of pixels and a vertical sync is to specify the time taken to scan through an entire screen of pixel rows.

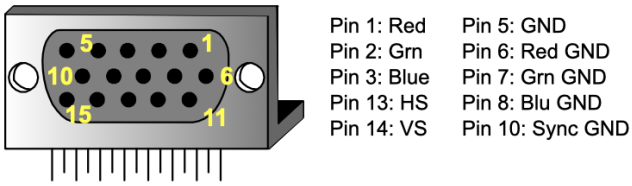

Fig.3. VGA connector.

3. EXPERIMENTAL PROCEDURE

The project implementation is in the board written in Verilog as VHDL and using a VGA cable to transmit the color signal to monitor. Therefore, in this part will be focused on the design and function of the important modules that control this video game.

3.1 Setting the VGA connection

Pixel displayed on the monitor has red, green, and blue color component, where all of these color component signals can be transmitted via VGA cable. Basys3 has 12-bit color depth capabilities. The color signal generated by resistor ladder DACs on the Basys3 board that receives input from IO lines which in this case will be 12 IO lines.

As mentioned in the equipment section above, the VGA port has two synchronization signals, the horizontal sync (hsync) and the vertical sync (vsync) are the critical signals to connect the VGA with the board. We want to display 640 x 480 pixels, so we created a module with 2 inputs and 6 outputs (see Fig.4)

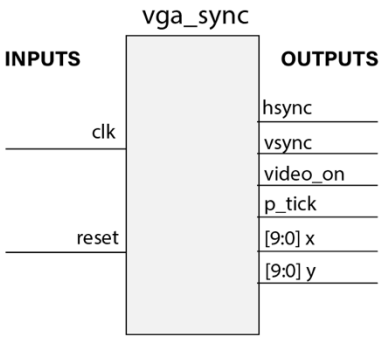

Fig.4. module for VGA connection.

We have to divide the clock from originally 100MHz to 25 MHz using mod-4 counter according to the table below which shows the timing specific for 640 x 480 resolution at 60Hz frame rate values from Learn.digiletinc

TABLE 1. The timing specification for 640x480 at 60Hz Frame Rate

Description	Notation	Time	Width/Freq
Pixel Clock	t_{clk}	39.7 ns (± 0.5%)	25.175MHz
Hor Sync Time	t_{hs}	3.813 µs	96 Pixels
Hor Back Porch	t_{hbp}	1.907 µs	48 Pixels
Hor Front Porch	t_{hfp}	0.636 µs	16 Pixels
Hor Addr Video Time	t_{haddr}	25.422 µs	640 Pixels
Hor L/R Border	t_{hbd}	0 µs	0 Pixels
V Sync Time	t_{vs}	0.064 ms	2 Lines
V Back Porch	t_{vbp}	1.048 ms	33 Lines
V Front Porch	t_{vfp}	0.318 ms	10 Lines
V Addr Video Time	t_{vaddr}	15.253 ms	480 Lines
V T/B Border	t_{vbd}	0 ms	0 Lines

3.2 Making the main character and its mechanics

The character is created by plotting the pixel by pixel value and converting both the color plotted and its position in the pixel matrix to binary numbers. Fig.5. below is an example of the skin design for the character in a pixel plot. The width and height of the character is 20 x 24 pixels.

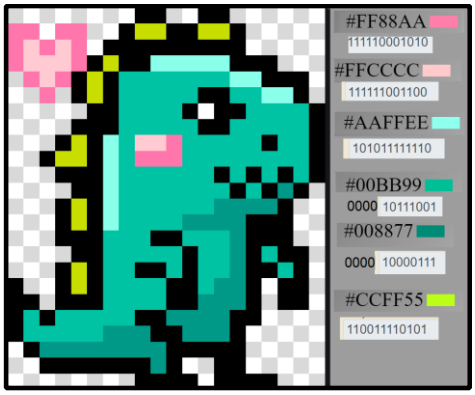

Fig.5. One of the skins of the main character in pixel plot.

The character may only move up and down so it will have a fixed x-position with alterable y-position. When starting the game, the character is fixed at a position. If the up button is pressed, the finite state machine with a datapath (FSMD) for moving the character will be used. Fig.6 show the pseudo code for the logic use in the module.

```
If in jump down state
     if jump count is more than 0
          decrease jump count by 1
     if jump count is equal to 0
          if the game state is idle state (start screen)
               still stay in mid-air
          else if character is within the screen
               start to fall down
          else if the character is at the bottom of the screen
               remain at the same position
     if up-botton is pressed
          next state equal to jump up state

If in jump up state
     if jump count is more than 0
          decrease jump count by 1
     if jump count is equal to 0
          if can still go up
               if character is below to top of the screen
                    go up in to a higher position
               else
                    stay in the same position
     else
          next state equal to jump down state
```

Fig.6. the pseudo code for the logic use in player module.

3.3 Create ROM for images

Same as the character's skins, any images displayed have to be converted to 12bit RGB color so it is possible to transmit via VGA port. Fig.7 is an example code of how the value of each color is assign to a pixel matrix. 15'b000000000000011 is a position of row and column in pixel and 12'000100010010 is a 12bit RGB color code.

```
15'b000000000000011: color_data = 12'b001000100010;
15'b000000000000100: color_data = 12'b000000000000;
```

Fig.7. sample assign the color of each pixel

3.4 Generating the obstacles and their motion

To generate the obstacles with difference height, we use the current character's (player's) y-position to randomize the height of the obstacles. We use the y-position of the character because it is the most frequently changed value. Fig.8 shows the logic used to random the height, with this logic it is rarely to generate the obstacles with the same height which increases the difficulty.

```
obstacle height = {(y-position of the character)%200}+ (difference number for each obstacle);
```

Fig.8. The logic used to random the height of obstacles

The obstacles move by updating their locations by 1 pixel to the left on every positive edge of the tick signal. The tick signal is asserted when the count time of position update reach maximum set time. Down below (see Fig.9.) is the code from one of the obstacles modules showing the part in the module that make the obstacle in motion.

```
localparam TIME_MAX    =   4000000;

    reg [25:0] time_reg;  // register to keep track of count time between position updates
     wire [25:0] time_next;

    // infer time register
    always @(posedge clk, posedge reset)
     if(reset)
          time_reg <= 0;
     else
          time_reg <= time_next;

    // next-state logic, increment until maximum, then reset to 0
    assign time_next = (time_reg < TIME_MAX - speed_offset ) ? time_reg + 1 : 0;

    // tick signal is asserted when time reg reaches max
    wire tick;
    assign tick = (time_reg == TIME_MAX - speed_offset ) ? 1 : 0;

     // on positive edge of tick signal, or reset, update obstacle location
    always @(posedge tick, posedge reset)
          begin
          //defaults
          s_x_next = s_x_reg;
          s_x_next = s_x_reg;
          S_H_next = S_H;

          if(reset)
            begin
               s_x_next = 670;            // reset to starting x position
               S_H_next = {(p_y)%200}+60; // random height by calculate current y position of player
               if(S_H_next > 230 || S_H_next < 0)
                  S_H_next = 185;
            end
         else if(s_x_next == 0)
            begin
             s_x_next = 670;            // reset to starting x position
             S_H_next = {(p_y)%200}+50; // random height by calculate current y position of player
             if(S_H_next > 230 || S_H_next < 0)
                  S_H_next = 185;
            end
          else
               s_x_next = s_x_reg - 1;     // move obstacle to the left

          end
```

Fig.9. Part of obstacle module that make the obstacle move.

3.5 Determining the collisions of character, obstacle, and floor

Checking for a collision of the character and an obstacle requires both x and y position of both elements. Then we compare the positions of both elements whether they are the same or not, and update the value to the main stage of the game that the game is over. See Fig.10 for sample code in the collision module where p_x and p_y are the main character xy position, o1_x and o1_y are the obstacle position, and S_H1 is the safe zone that the character can past though obstacle 1.

```
// register for collision state
    reg collide;

    always @ *
         begin
               // default
               collide = 0;

               //floor
               if(p_y + 25 == 480)
                  collide = 1;

               //obstacle1
               if(((p_x + 20 > o1_x - 29) && ( p_x + 20 < o1_x )
                  && ( (S_H1 > p_y) || (p_y+25 > S_H1+120) ) )
                  || ( (p_x + 20 > o1_x) && (p_x < o1_x )
                  && ( (S_H1 > p_y) || (p_y+25 > S_H1+120) ) ) )
                  collide = 1;
```

Fig.10. A segment from collision module.

3.6 Counting and displaying the score

We design a module that queues the obstacles by using the x-position of the obstacle ahead of it. Fig.11 shows a part of code in the obstacle_manager module where its main function is to check the x-position of the character played by user and compared to the x-position of the obstacles if the player and the obstacle have the same x-position, 1 score will be added.

```
//enable signal register
    reg [3:0]enable_reg; // this enable is like reset signal : 0 is enable 1 is disable

    //score register
    reg score_reg;

    always @(posedge clk, posedge reset)
    begin
        if(reset)
        begin
            enable_reg = 15; //disable all
            score_reg = 0;
        end
        else
            begin
            //queue each obstacle by x position of obstacle ahead
            if(((o4_x <= 505 && o1_x > o4_x)||(o4_x > o1_x)||(o4_x == o1_x))&& game_en)
                enable_reg[0] = 0; //enable obstacle 1

            if(((o1_x <= 505 && o2_x > o1_x)||(o1_x > o2_x))&& game_en)
                enable_reg[1] = 0; //enable obstacle 2
                                                        Lorem ipsum
            if(((o2_x <= 505 && o3_x > o2_x)||(o2_x > o3_x))&& game_en)
                enable_reg[2] = 0; //enable obstacle 3

            if(((o3_x <= 505 && o4_x > o3_x)||(o3_x > o4_x))&& game_en)
                enable_reg[3] = 0; //enable obstacle 4

            //when player jump pass each one of obstacle, they will get a score
            if( game_en && ( (p_x-3 == o1_x)  || (p_x-3 == o2_x) || (p_x-3 == o3_x) || (p_x-3 == o4_x) ) )
            begin
                score_reg = 1;
            end
               else
                score_reg = 0;

            end
    end
```

Fig.11. a part of obstacle_manager module.

To display the score on both monitor and 7-segment display, we create a score module that receive an output from obstacle manager, which determines when to add the score, as an input. In the score module, there is a value that keeps track of the score then sends the score to a decoder to display on 7-segment display, and calls the module that store number bitmap ROM to display on screen.

3.7 Implementing the main state of the game and top module

There are 4 main stages in the game which are initial, idle, playing, and game over. Fig.7 shows the pseudo code for the main logic of this module.

```
Start the game at init stage

Always (infinite loop)
     If at init stage
          if time count greater than 1
               decrease the time count value
          else
               next game stage equal to idle stage
     If at idle stage
          if player pressed up-button
               begin the game
               next stage equal to playing stage
     If at playing stage
          if there is a collision
               next stage equal to game over stage
     If at game over stage
          if player pressed up-button
               next stage equal to initial stage
```

Fig.7. Brief pseudo code of main states in the program.

Top module is the module that call other modules to able to test the program. We instantiate the VGA sync, character, obstacles, and other modules in the top module. All the switches are also declared in this module. In our program, this module also includes the RGB multiplexing circuit to make changes to the themes and game mode for monitor display.

4. EXPERIMENTAL RESULT

The program came out working perfectly with all features, ability to change character's skin, adjust the color themes and modes, and display the score. Fig.8-10 show the experimental results.

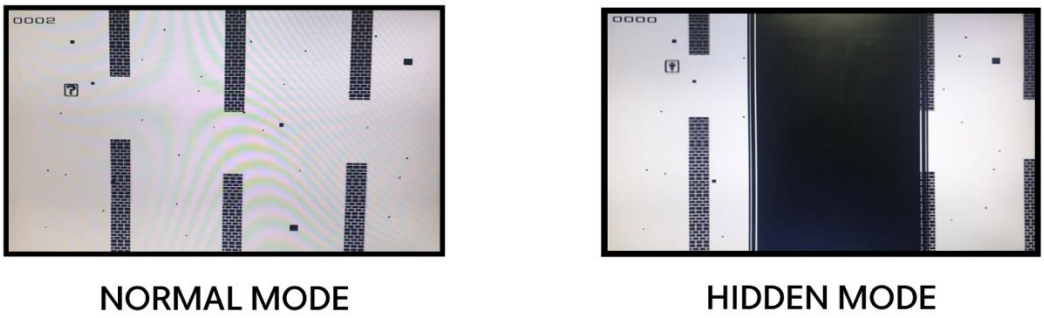

Fig.8. different mode example.

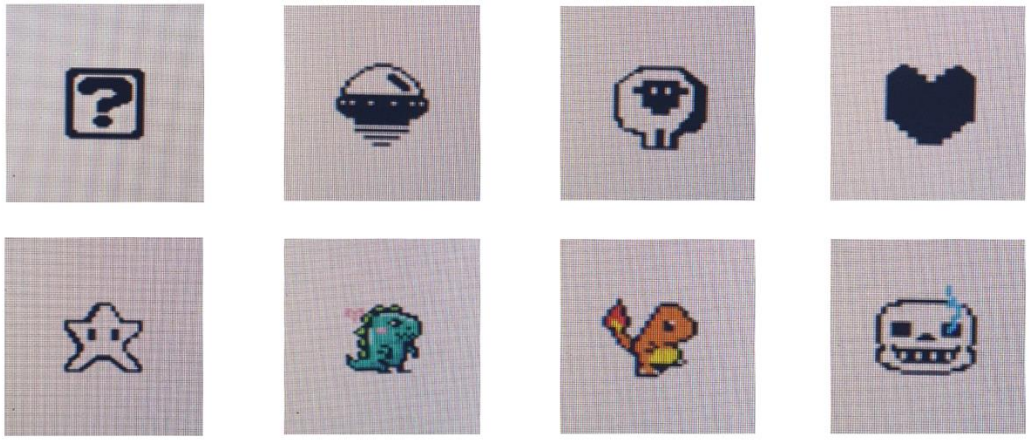

Fig.9. all character's skins.

Fig.10. sample themes.

5. DISCUSSION

The experiment can be performed as expected and met all objectives of the project. Even though the game is working perfectly, there are a lot of warnings about ROM. The calculation for any ROM value is correct. We think that it is just a normal warning from the reason that we are using large ROM spaces.

6. CONCLUSION

Many knowledges that studied in class are used in this project including finite state machine, decoder, multiplexer, ROM, and counter. There is also other knowledges such as VGA connection and assigning 12bit color code to pixel.

MOD-2 counter is used to divide the input clock signal to the desired lower clock so that it is able to display on monitor with 6oHz frame rate. Multiplexer is used for changing the color theme of everything displayed on the monitor at once. All the patterns showing on monitor such as skins and game logo are stored in ROM by converting the value of RGB color to 12bit color code. The finite state machine controls the main flow of the game, also the same for how the character (player) moves. The decoder uses for converting the 4bit value to the 7-segment display.