CS M152A

Lab 4 Report

DDR

Authors:

Aaron Yoo, 004745379

Chayanis Techalertumpai, 404798097

Christian Rodriguez, 804789345

Date: 12/5/2019

TA: Logan Kuo

Introduction and Requirements

In this lab, we were given a freedom to choose and implement any project we want. We then came up with a classic rhythm game that challenges a player's sense of rhythm. The game requires a player to press buttons in a sequence dictated on the VGA screen. The sequence of colored blocks (representing Up, Down, Left, and Right) moves forward at a certain rhythm towards the hit box located at the bottom of the screen. Each time a player correctly hits the button corresponding to the colored block in the hit box, he/she scores one point in the game. The game ends when the player scores 20 points. The score is always displayed on the seven segment display on the board. When the game ends, the screen blacks out and a player must hit the reset button to reset the game state. Our inspiration for building this game comes from our love in music and the versatile application of the FPGA board which enables us to implement the game with the hardware offered in this lab.

For our program to be functional, it must meet certain requirements as stated under Key sub-function in the project proposal.

- It must contain a button debouncer for reliability. This ensures that the player's score is calculated accurately (only when the correct button is hit) and without delays.
- It must be able to generate sequences of colored block which constitutes a rhythm of the game.
- It must be able to detect if the right button is pressed or if a block is missed and increment the score accordingly. This will determine whether the player hit or missed the note.
- It must be able to manipulate the game state based on the input from the user. It should be able to manipulate the "notes" that are on the screen, store the player's current score, and reset the game when needed.
- It must be able to render video to the board over VGA. The video output will reflect the game state.

Design Description

Initially, we planned to make our program as modular as possible. However, we faced the issue of accessing multiple drivers, so we combined most of our game logic in one module in order to avoid the error. Below is a screenshot of our high level design for our rhythm game.

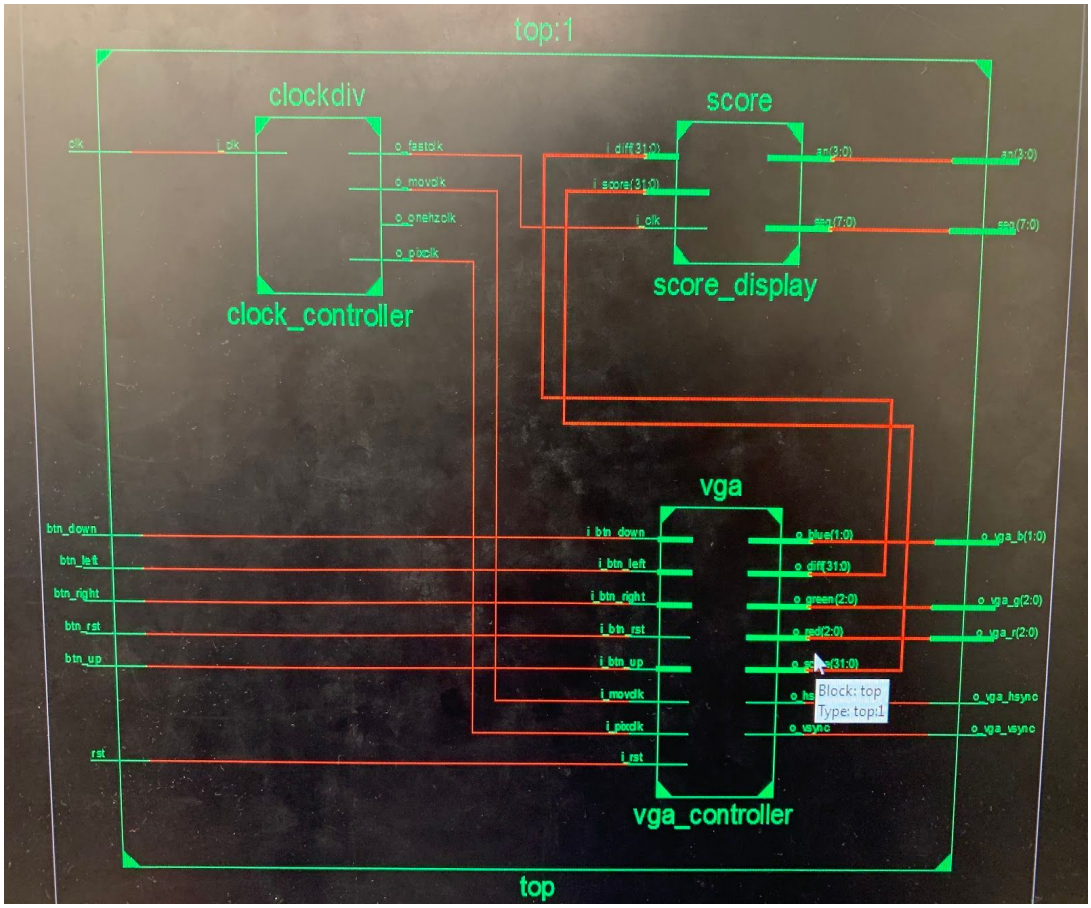

Figure 1: The program has three main modules: vga (referred to in this report as The Brain), clockdiv, and score.

Our clockdiv module is responsible for holding all of our clocks while our score module is responsible for displaying the player's score onto the seven segment display and our vga module is responsible for our game logic and our monitor display.

Rhythm Generator

Although not a module, it is still important to note how we generated the rhythms for our game. Essentially, we encoded the color of our buttons into binary digits and stored the encoding in a text file. We then proceeded to preload the text file into our program and put it inside an array. Since our array contained our rhythms, we looped through our and decoded the binary digits and displayed them in the screen appropriately. 000 means red, 001 means white, 010 means green, 011 means blue, and 100 means a blank color.

```
100
100
000
100
001
100
010
100
011
100
100
001
011
010
010
011
001
010
011
100
001
100
001
001
010
```

Figure 2: Binary text file containing encoded colors

Debouncer

```
module debouncer(
      input clk,
      input btn,
      output state
);
```

In order to sample buttons in a reliable way, we need a debouncer (similar to lab 2). The debouncer takes in a clk and the btn input and outputs the debounced “state” of the button. The debouncer is used to debounce our ADJ and RST buttons.

Clock module

```
module clockdiv(
    input wire i_clk,

    output wire o_pixclk,   // pixel clock (25 MHz)
        output wire o_movclk,   // clock to control arrow movement
    output wire o_fastclk   // clock to update seven segment display
);

reg clk_25mhz = 0;
integer cnt_25mhz = 0;
always @(posedge i_clk) begin
    if (cnt_25mhz >= 1) begin
        cnt_25mhz = 0;
        clk_25mhz = ~clk_25mhz;
    end
        else begin
                cnt_25mhz = cnt_25mhz + 1;
        end
end
assign o_pixclk = clk_25mhz;

reg clk_mov = 0;
integer cnt_mov = 0;
always @(posedge i_clk) begin
        if (cnt_mov >= 312500) begin  // right now its 160 Hz
                cnt_mov = 0;
                clk_mov = ~clk_mov;
        end
        else begin
                cnt_mov = cnt_mov + 1;
        end
end
assign o_movclk = clk_mov;

reg clk_fast = 0;
integer cnt_fast = 0;
always @(posedge i_clk) begin
    if(cnt_fast >= 5000) begin
        cnt_fast = 0;
        clk_fast = ~clk_fast;
    end
    else begin
        cnt_fast = cnt_fast + 1;
    end
end
assign o_fastclk = clk_fast;

endmodule
```

Figure 3: Clock module code

A clock module takes in a master clock and outputs four different clock signals at varying rates. Each clock has different purposes:

- pixel clock: a 25 MHz clock
- mov clock: a 160 Hz clock to control the speed of the colored blocks
- fast clock: a 5 kHz clock to update the seven segment display

The Brain

Our vga.v contains the “brain” of our game. It contains the code that is responsible for communicating to the VGA ports and manipulating all game states. We first draw 5

blocks onto the screen. These blocks are colored depending on whatever colors are encoded by our binary rhythm file.

```
// Define the drawing regions for each symbol and each type
assign c5_block = (center && c5 - 40 < v_count && v_count < c5 + 40);
assign c4_block = (center && c4 - 40 < v_count && v_count < c4 + 40);
assign c3_block = (center && c3 - 40 < v_count && v_count < c3 + 40);
assign c2_block = (center && c2 - 40 < v_count && v_count < c2 + 40);
assign c1_block = (center && c1 - 40 < v_count && v_count < c1 + 40);
assign c0_block = (center && c0 - 40 < v_count && v_count < c0 + 40);
```

We use a clock in order to move each block down on the screen by a constant amount. This happens so fast that it looks like the blocks are smoothly gliding across the screen. We also created a hitbox so that the player knows when to press the button on the FPGA Board. Once a box enters the hitbox, if a player hits the correct button that corresponds to the box color, then we increment the player's score and output a white box on the bottom left corner to symbolize a correct button press.

```
// push_range_c5 = 1 when c5 is in the target box
reg push_range_c5 = 0;
always @(*) begin
    if (VFP - 40 - 25 < c5) begin
        push_range_c5 = 1;
    end
    else begin
        push_range_c5 = 0;
    end
end
```

If the player presses the wrong button, the score does not increment and the bottom left corner will show a black box. Once the player reaches a score of 20, then the game will end and the screen will turn black. In this state, the player can press the middle button on the board in order to reset the game and play again!

```
always @(posedge correct or posedge reset_me) begin
    if (reset_me == 1) begin
        game_end = 0;
       real_score = 0;
    end
    else begin
        real_score = real_score + 1;
```

```
        if (real_score >= 20) begin
                game_end = 1;
        end
    end
end
```

We had to do a lot of math in order to display the shapes and colors on the screen correctly. If our math was wrong then our boxes would be placed on a random part of the screen instead of the center. The output of this module are the vga outputs and the score, which we pass onto the ucf file and the score module respectively in order for our game to work flawlessly.

Score Display

Score display takes in the score integer from The Brain and displays it on the seven segment display. We decided to only use the two leftmost digits on the display for stylistic purposes. To display the score, we obtain each digit from the score integer and store it in a 4-bit wire. Each wire is then continuously assigned to the segment display at the rate of the fast clock from Lab 3, so that the digits appear constantly bright and illuminated.

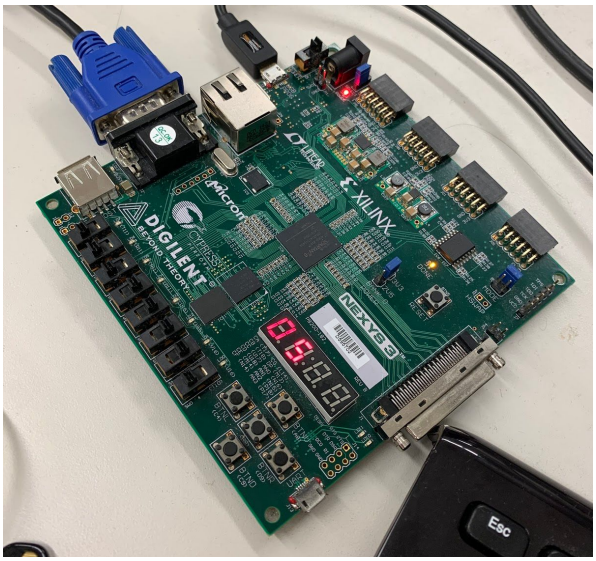

Figure 4: The score is being displayed and updated when player is playing the game.

Top

```
module top(
    input wire clk,
    input wire rst,
        input wire btn_up,
        input wire btn_right,
        input wire btn_left,
        input wire btn_down,
        input wire btn_rst,
    output wire o_vga_hsync,
    output wire o_vga_vsync,
    output wire [2:0] o_vga_r,
    output wire [2:0] o_vga_g,
    output wire [1:0] o_vga_b,
    output wire[7:0] seg,
    output wire[3:0] an
);

    wire pixclk;
    wire movclk;
    wire fastclk;
    wire integer score;

    clockdiv clock_controller(
        .i_clk(clk),
        .o_pixclk(pixclk),
        .o_movclk(movclk),
        .o_fastclk(fastclk)
    );

    vga vga_controller(
        .i_pixclk(pixclk),
        .i_rst(rst),
        .i_btn_up(btn_up),
        .i_btn_right(btn_right),
        .i_btn_down(btn_down),
        .i_btn_left(btn_left),
        .i_btn_rst(btn_rst),
        .i_movclk(movclk),
        .o_hsync(o_vga_hsync),
        .o_vsync(o_vga_vsync),
        .o_red(o_vga_r),
        .o_green(o_vga_g),
        .o_blue(o_vga_b),
        .o_score(score)
    );

    score score_display(
        .i_clk(fastclk),
        .i_score(score),
        .seg(seg),
        .an(an)
    );

endmodule
```

Figure 5: Top module code

Our top module links everything we've built together in order for our rhythm game to work. In this module we hook up our clocks from our clockdiv.v file so that we can use them inside our score.v and vga.v. Our score.v needs a clock in order for it to display the players score constantly onto the seven segment display. Our vga.v (the brain) needs a clock so that we can control how the pixels on the screen are moving. Finally, we make sure to link the outputs of our brain to the vga ports in order ucf file so that we can display our game onto the computer monitor.

Conclusion

This project is very exciting and fun because we got to choose what we want to create and design everything by ourselves without guidelines. We therefore got to experience the actual workflow and processes industrial professionals have to go through in designing, developing, and perfecting a product. We also learned some important lessons; one of which being that we sometimes overestimated our capabilities and did

not think through some designs carefully, causing our final project to be slightly different from the project proposal as follows.

We initially were going to display arrows Up, Down, Left, Right on the VGA screen to indicate which button a player has to press. However, drawing arrows were much more complicated than expected. Since we only have a relatively short period of time to finish the project, we decided to implement a block with different colors, each corresponds to a certain position instead.

In addition, we initially wanted to make the game so that it ends when a player misses a number of blocks. However, this turns out to be more difficult than expected because we could not find a good way of calculating the differences between the real score (what the player actually scores) and the perfect score (the number of blocks passed the hit box at one point in time). We used to increment the perfect score when a block hits the bottom of the hit box. However, that is proven to be inaccurate since we perform that for every posedge of the clock, causing the perfect score to be incremented too many times. We then came up with an alternative way to end the game, by allowing the player to play for a certain amount of time so they can compete by comparing who can score higher within limited time. This implementation was thought to work, but then we exceeded the board's capacity as the program was too large. After consulting with the TA, there is no way around it so we have to change our logic for ending the game. Finally, we decided to end the game when the player reaches a certain score, 20 in our complete version.

Overall, we feel that we have learned crucial skills that will be helpful in our career from this lab and this class in general. We were allowed to explore the bridge between software and hardware and the possibilities the combination offers.