Project Report

Group 99
Keyu Ji & Xiaoxu Nan

March 2020

1 Introduction

In this project, we built a remake of coin-seeking competitive two-player Mario game[1]. In our game, each player can control a character to move to the left, to the right, or jump onto the platforms using either on-board buttons or PMOD joystick. The goal is to compete to catch the coin appearing on the screen. If caught, a new coin will be generated at a new location. Whoever gets the coin scores one point, and the first player to gain fifteen points wins the game. The scores are shown on the on-board seven segment display.

2 Design Description

The design is comprised of five major modules, namely the top module, the game logic module, the VGA display module, the joystick module, and the seven segment display module. Their interaction is illustrated in figure 1. We will introduce each of the five modules in the following sections.

2.1 Top Module

```
module main(MemOE, MemWR, RamCS, FlashCS, QuadSpiFlashCS, // Disable them
        ClkPort, // the board's 100MHz clk
        BtnL, BtnU, BtnR,
        Sw0, // For reset
        vga_hsync, vga_vsync,
        vga_r0, vga_r1, vga_r2,
        vga_g0, vga_g1, vga_g2,
        vga_b0, vga_b1,
        MISO, SS, MOSI, SCLK, //output from joystick
        ca, cb, cc, cd, ce, cf, cg, an,
    );
```

The top module takes in all the input signals and puts together all other modules to coordinate ports and execute the game program. It redirects HSYNC, VSYNC and rgb values to the monitor, as well as values for seven segment display. It calls a simple clock-dividing module to output a 200Hz for game logic update.

[1] http://2.6822.com/www.9/full_26065.html, requires flash to run

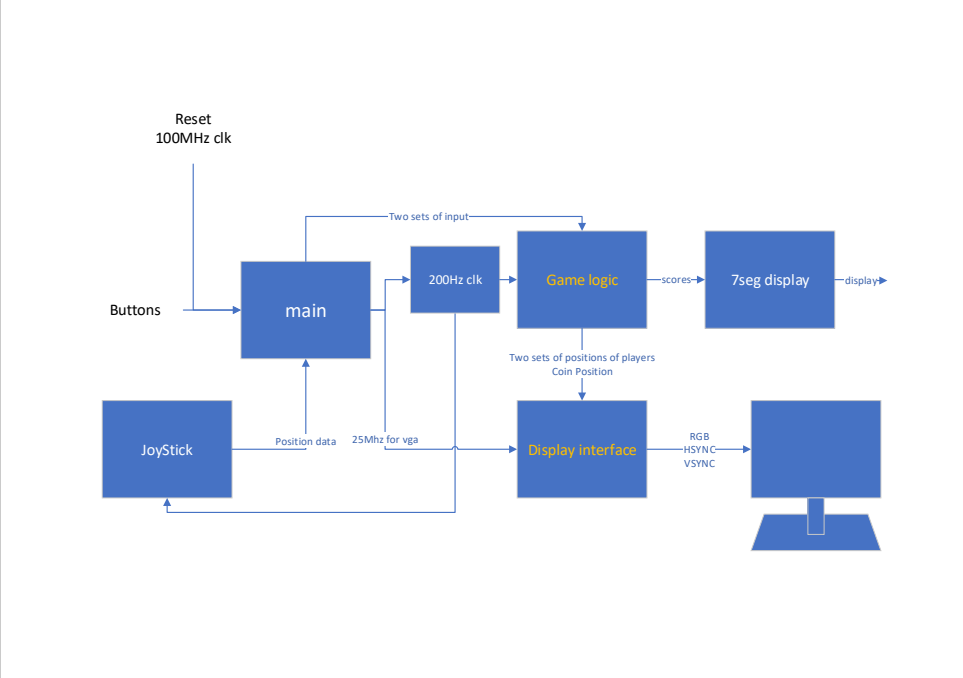


Figure 1: Interaction among major modules

2.2 Game Logic Module

```
module game_logic(
    input clk,
    input rst,
    input BtnL,
    input BtnU,
    input BtnR,
    output reg [9:0] mario_x,
    output reg [9:0] mario_y,
    output reg [9:0] coin_x,
    output reg [9:0] coin_y,
    output reg [3:0] score_ten,
    output reg [3:0] score_one,
    input SigU,
    input SigL,
    input SigR,
    output reg [9:0] mario_x1,
    output reg [9:0] mario_y1,
    output reg [3:0] score1_ten,
    output reg [3:0] score1_one
);
```

The game logic module handles the logic of the game as its name suggests. It takes in directional signals that control the movements of the characters and outputs the updated position of the characters and the coin.

The horizontal movement is always of uniform speed. In each clock cycle, if left or right signal is received, the module updates the horizontal position of the character to

the left or right by one pixel. The characters are not allowed to move out of the scene horizontally and would stop if they reach the left or right edge of the scene. Same applies to the floor, but not to the ceiling – the character can jump over the ceiling but will eventually fall back into the scene. The vertical movements are also of uniform speed, without gravitational acceleration, but with more details. There are two important states that control the character's vertical movements – up_state and on_ground. The former indicates whether the character should move up, and the latter indicates whether the character is on the floor or any other specific map boundaries where the characters can stand on. If on_ground is 1, then when a up signal is received, the up_state of the character is set to 1 on_ground is set to 0. When a character's up_state is set to 1, it begins to move up 1 pixel per clock cycle for 110 clock cycles. After that, the up_state is set to 0. If a character's up_state is 0, the character moves down 1 pixel per clock cycle, until it hits any boundary. Since the character moves pixel by pixel, using hard-code if statements, the module is able to determine whether the character is on any boundary. When it hits any boundary, on_ground is set to 1, and now the next up signal can be valid.

As mentioned, if the coin is caught by one player, then it will appear at a new location. There are 10 fixed locations and the coin will be generated in a specific sequence of these locations.

2.3 VGA Display Module

```
module display_interface(
    input clk,
    input rst,
    input [9:0] mario_x,
    input [9:0] mario_y,
    input [9:0] mario_x1,
    input [9:0] mario_y1,
    input [9:0] coin_x,
    input [9:0] coin_y,
    output HSYNC,
    output VSYNC,
    output wire [2:0] R,
    output wire [2:0] G,
    output wire [1:0] B
);
```

The VGA display module and related sub-modules for generating hsync and vsync signals are adapted from online sources[2]. We made changes to it to suit our purposes. The module takes in positions of the characters and the coin so that the module knows what ranges of pixels should display the characters and the coin. The background shown in figure 2b is unchanged as hard-coded in this module. Using hsync and vsync signals, the electron gun of the display synchronizes with a counter at the backend of this module. The basic idea is that with the value of counter, the module determines the color of the "current" pixel using if-statements. Since the background only consist of color blocks, the we simply need to check which block "current" pixel is at to determine the color.

[2] https://github.com/alexallsup/Chess-FPGA

2.4 Joystick Module

```
module PmodJSTK(
    input CLK; // 100MHz onboard clock
    input RST; // Reset
    input sndRec; // Send receive, initializes data read/write
    input [7:0] DIN; // Data that is to be sent to the slave
    input MISO; // Master in slave out
    output SS; // Slave select, active low
    output SCLK; // Serial clock
    output MOSI; // Master out slave in
    output [39:0] DOUT;     // All data read from the slave
);
```

The joystick module and related sub-modules for data transmission are adapted from the "PmodJSK Demo Project" made by Digilent[3]. Besides regular master clock and reset signal, the joystick module has three inputs and four outputs. We did not dig deep into the entire design but recognized that the most significant ones are `sndRec` and `DOUT`. `sndRec` is another clock received that controls the frequency of data transmission. In our project, we set it consistent with the game logic clock, which is 200Hz, so that the joystick would send and receive data in 200Hz. `DOUT` sends out the position data of the joystick, which consists of two 20-bit value from 0 to 1023. Each value represents the position in x and y-axis respectively. When the joystick is at center, the reading is around 510. In the top module, thresholds of 300 and 630 are used to avoid misinput. For instance, the position data in y-axis needs to be greater than 630 to be considered as a valid upward input.

2.5 Seven Segment Display Module

```
module bcd_display(
    input clk,
    input [3:0] min_ten,
    input [3:0] min_one,
    input [3:0] sec_ten,
    input [3:0] sec_one,
    output reg [3:0] an,
    output ca,
    output cb,
    output cc,
    output cd,
    output ce,
    output cf,
    output cg
);
```

We have adapted the seven segment display module from lab 3. It makes use of input clock and cycles through the four digits of seven segment display. In each clock cycle, one of the digit would be lit up using the decoded signal of numerical input. Since the

[3] https://reference.digilentinc.com/_media/reference/pmod/pmodjstk/pmodjstk_demo_verilog.zip

clock has high frequency, the switching between digits would not be noticeable to human eyes and thus the display would seem stable, with all four digits on display. In our case, the first two digits take in the score of the first player, whereas the last two digits take in the score of the second player.

3 Simulation Documentation

Simulation was minimal in this lab. All testing was done by directly observing the vga display, characters' movement and seven segment display. There is no testbench either.

There is no known bug of our program. All the features we would like to implement have been implemented.

4 Contribution

Keyu found the original game and and made the blueprint. Among the five major modules, top module is relatively simple, joystick module is adapted from Digilent's website, seven segment display module is directly from lab 3. That leaves game logic module and display module to be the major ones in this project. Keyu wrote the display module, which is not hard but really time consuming. We together wrote the game logic module – Keyu wrote the prototype for one player and Xiaoxu modified it to accommodate two players.

5 Conclusion

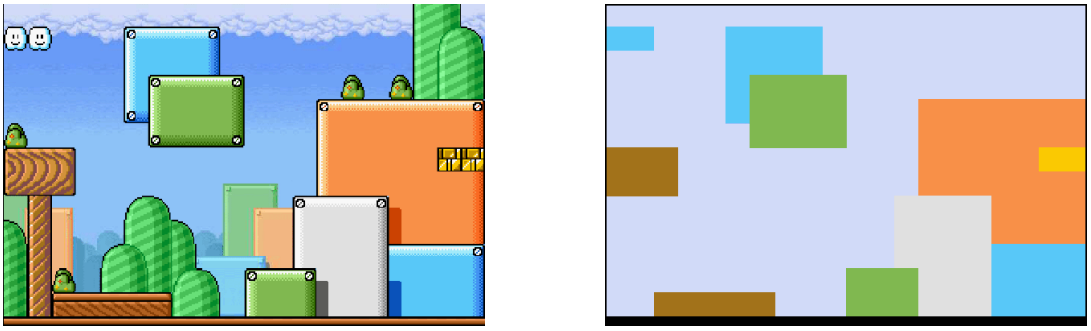
In this project, we built a remake of coin-seeking competitive two-player Mario game that consists of five major modules. Top module calls other modules and redirects output and input. Joystick module reads position data from joystick. Game logic module updates the positions of characters and the coin. Display module decides the color of each pixel. Seven segment display module shows the scores of players on the on-board display.

There are mainly three difficulties we encountered.

The first one is background setup. At the beginning, using `memreadh()`, we would like the FPGA board to simply load the data of the entire background picture. We turned the picture from bitmap to a file of hexadecimals. The dimension of the background is 600*400, so there are 240000 8-bit hexadecimals. We spent hours debugging the code and eventually found that it could not complete synthesizing in 40 minutes. After realizing this is not viable, we had to reduce the complexity of the background and make the picture, figure 2a, into color blocks, figure 2b. The attempt of using `memreadh()` cost a large amount of time.

The second is character's movement. Initially we were not sure what speed is appropriate and what frequency of the game logic clock is appropriate. We spent some time trying with the master clock and found it is simply too fast and even causes distortion of the character's figure. We turned to observe the behavior of the original game and found 200Hz is a good choice.

The third is Verilog's syntactical issue. Still, we were not competent with Verilog's syntax and had made numerous syntactical mistakes, which took us hours to debug and fix it. One major example is that originally we would like to store the positions of all



(a) Original background (b) Color blocks

Figure 2: Change of Background

the boundaries as parameters in the form of unpacked arrays. However, errors showed up when synthesizing and we had to directly hard code all the values in the if-statements that is responsible for checking boundaries. These problems also cost a large amount of time.

One improvement we could make is to add gravitational acceleration to the characters when they jump. To do this, changes have to be made with the game logic module and there are two options to implement this non-uniform speed. One is to let the module takes more clocks with different frequencies. Since the character moves one pixel per clock cycle, choosing different clocks enables the character to move with different speeds. One issue with this approach would be synchronization with horizontal movements, since the horizontal speed does not change and uses the same clock for all the time. The other option is to introduce more states for the character and move multiple pixels in one clock cycle. Issue with this approach is that the boundary detection algorithm has to be changed. It has to be able to detect a range of pixels underneath the character. For now, the algorithm only checks the one and only one pixel below the character.