ECE 385
Final Project
Report

Li, Ziying
Kang, Xingjian

May 27, 2024

1 Idea and Overview

1.1 Overview of the Circuit

The circuit need to realize the fundamental gameplay features: a Doodler, a four-legged critter that attempts to climb platforms to collect points. The doodler receives a greater score as he advances. The doodler constantly jumps in its position, making game control simple by allowing the player to move the doodle to the left or right. The platforms appear indefinitely as the doodler climbs higher. The game has no clear ending, but each gaming session ends when the player hits a monster or falls to the bottom of the screen. The left and right ends of the screen connect, forming a continuous horizontal plane. The user can control the left and right movement of the Doodle by using the "A" and "D" keys on the keyboard.

1.2 General flow of the Circuit

As the detailed block diagram (Figure 3) shows. The flow start with the USB keyboard input, the OTG communication was processed by the Nios II system and hpi_io_inf module in the hardware. Then the keycode can trigger the game state to start. Modules like doodle and platform will cooperatively calculate the element in the game. The drawing_engine is the main graphic processing unit that takes the position of all elements and draw them pixel by pixel to the framebuffer. At last, the palette_mapper convert color to VGA signals and audio_controller convert music to audio. Together, output the image and audio to player.

2 The Implementation of Major Entity

2.1 doodle

Doodle is the main character in this game. This module processes the motion of the doodle, including: keyboard controlled move, gravity acceleration, collisions with platforms, jumps. And also its health, which will be deducted if doodle drops out of screen. Inputs and outputs described as below

```
module doodle #(
    parameter [9:0] W = 640,        // screen width
    parameter [9:0] H = 480,        // screen height
    parameter [9:0] X_min = 140,    // game screen left bound
    parameter [9:0] X_max = 499     // game screen right bound
) (
    input Clk,                                  // 50 MHz clock
    input Reset,                                // Active-high reset signal
    input frame_clk,                            // The clock indicating a new frame (~60Hz)
    input [1:0] frame_clk_edge,
    input [7:0] frame_count,

    input [7:0] keycode, state,                 // game input
    input [9:0] Platform_X [0:7],               // up to 8 platforms on the screen
    input [9:0] Platform_Y [0:7],
    input [7:0] platform_size,                  // addjustable platform size
    //doodle state output
    output [9:0] Doodle_X_out, Doodle_Y_out,    // doodle position
    output [3:0] health,                        // doodle health
    output logic doodle_facing                  // 0: left, 1: right
);
```

2.2 drawing_engine

This is the main graphic processing unit in our design. Running with the 50Mhz system clock, this module will draw every element into frame_buffer one by one, including: background, platforms, Doodle, health bar, score, game state, double buffer indicator, etc. One pixel drawn in each clock cycle.

```
module drawing_engine #(
    parameter [9:0] W = 640,        // screen width
    parameter [9:0] H = 480,        // screen height
    parameter [9:0] X_min = 140,    // game screen left bound
    parameter [9:0] X_max = 499     // game screen right bound
) (
    //system
    input Clk, Reset,               // system clock and reset
    input frame_clk,                // The clock indicating a new frame (~60Hz)
    input [1:0] frame_clk_edge,     // {previous frame_clk, current frame_clk}
    input [7:0] state,              // game state {0: start screen, 1: game running, 2: end}
    input buffer_using, wr_en,      // frame buffer using, write enable
    //elements
    input [9:0] Doodle_X, Doodle_Y, // doodle position
    input [3:0] health,             // doodle health
    input logic doodle_facing,      // doodle disrection {0: left, 1: right}
    input [9:0] Platform_X [0:7],   // up to 8 platforms on the screen
    input [9:0] Platform_Y [0:7],
    input [7:0] platform_size,      // addjustable platform size
    //output to frame buffer
    output [9:0] draw_x, draw_y,    // draw pixel(x,y) to frame
    output [7:0] draw_color         // palette color of pixel
);
```

2.3 framebuffer

This is the framebuffer we used in the design for buffering the whole screen and waiting for the VGA signal to output. We take https://github.com/opengateware-modules/video-framebuffer/tree/master as reference to build our own framebuffer that allows writing to any x-y coordinate pixel we want. And also enables double buffering so that we have ample graphic processing time without affecting the on-screen image.

```
module framebuffer2 #(
    parameter             WIDTH   = 320, // Resolution Width
    parameter             HEIGHT  = 240, // Resolution Height
    parameter             DW      = 8,   // Video Data Width
    parameter             BUFF2X  = 0,   // Use Double Buffering
```

```
    parameter             SYS_CLK = 0    // Use System Clock
) (
    // Clocks
    input  logic          clk_sys,       // System Clock(not used)
    input  logic          clk_pix,       // Core Pixel Clock: 50Mhz
    input  logic          clk_vga,       // VGA Output Clock: 25Mhz
    // Video Input
    input  logic [9:0] x, y,             // write pixel position
    input  logic [DW-1:0] rgb_in,        // palette color input
    input  logic          hblank_in,     // Horizontal Blank
    input  logic          vblank_in,     // Vertical Blank
    // Video Output
    output logic [DW-1:0] rgb_out,       // palette color output
    output logic          hsync_out,     // Horizontal Sync
    output logic          vsync_out,     // Vertical Sync
    output logic          blank_out,     // Video Blank
    // state signal
    output logic buffer_using, wr_en
);
```

2.4 palette_mapper

This module is an essential part for VGA output, the color we used in the previous modules are store in 8-bit(256) palette colors. The palette_mapper module use onchip memory to convert the 8-bit palette color into 3*8-bit RGB output.

```
module palette_mapper (
  input Clk,
    input [7:0] draw_color, //input from frame buffer
    output logic [7:0] VGA_R, VGA_G, VGA_B // VGA RGB output
);
```

2.5 audio_controller

This module communicates with audio interface.vhd with specific ports, and uses state machine to control whether the music is playing or not. A address counter is also used to increment the reading address of the music file triggering by the signal of indicating whether the current line of music is done. We take https://github.com/0x60/385-audio-tools as reference.

```
    input logic     Clk,          // system clock
  input logic     Reset,        // reset

  // the following ports are I/O from board to the I2C codec
  input logic     AUD_BCLK,
  input logic     AUD_ADCDAT,
  input logic     AUD_DACLRCK,
  input logic     AUD_ADCLRCK,
  output  logic     AUD_XCK,
  output  logic     AUD_DACDAT,
  output  logic     I2C_SDAT,
  output  logic     I2C_SCLK
```

3 The implementation process of the C algorithm

C algorithm in our project mainly works as the USB-OTG interface. Generally, by adding some imputs/outs to the NIOS II to help the hardware read numbers from the keyboard.

3.1 The execution cycle of the program

EZ-OTG performs polling periodically to check the keyboard's interrupt IN endpoint, and the keyboard provides a keycode descriptor to EZ-OTG. The data is then kept in RAM by EZ-OTG. The given NIOS II code consists of a while loop that iteratively attempts to use hpi-to-intf to retrieve the data through io_read and io_write from EZ-OTG. The keycode will be finally sent to our top level.

3.2 The data synchronization mechanism of the hardware (which registers are used for synchronization)

3.2.1 Read:

The process of reading requires io_write the address to the **HPI_Address Register**. Then, io_read from the **HPI_Data Register**, and the data will automatically synchronize through reading from RAM to HPI Reg or write from HPI Reg to RAM so that the data in the HPI_Data Register can be guaranteed to be the same as the data in RAM given the specific HPI_Address.

3.2.2 Write:

Similarly, the process of write requires io_write the address to the **HPI_Address Register** and then another io_write to the **HPI_Data Register**.

4 Designs

4.1 Overview of the design procedure

4.1.1 What project/codes is used as the foundation of the project?

We based on Lab8 (with VGA screening, Keyboard input, ball's motion) to build this project.

4.1.2 What are the different objectives of the project (choices of inputs, state machine, sprites, algorithm IPs, storage units, choices of outputs)?

Create an graphic processing unit with high freedom so that we can add backgrounda, sprites, and entities layer by layer with correct video output. Let the main character behave as the game that can jump, stand on the platform

4.1.3 What research/background study has been done to achieve the objectives?

- **Frame Buffer:** An github project that wrote and universal frame buffer in SystemVerilog https://github.com/opengate modules/video-framebuffer/
- **Audio:** We have searched for seniors' logs, WM8731/WM8731L datasheet, etc., and finally we find a ECE385 related tool repository on Github is usefull. However, the code provided in that repo is rather complicated, so we simplify it as the version we are now using.
- **Pixel to txt:** We find a ECE385 related tool repository on Github is usefull. https://github.com/atrifex/ECE385-HelperTools/tree/master

4.1.4 How are the different objectives linked together to form a complete project?

The drawing_engine, frame_buffer, and palette_mapper, together as the graphic processing unit work for image outputting, and audio_controller for audio outputing. The doodle, platforms, game_state, works for in game system computing.

4.2 If some sort of serial processing is used, especially for the projects that deals with algorithms, a state machine and a simulation waveform should be included in the report

The game_state module control the state. State 0 intend for pause/not start, and can be triggered by pressing "Esc" on keyboard. State 1 indicate game running, which can be started by pressing "Enter".

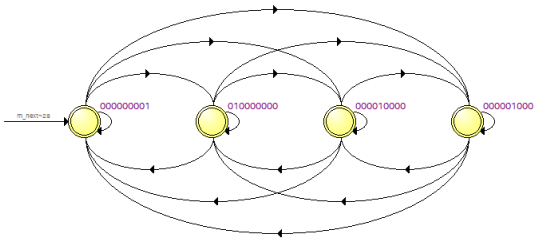

Figure 1: Enter Caption

4.3 Document the Design Resources and Statistics in following table.

LUT	6892
DSP	0
Memory (BRAM)	3,815,424 bits
Flip-Flop	2177
Frequency	139.7MHz
Static Power	102.42 mW
Dynamic Power	0.85 mW
Total Power	188.77 mW

5 Block Diagram

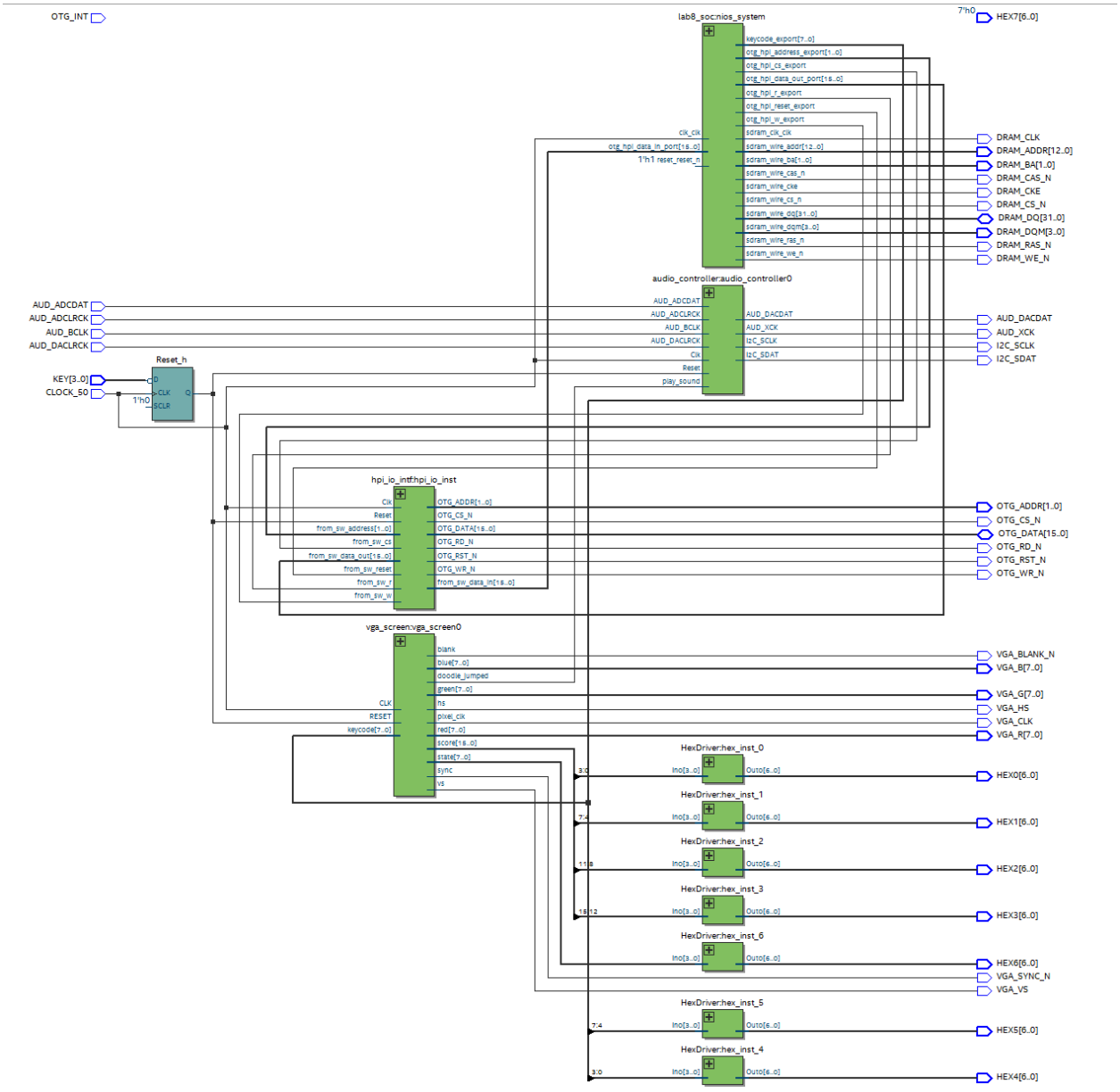

Figure 2: Top-level block diagram

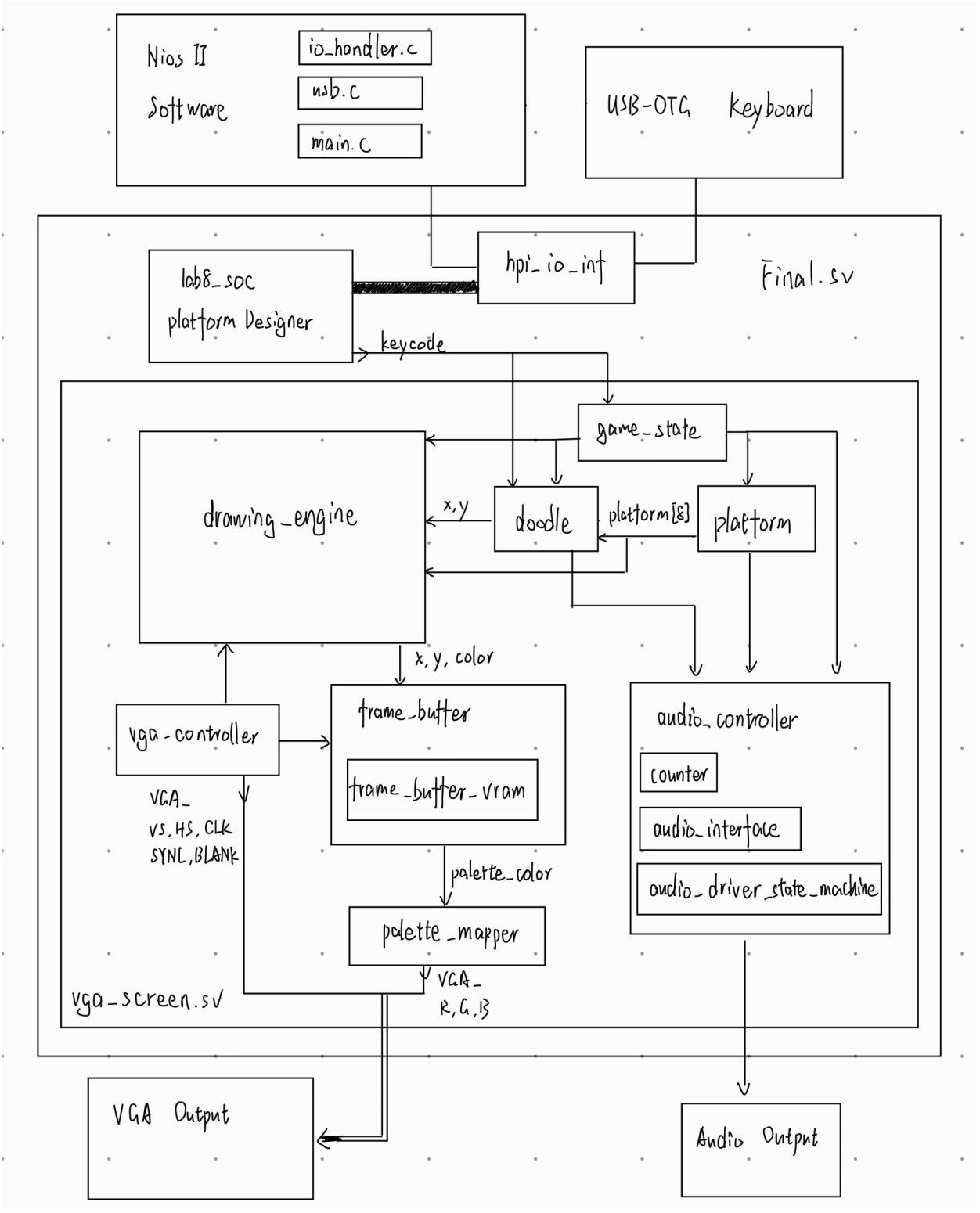

Figure 3: Detailed Block Diagram

6 Conclusion

6.1 implementing and Debuging

It is a tough task to write all things from 0, so we search through the internet and find frame_buffer and audio_controller. We connect our signal to these two modules. But the frame_buffer did not accept x, y, color at first. Instead, it only accept pixel color in the order of line by line then pixel by pixel. So we modified the input and address assignment according to x,y. But the pixel misaligned. So we move the address assignment out of always_ff and it works. Similiar things happens when reading spirites from memory, so we add a logic signal "wait¡=2;" and count donw to solve the problem. The doodle's position is also a issue because it may exceed to smaller than 0 and become things like "12'hFFF", we used int type to solve it.

6.2 Accomplishments

We successfully built the basic framework of the game based on the knowledge we learnt in the class and on the internet. The graphic processing unit works correctly and has good operational efficiency and expandability. The main character can move freely on the screen and has health bar and boundary check. The platforms can move as we want and the character can stand on them. Background music was added also. Reached the difficulty of about 5-7. The demo was great as we wanted.

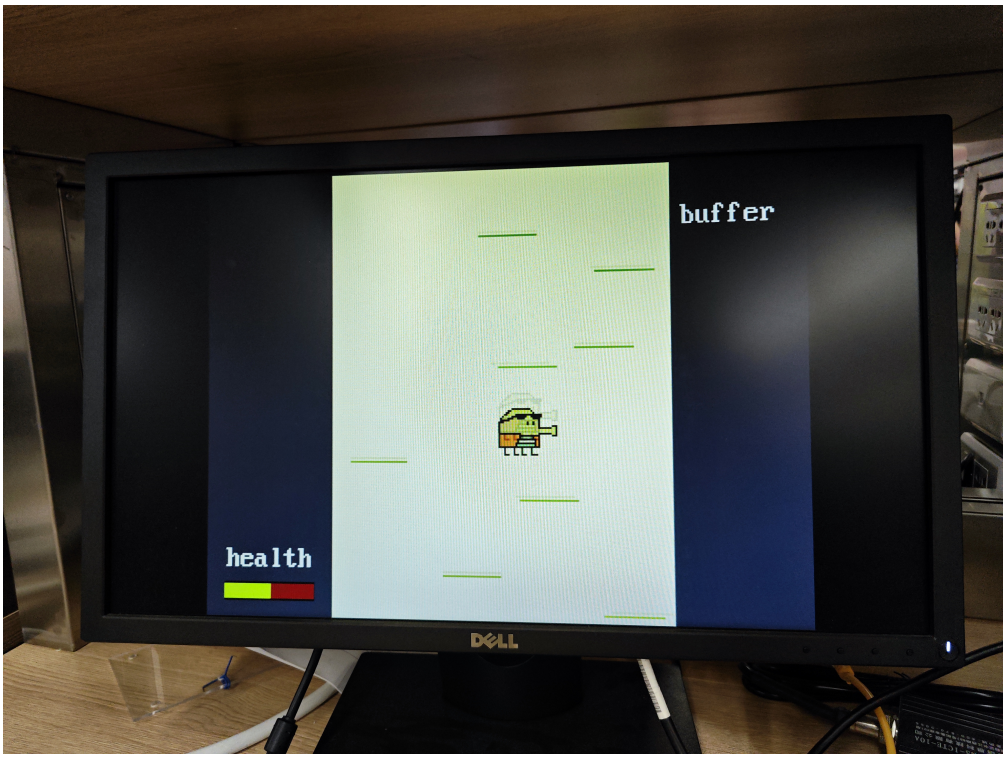

Figure 4: Final Demo