

ECE 385

Fall 2021

Final Project

FePpaGiA

Li Zihao 3190110098

Qi Zhenting 3190112155

D225

Introduction

In the final project, we implemented a simple version of Terraria, a popular sandbox 2D game, on the FPGA. Specifically, we name our version Feppagia. We first implemented essential components related to the game, including the background, the character movement controlling, the blocks, and the item bar. To reproduce the game to the largest extent, after realizing modules above we also added music. SystemVerilog was used to implement these modules and their interface connections, and C code was modified from Lab 8 to realize multiple key pressed simultaneously and to simulate mouse. Our goal is to realize the game using the FPGA board, the USB keyboard, and the VGA monitor to provide basic experience of Terraria.

Written Description

Description of the overview of the circuit

We built our final project based on lab8, SOC with USB and VGA Interface in SystemVerilog. The top-level entity of the circuit is still lab8.sv, but we added a lot of components into it. A block schematic of the top-level entity is also submitted in the .zip file.

What is the purpose of the circuit?

The purpose of the circuit is to implement a simple version of Terraria.

What are its features?

We aim to develop a version with background moving with character, background music, character with anime, cursor controlled by the keyboard, ability to place the block, break the block, choose which block to place, and collision between character and blocks.

Description of the general flow of the circuit

What are the inputs/outputs?

Here is the declaration of the top-level entity module.

```

module lab8(
    input          CLOCK_50,
    input          KEY,           //bit 0 is set up as Reset
    output logic [3:0] HEX0, HEX1, HEX2, HEX3,
    output logic [7:0] LEDG,
    output logic [17:0] LEDR,

    // VGA Interface
    output logic [7:0] VGA_R,     //VGA Red
    output logic [7:0] VGA_G,     //VGA Green
    output logic [7:0] VGA_B,     //VGA Blue
    output logic [7:0] VGA_CLK,   //VGA Clock
    output logic [7:0] VGA_SYNC_N, //VGA Sync signal
    output logic [7:0] VGA_BLANK_N, //VGA Blank signal
    output logic [7:0] VGA_VS,    //VGA vertical sync signal
    output logic [7:0] VGA_HS,    //VGA horizontal sync signal

    // CY7C67200 Interface
    inout wire [15:0] OTG_DATA,    //CY7C67200 Data bus 16 Bits
    output logic [1:0] OTG_ADDR,   //CY7C67200 Address 2 Bits
    output logic [1:0] OTG_CS_N,   //CY7C67200 Chip Select
    output logic [1:0] OTG_RD_N,   //CY7C67200 Write
    output logic [1:0] OTG_WR_N,   //CY7C67200 Read
    input          OTG_RST_N,      //CY7C67200 Reset
    input          OTG_INT,        //CY7C67200 Interrupt

    // SDRAM Interface for Nios II Software
    output logic [12:0] DRAM_ADDR,  //SDRAM Address 13 Bits
    inout wire [31:0] DRAM_DQ,     //SDRAM Data 32 Bits
    output logic [1:0] DRAM_BA,     //SDRAM Bank Address 2 Bits
    output logic [3:0] DRAM_DQM,    //SDRAM Data Mast 4 Bits
    output logic [3:0] DRAM_RAS_N,  //SDRAM Row Address Strobe
    output logic [3:0] DRAM_CAS_N,  //SDRAM Column Address Strobe
    output logic [3:0] DRAM_CKE,    //SDRAM Clock Enable
    output logic [3:0] DRAM_WE_N,   //SDRAM Write Enable
    output logic [3:0] DRAM_CS_N,   //SDRAM Chip Select
    output logic [3:0] DRAM_CLK,    //SDRAM Clock

    // music interface
    output logic AUD_DACDAT, I2C_SDAT, I2C_SCLK, AUD_XCK,
    input logic AUD_BCLK,
    input logic AUD_ADCDAT,
    input logic AUD_DACLCK, AUD_ADCLCK);

```

The inputs are system clock, keys on the FPGA board, CY7C67200 Interrupt and some inputs related to the music module. The outputs are the HEX displays, the LEDs, outputs related to VGA, outputs related to the CY7C67200 Interface, outputs related to SDRAM Interface and outputs related to music interface.

How are the inputs/outputs processed?

The input system clock is used as the clock input of most of the modules, and the keys are used as reset signal of this project. The inputs and outputs of CY7C67200 are used together with software written in C program to enable USB communication between the keyboard and the FPGA board. The inputs and outputs of the music modules are processed by the I2C protocol. The outputs of the VGA module are sent out of the FPGA board into the VGA display screen as its input signals, then the frames will be shown on the screen according to the VGA protocol. The inputs and outputs related to the SDRAM Interface is used to enable the read and write operations on the SDRAM and further enable the communication between the USB device and the FPGA board according to the interface of SDRAM. The output HEX0, HEX1, HEX2, HEX3 will

be processed by the module HexDriver. The output LEDG and LEDR will be shown by the LEDs on the FPGA board, which will be illustrated in the next section.

What's shown as the outputs?

The output HEX0, HEX1, HEX2, HEX3 is shown hex displayer on the FPGA board as the first two keyboard input of the character. The output LEDG is shown by the green LEDs on the FPGA board as the first two keyboard input of the keyboard-controlled mouse. The output LEDR is shown by the red LEDs on the FPGA board as the choice of block to place. The output related to VGA will be shown on the screen. The output related to the music interface will be shown as the background music.

Model Description

Describe the purpose and operation of major entity, describing its inputs and outputs

There are many entities in our project. We think the major entities in our project are, the top-level entity, the entity for showing pictures on the screen, the entity for the music, the entity for character, the entity for the mouse, the entity for the blocks, and the entity for collision between character and blocks.

Entity for shown pictures:

Purpose: show frames on the screen.

Inputs: current position to plot.

Outputs: which entity to draw for the current position.

Operation: for each position, the coordinates will be put into different modules character, cursor, block, itembar0, itembar1, dirt, stone to judge whether to plot character or cursor or block (and which block) or itembar1 or itembar2 or dirt or stone. Then there will be a signal is_something set to 1 to indicate which we're going to draw. Then the color will be driven using the address calculated by the coordinates and plot. If the color is 0x000000, which is purely black, it means this point actually doesn't belong to the current entity, but the next layer.

Entity for music:

Purpose: loop background music.

Inputs: some audio control signal AUD_BLCK, AUD_ADCDATA, AUD_DACLCK, AUD_ADCLCK.

Outputs: music that can be heard.

Operation: We first store the music in txt file, then use a loop and the audio interface to convert the txt file into the signals AUD_DACDATA, I2C_SDAT, I2C_SCLK, AUD_XCK which can be further processed by the sound speaker to loop our background music.

Entity for character:

Purpose: control the movement of the character.

Inputs: keyboard input for character.

Outputs: current character position.

Operation: We use a clock frame_clk to synthesize the movements between frames. For each frame_clk, we calculate the new position and the new motion, set the boundary situations and renew the position and movement when the system clock rises. We simulated gravity by Newton's law: speed up at each frame_clk with some acceleration.

Entity for cursor:

Purpose: control the movement of the mouse.

Inputs: keyboard input for mouse.

Outputs: current mouse position, whether the left or right button of the simulated mouse is pressed, and whether the roller is used.

Operation: We use a clock frame_clk to synthesize the movements between frames. For each frame_clk, we calculate the new position and the new motion, set the boundary situations and renew the position and movement when the system clock rises. If we detect specific keyboard input, we'll regard this as a left/right press. So as the roller.

Entity for block:

Purpose: record the status of each block: if there is a block and which block there is.

Inputs: keyboard-controlled mouse input.

Outputs: current block status.

Operation: We initialized a module blockReg to record the status of the blocks. Each time when the left/right button of the mouse is pressed, the current position of the mouse will be used to calculate which address of the block should be changed. And then it'll be changed according to the specific operation of the mouse.

Entity for collision between character and blocks:

Purpose: judge if there is collision between the character and the blocks. If so, output signals to stop the character moving in that direction.

Inputs: position of character and the block register.

Outputs: whether and in which direction. the character collides with the block

Operation: We calculate a bounding box for the character, and then use 12 points, 3 for each direction to judge if there is any collision. We calculate if there is block for each point, and if so, a collision signal in the direction of that point will be outputted to the character module to stop further moving in that direction.

Design Procedure / State Diagram / Simulation Waveform

Overview of the design procedure

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

We used lab8, SOC with USB and VGA Interface in SystemVerilog as the foundation of the project.

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

There is no big difference between the hardware usage of lab8 and the hardware usage of our final project, except that we added the music interface. The main difference of this object is about the software: we revised the .qsys file and the software of lab8 into a version that two keycode, each 16 bits will be driven from the USB keyboard. One is keycode for the character, another is keycode for the mouse. Two keyboard input can be driven for each keycode.

By the way, in order to get 16 bits keycode, we changed the char* data structure of keycode_base in the C code into int* data structure. We also add a new base address for the mouse.

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

Actually, at first, we plan to use a PS2 mouse, and we did a lot of research about the PS2 protocol. We even find some code about the PS2 interface which looks promising. However, the PS2 mouse we bought online seems incompatible with the FPGA PS2 protocol, and unfortunately, we failed to implement the PS2 mouse. Instead we used the keyboard to control the mouse. We reviewed our lab8 to revise the software C code so that we can achieve this objective.

As for the background display, we learned more about the VGA protocol, the usage of on-chip memory and the converting from .xnb file into .png file, then into .txt file. We also learned how to calculate which the address of the stored image should be accessed by the current DrawX and DrawY.

As for the music module, we learned a lot about the I2C protocol, which is used to output music signals on FPGA. And we learned about the code given on BlackBoard about the music interface. We tried to implement the music module and it succeeded.

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

The different objective about the mouse is linked together with the project based on the way taught in lab8. We modified the .qsys file using the platform designer to add a new parallel I/O keycode_mouse. We used a module similar to ball.sv provided in lab8 to implement the actions of the mouse by modifying the case statement part.

As for the background display, for each DrawX and DrawY a point will have multiple variables signaling which entity should be drawn at that display point. For each DrawX and DrawY we'll have is_tree, is_mountain, is_character, is_cursor, is_block, is_bar0, is_bar1, is_dirt, is_stone to signal the status of that point, then display the corresponding color.

As for the music, we stored our music in txt file, then used the audio interfaced provided to convert the txt tile into audio signals. Then outputting these signals into the sound speaker would make it.

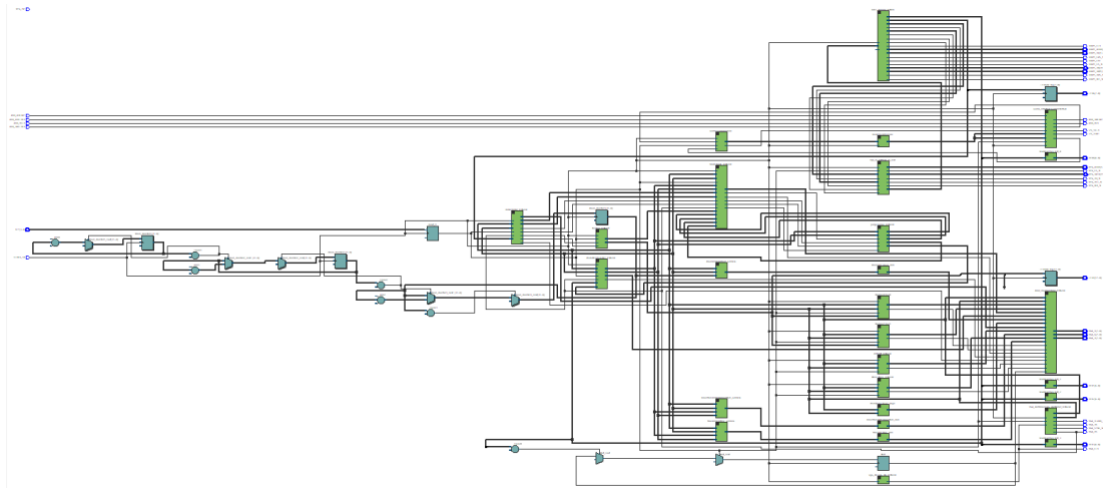
If some sort of serial processing is used, especially for the projects that deals with algorithms, a state machine and a simulation waveform should be included in the report

If a state machine and/or simulation waveform is not used, then the students will need to justify it (e.g., pure graphics, FSM way too simple (game start, game, game end), etc.)

In our project, there are only two states, the state before the game starts and the state of game. Before the game starts, the cover page will show up on the screen, and when the player press space, the game will start, and the player can enjoy the relaxing sandbox game with beautiful scenery. So, there is no need for a state machine.

Block Diagram

The TOP-LEVEL block diagram is as below, just to show the complexity:







Next, we will elaborate each part of the diagram.

Inputs

- OTG_INT 

This is CY7C67200 Interrupt signal. We didn't use it.

- AUD_ADCDAT 
 - AUD_ADCLRCK 
 - AUD_BCLK 
- AUD_DACLK 

These are essential input signals for controlling audio interface.











- KEY[3..0] 

This is for Reset signal. Bit 0 is set up as Reset.

- CLOCK_50 

This is the clock signal that drives the whole circuit. 50 means 50MHz.

Outputs

-  DRAM_CLK
-  DRAM_ADDR[12..0]
-  DRAM_BA[1..0]
-  DRAM_CAS_N
-  DRAM_CKE
-  DRAM_CS_N
-  DRAM_DQ[31..0]
-  DRAM_DQM[3..0]
-  DRAM_RAS_N
-  DRAM_WE_N

These signals act as the SDRAM Interface for Nios II Software.

-  LEDG[7..0]







This is the LED signals, used for visualizing some of the signals inside the circuit.

-  AUD_DACDAT
-  AUD_XCK
-  I2C_SCLK
-  I2C_SDAT

These are output signals from audio interface.

-  HEX0[6..0]

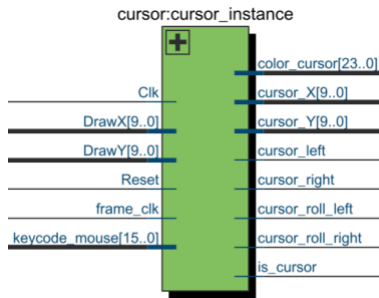
This is the output signal from hex-driver.

-  OTG_ADDR[1..0]
-  OTG_CS_N
-  OTG_DATA[15..0]
-  OTG_RD_N
-  OTG_RST_N
-  OTG_WR_N

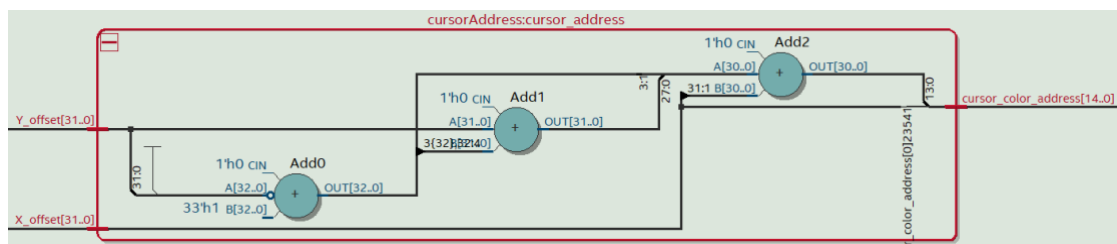
These are signals output from CY7C67200 interface. OTG_DATA is the data on Data bus (16 Bits); OTG_CS_N is chip select signal; OTG_ADDR is address signal (2 bits); OTG_RD_N is write signal, while OTG_WR_N is read signal; OTG_RST_N is reset signal.

Sub-modules

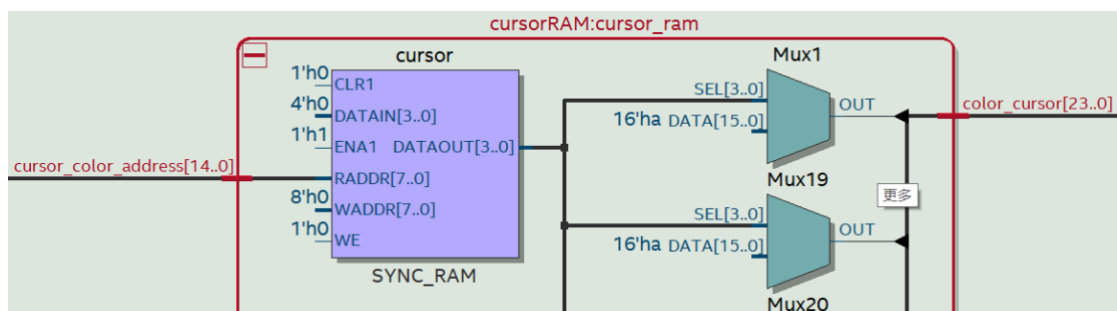
- cursor



The cursor is simulated by keyboard in our project. Therefore, the module requires input “keycode_mouse” to control the movement of cursor on the display. Also, given “DrawX” and “DrawY”, this module should output “is_cursor”, i.e. whether current coordinates belong to the cursor, and “color_cusor”, i.e. the corresponding color of that pixel in order to draw a cursor. “cursor_X” and “cursor_Y” are the coordinates of the upper left point of the cursor, and “cursor_left”, “cursor_right” are the left clicks and right clicks of the simulated cursor; they are used by block module to determine whether a block should be placed or broken. “cursor_rol_left” and “cursor_rol_right” are used by bar module to shift the chosen item.

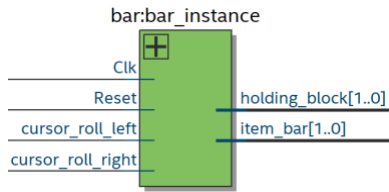


This is the sub-module of cursor module, which is used to calculate the address of the pixel in the cursor region. Next,



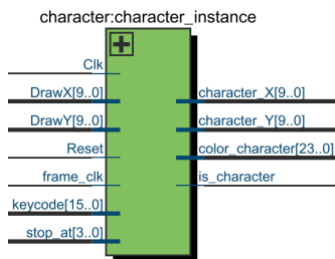
another sub-module of cursor module is used to map the address obtained above to a certain color, so that the cursor module can output the cursor color.

- bar

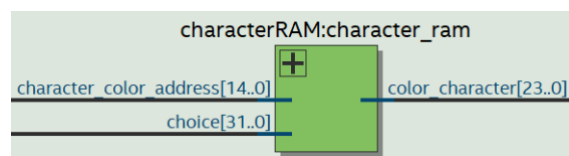
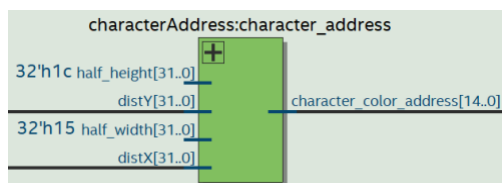


Given signal from cursor that controls rolling left or right, the bar instance should output which item is being held, so “holding_block” is output as a two-bit one-hot vector to choose a particular block, used by block instance; “item_bar” is output as a two-bit one-hot vector to choose a particular bar, used by item_bar0 and item_bar1 instance.

- character

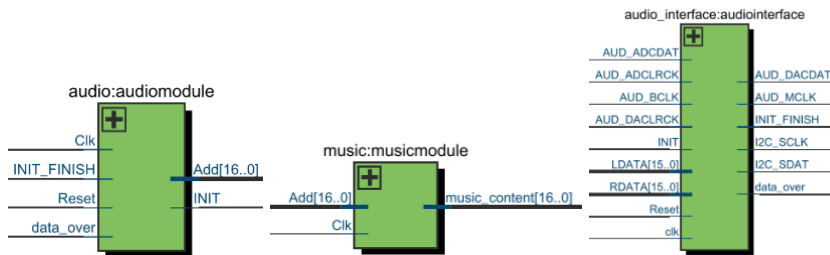


Character module controls the movement of character, just as lab8 does. Given input keycode from keyboard, the module should determine the character’s walking left, walking right, or jumping; “stop_at” is used for determining whether the character has bumped into a block, so that he should stop where there is a block in the way; same as above, given “DrawX” and “DrawY”, the module should output “is_character”, i.e. whether current coordinates belong to the character, and “color_cusor”, i.e. the corresponding color of that pixel in order to draw a character; “character_X” and “character_Y” are the coordinates of the character, used by tree module, mountain module to move with the character, and by block module to determine whether the character is surrounded by blocks or not, and by judge module to determine whether the character can move towards a certain direction or not.



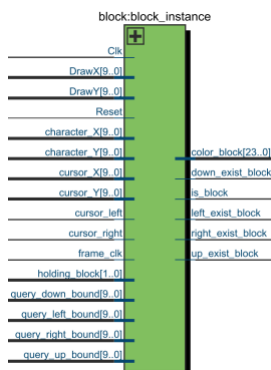
These two sub-modules of character module are of the same usage as those two sub-modules of cursor module, i.e. one for calculating the coloring address and one for outputting the corresponding color of that address.

- audio



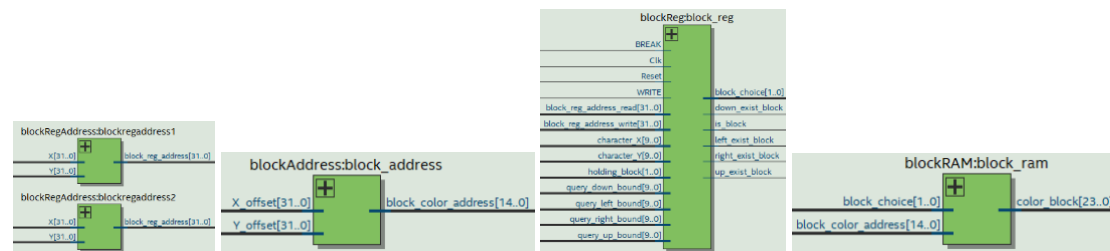
These three modules work together to realize the audio module. It should be noticed that this is a circular logic. “INIT_FINISH” is output from `audio_interface` module to signify that initialization is finished; “data_over” is output from `audio_interface` module to signify loop playback of the music. Audio module constantly output “Add”, which means looping the address; and “INIT” means initialized. Music module read the address and output a particular music content from music memory. `Audio_interface` module read the music content as “LDATA” and “RDATA” to output to the outside world.

- block



Given “DrawX” and “DrawY”, the block module should determine whether the location belongs to a block (“is_block”) and what color should be drawn on the display (“color_block”). As for how to determine that color, the module should have access to three sources of information: which block the character is holding (“holding_block”), place a block or break a block (“cursor_left”, “cursor_right”), where to place a block or break a block (“cursor_X”, “cursor_Y”). Plus, the module should have the ability to determine which directions are blocked by blocks at a certain position of character. To this end, the module should know four bounding lines (query_up_bound, etc.), which encloses the character, and the character’s position (“character_X”, “character_Y”). Given these signals, the module can output

four “yes” or “no” signals indicating the blocks blocking the character (“up_exist_block”, etc.).



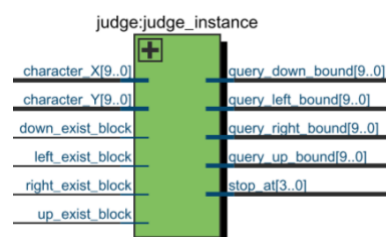
block_address and block_ram work in the similar way as above. block_reg module stores the block matrix. Each entry stores a block value: “00” means there is no block; “01” means the block is a dirt block; “10” means the block is a grass block; “11” means the block is a stone block. blockRegAddress1 module is for reading a block information from the block matrix, and blockRegAddress2 module is for writing a block information to the block matrix.

● backgrounds



All the combinations of “address” and “ram” are for drawing some color at (DrawX, DrawY), as stated above. “address” module calculates the address in the ram, and “ram” module output the color it stores at the address.

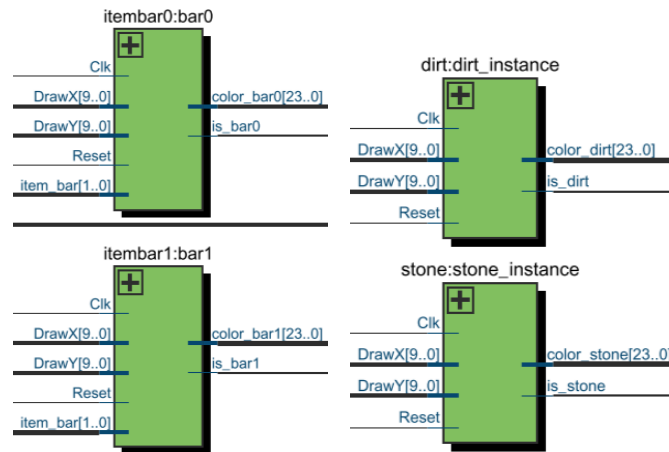
● judge



This module is used to determine to which directions the character (“character_X”, “character_Y”) cannot proceed. “stop_at” is a 4-bit multiple-hot vector, which

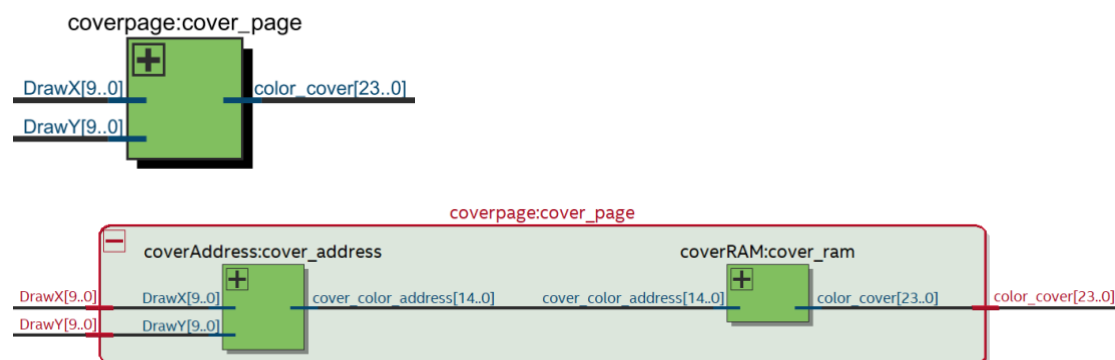
indicates the directions that are blocked. At the four “exist” signals are used to calculate the vector stated above.

- item bar



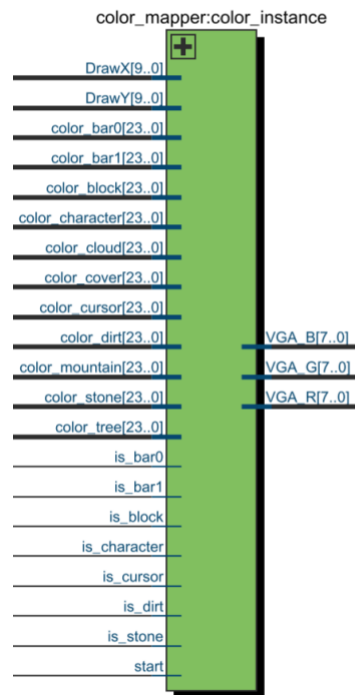
itembar0, itembar1 modules is for the two choices of the item. If “item_bar” is “01”, then itembar1 is chosen; if “item_bar” is “10”, then itembar0 is chosen. The choice of the bar determines the output color of the bar, because if we choose an item on the bar the surroundings should be gold. dirt, stone modules work together with item bars. They should lie inside the item bar. Same as above, each of them have a combination of “address” and “ram” in order to output the correct colors for the icon.

- cover page



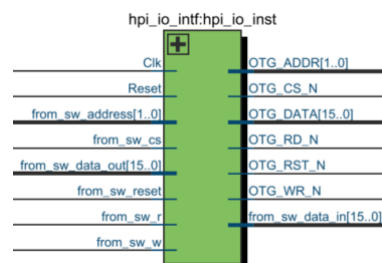
The cover page should be displayed as soon as we start the game. Same as above, it has a combination of “address” and “ram” in order to output the correct colors for the picture of the cover page.

- color mapper



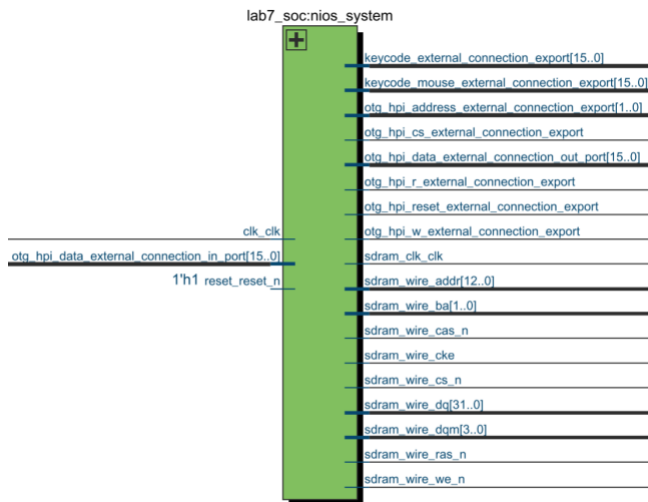
Adapted from Lab8. We added all the things we need to draw by connecting the signals of “color” and “is”.

- `hpi_io_intf`



Interface between NIOS II and EZ-OTG chip. Same as Lab8.

- `SOC`



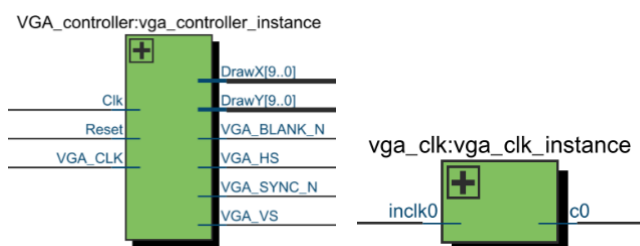
Same as Lab8, with the only difference lying in a “keycode_mouse_extrenal_connection_export” signal. This is for simulating mouse on the keyboard.

- hex-driver



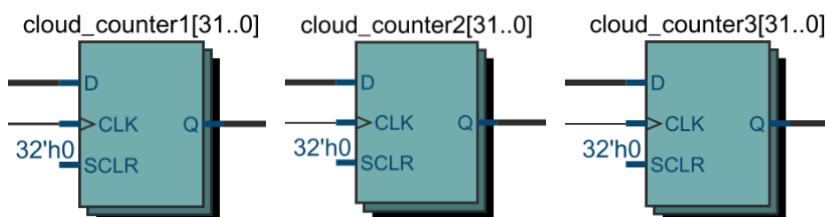
Same as previous labs.

- VGA

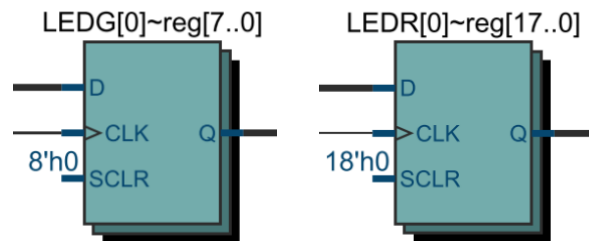


Same as Lab8.

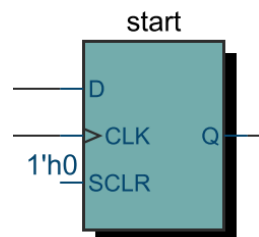
Other Logics



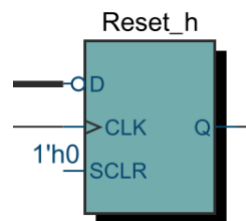
The three counters are used to create the animation of clouds. The clouds should circularly move from left to right, with time interval set to a certain value according to the clock frequency.



These are LED signals.



Once “space” is pressed, the game is started (the cover page is passed).



Reset signal.

Core SV Codes for Controlling

Drawings

All the drawings need two modules: “address”, which calculates the address of a pixel in the ram, and “ram”, which stores the color of that pixel at the address. Let’s take trees as example.

```

1  module treeAddress (input [9:0] DrawX, DrawY,
2                      input [9:0] character_X, character_Y,
3                      output logic [14:0] tree_color_address);
4                      assign tree_color_address = (DrawY / 4 + character_Y / 32) * 200 + DrawX/4 + character_X / 32;
5  endmodule
6
7  module treeRAM
8  (
9      input [14:0] tree_color_address,
10     output logic [23:0] color_tree
11 );
12     logic [3:0] mem[0: 29999];
13     logic [23:0] colors_tree [2:0];
14     assign colors_tree[0] = 24'h000000;
15     assign colors_tree[1] = 24'h286f58;
16     assign colors_tree[2] = 24'h289260;
17     assign color_tree = colors_tree[mem[tree_color_address]];
18
19     initial
20     begin
21         $readmemh("background/trees_200_150.txt", mem);
22     end
23
24 endmodule

```

Given “DrawX” and “DrawY”, address module first calculates the color address. Note that this address means the address of a pixel IN THE TREE PICTURE, but not in the whole display range. Then, the ram module uses the color address to locate a value in the RAM, which is the color number of that pixel. And the color number is mapped to an RGB value using the assignments.

Blocks

```

module blockReg
(
    input logic Clk, Reset,

    input int block_reg_address_read,
    input int block_reg_address_write,

    // read from block_matrix
    output logic is_block,
    output logic [1:0] block_choice,

    // write to block_matrix
    input logic WRITE,
    input logic BREAK,

    input logic [9:0] character_X, character_Y,
    input logic [9:0] query_left_bound, query_right_bound, query_up_bound, query_down_bound,
    output logic left_exist_block, right_exist_block, up_exist_block, down_exist_block,
    input logic [1:0] holding_block
);

```

The information of blocks is stored in a register file as below.

```
logic [1199:0][1:0] matrix; // 30 * 40
```

Since the display is of size 640 * 480, and our blocks are of size 16 * 16, there should be 40 * 30 block positions totally. For every position, a state code is used to represent

the information: 00 means empty; 01 means dirt block; 10 means grass block; and 11 means stone block.

```
initial begin
    for (int i = 0; i < 30; i++)
        begin
            for (int j = 0; j < 40; j++)
                begin
                    if (i < 24)
                        matrix[i * 40 + j] <= 2'b00; // nothing
                    else if (i == 24)
                        matrix[i * 40 + j] <= 2'b10; // grass
                    else
                        matrix[i * 40 + j] <= 2'b01; // dirt
                end
            end
        end
    end
```

Initially, we hope that the layers at the bottom most should be dirt, and the upper layer of the dirt should be grass. Then there is nothing above.

```
else if (WRITE)
    matrix[block_reg_address_write] <= holding_block;
else if (BREAK)
    matrix[block_reg_address_write] <= 2'b00;
```

If the user press the right click of the simulated mouse, a block should be placed where the cursor is. That block is represented by “holding_block”, and that location of the cursor is represented by “block_reg_address_write”. Else, the user might want to break a block, which means he should press the left click.

```
always_comb
begin
    is_block = 0;
    block_choice = 2'b00;
    if ( matrix[block_reg_address_read] != 0 )
        is_block = 1;
    block_choice = matrix[block_reg_address_read];
end
```

When outputting the block information, the module would output whether there is a block at a certain address, and if there exists a block it will also output the block type.

```

// test left
left_exist_block = 0;

// test_point1
point_X = character_X - 15;
point_Y = character_Y;
query_address = point_Y / 16 * 40 + point_X / 16;
if (matrix[query_address] != 0)
    left_exist_block = 1;

// test_point2
point_X = character_X - 15;
point_Y = character_Y - 14;
query_address = point_Y / 16 * 40 + point_X / 16;
if (matrix[query_address] != 0)
    left_exist_block = 1;

// test_point3
point_X = character_X - 15;
point_Y = character_Y + 14;
query_address = point_Y / 16 * 40 + point_X / 16;
if (matrix[query_address] != 0)
    left_exist_block = 1;

// test up
up_exist_block = 0;

// test_point1
point_X = character_X;
point_Y = query_up_bound;
query_address = point_Y / 16 * 40 + point_X / 16;
if (matrix[query_address] != 0)
    up_exist_block = 1;

// test_point2
point_X = character_X - 14;
point_Y = query_up_bound;
query_address = point_Y / 16 * 40 + point_X / 16;
if (matrix[query_address] != 0)
    up_exist_block = 1;

// test_point3
point_X = character_X + 14;
point_Y = query_up_bound;
query_address = point_Y / 16 * 40 + point_X / 16;
if (matrix[query_address] != 0)
    up_exist_block = 1;

// test right
right_exist_block = 0;

// test_point1
point_X = character_X + 15;
point_Y = character_Y;
query_address = point_Y / 16 * 40 + point_X / 16;
if (matrix[query_address] != 0)
    right_exist_block = 1;

// test_point2
point_X = character_X + 15;
point_Y = character_Y - 14;
query_address = point_Y / 16 * 40 + point_X / 16;
if (matrix[query_address] != 0)
    right_exist_block = 1;

// test_point3
point_X = character_X + 15;
point_Y = character_Y + 14;
query_address = point_Y / 16 * 40 + point_X / 16;
if (matrix[query_address] != 0)
    right_exist_block = 1;

// test down
down_exist_block = 0;

// test_point1
point_X = character_X;
point_Y = query_down_bound;
query_address = point_Y / 16 * 40 + point_X / 16;
if (matrix[query_address] != 0)
    down_exist_block = 1;

// test_point2
point_X = character_X - 14;
point_Y = query_down_bound;
query_address = point_Y / 16 * 40 + point_X / 16;
if (matrix[query_address] != 0)
    down_exist_block = 1;

// test_point3
point_X = character_X + 14;
point_Y = query_down_bound;
query_address = point_Y / 16 * 40 + point_X / 16;
if (matrix[query_address] != 0)
    down_exist_block = 1;

```

These four tests are for detecting blocks in the surroundings of the character. Take “test left” for example. We have three test points, scanning the left side of the character from up to down. Once there exists a block at the left side, i.e. `matrix[query_address]` is not 00, the “left_exist_block” flag should be turned on.

Character

```

parameter [9:0] ball_X_Start = 10'd320; // Start position on the X axis
parameter [9:0] ball_Y_Start = 10'd320; // Start position on the Y axis
parameter [9:0] ball_X_Min = 10'd0; // Leftmost point on the X axis
parameter [9:0] ball_X_Max = 10'd639; // Rightmost point on the X axis
parameter [9:0] ball_Y_Min = 10'd10; // Topmost point on the Y axis
parameter [9:0] ball_Y_Max = 10'd449; // Bottommost point on the Y axis
parameter [9:0] ball_X_Step = 10'd2; // Step size on the X axis
parameter [9:0] ball_Y_Step = 10'd4; // Step size on the Y axis
parameter [9:0] ball_Height = 10'd56; // ball height
parameter [9:0] ball_Width = 10'd42; // ball width
parameter [9:0] gravity = 10'd2;

```

First there are some basic parameters. It should be noticed that we added “gravity” to simulate the physics in the real world to make the game look more authentic.

```
initial
begin
    ball_X_Pos      <= ball_X_Start;
    ball_Y_Pos      <= ball_Y_Start;
    ball_X_Motion   <= 10'd0;
    ball_Y_Motion   <= 10'd0;
    counter         <= 0;
    counter_anime   <= 0;
    counter_anime2  <= 0;
    choice          <= 3;    // default: facing right, standing still
    left            <= 0;
    right           <= 0;
    side            <= 1;
end
```

Initially, the character is placed at the bottom middle position of the display. “left” and “right” record walking towards which direction, and “side” records facing with direction while either walking or standing still. Both the X-axis speed and Y-axis speed are set to be zero. He should be facing right and standing still.

```
// counter cannot exceed:
counter_next = counter + 1;
if (counter == counter_max)
    counter_next = 0;

counter_anime_next = counter_anime + 1;
counter_anime2_next = counter_anime2;
if (counter_anime == 5000)
begin
    counter_anime_next = 0;
    counter_anime2_next = counter_anime2 + 1;
end
if (counter_anime2 == 5000)
    counter_anime2_next = 0;
```

We used three counters to realize the walking animation of the character. All three counters are synchronized with the clock, but they have different jobs. Counter is used to control the acceleration of falling of the character:

```
if (counter == counter_max)
    ball_Y_Motion_in = ball_Y_Motion + gravity;
```

without the judging condition, the character will fall too fast since the clock frequency is high; so we make the character accelerate a little for a cycle of several clocks. counter_anime and counter_anime2 are used together to create a very long cycle, since it is “int” type and the value cannot exceed 65535. The very long cycle is used to slow down the shifts of the frames of the walking character:

```

if (left)
begin
    // direction control
    if (counter_anime2 < 2500)
        choice_next = 1;
    else
        choice_next = 2;
end
else if (right)
begin
    // direction control
    if (counter_anime2 < 2500)
        choice_next = 4;
    else
        choice_next = 5;
end

```

“choice” is the action of the character. 1 means facing right, taking the first step; 2 means facing right, taking the second step; 4 means facing left, taking the first step; 5 means facing left, taking the second step. With the choice being shifted, different picture of the character would be displayed, creating the animation of the character’s walking.

```

else
begin
    // direction control
    if (side == 0)
        choice_next = 0;
    else
        choice_next = 3;
end

```

What’s more, 0 means facing right, standing still; 3 means facing left, standing still. If the character is not walking, he should face the same direction as previously does and stand still.

```

// check boundaries
// left boundary
if ((ball_X_Pos < ball_X_Max/2 + half_width) && (ball_X_Pos_in < half_width) )
begin
    ball_X_Pos_in = ball_X_Min + half_width;
    ball_X_Motion_in = 0;
end
// right boundary
if ((ball_X_Pos + half_width > ball_X_Max/2) && (ball_X_Pos_in + half_width > ball_X_Max))
begin
    ball_X_Pos_in = ball_X_Max - half_width;
    ball_X_Motion_in = 0;
end
// up boundary
if ((ball_Y_Pos < ball_Y_Max/2 + half_height) && (ball_Y_Pos_in < half_height))
begin
    ball_Y_Pos_in = ball_Y_Min + half_height;
    ball_Y_Motion_in = 0;
end
// down boundary
if ((ball_Y_Pos + half_height > ball_Y_Max/2) && (ball_Y_Pos_in + half_height > ball_Y_Max))
begin
    ball_Y_Pos_in = ball_Y_Max - half_height;
    ball_Y_Motion_in = 0;
end
end

```

When the character reaches the any of the four boundaries, he should stop. This part is basically the same as Lab8.

```

// left_collision
if ( stop_at[3] == 1 )
begin
    if ( (ball_X_Motion_in[9] == 1) || (ball_X_Motion_in == 9'b0) )
    begin
        ball_X_Pos_in = ((character_X - horizontal_space)/ 16 * 16) + horizontal_space + 15;
        ball_X_Motion_in = 0;
    end
end
// right_collision
if ( stop_at[2] == 1 )
begin
    if (ball_X_Motion_in[9] == 0)
    begin
        ball_X_Pos_in = ((character_X + horizontal_space)/ 16 * 16) - horizontal_space;
        ball_X_Motion_in = 0;
    end
end
end
// up_collision
if ( stop_at[1] == 1 )
begin
    if ( (ball_Y_Motion_in[9] == 1) || (ball_Y_Motion_in == 9'b0) )
    begin
        ball_Y_Pos_in = ((character_Y - half_height)/ 16 * 16) + half_height + 15;
        ball_Y_Motion_in = 0;
    end
end
// down_collision
if ( stop_at[0] == 1 )
begin
    if (ball_Y_Motion_in[9] == 0)
    begin
        ball_Y_Pos_in = ((character_Y + half_height)/ 16 * 16) - half_height;
        ball_Y_Motion_in = 0;
    end
end
end

```

Remember that “stop_at” is a multiple-hot vector that records the directions that are blocked by blocks. Under the conditions that the character is moving against the

directions indicated by “stop_at”, the corresponding speed should be set to zero and the next position will be adjusted.

```
case (keycode[15:0])  
    // jump while right  
    16'h2c07:  
    begin  
        side_next = 1;  
        right_next = 1;  
        left_next = 0;  
        // motion control  
        ball_X_Motion_in = ball_X_Step;  
        ball_Y_Motion_in = ~(ball_Y_Step) + 1'b1;  
    end  
    16'h072c:  
    begin  
        side_next = 1;  
        right_next = 1;  
        left_next = 0;  
        // motion control  
        ball_X_Motion_in = ball_X_Step;  
        ball_Y_Motion_in = ~(ball_Y_Step) + 1'b1;  
    end
```

Our design is compatible with multiple keys. That means, the character can walk and jump at the same time. Take jumping while walking right as example. If the keycode combination detected is 16'h2c07, it means “D” and “space” are pressed at the same time. So the next X-axis speed should be the X speed constant, and the next side should be right, and the right flag should be turned on, while the left flag should be turned off; as for Y axis, the speed should be the Y speed constant and upward.

Conclusion

Deep Bugs

1. When displaying trees, the output image was horizontally torn. It turned out that the indexing of X and Y coordinates were wrongly calculated by multiplying 200 first and dividing by 4 next. The correct version is:

```
tree_color_address = (DrawY / 4 + character_Y / 32) * 200 + DrawX/4 + character_X / 32;
```

2. The character did not stop when reaching the bottom. It was because the boundary check was at the wrong place. After every keypress, the check program should be processed. So checking boundaries should follow the case selection of keycode.
3. The input keycode cannot be read as 16-bit value. This is because when declaring the pointer to the keycode content, we write the pointer as “char *”, which points to 8-bit memory content. This is fixed by changing the declaration to “int *”, which points to 16-bit memory content.

Accomplishments

We implemented a simple version of Terraria on the FPGA board. Although we just realized a small portion of the whole game, the player of our game can still get a similar experience as playing real Terraria.