# ECE385

Fall 2020

Final Project

# Bongo Touch

Li Taoran: 3180110750

Yuan Xinkai: 3180110983

LA3/2020.10.17

Yin Haocheng

1. Introduction

   In the final project, we design a game named Bongo Touch, which is a 2D percussion music game. In the hardware part, we use System Bus, on-chip-memory, SDRAM in our SoC and implement control modules, music modules, display modules and so on. We use NIOS II CPU to interface with USB keyboard as we have done in lab 8. In our software part, we modified the C code about USB and IO to accept 4 keycodes input at the same time.

   In the game, the player should press "S" or "F" if there is a red note coming and press "A" or "G" if there is a blue note coming. The player should press the keyboard as the same time with the flowing note reaching the destination. The note will become smiling if the player press the right keyboard at the right time and then he will be awarded one score. The total score will display at the left of the destination. Meanwhile, we have two patterns, which are 1P (one player) and 2P (two players). The second player should press "↓" or "1" if there is a red note coming and press "←" or "2" if there is a blue note coming. There are two notes coming at the same time separately for each player and the scores will be calculated and display separately.

2. Written Description

   1) Description of the overview of the circuit

      a. In our SoC part, we use NIOS II processor of economic pattern. The processor will perform the instructions in our hardware part of different modules and interface with USB keyboard. The SDRAM is used with NIOS II to store the C code. The on-chip-memory are used to store different pictures, background music and other data. We also use JTAG UART to use the terminal of the host computer (the one running eclipse) to communicate with the NIOS II (using print and scan statements in c). we also have Parallel Input/Output of keycode, address, data, read, write, chip select, and reset signal. In our hardware part, we implement the control logic of the note, which will control the running of notes, judge whether the player should be awarder and calculate and display the scores. The background music is also controlled in this part. Meanwhile, all the pictures is read from OCM in this part and display through interface and VGA controller. In our software part, we used IO_read, IO_write, USBRead and USBWrite, which could read at most 4 keyboard inputs at the same time and used to control the different options of the game.

      b. The basic function is to implement the rhythm game with the pathways, notes and audio supports to work as a complete game part, use keyboard as player's input and display them using monitor and audio module. About the additional functions, we also made a main menu to choose patterns of 1P (one player) and 2P (two players). There is also game scene like the background and the pathways, the notes will be graphs with special effects added instead of pixel blocks. There will be also PvP mode which supports two players to play for the same song at the same time.

   2) Descriptions of general flow of the circuit

      a.

```
module toplevel(
            input              CLOCK_50,
            input        [3:0] KEY,          //bit 0 is set up as Reset
            output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7,

            output logic [19:0] SRAM_ADDR,
            inout wire   [15:0] SRAM_DQ,
            output logic        SRAM_UB_N,
                                SRAM_LB_N,
                                SRAM_CE_N,
                                SRAM_OE_N,
                                SRAM_WE_N,

             // VGA Interface
             output logic [7:0]  VGA_R,         //VGA Red
                                 VGA_G,         //VGA Green
                                 VGA_B,         //VGA Blue
             output logic        VGA_CLK,       //VGA Clock
                                 VGA_SYNC_N,    //VGA Sync signal
                                 VGA_BLANK_N,   //VGA Blank signal
                                 VGA_VS,        //VGA vertical sync signal
                                 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        OTG_CS_N,      //CY7C67200 Chip Select
                                 OTG_RD_N,      //CY7C67200 Write
                                 OTG_WR_N,      //CY7C67200 Read
                                 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        DRAM_RAS_N,    //SDRAM Row Address Strobe
                                 DRAM_CAS_N,    //SDRAM Column Address Strobe
                                 DRAM_CKE,      //SDRAM Clock Enable
                                 DRAM_WE_N,     //SDRAM Write Enable
                                 DRAM_CS_N,     //SDRAM Chip Select
                                 DRAM_CLK,      //SDRAM Clock
            input        [17:0] SW,
            output logic [17:0] LEDR,
            inout  wire         I2C_SDAT,
            output logic        I2C_SCLK, AUD_XCK, AUD_BCLK, AUD_DACLRCK, AUD_DACDAT

                  );
```

CLOCK_50: An input clock of 50 MHz from SoC

KEY: Inputs from the keys of FPGA

HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7: Outputs to display on the LED screen of FPGA

SRAM_ADDR: Input address of NIOS II logic

SRAM_DQ: Inout wire of NIOS II

SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N, SRAM_WE_N: Output control signal of SRAM of NIOS II

VGA_R: VGA Red

VGA_G: VGA Green

VGA_B: VGA Blue

VGA_CLK: VGA Clock

VGA_SYNC_N: VGA Sync signal

VGA_BLANK_ VGA: Blank signal

VGA_VS VGA: Vertical sync signal

VGA_HS VGA: Horizontal sync signal

OTG_DATA: CY7C67200 Data bus 16 Bits
OTG_ADDR: CY7C67200 Address 2 Bits
OTG_CS_N: CY7C67200 Chip Select
OTG_RD_N: CY7C67200 Write
OTG_WR_N: CY7C67200 Read
OTG_RST_N: CY7C67200 Reset
OTG_INT: CY7C67200 Interrupt
DRAM_ADDR: SDRAM Address 13 Bits
DRAM_DQ: SDRAM Data 32 Bits
DRAM_BA: SDRAM Bank Address 2 Bits
DRAM_DQM: SDRAM Data Mast 4 Bits
DRAM_RAS_N: SDRAM Row Address Strobe
DRAM_CAS_N: SDRAM Column Address Strobe
DRAM_CKE: SDRAM Clock Enable
DRAM_WE_N: SDRAM Write Enable
DRAM_CS_N: SDRAM Chip Select
DRAM_CLK: SDRAM Clock
SW: Input switch signal from FPGA
LEDR: Output signal controlling LED light of FPGA
I2C_SDAT: Inout wire of simple I2C driver
I2C_SCLK, AUD_XCK, AUD_BCLK, AUD_DACLRCK: Different clock controlling the output of music
AUD_DACDAT: Input data of the music module

b. For the SRAM part, the program write or read data from the given address and give output to the output signals and hpi_to_intf interface which connect the NIOS II and EZ-OTG chip.

   For the EZ-OTG part, the input signal is also connected with the hpi_to_intf interface and processed by the EZ-OTG part, giving output to the interface.

   For SDRAM interface for NIOS II software, all the inputs and outputs are processed and generated by the NIOS II processor.

   For the music part, we load our music to OCM and run the music module to play music. Subject to the storage capacity of OCM, we just play the 20 second background music on a loop.

c. Overview

   In this final project, we use the same input and output logic of lab 8, which use keyboard to control the running of our game. The color mapper paints the image onto the VGA with a fixed refresh rate and decide which color to display on the screen according to the instruction of green, red or blue and display on the correct position according to DrawX and DrawY signal and correct priority. The note will be controlled by the sprite and note module, which controls the rhythm and running separately. The NIOS II read the keyboard and give its output value to determine whether the play give the correct keycode at the correct time or not. We also have scores, which will be calculated in the scoring module and display on the screen by color mapper. If the player makes correct decision, the

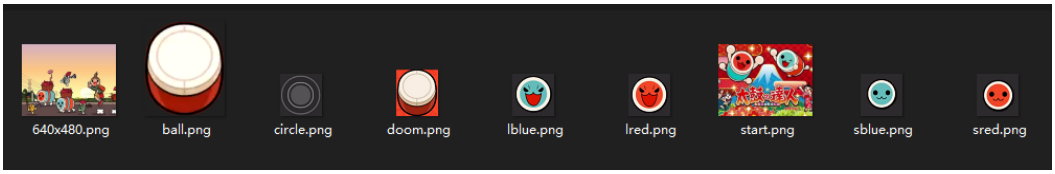
note will become a smile through the color mapper.

① Keyboard

In order to make PvP mode, we modify the C code in lab 8. We add more pointers and peripheral unit in the .c program and make more assignments in the .h program. Our program could read at most four keycode from our keyboard at the same time.

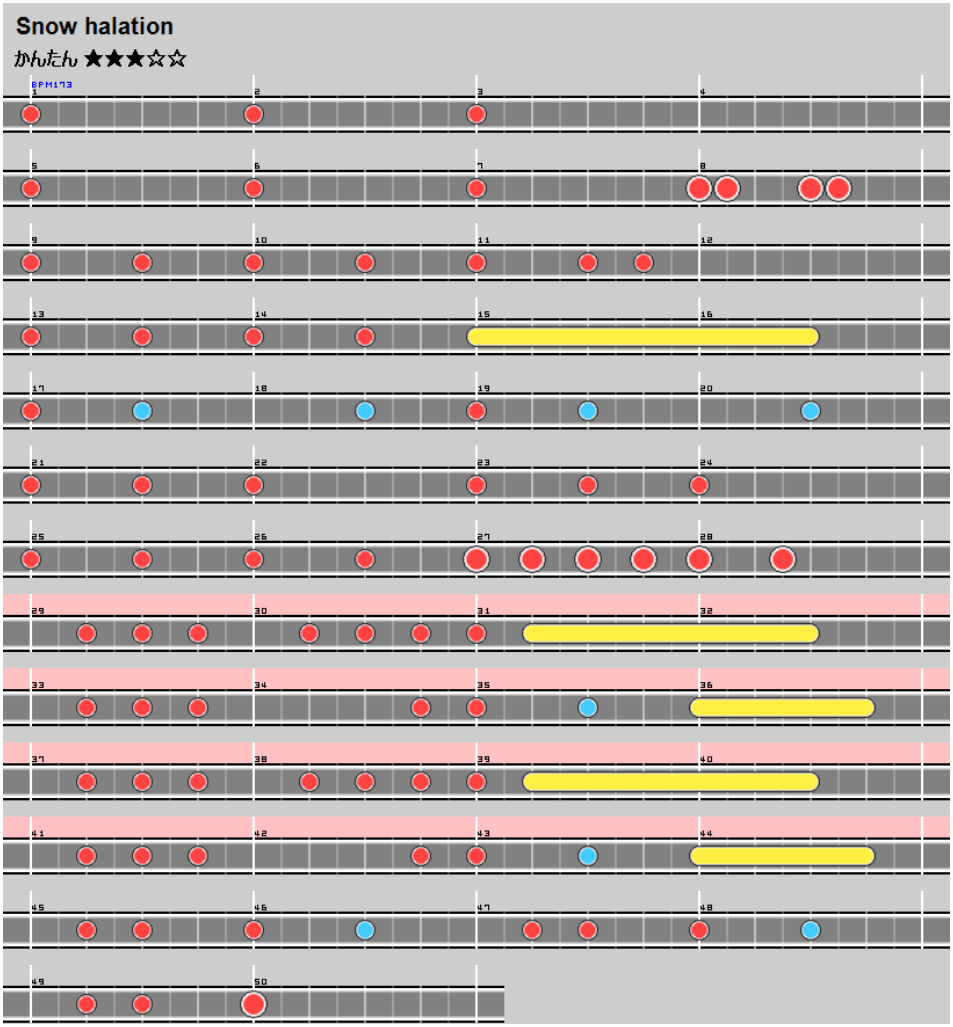
② Picture Drawing

In our project, figures are stored in On-chip Memory (OCM) in the form of 32-bit hex numbers. The use of OCM and the change from pictures to hex is implemented clearly on ECE385 Helper Tools. The Color mapper receive the 32-bit hex for RGB and just output as the VGA_R, VGA_G, VGA_B. And the module of color-mapper will determine when and where the pixel will display which 32-bit hex of the figures. This is the process of the picture drawing.




③ Map Design

This is the reference of the music map. We translate the map to a txt file which shows the state of the map. The file will be loaded in the note_modules.sv, and we implement a clock which will be active every half beat. A half beat is just the one gird of the map,  like this. And the .txt file store 0 if no note should be generated, and it store X if the Xth instantiated note should be generated. In this way, the map will be generated fast, and cost very small space.




④ Score

This figure is used as the reference of our type of the numbers and letters. We use 1-bit number for each pixel to store all the numbers and letters. The size is 11*15. The blank space is filled with 0, and the letter/number space is filled with 1. When displaying in the color-mapper, we can just judge the space to display the number/letter in whatever color we want.


⑤ Game State

There are several special states we need to handle with in the game: starting interface, mode-select menu, and game interface. First, the player will see the starting interface of the game name. Then if the player presses the SPACE button, the state will change to the mode-select menu. The player can press "↑" (up) and "↓" (down) button to choose the 1p/2p mode, and the selected-mode option block will be highlighted. After determining the mode, if the player press "Enter", the game will start at once. The game interface is displayed, the music starts to play, and the notes start to fall. If the player press "Reset", the game will back to the starting interface and the player can play the game again.

⑥ Background Music

Our game has a background music of 20 seconds, which will automatically play as soon as the game starts, and the note is coming. We use OCM to store the music used here. We add a one port ROM with type M9K and 18 bits output with 65536 words memory in our IP Component to store the 20 second music. In order to store a longer music, we decrease the image quality of our pictures to save space in OCM. Meanwhile, we add a parallel clock USB_Clock_PLL with 50MHz input and 12MHz output clock1 and 0.04803 MHz output clock2 to control the playing of music, which is calculated according to the 16000Hz sampling frequency. We also use a python program to convert the .wav music file to a .hex or .mif or .txt file, which could be stored in the memory. We have a music top level module, which is a state machine to control the program read the content in the OCM line by line and play the music out. It also controls the music to play another time as soon as it finished. The background music component is relatively independent of other part of this project. Therefore, we just add it at the end of our project.

3. Module Description

Module: note_module.sv (RNG2)

Input:

Clk: The Clk signal CLOCK_50

Frame_clk: The Screen refreshing clock, VGA_VS

Reset: The reset signal with the 0 button.

Start_game: The 1-bit signal which is 0 when the game is not start or Reset is active and becomes 1(active) when the game starts (when the music goes on and the notes becomes to fall).

Output:

G_activate,r_activate,y_activate,b_activate: The 1-bit signals which controls the notes generation. If g_activate becomes 1, then the sprite module will generate a small blue note at the right of the screen and generate the second until g_acticate change to 0 and becomes 1 again.

Description:

This module controls the generation of the notes. One counter is implemented in this module which gains frame_clk (60Hz) and counts 10 as one-time cycle (T = 1/6s). The

music's BPM (Beat Per Minute) is 173, and we use this counter as a clock with T = 1/6s to count as about half beat. We use one .txt file filled with 4-bit signals to record the information of notes. Every time cycle(1/6s), the module will read one address from the file and a mux is used to select which note should be generated at this half beat, and the corresponding 1-bit output will be active then.

Module: sprites.sv (sprites_yellow, _blue, _green, _red)
Input:
Clk: CLOCK_50 signal
Reset: the reset signal
Frame_clk: the screen refreshing clock, VGA_VS
DrawX, DrawY: The 10-bit signals representing the current position of the X-axis and Y-axis of the whole VGA scanner. 10-bit is used to represents 0-639 and 0-479 for X and Y individually.
Rng_: The 1-bit signal which is the _activate signal. This controls the generation of notes mentioned above in the note_module.sv.
Output:
Is_sprite_: This is the 1-bit output signal which controls the appearance of the note. It connects with color-mapper, and if is_sprite_ is 0, the note will not be in the screen; while the note will be in the screen at the position (_x_pos, _y_pos) if is_sprite_ is 1.
_x_pos, _y_pos: These are the 10-bit signals which represents the current position (_x_pos, _y_pos) of the note. 10-bit is used to represents 0-639 and 0-479 for x and y individually.
Description:
This module updates the position of the note and tells the color-mapper if the note should be displayed in the screen. The position of the note will be initially at (639, y), (y is different in 1p/2p mode, while the y position is fixed during game since the note is falling from the rightmost to the left most of the screen). If input signal rng becomes 1, _x_pos will decrease with a rate of 3 pixels every run circle. When the note falls to the left most of the screen, the decrease rate will be reset to 0, and the position will be back to (639,y). In this case, the module is ready for the rng to be 1 again to update the next note's position. If the DrawX and DrawY match the _x_pos and _y_pos, is_sprite_ will be active to tell the color mapper to display this note.

Module: Scoring.sv
Input:
Clk: CLOCK_50 signal
Reset: the reset signal
Keycode: the 32-bit keyboard signal. We can afford 4 different keyboard inputs at the same time, each with 8-bit signal. The first pressed key is stored in [7:0], and the second is in [15:8], similar for the rest.
_x_pos, _y_pos: this is the current notes' X position and Y position. It's 10-bit length, which ranges (0,639) for x, and (0,479) for y.
Output:
Score_1, score_2: it's the score for two players. It's 8-bit length, and the maximum score

is less than 200.
Get_(r,b,y,g): it's the 1-bit signal which determines whether the player press the note at the correct time and tells the color mapper to hide the notes if the player press the note at the correct time.
Description: This module is the score counter. It first get the position of the note, and then check if it's in the correct time to press the key. If the keycode matches the note, and the press time is correct, then the score of that player will add 1, and the note will be disappeared from the track to show the player that he hit the note and get the point, while if he does not hit the note, the note will just pass away and continue falling.

Module: color_mapper
Input:
Is_sprite_: This is the 1-bit output signal which controls the appearance of the note. if is_sprite_ is 0, the note will not be in the screen; while the note will be in the screen at the position (_x_pos, _y_pos) if is_sprite_ is 1.
Clk: CLOCK_50 signal
Reset: the reset signal
Frame_clk: the screen refreshing clock, VGA_VS
Get_: it's the 1-bit signal which tells the color mapper to hide the notes if the player presses the note at the correct time.
Keycode: the 32-bit keyboard signal. We can afford 4 different keyboard inputs at the same time, each with 8-bit signal. The first pressed key is stored in [7:0], and the second is in [15:8], similar for the rest.
Score_1, _2: it's the score for two players. It's 8-bit length, and the maximum score is less than 200.
DrawX, DrawY: The 10-bit signals representing the current position of the X-axis and Y-axis of the whole VGA scanner. 10-bit is used to represents 0-639 and 0-479 for X and Y individually.
_x_pos, _y_pos: this is the current notes' X position and Y position. It's 10-bit length, which ranges (0,639) for x, and (0,479) for y.
Start_game: The 1-bit signal which is 0 when the game is not start or Reset is active and becomes 1(active) when the game starts (when the music goes on and the notes becomes to fall).
Output:
VGA_R, VGA_G, VGA_B: These are VGA RGB outputs.
Description:
This module controls all the graphic output of the game. It gets all the figures and files in the on-chip memory and determine the correct time to display these elements. There is a state machine in the module. It displays the game start menu, mode select menu, and the game interface. The static elements' positions are determined, and the state machine determine when they are displayed. The input signals and the state machine both determine the motioning elements. Each pixel will be drawn by the output VGA_R, VGA _G, VGA _B.

Module: Sound_Top

```
module Sound_Top (
    input clk,reset,
    inout SDIN,
    output SCLK,USB_clk,BCLK,
    output reg DAC_LR_CLK,
    output DAC_DATA,
    output [2:0] ACK_LEDR
);
```

Description: This is the top-level module about music playing.

Purpose: This module is a state machine which controls the playing of music. It read music data from OCM line by line and play it out according to the output of state machine through the protocol. If the music has been finished, it will automatically jump to the first level and run another time to realize playing on loop.

Module: I2C_Protocol

```
module I2C_Protocol(
    input clk,reset,ignition,
    input [15:0] MUX_input,
    inout SDIN,
    output reg finish_flag,
    output reg [2:0] ACK,
// output reg [15:0] data_check,
    output reg SCLK
// output reg [4:0] LEDG
);
```

Description: This is the protocol of music playing.

Purpose: Playing the background music according to the control signal of music playing state machine Sound_Top.sv.

Module: hpi_io_intf.sv

```
module hpi_io_intf( input         Clk, Reset,
                    input [1:0]   from_sw_address,
                    output[15:0]  from_sw_data_in,
                    input [15:0]  from_sw_data_out,
                    input         from_sw_r, from_sw_w, from_sw_cs, from_sw_reset, // Active low
                    inout [15:0]  OTG_DATA,
                    output[1:0]   OTG_ADDR,
                    output        OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N // Active low
                  );
```

Description: This module ensures the the OTG_DATA is high when NIOS is not writing to OTG_DATA inout bus and handles buffers.

Purpose: Performs as a tri-state buffer and an interface between NIOS II/e and EZ-OTG chip.

Module: VGA_controller.sv

```
module  VGA_controller (input              Clk,          // 50 MHz clock
                                           Reset,        // Active-high reset signal
                        output logic       VGA_HS,       // Horizontal sync pulse.  Active low
                                           VGA_VS,       // Vertical sync pulse.  Active low
                        input              VGA_CLK,      // 25 MHz VGA clock input
                        output logic       VGA_BLANK_N,  // Blanking interval indicator.  Active low.
                                           VGA_SYNC_N,   // Composite Sync signal.  Active low.  We don't use it in
                                                         // but the video DAC on the DE2 board requires an input for
                        output logic [9:0] DrawX,        // horizontal coordinate
                                           DrawY         // vertical coordinate
                        );
```

Description: This module produces synchronization timing signals to VS and HS of the VGA to help display on the VGA monitor.

Purpose: This module controls the VGA.

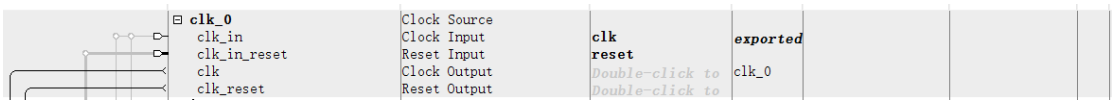
Module: HexDriver.sv

```
module HexDriver (input  [3:0]  In0,
                  output logic [6:0]  Out0);
```

Description: Generate hexadecimal number.

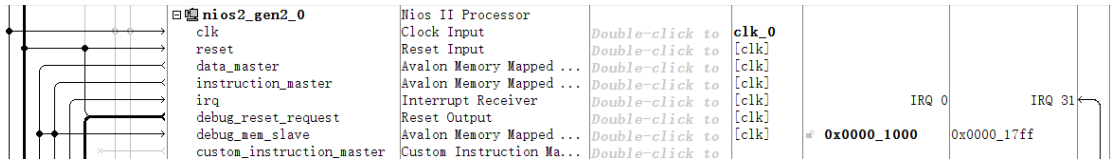
Purpose: Translate the result from binomial to hexadecimal to display on the FPGA board.

Module: clk_0



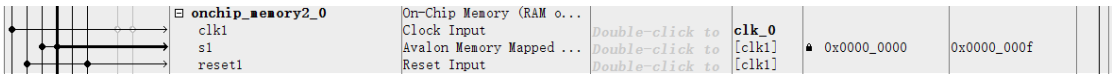

This block is the main clock signal generation for other blocks.

Module: nios2_gen2_0



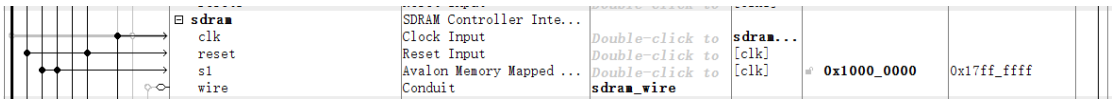

This block is the NIOS II processor of economic pattern. It is used to perform instructions in hardware part.

Module: onchip_memory2_0



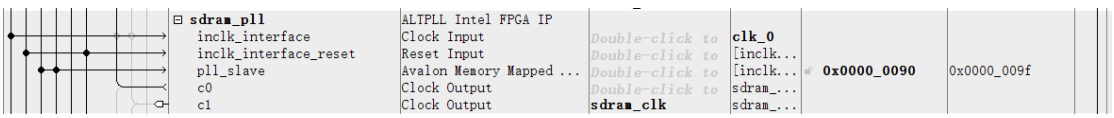

This is the memory for NIOS II and could be used to store or read/write data.

Module: sdram



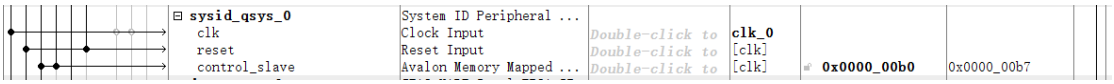

This block is the Synchronous Dynamic Random-Access Memory (SDRAM).

Module: sdram_pll



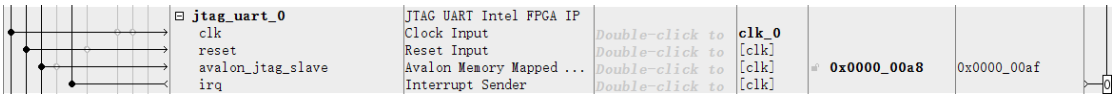

This block generates a PLL for the running of sdram, which is -3 ns before the clock to compensate for the loss in the transmission.

Module: sysid_qsys_0



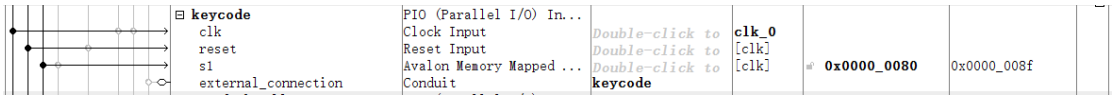

This block is a system id checker to make sure that the id configuration of our hardware part is corresponding to the software part.

Module: jtag_uart_0



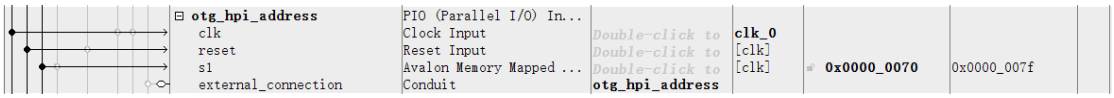

This is so that you can use the terminal of the host computer (the one running eclipse) to communicate with the NIOS II (using print and scan statements in c).

Module: keycode



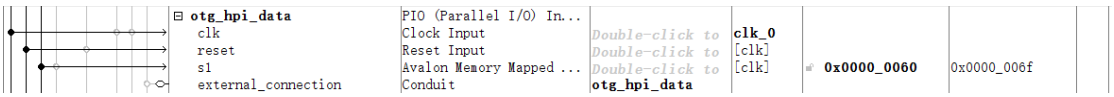

This block is a PIO part with 8-bit output form the keyboard.

Module: otg_hpi_address



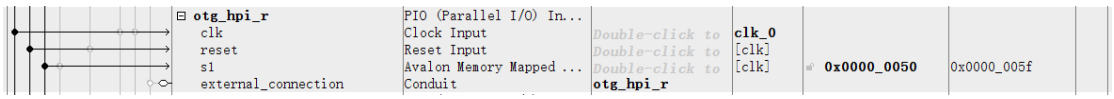

This block is a PIO part with 2-bit output of the address

Module: otg_hpi_data



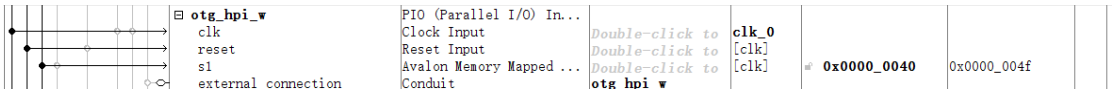

This block is a PIO part with 16-bit input of the data.

Module: otg_hpi_r



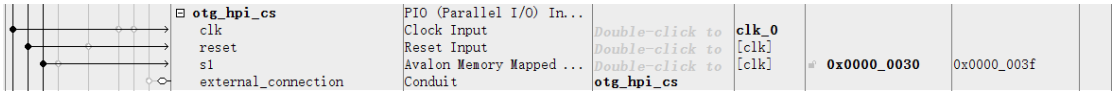

This block is a PIO part with 1-bit output of the read signal.

Module: otg_hpi_w



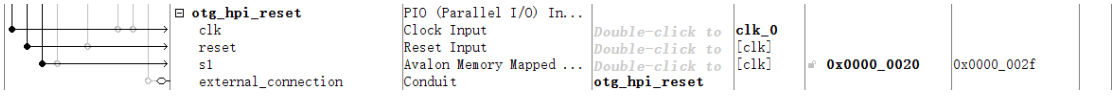

This block is a PIO part with 1-bit output of the write signal.

Module: otg_hpi_cs




This block is a PIO part with 1-bit output of the chip select signal.

Module: otg_hpi_reset



This block is a PIO part with 1-bit output of the reset signal.

4. Design Procedure/State Machine/Simulation Waveform

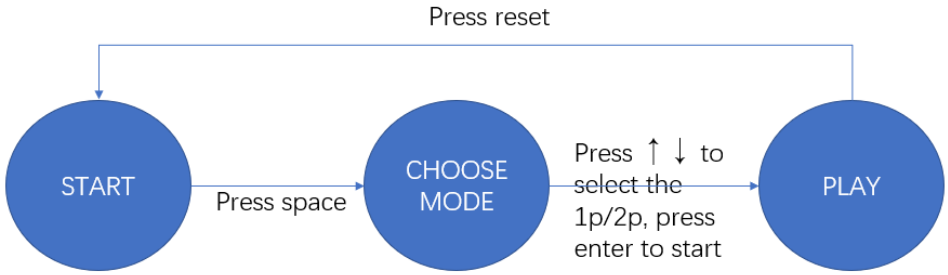
1) Overview of the design procedure

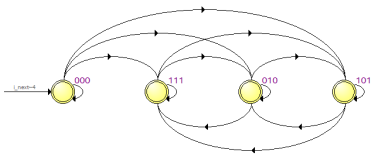
① We use lab 8 as the foundation of our foundation and using the platform designer. We modified the ball.sv and added other modules about background music, pictures and running of the notes.

② If the player presses the right answer, it will become a smiling, which is a signal to show the player a successful choice. We learnt from the GitHub how to transform .png picture and .wav music to .txt or .mif file, which can be stored into the OCM. We also learnt how to read a music file from the OCM and play it out.

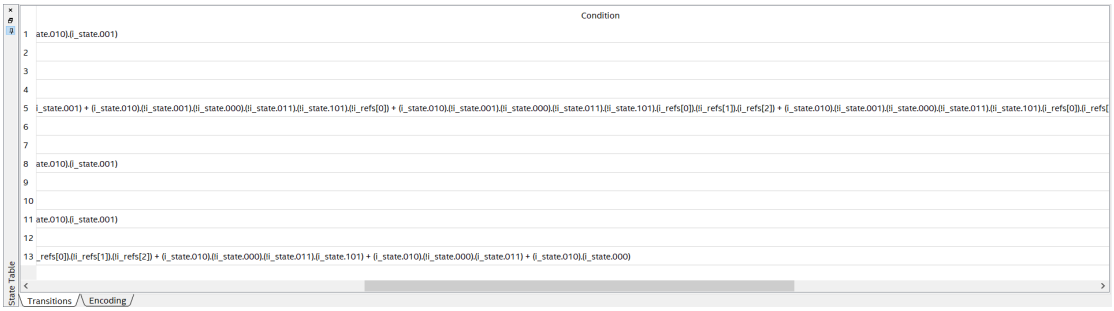
③ We made our own color mapper after learnt the running procedure. We use interface and Avalon bus to connect the hardware part and software part. The C code will read data from keyboard and export data to hardware part through interface and Avalon. Different engine will receive different position values and the color mapper will decide which and how to display and select one color according to the priority. Then the image will display on the screen through VGA controller and VGA port.

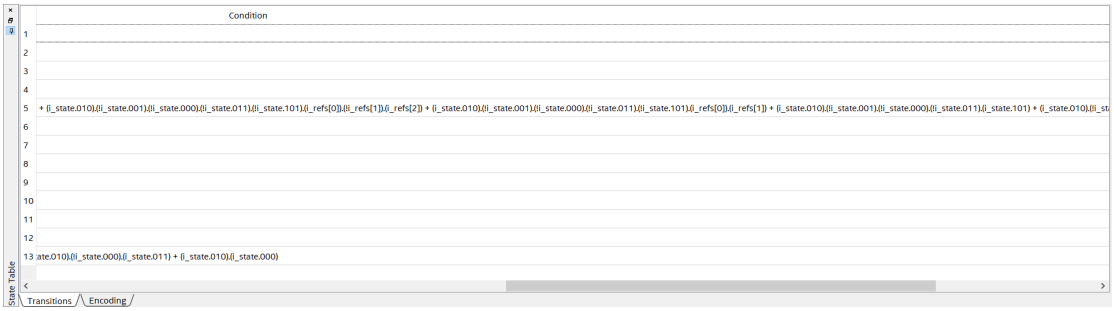
2) State Machine/Simulation Wave

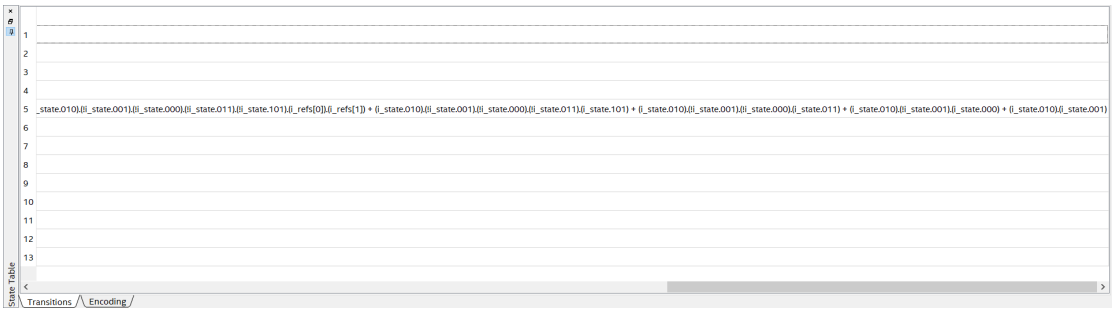


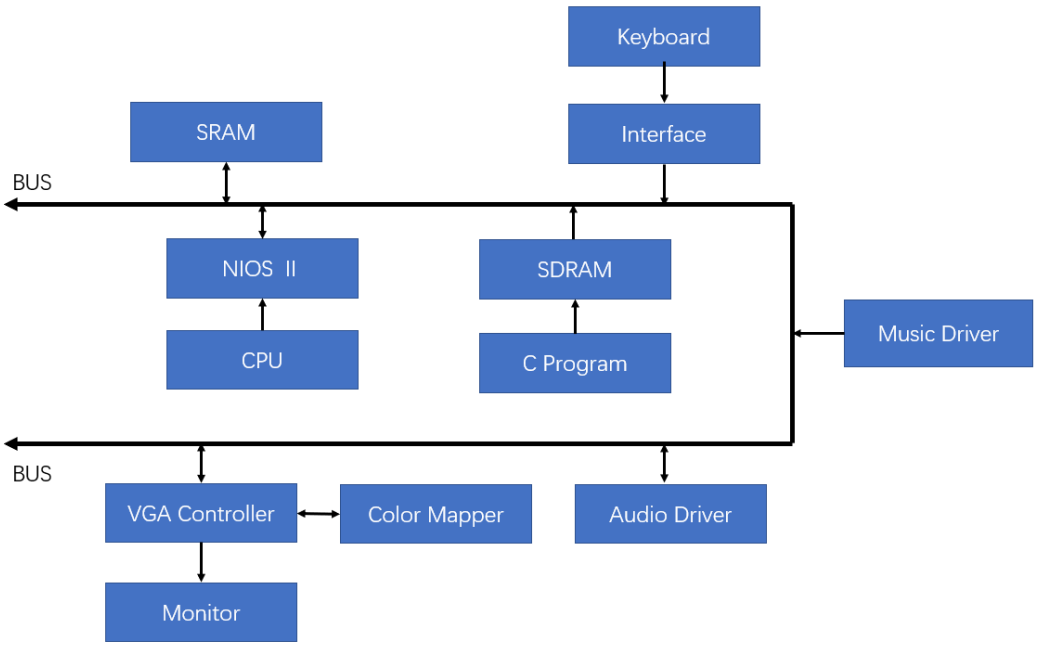

| | Source State | Destination State | |
|---|---|---|---|
| 1 | 000 | 010 | (ii_state.010),(i_state.001) + (i_state.010),(ii_state.001),(ii_refs[0]) + (i_state.010),(ii_state.001),(i_refs[0]),(ii_refs[1]),(i_refs[2]) + (i_state.010),(ii_state.001),(i_refs[0]),(i_refs[1]) + (i_state.010),(i_state.001) |
| 2 | 000 | 101 | (i_state.111) |
| 3 | 000 | 111 | (i_state.010),(i_refs[0]),(ii_refs[1]),(ii_refs[2]) |
| 4 | 000 | 000 | (ii_state.000),(ii_state.011),(i_state.101) + (ii_state.000),(i_state.011) + (i_state.000) |
| 5 | 010 | 010 | (ii_state.010),(ii_state.001),(ii_state.000),(ii_state.011),(i_state.101) + (ii_state.010),(ii_state.001),(ii_state.000),(i_state.011) + (ii_state.010),(ii_state.001),(i_state.000) + (ii_state.010),(i_state.001) + (i_state.010),(ii_state.001),(ii_state.000),(ii_state.0 |
| 6 | 010 | 101 | (i_state.111) |
| 7 | 010 | 111 | (i_state.010),(i_refs[0]),(ii_refs[1]),(ii_refs[2]) |
| 8 | 101 | 010 | (ii_state.010),(i_state.001) + (i_state.010),(ii_state.001),(ii_refs[0]) + (i_state.010),(ii_state.001),(i_refs[0]),(ii_refs[1]),(i_refs[2]) + (i_state.010),(ii_state.001),(i_refs[0]),(i_refs[1]) + (i_state.010),(i_state.001) |
| 9 | 101 | 101 | (ii_state.111),(ii_state.000),(ii_state.011),(i_state.101) + (ii_state.111),(ii_state.000),(i_state.011) + (ii_state.111),(i_state.000) + (i_state.111) |
| 10 | 101 | 111 | (i_state.010),(i_refs[0]),(ii_refs[1]),(ii_refs[2]) |
| 11 | 111 | 010 | (ii_state.010),(i_state.001) + (i_state.010),(ii_state.001),(ii_refs[0]) + (i_state.010),(ii_state.001),(i_refs[0]),(ii_refs[1]),(i_refs[2]) + (i_state.010),(ii_state.001),(i_refs[0]),(i_refs[1]) + (i_state.010),(i_state.001) |
| 12 | 111 | 101 | (i_state.111) |
| 13 | 111 | 111 | (ii_state.010),(ii_state.000),(ii_state.011),(i_state.101) + (ii_state.010),(ii_state.000),(i_state.011) + (ii_state.010),(i_state.000) + (i_state.010),(ii_state.000),(ii_state.011),(ii_state.101),(i_refs[0]),(ii_refs[1]),(i_refs[2]) + (i_state.010),(ii_state.000),(ii_st |

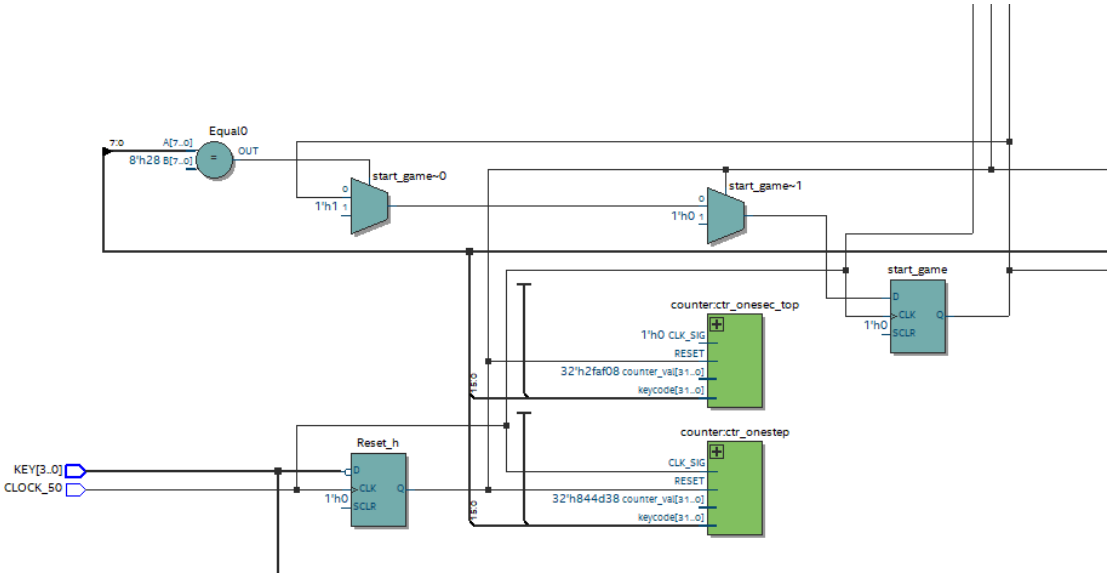


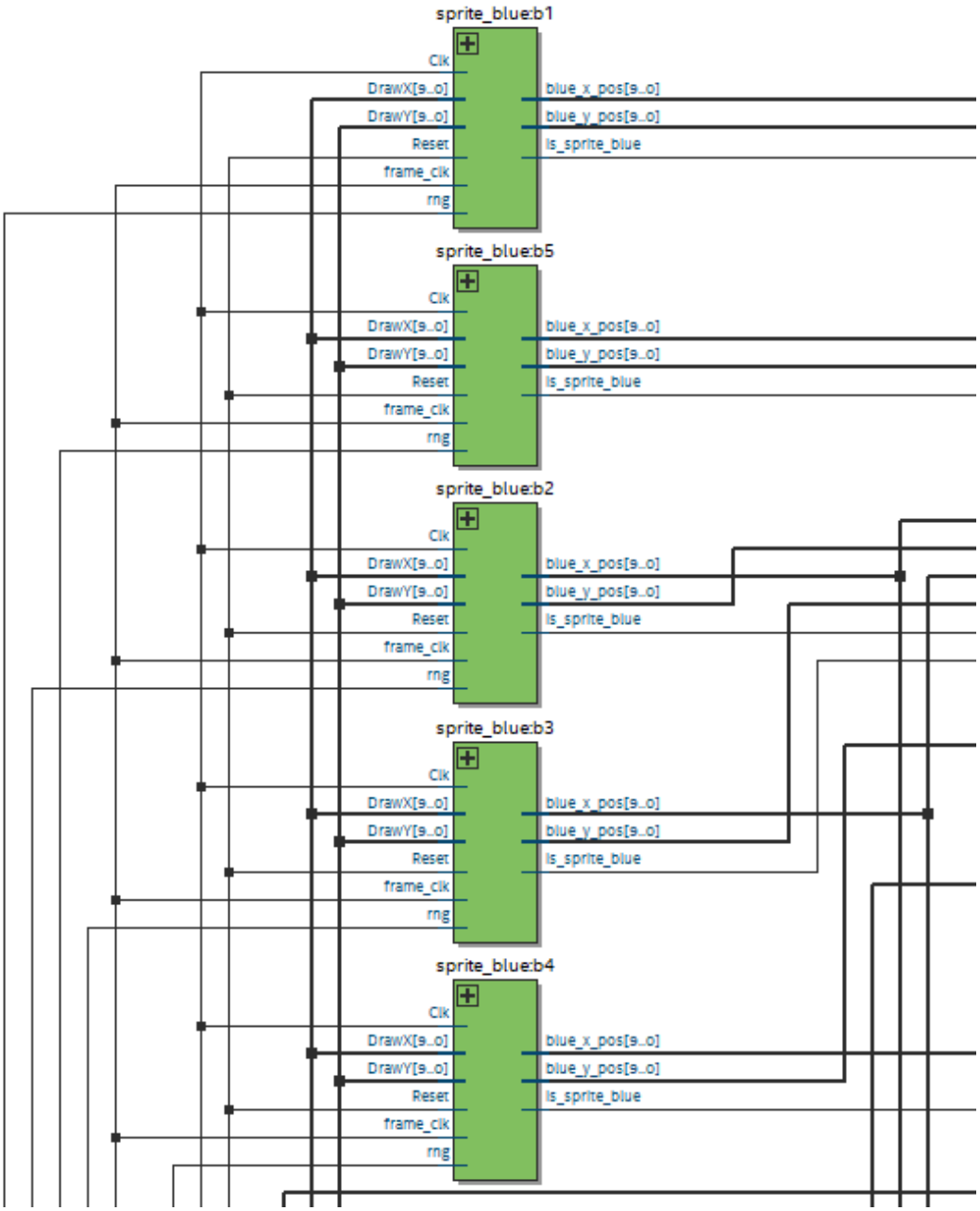



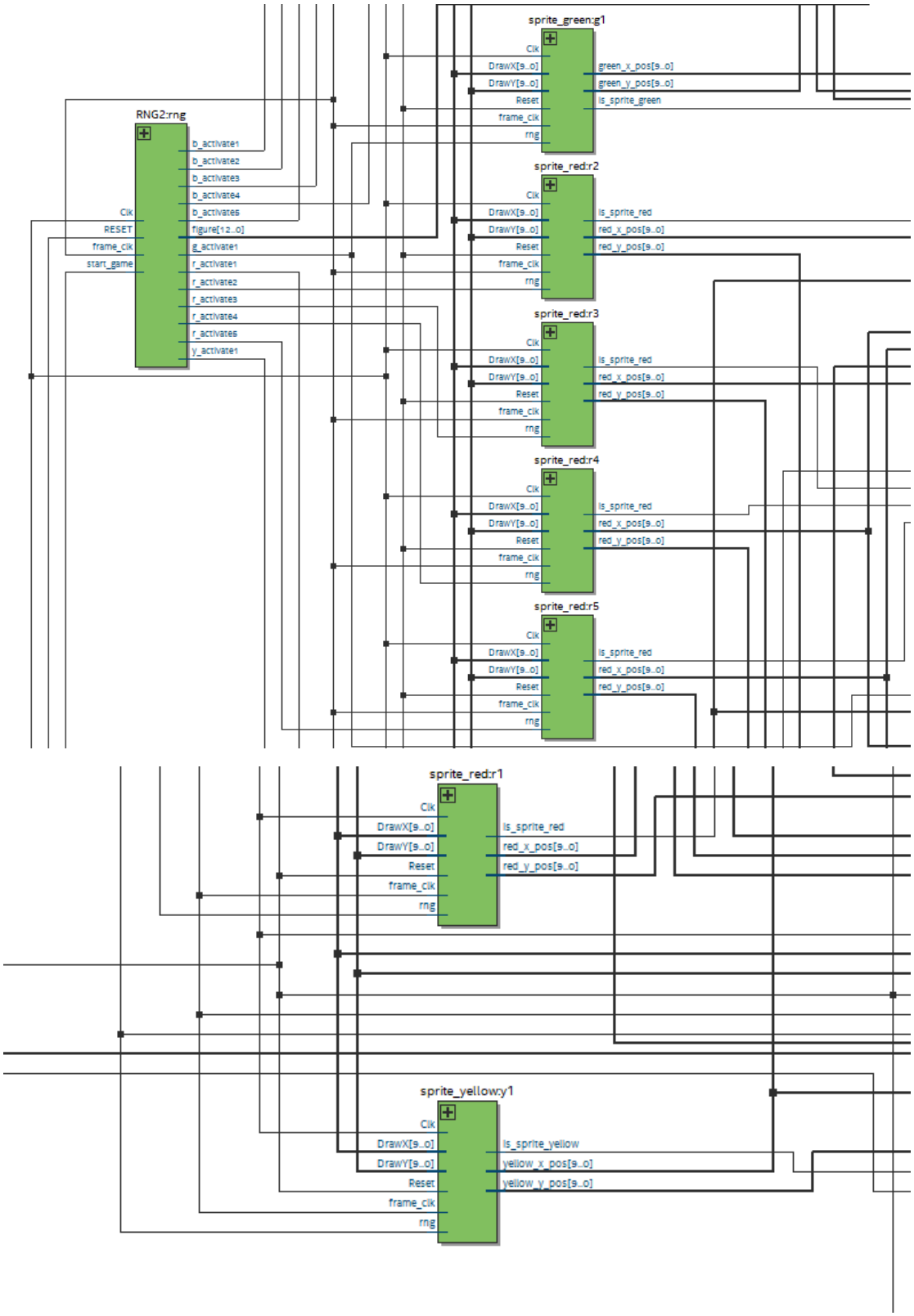



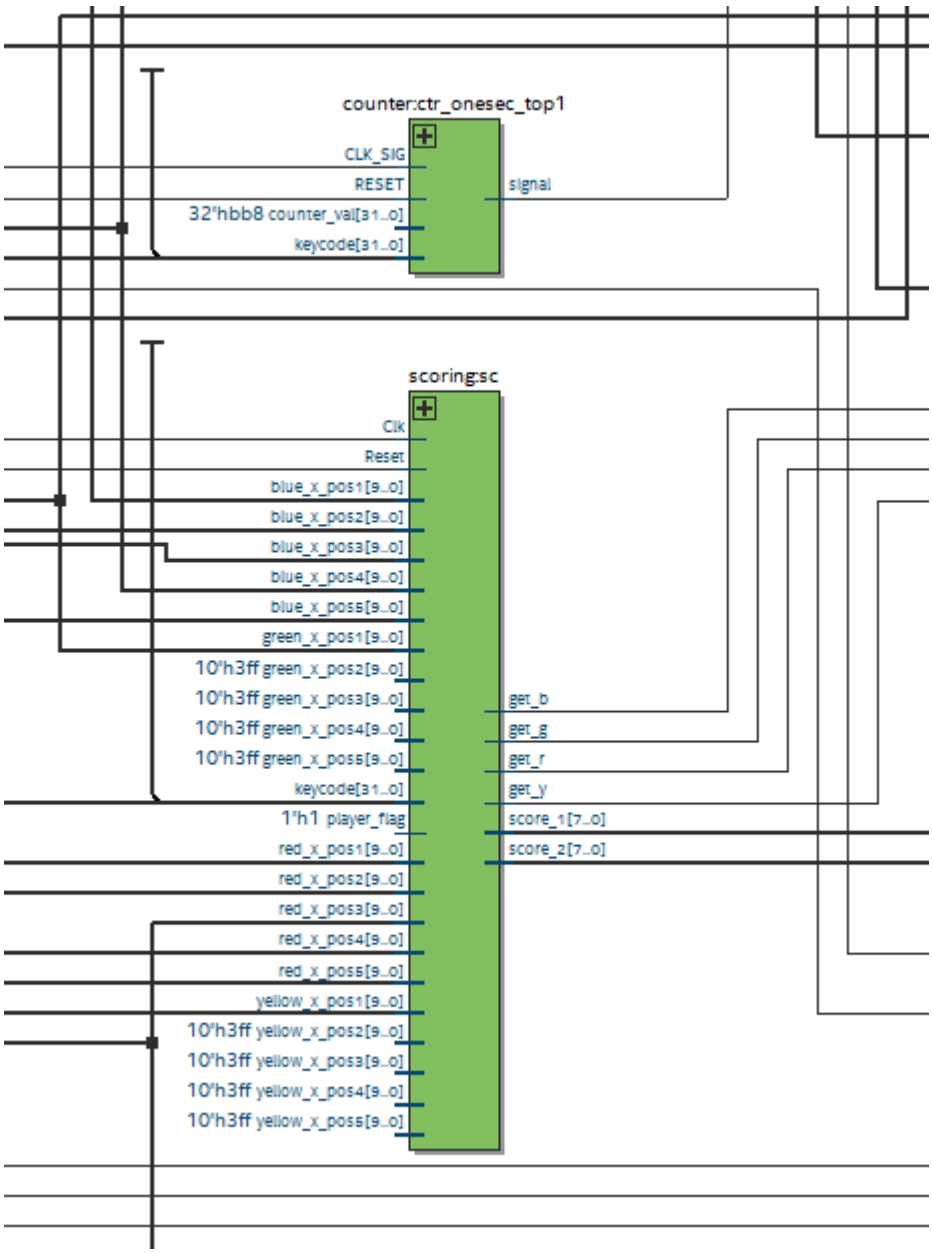

## 5. Block Diagram

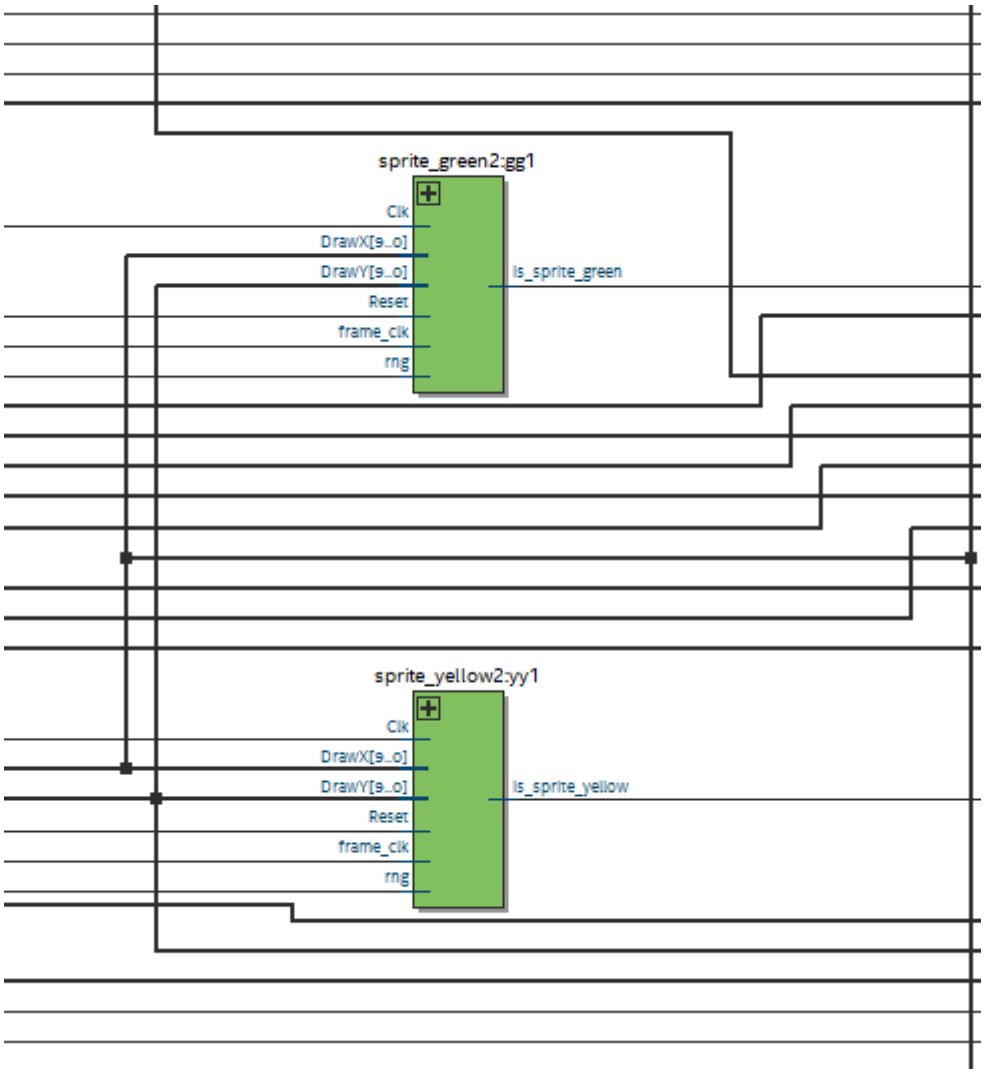


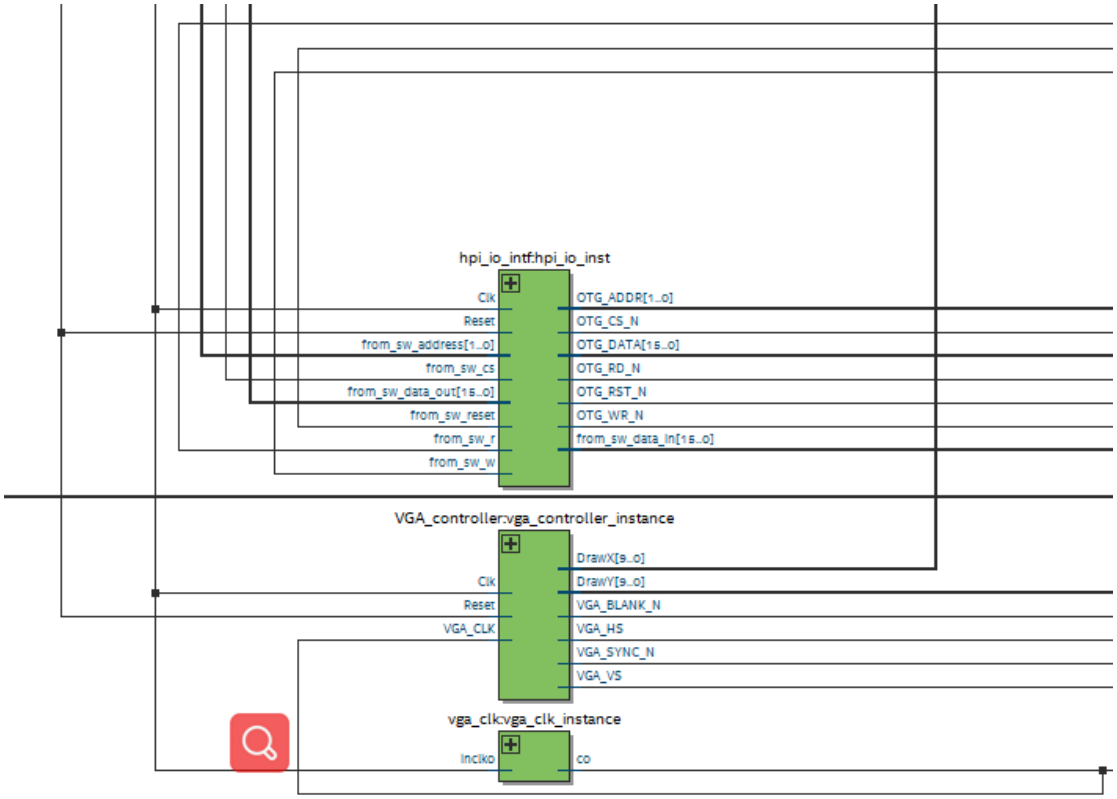

Equal0
7:0
A[7..0]
8'h28 B[7..0]
=
OUT
start_game~0
0
1'h1 1
start_game~1
0
1'h0 1
start_game
D
CLK
SCLR
Q
1'h0
counter:ctr_onesec_top
1'h0 CLK_SIG
RESET
32'h2faf08 counter_val[31..0]
keycode[31..0]
counter:ctr_onestep
CLK_SIG
RESET
32'h844d38 counter_val[31..0]
keycode[31..0]
Reset_h
KEY[3..0]
CLOCK_50
D
CLK
SCLR
Q
1'h0

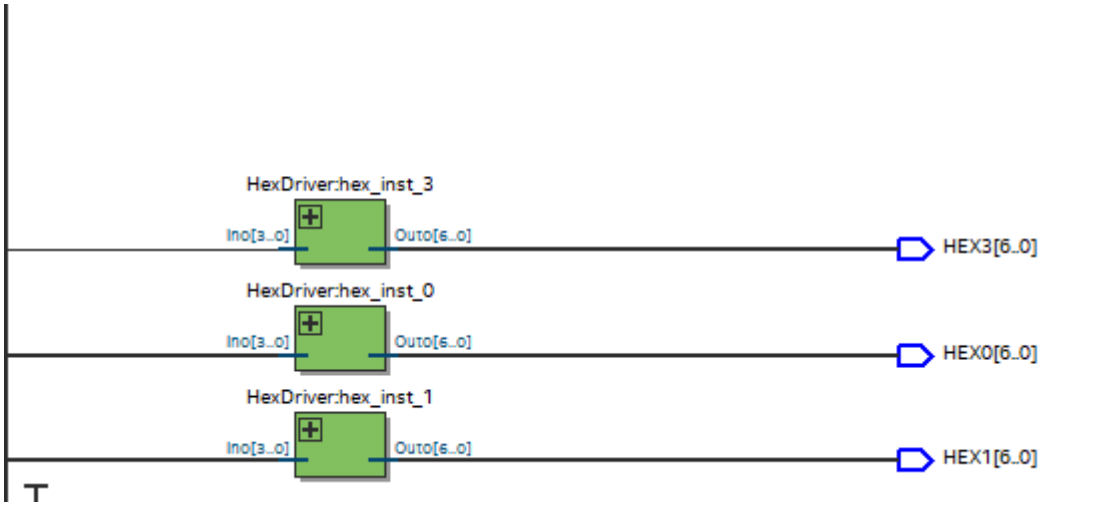

sprite_blue:b1
Clk
DrawX[9..0]
DrawY[9..0]
Reset
frame_clk
rng
blue_x_pos[9..0]
blue_y_pos[9..0]
is_sprite_blue
sprite_blue:b5
Clk
DrawX[9..0]
DrawY[9..0]
Reset
frame_clk
rng
blue_x_pos[9..0]
blue_y_pos[9..0]
is_sprite_blue
sprite_blue:b2
Clk
DrawX[9..0]
DrawY[9..0]
Reset
frame_clk
rng
blue_x_pos[9..0]
blue_y_pos[9..0]
is_sprite_blue
sprite_blue:b3
Clk
DrawX[9..0]
DrawY[9..0]
Reset
frame_clk
rng
blue_x_pos[9..0]
blue_y_pos[9..0]
is_sprite_blue
sprite_blue:b4
Clk
DrawX[9..0]
DrawY[9..0]
Reset
frame_clk
rng
blue_x_pos[9..0]
blue_y_pos[9..0]
is_sprite_blue

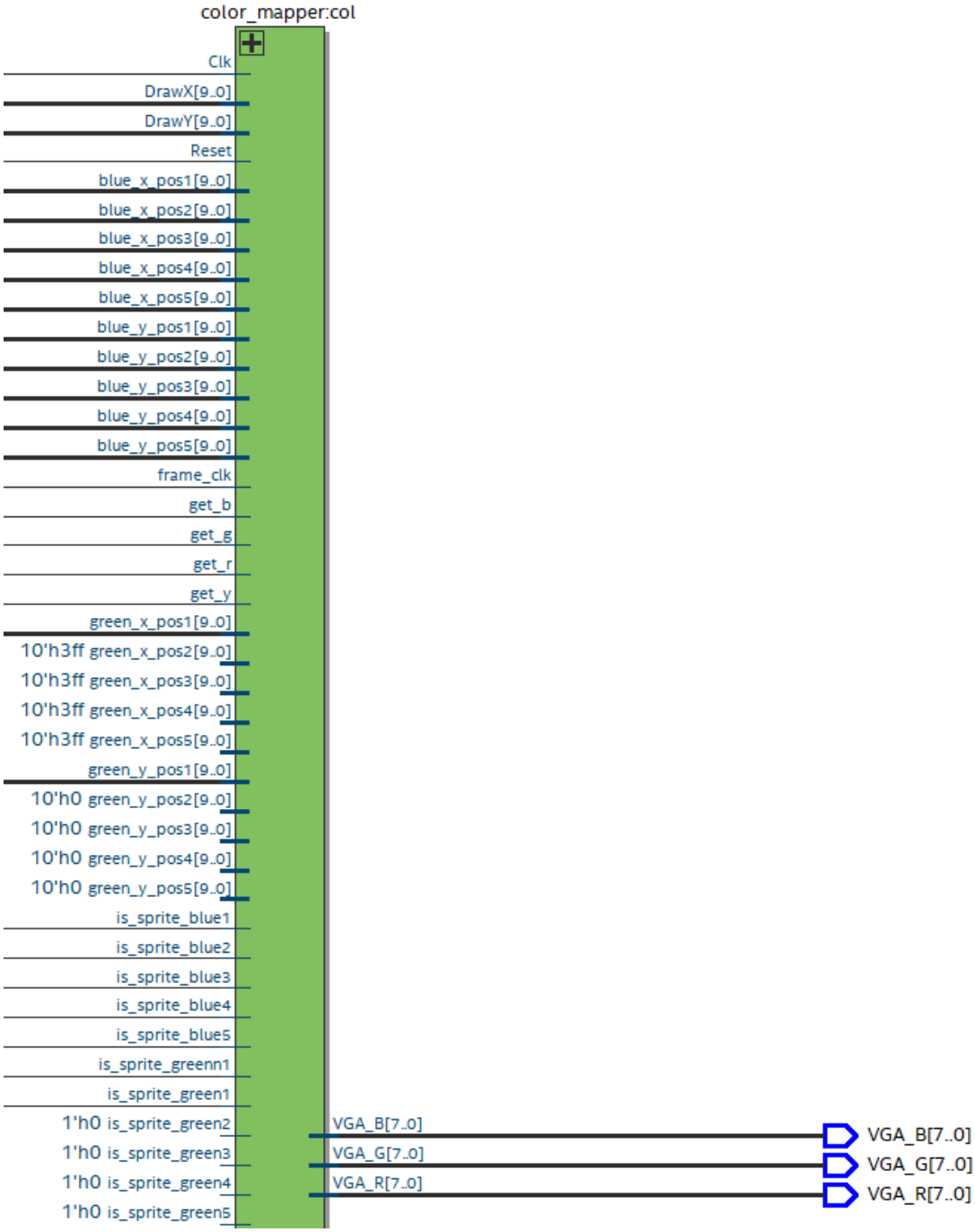

sprite_green:g1
Clk
DrawX[9..0]
DrawY[9..0]
Reset
frame_clk
rng
green_x_pos[9..0]
green_y_pos[9..0]
is_sprite_green
RNG2:rng
Clk
RESET
frame_clk
start_game
b_activate1
b_activate2
b_activate3
b_activate4
b_activate5
figure[12..0]
g_activate1
r_activate1
r_activate2
r_activate3
r_activate4
r_activate5
y_activate1
sprite_red:r2
sprite_red:r3
sprite_red:r4
sprite_red:r5
is_sprite_red
red_x_pos[9..0]
red_y_pos[9..0]
sprite_red:r1
sprite_yellow:y1
is_sprite_yellow
yellow_x_pos[9..0]
yellow_y_pos[9..0]

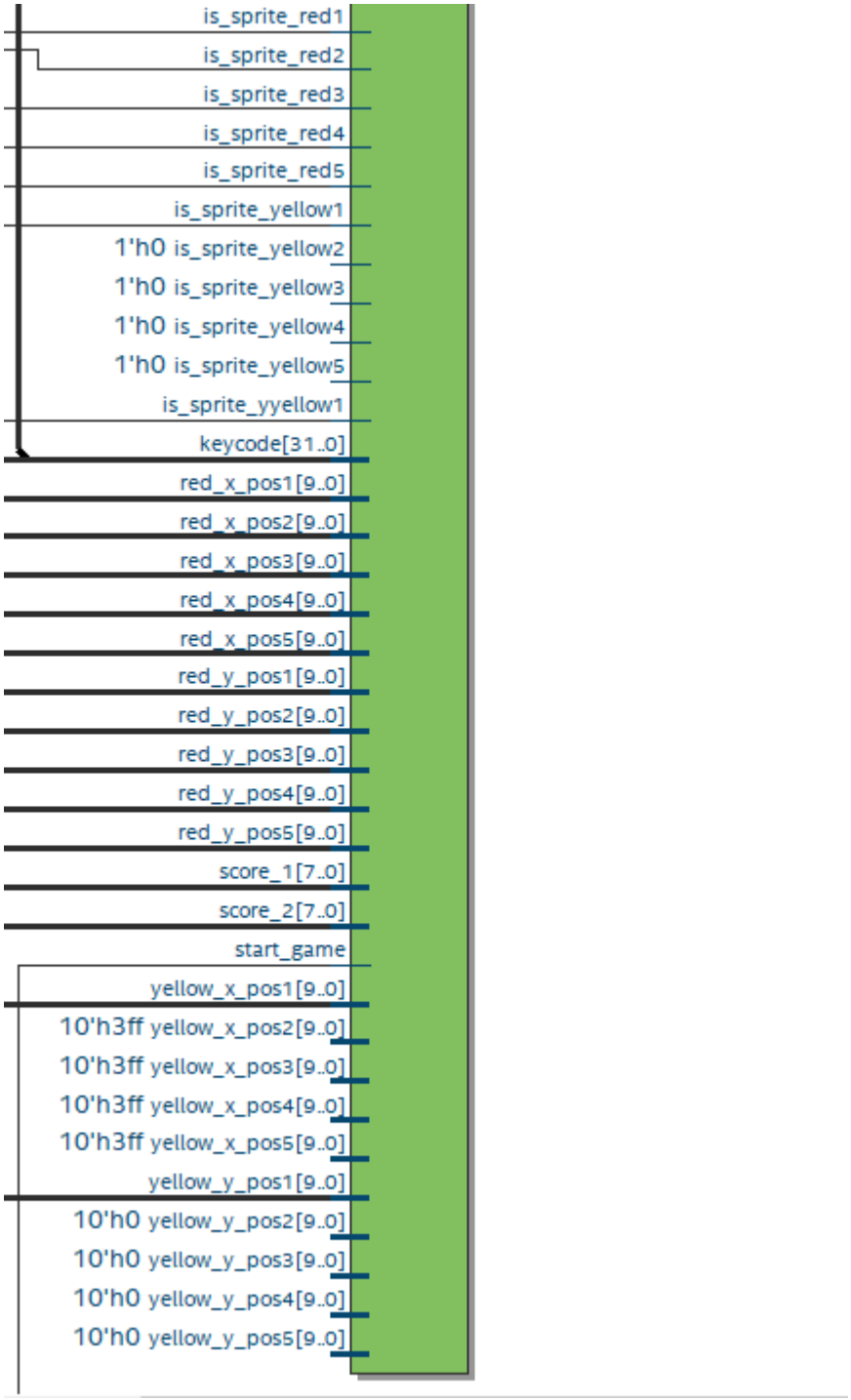

counter:ctr_onesec_top1
CLK_SIG
RESET
32'hbb8 counter_val[31..0]
keycode[31..0]
signal
scoring:sc
Clk
Reset
blue_x_pos1[9..0]
blue_x_pos2[9..0]
blue_x_pos3[9..0]
blue_x_pos4[9..0]
blue_x_pos5[9..0]
green_x_pos1[9..0]
10'h3ff green_x_pos2[9..0]
10'h3ff green_x_pos3[9..0]
10'h3ff green_x_pos4[9..0]
10'h3ff green_x_pos5[9..0]
keycode[31..0]
1'h1 player_flag
red_x_pos1[9..0]
red_x_pos2[9..0]
red_x_pos3[9..0]
red_x_pos4[9..0]
red_x_pos5[9..0]
yellow_x_pos1[9..0]
10'h3ff yellow_x_pos2[9..0]
10'h3ff yellow_x_pos3[9..0]
10'h3ff yellow_x_pos4[9..0]
10'h3ff yellow_x_pos5[9..0]
get_b
get_g
get_r
get_y
score_1[7..0]
score_2[7..0]

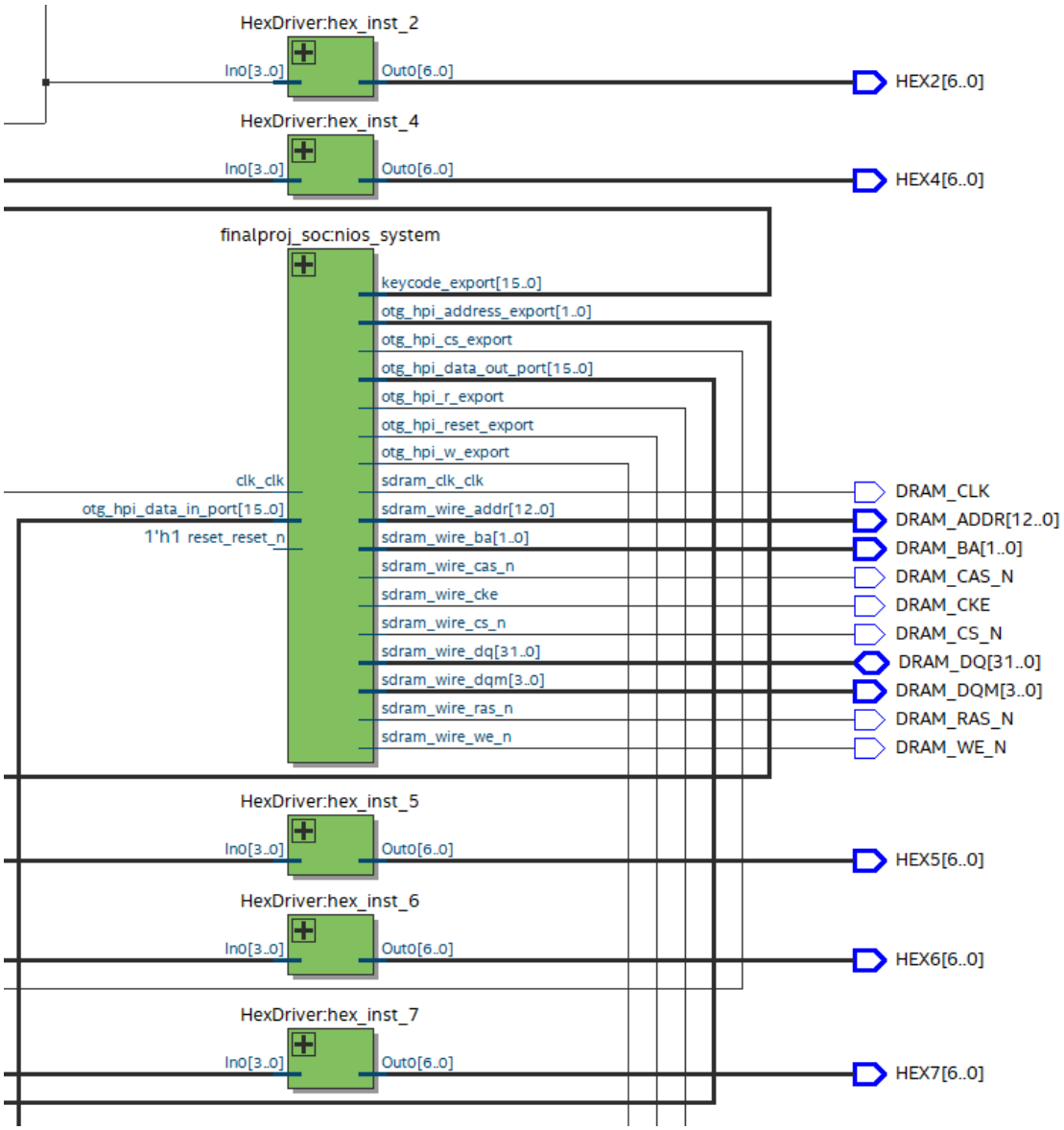

sprite_green2:gg1
Clk
DrawX[9..0]
DrawY[9..0]
Reset
frame_clk
rng
is_sprite_green
sprite_yellow2:yy1
Clk
DrawX[9..0]
DrawY[9..0]
Reset
frame_clk
rng
is_sprite_yellow

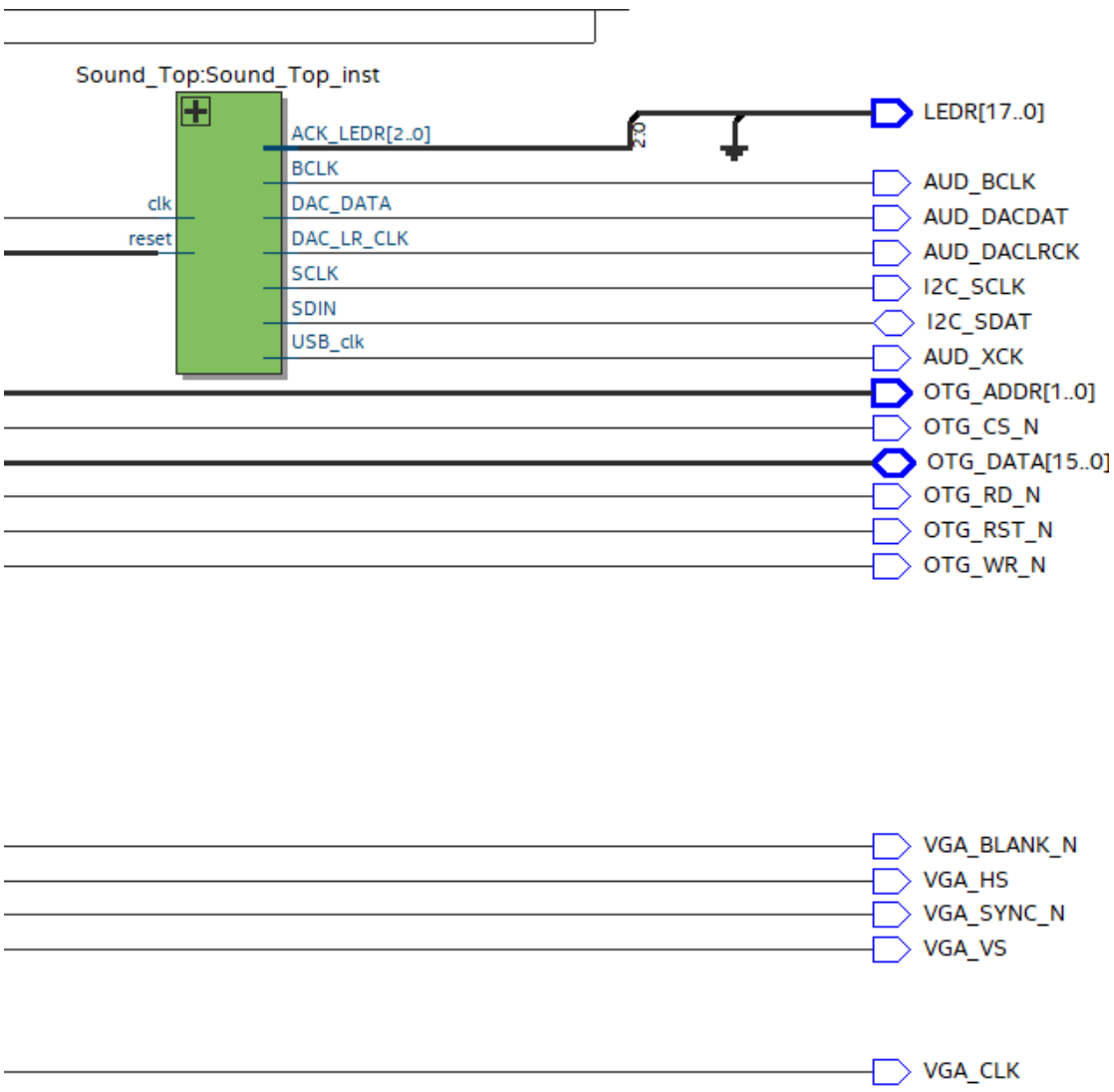

hpi_io_intf:hpi_io_inst
Clk
Reset
from_sw_address[1..0]
from_sw_cs
from_sw_data_out[15..0]
from_sw_reset
from_sw_r
from_sw_w
OTG_ADDR[1..0]
OTG_CS_N
OTG_DATA[15..0]
OTG_RD_N
OTG_RST_N
OTG_WR_N
from_sw_data_in[15..0]
VGA_controller:vga_controller_instance
Clk
Reset
VGA_CLK
DrawX[9..0]
DrawY[9..0]
VGA_BLANK_N
VGA_HS
VGA_SYNC_N
VGA_VS
vga_clk:vga_clk_instance
inclko
co


HexDriver:hex_inst_3
In0[3..0]
Out0[6..0]
HEX3[6..0]
HexDriver:hex_inst_0
In0[3..0]
Out0[6..0]
HEX0[6..0]
HexDriver:hex_inst_1
In0[3..0]
Out0[6..0]
HEX1[6..0]

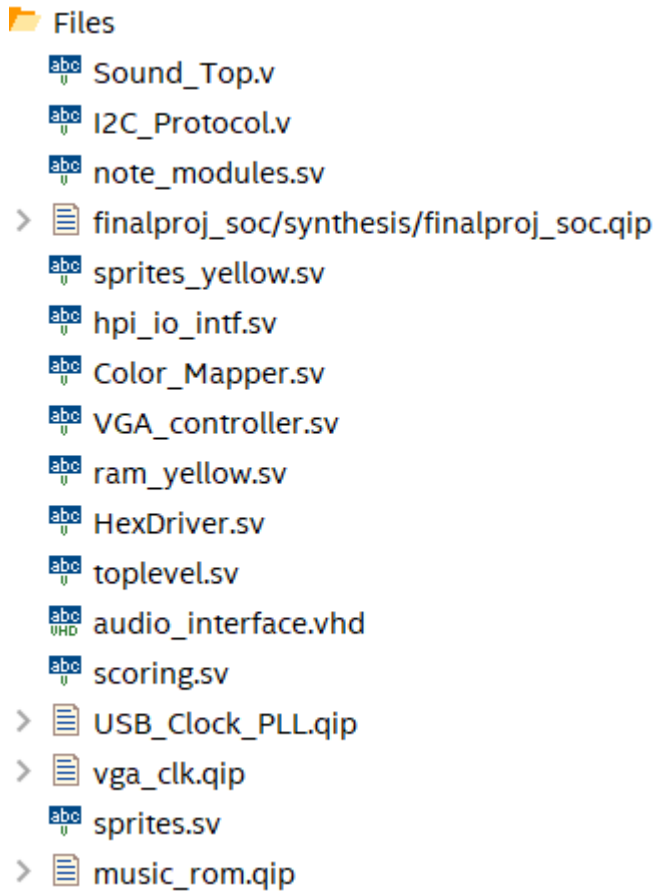

color_mapper:col
Clk
DrawX[9..0]
DrawY[9..0]
Reset
blue_x_pos1[9..0]
blue_x_pos2[9..0]
blue_x_pos3[9..0]
blue_x_pos4[9..0]
blue_x_pos5[9..0]
blue_y_pos1[9..0]
blue_y_pos2[9..0]
blue_y_pos3[9..0]
blue_y_pos4[9..0]
blue_y_pos5[9..0]
frame_clk
get_b
get_g
get_r
get_y
green_x_pos1[9..0]
10'h3ff green_x_pos2[9..0]
10'h3ff green_x_pos3[9..0]
10'h3ff green_x_pos4[9..0]
10'h3ff green_x_pos5[9..0]
green_y_pos1[9..0]
10'h0 green_y_pos2[9..0]
10'h0 green_y_pos3[9..0]
10'h0 green_y_pos4[9..0]
10'h0 green_y_pos5[9..0]
is_sprite_blue1
is_sprite_blue2
is_sprite_blue3
is_sprite_blue4
is_sprite_blue5
is_sprite_greenn1
is_sprite_green1
1'h0 is_sprite_green2
1'h0 is_sprite_green3
1'h0 is_sprite_green4
1'h0 is_sprite_green5
VGA_B[7..0]
VGA_G[7..0]
VGA_R[7..0]
VGA_B[7..0]
VGA_G[7..0]
VGA_R[7..0]


is_sprite_red1
is_sprite_red2
is_sprite_red3
is_sprite_red4
is_sprite_red5
is_sprite_yellow1
1'h0 is_sprite_yellow2
1'h0 is_sprite_yellow3
1'h0 is_sprite_yellow4
1'h0 is_sprite_yellow5
is_sprite_yyellow1
keycode[31..0]
red_x_pos1[9..0]
red_x_pos2[9..0]
red_x_pos3[9..0]
red_x_pos4[9..0]
red_x_pos5[9..0]
red_y_pos1[9..0]
red_y_pos2[9..0]
red_y_pos3[9..0]
red_y_pos4[9..0]
red_y_pos5[9..0]
score_1[7..0]
score_2[7..0]
start_game
yellow_x_pos1[9..0]
10'h3ff yellow_x_pos2[9..0]
10'h3ff yellow_x_pos3[9..0]
10'h3ff yellow_x_pos4[9..0]
10'h3ff yellow_x_pos5[9..0]
yellow_y_pos1[9..0]
10'h0 yellow_y_pos2[9..0]
10'h0 yellow_y_pos3[9..0]
10'h0 yellow_y_pos4[9..0]
10'h0 yellow_y_pos5[9..0]


HexDriver:hex_inst_2
In0[3..0]
Out0[6..0]
HEX2[6..0]
HexDriver:hex_inst_4
In0[3..0]
Out0[6..0]
HEX4[6..0]
finalproj_soc:nios_system
keycode_export[15..0]
otg_hpi_address_export[1..0]
otg_hpi_cs_export
otg_hpi_data_out_port[15..0]
otg_hpi_r_export
otg_hpi_reset_export
otg_hpi_w_export
clk_clk
otg_hpi_data_in_port[15..0]
1'h1 reset_reset_n
sdram_clk_clk
sdram_wire_addr[12..0]
sdram_wire_ba[1..0]
sdram_wire_cas_n
sdram_wire_cke
sdram_wire_cs_n
sdram_wire_dq[31..0]
sdram_wire_dqm[3..0]
sdram_wire_ras_n
sdram_wire_we_n
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
HexDriver:hex_inst_5
In0[3..0]
Out0[6..0]
HEX5[6..0]
HexDriver:hex_inst_6
In0[3..0]
Out0[6..0]
HEX6[6..0]
HexDriver:hex_inst_7
In0[3..0]
Out0[6..0]
HEX7[6..0]


Sound_Top:Sound_Top_inst
clk
reset
ACK_LEDR[2..0]
BCLK
DAC_DATA
DAC_LR_CLK
SCLK
SDIN
USB_clk
3'h0
LEDR[17..0]
AUD_BCLK
AUD_DACDAT
AUD_DACLRCK
I2C_SCLK
I2C_SDAT
AUD_XCK
OTG_ADDR[1..0]
OTG_CS_N
OTG_DATA[15..0]
OTG_RD_N
OTG_RST_N
OTG_WR_N
VGA_BLANK_N
VGA_HS
VGA_SYNC_N
VGA_VS
VGA_CLK

6. SV Code




7. Design Statistics and Discussion

| | |
|---|---|
| LUT | 5634 |
| DSP | 0 |
| Memory (BRAM) | 3224440 |
| Flip-Flop | 2431 |
| Frequency | 135.21MHz |
| Static Power | 108.75mW |
| Dynamic Power | 0.77mW |
| Total Power | 210.68mW |

8. Credits

   General Framework: https://github.com/kushpatel0703/GuitarZero

   Drawing Tutorial: https://github.com/atrifex/ECE385-HelperTools

   Music Playing: https://github.com/0x60/385-audio-tools,
   https://github.com/AhmadGon/WM8731-Audio-CODEC

9. Conclusion

   1) Debug

      The note first can't display with the map we generate. We first think that's the bug in the map generation, while finally we found out that it's the bug of the sprite of note. After being in the leftmost of the screen, the note's motion is set to 0, and the note's position should be set to 639, while we let it be 0 instead that make the bug. In 2p mode, the first player's display is correct since it's just the same as 1p mode, but the

second player's display is not correct. We debug this by setting the position of the second player's elements according to the first players (just like translation of the first player), instead of creating the new instantiation: player2.

In the music part, we first encounter a bug of converting .wav file to .hex file. Some .wav files cannot be read by the python code correctly. Then we get the other music from internet and convert it to .wav by Adobe Audition. About the storage, we used flash to store the music firstly. But after we finished the control logic of this part, there is no music played out from the FPGA. Then we use the OCM to store the music, but it is too long for the capacity of OCM. We decided to clip the audio length and rewrote the controlling state machine module to finish the playing of background music.

2) Accomplishment

In this final project, we finished the basic function of a 2D percussion music game about rhythm and hit the drumbeat with the background music. Meanwhile, we also added the scoring part to calculate and display the score the player awarded. We also finished two modes, which are one player mode and PvP mode. Unfortunately, we did not arrange our drumbeat according to the rhythm of our background. Secondly, we have a background music of just 20 seconds limited by the capacity of OCM, which is a little shorter for a percussion music game. Thirdly, we just finished the PvP mode, we did not finish the cooperation mode, which should be passed through the collaboration of two player. Finally, we did not have an end of game and score settlement interface after the music finished.