**ECE385**

Fall 2020

Final Project

**Bongo Touch**

Li Taoran: 3180110750

Yuan Xinkai: 3180110983

LA3/2020.10.17

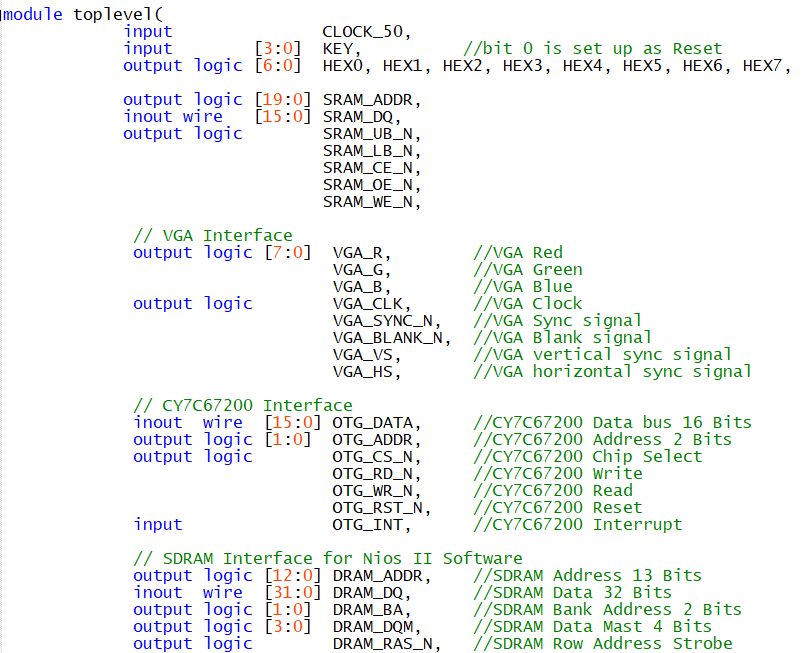
Yin Haocheng

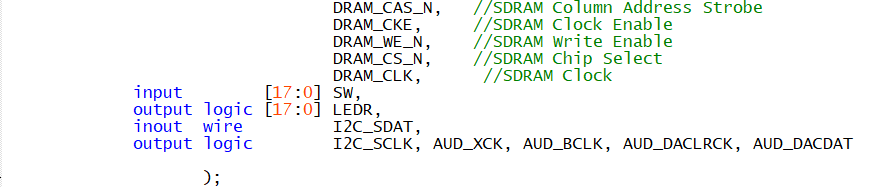
1. Introduction

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.

1. Written Description
2. Description of the overview of the circuit
3. 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.
4. 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.
5. Descriptions of general flow of the circuit





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 Ⅱlogic

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

1. 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.

1. Overview

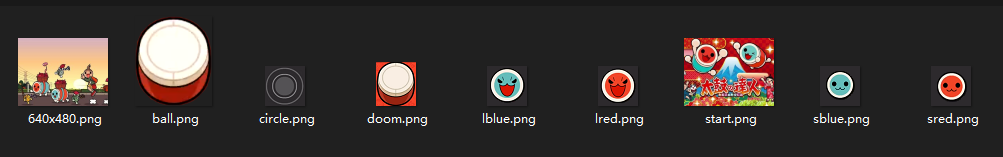
In this final project, we use the same input and output logic of lab 8, which use keyboard to control the running of our game. The color mapper paints the image onto the VGA with a fixed refresh rate and decide which color to display on the screen according to the instruction of green, red or blue and 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.

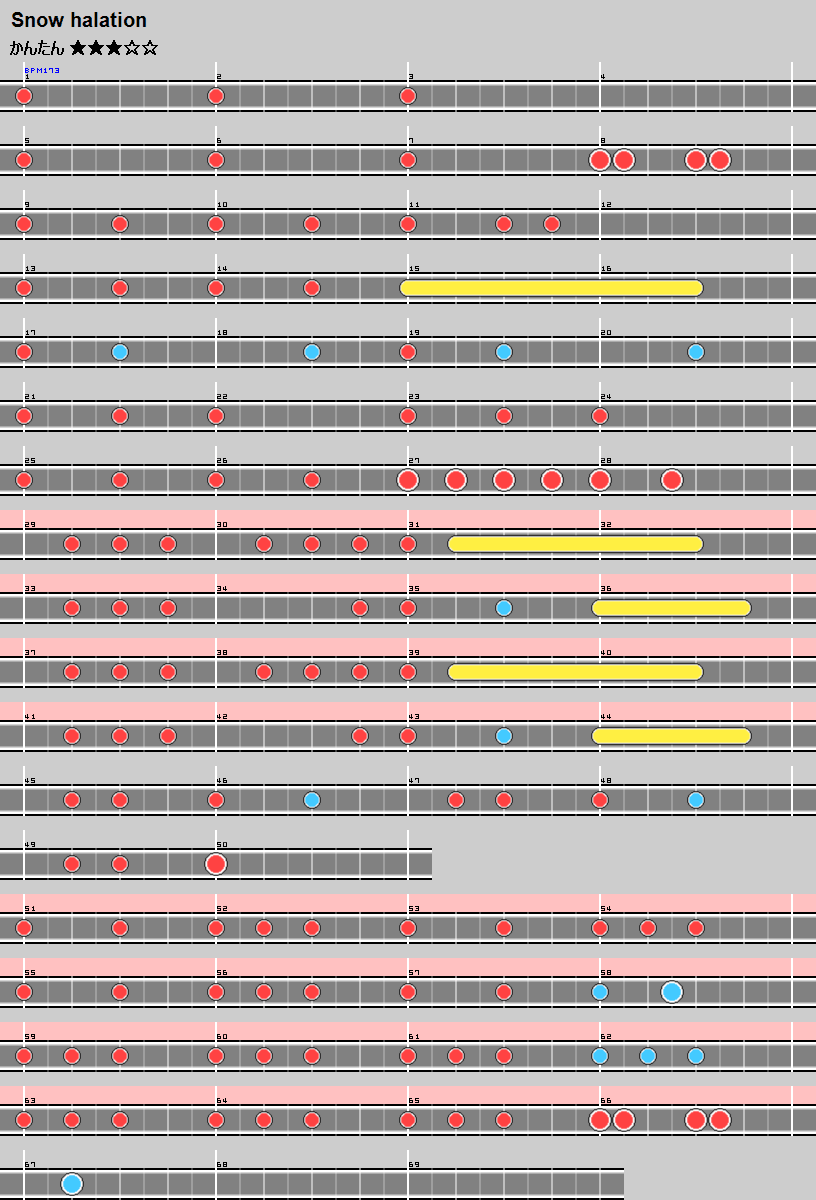
* 1. Picture Drawing

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.



* 1. 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.



* 1. 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.



* 1. 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.

* 1. 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.

1. 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).

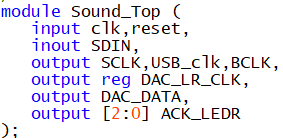
Output:

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.

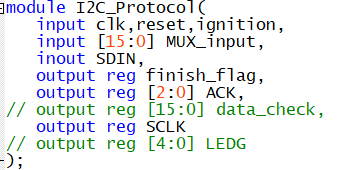
Module: Sound\_Top



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.

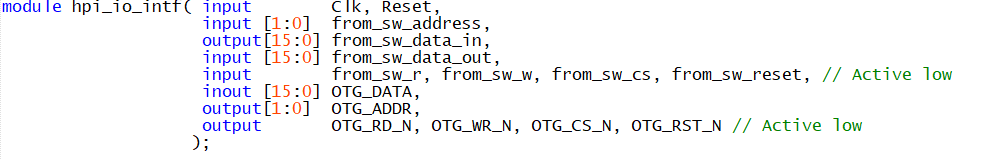
Module: I2C\_Protocol



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.

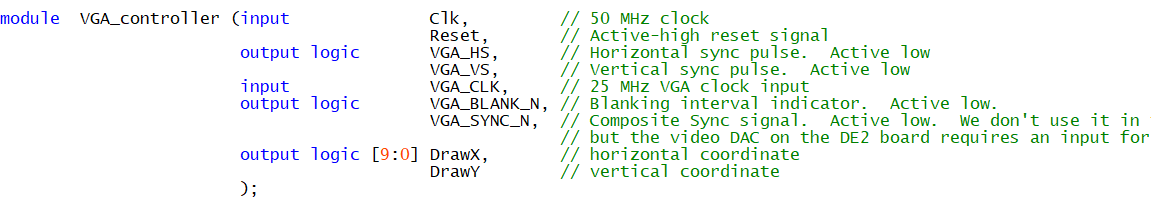
Module: hpi\_io\_intf.sv



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



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.

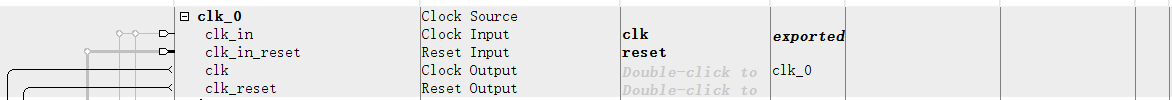
Module: HexDriver.sv



Description: Generate hexadecimal number.

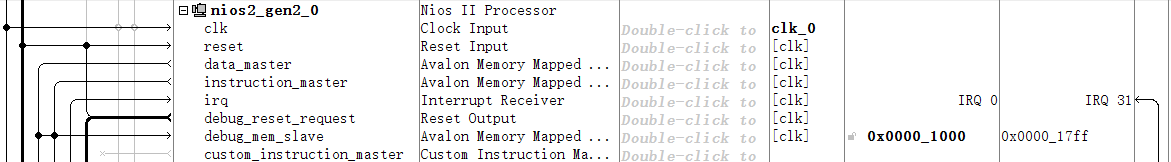
Purpose: Translate the result from binomial to hexadecimal to display on the FPGA board.

Module: clk\_0



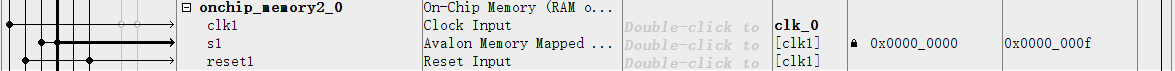
This block is the main clock signal generation for other blocks.

Module: nios2\_gen2\_0



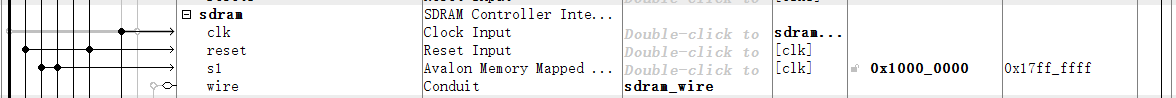
This block is the NIOS Ⅱ processor of economic pattern. It is used to perform instructions in hardware part.

Module: onchip\_memory2\_0



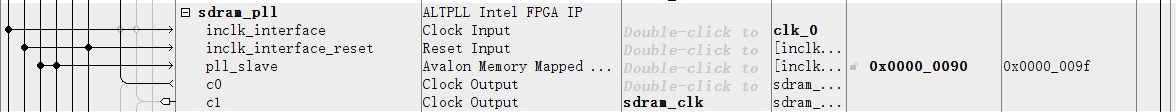
This is the memory for NIOS Ⅱ and could be used to store or read/write data.

Module: sdram



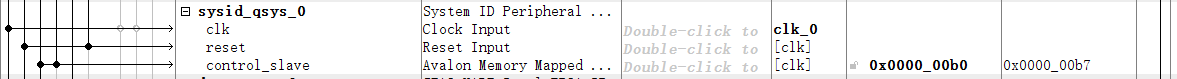
This block is the Synchronous Dynamic Random-Access Memory (SDRAM).

Module: sdram\_pll



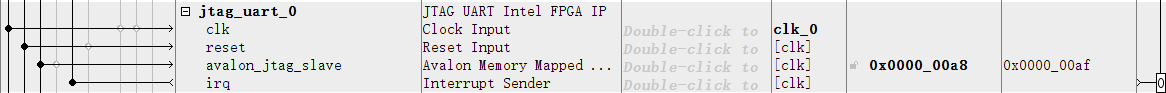
This block generates a PLL for the running of sdram, which is -3 ns before the clock to compensate for the loss in the transmission.

Module: sysid\_qsys\_0



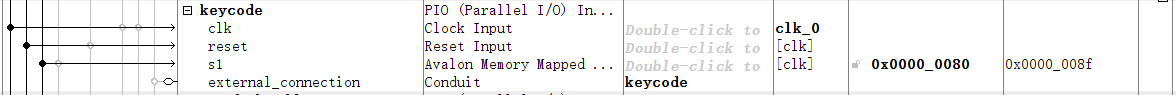
This block is a system id checker to make sure that the id configuration of our hardware part is corresponding to the software part.

Module: jtag\_uart\_0



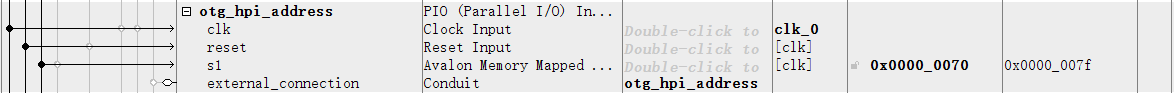
This is so that you can use the terminal of the host computer (the one running eclipse) to communicate with the NIOS II (using print and scan statements in c).

Module: keycode



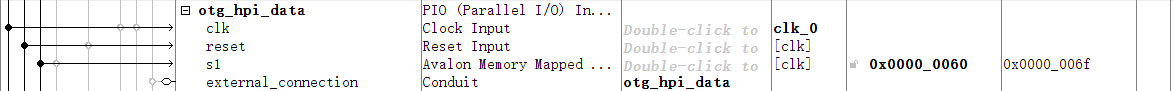
This block is a PIO part with 8-bit output form the keyboard.

Module: otg\_hpi\_address



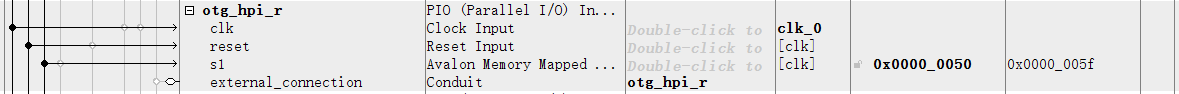
This block is a PIO part with 2-bit output of the address

Module: otg\_hpi\_data



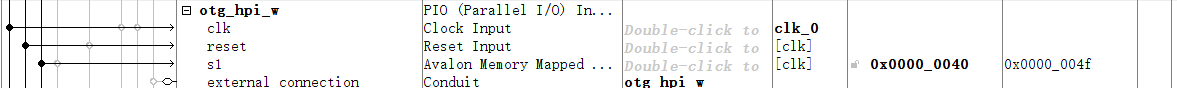
This block is a PIO part with 16-bit input of the data.

Module: otg\_hpi\_r



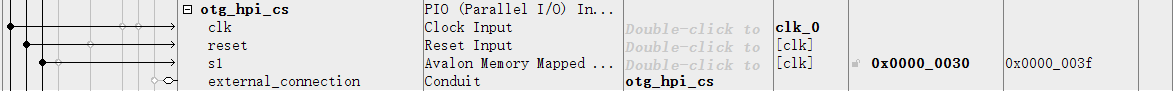
This block is a PIO part with 1-bit output of the read signal.

Module: otg\_hpi\_w



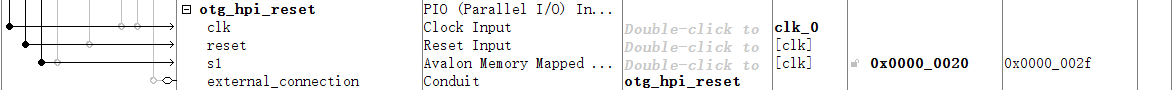
This block is a PIO part with 1-bit output of the write signal.

Module: otg\_hpi\_cs



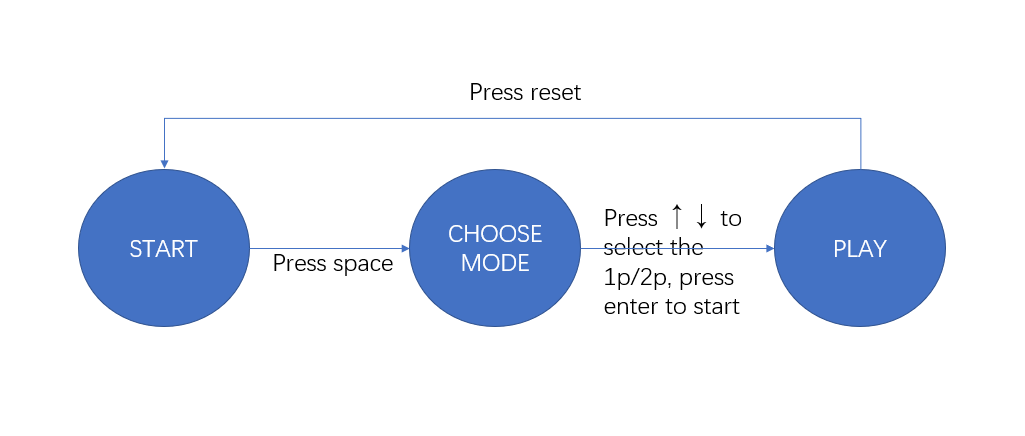
This block is a PIO part with 1-bit output of the chip select signal.

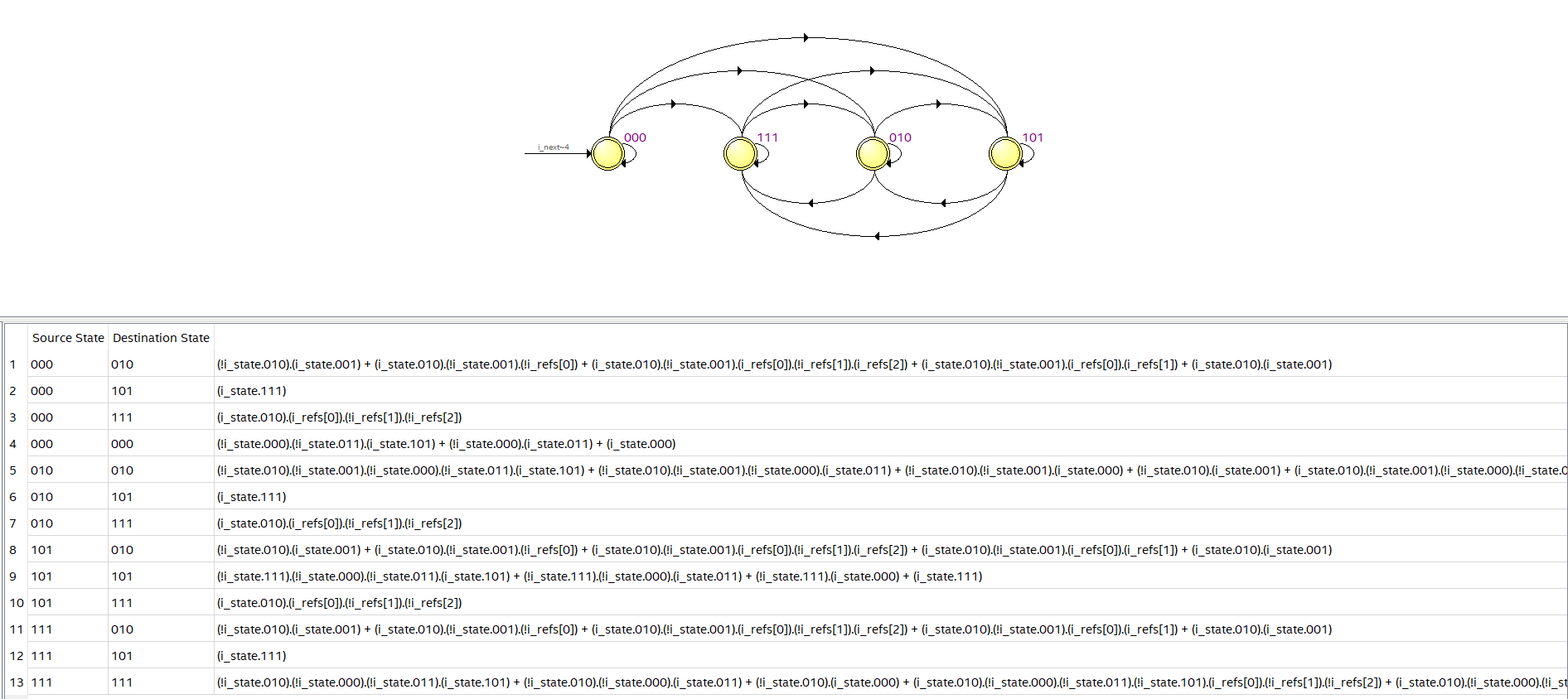
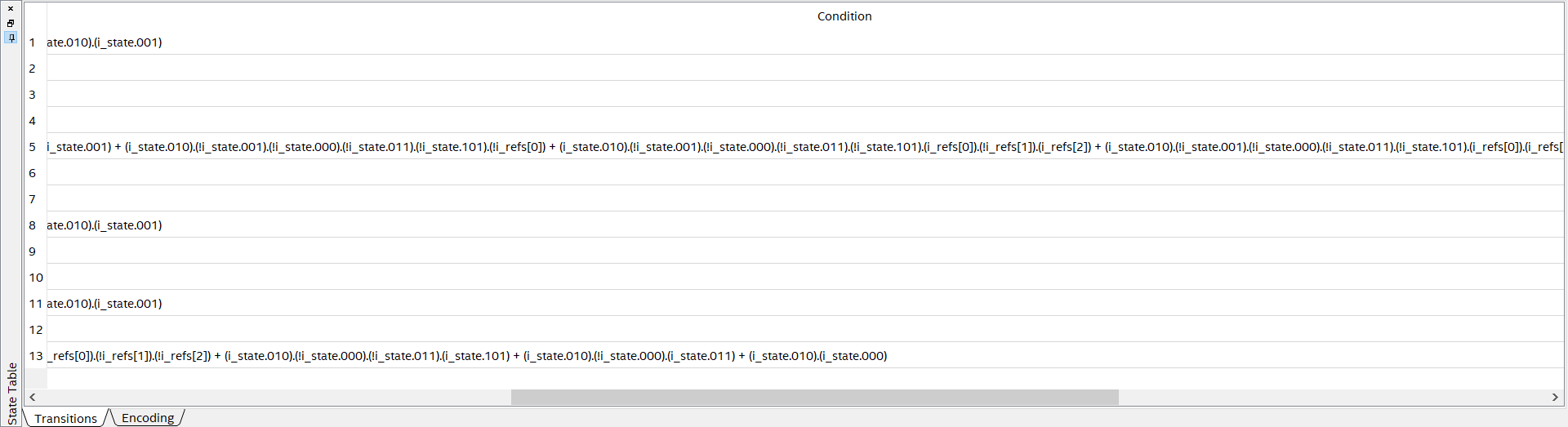
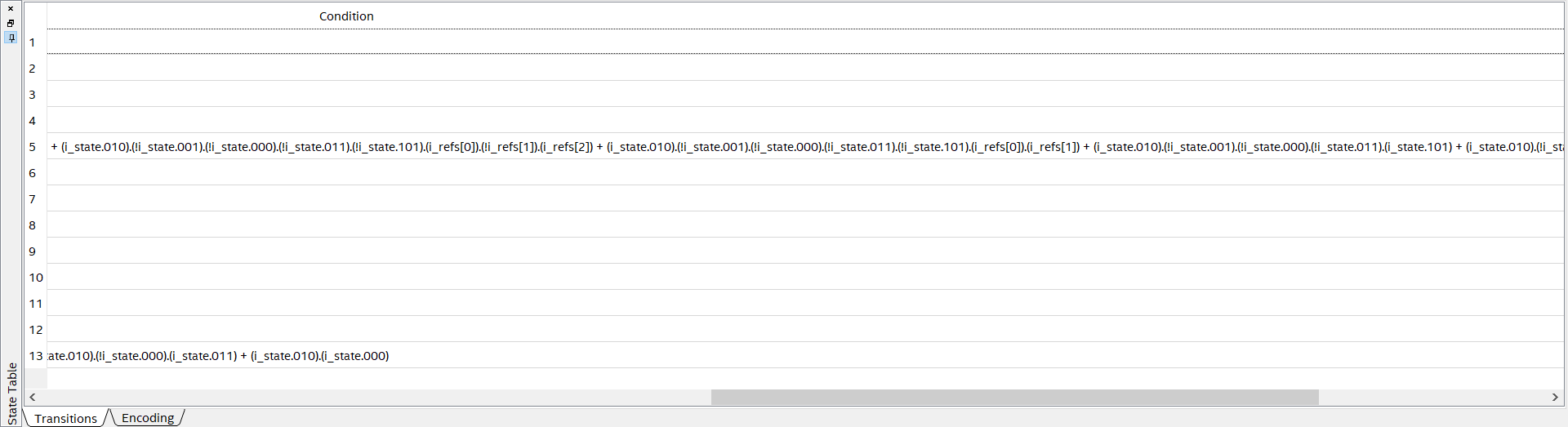
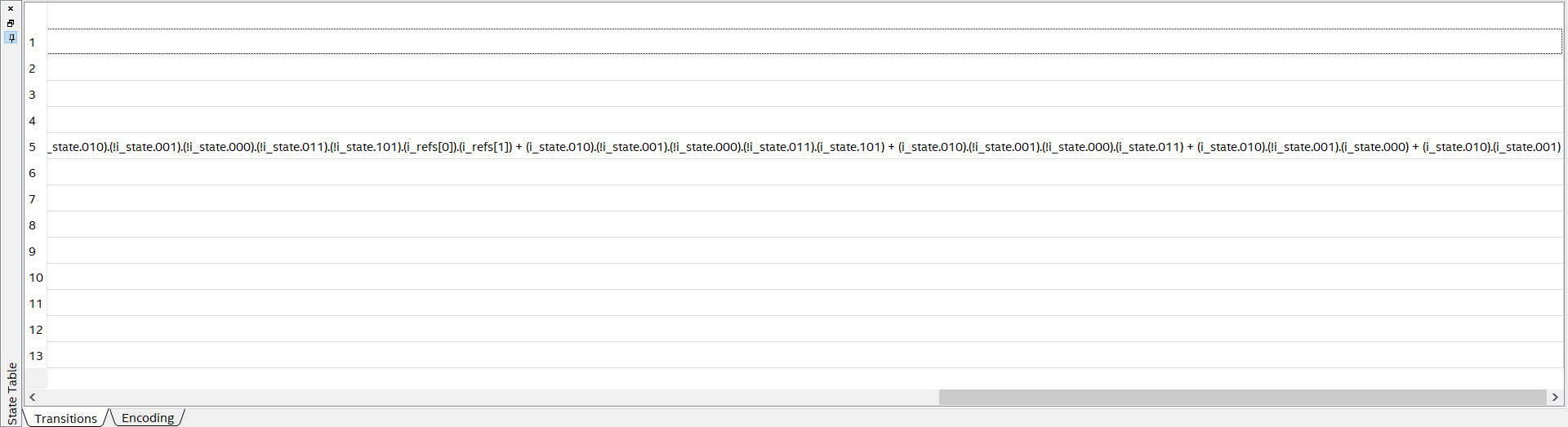
Module: otg\_hpi\_reset



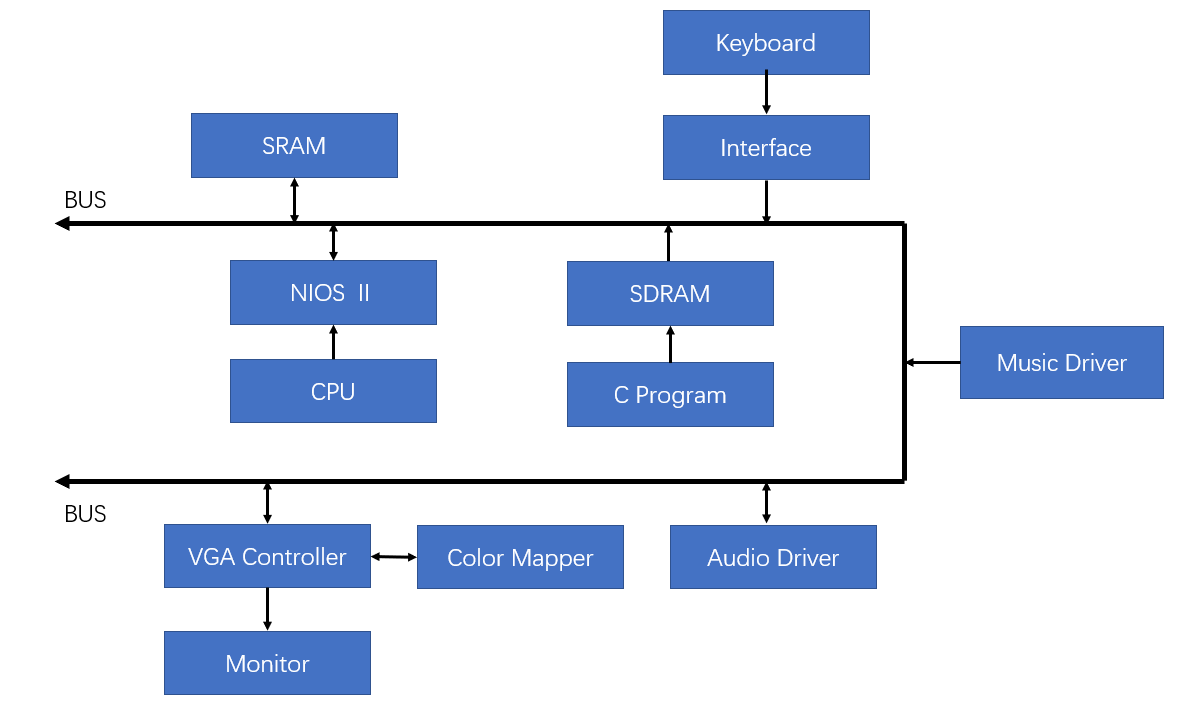
This block is a PIO part with 1-bit output of the reset signal.

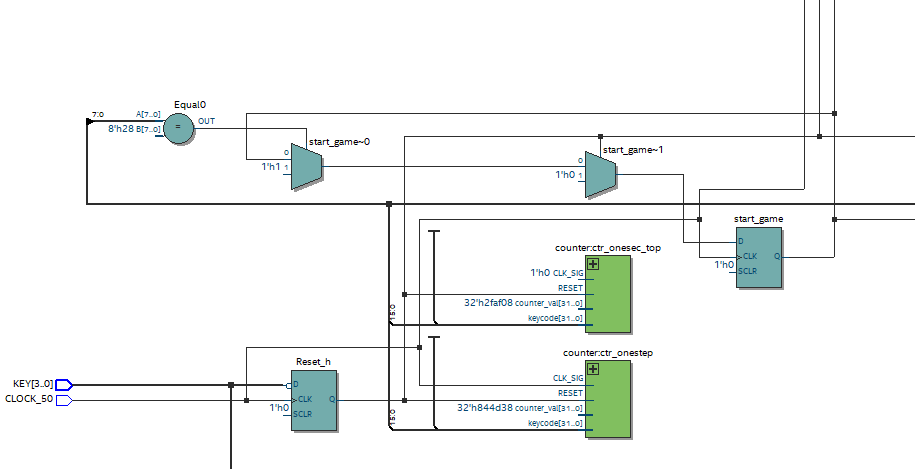
1. Design Procedure/State Machine/Simulation Waveform
   * 1. Overview of the design procedure
2. We use lab 8 as the foundation of our foundation and using the platform designer. We modified the ball.sv and added other modules about background music, pictures and running of the notes.
3. If the player 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.
4. 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.
   * 1. State Machine/Simulation Wave

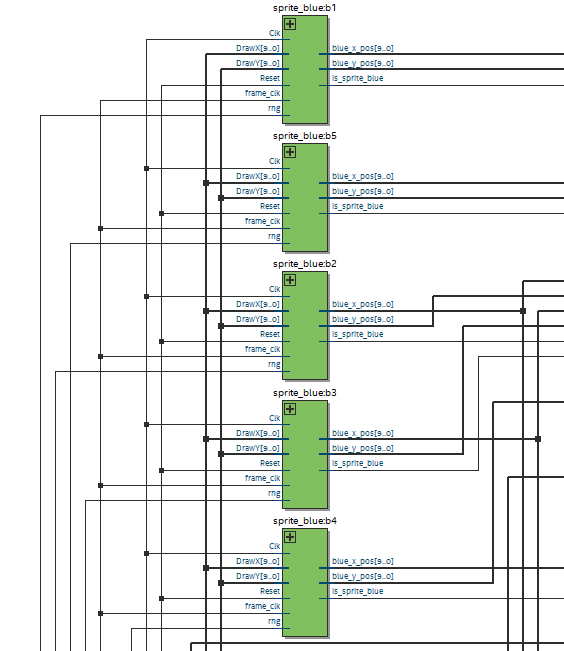
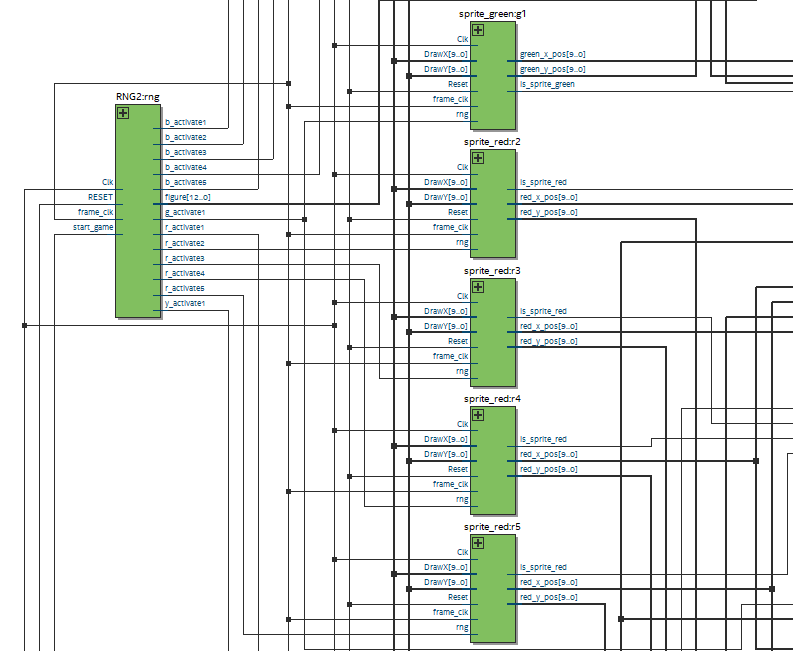
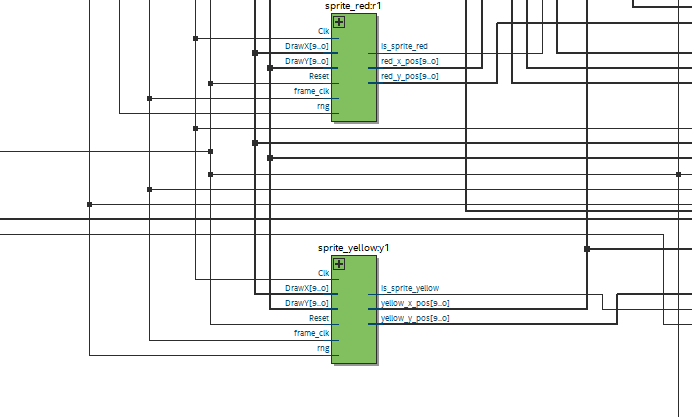
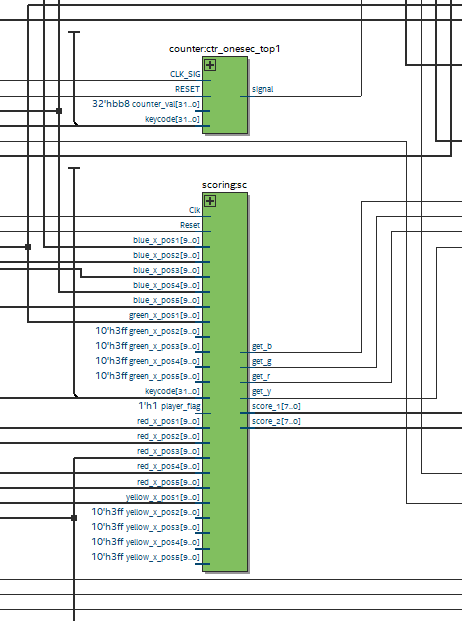
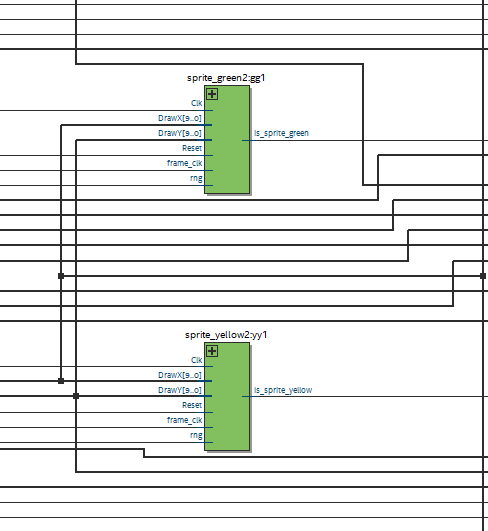
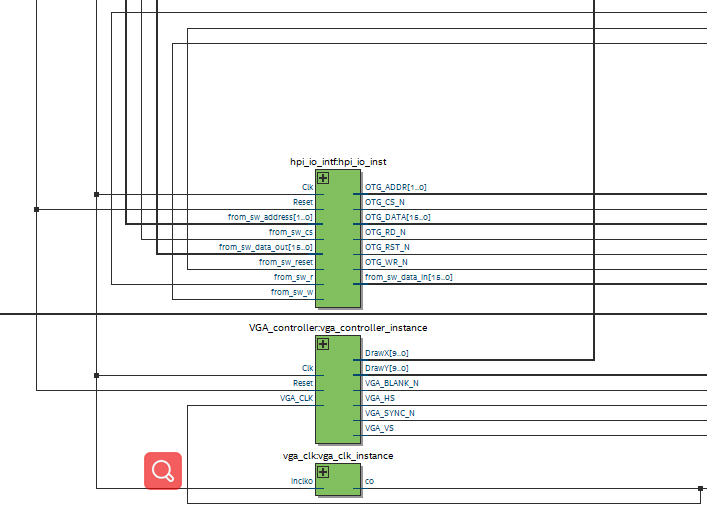
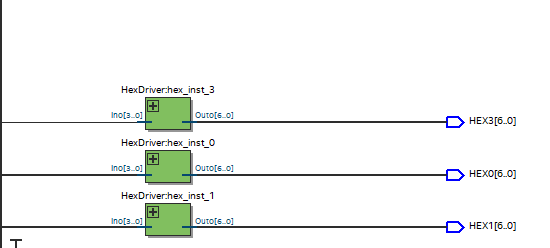
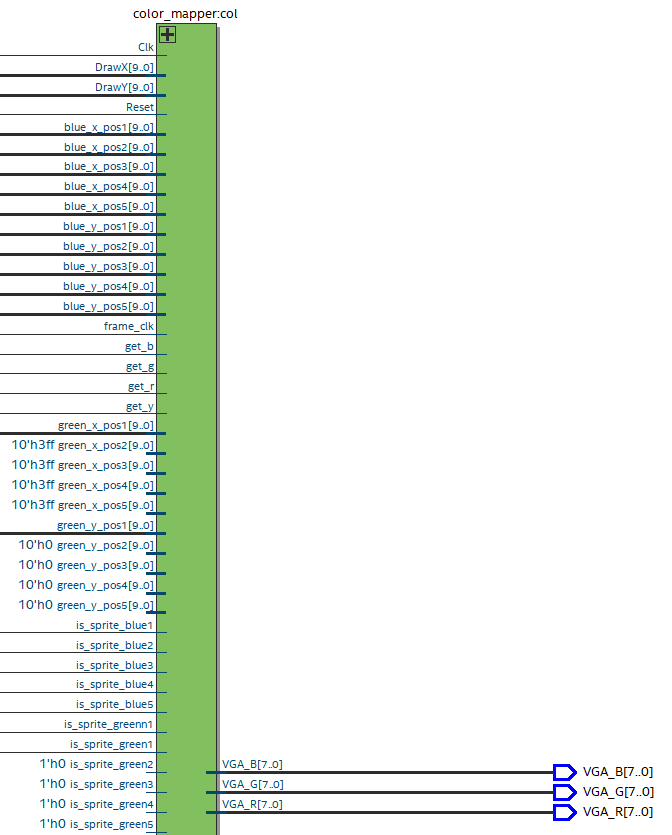
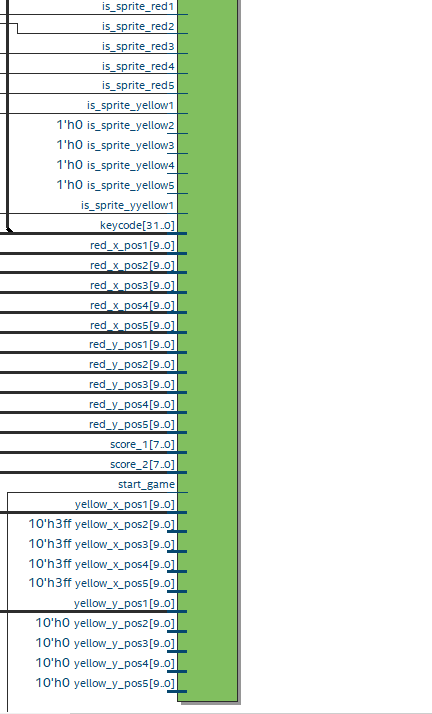
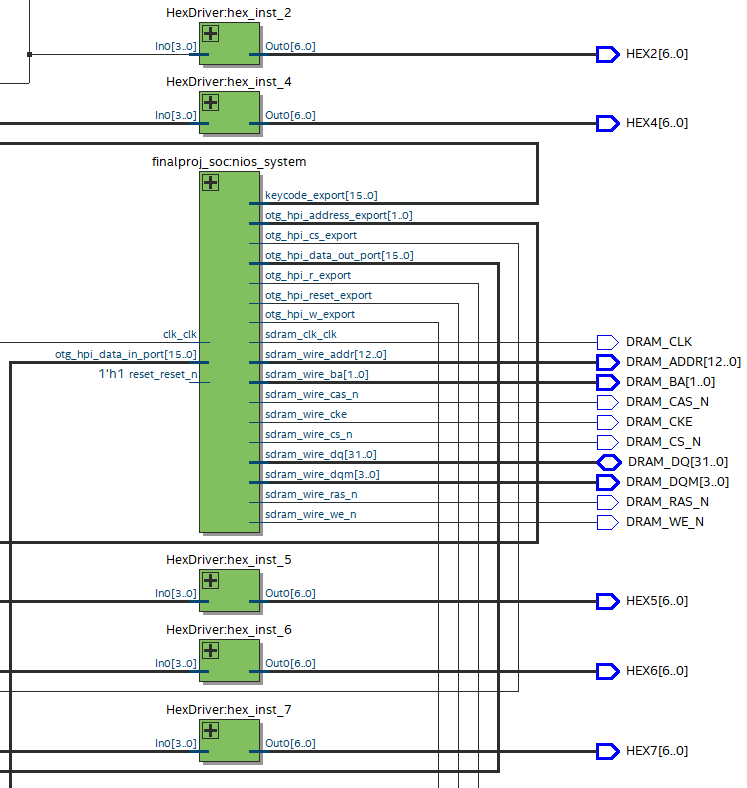
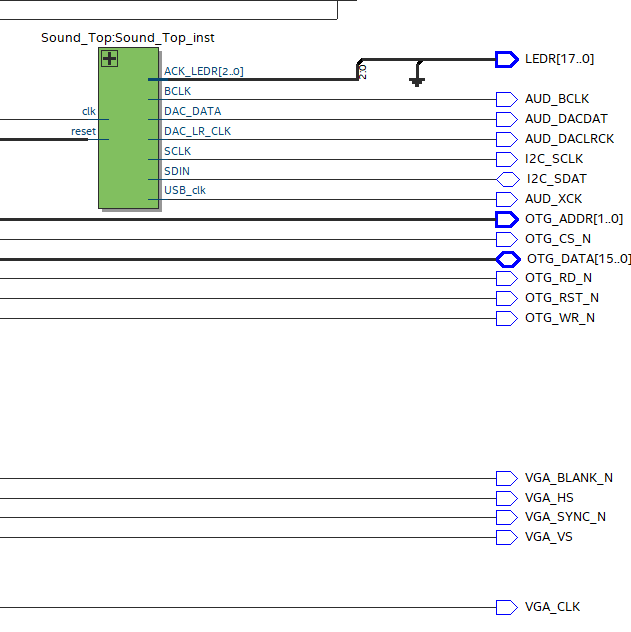


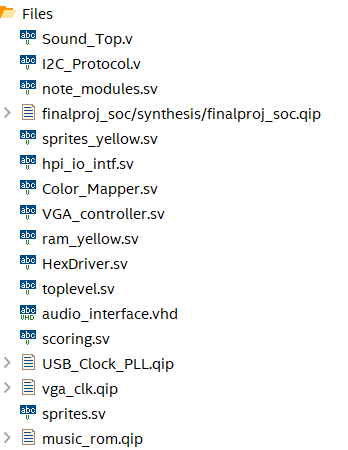
1. Block Diagram





1. SV Code



1. 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 |

1. 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>

1. Conclusion
2. 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.

1. 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.