ECE385

Spring 2020

Final Project

Roguelike RPG ‘Soul Knight’ in SystemVerilog

Yuantao Lu, Yizhen Lu

Section ABD

TA: Nicholas Cebry, Wenjie Pan

Introduction and Overview

In this final project, we incorporated all we learnt during the semester and eventually implemented an Roguelike RPG ‘Soul Knight’ mainly using NIOS-II microprocessor on FPGA.
In the game, the player should use the USB keyboard to control the character to explore through the dungeons, dodging bullets and shooting down all the monsters hidden in the dungeons.
The project uses a USB keyboard as the only input source, while outputting image streams on the VGA monitor and sound effect on a speaker. The software-oriented USB keyboard driver and HPI interface we realized in lab8 are both applied in the project to take care of the interactions between NIOS-II and EZ-OTG microprocessor for basic I/O functionalities between the keyboard and the core processor. The VGA controller from lab8 is also applied here to configure the image info into the form that could be correctly displayed on the monitor. All image data are stored in on-chip memory and audio data stored in flash.

Written Description of Final Project System

Written Description of the entire system
The project mainly consists of the following four components: the NIOS-II and EZ-OTG microprocessors on the FPGA board, the USB keyboard that is connected to EZ-OTG(on chip CY7C67200), the VGA that is connected to the NIOS-II. We apply multiple interfaces and controllers to connect all the components and make them work in correct order. The NIOS-II processor and EZ-OTG processor are both SOCs and are implemented through the Platform Designer as hardware cores of the system. The HPI interface and the USB keyboard driver work together to make sure the functionality of receiving keyboard input. Then, we have a VGA controller to map the info we have correctly onto the monitor to display video game images.

Software Component Summary

At the software end, we are running the USB keyboard driver from lab8, and since the driver is a while loop, which constrained us from using software to implement more functionalities in this project.

Written Description of the USB protocol

Since the keyboard is plugged into the FPGA development board through a USB connector, the keyboard driver has to follow the working USB protocol in order to have the correct inputs and outputs.

To safely communicate between software and hardware, the following functions are critical because they are responsible for the basic I/Os between the two chips, NIOS-II and CY7C67200, manipulated using the software programming language.

IO_read:

This function serves as the connection between NIOS-II microprocessor and chip CY7C67200. It mainly plays with PIO ports on the NIOS-II that take control of the hpi_io_intf.sv module. By setting the address, the cs, the r signals that each correspond to a PIO module on NIOS-II, the driver on the chip will send back the data corresponding to the address back to NIOS-II. The data exchange is done through hpi_io_intf.sv module that serves as the tristate buffer.

Steps in c code:

Set the *otg_hpi_address we wanted to read from;

Set *otg_hpi_cs control signal to 0(active);

Set *otg_hpi_r control signal to 0(active);

Get the data from chip by reading from *otg_hpi_data;

Set *otg_hpi_r control signal to 1(inactive);

Set *otg_hpi_cs control signal to 1(inactive);

IO_write:

This function serves as the connection between NIOS-II microprocessor and chip CY7C67200. It mainly plays with PIO ports on the NIOS-II that take control of the hpi_io_intf.sv module. By setting the address we want to write to, the data we want to write, the cs control bit and the w control signal on the PIO ports on NIOS-II, we are able to write data to a certain register on the chip. The data flow is done through hpi_io_intf.sv module that serves as the tristate buffer.

Steps in c code:
Set the *otg_hpi_address we wanted to write to;
Set the *oth_hpi_data we wanted to write;
Set *otg_hpi_cs control signal to 0(active);
Set *otg_hpi_w control signal to 0(active);
Set *otg_hpi_w control signal to 1(inactive);
Set *otg_hpi_cs control signal to 1(inactive);

Note: The orders of setting those signals in both functions above are following the hpi communication in AN6010 data sheet.

USB_write:
This function writes data to the internal registers of the CY7C67200 USB controller. It first takes the Address of the register as the data and HPI_ADDR as the address and calls IO_write, i.e. writing Address to HPI_ADDR register. It then takes Data we wanted to write to register as the data and HPI_DATA as the address. Note HPI_ADDR and HPI_DATA are registers on the chip. The reason we wanted to write both data and address into registers on the chip is because we do not have direct access to the address on the chip but could only modify registers. Having the above two registers set, EZ-OTG will then store the data in the DATA register into the address in the ADDR register.

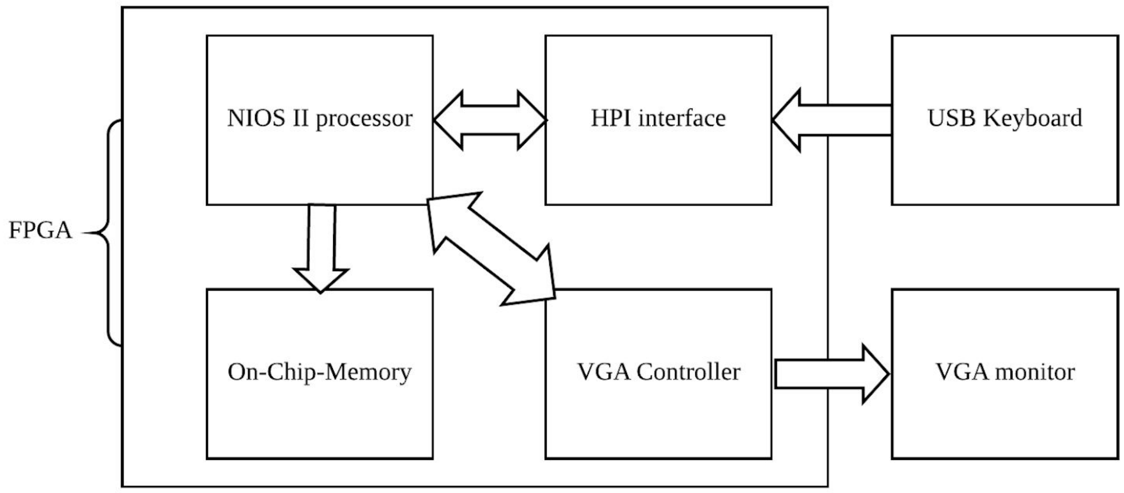
USB_read:
This function reads data from the internal registers of the CY7C67200 USB controller. It first takes the Address of the register as the data and HPI_ADDR as the address and calls IO_write, i.e. writing Address to HPI_ADDR register. EZ-OTG will then fetch the data in the address specified by the HPI_ADDR register and load it into the HPI_DATA register. The next step is to read the data from the HPI_DATA register by calling IO_read.

Hardware Component Summary

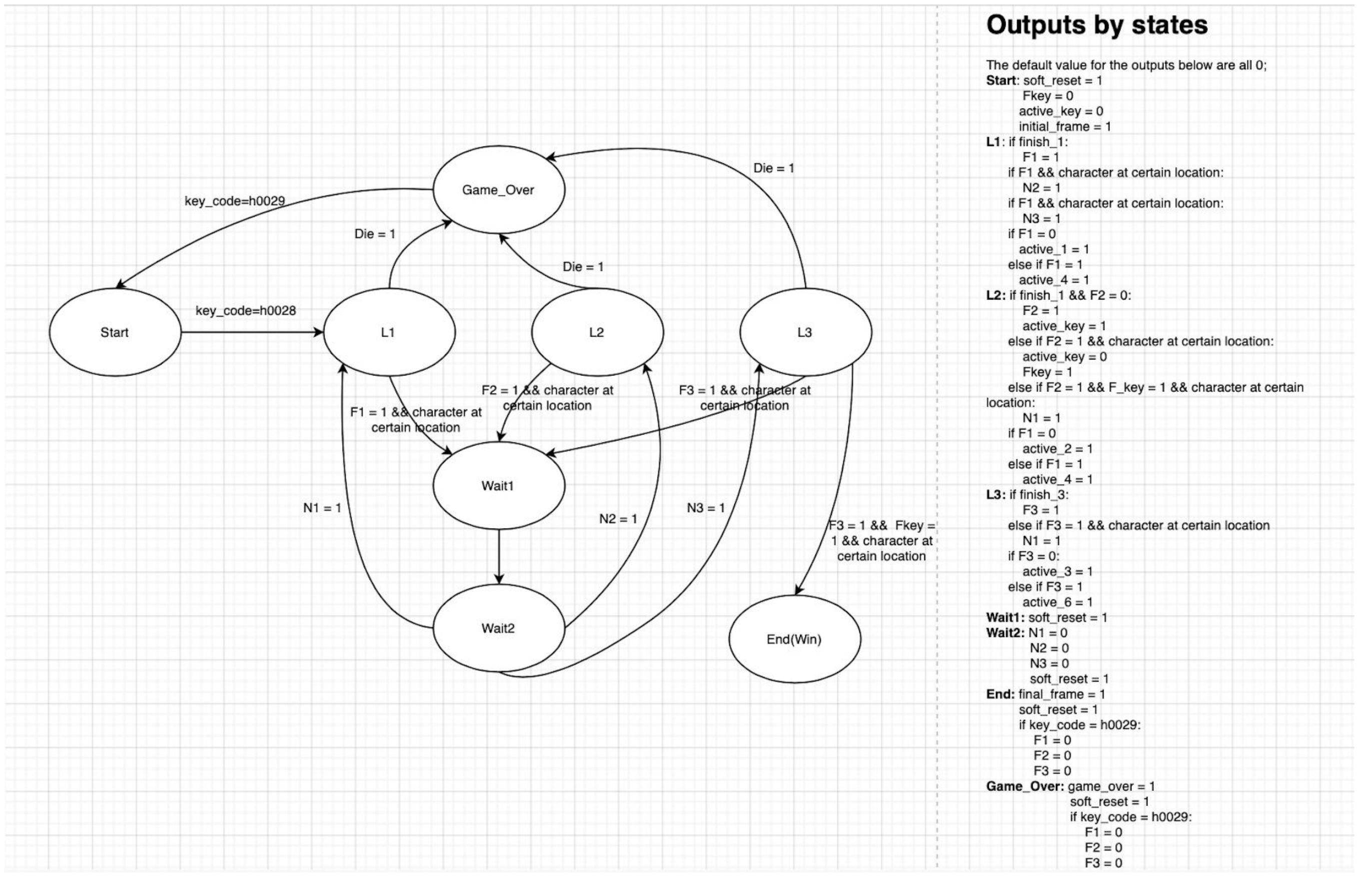
As stated earlier, the software side is completely occupied to run the keyboard driver, the rest of the project systems have to rely on the hardware implementations.

As the block diagram would suggest, hardware ends are responsible for the operation of the games. We have a core NIOS-II processor, which controls the main data flow, then HPI interface and VGA controller are responsible for receiving keyboard inputs and output screen images respectively. Also, all images' data are stored in on-chip memory.

Block Diagrams



Master State Machines


key_code=h0029
Game_Over
Die = 1
Die = 1
Die = 1
Start
key_code=h0028
L1
L2
L3
F2 = 1 && character at certain location
F3 = 1 && character at certain location
F1 = 1 && character at certain location
Wait1
N1 = 1
N2 = 1
N3 = 1
F3 = 1 && Fkey = 1 && character at certain location
Wait2
End(Win)

Outputs by states

```
The default value for the outputs below are all 0;
Start: soft_reset = 1
         Fkey = 0
        active_key = 0
        initial_frame = 1
L1: if finish_1:
         F1 = 1
      if F1 && character at certain location:
         N2 = 1
      if F1 && character at certain location:
         N3 = 1
      if F1 = 0
        active_1 = 1
      else if F1 = 1
        active_4 = 1
L2: if finish_1 && F2 = 0:
         F2 = 1
         active_key = 1
      else if F2 = 1 && character at certain location:
         active_key = 0
         Fkey = 1
      else if F2 = 1 && F_key = 1 && character at certain
location:
         N1 = 1
      if F1 = 0
         active_2 = 1
      else if F1 = 1
         active_4 = 1
L3: if finish_3:
         F3 = 1
      else if F3 = 1 && character at certain location
         N1 = 1
      if F3 = 0:
         active_3 = 1
      else if F3 = 1
         active_6 = 1
Wait1: soft_reset = 1
Wait2: N1 = 0
          N2 = 0
          N3 = 0
          soft_reset = 1
End: final_frame = 1
        soft_reset = 1
        if key_code = h0029:
           F1 = 0
           F2 = 0
           F3 = 0
Game_Over: game_over = 1
                  soft_reset = 1
                  if key_code = h0029:
                     F1 = 0
                     F2 = 0
                     F3 = 0
```

Diagram of the level transition state machine

There are three levels in the dungeon, represented by L1, L2, L3 in the state machine. The state machine will remember whether the player has finished a level or not and will choose to display the monsters or not. If the health of the character drops to zero, the game is over. To win the game, the player must kill all the monsters in the three levels and get the key from level 2. Depending on the conditions, the level 2 will display the keys, the player must control the character to move to the position of the key to pick it up.

With the key, the player must control the character to stand at the middle of the level three to win. At each transition between levels, the position of the player reset through the Wait1, and Wait2 states.

Written Description of the meaning of the outputs:

key_code: the current key from the keyboard that the user is pressing

Die: determined by the character health from the other module

soft_reset: the output from the module that is used to reset the character's position

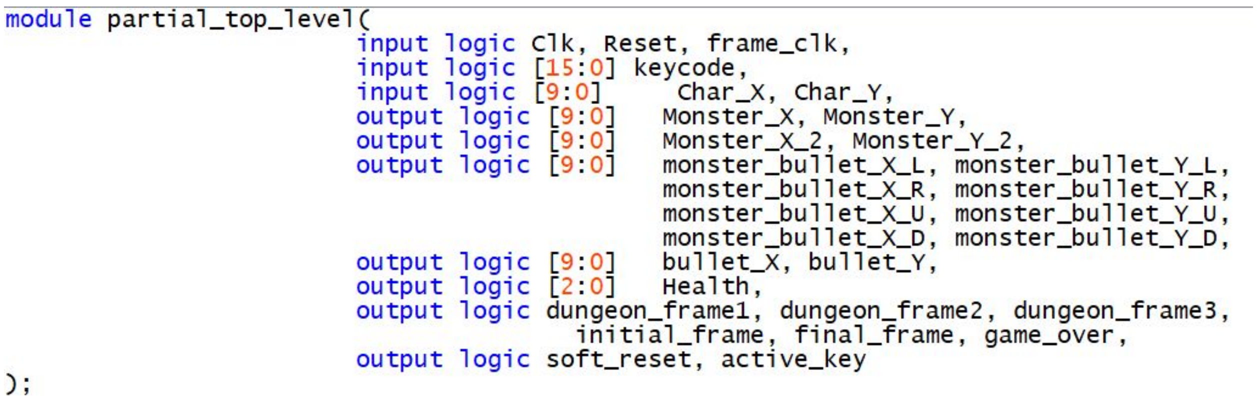
finsih_1: input that becomes high when player first pass level 1
finish_2: input that becomes high when player first pass level 2
finish_3: input that becomes high when player first pass level 3
F1: indicate that the player has passed level 1
F2: indicate that the player has passed level 2
F3: indicate that the player has passed level 3
active_key: output passed to other modules that draws the key in the middle of the screen
Fkey: indicate that the player has found the key
N1: indicate that the next level is level 1
N2: indicate that the next level is level 2
N3: indicate that the next level is level 3
initial_frame: output to other module to display the initial background
active_1: output to other module to display level 1 background and the monsters
active_2: output to other module to display level 2 background and the monsters
active_3: output to other module to display level 3 background and the monsters
active_4: output to other module to display level 1 background
active_5: output to other module to display level 2 background
active_6: output to other module to display level 3 background
final_frame: output to other module to display the "you win" background
game_over: output to other module to display the "game over" background

Written Descriptions of all modules

Module: *FinalProject.sv*
Input: *CLOCK_50, [3:0] KEY, [15:0] OTG_DATA, OTG_INT, [31:0] DRAM_DQ.*
Output: *[6:0] HEX0, [6:0] HEX1, [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B, VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS, [15:0] OTG_DATA, [1:0] OTG_ADDR, OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N, [12:0] DRAM_ADDR, [31:0] DRAM_DQ, [1:0] DRAM_BA, [3:0] DRAM_DQM, DRAM_RAS_N, DRAM_CAS_N, DRAM_CKE, DRAM_WE_N, DRAM_CS_N, DRAM_CLK.*

Description: This is the top level of this lab. It consists of all the hardware necessary for this lab including the module for NIOS-II with the DRAM, the modules for VGA, the module for the interface between NIOS-II and the EZ-OTG.
Purpose: This module is the top level for this lab. It connects all necessary modules together and serves as the top level interface for the user that makes all the inputs and outputs clear.

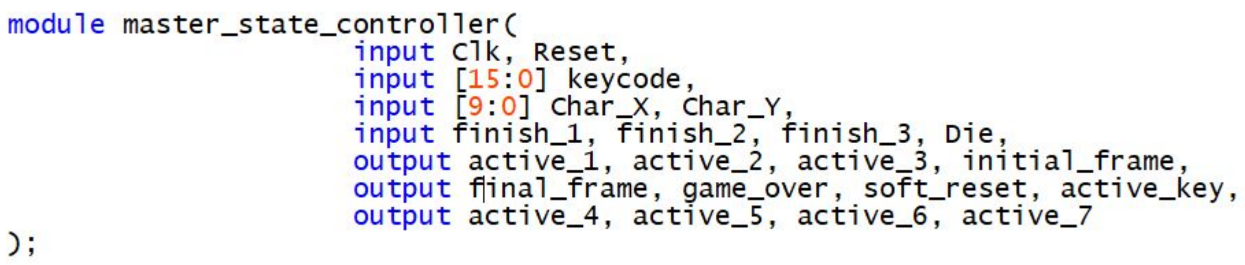
Module: *partial_top_level.sv*



```
module partial_top_level(
                        input logic Clk, Reset, frame_clk,
                        input logic [15:0] keycode,
                        input logic [9:0]     Char_X, Char_Y,
                        output logic [9:0]   Monster_X, Monster_Y,
                        output logic [9:0]   Monster_X_2, Monster_Y_2,
                        output logic [9:0]   monster_bullet_X_L, monster_bullet_Y_L,
                                             monster_bullet_X_R, monster_bullet_Y_R,
                                             monster_bullet_X_U, monster_bullet_Y_U,
                                             monster_bullet_X_D, monster_bullet_Y_D,
                        output logic [9:0]   bullet_X, bullet_Y,
                        output logic [2:0]   Health,
                        output logic dungeon_frame1, dungeon_frame2, dungeon_frame3,
                                       initial_frame, final_frame, game_over,
                        output logic soft_reset, active_key
);
```

Description: This partial top level is responsible for the intrinsic mechanism of the game, where it takes keyboard input and outputs position of characters, monsters and bullets so that the rest of the project can successfully use these positions to map the images of characters, monsters and bullets into their correct positions on the VGA monitor. It also determines which background image is used based on the current progression of the game, which is controlled by the master state machine. Moreover, the health system is incorporated in the module, which keeps calculating and restoring the health value of the character based on the time he was hit.
Purpose: Report positions for everything that needs to be plotted on the screen at realtime.

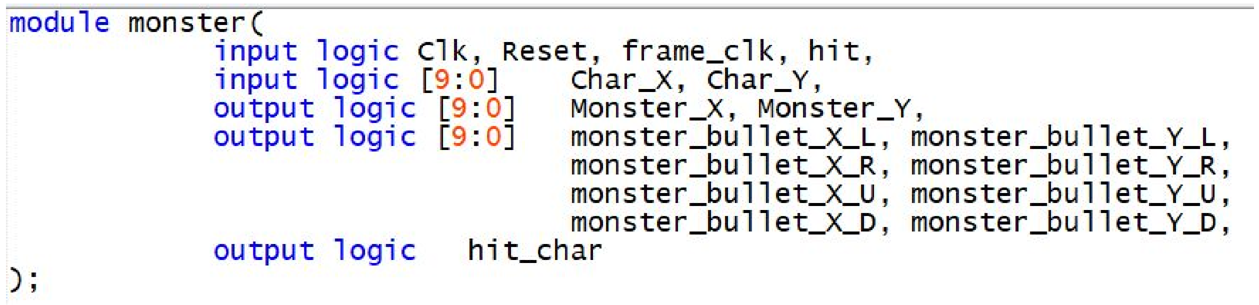
Module: *master_state_controller.sv*



```
module master_state_controller(
                        input Clk, Reset,
                        input [15:0] keycode,
                        input [9:0] Char_X, Char_Y,
                        input finish_1, finish_2, finish_3, Die,
                        output active_1, active_2, active_3, initial_frame,
                        output final_frame, game_over, soft_reset, active_key,
                        output active_4, active_5, active_6, active_7
);
```

Description: This is the master state machine of the project, which controller which level the character is currently at, when to transition to a new level, when to end and restart the game. Its diagram is shown above at the *Mater State Machines* section. Basically, it resets to the main menu, then goes to dungeon level1, after finishing this dungeon, two exits at the dungeon would lead to two other different levels. In the same manner, it can transit through all three levels, and when the character meets the end game conditions, either win or lose, the controller would then send out signals for the VGA controller to display the corresponding ending frame background pictures.
Purpose: Control the progression of the game to have a smooth transition and a fluent user experience.

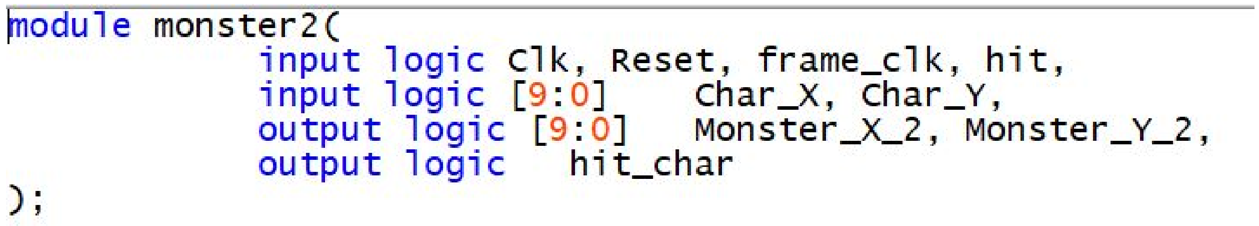
Module: *monster.sv*



```
module monster(
            input logic Clk, Reset, frame_clk, hit,
            input logic [9:0]    Char_X, Char_Y,
            output logic [9:0]   Monster_X, Monster_Y,
            output logic [9:0]   monster_bullet_X_L, monster_bullet_Y_L,
                                 monster_bullet_X_R, monster_bullet_Y_R,
                                 monster_bullet_X_U, monster_bullet_Y_U,
                                 monster_bullet_X_D, monster_bullet_Y_D,
            output logic   hit_char
);
```

Description: In the module, it takes current position for character and monster to design the next step for the monster to move. Also, the monster would continuously shoot out bullets in all four directions, so the positions of these bullets are also outputs of the module. When any of the bullets from the monster hits the character, *hit_char* is raised high so that the health system can receive this information and reflect the correct health value.
Purpose: This module is used to define the motion and attack patterns for the first monster, while sending out position information for everything necessary.

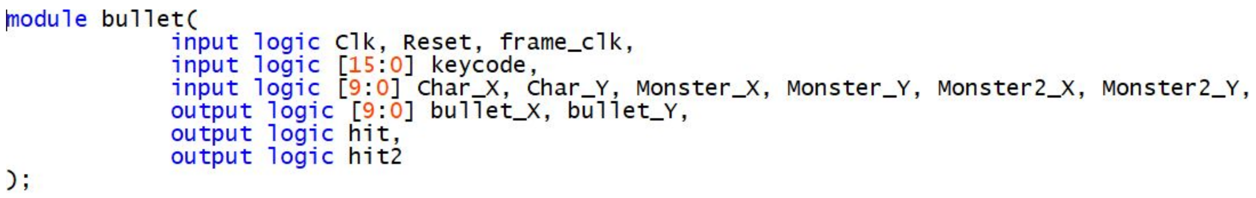
Module: *monster2.sv*



```
module monster2(
            input logic Clk, Reset, frame_clk, hit,
            input logic [9:0]    Char_X, Char_Y,
            output logic [9:0]   Monster_X_2, Monster_Y_2,
            output logic    hit_char
);
```

Description: This module takes character position as input and designs the monster's motion to firmly chase after the character. When it manages to catch the character, *hit_char* is raised high to send information to the health system.
Purpose: Similar to the module above, this is the module that controls the motion and attack pattern for the second monster.

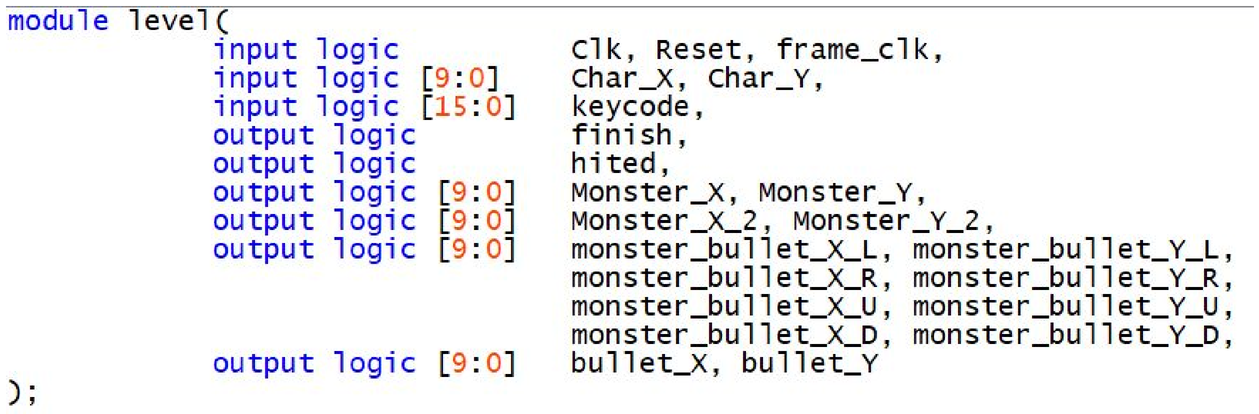
Module: *bullet.sv*



```
module bullet(
            input logic Clk, Reset, frame_clk,
            input logic [15:0] keycode,
            input logic [9:0] Char_X, Char_Y, Monster_X, Monster_Y, Monster2_X, Monster2_Y,
            output logic [9:0] bullet_X, bullet_Y,
            output logic hit,
            output logic hit2
);
```

Description: The module is designed to take inputs for the character, who is shooting, and two monsters, who are the targets of the bullet, the keycode to check for the firing signals. It outputs the current position of the bullet and two hit signals to indicate whether any of the targets got hit.
Purpose: This module depicts the behavior of firing a bullet and gives signals for whether the bullet hits something.

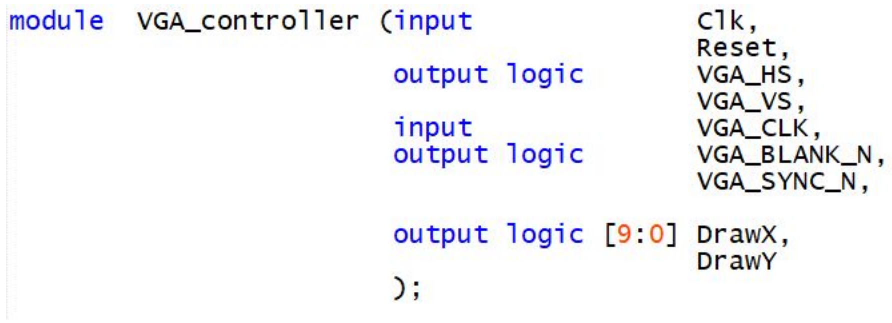
Module: *level.sv*



```
module level(
            input logic         Clk, Reset, frame_clk,
            input logic [9:0]   Char_X, Char_Y,
            input logic [15:0]  keycode,
            output logic        finish,
            output logic        hited,
            output logic [9:0]  Monster_X, Monster_Y,
            output logic [9:0]  Monster_X_2, Monster_Y_2,
            output logic [9:0]  monster_bullet_X_L, monster_bullet_Y_L,
                                monster_bullet_X_R, monster_bullet_Y_R,
                                monster_bullet_X_U, monster_bullet_Y_U,
                                monster_bullet_X_D, monster_bullet_Y_D,
            output logic [9:0]  bullet_X, bullet_Y
);
```

Description: This module combines inputs and outputs of *monster.sv, monster2.sv, bullet.sv* together to form a complete structure of one level, namely what would be inside a dungeon the player would face in the game.
Purpose: This module makes each level a more clear entity so that it could be easier to instantiate at *partial_top_level.sv*.

Module: *VGA_controller.sv*



```
module  VGA_controller (input              Clk,
                                           Reset,
                        output logic       VGA_HS,
                                           VGA_VS,
                        input              VGA_CLK,
                        output logic       VGA_BLANK_N,
                                           VGA_SYNC_N,

                        output logic [9:0] DrawX,
                                           DrawY
                        );
```

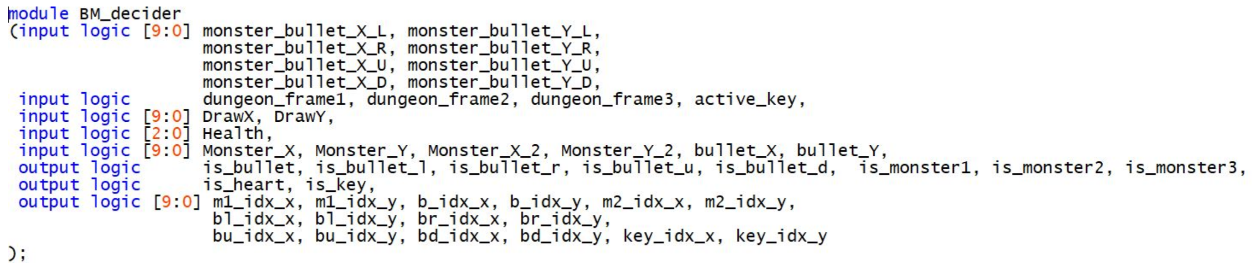
Description: This module takes necessary input VGA signals and outputs the x, y index of rendering concurrently. The VGA_HS is the horizontal pulse that becomes low when a row is rendered and VGA_VS is the vertical pulse that becomes low when a specific row corresponds to an index y is rendered. We can use this signal to determine whether we need to change a frame.
Purpose: This module is mainly the driver for the VGA that updates the necessary VGA signals at each VGA_CLK raising edge.

Module: *ram.sv*
Inputs: [3:0] data_In, [18:0] write_address, read_address,
    input we, Clk,

Outputs: [3:0] data_Out
Description: These modules read in the sprites data that contains all images used in the project and store them at the on-chip memory. The file includes modules that store images for the character, for three monsters, three backgrounds, and the golden key.
Purpose: Store image data at on-chip memory.

Module: *Palette.sv*
Input: [3:0] data_in,
Output: [7:0] Red, Green, Blue
Description: This module is an interpretation for the data stored by *ram.sv*. Each data in the memory corresponds to a different element color, so the palettes for different images are also different. Thus, it contains palettes for characters, monsters and background, the same as the types for the *ram.sv*.
Purpose: Map the image data in the on-chip memory with its corresponding color in red, blue and green, so that the image can be displayed without major distortions.

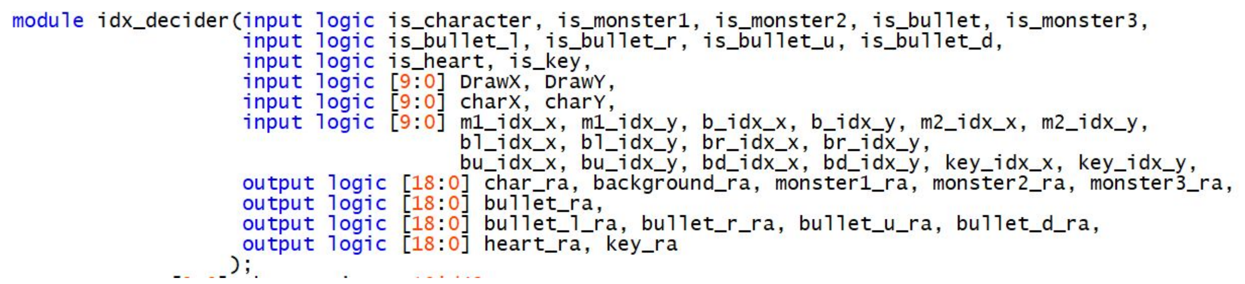
Module: *BM_decider.sv*



```
module BM_decider
(input logic [9:0] monster_bullet_X_L, monster_bullet_Y_L,
                   monster_bullet_X_R, monster_bullet_Y_R,
                   monster_bullet_X_U, monster_bullet_Y_U,
                   monster_bullet_X_D, monster_bullet_Y_D,
 input logic       dungeon_frame1, dungeon_frame2, dungeon_frame3, active_key,
 input logic [9:0] DrawX, DrawY,
 input logic [2:0] Health,
 input logic [9:0] Monster_X, Monster_Y, Monster_X_2, Monster_Y_2, bullet_X, bullet_Y,
 output logic      is_bullet, is_bullet_l, is_bullet_r, is_bullet_u, is_bullet_d,  is_monster1, is_monster2, is_monster3,
 output logic      is_heart, is_key,
 output logic [9:0] m1_idx_x, m1_idx_y, b_idx_x, b_idx_y, m2_idx_x, m2_idx_y,
                    bl_idx_x, bl_idx_y, br_idx_x, br_idx_y,
                    bu_idx_x, bu_idx_y, bd_idx_x, bd_idx_y, key_idx_x, key_idx_y
);
```

Description: This module takes in the pixel that is being drawn and positions for all items that need to be plotted on the screen. Then it decides which item is at the position of this pixel, which means the pixel should draw that item.
Purpose: This module decides which item the pixel is drawing.

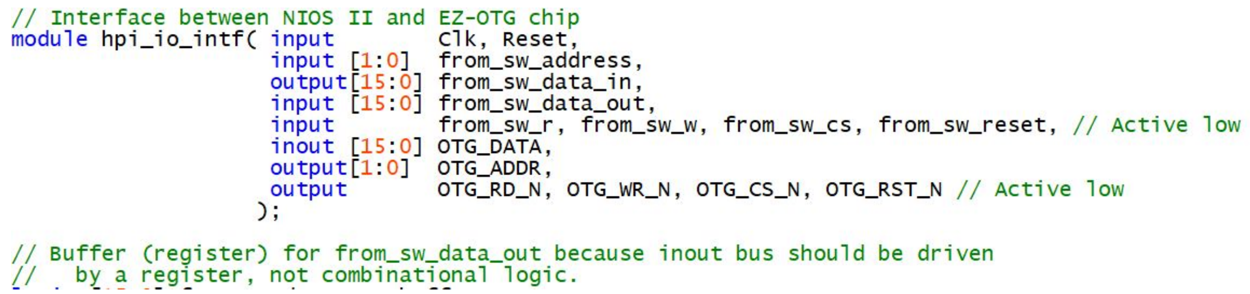
Module: *idx_decider.sv*



```
module idx_decider(input logic is_character, is_monster1, is_monster2, is_bullet, is_monster3,
                   input logic is_bullet_l, is_bullet_r, is_bullet_u, is_bullet_d,
                   input logic is_heart, is_key,
                   input logic [9:0] DrawX, DrawY,
                   input logic [9:0] charX, charY,
                   input logic [9:0] m1_idx_x, m1_idx_y, b_idx_x, b_idx_y, m2_idx_x, m2_idx_y,
                                     bl_idx_x, bl_idx_y, br_idx_x, br_idx_y,
                                     bu_idx_x, bu_idx_y, bd_idx_x, bd_idx_y, key_idx_x, key_idx_y,
                   output logic [18:0] char_ra, background_ra, monster1_ra, monster2_ra, monster3_ra,
                   output logic [18:0] bullet_ra,
                   output logic [18:0] bullet_l_ra, bullet_r_ra, bullet_u_ra, bullet_d_ra,
                   output logic [18:0] heart_ra, key_ra
                  );
```

Description: The module takes in input for what the pixel should represent, and then based on their positions, decides where this pixel is at with respect to the image of the item it represents, so that with this index, we can dig into on-chip memory to find the color for that pixel.
Purpose: This module determines which index the pixel should correspond to in the ram for the item it represents.

Module: *hpi_io_intf.sv*

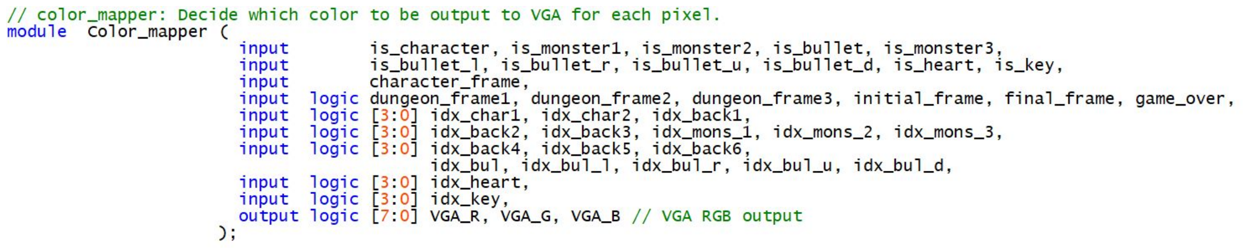


```
// Interface between NIOS II and EZ-OTG chip
module hpi_io_intf( input        Clk, Reset,
                    input [1:0]  from_sw_address,
                    output[15:0] from_sw_data_in,
                    input [15:0] from_sw_data_out,
                    input        from_sw_r, from_sw_w, from_sw_cs, 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
                   );

// Buffer (register) for from_sw_data_out because inout bus should be driven
//   by a register, not combinational logic.
```

Description: This module takes the input from the NIOS-II that is generally the port for the PIOs and outputs signals that connect the EZ-OTG microprocessor.
Purpose: This module serves as the tristate buffer that is the I/O interface between NIOS-II microprocessor and EZ-OTG microprocessor. Hardware data exchange between those two processors is done in the way specified by this module. The software is able to control the PIO ports on the NIOS-II in order to make changes to those output signals that are connected to EZ-OTG.

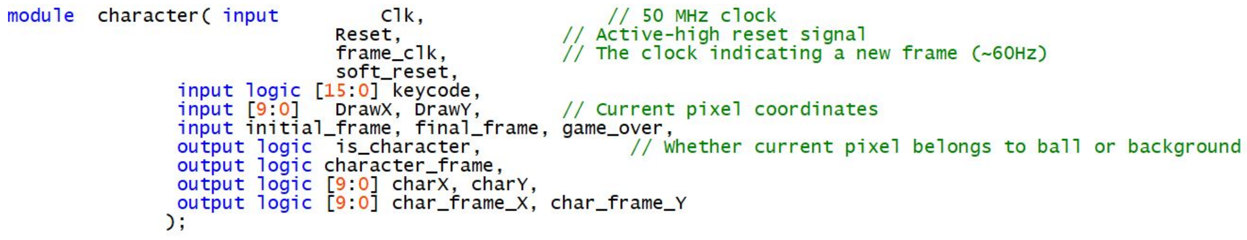
Module: *color_mapper.sv*



```
// color_mapper: Decide which color to be output to VGA for each pixel.
module  Color_mapper (
                        input        is_character, is_monster1, is_monster2, is_bullet, is_monster3,
                        input        is_bullet_l, is_bullet_r, is_bullet_u, is_bullet_d, is_heart, is_key,
                        input        character_frame,
                        input  logic dungeon_frame1, dungeon_frame2, dungeon_frame3, initial_frame, final_frame, game_over,
                        input  logic [3:0] idx_char1, idx_char2, idx_back1,
                        input  logic [3:0] idx_back2, idx_back3, idx_mons_1, idx_mons_2, idx_mons_3,
                        input  logic [3:0] idx_back4, idx_back5, idx_back6,
                                           idx_bul, idx_bul_l, idx_bul_r, idx_bul_u, idx_bul_d,
                        input  logic [3:0] idx_heart,
                        input  logic [3:0] idx_key,
                        output logic [7:0] VGA_R, VGA_G, VGA_B // VGA RGB output
                      );
```

Description: In the module, *palette.sv* are instantiated to receive color data for different images. From the *BM_decider* and *idx_decider*, we can get the color data for the pixel, then we decide, based on priority, which item the pixel should display at top. This gives the final color to be sent to VGA.
Purpose: Decide which color is the output to VGA for each pixel.

Module: *character.sv*



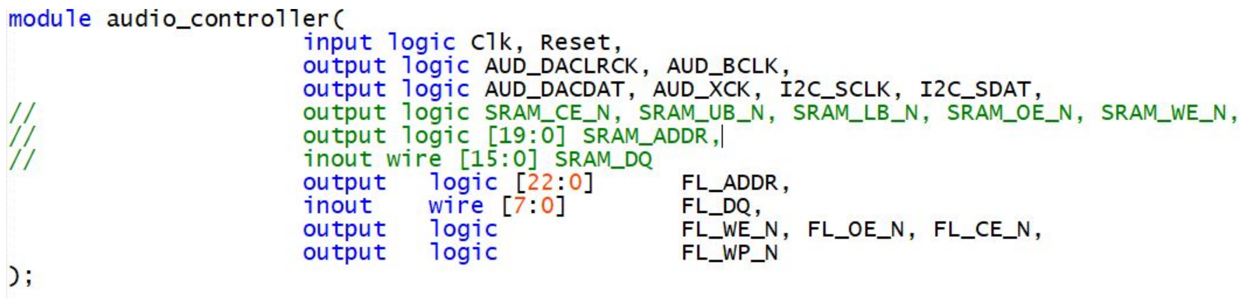
```
module  character( input          Clk,                // 50 MHz clock
                             Reset,              // Active-high reset signal
                             frame_clk,          // The clock indicating a new frame (~60Hz)
                             soft_reset,
               input logic [15:0] keycode,
               input [9:0]   DrawX, DrawY,       // Current pixel coordinates
               input initial_frame, final_frame, game_over,
               output logic  is_character,             // Whether current pixel belongs to ball or background
               output logic character_frame,
               output logic [9:0] charX, charY,
               output logic [9:0] char_frame_X, char_frame_Y
              );
```

Description: This module is the interface for the moving of character in the VGA. When VGA is rendering the screen pixel by pixel, it determines whether the current pixel defined by *Draw_X* and *Draw_Y* is the character. It introduced the *frame_clk* that determines whether the frame is changed. The motion is controlled by the *keycode* input, which is from the USB keyboard.
Purpose: This module is the control of the character movement on the screen. It tells VGA at each frame whether to render a pixel to be the character or not.

Module: *vga_clk.v*
Input: inclk0.
Output: c0.
Description: Generates a VGA clock via PLL that runs the VGA controller.
Purpose: We need to update the screen at around 60Hz, so we do not use the general clock but this VGA clock generated from PLL with 25MHz that has 800 * 525 clock edges per frame. Since 800 * 525 * 60Hz is around 25MHz.

Module: *HexDriver.sv*
Input: [3:0] In0
Output: [6:0] Out0
Description: This module converts 4-bit input value into 7-bit value in LED format.
Purpose: In order to show the pressed keycode on the FPGA board, we use this module to convert the keycode value into LED form.

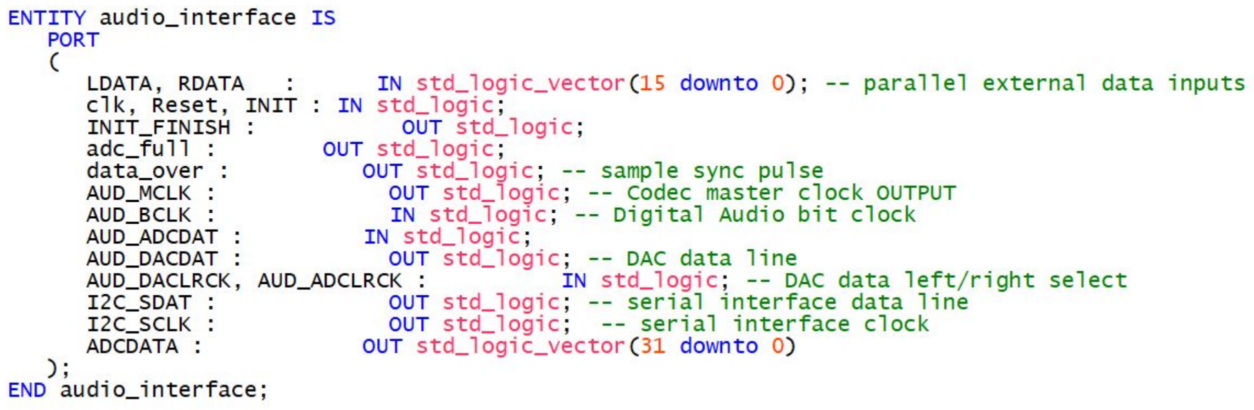
<u>Extra modules that are not included in the final product</u>
Module: *audio_controller.sv*



```
module audio_controller(
                    input logic Clk, Reset,
                    output logic AUD_DACLRCK, AUD_BCLK,
                    output logic AUD_DACDAT, AUD_XCK, I2C_SCLK, I2C_SDAT,
//                  output logic SRAM_CE_N, SRAM_UB_N, SRAM_LB_N, SRAM_OE_N, SRAM_WE_N,
//                  output logic [19:0] SRAM_ADDR,
//                  inout wire [15:0] SRAM_DQ
                    output   logic [22:0]      FL_ADDR,
                    inout    wire [7:0]        FL_DQ,
                    output   logic             FL_WE_N, FL_OE_N, FL_CE_N,
                    output   logic             FL_WP_N
);
```

Description: This module interacts with the flash memory(SRAM) and the audio interface provided on the website.

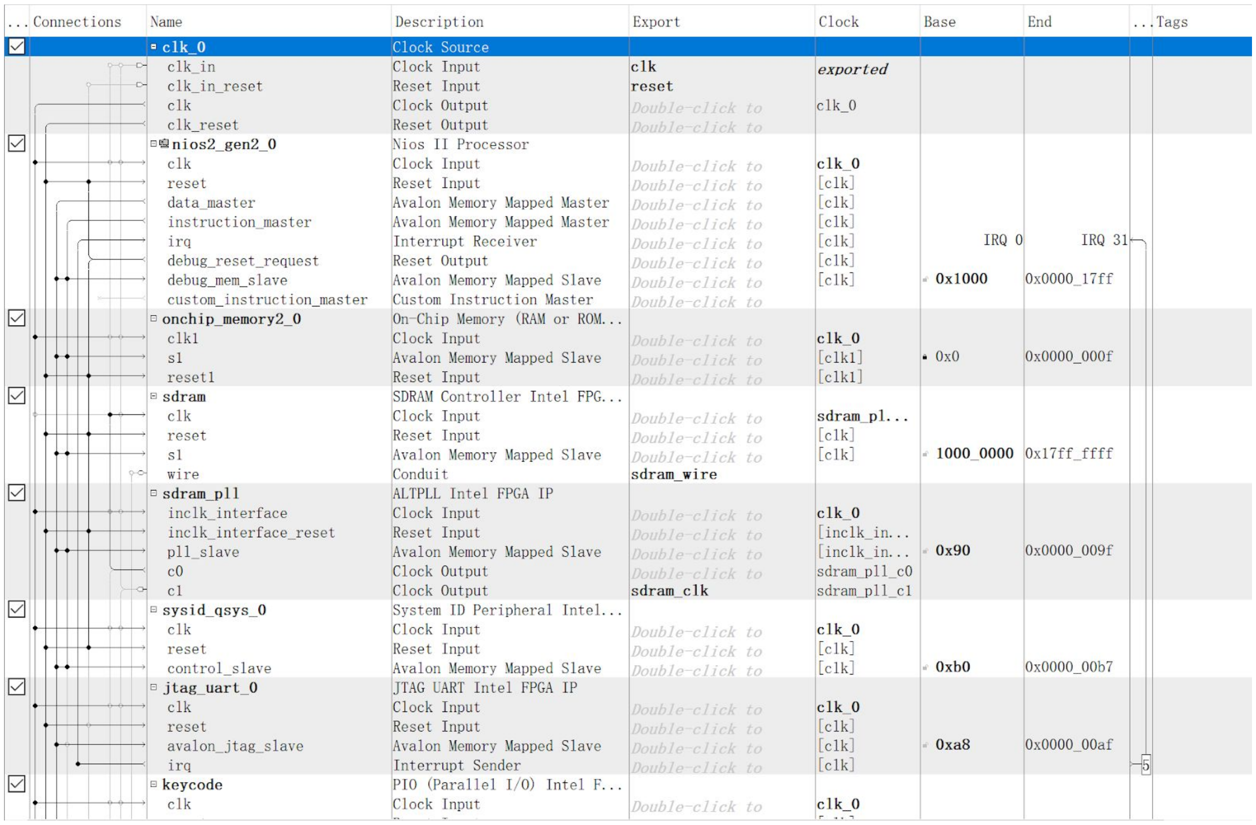
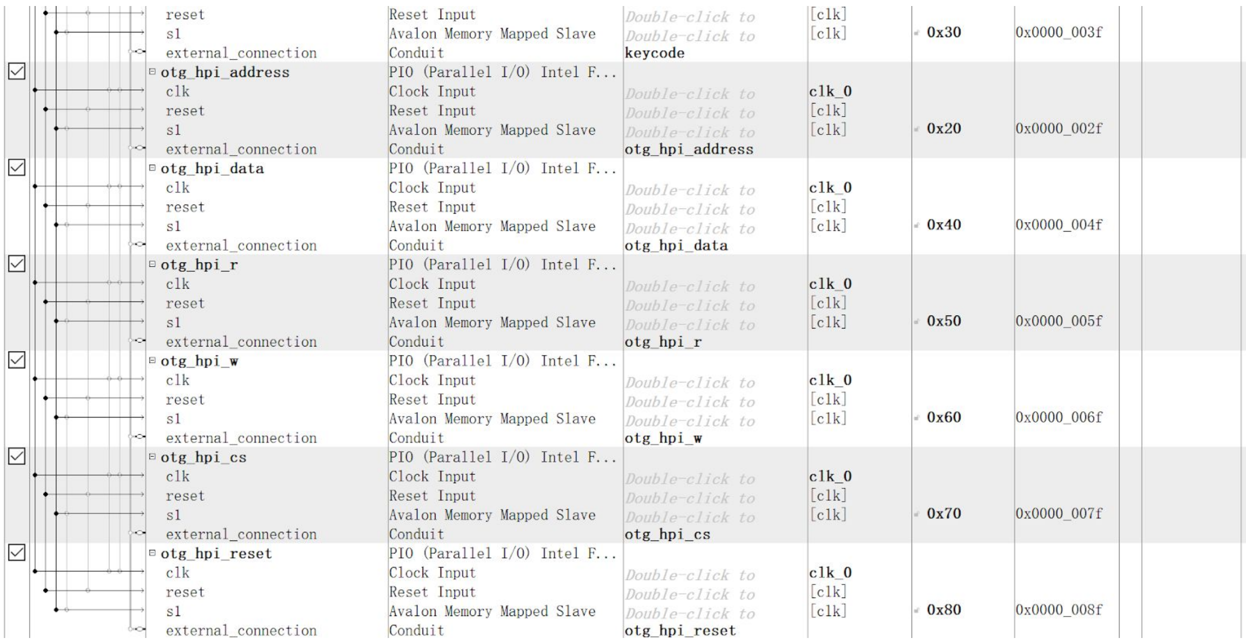
Module: *audio_interface.vhd*



```
ENTITY audio_interface IS
   PORT
   (
      LDATA, RDATA   :      IN std_logic_vector(15 downto 0); -- parallel external data inputs
      clk, Reset, INIT : IN std_logic;
      INIT_FINISH :           OUT std_logic;
      adc_full :        OUT std_logic;
      data_over :          OUT std_logic; -- sample sync pulse
      AUD_MCLK :             OUT std_logic; -- Codec master clock OUTPUT
      AUD_BCLK :             IN std_logic; -- Digital Audio bit clock
      AUD_ADCDAT :         IN std_logic;
      AUD_DACDAT :           OUT std_logic; -- DAC data line
      AUD_DACLRCK, AUD_ADCLRCK :          IN std_logic; -- DAC data left/right select
      I2C_SDAT :             OUT std_logic; -- serial interface data line
      I2C_SCLK :             OUT std_logic;  -- serial interface clock
      ADCDATA :            OUT std_logic_vector(31 downto 0)
   );
END audio_interface;
```

Description: This file is provided on the course website. This entity implements A/D and D/A capability on the Altera DE2 WM8731 Audio Codec

<u>*SOC Descriptions*</u>

...	Connections	Name	Description	Export	Clock	Base	End	...	Tags
☑		**clk_0**	Clock Source						
		clk_in	Clock Input	**clk**	*exported*				
		clk_in_reset	Reset Input	**reset**					
		clk	Clock Output	*Double-click to*	clk_0				
		clk_reset	Reset Output	*Double-click to*					
☑		**nios2_gen2_0**	Nios II Processor						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		data_master	Avalon Memory Mapped Master	*Double-click to*	[clk]				
		instruction_master	Avalon Memory Mapped Master	*Double-click to*	[clk]				
		irq	Interrupt Receiver	*Double-click to*	[clk]	IRQ 0	IRQ 31		
		debug_reset_request	Reset Output	*Double-click to*	[clk]				
		debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x1000**	0x0000_17ff		
		custom_instruction_master	Custom Instruction Master	*Double-click to*					
☑		**onchip_memory2_0**	On-Chip Memory (RAM or ROM...						
		clk1	Clock Input	*Double-click to*	**clk_0**				
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk1]	0x0	0x0000_000f		
		reset1	Reset Input	*Double-click to*	[clk1]				
☑		**sdram**	SDRAM Controller Intel FPG...						
		clk	Clock Input	*Double-click to*	**sdram_pl...**				
		reset	Reset Input	*Double-click to*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**1000_0000**	0x17ff_ffff		
		wire	Conduit	**sdram_wire**					
☑		**sdram_pll**	ALTPLL Intel FPGA IP						
		inclk_interface	Clock Input	*Double-click to*	**clk_0**				
		inclk_interface_reset	Reset Input	*Double-click to*	[inclk_in...				
		pll_slave	Avalon Memory Mapped Slave	*Double-click to*	[inclk_in...	**0x90**	0x0000_009f		
		c0	Clock Output	*Double-click to*	sdram_pll_c0				
		c1	Clock Output	**sdram_clk**	sdram_pll_c1				
☑		**sysid_qsys_0**	System ID Peripheral Intel...						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		control_slave	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0xb0**	0x0000_00b7		
☑		**jtag_uart_0**	JTAG UART Intel FPGA IP						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		avalon_jtag_slave	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0xa8**	0x0000_00af		
		irq	Interrupt Sender	*Double-click to*	[clk]			5	
☑		**keycode**	PIO (Parallel I/O) Intel F...						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x30**	0x0000_003f		
		external_connection	Conduit	**keycode**					
☑		**otg_hpi_address**	PIO (Parallel I/O) Intel F...						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x20**	0x0000_002f		
		external_connection	Conduit	**otg_hpi_address**					
☑		**otg_hpi_data**	PIO (Parallel I/O) Intel F...						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x40**	0x0000_004f		
		external_connection	Conduit	**otg_hpi_data**					
☑		**otg_hpi_r**	PIO (Parallel I/O) Intel F...						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x50**	0x0000_005f		
		external_connection	Conduit	**otg_hpi_r**					
☑		**otg_hpi_w**	PIO (Parallel I/O) Intel F...						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x60**	0x0000_006f		
		external_connection	Conduit	**otg_hpi_w**					
☑		**otg_hpi_cs**	PIO (Parallel I/O) Intel F...						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x70**	0x0000_007f		
		external_connection	Conduit	**otg_hpi_cs**					
☑		**otg_hpi_reset**	PIO (Parallel I/O) Intel F...						
		clk	Clock Input	*Double-click to*	**clk_0**				
		reset	Reset Input	*Double-click to*	[clk]				
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x80**	0x0000_008f		
		external_connection	Conduit	**otg_hpi_reset**					

Functions of each block in the System-level block diagram.

clk_0:

This block generates the main clock signal for all other modules.

nois2_gen2_0:

This block represents the NIOS-II economic version. It is the microprocessor we used in this lab, which performs the computing and processing job in the hardware.

onchip_memory2_0:

It is the memory corresponding to the NIOS-II for loading and storing data that requires high-level performance.

sdram:

This block represents the sdram module. It makes a connection with the processor, so the system can use the memory specified by the sdram.

sdram_pll:

This block represents the PLL for the system. It controls a clock with a phase shift of 3ns prior to the main clock. In this way, the sdram will start processing before the clock edge, and the date would be faster processed after the main clock edge.

sysid_qsys_0:

This block represents the system id checker. The purpose of this block is to check the configuration of the hardware part and the software part to make sure the two matches.

key_code:

This block represents the parallel I/O port for switch input for FPGA. The keycode signal is connected at the top level. Software could use the address of the port of this block to read the keycode from the keyboard.

otg_hpi_address:

This block represents the parallel I/O port for switch input for FPGA. This port serves as the interface on NIOS-II the EZ-OTG address. It is connected to the hpi_io_intf module at the top level.

otg_hpi_data:

This block represents the parallel I/O port for switch input for FPGA. This port serves as the interface on NIOS-II the EZ-OTG data. It is connected to the hpi_io_intf module at the top level.

otg_hpi_r:

This block represents the parallel I/O port for switch input for FPGA. This port serves as the interface on NIOS-II the EZ-OTG read. It is connected to the hpi_io_intf module at the top level.

otg_hpi_w:

This block represents the parallel I/O port for switch input for FPGA. This port serves as the interface on NIOS-II the EZ-OTG write. It is connected to the hpi_io_intf module at the top level.

otg_hpi_reset:

This block represents the parallel I/O port for switch input for FPGA. This port serves as the interface on NIOS-II the EZ-OTG reset. It is connected to the hpi_io_intf module at the top level.

otg_hpi_cs:

This block represents the parallel I/O port for switch input for FPGA. This port serves as the interface on NIOS-II the EZ-OTG chip-select. It is connected to the hpi_io_intf module at the top level.

Design Resources and Statistics

LUT	7596
DSP	0
Memory(BRAM)	2,285,568 / 3,981,312 (57 %)bits
Flip-Flop	2899
Frequency	118.79MHz
Static Power	105.52mW
Dynamic Power	0.79mW
I/O Thermal Power	71.71mW
Total Power	178.04mW

Game Mechanism

To play the game, the player has the following key buttons to use to successfully enjoy the dungeon exploration:

Operational Guidelines:

W:	To control the character to move upwards.
S:	To control the character to move downwards.
A:	To control the character to move to the left.
D:	To control the character to move to the right.
W,A:	To control the character to move up left.
W,D:	To control the character to move up right.
S,A:	To control the character to move down left.
S,D:	To control the character to move down right.
↑:	To fire a bullet up.
←:	To fire a bullet to the left.
→:	To fire a bullet to the right.
↓:	To fire a bullet down.

How to Start/Restart the game:
At the main menu, it has an instruction saying ‘Press ENTER to Start’. In this case, simply press ENTER and the game would move to the first dungeon you should conquer.
When you finish one attempt, either winning the game or losing the game, with the monitor displaying either ‘YOU WIN’ or ‘GAME OVER’, you need to press ‘Esc’ to go back to the main menu.

Goal:
There are in total three dungeons in the game, in each dungeon, there would be two monster minions inside.
In order to win the game, the character should explore all dungeons and kill all monster minions inside. In the second dungeon, a golden key would reveal. In the third dungeon, after finishing shooting down all monsters, the character should insert the golden key to the center of the altar to pass the game.
However, two monsters in each dungeon are not harmless. One of them can shoot bullets to all four directions continuously and the other chases the character and rests at a duty cycle of about 50%. Each bullet hitting the character would reduce the health of the character by one heart, while the character has seven hearts by default. If the character fails to dodge the bullets from monsters and as soon as the character's health goes down to zero, the character would die and would therefore lose the game. But for the second monster that chases you, as long as the character gets caught, the character would lose the game immediately.

List and Descriptions of Features incorporated

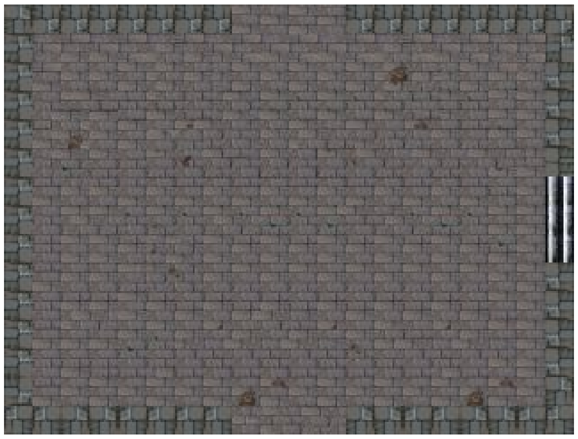
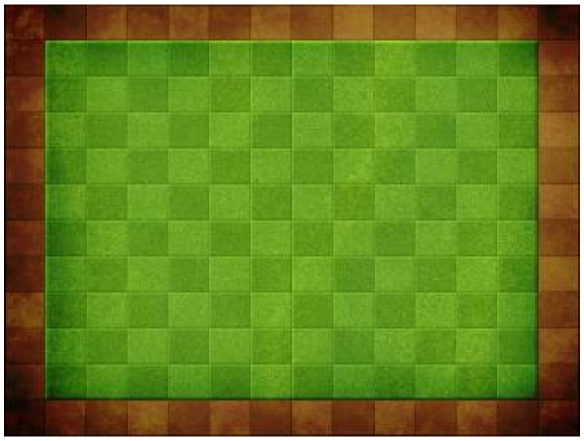
- Basic Features:
    - Character’s movement in accordance with keyboard input and in four directions. (Up, Down, Left, Right)
    - Character has initial Health which will be reduced by a certain amount if hit by monsters. When Health goes to zero, character would die and game over.

        - There is a health indicator on the top left of the monitor, to implement this, we use a counter to count the time that the character has been hit by the monsters.
    - Character’s weapon can fire a bullet once a time towards the direction indicated by the keyboard.(Up, Down, Left, Right)
        - We use a state machine to wait for the signals to fire a bullet, once received corresponding signals, it goes to the launching state and assign the bullet with indicated directions from the keyboard input, after the bullet hits something or flies beyond the boundary of the screen, the state machine goes to wait state to wait for a new input signal.
    - Bullets hitting monsters will disappear and reduce Health of the monster, when Health is reduced to zero, the monster will disappear.
- Advanced Features:
    - When holding two certain direction keys, the character should be able to move diagonally.
        - To implement this, we modify the USB keyboard driver to enable it to send out two key presses through the data bus.
    - Various sound effects for firing, hitting monsters, losing health, etc.
    - When finishing a level, the character should go to the exit of the current level, then the screen will flash back with character at the entrance of the new level.
        - The level transition is controlled by the master state controller.
    - The Health of the character is recorded and displayed.
        - Health system is placed outside the levels to maintain the same value while the character goes through different levels, then the current health value is displayed on the top left of the screen as the number of hearts.
    - Every monster would have a unique and random pattern of movement, and attack methods.
        - The first monster moves slowly towards the character while continuously firing bullets in all four directions around; the second monster moves slightly faster towards the character, but

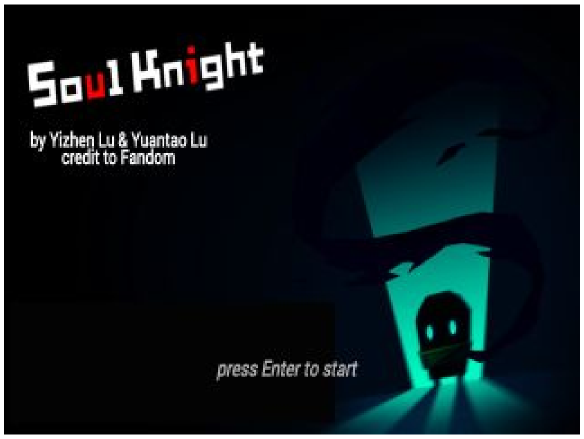
        would rest for a while after a period of chase, hitting this monster would be fatal.

- There are certain objects hidden in the level that the character can pick up to pass the game.
    - A golden key is hidden in the game, only uncovering itself after completing certain tasks. Simply moving the character to where the key is would be enough to pick up the key.
- The character is able to reenter the levels after they killed the monsters and move around without having to go through killing the monsters in those levels again.

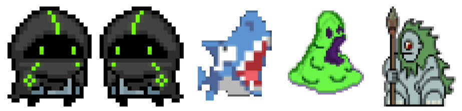
Sprites Used

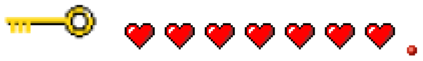



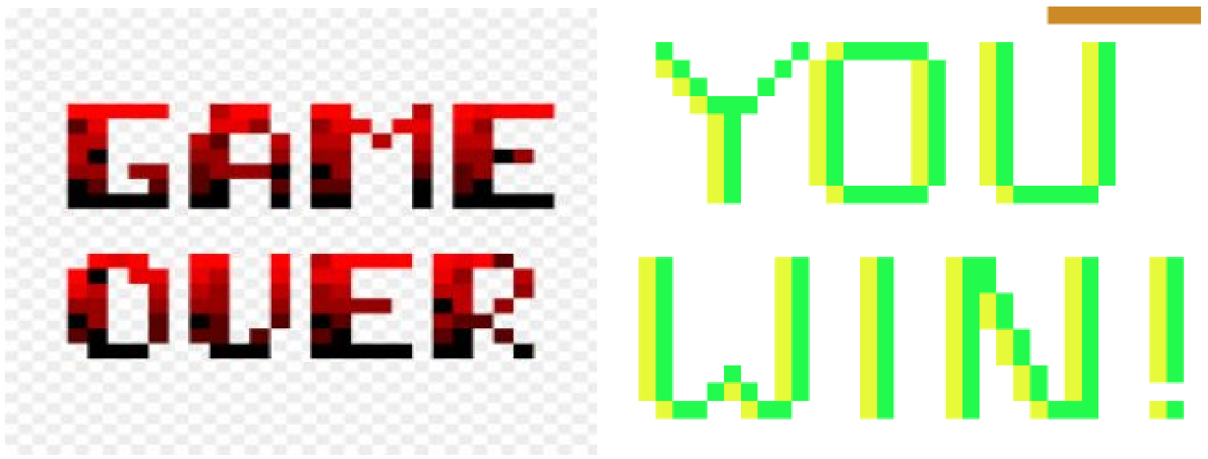
Sprites used for the background



Sprites used for the initial frame and the background


Sprites used for the character and the monsters


Sprites used for the stage props



Sprites used for the final frames

Hardship Experienced during the Project

1. The first difficulty we experienced was the way of loading the pictures effectively. Different from lab8, we will be playing with pictures in this lab. With limited amounts of on-chip memory, we need to find better data types for storing images. The approach we tried was the palette. We transformed the input png files into text files with at most 16 numbers and used a palette to map each of those numbers into 24 bit RGB values. We saved a large amount of memory this way so the 3.8 MB on-chip memory would be sufficient to operate all the pictures.
2. We did not use a frame buffer for this project, so it is very difficult to draw many components on the screen. For each pixel, we need to find a way to decide what object we need to draw on the screen. Is this pixel a monster, a character, a heart, a bullet or a background. What pixel from which picture is the one we want to draw on the screen. There are too many signals we need

to control and it is very easy to mess up with those signals. We have experienced many bugs coming from those many signals.

3. The state machine for the project is buggy at first. It is very hard to debug the state machine because some of those state machines depend on the frame_clk, which is the output from the VGA. Namely, we can't use the simulation to debug our code and it is very time consuming. We can only examine the code carefully and run the buggy code to find out the bugs.
4. The special condition this year increased the difficulty of this project. We stayed on campus and do not have the hardware (keyboard & screen monitor) we need to do the project. We bought the hardware and shared with one other group to have the hardware we need. We have to go to the one with the monitor each time to run the project. It is also very difficult to do group work. We live in different apartments so when we have to accomplish some important tasks, we need to come together and one of us needs to move a very long distance.
5. Without the computers from ECEB, we have to run the project on our personal computers. This project using Quartus has very high requirements for our PCs. It is easy to compile the project for ten more minutes and increase the CPU usage significantly. One of us had a Macbook and he had to run the Quartus on the Windows system on the Macbook, which is risky and difficult. The PC gets hot easily and shuts down unexpectedly for self protection. The database for the PC was damaged while running the buggy project and compiling at the same time. It is fortunate that this PC recovered at last.
6. We initially planned to equip audio with the project, but after a long time of efforts, we still failed to make it working with the provided audio driver. We think the issue is due to the improper interaction with the memory. Because when we first tested the driver and audio controller with data stored at on-chip memory, it managed to produce some high frequency beeping sound, but after we switched it to SRAM and flash, we got no sound output at all.

Conclusion

From the project, we have a deeper understanding of how to interact with the VGA monitor and the USB keyboard. Most knowledge learned in the class during the semester was applied in the project. Lab 8 is the most useful and helpful one, this project refers a lot to our own product from lab 8. State machines are also imperative components of the project, but with the exercise from lab9, using SystemVerilog to implement the decryption of the AES algorithm, our skills to handle state machines are nicely honed.

This is a meaningful project for us because we use our own knowledge, either taught during lectures or explored while working on the project, to complete one of our favourite video games.