ECE 385

Spring 2019

Final Project

Jinghan Huang, Chao Xu

Thursday 13:00-15:50

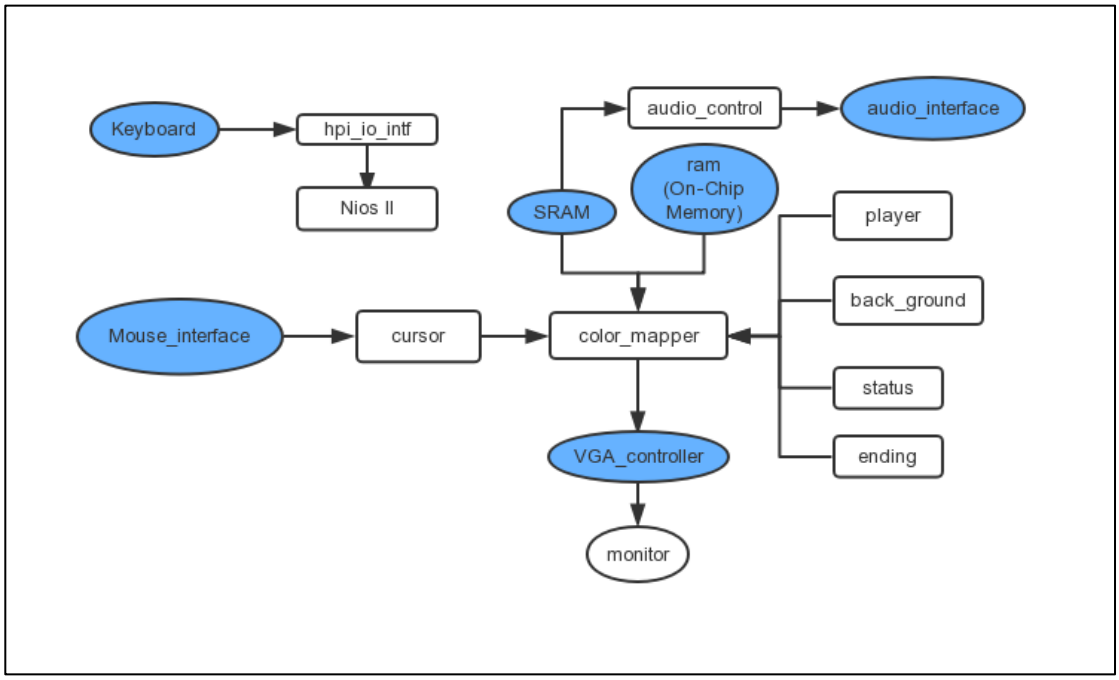
Chushan Li, Rui Lu

1 Introduction

In this project, we successfully implements a game, bomberman. We interface a keyboard and a monitor with the DE2 board using the on-board USB and VGA ports. Based on lab 8, we change the design of handling keycodes both in hardware part and software part to support multi-player mode. Two players can put bomb to hurt each other. Also, audio and mouse are added to this game. By putting bombs to explode beside the opponent, the player will win the game.

2 Written Description

2.1 Written Description of the Overview of the Circuit



Overview of the Circuit

- The purpose of the circuit

Our design include a NIOS II CPU for the purposes of interfacing with the USB keyboard as in lab 8. Because our FPGA board is equipped with the Cypress EZ-OTG (CY7C67200) USB Controller, we can use it to control the activities of the player. We connect the VGA monitor and mouse with our FPGA board. By using VGA_controller and cursor, we can print image on the screen and control the cursor of the mouse. SRAM will be used to store image data of the title (big image) and music data. Ram (On-Chip Memory) will be used to store image data of small sprites (small image). By using audio control, we can play music during the game. Most of our features will be implemented in hardware (SystemVerilog). Game logic is designed by using hardware. The color mapper will tell VGA controller the color data of each pixel.

- The feature of the circuit

a) The game can be played with background music. Some sound effects, like putting bombs and the explosion of bombs are added to this game.
b) We add mouse to our game, which can be used to click the start button to make the game begin.
c) By using our optimized keyboard, two players can move fluently with animation and put bombs. There must be some cool down time (CD) between two putting bombs. Players cannot put bombs constantly without waiting.
d) The map is designed based on bitmap. Different value in the bitmap will be used to represent different item on the screen.
e) The life of players and time are updated on the screen.
f) Bombs will explode after several seconds. The range of the explosion is usually one grid around the center of the bomb.If destructible trees are in the range of the explosion, they will become grass. If a player is in the range of the explosion, the health points (HP) of the player will be reduced.
g) When trees are destroyed, special buff (speed up) may appear. We will handle the change of the states.

What's more, Each component's purpose and features will be introduced in details from section 2.3 to section 2.10.

2.2 Description of the General Flow of the Circuit

- Inputs/Outputs of Top Level

```
// Basic I/O
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,
// 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 virtical 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
    // mouse interface
    inout                PS2_CLK, PS2_DAT,
    // audio Interface
    input                AUD_BCLK, AUD_ADCDAT, AUD_DACLRCK,
AUD_ADCLRCK,
    output logic         AUD_XCK, AUD_DACDAT, I2C_SDAT, I2C_SCLK,
    // SRAM Interface
    inout wire [15:0] SRAM_DQ,
    output logic SRAM_UB_N, SRAM_LB_N, SRAM_CE_N,   SRAM_OE_N,
SRAM_WE_N,
    output logic [19:0] SRAM_ADDR
```

- Input Processing and Output Generation

VGA Interface's toplevel I/O:

To interact with the VGA components, the processor connects to VGA controller, player module, and color_mapper. They are explained in the module description part. With proper Horizontal Sync and Vertical Sync signals produced by the VGA controller, it scans through the whole monitor and draws pixels in RGB colors (background or player or etc) mapped by color_mapper. They work together to draw the player, background, etc. and update the monitor according to pressed direction KEY.

CY7C67200 USB chip Interface's toplevel I/O:

To interact with the CY7C67200 USB chip, we first write a USB protocol to interface keyboard, utilizing memory-mapped PIO (Parallel Input Outputs) jtag_uart_0

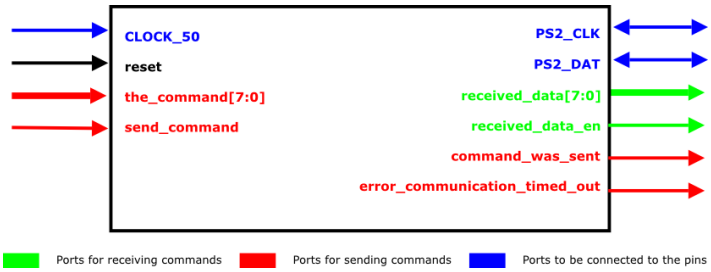
as an OTG interrupt to the processor. To control the data flow in the memory bus, we add multiple PIO connections (keycode, otg_hpi_address, otg_hpi_data, otg_hpi_r, otg_hpi_w, otg_hpi_cs, otg_hpi_reset) to the lab 8 setup. We fill in IO_read and IO_write, with the appropriate code to correctly read and write a single register on the CYC67200 chip, and write UsbRead and UsbWrite to read and write the CY7C67200's RAM by register reads and writes. After setting up the memory bus, the NIOS II processor was able to connect to Cypress EZ-OTG (CY7C67200) USB Controller. The hpi_io_intf.sv takes the control of data transmission via USB port and manages structured information to and from the DE2.

Besides VGA/CY7C67200/SDRAM interface that are also used in previous labs, here are two extra functionality groups of inputs and outputs typically used in our final project: mouse and audio.

Mouse Interface's toplevel I/O:

inout PS2_CLK, PS2_DAT

Description [1]: PS2_CLK, PS2_DAT are PS/2 clock and data lines, respectively. These bidirectional lines are connected to the correspondingly named pins on the DE2 board.



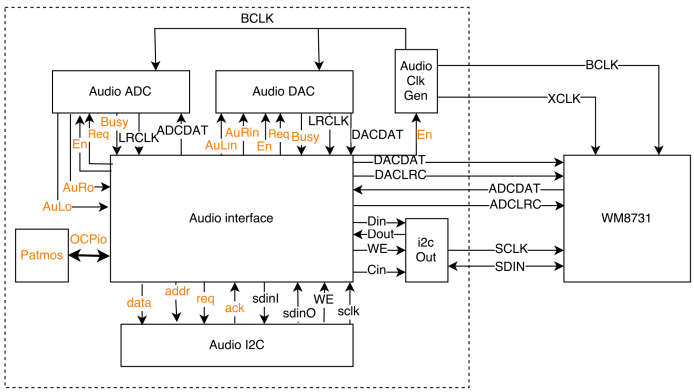
Overview of PS/2 Controller Interface Design [1]

Audio Interface's toplevel I/O:

input AUD_BCLK, AUD_ADCDAT, AUD_DACLRCK, AUD_ADCLRCK

output logic AUD_XCK, AUD_DACDAT, I2C_SDAT, I2C_SCLK

Description [2]: The audio interface will specify the desired output pins, which will be 1-bit outputs for the digital to analog data (AUD_DACDAT) and the sample rate clocks (AUD_DACLRCK, AUD_ADCLRCK) and a 1-bit input for the analog to digital data (AUD_ADCDAT). The audio chip will be used in slave mode and will be clocked by the FPGA hence there are also pins needed for the master clock (AUD_XCK) and the bit clock (AUD_BCLK). Besides, I2C_SDAT is the serial interface data line while I2C_SCLK is the serial interface clock.



Overview of Audio Interface Design [2]

- Outcome of Correct Outputs

Mouse Interface's Outcome: With a proper setup of PS2_CLK, PS2_DAT and the hand-drawn cursor pattern, the cursor will show up on the screen. Additionally, the cursor will move smoothly in the same direction as how we move the mouse by hand. Besides the basic movement. When we move the cursor to the start button, the button changes color to encourage a click. With a left click, the game will start accompanied by happy background music.

Audio Interface's Outcome: Three pieces of music will be played fluently. The background music will be played in loops from the beginning of the game to the end of the game. The put-bomb sound and the bomb explosion sound will be only played once, which is triggered together with the start of the corresponding animations. Details will be introduced in Section 2.3 and Section 3.

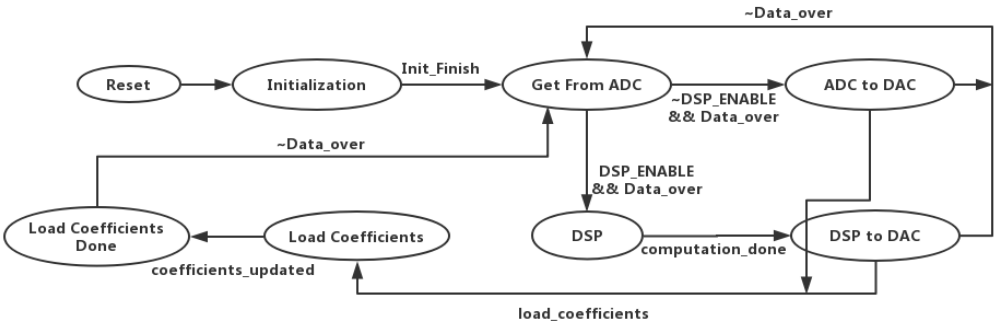
Other Interface's Outcome: VGA display should work normally; that is, the game is centered on the screen, and all the colors show as expected. As for the keyboard, multiple keycodes could be correctly interpreted.yingg

2.3 Written Description of Audio Control

- Audio Interface

To incorporate audio into our final project, we fully employs the VHDL audio driver kindly provided by Koushik Roy as suggested. Based on this audio driver as the foundation of the audio part, we follow its documentation and design an audio control state machine to interact with the driver, which is part of our entire system machine. Our audio control state machine design will be introduced in Section 3. Thus, we will only discuss audio chip interface and its state machine here.

Since our project just needs to play music and does not use the microphone, only DAC mode of the audio chip is used, i.e. audio conversion from digital signal to analog signal. While most of audio chip signals are set up as toplevel inputs and outputs, there are five inputs/outputs that we pay close attention to: Inputs: `LDATA, RDATA, INIT`, and Outputs: `INIT_FINISH, data_over`. As shown in the audio driver state machine, `INIT` and `INIT_FINISH` are used for the initialization of the driver. The audio driver will start initialization if `INIT` signal is raised high and will raise `INIT_FINISH` when it has finished the initialization process. As for the sound output, the DAC module will raise `data_over` if a single data sample (`LDATA + RDATA`) has been fed correctly. During the data reading process, `data_over` is set to low.



Audio Driver State Machine

- Audio processing and SRAM support

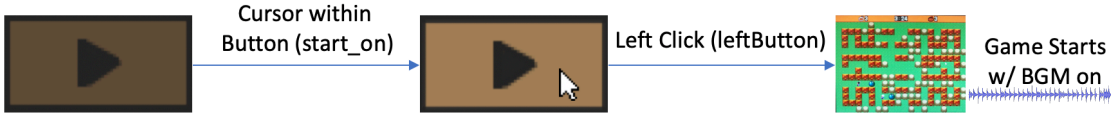
In addition to the state machine design, we also work on audio processing and SRAM to support smooth audio output in our game. SRAM is chosen for the music storage since it has larger storage space than the on-chip memory and considerable accessing speed for the audio output so that we could choose a longer background music for playing in loops.

To make the audio file support the SRAM format, our audio processing work is broken down to parts: resampling, mixing track, and format conversion (.mp3->.wav->.txt->.ram). According to the audio interface for the Patmos processor [2], the WM8731 audio codec expects a sampling frequency of 48kHz. We re-sample the chosen mp3 audio file at 48kHz. For the audio use in our game, the left channel usually has the same sound as the right channel; thus, we combine the stereo tracks into one mono-track to save the storage. Then, the edited audio file is exported in the .wav format. To handle .wav audio file, we use some MATLAB codes as well as Python codes to convert it into .txt format. In this txt file, each line contains a 16-bit music data sample in hexadecimal expression. Finally, we modify Rishi's ECE385 helper tool [3] to convert the audio .txt file to the .ram format. The original one is used to handle image pixels while we use it to handle audio samples; thus, the actual required format varies in some ways. Finally, we import .ram audio files into our FPGA board and use state machine signals to decide which music in the SRAM to be fed into audio chip.

2.4 Written Description of Mouse

PS/2 mouse plays a role to link up the game's start page with the main interface. Instead of a specific key to press, the graphical start button with mouse support provides users a more friendly and natural way to play this game. Specifically, when we move the cursor to the start button, the button changes color to encourage a click. With a left click, the game will start accompanied by happy background music.

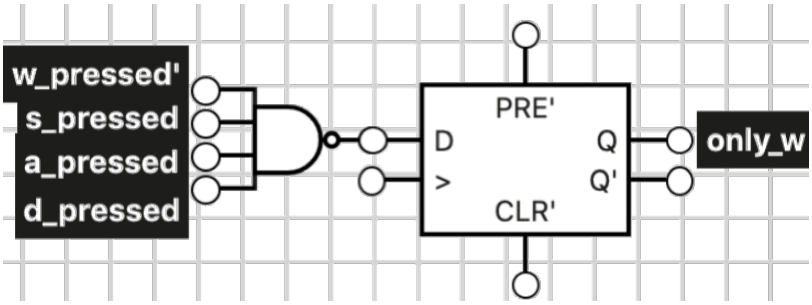
Notice the mouse cursor is fully HAND-DRAWN by us pixel by pixel to minimize the memory consumption. Besides, our mouse support is developed based on the mouse interface [4].

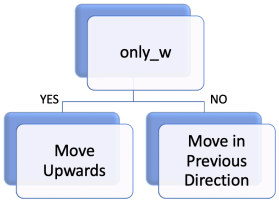


2.5 Written Description of Multi-Player/Multi-Keycode

The introduction of multiplayer mode brings the Bomberman game more fun. It enables two players to smoothly interact in the game without any conflicts or interference. Besides the collision detection which will be discussed later, the essential support of multi-player mode is handling multi-keycode correctly.

While some keys are naturally isolated, e.g. a direction key and a put-bomb key, there is interference between direction keys. That is, we do not expect race conditions between two direction keys when they are both pressed, which is quite common in the real practice. Actually, players tend to give the priority of moving direction to the earlier pressed button. To handle such issues, we take advantage of the non-combinational circuits to judge the case in two steps. As shown in a sample logic circuit below, which is interpreted from our codes. In the first step, we will translate the keycodes from software and determine if only one direction key is pressed for each player. In the second step, it will decide the moving direction from the multi-key situation following the flowchart below.

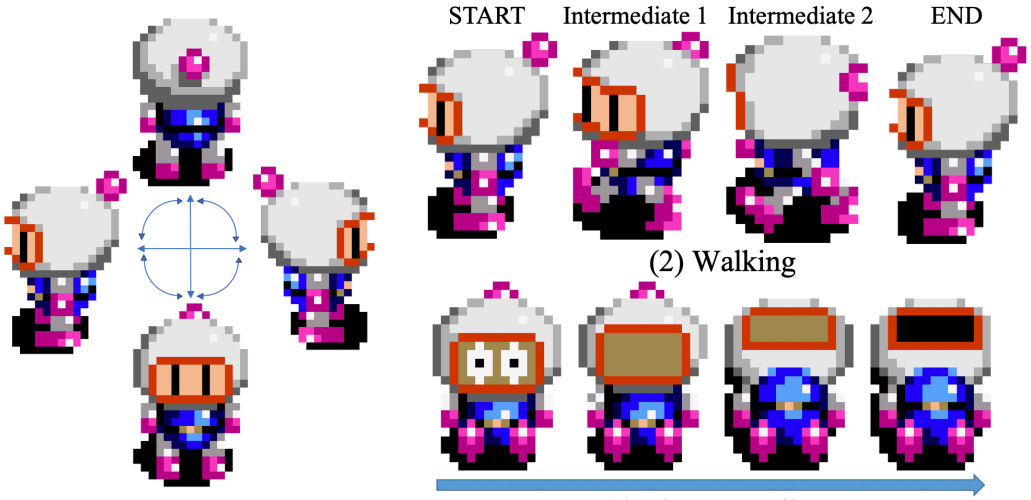




2.6 Written Description of Animation

The animation part in our game is used to display smoother state switch (mainly for player movement and dynamically-rendered bomb). Specifically, we embed SIX kinds of animations for (1) player orientation, (2) walking, (3) static bomb, (4) bomb explosion, (5) player's failure/death, and (6) result pop-up, respectively.

(1) There are four orientations as shown in the figure below that corresponds to WSAD/ ↑↓←→. That is, regardless of whether there is obstacle ahead, the player orientation will always switch to the current pressed key's correponding orientation unless the player is walking. It is always a 2-frame animation. (2) For walking process in each walking orientation, there is a 4-frame 1/3-second animation, which is supposed to be displayed when the key press time is long enough to change the player's position.



(1) Player Orientation

(5) Player's Failure

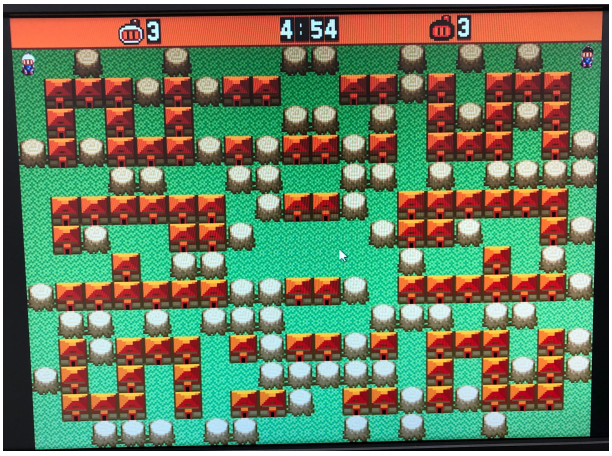
(3) Our bomb looks like a real bomb since there is 5-frame 2.5-second animation when it is static. (4) The bomb explosion contains a 5-frame animation which could penetrate and get rid of "trees" accompanied by explosion sound. (5) The player who lose the game will be die in a 4-frame animation at the end of the game. (6) At the end, there will be a pop-up window flying in showing the winner or tie.

(6) Result Pop-up Window (Fly-in)

2.7 Written Description of Map Design and Collision Detection

We use bitmap to design our map, which is a very convenient method. The original is a 20 x 14 map (width * height). Because the screen is 640 x 480 and the upper 640 x 32 is the status bar, we only use 640 x 448 for the map. Each unit is 32 x 32, so we will use 20 x 14 bitmap to design our map. Different values will be used to represent different items on the map. For example, 3'h0 - grass (open space), 3'h1 - tree (destructible stump), 3'h2 - room (nondestructible house), 3'h3 – shoe, 3'h4 - stump with shoe. By using this method, we can change the map easily.

```
parameter[0:20*14-1][2:0] original_map = {
    3'h0, 3'h0, 3'h1, 3'h0, 3'h0, 3'h4, 3'h0, 3'h0, 3'h0, 3'h1, 3'h1, 3'h0, 3'h0, 3'h1, 3'h0, 3'h4, 3'h0, 3'h1, 3'h0, 3'h0,
    3'h0, 3'h2, 3'h2, 3'h2, 3'h1, 3'h2, 3'h1, 3'h2, 3'h2, 3'h0, 3'h0, 3'h2, 3'h2, 3'h1, 3'h2, 3'h0, 3'h2, 3'h2, 3'h2, 3'h0,
    3'h0, 3'h2, 3'h0, 3'h2, 3'h0, 3'h2, 3'h0, 3'h0, 3'h0, 3'h1, 3'h1, 3'h0, 3'h1, 3'h0, 3'h2, 3'h1, 3'h2, 3'h1, 3'h2, 3'h0,
    3'h1, 3'h2, 3'h1, 3'h2, 3'h2, 3'h2, 3'h1, 3'h2, 3'h1, 3'h2, 3'h2, 3'h1, 3'h2, 3'h0, 3'h2, 3'h2, 3'h2, 3'h0, 3'h2, 3'h1,
    3'h0, 3'h0, 3'h0, 3'h4, 3'h1, 3'h0, 3'h0, 3'h1, 3'h1, 3'h0, 3'h0, 3'h4, 3'h1, 3'h0, 3'h1, 3'h0, 3'h4, 3'h1, 3'h1, 3'h1,
    3'h0, 3'h2, 3'h2, 3'h2, 3'h2, 3'h2, 3'h2, 3'h0, 3'h1, 3'h2, 3'h2, 3'h1, 3'h0, 3'h2, 3'h2, 3'h2, 3'h2, 3'h2, 3'h2, 3'h0,
    3'h0, 3'h2, 3'h1, 3'h0, 3'h0, 3'h2, 3'h2, 3'h4, 3'h0, 3'h0, 3'h0, 3'h0, 3'h1, 3'h2, 3'h2, 3'h1, 3'h0, 3'h0, 3'h2, 3'h1,
    3'h0, 3'h0, 3'h0, 3'h2, 3'h0, 3'h1, 3'h1, 3'h0, 3'h0, 3'h0, 3'h0, 3'h0, 3'h0, 3'h1, 3'h0, 3'h0, 3'h2, 3'h0, 3'h1, 3'h0,
    3'h1, 3'h2, 3'h2, 3'h2, 3'h2, 3'h2, 3'h2, 3'h1, 3'h1, 3'h2, 3'h2, 3'h1, 3'h0, 3'h2, 3'h2, 3'h2, 3'h2, 3'h2, 3'h2, 3'h4,
    3'h0, 3'h4, 3'h1, 3'h0, 3'h1, 3'h0, 3'h1, 3'h1, 3'h1, 3'h0, 3'h0, 3'h0, 3'h1, 3'h4, 3'h1, 3'h0, 3'h1, 3'h1, 3'h0, 3'h0,
    3'h0, 3'h2, 3'h1, 3'h2, 3'h2, 3'h2, 3'h0, 3'h2, 3'h1, 3'h2, 3'h2, 3'h1, 3'h2, 3'h0, 3'h2, 3'h2, 3'h2, 3'h0, 3'h2, 3'h1,
    3'h1, 3'h2, 3'h0, 3'h2, 3'h0, 3'h2, 3'h0, 3'h0, 3'h1, 3'h1, 3'h1, 3'h4, 3'h1, 3'h0, 3'h2, 3'h0, 3'h2, 3'h1, 3'h2, 3'h1,
    3'h0, 3'h2, 3'h2, 3'h2, 3'h0, 3'h2, 3'h0, 3'h2, 3'h2, 3'h1, 3'h0, 3'h2, 3'h2, 3'h1, 3'h2, 3'h1, 3'h2, 3'h2, 3'h2, 3'h0,
    3'h0, 3'h0, 3'h4, 3'h1, 3'h1, 3'h0, 3'h4, 3'h1, 3'h0, 3'h0, 3'h0, 3'h1, 3'h0, 3'h1, 3'h0, 3'h0, 3'h1, 3'h0, 3'h0, 3'h0,
};
```

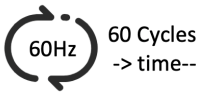


By using bitmap method, it is also easy for us to handle the collision detection. Because the player can only go through 3'h0 - grass (open space) and 3'h3 – shoe, if the player aims to move to the place, which is not 3'h0 or 3'h3, the corresponding signal (can_pass) will be set to low to forbid the player to move in this direction.

2.8 Written Description of Status Bar

The status bar is introduced to show the gaming status as well as provide a visual criterion for the game's winner judgment. The status bar is comprised of a timer as well as the display of two players' health points. The player whose health point is decreased to zero at any time or lower than the other player's health points at the end of game (time = 0:00) will lose the game.

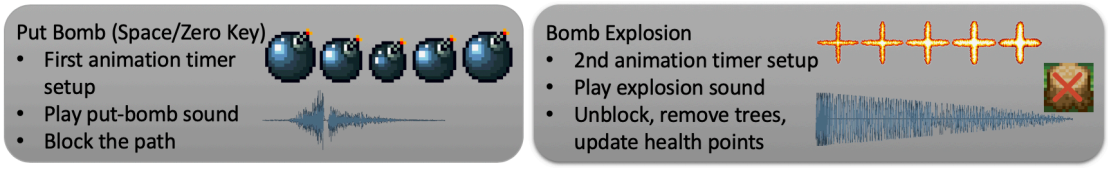
The timer takes advantage of the 60Hz frame clock to achieve time count down. A time counter is decreased by each cycle, and sixty cycles will suggest a passed second. Then, the time display will be updated accordingly.



2.9 Written Description of Bomb

Bomb is the most important and complicated item/prop in our game. It could hurt player's life and remove trees in the way. In addition, it is associated with two pieces of animation and two pieces of sound. In the whole state machine which will be introduced later, it provides several important signals for state transitions.

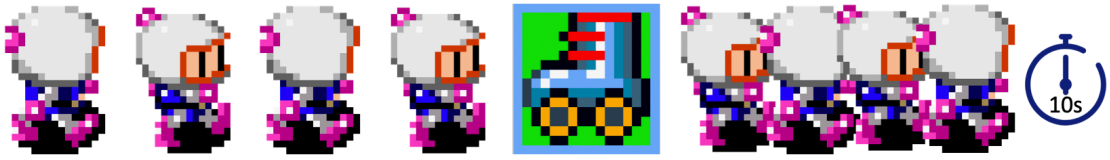
Here are two major sequential steps as shown in the following block diagram. When one player presses the put-bomb key (space/zero), the bomb counter will start countdown, and the first animation and the put-bomb sound will be immediately triggered. In addition, the bitmap and the life-point board introduced above will be updated. As the bomb counter reaches zero, the second block—bomb explosion—starts to work: The second animation and the explosion sound will be played. The bitmap and the status bar will be updated afterwards.



2.10 Written Description of Special Items

The special item is employed to increase the intensity of game and create a more competitive game atmosphere. In our game, there is an interesting special item: shoe. It is originally hidden inside some "trees." When the shield "tree" is removed in explosion, it will show up as a surprise. It will increase the player moving speed to 2x after the player grab it, which will only last 10 seconds in our game.

Technically speaking, the tree with the special item is marked as a new type of cell in the bitmap, which will show up as a shoe only after one explosion. When the player is within a revealed shoe cell. The shoe pattern will be replaced by normal grass, and the counter for this special item will start countdown. During this period, the walking animation play time will be cut by half to achieve the speed up.



3 Module Descriptions

- lab8.sv

Module: lab8

Inputs: CLOCK_50, OTG_INT,
AUD_BCLK, AUD_ADCDAT, AUD_DACLRCK, AUD_ADCLRCK,
[3:0] KEY

Outputs: DRAM_CLK, DRAM_CAS_N, DRAM_CKE,
DRAM_CS_N, DRAM_RAS_N, DRAM_WE_N,
VGA_CLK, VGA_SYNC_N, VGA_BLANK_N,VGA_VS, VGA_HS,
OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N,
SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N,
SRAM_WE_N,
[1:0] DRAM_BA, OTG_ADDR,
[3:0] DRAM_DQM,
[6:0] HEX0, HEX1,
[7:0] VGA_R, VGA_G, VGA_B
[12:0] DRAM_ADDR,
[19:0] SRAM_ADDR

Inout port/wire: PS2_CLK, PS2_DAT,
[31:0] DRAM_DQ, OTG_DATA, SRAM_DQ

Description: This is the toplevel module, which is modified from lab8. It connects hpi_io_intf module, nios_system module, Mouse_interface module, audio_interface

module, audio_control module, vga_clk module, VGA_controller module, player module, color_mapper module, status module, back_ground module, cursor module and HexDriver module together.
Purpose: This module is used to connect the SoC (system on chip) module with external inputs and outputs. In other words, it integrates the Nios II system with the rest of the hardware.

- keycode_handler.sv

Module: keycode_handler
Inputs: [31:0] keycode
Outputs: w_pressed, s_pressed, a_pressed, d_pressed,
up_pressed, down_pressed, left_pressed, right_pressed,
enter_pressed, space_pressed, zero_pressed, r_pressed, backspace_pressed
Description: The module will use keycode as input, which can read at most four keycodes at the same time. Each keycode is 8 bits wide. To know whether the corresponding key is pressed, we need check these four keycodes one by one.
Purpose: This is a module which can know whether the corresponding key is pressed.

- player.sv

Module: player
Inputs: Clk, Reset, frame_clk,
[7:0] keycode,
[9:0] DrawX, DrawY,
player_number, w_pressed, s_pressed, a_pressed, d_pressed,
white_win, black_win, tie, is_ending_exist, r_pressed,
[3:0] can_pass,
speed_up
Outputs: is_ball,
[5:0] player_color_Idx,
[9:0] Ball_X_Pos, Ball_Y_Pos
Description: Player 1 and Player 2 will use the same module, and we can identify them by using 1-bit data **(**player_number). This module accepts the current pixel coordinates (DrawX and DrawY) and return a signal (is_ball, which means player actually) to tell whether the current pixel belongs to the player. Additionally, this module can accept keycode (w_pressed, s_pressed, a_pressed, d_pressed) to control the position of the player in the screen and update the frame with frame_clk periodically. It recognizes the commands from the keyboard and changes the direction of the player to up, down, left, or right (4-bit signals can_pass will determine whether the player can go through in this direction). However, the order of these four keycodes is of smallest to largest. The priority of keycode is also handled in this module. The animation of moving is also added for this module. When the player receive the command from the keyboard, the player will enter the walking state. When the player is in the walking state, the player will reject the command from the keyboard. The signal (speed_up) will determine whether the player is in the speed-up state. These four signals (white_win, black_win,

tie, is_ending_exist) are used to determine ending animation of players. If the current pixel is belong to the player, the signal (is_ball) will be set to high, and the palette index (player_color_Idx) will be used by color_mapper.
Purpose: This is a module used to computes the position of the player in every frame. Additionally, it could compute whether the current pixel corresponds to the player. All the things related to the player, like position, direction, motion and animation, will be handled in this module.

- back_ground.sv

Module: back_ground
Inputs: Clk, Reset, frame_clk,
[9:0] DrawX, DrawY,
[9:0] Ball_X_Pos, Ball_Y_Pos, Ball_X_Pos_2, Ball_Y_Pos_2,
space_pressed, zero_pressed, r_pressed,
is_ending_exist,
Outputs: [9:0] Bomb_X_Pos, Bomb_Y_Pos, Bomb_X_Pos_2, Bomb_Y_Pos_2,
[4:0] bomb_color_Idx,
[3:0] tree_color_Idx, room_color_Idx, shoe_color_Idx,
[2:0] explosion_color_Idx,
[1:0] grass_color_Idx,
[7:0] can_pass,
speed_up_1, speed_up_2,
is_bomb_place, is_bomb_place_2,
is_room, is_tree, is_bomb, is_shoe, is_explosion, is_grass,
explosion_signal, explosion_signal_2, explosion_start_signal, explosion_start_signal_2,
Description: In this module, 20x14 bitmap will be used to construct the design of the background. Different values in the bitmap will be used to determine the item of 32x32 area. For example, 3'h0 (grass – open space), 3'h1 (tree – destructible stump), 3'h2 (room – nondestructible house), 3'h3 (shoe), 3'h4(stump with shoe). Also, the 8-bit result of handling collision (can_pass) wil be output for two players. In addition, space and zero key will be received to put bomb. Picking up special items will also be handled in this module. If the player moves to the position If the current pixel is belong to the background, the corresponding signal (is_room, is_tree, is_bomb, is_shoe, is_explosion or is_grass) will be set to high, and the corresponding palette index will be used by color_mapper.
Purpose: The module is used to draw sprites of the background and handle putting bomb and the explosion of the bomb. Picking up special items and collision detection will also be handled in this module.

- grass.sv

Module: grass
Inputs: Clk,
[9:0] DrawX, DrawY,

Outputs: is_grass,
[1:0] grass_color_Idx
Description: If DrawY is greater than 32, is_grass will be set high, which means grass will be drawn as default.
Purpose: The module is used to know the palette index for each pixel we need to draw grass.

- cursor.sv

Module: cursor
Inputs: Clk, Reset, frame_clk,
[9:0] DrawX, DrawY,
[9:0] cursorX, cursorY,
leftButton, middleButton, rightButton,
enter_pressed, r_pressed, is_ending_exist,
Outputs: is_cursor, is_startbutton, is_opening,
[1:0] cursor_color_Idx,
[3:0] startbutton_color_Idx
Description: Based on the position of the cursor (cursor and cursorY), if the current pixel is belong to the cursor, the corresponding signal (is_cursor) will be set to high, and the corresponding palette index (cursor_color_Idx) will be used by color_mapper. It also helps to draw the start button at the bottom right corner. If the cursor is in the range of the start button, the start button will change the color. If (leftButton) is high when the cursor is in the range of the start button, the game will start.
Purpose: The module is used to give the drawing information of the cursor and the start button. Also, it handles the function of cursor, which can click the start button to enter the game.

- audio_control.sv

Module: audio_control
Inputs: Clk, Reset,
INIT_FINISH, data_over, load_startpage,
is_opening, put_pulse, explode_pulse, is_ending_exist, r_pressed,
Outputs: INIT, music_select, state_status,
[18:0] music_addr
Description: INIT_FINISH suggests that the audio chip is ready to play music now. The DAC module will raise data_over to high if a single data sample (LDATA + RDATA) has been fed correctly. During the data reading process, data_over is set to low. is_opening suggests the user just clicked the start button and the game is supposed to start. Background music will start then. put_pulse is the signal to play put_bomb sound. In the same way, explode_pulse is the signal to play bomb explosion sound. is_ending_exist is the signal to stop the background music. r_pressed is the signal to restart the game. As for the module outputs, INIT is set to high so as to start the audio chip initialization. music_select is set to low if we want play put_bomb sound, and set to high if we want play bomb explosion sound. state_status is set to low if we want play

background music, and set to high if we want play non- background music (put_bomb sound or bomb explosion sound).
Purpose: This module is named as audio control though, it actually acts as the overall state machine for the game, which will be introduced in details later in the "Design Procedure" section. Specifically speaking, it controls the playing period (start and end) of three pieces of music here.

- ending.sv

Module: ending
Inputs: Clk, Reset, frame_clk,
[9:0] DrawX, DrawY,
r_pressed,
is_ending_exist, white_win, black_win, tie,
Outputs: is_ending,
[3:0] end_color_Idx,
Description: When is_ending_exist is high, the ending animation will start to play. According the game end states (white_win, black_win, tie), different ending images will be chosen. If the current pixel is belong to the ending animation, the corresponding signal (is_ending) will be set to high, and the corresponding palette index (end_color_Idx) will be used by color_mapper.
Purpose: It gives the drawing information of the ending animation. Based on the game end state, the chosen ending image will move from down to up in the center of the screen.

- status.sv

Module: status
Inputs: Clk, Reset, frame_clk,
[9:0] DrawX, DrawY,
[9:0] Ball_X_Pos, Ball_Y_Pos, Ball_X_Pos_2, Ball_Y_Pos_2,
[9:0] Bomb_X_Pos, Bomb_Y_Pos, Bomb_X_Pos_2, Bomb_Y_Pos_2，
explosion_signal, explosion_signal_2, is_opening, r_pressed
Outputs: is_status,
[2:0] status_color_Idx,
white_win, black_win, tie, is_ending_exist
Description: According to the position of the players and the bombs (Ball_X_Pos, Ball_Y_Pos, Bomb_X_Pos and Bomb_Y_Pos), the life of players will be updated at the moment the bomb is exploded (explosion_signal). Also, the timer will be counted by using frame_clk, which is 60Hz. If the current pixel is belong to the status bar, the corresponding signal (is_status) will be set to high, and the corresponding palette index (status_color_Idx) will be used by color_mapper.
Purpose: It gives the drawing information of the status bar, which include the life count of both players and the time. In addition, it is used to update the life count of the players, and count the time. When the time becomes zero or one of the players' life becomes zero, the module will output the corresponding game end state.

- bomb.sv

Module: bomb
Inputs: Clk, Reset, frame_clk,
is_bomb_place,
[9:0] DrawX, DrawY,
[9:0] Ball_X_Pos, Ball_Y_Pos, Ball_X_Pos_2, Ball_Y_Pos_2,
Outputs: [9:0] Bomb_X_Pos, Bomb_Y_Pos,
[7:0] can_pass,
[4:0] bomb_color_Idx,
[2:0] explosion_color_Idx,
is_bomb, is_explosion, is_bomb_exist,
explosion_signal, explosion_start_signal
Description: When is_bomb_place is set to high, the bomb counter will start to count and the position of the bomb will be the same as the position of the player. The 8-bit data (can_pass) will be used to handle the collision detection for two players. According to the bomb counter, the animation of the bomb and the explosion will be played. When the bomb counter becomes zero, the bomb will disappear. If the current pixel is belong to the status bar, the corresponding signal (is_bomb or is_explosion) will be set to high, and the corresponding palette index (bomb_color_Idx or explosion_color_Idx) will be used by color_mapper.
Purpose: It gives the drawing information of the bomb and explosion, and handle the collision detection of bomb.

- ram.sv

Module: playerRAM
Inputs: Clk,
[18:0] read_address
Outputs: [3:0] data_Out
Description: It load txt file first. By using read address, we can get the data in the txt file. The data is the palette index for the player
Purpose: The module is used to read the data from the On-Chip Memory based on the input address.
(There are many similar modules in the ram.sv file, so we only describe only one module)

- palette.sv

Module: player1_palette
Inputs: [5:0] player_color_Idx_1
Outputs: [7:0] Red, Green, Blue
Description: By using the palette index, the corresponding color data will be chosen.
Purpose: The module is used to obtain color data based on the palette index.
(There are many similar modules in the palette.sv file, so we only describe only one module)

- SRAM.sv

Module: SRAM
Inputs: Clk,
[9:0] DrawX, DrawY,
is_opening, state_status, music_select,
[18:0] music_addr,
Outputs: [15:0] Color_Idx, bgm_data,
SRAM_UB_N, SRAM_LB_N, SRAM_CE_N,
SRAM_OE_N, SRAM_WE_N,
[19:0] SRAM_ADDR
Inout port/wire: [15:0] SRAM_DQ
Description: Based on the game state (is_opening), we will choose to read image or music by changing the read address of the SRAM (SRAM_ADDR). Based on the game state (state_status and music_select) during the game, the different kinds of music will be chosen. We output music data throught the port (bgm_data) and output the image data by using the palette index (Color_Idx). Because this is the read-only SRAM, only SRAM_WE_N is set to high, and SRAM_UB_N, SRAM_LB_N, SRAM_CE_N and SRAM_OE_N are all set to low.
Purpose: The module is used to read the data from the SRAM based on the input signals.

- Mem2IO.sv

Module: Mem2IO
Inputs: Clk, Reset, CE, UB, LB, OE, WE,
[15:0] Data_from_CPU, Data_from_SRAM,
Outputs: [15:0] Data_to_CPU, Data_to_SRAM
Description: This module is a I/O and data transmission manager managing all I/O with the DE2 physical I/O devices, namely, the switches and 7-segment displays. It also transmit data between CPU and SRAM.
Purpose: Bidirectional data transmission. As CPU I/O is memory-mapped, MEM2IO is in charge of bidirectional data transfer between CPU and memory. It could load data from SRAM, or pass data from CPU to SRAM.

- tristate.sv

Module: tristate
Inputs: [15:0] Data_write,
Clk, tristate_output_enable
Outputs: [15:0] Data_read
Inout port/wire: [15:0] Data
Comment: 16 bits are the default parameter, other widths are also supported.
Description: This module is a tristate buffer. It enables a port to read from/write to the bus. It uses a output_enable signal to control reads and writes. If not writing to the bus, the output will be assigned with a high-impedance value.

Purpose: This module is used to control bidirectional signals between Mem2IO and SRAM.

- nios_system.v

Module: nios_system
Inputs: clk_clk, reset_reset_n,
[15:0] otg_hpi_data_in_port
Outputs: sdram_clk_clk, sdram_wire_cas_n, sdram_wire_cke,
sdram_wire_cs_n, sdram_wire_ras_n, sdram_wire_we_n,
otg_hpi_cs_export, otg_hpi_r_export, otg_hpi_w_export, otg_hpi_reset_export,
[1:0] sdram_wire_ba, otg_hpi_address_export,
[3:0] sdram_wire_dqm,
[7:0] keycode_export,
[12:0] sdram_wire_addr,
[15:0] otg_hpi_data_out_port
Inout port/wire: [31:0] sdram_wire_dq
Description: This is the SoC (System-On-Chip) module/NOIS II based system. All the inputs and the outputs of this module corresponds to those of the top level module (all sdram-signals in this module corresponds to the DRAM-signals in the top level module; all otg_export-signals in this module corresponds to the OTG-signals in the top level module). Additionally, it will produce an 32-bit signal called keycode_export to represents the key on the USB keyboard. This keycode_export is used to connect with player module to control player motion.
Purpose: This module is used to perform as a small SoC/ Nios II system.

- hpi_io_intf.sv

Module: hpi_io_intf
Inputs: Clk, Reset, from_sw_r, from_sw_w, from_sw_cs, from_sw_reset,
[1:0] from_sw_address,
[15:0] from_sw_data_out
Outputs: OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N,
[1:0] OTG_ADDR,
[15:0] from_sw_data_in
Inout port/wire: [15:0] OTG_DATA
Description: This module is an interface module. It uses OTG_DATA as a bidirectional wire to transfer the data. Each OTG-signal corresponds to a from_sw-signal. In this way, EZ-OTG acts as a USB Controller to control the connected USB keyboard. Besides, it implements a tri-state buffer inside.
Purpose: This is a module acting as the interface between NIOS II and EZ-OTG chip.

- vga_clk.sv

Module: vga_clk
Inputs: inclk0
Outputs: c0

Description: This module is a controller module inside the VGA interface. It accepts a normal input clock signal inclk0, and then produces another clock signal c0 used by VGA controller.
Purpose: This is a module used to generate the 25MHZ VGA_CLK using PLL.

- VGA_controller.sv

Module: VGA_controller
Inputs: Clk, Reset, VGA_CLK
Outputs: VGA_HS, VGA_VS, VGA_BLANK_N, VGA_SYNC_N,
[9:0] DrawX, DrawY
Description: This module is a controller module inside the VGA interface. It accepts the VGA_CLK to scan the whole screen. VGA_HS is the frequency to update each line and VGA_VS is the frequency to update the whole frame. This module will continuously traverse the screen and return the coordinates of the current traversaled pixel (DrawX and DrawY).
Purpose: This is a module used to produce timing signals Horizontal Sync (hs) and Vertical Sync (vs) required by any VGA monitor. It outputs the current traversed pixel coordinates and connects to the ball module and the color_mapper module.

- color_mapper.sv

Module: color_mapper
Inputs: is_ball,
[9:0] DrawX, DrawY
Outputs: [7:0] VGA_R, VGA_G, VGA_B
Description: This module is a simple color mapper inside the VGA interface. It simply computes RGB values given the current beam position. DrawX and DrawY indicates the current pixel coordinate and is_ball signal indicates whether the current pixel belongs to ball or background. VGA_R, VGA_G, VGA_B are the corresponding/mapped output color to VGA monitor.
Purpose: This is a module used to perform object or shape rendering and coloring - This entity puts out RGB signals to the monitor. Specifically, it draws (decides RGB values) pixels in ball color or background color according to the isball info from the ball module.

- HexDriver.sv

Module: HexDriver
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: This module transforms value of numbers to the corresponding hexadecimal number.
Purpose: This is a module used to get a correct hexadecimal number to be displayed on FPGA.

3 Design Procedure / State Diagram

3.1 Design Procedure

- Foundation Codes

Our final project is basically based on the lab 8, where we exploits the VGA interface as well as the PS/2 keyboard interface. Since we also use VGA display and keyboard input, this two interfaces from lab 8 are . Moreover, as we introduced audio and sound into our final projects, we refer to the VHDL Audio Driver on the Blackboard and the mouse interface [4] introduced in Section 2.4. In addition, Rishi's ECE385 helper tool [3] is also modified by us to convert image or audio .txt file to the .ram format file.

- Background Study

We conduct heavy research/background study in mouse interface and audio interface besides these codes that we directly use. We read PS/2 Controller Interface Design [1] and Audio Interface Design [2] for better understanding.

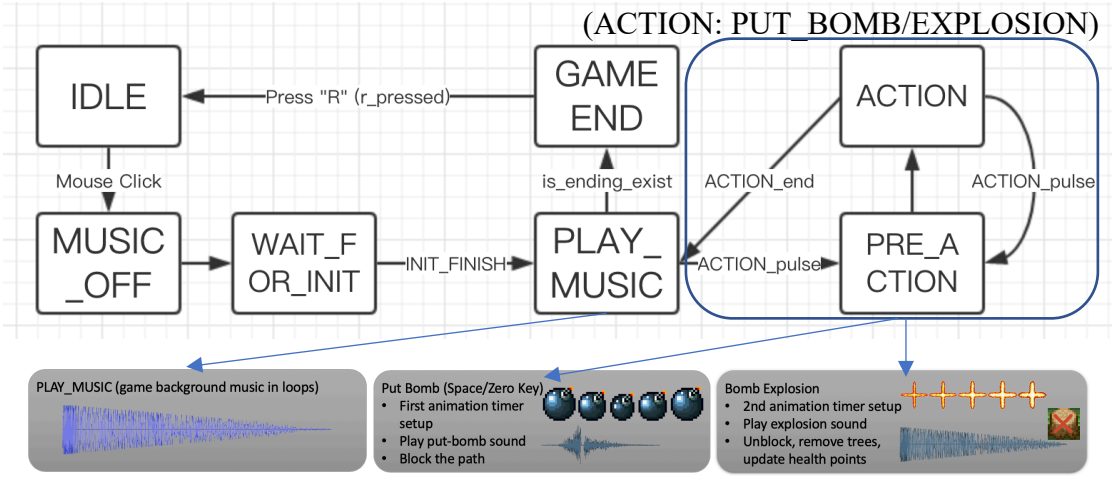
- Different Objectives of the Project

We choose SRAM to store image data of the title (big image) and music data. We choose Ram (On-Chip Memory) to store image data of small sprites (small image). The reason is SRAM has larger space but On-Chip Memory has steadier performance. When we read multiple images in the SRAM. It is probable for images to become blurry. Each item will handle the logic of itself, which outputs if the current pixel belongs to this item and the corresponding palette index.

- Form a complete project

By using audio control, the music can be played based on the game state. By using cursor, we can easily control the position and the function of the mouse. By using the drawing information from different items, color mapper can easily draw the whole image on the screen, which is controlled by VGA_controller. By using Nios II, we can get multiple keycodes to control the activity of the players.

3.2 State Machine Diagram



State Machine Diagram

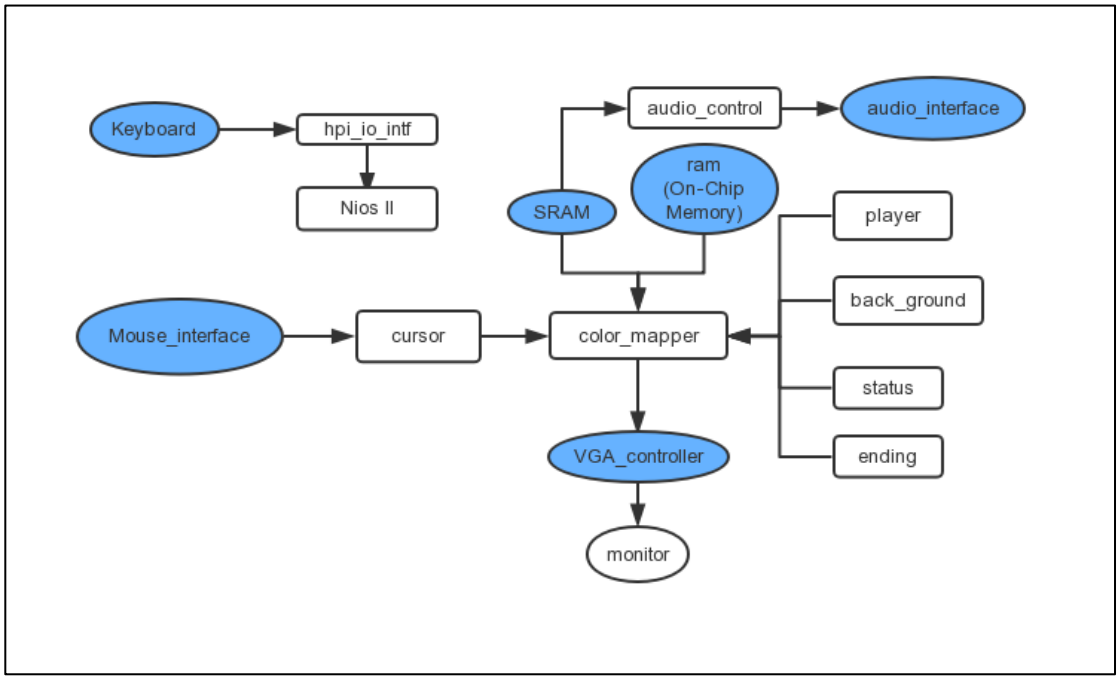
In our audio control state machine design, we interact with the five audio chip signals described in Section 2.3. Originally, the state machine stays at the "IDLE" state. Once we enter the main interface of the game "MUSIC_OFF" by a mouse click, `INIT`

is raised high. Then, it enters “WAIT_FOR_INIT” state. As soon as the state machine successfully capture the raised `INIT_FINISHED` signal, it will enter the next state “PLAY_MUSIC” to turn on the background music. In this state, we update the sound signal address in the SRAM according to the `data_over` signal. That is, next sound sample will not be fed into audio chip until the preceding sound sample is fully read and played. The sound SRAM address is updated in a cycle so that the background music is played in loops until interrupted by other sounds or the end of the game.

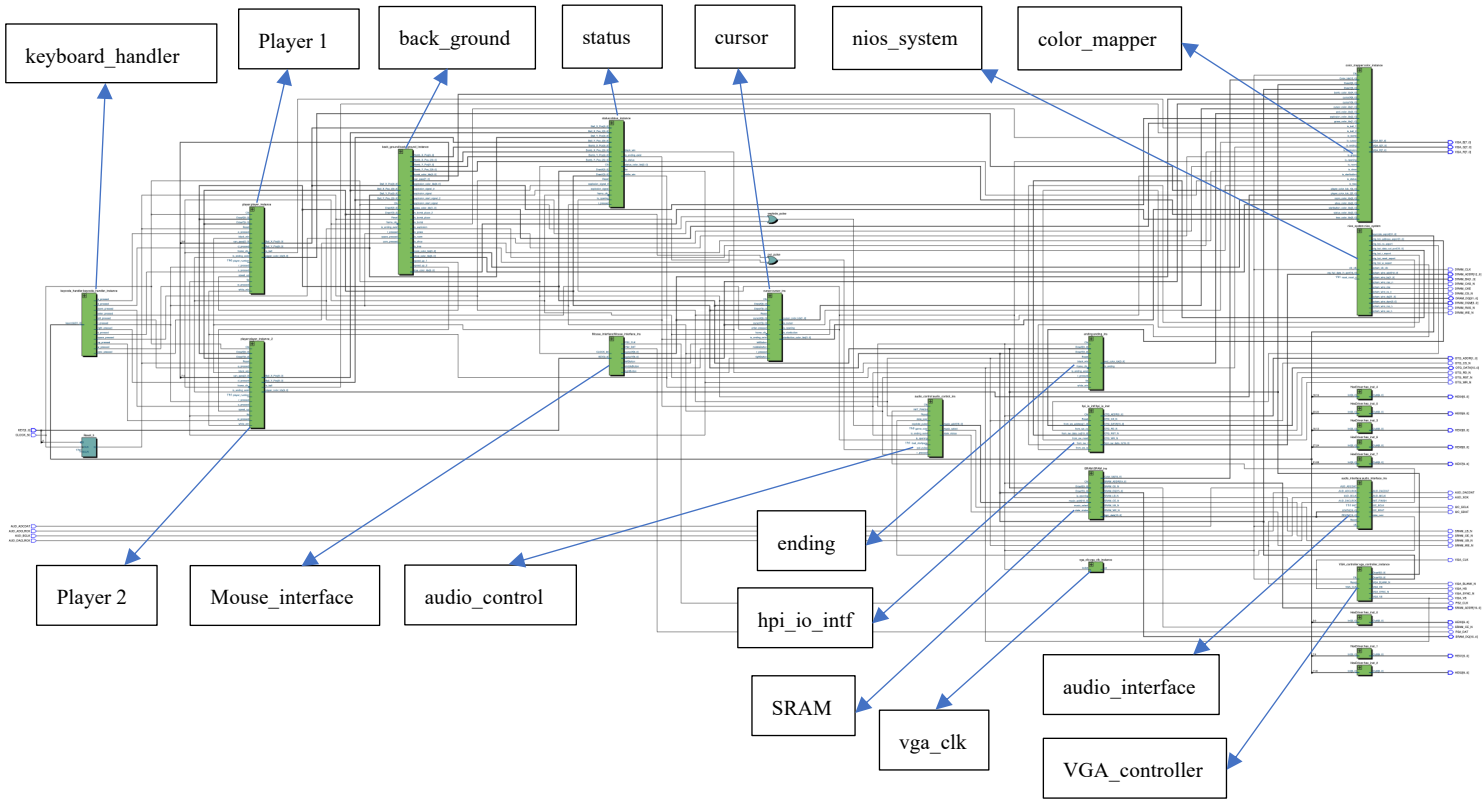
Besides the background music, we also incorporate the sound of put bomb and bomb explosion sound into our game to promote the game atmosphere. Each sound is implemented using two somewhat sub-states of the “PLAY_MUSIC.” That is, taking bomb explosion sound as an example; the first sub-state “PRE_EXPLOSION” initializes the starting condition for the specific piece of music, and the second sub-state “EXPLOSION” executes the music address update process (similar to the “PLAY_MUSIC” state) and quit at the end of music. The sub-states “PRE_PUT_BOMB” and “PUT_BOMB” are executed in a similar way. Our state machine design is also compatible with some special cases like multi-bomb issue (two players put bomb almost simultaneously). The second bomb signal will reset the starting address of the music piece in the sub-state to handle this issue.

The state machine will enter the end state of the game “GAMEEND” when the time is up or one of two players has zero health points, which is signaled by “is_ending_exist.” The “GAMEEND” is not really the end. If we press button “R” on the keyboard which is signaled by “r_pressed”, the game will restart again, i.e. go back to the “IDLE” state.

4 Block Diagram

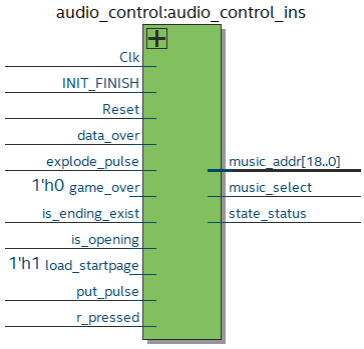


Brief Block Diagram (the Same as the Overview of the Circuit)

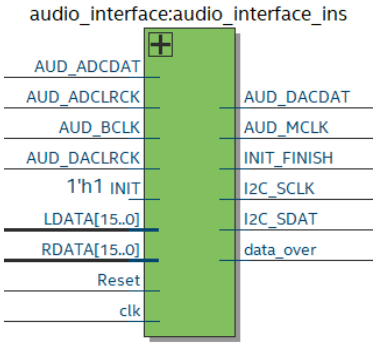


The Block Diagram of Top-level Circuit (Generated by Quartus)
(HD version is at the last page of the report)

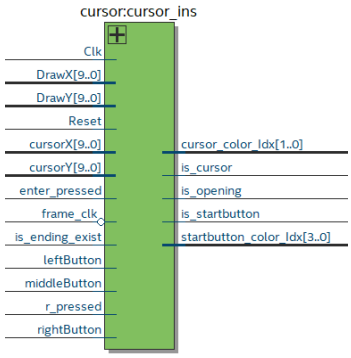
Because our project is not simple, it is hard to see the schematic diagram in one image clearly. Next, we will show the important modules one by one. What's more, you can see the overview clearly in the Appendix (last page) of our report.



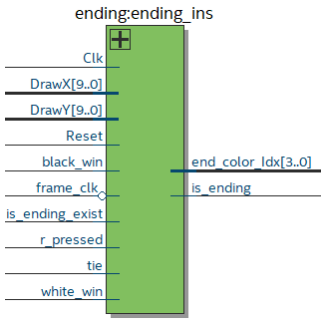
audio_control



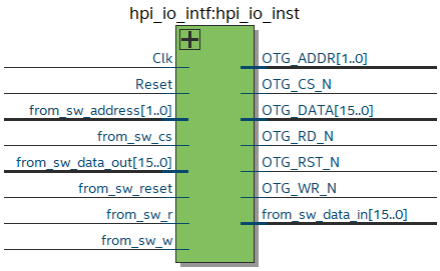
audio_interface



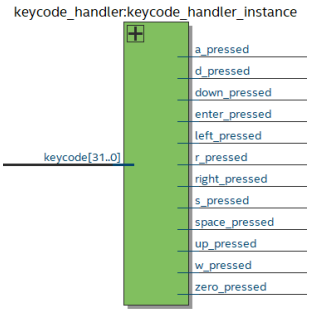
cursor



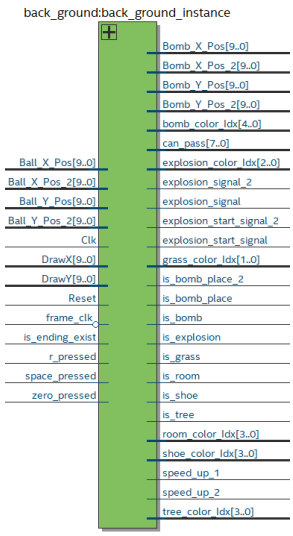
ending



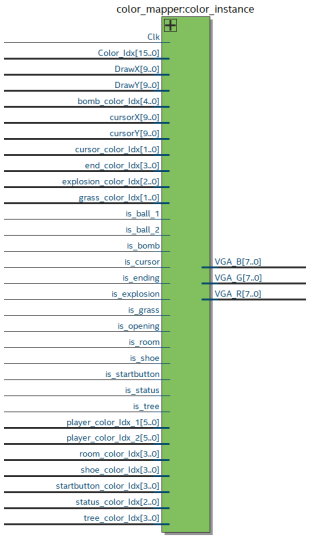
hpi_io_intf



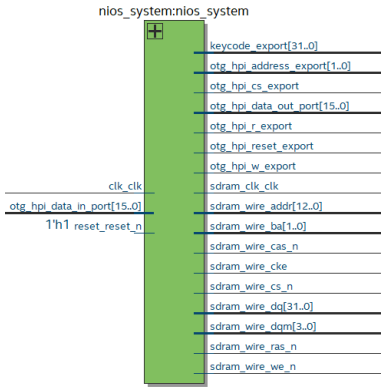
keycode_handler



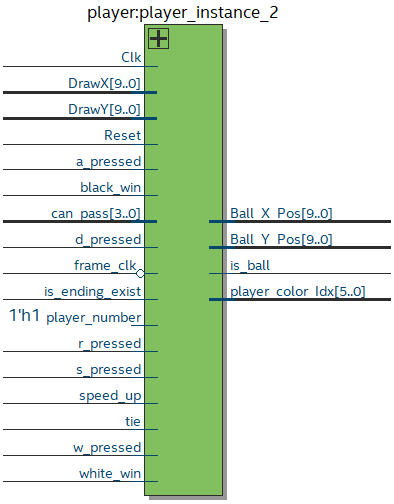
back_ground



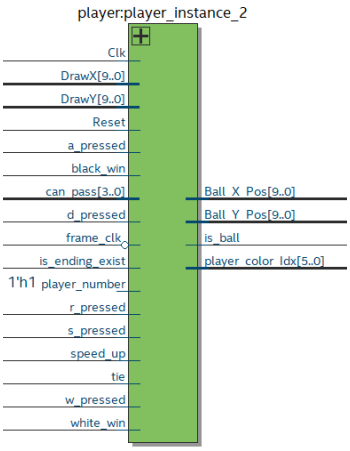
color_mapper



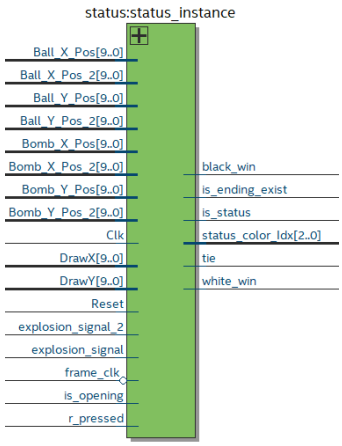
nios_system



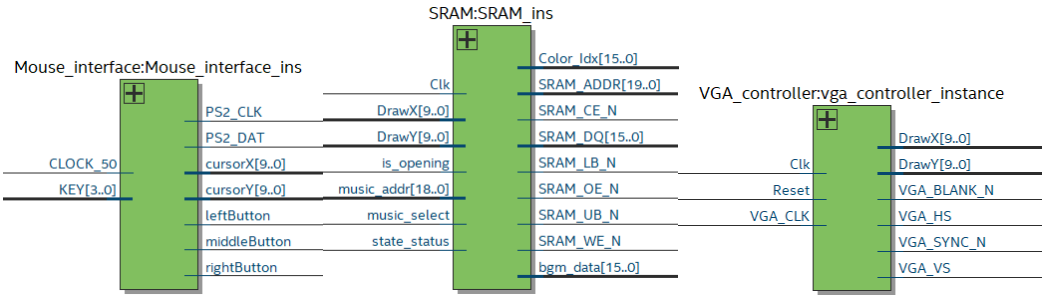
Player 1



Player 2



status



Mouse_interface SRAM VGA_controller

5 Design Statistics

LUT	26738
DSP	0
Memory (BRAM)	718848
Flip-Flop	3417
Frequency	122.4MHz
Static Power	107.18mW
Dynamic Power	0.8mW
Total Power	213.41mW

Design Resources and Statistics Table

6 Conclusion

- Bugs and Corrections

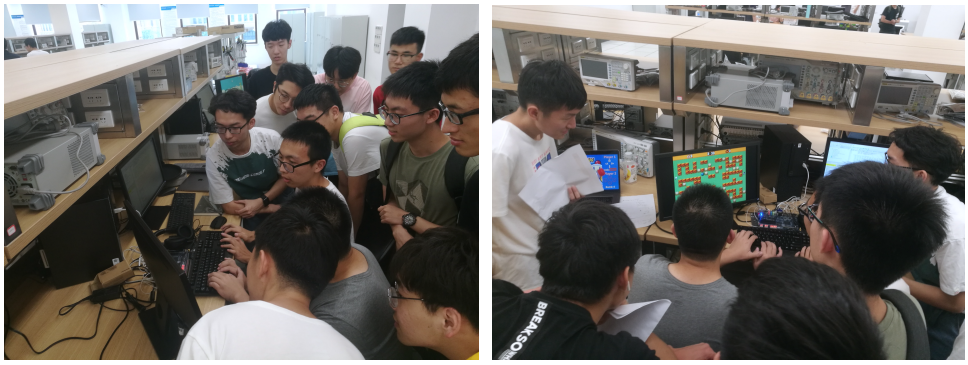
Function-ality Part	Bug Description	Resolution
Walking Animation and Player	This bug is a synchronization issue between animation play and player coordinate update. Occasionally,	This bug is probably the long-lasting one before the final resolution. Synchronization issue

Position (Display)	our players seemed to have the capability of the famous “space walk.” That is, if the key press time was shorter than the threshold time, the player was not supposed to move. However, in the real practice, the moving animation would be played and the player would flash back like the “space walk.”	usually comes with the clock settings. Thus, we drawed many simulation graphs of various tasks under several clocks. It was found that the frame_clk_rising_edge signal defined by us is different from the posedge notation. Thus, we put both two function parts under the posedge part to resolve the occasional asynchronization.
Audio	This bug was the most sophiscated one in our audio control part. It really shows the difficulty to work in hardware. It happened when we want to add bomb explosion sound after we successfully incorporate back-ground music sound in our game. It appeared that there was simply no sound when the state machine entered the bomb explosion state while the background music could still be played normally. It was very strange since the state to play bomb explosion is implemented almost the same way as the background music’s.	To handle multiple sounds in our game, we introduce an address `music_addr` to keep track of the music. It switches between the starting address of two pieces of music in SRAM. In the always_comb part of the audio state machine module, we first set the initial assignment of the `music_addr` signal. However, in the “if” statement of the following state, we forgot to set this address in the “else” part despite of initial assignment. After we assigned the address in the “else” part, this bug was resolved, and two music pieces could switch seamlessly.
Audio	There was always some sharp noise at the end of each piece of music when it’s played on FPGA board, which made the background music extremely uncomfortable for the players since it was displayed in loops. However, such kind of noise actually did not appear when those pieces of music are played in our computer. Thus, this bug was quite confusing.	When our audio control state machine executes music playing, there is an ending SRAM address as the end of the music. We round this ending address downto the closest 100s, which means about 1/100 percent of the music that locates at the end is cut. After this optimization, the ending noise is successfully removed.
Mouse	Our drawn mouse pattern did not correspond correctly to our actual mouse movement at the beginning. Sometimes, it even moved in the	It was found that the mouse should be connected with the FPGA DE2-115 board before we pressed the reset buttion. Only in this order,

	opposite direction of our mouse movement's.	the mouse could work normally as expected.

- Project Demo Issues

We accomplishes the project demo successfully without any bug occurrences. Many students and professors played our game during the demo; thus, it is heavily tested on the user level, and the functionality is fully proved.



Our On-Site Demo: Two Players Compete in *Bomberman*

- Accomplishment Summary

While the course slides warn us to "be wary of projects which require critical hardware to work," we fully take advantage of our learned knowledge from this course and employ them intensively in our almost hardware-based final project despite of the overwhelming difficulty. Additionally, this kind of difficulty in hardware is not unnecessary work compared to the software implementation, it significantly strengthens the high performance portion of our game. Certainly, we have also become more experted and experienced in the hardware programming as well as hardware/software partitioning.

Our bomberman game outperforms in both functionality depth and width. There are as many as six kinds of animation effects which are displayed and required to achieve highly precise synchronization with many other tasks like music playing, multi-player status update, collision detection, etc. In addition, we implemented many extra difficulty points and nearly all the features proposed in the project proposal: Mouse (only 1 group), sound (only 2 groups), multi-player (only 2 groups), etc. Despite of the unexpected difficulty and functionality performance, our game also wins the "Audience's Favorite Game" on the demo day, which proves that our sophiscated project design also wins the peer acknowledgement and great appreciation. Finally, we'd like to appreciate our professor and TA for your patient and kind advice throughout the whole project as well as reading this longish report.

References

[1] "PS/2 Controller," University of Toronto, [Online]. Available: http://www.eecg.toronto.edu/~jayar/ece241_08F/AudioVideoCores/ps2/ps2.html.

[2] Daniel Sanz Ausin and Fabian Goerge, "Design of an Audio Interface for Patmos," 2017. [Online]. Available: http://arxiv.org/abs/1701.06382.

[3] R. Thakkar, "Rishi's ECE 385 Helper Tools," [Online]. Available: https://github.com/Atrifex/ECE385-HelperTools.

[4] KTTECH, "PS/2 Mouse IP Core," [Online]. Available: https://kttechnology.wordpress.com/2017/04/29/ece-385-final-project-notes-ps2-mouse-ip-core/.