ECE 385

Spring 2019

Final Project

2-player Bomberman in System Verilog

Yifan Chen, Ziyang Xu
ABH / Friday 8:00-10:30am
Hadi Asgharimoghaddam, Lian Yu

Table of Content

Introduction

For the final project for ECE 385, we designed a game called “Bomberman”. The game is implemented in System Verilog along with NIOS II softcore processor. Our game supports two-player interactions, keyboard, mouse control and audio, video output based on DE-2 board. Our game inherits the basic rule of the original Bomberman game developed by Hudson Soft. The basic game rule is to beat down the enemy by placing bombs. Each player initially possesses three lives, and will lose one life once touched the flame of the exploding bomb. To add more flexibility and playability, we add treasures to change status of player. If one player eats the hat, it will be protected from being killed by the exploding bomb for about 5 seconds. If getting one bottle of liquid, the player will recover one life without upper limit. If he gets a shoe, he will be moving in faster speed for about 5 seconds. We also program the board to recognize whether a player wins the game or the two reach a tie. To support complete game experiment, we add loading page, starting page and game over page.

The majority logics of the game are implemented on FPGA using System Verilog. We support USB keyboard input device utilizing NIOS II processor. To response to keyboard control from two players, we add another PIO on the platform designer with 16-bit export data named keycode1 to support response for at most 4 key pressed at the same time. To support another external device -- mouse, we choose to use PS/2 port since there is only one USB port on board. The code for PS2 mouse is from ECE385 KTTECH [1]. The game is supported with audio output with audio interface from the course website by Koushik Roy [2]. The graphics and video effice of the game is drawn using 40 by 40 sprites stored in on-chip memory according to maps for multiple layers, while the starting and loading page animation is loaded into SRAM by control panel. The score recording and game result are displayed on the status bar separated from the game graphics using the given fonts from font-rom.sv [3].

Description & Operation of Circuit

Our design contains the following parts : game control, rendering logics, VGA controller, ROM, input devices.

Game control is implemented completely on SystemVerilog using several state machines in response to input signals from KEY buttons on board, keyboard and mouse in certain conditions following the rule. The out-most state is implemented to control the start and end of the game. After loading, it comes to START state, where we need to press on the “PRESS START” button on the screen using mouse to start. After pressing it, it comes to GAME_ON state, which outputs a signal to start the state machines for two players. The player state machine start to handle different the cases such as when this player die, is protected and the player finally runs out of lives and ends. To enable functions of other treasures, we write two other state machines to handle the following two circumstances respectively --

one life earned after encountering the liquid, and acceleration on player's speed after getting a shoe. To implement the effect of placing bomb and bomb exploding scene, we instantiate 4 bomb states for each player to enable 4 bombs running at the same time. The last part but also core of game control is to handle player movement. We inherit the ball module from lab 8 and modify it to let the player move only when instructing keycode is pressed.

The rendering logics again are divided to the data read of ROM, and the actually rendering to realize multi-layers and multi-parts effect. We use both on-chip ROM and SRAM to support the data reading for each pixel on the 480*640 screen. To draw the background scene in the beginning and after restart, we load a complete encoded image pixel by pixel from SRAM, which can support data fetched at high speed. In addition, we also store the "GAME OVER" sign at the end of the game in SRAM from a different starting address. As for the scene of the game, we create different sprites of size 40 by 40 and load the color indices (encoded image) onto the on-chip memory. For the characters, we draw 16 by 8 fonts. To locate each element(sprites and fonts), we created multiple **maps** for different purposes. We draw the corresponding elements according to DrawX/sizeX and DrawY/sizeY and retrieve color by DrawX%sizeX, DrawY%sizeY to transfer coordinates to the relative X and Y coordinates with respect to the top left pixel of that particular element. The multi-layers effect is realized by if-else statements according to state-indicators.

To interface keyboards, 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. We fill in IO_read, IO_write, UsbRead and UsbWrite completed in lab 8. 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 realizes the control of data transmission via USB port and manages structured information to and from the DE-2.

To interface PS/2 mouse, we added on-line mouse_controller.v and instantiate a mouse interface. To display the mouse, we draw a mouse with the coordinate of left top corner (cursorX, cursorY).

To interface audio, we added on-line audio_interface.vhd and we write a music control FSM module (we named it as music_control module) follow the instruction on the website. We used an app called "audacity" to convert a part of music into .wav format with 48kHZ and used matlab tool to convert this .wav document into hex and read them into music.txt. We them load the music.txt to the on-chip memory.

Hardware Module Descriptions

- **State Diagrams and Corresponding Module Descriptions**
    - **game state**

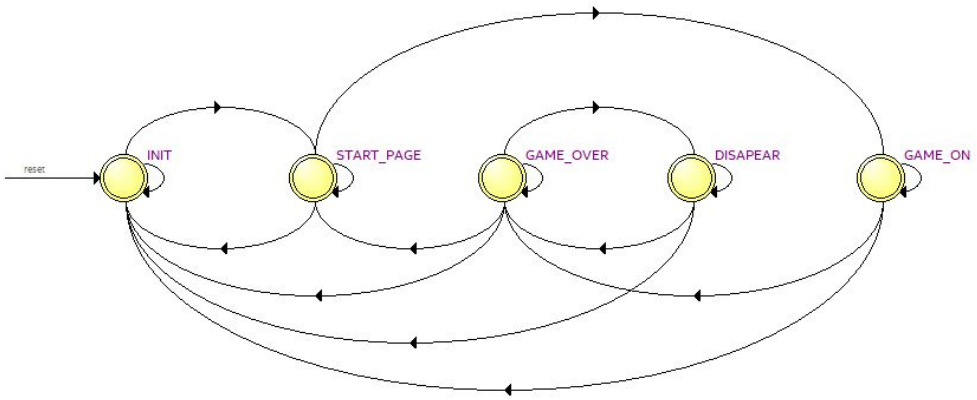


Figure 1. State Diagram of game_state.sv

State transition: INIT ----- load_finish -----> START_PAGE
START_PAGE ---------start ----------> GAME_ON
GAME_ON ----- player1_lives==0 or player2_lives==0 -----> GAME_OVER
GAME_OVER --------- little_counter1==0 ------> DISAPEAR
DISAPEAR ----- little_counter2==0 -----> GAME_OVER
GAME_OVER ----- restart -----> START_PAGE

Module: game_state.sv

Inputs: Clk, Reset, VGA_VS,
start, restart, load_finish,
[3:0] player1_lives, player2_lives,

Outputs: display_init, load_startpage, game_on, load_map, game_over,
draw_cursor, draw_gameover,
[7:0] counter

Description: This constructs an overall game state machine. It contains five states: INIT, START_PAGE, GAME_ON, GAME_OVER, DISAPEAR. INIT state can raise the signal *display_init* and load the initial page (see figure. 2) to VGA. START_PAGE can raise the signal *load_startpage*, *load_map* and *draw_cursor* and load the start page (see figure. 3) to VGA. GAME_ON state will raise the signal *game_on* which will activate the game and load the game to VGA. GAME_OVER state will raise the signal *draw_gameover* and *draw_cursor* and displays the cursor and "GAME OVER" (see figure. 5) to VGA. It also raises the signal *game_over* as control output. Especially, we instantiate two 8-bit counters to expand the time of GAME_OVER and DISAPEAR states to blink "GAME_OVER".

Purpose: This module divides the overall game into several parts and each part has a respective control signal.

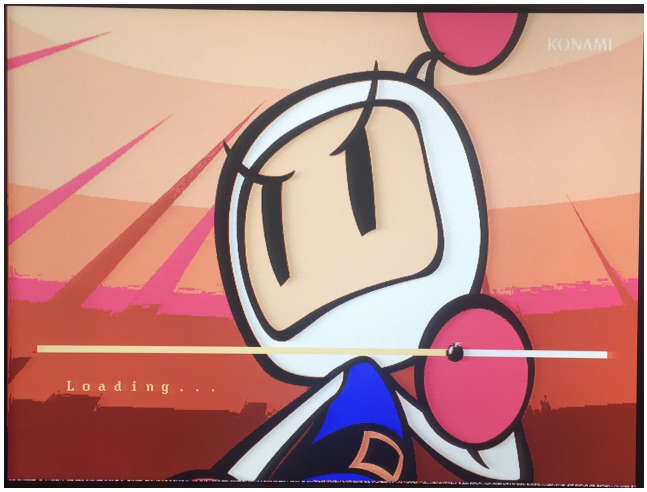


Figure 2. INIT state (Initial Page)

Figure 3. START_PAGE state (Start Page with cursor)

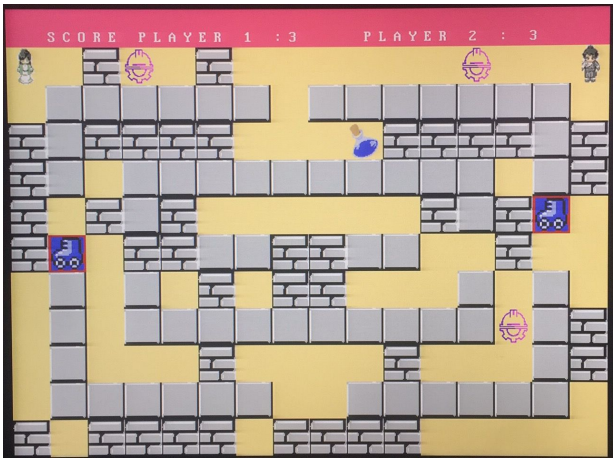


Figure 4. GAME_ON state (game page)

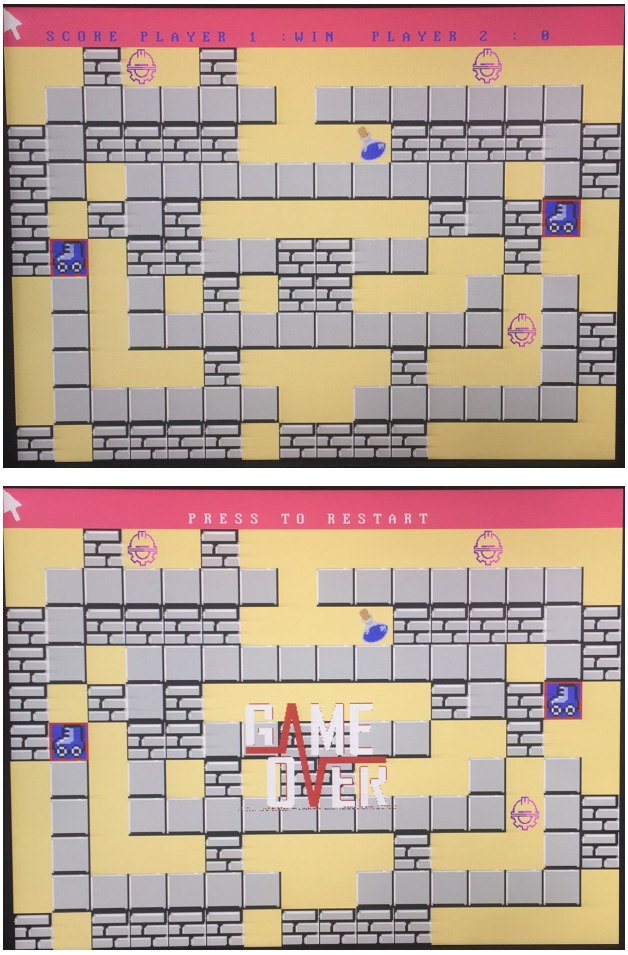


Figure 5. GAME_OVER state (with cursor and “GAME OVER”)

- **player state**

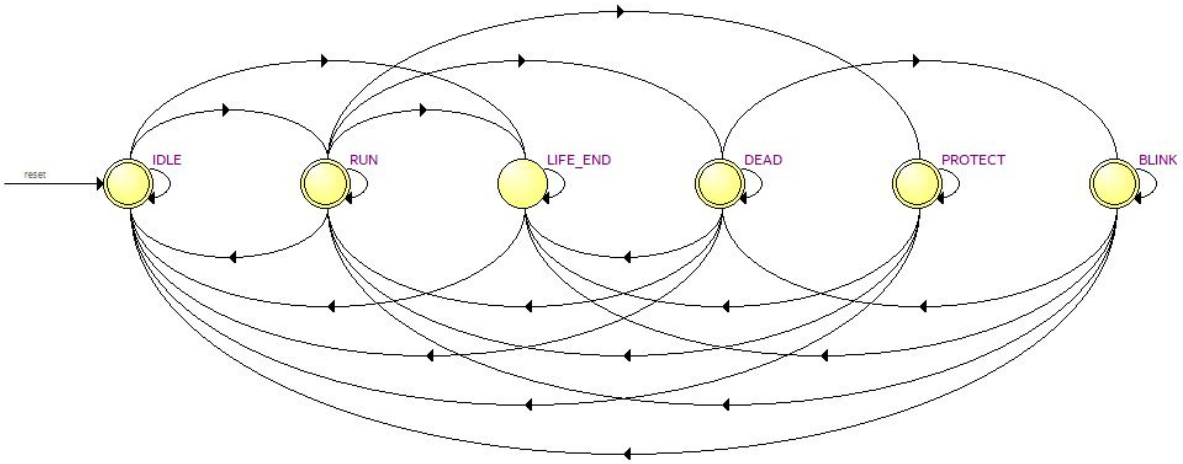


Figure 6. State Diagram of play_state.sv

State transition: IDLE ----- game_on -----> RUN

RUN ----- (the object in the map_array at the player's position is flame) -----> DEAD

RUN ----- protect -----> PROTECT

PROTECT ----- protect_counter==0 -----> RUN

DEAD ----- dead_counter%4==0 -----> BLINK

DEAD ----- dead_counter==0 -----> RUN

DEAD ----- player_lives==0 -----> LIFE_END

BLINK ----- dead_counter==0 -----> RUN

BLINK ----- dead_counter%4==0 -----> DEAD

arbitrary state ----- game_over -----> LIFE_END

Module: player_state

Inputs: Clk, Reset, VGA_VS,

game_on, life_plus, protect, game_over,

[7:0] counter,

[9:0] Ball_X_Pos, Ball_Y_Pos,

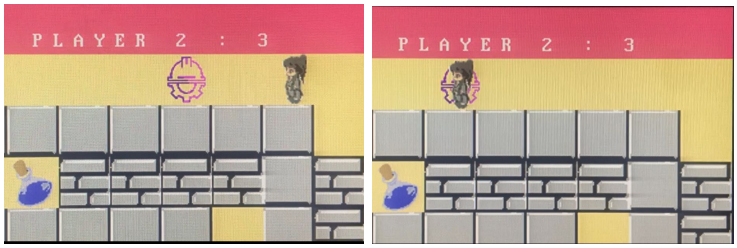
[0:12*16-1][3:0] map_array,

Outputs: player_alive, player_display, protecting

[3:0] player_lives

Description: This constructs a state machine of a player. IDLE state means that the player cannot be displayed on VGA and cannot move. RUN state will raise the signal *player_alive*, which allows the player to move, and the signal *player_display*, which allows the player to appear on VGA. PROTECT state will raise the signal *protecting*, which allows the player to move with a hat (see figure 7. b and c), and the signals *player_alive* and *player_display*. Since in PROTECT state, we do not check whether there is flame at the player's position, thus the player will not be exploded and seems to be protected by the hat (see figure 7. a to d). BLINK state will raise the signal *player_display*, while DEAD state will lower it. Therefore, alternating between DEAD and BLINK states will make the player seem to be blinking when the player is exploded by a bomb. During this period, the player cannot move (see figure 8. a to d). DEAD state will also raise the signal *LD_lives* to decrease the life counter by one (see figure 8. a and b). LIFE_END state means that one of the two players or both are dead. Especially, we instantiate two 8-bit counters to expand the time of PROTECT and DEAD and BLINK. We can see here DEAD and BLINK shares the same counter. Additionally, we instantiate a 4-bit counter to record the pH of the player (aka life counter, as in mentioned above).

Purpose: This module is used to control the status of the players.



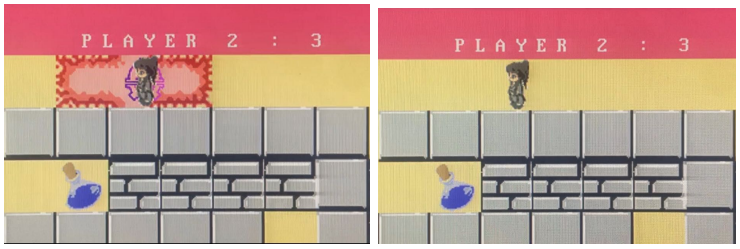


Figure 7. a, d RUN state; b, c PROTECT state

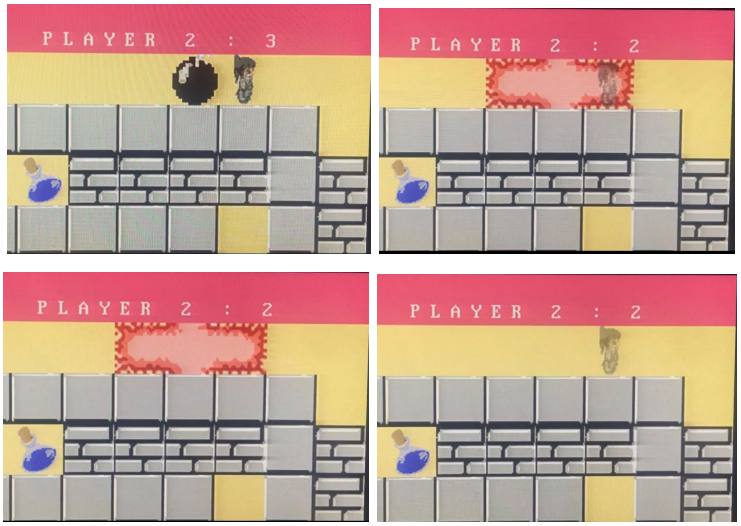


Figure 8. a, b, c, d Alternating between DEAD and BLINK states

- **bomb state**

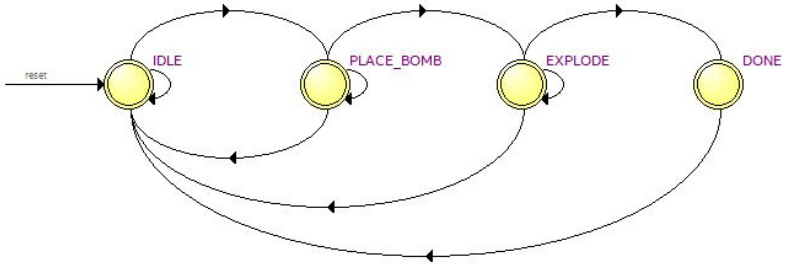


Figure 9. State Diagram of bomb_state.sv

State transition: IDLE ----- bombstart -----> PLACE_BOMB
PLACE_BOMB ----- counter==0 -----> EXPLODED
EXPLODED ----- explode_wait==0 -----> DONE
DONE ----------> IDLE

Module: bomb_state

Inputs: Clk, Reset, VGA_VS,

bombstart,

Outputs: bomb_on, bomb_placed, bomb_exploding, bomb_exploded

Description: This constructs a bomb state to control each bomb. IDLE means the bomb is not placed in the game. If the bomb is placed, the signal *bomb_on* will be constantly raised until the bomb is exploded. PLACE_BOMB state will raise the signal *bomb_placed* and a bomb will be placed (see figure 10. a). EXPLODE state will raise the signal *bomb_exploding* to display the exploding process (see figure 10. c). DONE state will raise the signal *bomb_exploded* and the objects that can be eliminated, such as brick, will disappear (see figure 10. d). Especially, we instantiate two 8-bit counters to expand the time of PLACE_BOMB and EXPLODE states.

Purpose: This module uses the output signals to change the content of the map_array, and therefore, a modified map can be displayed on VGA.

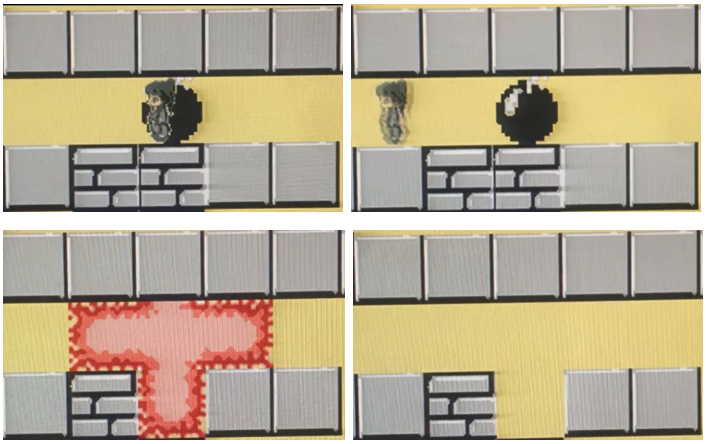


Figure 10. a, PLACE_BOMB state; c EXPLODE state; d DONE state

- ## life plus state

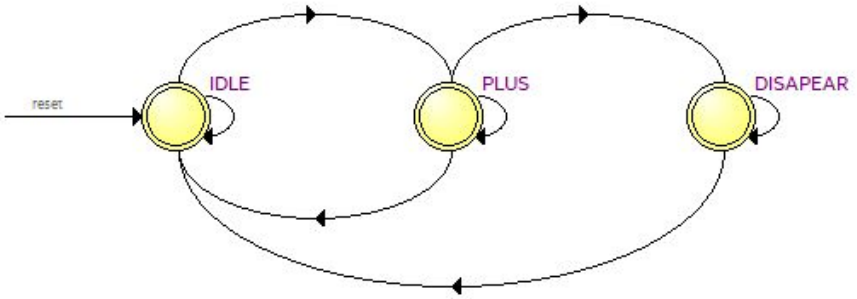


Figure 11. State Diagram of lifeplus_state.sv

State transition: IDLE ----- life_plus -----> PLUS
PLUS ----- plus_counter==0 -----> DISAPEAR
DISAPEAR ----- disapearcounter==0 -----> IDLE

Module: lifeplus_state

Inputs: Clk, Reset, VGA_VS,
life_plus,
Outputs: plus1,
[7:0] disapearcounter
Description: This constructs a state machine that will be revoked when the player drink the liquid. IDLE state means no liquid is taken by the players. PLUS state will raise the signal *plus1* and display "+1" behind the pH of the player (see figure 12. b). DISAPEAR state will continuously raise the signal *plus1* and eliminate the value of pH and "+1" (see figure 12. c). Especially, we instantiate two 8-bit counters to expand the time of PLUS state and DISAPEAR state.
Purpose: This module is used to change the content of the status_array, and therefore, a modified status bar can be displayed on the top of VGA.

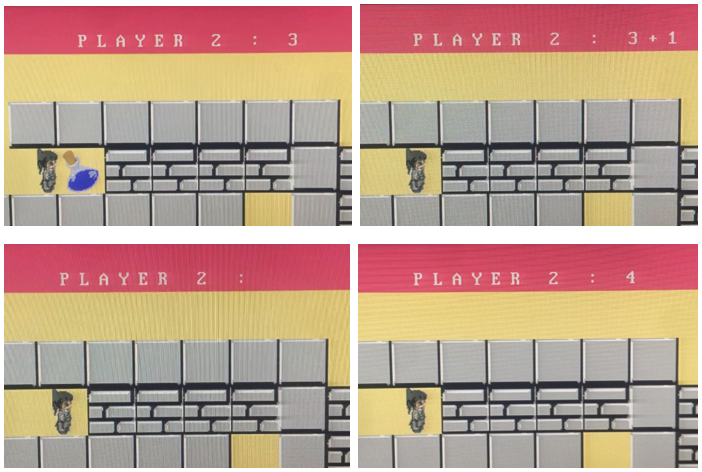


Figure 12. a, b, c, d Change of the Status Bar after Taking Liquid

- **shoe state**

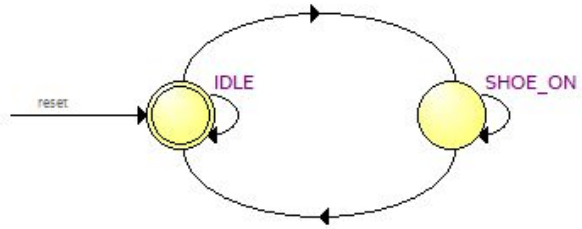


Figure 13. State Diagram of shoe_state.sv

State transition: IDLE ----- life_plus -----> SHOE_ON
SHOE_ON ----- counter==0 -----> IDLE

Module: shoe_state

Inputs: Clk, Reset, VGA_VS,
shoe,
Outputs: shoe_on
Description: This constructs a state machine that will be revoked when the player take the shoes. IDLE means no shoes are taken by the players. SHOE_ON state will raise the signal *shoe_on* and the speed of the player will be changed from 2 to 4. Especially, we instantiate a 8-bit counter to expand the time of SHOE_ON state.
Purpose: This module is used to change the speed of the players.

- Other Game Logics
  - **ball (player)**

Module: ball
Inputs: Clk, Reset, frame_clk,
player_id, player_alive, protecting, shoe_on, game_over,
[9:0] DrawX, DrawY,
[15:0] keycode, keycode1,
[0:12*16-1][3:0] map_array,
Outputs: is_ball,
[4:0] coloridx,
[9:0] Ball_X_Pos, Ball_Y_Pos
Description: This module mainly controls the motion of the players. It accept the input *player_id* to tell player1 and player2. For player1, the direction variables (W, A, S, D) relate to one group of keycode and for player2, the direction variables relate to another group of keycode. *keycode* and *keycode1* contains the keycode we have pressed on the keyboard. We use keycode as select signal to decide the the value of a variable aspect (0: back 1: front 2: right 3: left). *play_alive* enables the player to move, but if *game_over* is raised, the player still cannot move. *protecting* will allow the player walk with a hat, as is mentioned in player_state module. *shoe_on* will change the speed of the player.
In this ball module, we internally instantiate nine sprites modules: player1_front, player1_left, player1_right, player1_back, player2_front, player2_left, player2_right, player2_back and hat. We use aspect as select signal to decide the *coloridx*. This module outputs the signal *is_ball* and the corresponding *coloridx* at the pixel coordinate (*DrawX*, *DrawY*). Additionally, we instantiate is_barrier module to restrict the player to only go along the ground and cannot go through the walls and bricks.
Purpose: This module connects with lab_8 module to color the pixel at coordinate (DrawX, DrawY).

  - **is_barrier**

Module: is_barrier
Inputs: [9:0] Ball_X_Pos, Ball_Y_Pos,
[0:12*16-1][3:0] map_array,

Outputs: stop

Description: This module outputs the signal *stop* to restrict the players' motion. We check four sides of the player, and for each sides we check three points: the middle point and two endpoints.

Purpose: This module is revoked in ball module to check the boundary of the players' motion.

- ### bomb

Module: bomb

Inputs: Clk, Reset,
[15:0] keycode, keycode1,
[9:0] Ball_X_Pos, Ball_Y_Pos,
player_id, [3:0] bomb_on

Outputs: [3:0][5:0] Xbomb, Ybomb, [3:0] bombstart

Description: In this module, we use the four bit bomb_on from bomb_state to decide which bomb is initiate if player presses to place a bomb. The rule is to start to activate from the lowest bit and in order to the most significant one. At the same time, we maintain the location of bomb in map_array using registers Xbomb, Ybomb while the bomb is on.

Purpose: This module is to decide and store the information of 4 bombs of each player.

- rendering logics

  - ### ColorMapper

Module: color_mapper

Inputs: [9:0] DrawX, DrawY,
[3:0] sprite_idx, treasure_idx,

Outputs: [4:0] coloridx1

Description: this module instantiate all the sprites we need for the treasure map. We calculate the relative x and y coordinate of the pixel being drawn with respect to the top left pixel of that particular sprite. The sprite_idx and treasure_idx are sprite index at DrawX/40 and DrawY/40 in the corresponding map. We output color of the potential treasure if the index is not zero, or it outputs the color of a sprite in map array.

Treasure_idx: 4'h0 - ground; 4'h1 - liquid; 4'h2 - hat; 4'h3 - shoe;

Sprite_idx : 4'h0 - ground; 4'h1- brick; 4'h2 - wall; 4'h3 - bcenter (flame center)
4'h4 - player1; 4'h5 - player2; 4'h6 - bomb; 4'h7 - bup(upper flame);
4'h8 - blow(lower flame); 4'h9 - bright(right flame); 4'ha - bleft(left flame)

Purpose: This module output color index for the corresponding sprite indicated by trerasure_array and map_array to draw at current VGA rendering location by further logics.

  - ### statusbar_mapper

Module: statusbar_mapper

Inputs: [9:0] DrawX, DrawY,
[3:0] status_array,

game_over,

Outputs: [4:0] coloridx_bar

Description: this module instantiate all the sprites we need for the treasure map. We calculate the relative x and y coordinate of the pixel being drawn with respect to the top left pixel of that particular font. We output color of the potential font if the index of the array is not zero, or it outputs the background color of status array (pink). The status bar will change the color of its character in different cases. When game is over, the color is changed to blue while it normally stays in white.

Purpose: This module outputs color index for the corresponding sprite indicated by status_array to draw at current VGA rendering location by further logics.

- ### loading

Module: loading

Inputs: Clk, Reset, frame_clk,

[9:0] DrawX, DrawY,

Outputs: load_finish,

is_bomb, is_bar_left, is_bar_right, is_loading,

[4:0] coloridx,

Description: This module is pretty similar with ball module. It allows a small bomb automatically to move in the initial page. is_bar_left and is_bar_right are used to color the two bar sides of the small bomb with different colors. is_loading means the pixel at (DrawX, DrawY) is a part of the "Loading ..." (see figure 14). Here we instantiate font_rom module.

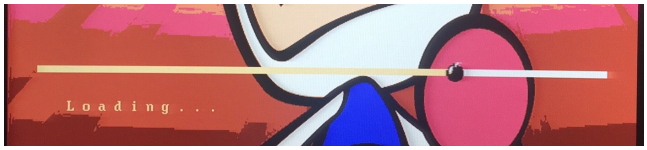


Figure 14. bar, small bomb, "Loading"

Purpose: This module is used to decorate the initial page.

- ### press_start

Module: press_start

Inputs: leftButton,

[9:0] DrawX, DrawY, cursorX, cursorY,

Outputs: is_press_start, is_pressed, is_start_range, click_start

Description: This module *is_press_start* will color "PRESS START" with a color white. *is_pressed* means that the cursor is placed over "PRESS START" and these characters will be colored by another color yellow (see figure 15. and 16.). *is_start_range* will color the specified range around "PRESS START" with same color. Here we instantiate font_rom module.

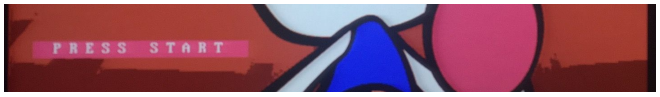


Figure 15. “PRESS START”

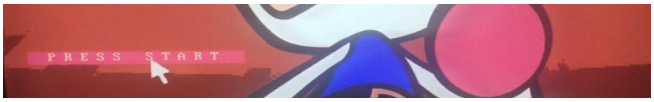


Figure 16. the cursor placed over “PRESS START”

Additionally, if *leftButton* is raised and the cursor is in the specified range (pink block above) at the same time, *click_start* will be raised and the game state machine will turn START_PAGE state to GAME_ON state.

Purpose: This module is used to decorate the start page and connects the start page with the game page.

- ### color-rendering part in lab8.sv

Inputs: [9:0] DrawX, DrawY,
[4:0] coloridx1, coloridx_bar,
[19:0] SRAM_ADDR,
[15:0] SRAM_DQ,
game_over, is_restart, display_init, is_bar_left, is_bar_right, is_loading, load_startpage, is_press_start, is_start_range, is_pressed

Outputs: [4:0] coloridx

Description: First draw cursor if there is one (the mouse), then draw bomb and background of the loading page if it’s currently loading. After that, it puts the “PRESS START” button onto the background image of starting page if load_startpage is active. Then, we draw status bar indicating who wins if the counter for “GAME OVER” blink isn’t over, while the counter is done counting, we will draw “PRESS TO RESTART” in white if cursor is within the range of status bar and blue otherwise. And finally, we will draw two players and then the color for the specific sprite in the game page.

Purpose: This module does final decision of multi-layer rendering of VGA. It will generate the final coloridx that needs to draw to the current location indicated by DrawX and DrawY under the status of current input signals.

- ### palette

Inputs: [4:0] coloridx

Outputs: [7:0] Rout, Gout, Bout

Purpose: Output the corresponding RGB color as indicated by the input coloridx

- ### map

Module: map

Inputs: Clk, load_map,
[3:0][5:0] Xbomb1, Ybomb1, Xbomb2, Ybomb2,
[3:0] bomb_placed1, bomb_placed2, bomb_exploding1, bomb_exploding2,
[3:0] bomb_exploded1, bomb_exploded2,

[3:0] player1_lives, player2_lives,

game_over, plus1_1, plus2_2,

[7:0] counter, disapearcounter1, disapearcounter2

Outputs: [0:12*16-1][3:0] map_array, treasure_array,

life_plus1, life_plus2, protect1, protect2, shoe1, shoe2

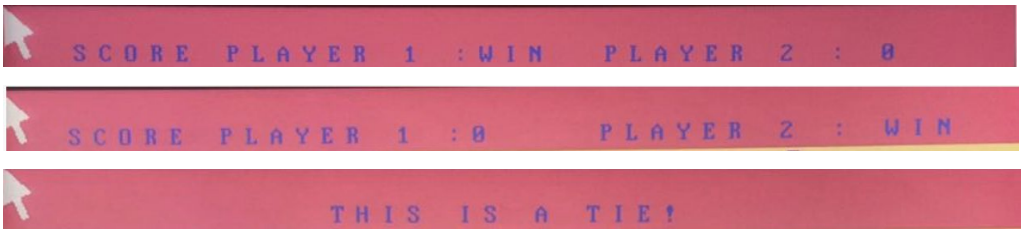
Description:

All the three arrays are reset to the original ones if and only if load_map is active.

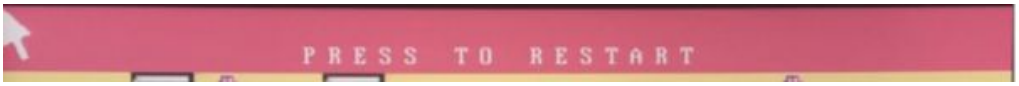
1. Update status_array

During the game, the status array only update the lives of each player.

When the game over counter to count time of blink isn't over, the status array is updated show the result.



When counting is over, the status array is updated to show



When encountering the liquid, the status bar will also be updated according to the signal plus1_1 and plus1_2. The effect has been displayed above in the **life plus** state machine.

2. Update map_array

Map_array is updated following the status of active bomb indicated by bomb_placed(1,2 shows which player placed this bomb), bomb_exploding(1,2), bomb_exploded(1,2).

Once bomb_placed is high, there will be a bomb placed at the corresponding bomb location stored by registers Xbomb(1,2) Ybomb(1,2).

When bomb_exploding is high, the center, upper, lower, left, right cells to the bomb location are being updated to corresponding flame sprites only if origin_map tells us these locations were not wall. That is, the flame will only appear on those cells that are not wall. Also, we need to do boundary check when updating the cell to prevent part of the flame appearing at the other side of the screen.

When bomb_exploded for corresponding bomb is high, all of the upper, lower, left, right and center cells from the bomb locations are cleared if these cells were not wall in the origin_map nor out of bound.

3. Update treasure_array

Treasure will become zero after either player encounters it. As the treasure_array is updated, there will also be signals indicating which treasure is going to be effective later on.

Purpose: This module is to update each map according to input signal so as to synchronize with the state machines. It also output signals indicating which treasure is going to be active in the next cycle.

- ROM
  - **sprites and fonts_rom**

    This part contains multiple modules. Visualization can be view in *Sprites Description and Color Encoding* part.
  - **bgm (music.txt)**

Module: bgm

Inputs: Clk,

[18:0] Addr,

Outputs: [15:0] music_data

Description: This module uses “$readmemh("music.txt", music_memory)” to load the music.txt to the ROM.

- input devices
  - **audio**

Module: music_control

Inputs: Clk, Reset

INIT_FINISH, data_over, game_over, load_startpage

Outputs: INIT,

[18:0] Addr

Purpose: This play the music when them signal *load_startpage* is raised and ends when the signal *game_over* is raised.

  - **mouse**

Module: Mouse_interface

Inputs: CLOCK_50, [3:0] KEY,

Outputs: leftButton, middleButton, rightButton,

[9:0] cursorX, cursorY

Inouts: PS2_CLK, PS2_DAT,

Purpose: This interface handles the input PS2_DAT and is modified to display mouse within the boundaries of a VGA display.

Module: PS2_Controller

Inputs: clk, reset,

[7:0] the_command, send_command,

Outputs: command_was_sent, error_communication_timed_out,

[7:0] received_data, received_data_en,

Inouts: PS2_CLK, PS2_DAT,

Purpose: This code is based on Altera's PS/2 IP core.

- **keyboard**

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,

Inouts: [15:0] OTG_DATA

Description: This module uses OTG_DATA as a bidirectional wire to transfer the data. Each OTG-signal corresponds to a from_sw-signal. In this case, EZ-OTG acts as a Host Controller to control the connected USB keyboard.

Purpose: This module is used to form the interface between NIOS II and EZ-OTG chip.

- VGA controller
  - **VGA_CLK**

Module: vga_clk

Inputs: inclk0,

Outputs: c0

Description: This module accepts an input clock signal inclk0, and then produces another clock signal c0.

Purpose: This module uses PLL to generate the 25MHZ VGA_CLK.

- **VGA_controller**

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 accepts the VGA_CLK to traversal the 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 traversal the screen and return the coordinates of the current traversaled pixel (DrawX and DrawY).

Purpose: This module is used to output the coordinates of the current traversaled pixel and connect to the *color_mapper* module and *ball* module.

- Nois II
  - **lab8_soc**

Module: lab8_soc

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,

Inouts : [31:0] sdram_wire_dq

Description: This constructs the Nios II 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 8-bit signal called keycode_export to represents the key on the USB keyboard. This keycode_export is used to connect with *ball* module to control ball motion.

Purpose: This module is used to build a Nios II system, which can implement the foundation of our System-On-Chip (SOC) projects.

Sprites Description and Color Encoding

We have 22 sprites in total. They are:
player1_front, player1_left, player1_right, player1_back, player2_front, player2_left, player2_right, player2_back,

brick, wall, bup, bleft, blow, bright, bcenter, bomb, hat, cursor, liquid, shoe, small_bomb (resize bomb to 20x20)

We also support write of all characters of size 16*8. Those data are stored in fonts_rom.
The RGB colors are encoded to 5-bit index in our design to support 32 colors at most, and we have used 26 of them. All the encoded color can be seen in palette.sv, so we won't state them again here in the report.

Modifications in Software

We modify the memory written method of keycode from *keycode_base = keycode & 0xff to IOWR(base, offset, data) where base is the base address of the peripheral you are accessing. Offset is the word offset of the register you are accessing in the peripheral. The word size is assumed to be 32-bit so offsets 0, 1, 2, 3, etc... map to byte offsets 0, 4, 8, 12, etc....
Here are code modified from main.c under software folder.

```
usb_ctl_val = UsbWaitTDListDone();
// The first two keycodes are stored in 0x051E. Other keycodes are in
// subsequent addresses.
keycode = UsbRead(0x051e);
printf("\nfirst two keycode values are %04x\n",keycode);
// We only need the first keycode, which is at the lower byte of keycode.
// Send the keycode to hardware via PIO.
IOWR(keycode_base, 0, keycode & 0xffff);

keycode1 = UsbRead(0x0520);
printf("\nsecond two keycode values are %04x\n",keycode1);
// We only need the first keycode, which is at the lower byte of keycode.
// Send the keycode to hardware via PIO.
IOWR(keycode1_base, 0, keycode1 & 0xffff);

usleep(200);//usleep(5000);
usb_ctl_val = UsbRead(ctl_reg);

if(!(usb_ctl_val & no_device))
{
        //USB hot plug routine
        for(hot_plug_count = 0 ; hot_plug_count < 7 ; hot_plug_count++)
        {
                usleep(5*1000);
                usb_ctl_val = UsbRead(ctl_reg);
```

```
            if(usb_ctl_val & no_device) break;
        }
        if(!(usb_ctl_val & no_device))
        {
            printf("\n[INFO]: the keyboard has been removed!!! \n");
            printf("[INFO]: please insert again!!! \n");
        }
}
```

Schematic Block diagram

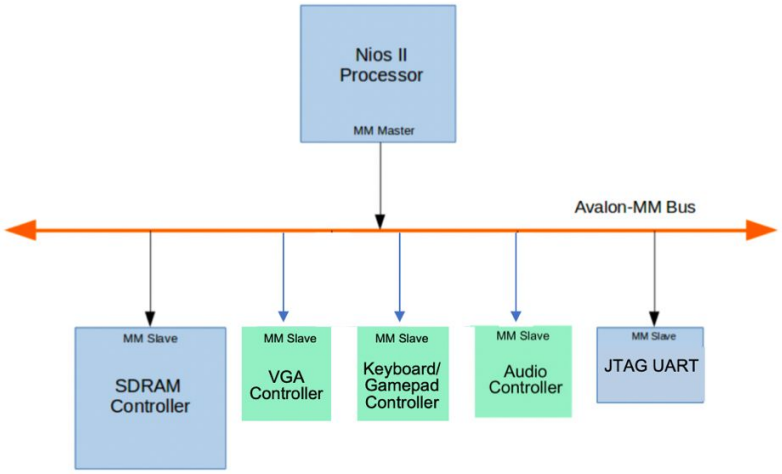
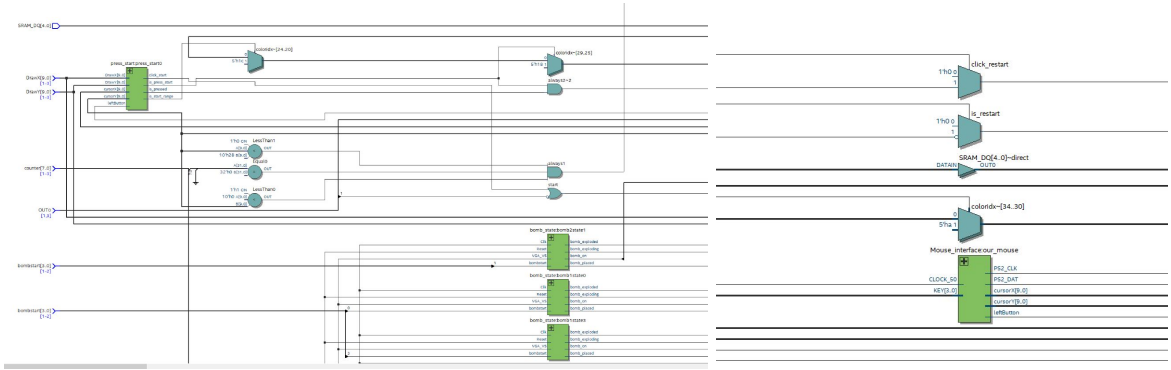


Figure 17. Block Diagram of our design

Since our block diagram is very complex and too dense, here we only show fragments from the whole diagram.



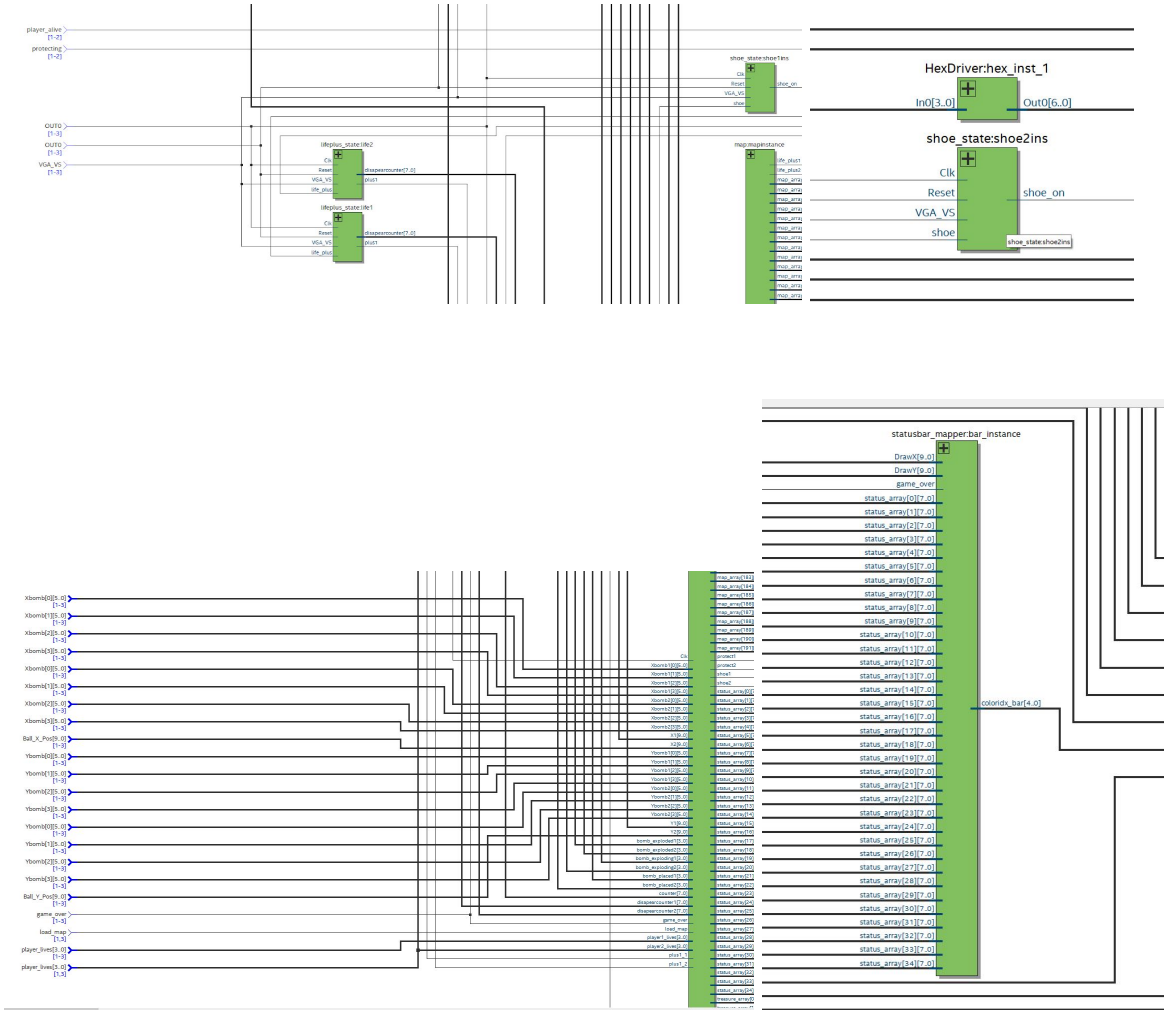


Figure 18. Block diagram fragments generated by quartus

Resource Usage

- Document the Design Resources and Statistics in following table

LUT	70307
DSP	0
Memory (BRAM)	1640448 (41%)
Flip-Flop	3792
Frequency	128.66 MHZ

Static Power	111.11mW
Dynamic Power	0.83mW
Total Power	216.89mW

Table 1. Table for Design Statistics

Conclusion

Our project is challenging. By the time we do the demo, we still think about adding more features to the game, such as some movable enemies that can kill the player by touching his/her. The good part is that we finished implementing the basic game and realized most of the extra parts.

The project helps us integrate what we've learnt in ECE 385. It's a time-consuming project that we have spent approximately 150-180 hours on it. While we still enjoy doing it, both the debugging part and the implementing part. During this project, we reviewed how to use SRAM, on-chip memory, VGA drawing. We also explored how to simplify our state machine, how to build sophisticated animation, how to enable audio output and so on.

We find enabling 2 players with 4 bombs at one time each is the most difficult part. We met several bugs due to the ignorant declaration on number of bits. Another problem is to draw the player, we need to locate player to its center and use power of distance to check, because simply use subtraction brings out problem. To read from SRAM, we need to maintain the address length to be 20 bits, or it will locates to the wrong address.

After debugging, our game can be played smoothly and it's user friendly.

Reference

[1] [ECE 385] Final Project Notes – PS/2 Mouse IP Core
https://kttechnology.wordpress.com/2017/04/29/ece-385-final-project-notes-ps2-mouse-ip-core/

[2] Koushik Roy Audio Interface in Verliog https://wiki.illinois.edu/wiki/display/ece385sp19/Final+Project

[3] Font file in the form of sprite table
https://wiki.illinois.edu/wiki/display/ece385sp19/Final+Project