ECE 385

Fall 2017

Final Project

Game: I Wanna Be 385

Ruhao Xia, Ximin Lin

AB2 / Friday 3:00 pm

Po-Han Huang

Introduction

In this final project, we designed a game from the game series: I wanna be. Our game supports NIOS II SOC to support keyboard control, VGA to display the game, Audio to play the background music, SDRAM to store the image sprites, SRAM as double buffer for the project, Flash for storing the music, on-chip memory to store the NIOS C program. This game features two stages. The first is a simple Mario-like game, where you control the character to pass through different traps to reach the portal to the next level. In the second stage, you will control the character to escape the attacks of a Boss and try to shoot the Boss. This game also supports save and load of different game stages. If you have entered the second stage of the game but killed by Boss, when you load the game again, you will be in the second stage of the game. After you have killed the Boss, you will enter a Final Grade page where you can randomly draw a letter from fastly-blinking letters of A to F to represent your final grade. In the welcome page, if you choose to exit by pressing the Esc in the keyboard, you get to see the legend of ECE 385, Prof. Deming Chen in the screen.

Written Description of Display Logic

We map all our images to a palette of 206 colors. We will then need 8 bits for color of each pixel. We used SDRAM to store our image sprites and SRAM as our double buffer. We run both SRAM and SDRAM at 50 MHz. Because we are using SRAM for double buffering, we build a state machine in SRAM interface that alternates between the reading state and writing state. So for every other 50 MHz rising edge, we will read one 8-bits palette index from SRAM. We will use the Read state of SRAM to preload the 8-bits value to be written into SRAM into a write buffer. When the SRAM is in write state, we will set WE_N low to write to SRAM.

We have measured that at most time, the valid data is five clocks delay after we give the SDRAM correct address. But it is OK since we are using burst mode of reading. Once we are doing consecutive reading from the same row, we can get valid data by only on cycle. However, in this case, the reading speed of SDRAM is faster than the writing speed of SRAM, it is possible that SDRAM has read more data to be written to SRAM. To compensate for this behavior, we build a circular buffer between SDRAM and SRAM that can store 64 8-bits data to be sent to SRAM. We build a sdram control module print, which contains state machine that has three states: reset, initialization, and normal. During reset state, the print module will wait until the waitrequest signal of SDRAM is low, so allowing data to be read. Then from reset state, the print module will go to initialization state. In initialization state, the print module will wait for five cycles until the first data will become valid in the DRAM_DATA. Then the print module will go to normal state to constantly update the sdram read address if waitrequest signal is low. So the data for each pixel will be valid for only one cycle.

In case that the buffer between SDRAM and SRAM is about to be filled with new data to be written to SRAM, the buffer will send signal buffer_full to the print module to prevent the interface to read later address. Since the data from SDRAM will be valid 5 cycles after the address is sent to SDRAM, we will need to let the print module to stop reading more data when the sdram buffer is 5 data before full.

VGA has its 10-bits DrawX and DrawY signals for reading from SRAM. The print module also produces 10-bits DrawX_write and DrawY_write for writing to SRAM. These two sets of DrawX and DrawY are independent of each other. When the buffer_full signal is received in the print module, DrawX_write and DrawY_write will be fixed to current value until SRAM has read enough values from the buffer (In our case, when there are 32 free space in the buffer for new values read from SDRAM).

For double buffering, we are running our game in 30 Hz just because the 60Hz speed is too fast to play with. We are using the VGA_controller.sv from lab 8.

Taken the method from Ball.sv from lab 8, we draw different component of image using different modules. Each module will give a correct SDRAM address corresponding to that pixel value and a signal telling whether DrawX_write and DrawY_write is drawing this component right now. On top of these modules, we have an address selector that takes all the SDRAM address signal from these modules, and will determine the priority of these SDRAM addresses to be sent to SDRAM. Because we have different stages of game, including welcome, first level, second level, gameover, finalscore, and exit. Different stages of the game will draw to the screen with different logics. So we put different address selectors inside different stage modules. And on top of these stages modules, we have an address selector module that can decide what stage's image should be displayed right now.

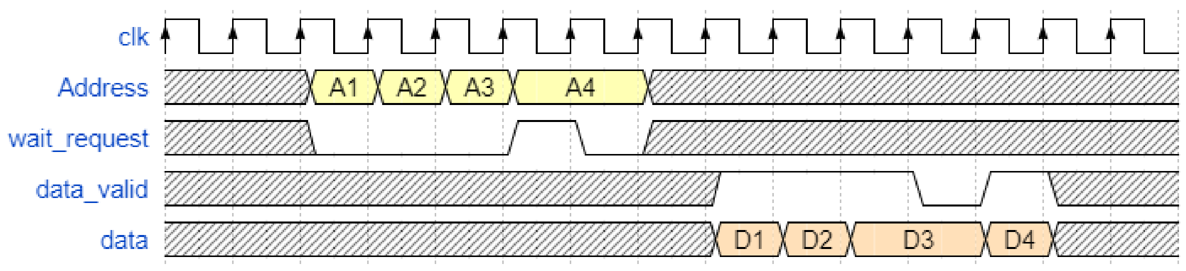


Figure 1: Timing diagram of SDRAM

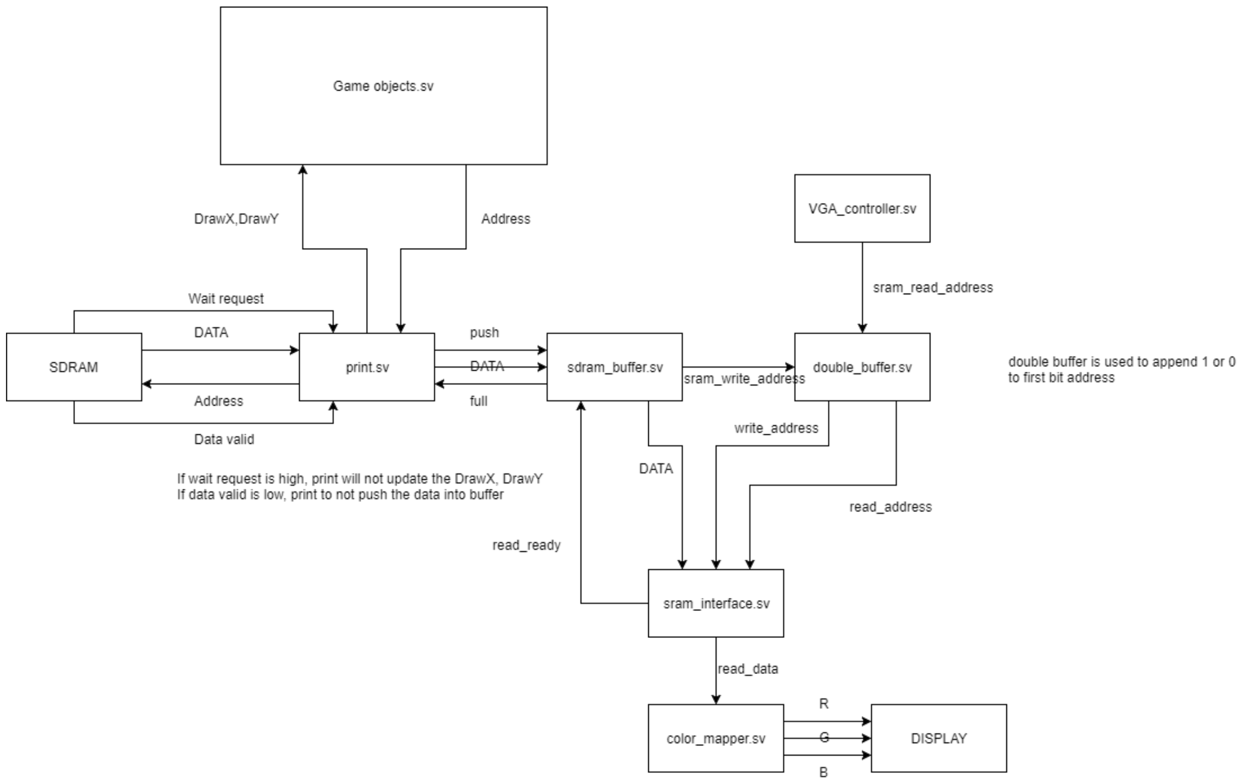


Figure2: Block Diagram for display logic

Written Description of Audio Logic

We used the 8MB Flash storage to store our background music. We have saved three different musics for different stages of the game. We play the same music to left and right channel. We used the default audio driver provided in the ECE 385 website, and played the music at 48 kHz. Since the Flash memory supports 90 ns maximum delay for reading, we used a 6.25 MHz clock to drive the Flash interface. So between each read of 8-bits data, we will wait for 160 ns to make sure that the data is indeed valid. Since the audio needs 16-bits signed values from the Flash. We had a state machine that can read consecutive two memory addresses from Flash to provide Audio Driver 16-bits data.

Different music for different stages of the game is controlled by a stage signal. Inside the Flash controller, once it detects the change of stage, it will change its address to the initial address of where the music of the new stage starts. Otherwise, the Flash controller will keep incrementing the reading address whenever the audio driver sends a dataover signal saying that new data should be sent. When Flash controller has played to the end of the music before game stage changes, Flash controller will loop back to the initial address of the music at this stage except for the dead music, because it does not sound very comfortable.

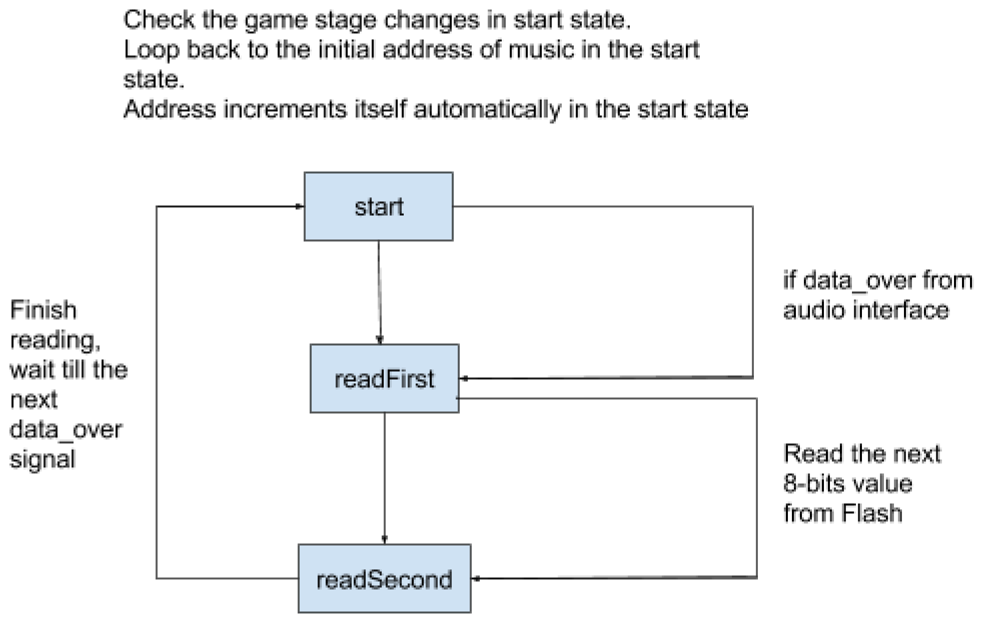

Check the game stage changes in start state.
Loop back to the initial address of music in the start state.
Address increments itself automatically in the start state
start
readFirst
readSecond
if data_over from audio interface
Read the next 8-bits value from Flash
Finish reading, wait till the next data_over signal

Figure 1: state machine of the Flash controller to play music

Written Description of Keyboard

We used the Keyboard setup C code from lab 8. Since our game requires multi-keys recognition. We took the changes taught in kttech (https://kttechnology.wordpress.com/2017/04/26/ece-385-final-project-notes-reading-multiple-4-keycode-at-the-same-time/). Due to the 16-bits limit of EZ-OTG, and each keycode is 8-bits wide, each time we can read at max 16-bits data. We accepts reading of four keypress at the same time. First we changed the PIO of keycode to 32-bits output. In the C code, we read from OTG Ram 0x051E for the last 16-bits data and then from OTG Ram 0x0520 for the first 16-bits data. Then we and the 32-bits data with 0xFFFFFFF and send the result to keycode.

In the SystemVerilog, we took the method taught in kttech. For keypress W, for example, we check the 4 bytes of the keycode data to see if any of the four bytes equals to 8'd44. We do this for all the available keypresses, including W, A, S, D, R, Enter, space and Esc.

Written Description of Game Logic

Kid: This is the character the player controls. This character includes the direction and different movements. Different direction and different movements correspond to different images in the SDRAM to be drawn in the screen. We used a state machine to control different states of the character. The player used A and D to control the character to move left and right. The player can press space to jump. The kid can jump twice in the air. The player can press Enter to shoot, one bullet at a time. The bullet is shot in the direction the Kid is facing.

Module: Character.sv
Inputs: frame_clk, Reset_h,
[9:0] DrawX, DrawY,
hit_y, hit_top, Ground, collide, W, A, D,
[9:0] Kid_position_Y, Kid_position_Y_top
Outputs: direction,
[24:0] Kid_address,
isKid,
[9:0] Position_X, Position_Y
Descritption: This is used to determine the current state of the character and calculates how to draw the character. Also from the inputs signals, the module will determine the new positions of the character to be drawn in the screen.
Purpose: Draw the character and interact with other modules for game logic.

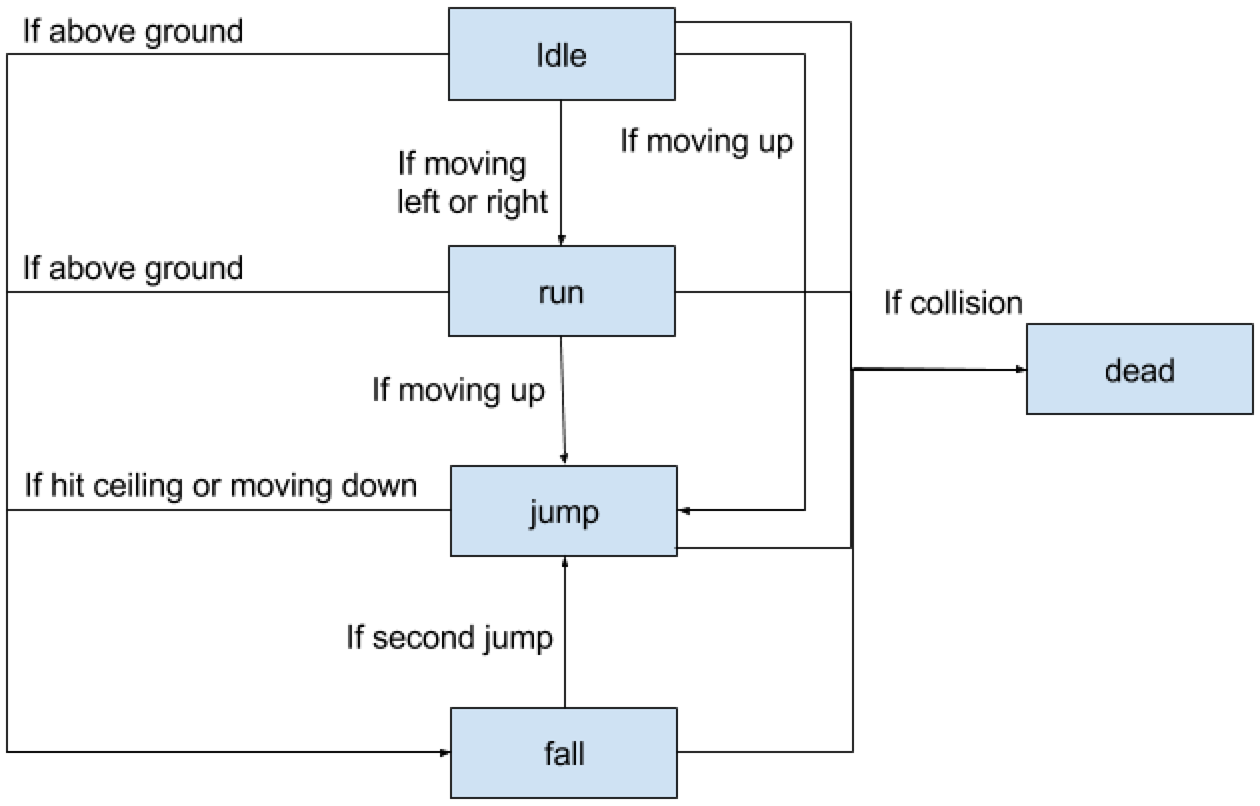


Figure 2: state machine in the character

Spur: This is the Spur attached to the floor. We have fourteen spurs, including thirteen facing-up spurs and one facing-down spur. When the Kid touches the Spur, the Spur will send a hitSpur signal to tell the Kid module that there is a collision and Kid should die. Besides those un-moving spurs, we have moving spurs that move in designed patterns when Kid move to certain trigger positions. Since the image for the spur is rectangular with the center triangle being spur and the reset all white, we improves the collision condition by adding some points along the skewed edges of the triangle. So when the kid hits one of these collision points, hitSpur signal will be sent. We built different state machine to control the movements of the spurs.

```
Module: allSpur.sv
Inputs: Reset_h, frame_clk,
        [9:0] DrawX, DrawY,
        [9:0] Kid_position_X, Kid_position_Y,
Outputs:  [24:0] Spur_address,
```

isSpur, hitSpur
Description: This module integrates all the spurs drawing logic, controls their movements. This module outputs the presence signal to the main controller and outputs the correct SDRAM address corresponding to the color of the pixel to the address selector. This module also detects whether there is a hit between the character and the spur.
Purpose: This is used for controlling all the spurs in the first level of the game.

Apple: This is the apple that is stuck inside the wall and completely harmless to the character before the character move to specific trigger positions. When the Kid move to specific trigger positions, the apple will either fall or flies to attack the Kid. We have included seven apples for three different traps. For the first trap, the apple will fall to attack the character when the character is moving close to the apple. For the second trap, the four apples will flies to the Kid when the Kid is in a specific position. This trap can be avoided by standing in a specific position what apple won't fly to. For the third trap, the apple will fall to the Kid when the Kid is standing close to the apple. The fell apple will rise up again when the Kid hits another trigger position. Since the image for the apple is rectangular, to improve the drawing, we only draw the circle part of the apple. We also improve the collision condition to be when Kid hits in the circle of the apple.

Module: allApple.sv
Inputs: frame_clk, Reset_h,
[9:0] DrawX, DrawY,
[9:0] Kid_position_X, Kid_position_Y,
Outputs: isApple, hitApple,
[9:0] Apple_address
Description: This module contains state machine that perform several traps to hit the character when the character walks to some trigger conditions. This modules calculates the distance between the character and each of the apples to determine whether there is a collision between the apple and the character.
Purpose: This module groups all the apples submodules together and control the behaviors of these apples in the first level of the game.

Wall: This is the floor. This module will checks whether the character's position is within bounds. The Wall module will send signals to the Kid indicating whether the character is standing in the ground and not hitting any ceilings. When the Ground signal sent by wall goes to zero, the character will changes to fall state except when the character is still in jump state.
Because our falling state is influenced by acceleration, it is possible for the char to falling through the wall. So our wall will determine whether the character is falling

through or jump through the ceiling by checking the current and future position of character. Upon touchdown of the character, the Wall will send to the character the final Y position to make sure the character do not end up leaking one or more pixels in the floor. It is the same for the ceiling checking. When the Kid is jumping too high that it hits the bottom of the wall, the Wall module will send us a hitY signal to force the character to go to fall state, and set the Y position of the character to be right below the bottom of the wall.

Module: Wall_set.sv and Wall_set2.sv Inputs: [9:0] DrawX_write, DrawY_write, [9:0] Kid_position_X, Kid_position_Y, [9:0] Kid_move_X, Kid_move_Y, Outputs: isWall, hit_y, hit_top, Ground, [9:0] Kid_position_Y_top, Kid_position_Y_hit, [24:0] Wall_address Description: This is the wallset that represents three walls in the first level of the game. This module compute the correct SDRAM address from DrawX_write and DrawY_write. It also takes the positions of Kid to tell the Kid module whether it has hit the wall. Purpose: This is for drawing the walls in the first level of the game. This also helps to do the edge checking for the character.

Bullet: Our design of bullet is a simple module of 2 * 2 square. It has two states: ready and flying. It has a input to record the position of character so it will start to shoot out from the current position of character. When we hit the enter button, it will change to flying speed and it will constantly increase its x position and it will go back to ready state when it fly out of the screen boundary or hit some valid object. Once it hits the valid object, the object will generate a hit signal to tell the bullet module it should stops flying, so the bullet will disappear and we can press enter to shoot the second bullet.

Module: bullet.sv Inputs: frame_clk, Reset_h, shoot, direction, hit, [9:0] PositionX, PositionY, [9:0] DrawX_write, DrawY_write, Outputs: [9:0] Bullet_X, Bullet_Y, isBullet, [24:0] Bullet_address Description: This module takes in the position of the character to be the initial position of the bullet. The direction of the character sets the direction of the bullet. The bullet is a small 2 by 2 pixels rectangle. The module will send the presence isBullet and the correct SDRAM address

to the address selector to draw the bullet.
Purpose: This module draws the bullet and controls the movements of the bullet.

RedBomb: This is the RedBomb module that draws a death light from the center of the boss to the character. The death light will die out after 3 seconds. At the end of the 3 seconds, the character will be actually hit if the character is still within some ranges of the initial position when the death light is first activated. Redbomb module will send noBomb signal to the Boss to indicate the end of attacks death light so that Boss can continue to do other attacks. For drawing skewed line in the screen connecting the character and the boss, we used a modification of similar triangles formula to draw the skewed line. Since the pixels in the screen are integer coordinates, we allow the difference between the ratios to be within the absolute value of the maximum X dimensional or Y dimensional differences between the character and the boss.

Module: RedBomb.sv
Inputs: frame_clk,
    [9:0] DrawX, DrawY,
    shoot, Reset_h,
    [9:0] Kid_position_X, Kid_position_Y,
    [9:0] Boss_position_X, Boss_position_Y.
Outputs: isRedBomb, hitBomb, NoBomb,
    [24:0] Bomb_address
Description: This modules takes in the position of the kid and the position of the boss and calculate the skewed line that should be drawn connecting these two to represent the death light or laser. It also calculates the distance between the current position of the Kid and the previous position of the kid to determine if there is a collision.
Purpose: This module will communicate with other modules to draw the skewed line of laser that could kill the character. This module also tells the boss when the death light or laser attack is over that the boss could enter the next state.

allBall: This is the allBall module that draws five balls in the screen flying with fixed velocities and directions. We took the ball.sv module from lab 8 and made some modifications. We have deleted the edge conditions checking so that the ball can fly out of the boundary and then fly back. This change is to add some uncertainties to the ball’s movements. These five balls all start from the center of the boss. Depending on the relative position of the character to the boss, the balls can either fly to the upper left or the upper right of the screen. Every ball has the character’s positions. So every ball will check for themselves whether there is a collision between the balls and the character. Balls last for 3 seconds before they disappear.

Module: allBall.sv
Inputs: frame_clk, Reset_h, moreBall,
    [9:0] DrawX, DrawY,
    [9:0] Boss_position_X, Boss_position_Y,
    [9:0] Kid_position_X, Kid_position_Y,
Outputs: isBall, hitBall,
    [24:0] Ball_address
Description: This module sets the position of the boss to be the initial positions of five balls. Based on the relative positions of the boss and the kid, this module will determine the moving directions of these five balls.
Purpose: This module is used for drawing the balls that can kill Kid. This module tells the Kid module when there is a collision by setting hitBall high.

Boss: The boss is in the second level of the game. The boss is a 64 by 64 pixels circular Huaji emoji. The character will be killed when the character hits the boss. We used the circle around the boss as the collision condition. It will make attacks to the character when the character enter some trigger conditions. We build a state machine to control the behaviors of the boss. When the boss is hit by the Kid's bullet, the boss will first fall to the ground if it is in the air, and then enter the wudi state, where the boss blinks and cannot be hurt again. In the wudi state, the boss will move slowly towards the character only in the X dimension. The wudi states last for 3 seconds. After 3 seconds, the boss is back to normal state and ready to make attacks. There are two levels of the wall in the second level of the game, when the boss found the character's Y position smaller than a trigger height, the boss will rise to the same height of the character and make attacks. When the character is within a certain range of the Boss, if the Boss is not inside any other attacking states, the Boss will expand itself to 128 by 128 pixels. When the Kid's Y position is smaller than the other trigger height, the Boss will send the death light and death balls to the character. Inside this state, the Boss is idle and the death light and death balls will be explained below. Boss module will send hitBoss signal to indicate that the Kid is killed. Boss have bullet's position. It will check for itself whether it has hit a bullet. Boss sends Boss_dead signal to the outside game stages controller to indicate that the character has passed the second level.

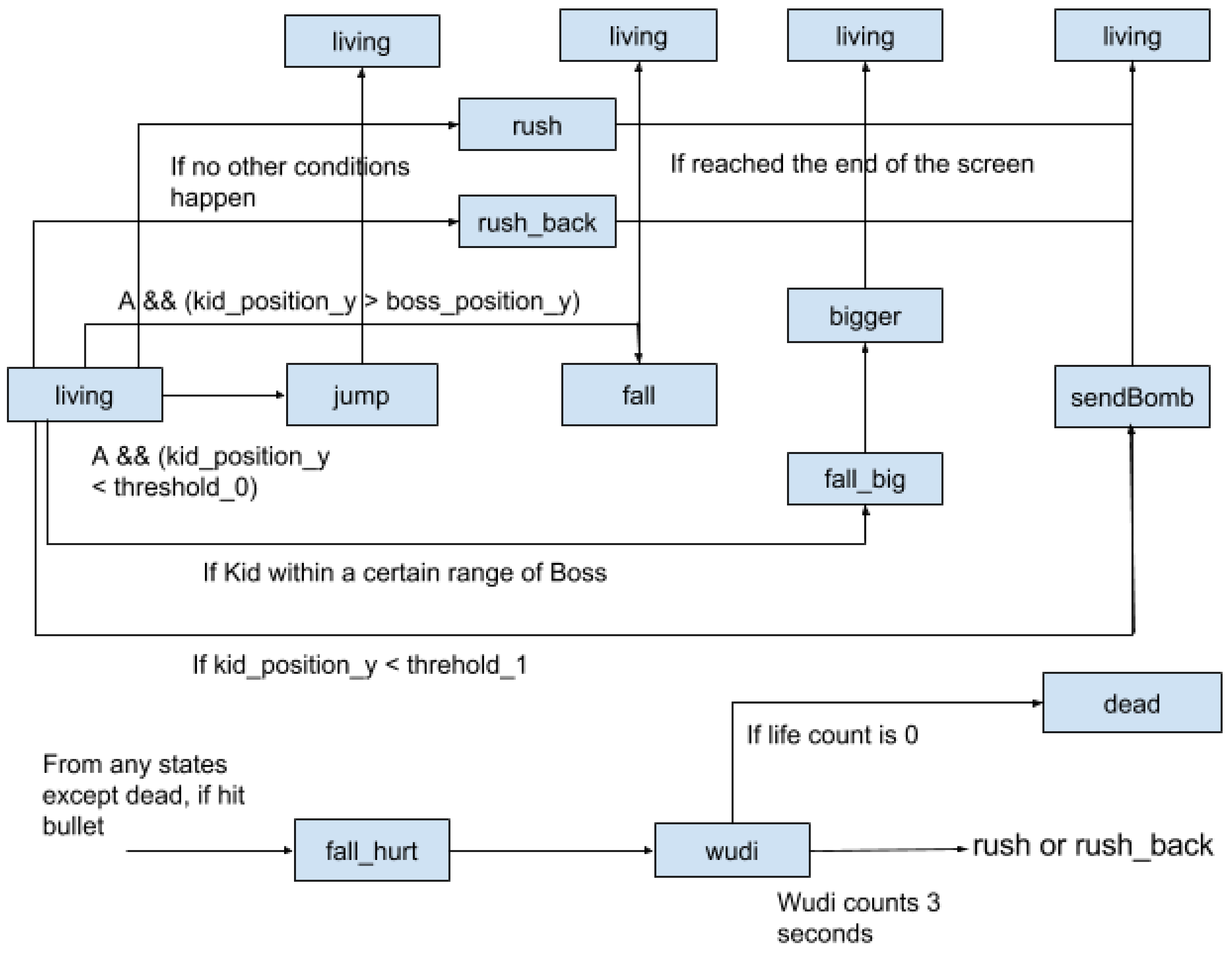

living
living
living
living
rush
If no other conditions happen
If reached the end of the screen
rush_back
A && (kid_position_y > boss_position_y)
bigger
living
jump
fall
sendBomb
A && (kid_position_y < threshold_0)
fall_big
If Kid within a certain range of Boss
If kid_position_y < threhold_1
dead
If life count is 0
From any states except dead, if hit bullet
fall_hurt
wudi
rush or rush_back
Wudi counts 3 seconds

Figure 3: state diagram for the Boss(condition A: if not hitbullet)

Module: Boss.sv
Inputs: frame_clk, Reset_h, NoBomb
[9:0] DrawX, DrawY,
[9:0] bullet_X, bullet_Y,
[9:0] Kid_position_X, Kid_position_Y
Outputs: isBoss, hitBoss, shoot, Boss_dead,
[24:0] Boss_address,
[9:0] Boss_position_X, Boss_position_Y
Description: This module takes in the position of the position of the Kid to make the boss to do certain attacks. This module calculates the distance between the bullet and the boss to determine if the boss is hit. This module contains the state machine to determine if the boss is dead and the game should enter the next stage.
Purpose: This module controls the behaviors of the boss. Send signals to the outer main control unit to control whether to change states. Outputs the position of the Boss for the RedBomb

module and the allBall module.

Stages: stages module groups the content of the game into separate stages. Because each stage has different content, we initialize different modules at different stages to construct our game. We have the main game stages, title, dead stage and final grade stage. They will have input signal stage select to choose which stage to be active and they will output address signals and all the address signals will be ORed together so when one stage is inactive, its output will be all zero. We do this because we want to make the delay as small as possible.

Module: stage0.sv
Inputs: [3:0] stages,
W, S, Reset_h, enter, frame_clk,
[9:0] DrawX_write, DrawY_write,
Outputs: confirmed,
[3:0] selected_stage,
[24:0] Address
Description: This module contains the state machine to let user to choose what three choices of the way to start the game. When the player pressed enter, the confirmed signal to be sent and the selected stage will then be displayed in the screen. The module allows the user to choose stages using W, S, and enter keys.
Purpose: This module displays the welcome page of the game. This directs the player to different pages of the game, including start, load, and exit. Start will direct player to the first level and erased the saved point. Load will direct people to the saved level of the game. Exit will direct people to the reset page of the game.

Module: stage1.sv
Inputs: [3:0] stage,
W, A, D, shoot, Reset_h, frame_clk, clk_50_shift, test_mode,
[9:0] DrawX_write, DrawY_write
Outputs: saved, reach_final, death,
[24:0] Address
Description: This module takes in the signal for controlling the character. This module outputs the signals to the outside main control logic to determine if to change and to change to which different stage. It outputs the reach_final signal to the outside main control logic to indicate the change to second level of the game. It outputs the saved signal to indicate that the progress in the first level of the game has been saved.
Purpose: This module groups all the components in the first level of the game together, including: spurs, apples, walls, teleport, and cat.

Module: stage2.sv
Inputs: [3:0] stages,
    [9:0] DrawX_write, DrawY_write,
    W, A, D, Reset_h, test_mode, shoot, frame_clk, clk_50_shift,
Outputs: [24:0] Address,
    saved, death, victory
Description: This module allows the player to control using W, A, D, and shoot. This module send saved to the outside major control logic to save the progress in the second level of the game. When there is a victory, the stage will be directed to the final_grade page.
Purpose: This module groups all the components for the second level of the game, including Boss, Wall, Kid, Bullet, RedBomb, and allBall. It will indicate the victory of the player to be directed to the final_graad page.

Module: stage_exit.sv
Inputs: [3:0] stages,
    [9:0] DrawX_write, DrawY_write,
    clk_50_shift,
Outputs: [24:0] Address
Descriptions: This module only contains module that will draw the exit image to the screen. This module will calculate the SDRAM address from the DrawX_write and DrawY_write data.
Purpose: This module is used to draw the exit image to the screen.

Module: stage_death.sv
Inputs: [9:0] DrawX_write, DrawY_write,
    [3:0] stages,
    Reset_h, R, collide, frame_clk, clk_50_shift,
Outputs: [24:0] Address
Descriptions: This module contains the print_score module that will count the number of death the player has made. This module will then calculate the SDRAM address to be displayed using the DrawX_write and DrawY_write.
Purpose: This module displays the GAMEOVER image and shows the number of death you have made.

Module: stage_final.sv
Inputs: [9:0] DrawX_write, DrawY_write,
    [3:0] stages,
    Reset_h, frame_clk, clk_50_shift, enter,
Outputs: finish_game,
    [24:0] Address
Description: This module contains the state machine that will loop through the displays images

of different letters from A to F and the image of the Final Grade. The player can press enter to select one of these letters as the player's final grade.
Purpose: This module allows the player to select one of the five letters to be the final grade. If the player press the enter again, the module will send finish_game to the outside major control logic to change to welcome page of the game.

Cat: This Cat module is hidden in the first level of the game, It will load a 320 by 131 pixels cat image. In the first level, the cat will show up suddenly and keep moving to the left end of the screen, while the character is moving to the right. The character is killed when it hits the cat. The cat could be killed when a bullet hits the cat. Like Boss module, Cat module receives bullet's positions for checking whether the Cat is dead now. Cat module also receives the Kid's position for checking whether the character has hit the cat. Cat module sends the signal hitCat to indicate the character is killed.

Module: cat.sv
Inputs: frame_clk, Reset_h,
[9:0] DrawX, DrawY,
[9:0] Bullet_position_X, Bullet_position_Y,
[9:0] Kid_position_X, Kid_position_Y,
Outputs: isCat, hitCat, hitBullet,
[24:0] Cat_address
Description: This module takes in the position of the kid and the bullet. When the position of kid enters a certain range of trigger position, the cat will show up and move to the left. When the bullet collide with the position of the cat, the cat is killed. This module calculates the correct SDRAM address of the cat. When the cat is hit, the hitBullet signal will be raised high.
Purpose: This controls the movements of the cat and draw the cat in the screen.

teleport: This module is in the end of the first level of the game. This is a 32 by 32 pixels image. The player will need to jump to reach the teleport image to "teleport" to the second level of the game. This module check the position of the character and the position of the teleport. If there is an overlap between the two objects, we say that there is a collision and the player will be guided to the second level of the game.

Module: teleport.sv
Inputs: Reset_h, frame_clk,
[9:0] DrawX_write, DrawY_write,
[9:0] Kid_position_X, Kid_position_Y,
[9:0] PositionX, PositionY,

Outputs: isTeleport, reach_final,
[24:0] Teleport_address,
Description: This module calculates the SDRAM address of the teleport. It takes PositionX and PositionY as the points of the teleport. It will detect if the character has reached the teleport. The reach_final will be raised high if the character made to the teleport and then the game will be moved to the second level.
Purpose: This module draws the teleport image in the screen. It is used for sending signals to the outside stage controller to move the stage of the game to the second level.

save: This module draws a 32 by 32 pixels savepoint box. The player can shoot the savepoint box to save the current progress in the game as in which of the two levels of the game. For example, if the player shoot the savepoint in the second level of the game, the next time in the welcome page if the player selects load, the player will be directly sent to the second level of the game.

Module: save.sv
Inputs: frame_clk, Reset_h,
[9:0] DrawX_write, DrawY_write,
[9:0] Bullet_X, Bullet_Y,
[9:0] Position_X, Position_Y,
Outputs: isSave, saved, hit,
[24:0] Save_address
Description: This module draws the 32 by 32 pixels savepoint image in the screen. It contains a state machine that can detect the bullet hit to the savepoint as signal saved to save the current progress of the player. It also outputs a hit signal to the bullet to indicate that the bullet has hit something and should disappear then. The savepoint is drawn from the position specified in PositionX and PositionY.
Purpose: This module draws the savepoint in the screen and enables the player to save the current progress of the game.

getKeycode: This modules take in the 32-bits keycode from the PIO in Qsys and translate it to different keypresses, including W, A, S, D, R, space, Enter, and Esc.

Module: getKeycode.sv
Inputs: Clock_50, Reset_h,
[31:0] keycode,
Outputs: W, A, S, D, R, space, Enter, Esc
Description: see above
Purpose: since the the 8-bits scan code can be in any of the four bytes of the 32-bits keycode. We check every byte of the 32-bits keycode to see if a signal is present.

Processor: The processor module contains our core state machine for the entire game, it is used for select different stages. It will generate the corresponding stage select signal and BGM select signal for each stage. The transition between each stage is based on keyboard input and game stage's output.

Module: Processor.sv
Inputs: CLOCK_50, OTG_INT,
[3:0] KEY,
[17:0] SW,
FL_RY,
AUD_BCLK, AUD_ADCDAT, AUD_DACLRCK, AUD_ADCLRCK,
Outputs: [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7,
[7:0] VGA_R, VGA_G, VGA_B,
VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS,
[1:0] OTG_ADDR,
OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N,
[12:0] DRAM_ADDR,
[1:0] DRAM_BA,
[3:0] DRAM_DQM,
DRAM_RAS_N, DRAM_CAS_N, DRAM_CKE, DRAM_WE_N, DRAM_CS_N,
DRAM_CLK,
[19:0] SRAM_ADDR,
SRAM_CE_N, SRAM_LB_N, SRAM_OE_N, SRAM_UB_N, SRAM_WE_N,
[22:0] FL_ADDR, FL_CE_N,
FL_OE_N, FL_RST_N, FL_WE_N, FL_WP_N,
AUD_XCK, AUD_DACDAT, I2C_SDAT, I2C_SCLK,
Inouts: [15:0] OTG_DATA,
[31:0] DRAM_DQ,
[15:0] SRAM_DQ,
[7:0] FL_DQ
Description: This is the top level of the game that groups all the stages and include the major state machine that control the switches between different states. This module includes SOC, VGA_controller, ColorMapper, keyboad, Double buffer, sdram_buffer ....
Purpose: This module connects all the necessary components together to play the game.

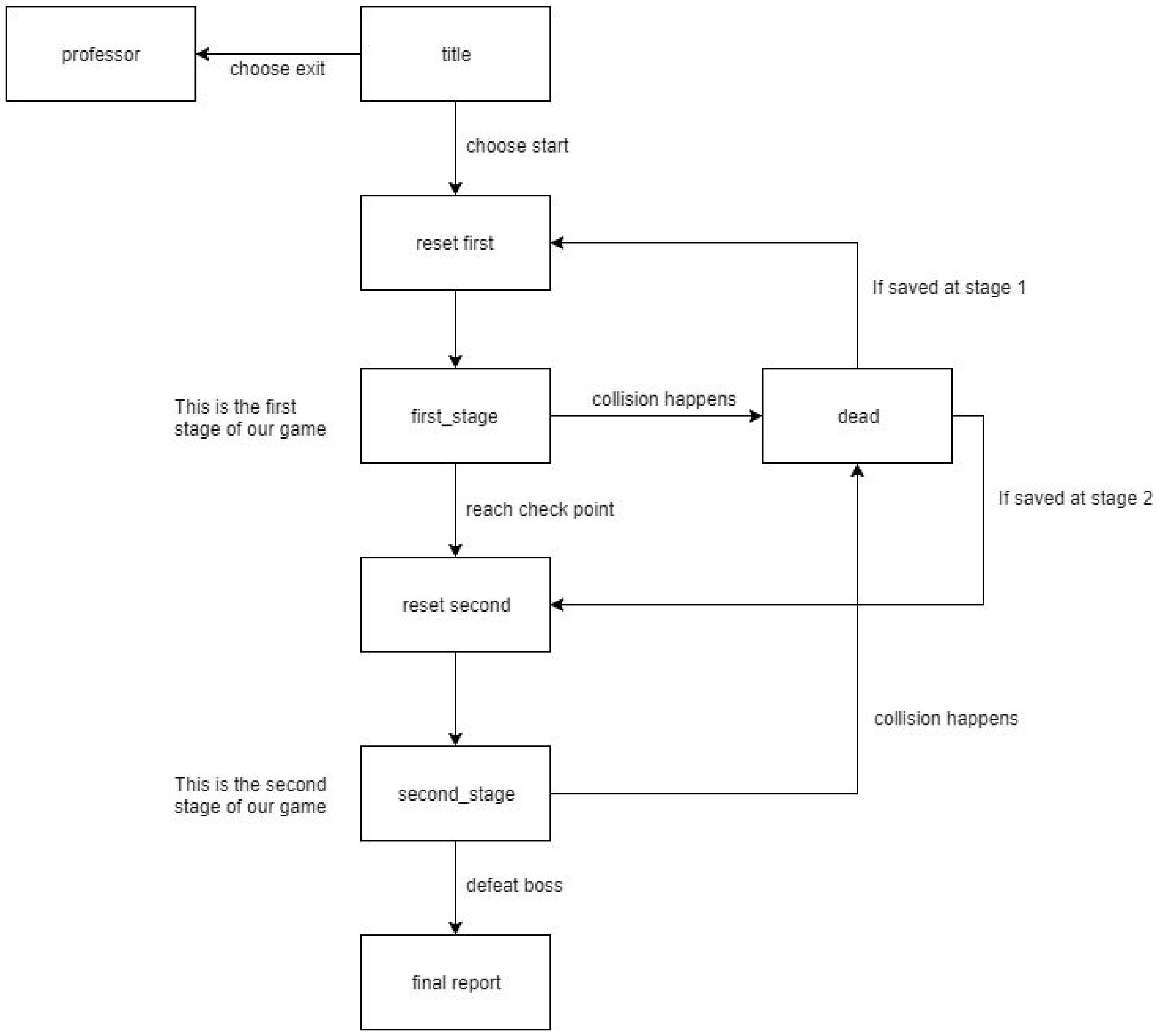


Figure 2: state machine of the game stages

Other Modules:

Module: sdram_buffer.sv
Inputs: sram_read_ready, new_frame, CLOCK_100, Reset_h, sdram_valid,
[31:0] sdram_data_in,
Outputs: [19:0] sram_write_addr,
[15:0] sram_data_out,
full
Description: This module contains the state machine that will tell the SDRAM interface when the buffer is five data before it is full. It also based on the each read of the SRAM to update the write address of the SRAM.
Purpose: This module serves as a buffer between the SDRAM and the SRAM since they have

inconsistent read and write speed.

Module: SOC.v
Inputs: clk_clk, reset_reset_n,
[24:0] sdram_control_address,
[15:0] otg_hpi_data_in_port,
[3:0] sdram_control_byteenable_n,
sdram_control_chipselect,
[31:0] sdram_control_writedata,
sdram_control_read_n, sdram_control_write_n,

Outputs: clk_25_clk, clk_50_clk, clk_50_lead_clk,
[31:0] keycode_export,
[1:0] otg_hpi_address_export,
otg_hpi_cs_export,
[15:0] otg_hpi_data_out_port,
otg_hpi_r_export, otg_hpi_w_export, sdram_clk_clk,
[31:0] sdram_control_readdata,
sdram_control_readdatavalid, sdram_control_waitrequest,
[12:0] sdram_wire_addr,
[1:0] sdram_wire_ba,
sdram_wire_cas_n,sdram_wire_cke,sdram_wire_cs_n,
[3:0] sdram_wire_dqm,
sdram_wire_ras_n,sdram_wire_we_n, slow_clk_clk
Inouts: [31:0] sdram_wire_dq
Description: This is the top level of the Nios II processor that includes all the thing we need to communicate with SDRAM, and C console and using different frequency clocks.
Purpose: This module groups all the submodules including EZ-OTG, SDRAM, NIOS II, and PLL.

Module: SRAM_interface.sv
Inputs: VGA_CLK, Clk_50, Reset_h,
[19:0] Read_ADDR, Write_ADDR,
[15:0] Data_write,
Outputs: SRAM_CE_N, SRAM_OE_N, SRAM_WE_N, SRAM_UB_N, SRAM_LB_N,
[19:0] SRAM_ADDR,
Read_ready,
[15:0] Data_read
Inouts: [15:0] SRAM_DQ
Description: This module contains the state machine that alternates between the read state and the write state at every rising edge of the 50 MHz clock. This module will change the SRAM_ADDR line to either Read_ADDR or Write_ADDR based on the current state. Inside

it contains a tristate buffer that will hold the value of SRAM_DQ.
Purpose: This is the interface for the other logics to write to the SRAM buffer and for the VGA to read from the SRAM.

Module: print.sv
Inputs: clk_50, Reset_h, new_frame, wait_request, sdram_valid, buffer_full,
    [24:0] Address,
    [31:0] sdram_data_in,
Outputs: sdram_read_n,
    [9:0] DrawX_write, DrawY_write,
    [24:0] sdram_address,
    [31:0] buffer_data,
    push, sdram_cs, sdram_read_n,
Description: This module organizes the reading of images from the SDRAM and when to push the data to the buffer. It organizes the correct increments of the DrawX_write, and DrawY_write. And it organizes the control of the SDRAM.
Purpose: This module is needed for reading the sprites from the SDRAM and sending data to the buffer whose data will be written to SRAM.

Module: print_score
Inputs: frame_clk, resur, clear_score, collide,
    [9:0] DrawX, DrawY,
Outputs: [24:0] Score_address,
    isScore
Description: This module contains the submodules that will look up the font_rom,sv data to get the 16 by 8 pixels letter and then expand the letter to 64 by 32 pixels to be displayed in the screen. This module also contain a module that will take the 32 by 32 pixels Kid facing right picture and expand it to 64 by 64 pixels image.
Purpose: This module is used for drawing the death counts and a character symbol in the gameover page of the game.

Problems and features

Double Buffer:

We decide to use double buffer because we want to make advantage of the big capacity of SDRAM, and SRAM's capacity is just large enough for double buffer. Using SDRAM as sprite storage and SRAM as double buffer is an efficient way to use memory resources.

According to our test, the SDRAM can be driven under a clock frequency of 100MHz, and it is OK for SRAM to read under frequency 100MHz. But since we want to make SRAM work as a dual port memory, we have to use half of time to write. And if we want to use SDRAM under 100MHz frequency, it is better to also drive the interface of SRAM at 100MHz to avoid synchronize problem. However, we found that if we drive the SRAM interface at 100MHz, because we are doing read and write operation alternatively, we will keep reading the data we just write in. This may because we are lack of corresponding time constraint in sdc file, and we do not have time to solve this problem. As a result, we lower our SDRAM frequency to 50MHz as well as SRAM interface.

Ideally, the valid data will be 5 cycles later after we give the correct address to SDRAM but sometimes exception may happen because of refresh and some unknown reason, this relation does not hold. But our print.sv logic is based on this assumption, so the graph print to second buffer may have offset of 1 pixel. To fixed that, we add the flag into our SDRAM and make sure that for the first line of each frame, we always read from address 0 to 639 in SDRAM. And we set the data in address 31 to xFFFF, so when the sdram_buffer try to read the data, it will recognize that the next pixel is 32th pixel in first line and compensate for the offset in the previous pixel. And we successfully stabilize our display.

The main problem of this method of double buffering is that it cannot handle the different layer of graphics. If we choose to read sprite and background alternatively, we will lose the advantage of burst mode and SDRAM cannot meet the time specification of VGA display. So when two sprite overlaps, it cannot determine how to handle the background color of sprite. I think the solution is to have another memory to read in parallel so we can have two layer of graphics.

Traps:

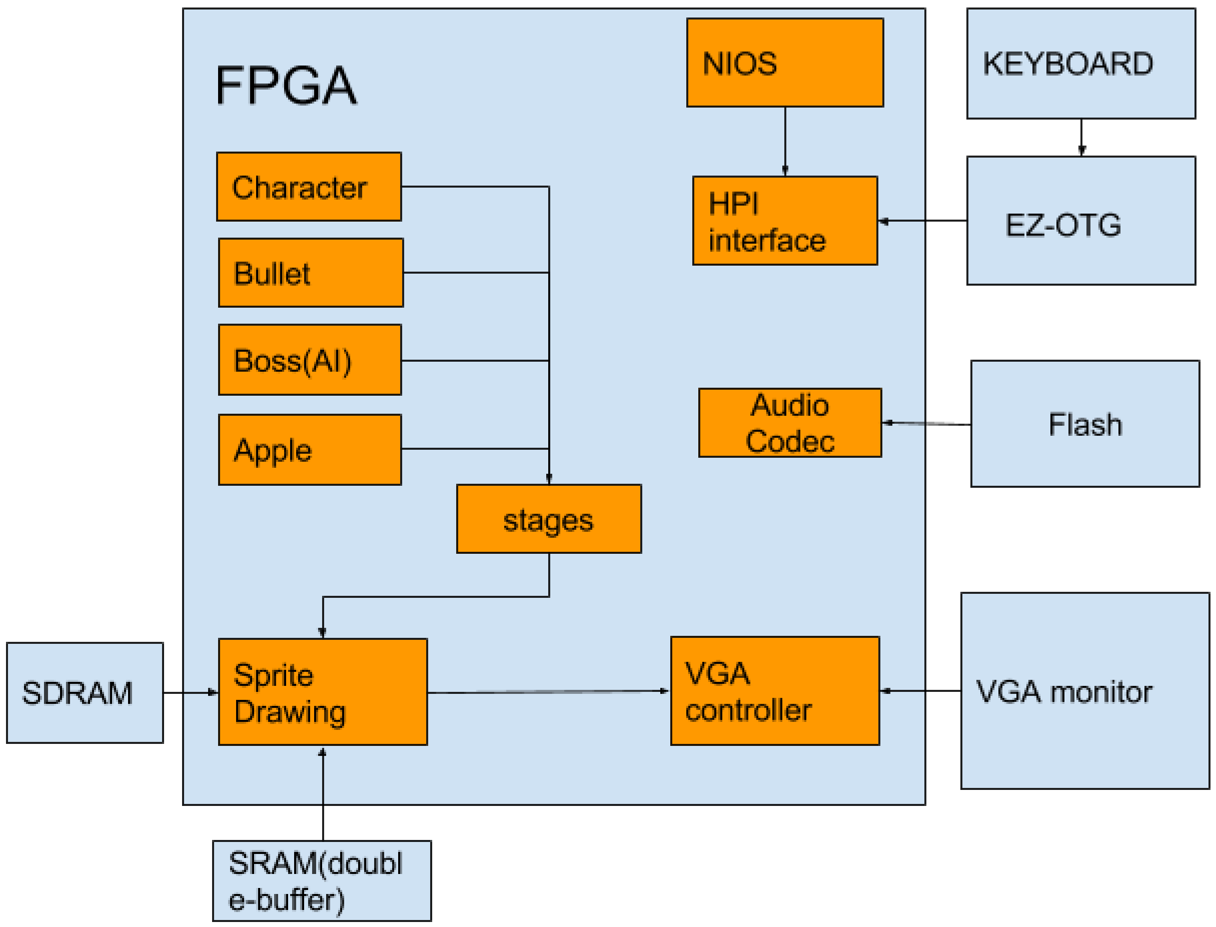
Many unexpected traps is one of feature of our game, although may be a little annoying sometimes. All of those traps are state machines. They are triggered based on the current position of our character. But it is hard to design and test those traps especially when they are built with purely hardware.

We think the main problem is our abstraction level is still too low, although we build the general objects can be set at any position. To develop more complex logic game, we should design a higher level of game event state machine and corresponding objects to coordinate the traps better.

Line Drawing:

Another feature is the laser shooted by the Boss, which is a line connect the Boss and center of our character. Though drawing a line seems to be very simple in general, it actually takes a little trick with pure hardware, which does not have floating point. For a straight line, the slope if a constant. If we set a reference point and let the x,y distance from arbitrary point to that point to be c,d. And another reference point has x,y distance a,b. We should have ac = bd. But for integer, we may not find such x when fixing y or y when fixing x. But we can get the smallest deviation by set |ac - bd| < min(a,b), and we successfully draw a straight thin line on VGA display with this logic.

Block Diagram



Design Stats

LUT	16347
DSP	0
Memory(BRAM)	2108416 bits
Flip-Flop	3555

Frequency	49.35 MHz
Static Power	107.39 mW
Dynamic Power	232.84 mW
Total Power	477.10 mW

Clock Frequency	Mostly 50 MHz
Number of Images	37
Image bit depth	8 bits(used palette to transform into 24 bits)
Background images(final grade, welcome, and exit images)	640 by 480 pixels
Game over images(and have fun image)	640 by 240 pixels
Boss image	128 by 128 pixels
Letters image	240 by 240 pixels
Cat image	320 by 131 pixels
Other images	32 by 32 pixels
Number of Songs	3
Song length	Total: 2’51’’
Audio interface working frequency	50 MHz
Audio Codec Mode	Slave mode
Sample bit length	16 bits
Sample rate	48000 Hz

Video DAC working frequency	25 MHz
Frame rate	30 Hz
SDRAM CAS latency	3
SDRAM address	25 bits
SDRAM data length	32 bits
SDRAM capacity	128 MB
SRAM frequency	50 MHz
SRAM address	20 bits
SRAM data length	16 bits
SRAM capacity	2 MB
Flash frequency	6.25 MHz
Flash address	23 bits
Flash data length	8 bits
Flash capacity	8 MB

Conclusion

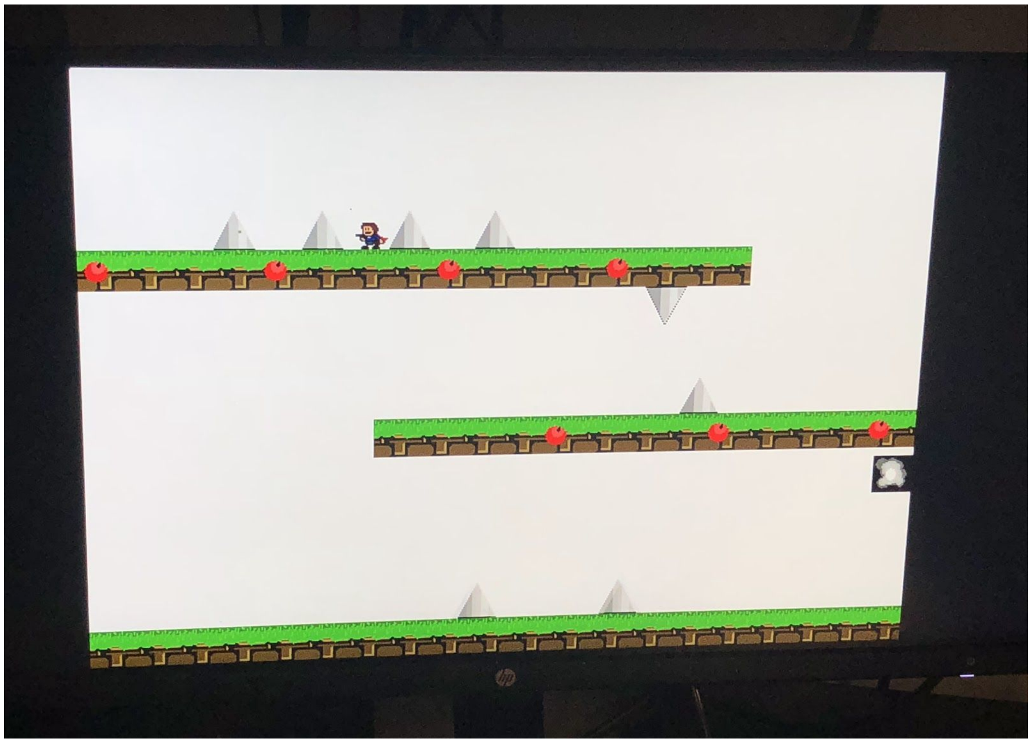
For this project, the most time consuming part is double buffer display. Although the final graphic effect may just look the same as single buffer or fixed function, we have learned a lot about SDRAM and SRAM. We have successfully to coordinate the communication between SDRAM and SRAM and eliminate the potential cause of glitch as much as possible. But we still have some wired problems, for example, even we change part of code does not influence display, the new compiled file will have a total different graphic display. We guess it is because the design is not fully constrained, and we may need to learn more about how to set time constraint for each path.

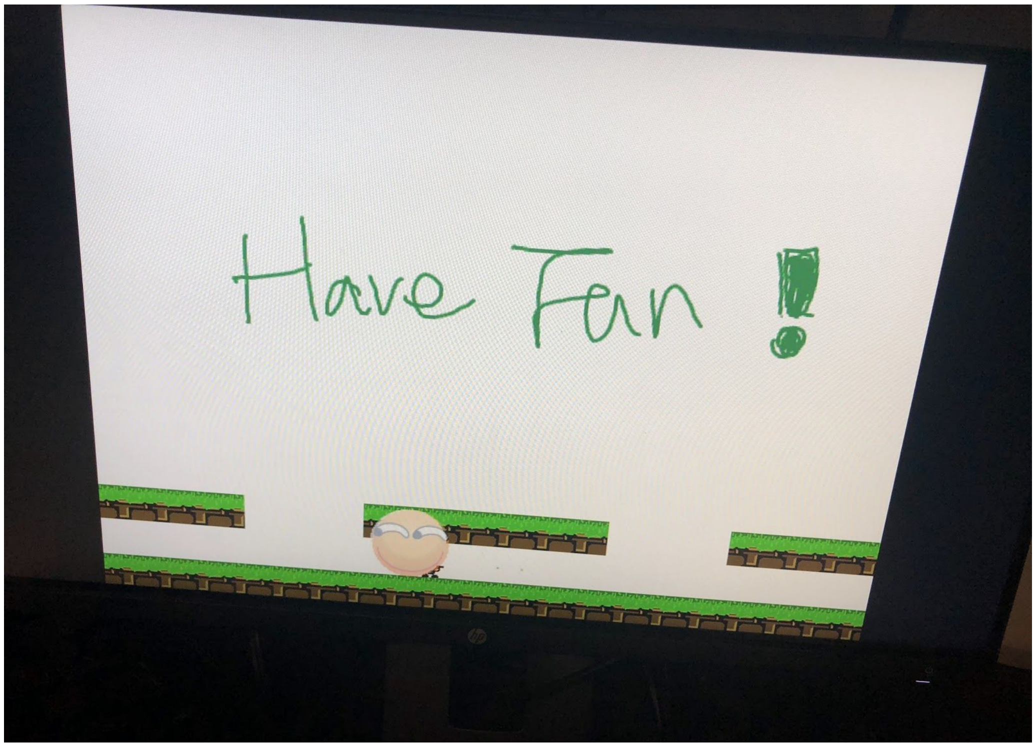
For the logic part, we do not waste too much time on debugging but we do spend a lot of time to debug the double buffer. We use Signal Tap to debug the memory operation all the time, and it is very helpful to add some test signals as indicator. We seldom make logic mistake in our state machine, and most time we found that we made the stupid wire mistake. But it is always painful to wait several minute for the code to compile and it seems that there is no better solution.

In general, our project has a large potential to be extended. We store all sprites in SDRAM, which means we can store more pictures. And we make our modules such wall and apple to be reusable: they can be set to any position just by giving them another input. This is a level of abstraction and benefits complex game design.

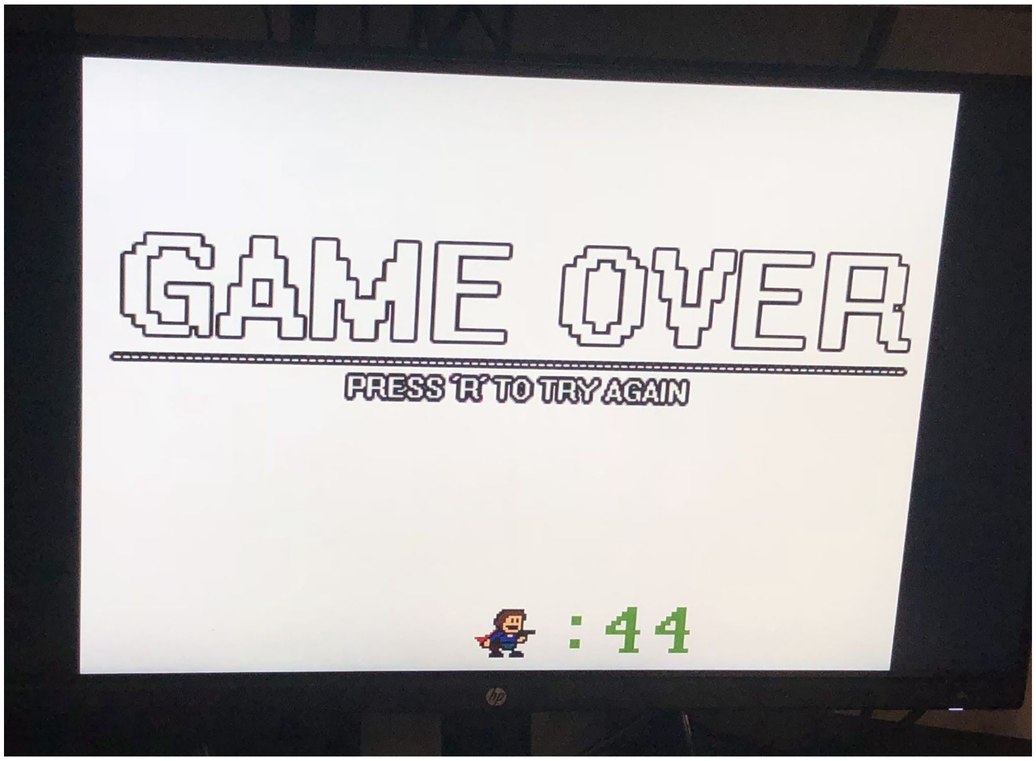
“I wanna” series game is always famous for its difficulty. And our “I wanna 385” is designed carefully to make sure it is not too hard but still interesting. Because of the time limitation, we only build 2 stages, but it includes all the core elements of this game. We can build more complex and interesting traps if time is allowed and state machine is proved to be very suitable for traps. It is a fun experience for ECE385 final project.

Appendix


Picture1: Stage 1 of the game



Picture2: Stage 2 of the game



Picture3: Game Over