ECE385

Spring 2022

Final Project

Final Project Report

Jiaru Zou, Simon Ge
Lab Section CY
TA’s Name: Curtis Yu

Introduction

For our final project, we make a Pac-Man game by implementing the FPGA board and System Verilog. In this game, the player can control the character of pacman to win the game by eating all the pac-dots without being eaten by four Ghosts.

In order to complete the game project, we first inherit files from lab 6 as our starting point. Several files in lab 6 such as ball.sv, VGA controller.sv, and the program written in C code in NIOS II can be used as a foundation for our project. After setting the basic files for the final project, we started designing the rom of maze as well as the rom of each character in order to show them on the VGA screen. Then we design the motion of pacman to allow it to move inside the maze. Additionally, we designed the motion logic of ghosts as game rules suggested and added the counter of both pacdots and pac-man lives on the screen. Finally, to increase the interest of the game, we added a new feature for the pacman which is the super pac-dot that pac-man is able to eat ghosts and add more points after eating it. We also implement the state machine to reset the game and design a new screen interface after the game is over.

**The design details of each part will be covered in later sections of this final project report.*

Game Rules and Features

The list of rules for the game is shown below:

- Users need to control the character of pac-man through the keyboard. Specifically, “W” represents mowing up, “S” represents moving down, “A” represents moving left, and “D” represents moving right. And when the game is over, players can click “Space” to restart the game.
- When the game does not start, that is when the pacman is not moving, the four ghosts are locked in the middle square which is the “cage” for them. Then the first ghost starts showing in the maze when the pacman starts moving. And when the pacman eats the pac-dots and separately reach to 25, 40, and 75 scores, the second, third, and fourth ghost come from the cage correspondingly.
- Each player gives two lives to the pac-man and to win the game: the pac-man needs to reach a score of 200, which means that it successfully eats all the pac-dots in the maze. Otherwise, if the pac-man is eaten twice by the ghost before it reaches the 200 score, the player fails. Both scores and the lives of pac-man are recorded on the screen, the scores are recorded in digital numbers and the lives of pac-man are recorded through the number of pacman diagrams in the top right of the screen. When users initially start the game, they will have two chances, represented by the pac-man diagrams. If they are eaten by ghosts, the pac-man will reset to its original position.
- Four “super pac-dots” are added to the game, separately located in the four corners of the maze with the color green. When pac-man eats the super pac-dot, all four ghosts become the color of blue, meaning ghosts can be eaten by the pac-man. And 10 points are added to the score shown in the screen for each time the pacman eats one ghost. The ghost will reshow on its original resurrection point after being eaten.

Block Diagram

This is the overall architecture of our project:

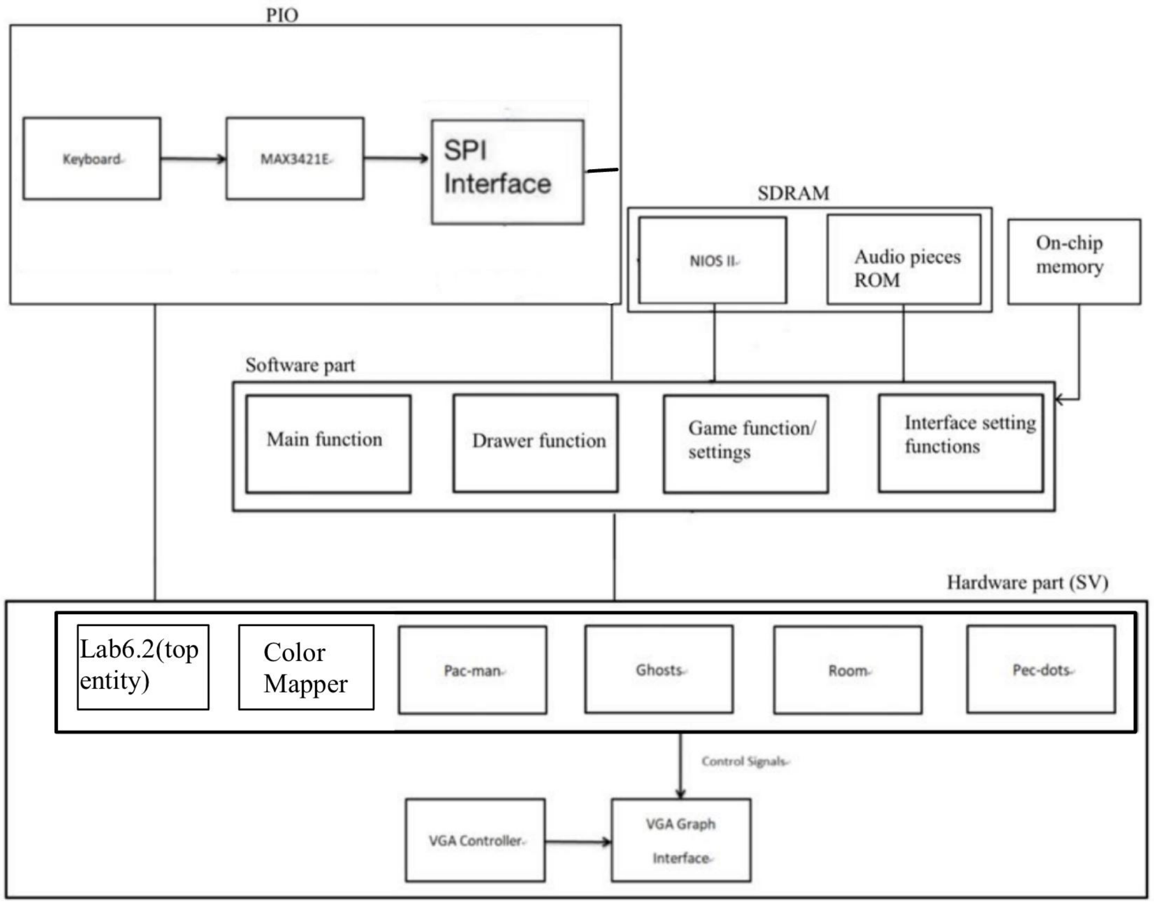


Figure one

This is the top-level diagram generated by Quartus.

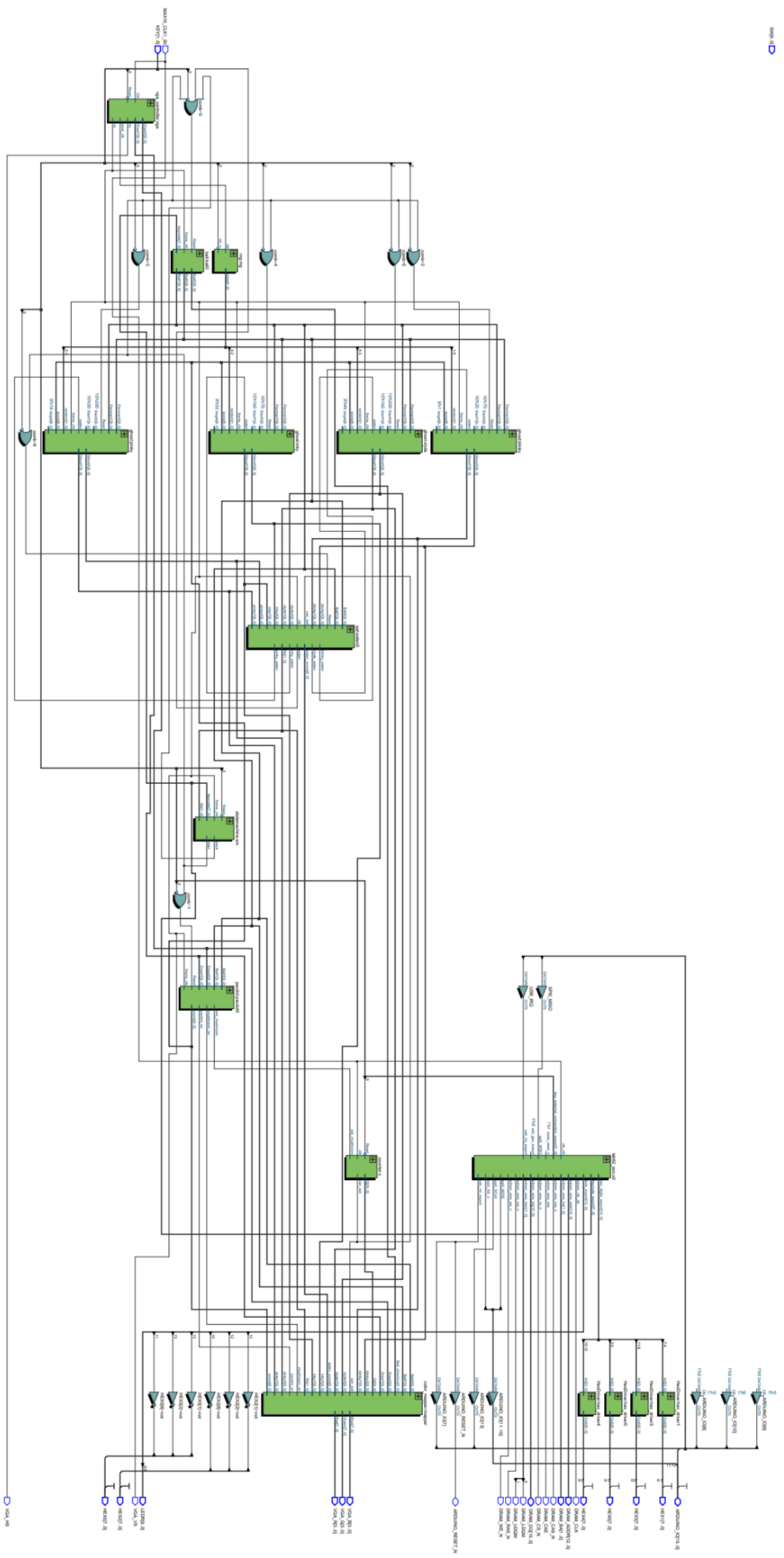


Figure two

Written Description

a.The hardware component of the final project

We design our final project mainly on NIO2 processor, memory, keys, LEDS, and a system ID checker. Specifically, the memory stores basic data and instructions required by NIO2 to run, and also stores the data from key and to LEDs. The NIO2 processor then processes the input from keys, and displays output to LEDs. Besides, there is a system ID checker that makes sure the compatibility of the hardware and the software.

b.The interaction of NIOS with both the MAX3421E USB chip and the VGA components

The NIOS interacts with MAX3421E through the SPI port. When the NIOS is writing to MAX3421E, the NIOS will first write the command byte and the data being written to the register buffer. Then, the data in register buffer will transfer the data to the slave through MOSI port. On the contrary, when NIOS is receiving data from MAX3421E, the MAX will write the data into the register buffer through MISO port, and then the NIOS is able to fetch the data from the buffer. The NIOS interacts with VGA components by generating horizontal and vertical sync pulses. The NIOS uses a counter to iterate over all the x and y coordinates, and assign the corresponding pixel value. It also receives the keycode from MAX and controls the movements of the ball in the VGA components.

c.SPI protocol.

The SPI protocol mainly consists of two parts: the command byte and the data being read or written. The command byte essentially includes the information of the register and the direction of the interaction. It tells the register where the read data is located if “dir” is 1 or the write data is located if “dir” is 0. Then, the data being transferred is followed by the command byte, and then write to the buffer, and transmitted to either the master or the slave.

d. Functions in C code description

MAXreg_wr(BYTE reg, BYTE val): The MAXreg_wr function write the value val with length 1 to the register buffer reg, and then transfer to the slave.
MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data): The MAXbytes_wr function write the value stored at data with length nbyte to the buffer, and the transfer to the slave.
MAXreg_rd(BYTE reg): The MAXreg_rd function read the value stored in the register buffer reg transferred from the slave
MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data): The MAX_bytes_rd function read the nbytes data stored in the register buffer reg transferred from the slave, and then stored it to data.

e. All feature implementation description

Background Maze

To design a maze for players to play on, we mimic a maze based on the traditional pac-man game. We have known that the VGA screen size is 640 x 480, thus we choose to make 16 pixels as a unit and divide the size of screen by 16 for hardcoding. Thus in our rom, there are total 40 units in the horizontal direction and 30 units in the vertical direction. For the paths between the maze, we design them with the size of one unit which is 16 pixels for the convenience of implementing pacman. When we draw the maze in the color mapper, we use drawX and drawY to traverse every pixel and draw the wall with the color of blue, and draw paths and other areas with the color of back. This is our foundation of the game shown on the screen.

Pacman
In order to fix the size of the path implemented in the maze, we design our pacman as a shape of square with the size of 16x16 pixels. And we hardcode four directions of pacman in the character rom. And we set the origin position of the pacman in the middle position of the maze as starting point to play. Like lab6, we let the pacman to determine the movement corresponding to the keycode. In addition, we determine the condition for pacman to stop when facing the wall. Specifically, for each side, we use two checking pixels to determine if the pacman is stoped or keep moving in one direction. If the checking point is in the wall corresponding to the pacman's moving direction, the pacman will stop. Otherwise, it will move based on the keycode. From the figure, we can notice that each checking point has one pixel distance from the edge of each size, this is because of the delay of clock cycle between the checking logic in always_ff and the movement of pacman.

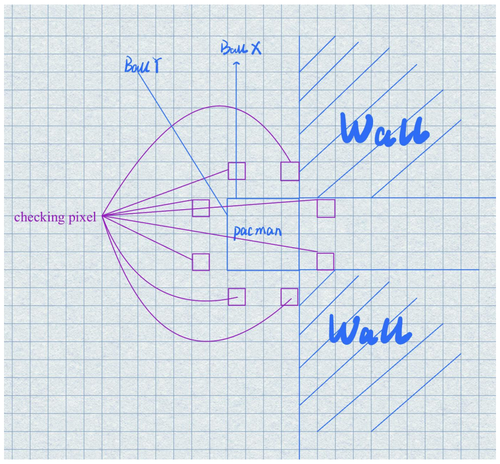


Figure three

Inside each pacman unit, we also determine the color of each pixel in the color mapper based on the hardcoding in the character rom. First there are four diagrams of pacman with mouth pointing to four directions, we use the keycode to determine which diagram to be drown. Second, inside each diagram of the rom, it is a size of square. The part of pacman is shown with 1, and the part not belonging to pacman is shown with 0. Thus when drawing in the color mapper, we draw the pacman part with the color of yellow, and the part not belonging to pacman with the color of black.

Pacdot
We use a two-dimensional array to first copy the 30x40 size of the maze background. Then we change the unit from 1 to 0 if there is a pacdot inside that unit. Then we control the shape of pacdot to a square with size 3x3. And we use the signals pacdot_on to determine when to draw the pacdot with white color in the color mapper. Also, we deal with the situation when the pacdot is eaten by pacman, that is when the middle pixel of pacman has the same position as the pacdot, the pacdot part will be replaced with the color of black. And we also output a signal score to calculate the number of pacdots being eaten by the pacman.

Special Pacdot
On the basis of Pacdot, we make some modifications to the module of Pacdot. At four corners of the maze, the pacdots are shown in red, and once the special pacdot is eaten by the pacman, a signal is outputted to color mapper and ghosts to indicate that now the pacman can eat ghosts in reverse.

Ghosts
Similar to the logic in the pacman module, the ghost module is actually a special pacman entity that moves randomly based on a random number generator. The ghosts have the same collision logics as the pacman and a new direction is moved when the pacman stops. The new direction of each ghost is determined by a 2 bits random number, which derives from XOR nearby bits of a random number. Normally, when ghosts eat the pacman, the ghosts will remain in their state of motion, whereas when the signal generated by special pacdots is on, indicating that the pacman can in turn eat ghosts, the ghosts will turn into blue. Once the ghost is eaten by the pacman, the eaten ghost will go back to its original corner, and ten points will be added to the score that is shown on the screen.

Score on screen
In order to keep the score player obtained on screen, we use three units of place to show the number of score. We first harcode the shape of number with size 16x16 according to the source from lab7 in a new rom, then draw the number with the color of white on the color mapper. The number of score is based on the signal "score" obtained from the pacdot.sv.

Life on screen
Similar to score on the screen, we pick up a spce outside of the maze to display the number of lives. There are two lives in the game corresponding to two diagrams of pacman at the beginning. And if the pacman is eaten by the ghost, the life in the eat.sv will deduct one until it reaches to zero. And the pacman diagram drawing in the color mapper will be replaced by the background black color each time pacman being eaten.

Module Descriptions

Module: lab62.sv
Inputs: MAX10_CLK1_50, [1:0] KEY, [9: 0] SW,
Outputs:[9:0] LEDR, [7: 0] HEX0, [7: 0] HEX1, [7:0] HEX2, [7:0] HEX3, [7:0] HEX4, [7:0] HEX5, DRAM_CLK, DRAM_CKE, [12:0] DRAM_ADDR, [1:0] DRAM_BA, [15:0] DRAM_DQ, DRAM_LDQM, DRAM_UDQM, DRAM_CS_N, DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N, VGA_HS, VGA_VS, [3:0] VGA_R, [3:0] VGA_G, [3:0] VGA_B, [15:0] ARDUINO_IO, ARDUINO_RESET_N
Description: This module connects the inputs and outputs of modules such as pacman, ghost, eat, and statemachine.
Purpose: This module serves as the top-entity of the final project that connects the soc to the FPGA and interacts with VGA components.

Module: lab62soc.v
Inputs: clk_clk, [1:0] key_external_connection_export, reset_reset_n, spi0_MISO, usb_gpx_export, usb_irq_export
Outputs: [15:0] hex_digits_export, [7:0] keycode_export, [13:0] leds_export, sdram_clk_clk, [12:0] sdram_wire_addr, [1:0] sdram_wire_ba, sdram_wire_cas_n, sdram_wire_cke, sdram_wire_cs_n, [15:0] sdram_wire_dq, sdram_wire_dqm, sdram_wire_ras_n, sdram_wire_we_n, spi0_MOSI, spi0_SCLK, spi0_SS_n, usb_rst_export
Description: This module consists of nios2, hex digit pio, jtag, key, keycode, led pio, on-chip memory, sdram, timer, usb_gpx modules required by the soc.
Purpose: This module is the soc for the final project generated by the Platform Designer.

Module: VGA_controller.sv

Inputs: Clk, Reset,
Outputs: hs, vs, pixel_clk, blank, sync, [9:0] DrawX, DrawY
Description: This module generates the horizontal and vertical sync pulse and iterates through all pixels.
Purpose: This module helps color mappers to determine the color of each pixel, and also provide a clock for the ball(vs).

Module: room_rom
Input: [4:0] addr
Output:[39:0] data
Description: This module contains a rom with size 30*40, which contains the maze of the pac-man game.
Purpose: We use this module as a data source to store the content of maze for our final project.

Module: HexDriver.sv
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: This module has hardcoded values that convert 4bit binary numbers from registers into hexadecimal using the unique case in the always_comb logic.
Purpose: This module helps to show the hex value of the adder's output on the LEDS when using an FPGA board.

Module:Color_mapper
Input: [9:0] BallX, BallY,DrawX, DrawY, Ball_direction, blinkyX, blinkyY, pinkyX, pinkyY, inkyX, inkyY, clydeX, clydeY; [8:0] score; [8:0] eaten_score; [1:0] life; [29:0] Q; can_eat; pacdot_on; mushroom_on;
Output:[7:0] Red, Green, Blue;
Description: This module uses logic to generate background, pacman, ghost, maze, score counter, and end scene which can show in different colors in the VGA screen.
Purpose: We use this module to determine the condition for drawing different contents with specific colors.

Module: ball.sv
Input: Reset, frame_clk; [7:0] keycode
Output: [9:0] BallX, BallY, BallD
Description: This module writes the motion of pacman based on the keycode, it also deals with the collision for each direction compared with the wall of the maze. Specifically, the pacman should stop when facing the wall of the maze.
Purpose: We create this module to design the logic of ball motion.

Module: character_rom
Input: [6:0] addr
Output: [15:0] data
Description: This module contains the hardcode of pacman in four directions(up, down, left, right). It also contains the hardcode of the ghost character.

Purpose: We use this module as the data source for implementing each character on the VGA screen.

Module: pacdot
Input: Reset, frame_clk;[9:0] BallY, BallX, DrawY, DrawX
Output: mushroom_on, pacdot_on, eat_mushroom, [8:0] score
Description: This module uses a two-dimensional array to create the pac-dots for the game. We also add conditions for checking when to create pac-dots in the maze and when the pac-dots are not shown. Specifically, there are two situations for pacdots not shown: 1. Overlapping by other contents including character, background outside the maze. 2. Being eaten by the pack-man.
Purpose: We write the part of pac-dots in this module as part of the pac-man game.

Module: rng
Input: clk, rst_n
Output: [4:0] data
Description: This module uses the two-always logic to generate the random number of data in an array and outputs the data as an array.
Purpose: This module generates the random data and later uses it for the random motion of the ghost in the game.

Module: ghost
Input: Reset, frame_clk, [1:0] random, StartX, StartY, eaten, [8:0] score, target
Output: [9:0] GhostX, GhostY
Description: This module writes the motion of ghosts in the game. In the logic, we define the location of the ghost in the maze as well as the collision of the ghost with the maze.
Purpose: This module defines the logic of the ghost motion for the pac-man game.

Module: eat
Input: Reset, clk, can_eat; [9:0] BallX, BallY, blinkyX, blinkyY, pinkyY, pinkyX, inkyX, inkyY, clydeX, clydeX
Output: eaten, [8:0] eaten_score, blinky_eaten, pinky eaten, inky_eaten, clyde_eaten, [1:0] life
Description: This module defines the situations and conditions for the ghost eating pacman as well as pacman eating ghost when eating the super pac-dot. It also includes the calculation of scores for pacman to eat ghosts and the number of lives for pacman.
Purpose: This module designs the logic for the eating function between pacman and ghost.

Module: score_rom
Input: [7:0] addr
Output: [15:0] data
Description: This module contains the hardcode of numbers from zero to nine.
Purpose: We use this module as the data source for implementing the number of scores showing on the VGA screen.

Module:statemachine
Input: Reset, frame_clk, [1:0] life, [7:0] keycode
Output: start, back

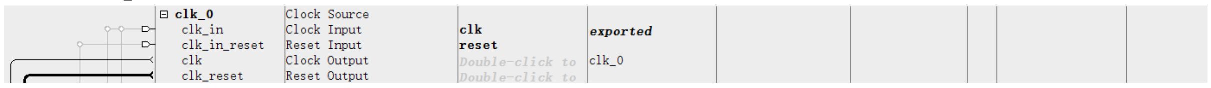
Description: This module designs the state machine for the project. It designs the reset option to restart the game.
Purpose: This module is used for resetting the game using the space keycode in the game.

Module: endscene_rom
Input: [4:0] addr
Output: [39:0] data
Description: This module contains the hardcode of the end scene, which is “GAME OVER”.
Purpose: This end scene takes place after the game is over, and the “GAME OVER” is shown on the VGA screen.

Module: counter
Input: Reset, clk, eat_mushroom
Output: can_eat, [29:0] Q
Description: This module designs a counter for the clock in the project. Since the clk in Quartus is 50 Mhz, we count the number from 0 to 400 Mhz for the purpose of counting the time since time is the reverse of rate.
Purpose: This module helps to calculate the time after pacman eats the super pac-dot, the time is about 10 seconds.

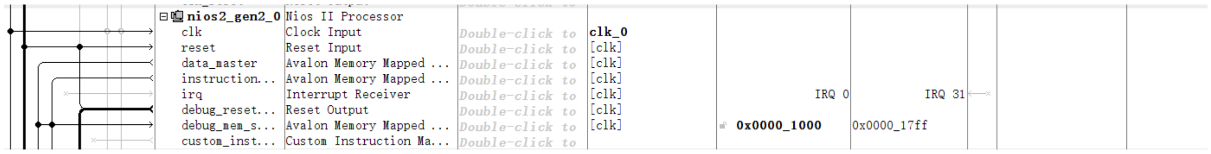
System Level Block Diagram

core components:



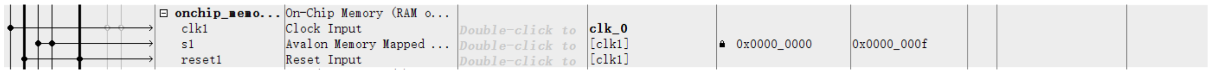
```
⊟ clk_0          Clock Source
    clk_in       Clock Input      clk               exported
    clk_in_reset Reset Input      reset
    clk          Clock Output     Double-click to   clk_0
    clk_reset    Reset Output     Double-click to
```

clk_0: The clk_0 provides a 50mhz clock to the system



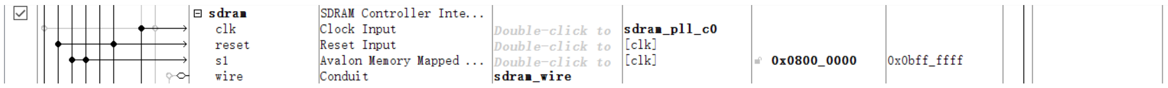
```
⊟ nios2_gen2_0    Nios II Processor
    clk           Clock Input               Double-click to  clk_0
    reset         Reset Input               Double-click to  [clk]
    data_master   Avalon Memory Mapped ...  Double-click to  [clk]
    instruction...Avalon Memory Mapped ...  Double-click to  [clk]
    irq           Interrupt Receiver        Double-click to  [clk]                 IRQ 0        IRQ 31
    debug_reset...Reset Output              Double-click to  [clk]
    debug_mem_s...Avalon Memory Mapped ...  Double-click to  [clk]    0x0000_1000  0x0000_17ff
    custom_inst...Custom Instruction Ma...  Double-click to
```

nios2_gen2_0: The nios2_gen2_0 is the processor of the system, processing instruction and data from the AVALON bus



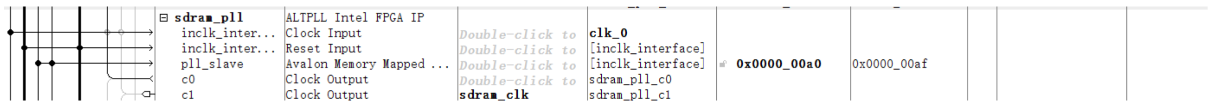
```
⊟ onchip_memo...  On-Chip Memory (RAM o...
    clk1          Clock Input               Double-click to  clk_0
    s1            Avalon Memory Mapped ...  Double-click to  [clk1]   0x0000_0000  0x0000_000f
    reset1        Reset Input               Double-click to  [clk1]
```

on-chip memory: The on-chip memory stores and access data for NIOS2 through AVALON data and instruction bus in a faster way



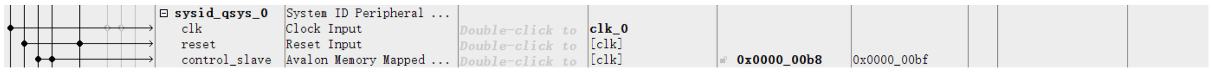
```
⊟ sdram     SDRAM Controller Inte...
    clk     Clock Input               Double-click to  sdram_pll_c0
    reset   Reset Input               Double-click to  [clk]
    s1      Avalon Memory Mapped ...  Double-click to  [clk]   0x0800_0000  0x0bff_ffff
    wire    Conduit                   sdram_wire
```

sdram: The sdram provides a larger placeholder for data storage and access for NIOS2 through AVALON data and instruction bus



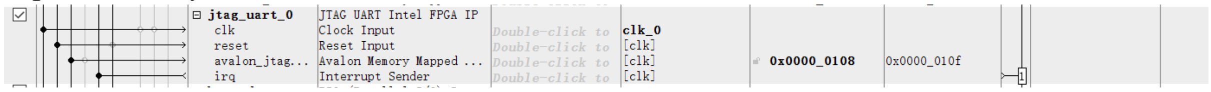
```
⊟ sdram_pll        ALTPLL Intel FPGA IP
   inclk_inter...  Clock Input             Double-click to  clk_0
   inclk_inter...  Reset Input             Double-click to  [inclk_interface]
   pll_slave       Avalon Memory Mapped ...Double-click to  [inclk_interface]  0x0000_00a0  0x0000_00af
   c0              Clock Output            Double-click to  sdram_pll_c0
   c1              Clock Output            sdram_clk        sdram_pll_c1
```

sdram_pll: The sdram reads and writes data from the sdram and transfers to corresponding peripherals. It also generates two clocks, one of which is 1ns delayed for accessing dara



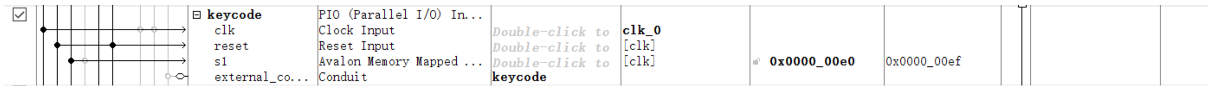
```
⊟ sysid_qsys_0     System ID Peripheral ...
   clk             Clock Input             Double-click to  clk_0
   reset           Reset Input             Double-click to  [clk]
   control_slave   Avalon Memory Mapped ...Double-click to  [clk]  0x0000_00b8  0x0000_00bf
```

sysid_qsys_0: The sysid_qsys_0 determines whether the software and the hardware are compatible to protect the system.



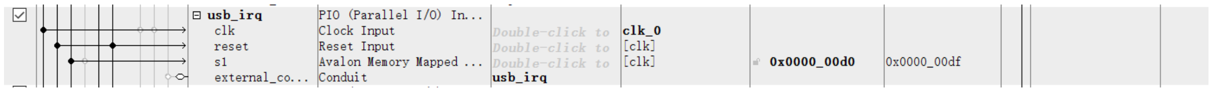
```
☑  ⊟ jtag_uart_0      JTAG UART Intel FPGA IP
      clk             Clock Input             Double-click to  clk_0
      reset           Reset Input             Double-click to  [clk]
      avalon_jtag...  Avalon Memory Mapped ...Double-click to  [clk]  0x0000_0108  0x0000_010f
      irq             Interrupt Sender        Double-click to  [clk]
```

jtag_urat_0: The JTAG URAT allows to use the terminal of the host computer to communicate with the NIOS II using scanf and printf



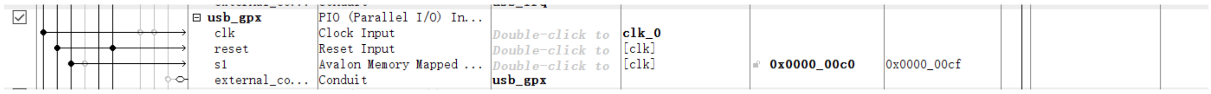
```
☑  ⊟ keycode          PIO (Parallel I/O) In...
      clk             Clock Input             Double-click to  clk_0
      reset           Reset Input             Double-click to  [clk]
      s1              Avalon Memory Mapped ...Double-click to  [clk]  0x0000_00e0  0x0000_00ef
      external_co...  Conduit                 keycode
```

keycode: The keycode is a pio that takes in the keycode pressed on the keyboard and transfer the data to the NIOS2



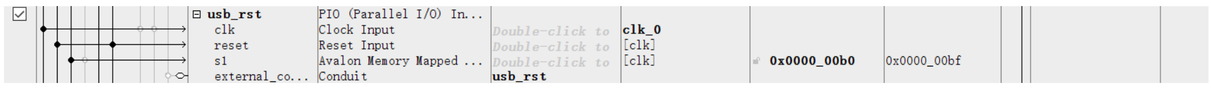
```
☑  ⊟ usb_irq          PIO (Parallel I/O) In...
      clk             Clock Input             Double-click to  clk_0
      reset           Reset Input             Double-click to  [clk]
      s1              Avalon Memory Mapped ...Double-click to  [clk]  0x0000_00d0  0x0000_00df
      external_co...  Conduit                 usb_irq
```

usb_irq: The usb_irq is a pio that takes in the interrupt request IRQ value and transfer the data to the NIOS2



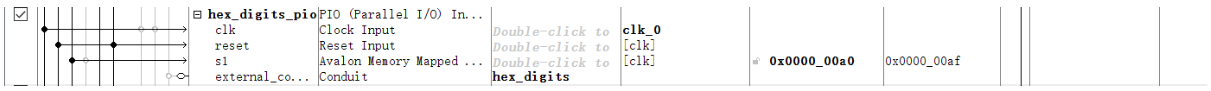
```
☑  ⊟ usb_gpx          PIO (Parallel I/O) In...
      clk             Clock Input             Double-click to  clk_0
      reset           Reset Input             Double-click to  [clk]
      s1              Avalon Memory Mapped ...Double-click to  [clk]  0x0000_00c0  0x0000_00cf
      external_co...  Conduit                 usb_gpx
```

usb_gpx: The usb_irq is a pio that takes in the GPX value and transfer the data to the NIOS2



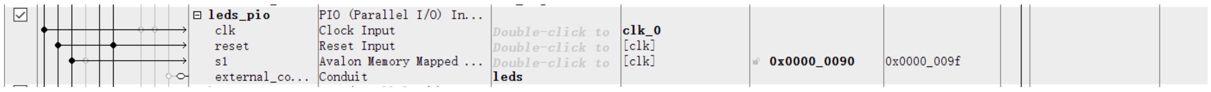
```
☑  ⊟ usb_rst          PIO (Parallel I/O) In...
      clk             Clock Input             Double-click to  clk_0
      reset           Reset Input             Double-click to  [clk]
      s1              Avalon Memory Mapped ...Double-click to  [clk]  0x0000_00b0  0x0000_00bf
      external_co...  Conduit                 usb_rst
```

usb_rst: The usb_irq is a pio that outputs the reset value and transfer the data



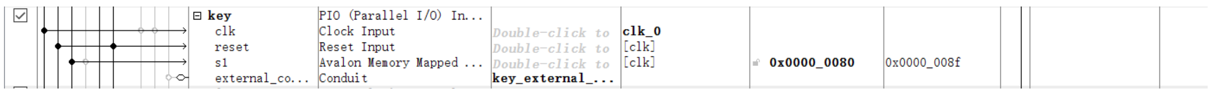
```
☑  ⊟ hex_digits_pio   PIO (Parallel I/O) In...
      clk             Clock Input             Double-click to  clk_0
      reset           Reset Input             Double-click to  [clk]
      s1              Avalon Memory Mapped ...Double-click to  [clk]  0x0000_00a0  0x0000_00af
      external_co...  Conduit                 hex_digits
```

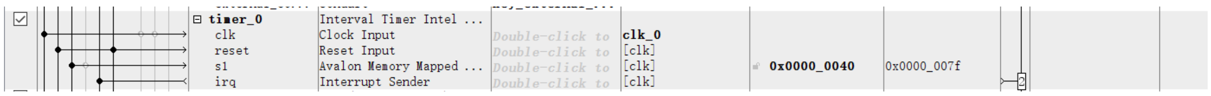
hex_digit_pio: The hex_digit_pio is a pio that outputs the hex display value and transfer the data to the Hex drivers

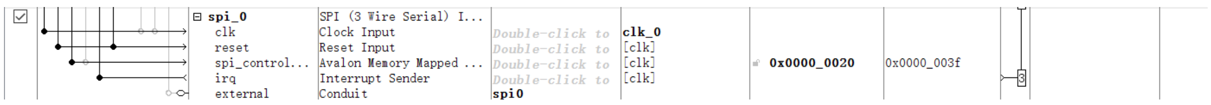


```
☑  ⊟ leds_pio         PIO (Parallel I/O) In...
      clk             Clock Input             Double-click to  clk_0
      reset           Reset Input             Double-click to  [clk]
      s1              Avalon Memory Mapped ...Double-click to  [clk]  0x0000_0090  0x0000_009f
      external_co...  Conduit                 leds
```

leds_pio: The led is a pio that output the value to the LEDs


```
key              PIO (Parallel I/O) In...                       clk_0
  clk            Clock Input              Double-click to      [clk]
  reset          Reset Input              Double-click to      [clk]
  s1             Avalon Memory Mapped ... Double-click to      [clk]   0x0000_0080   0x0000_008f
  external_co... Conduit                  key_external_...
```
key: The key is a pio that takes in the input value on the key0 which is the reset button and transfer the data to the NIOS2

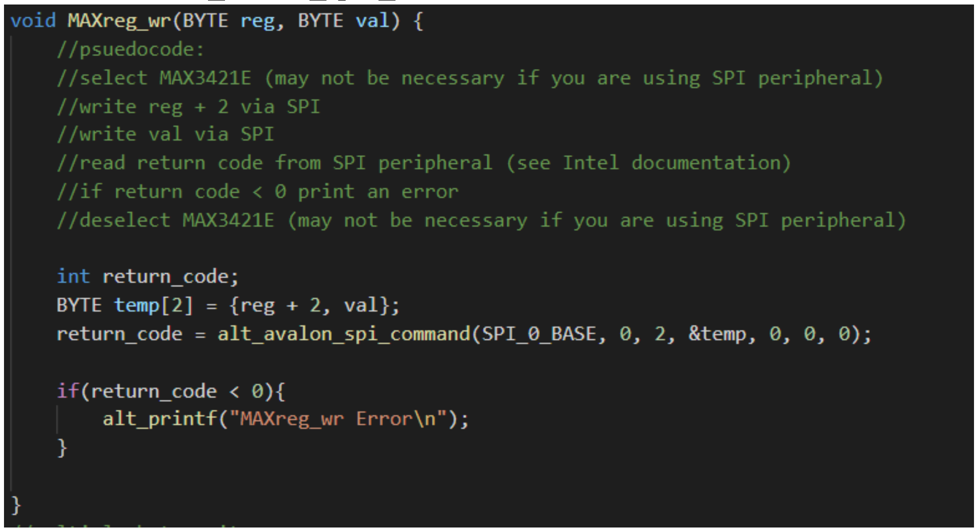

```
timer_0          Interval Timer Intel ...                       clk_0
  clk            Clock Input              Double-click to      [clk]
  reset          Reset Input              Double-click to      [clk]
  s1             Avalon Memory Mapped ... Double-click to      [clk]   0x0000_0040   0x0000_007f
  irq            Interrupt Sender         Double-click to      [clk]
```
timer_0: The timer_0 is a peripheral timer that keep track various time-outs that USB requires


```
spi_0            SPI (3 Wire Serial) I...                       clk_0
  clk            Clock Input              Double-click to      [clk]
  reset          Reset Input              Double-click to      [clk]
  spi_control... Avalon Memory Mapped ... Double-click to      [clk]   0x0000_0020   0x0000_003f
  irq            Interrupt Sender         Double-click to      [clk]
  external       Conduit                  spi0
```
spi_0: The spi_0 transfers data between the NIOS2 and the MAX3421E using the SPI protocol

Software Component Description

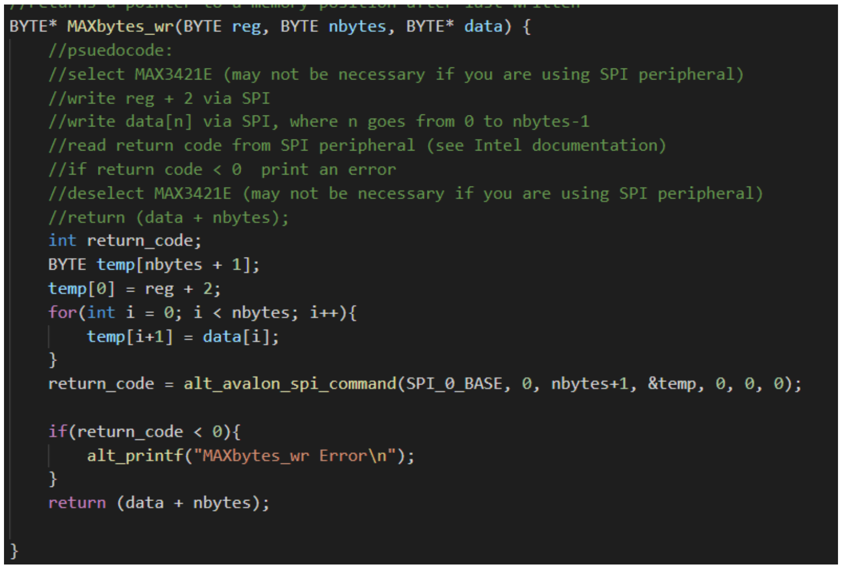
MAXreg_wr

input: reg (the register that we want to read data from) and val (the value that we need to write to the register)
return: void
Description: For this function, we firstly create an array with length of 2. The first value is the register that we would like to write to.And the reason for reg+2 is to tell NIOS2 with a write operation. Then the second value in the array is to tell the value we would like to write to the register. Then we call the alt_avalon_spio_command function to finish this function.


```c
void MAXreg_wr(BYTE reg, BYTE val) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg + 2 via SPI
    //write val via SPI
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0 print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)

    int return_code;
    BYTE temp[2] = {reg + 2, val};
    return_code = alt_avalon_spi_command(SPI_0_BASE, 0, 2, &temp, 0, 0, 0);

    if(return_code < 0){
        alt_printf("MAXreg_wr Error\n");
    }

}
```

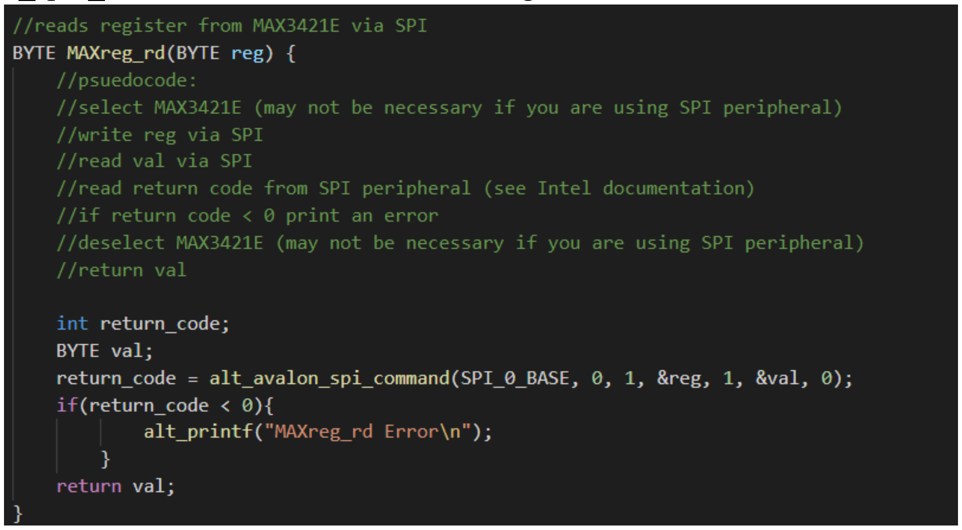
MAXbytes_wr

input: reg (the register that we want to read data from), nbytes (the length of data that we would like to read) data (the pointer to the memory position that we would like to start with)
return: data+nbytes (a pointer to a memory position after last written.)
Description: We first create an array with size (nbytes+1) and the first part is to store the register we want to write to. Then we write the data needed into the array. And we finish this function by calling the alt_avalon_spio_command function.



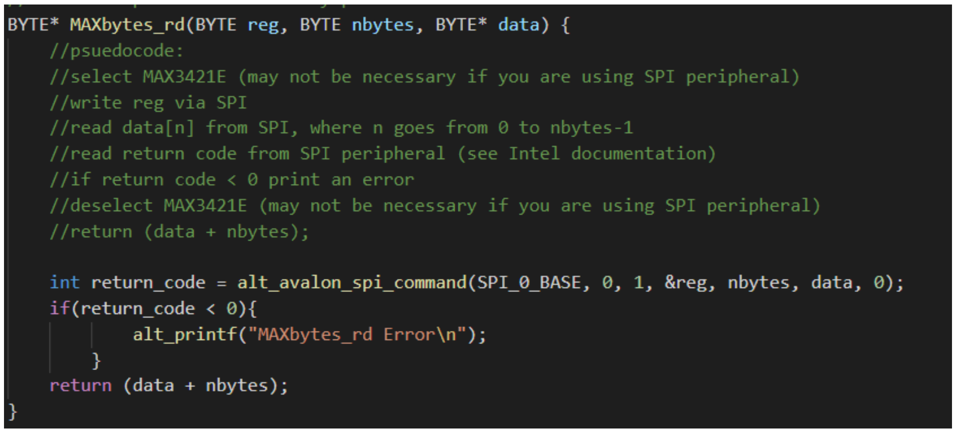
```
BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg + 2 via SPI
    //write data[n] via SPI, where n goes from 0 to nbytes-1
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0  print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    //return (data + nbytes);
    int return_code;
    BYTE temp[nbytes + 1];
    temp[0] = reg + 2;
    for(int i = 0; i < nbytes; i++){
        temp[i+1] = data[i];
    }
    return_code = alt_avalon_spi_command(SPI_0_BASE, 0, nbytes+1, &temp, 0, 0, 0);

    if(return_code < 0){
        alt_printf("MAXbytes_wr Error\n");
    }
    return (data + nbytes);

}
```

MAXreg_rd
input: reg (the register that we want to read from)
return: val (the value we read from.)
Description: We first create the parameter val that we want to return from this function and then call the alt_avalon_spio_command function to read the register from MAX3421E via SPI.



```
//reads register from MAX3421E via SPI
BYTE MAXreg_rd(BYTE reg) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg via SPI
    //read val via SPI
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0 print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    //return val

    int return_code;
    BYTE val;
    return_code = alt_avalon_spi_command(SPI_0_BASE, 0, 1, &reg, 1, &val, 0);
    if(return_code < 0){
            alt_printf("MAXreg_rd Error\n");
        }
    return val;
}
```

MAXbytes_rd
input: reg (the register that we want to read data from), nbytes (the length of data that we would like to read) data (the pointer to the memory that we would like to start storing read data in)
return: data+nbytes(a pointer to a memory position after last written.)
Description: We call the alt_avalon_spio_command function to write the data into a given data array and return a pointer to a memory position after last written.



```
BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg via SPI
    //read data[n] from SPI, where n goes from 0 to nbytes-1
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0 print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    //return (data + nbytes);

    int return_code = alt_avalon_spi_command(SPI_0_BASE, 0, 1, &reg, nbytes, data, 0);
    if(return_code < 0){
            alt_printf("MAXbytes_rd Error\n");
        }
    return (data + nbytes);
}
```

Design Resources and Statistics

LUT	**7356**
DSP	**10**
Memory(BRAM)	**11392(1%)**
Flip-Flop	**2873**
Frequency	**75.05 MHz**
Static Power	**96.18 mW**
Dynamic Power	**0.70 mW**

Total Power	**106.19 mW**

Difficulty Evaluation

We have implemented the baseline features, which is granted for 6 points

- The visualization of the maze, pacman, ghosts, and pec-dots.
- The pacman can move according to the keyboard response and collide with the maze.
- The pecman can eat small beans and show the corresponding score.
- The ghosts can chase the pacman randomly and uniformly, and the pacman would lose a life when catched by ghosts. The number of lives remaining is shown through a text board.

We have also implemented some additional features, which can add 2 more points

- The pacman can eat a kind of special dot called power pellet, with which the ghosts will run away from the pacman in a certain period of time. During this period, the ghosts would change colors, and the pacman can eat ghosts and win extra points.
- The game starts when the pacman eats the first pacdot, and the ghost starts to chase the pacman when the pacman eats 1, 25, 50 and 75 pacdots. When the score reaches 200 points, a game-over scene will show up. The user could restart the game by pressing the space.

Therefore, the total expected difficulty is 8 points.

Conclusion

Overall, we consider ourselves successfully finishing the base design of our final project as claimed in our proposal. The game can run functionally and smoothly. And we use the FPGA board to successfully show the play of the game during the final demo. This is a meaningful project for our team. For the first time, we use the language of System Verilog to design a traditional game on our own. This project also enhances our understanding of hardware and software implementations.

We can also improve our final projects in several aspects: first, we can design a new interface for showing if the game is winning or losing, we can also improve our state machine to add a multiplayer mode. Another player might take the control of a ghost to chase the pacman controlled by the other player. We can also add an audio effect using I2C& I2S interface to make the game more interesting. All of the features listed above are the improvements that we can make in this game, and we might improve the functionality of the game in the near future.