University of Illinois at Urbana-Champaign

ECE385
Digital Systems Laboratory

Final Project: Voice-Controlled Super Mario

TA: Abigail Wezelis, Harris Mohamed

Yan Miao, Guangxun Zhai
yanmiao2, gzhai5

May 2021

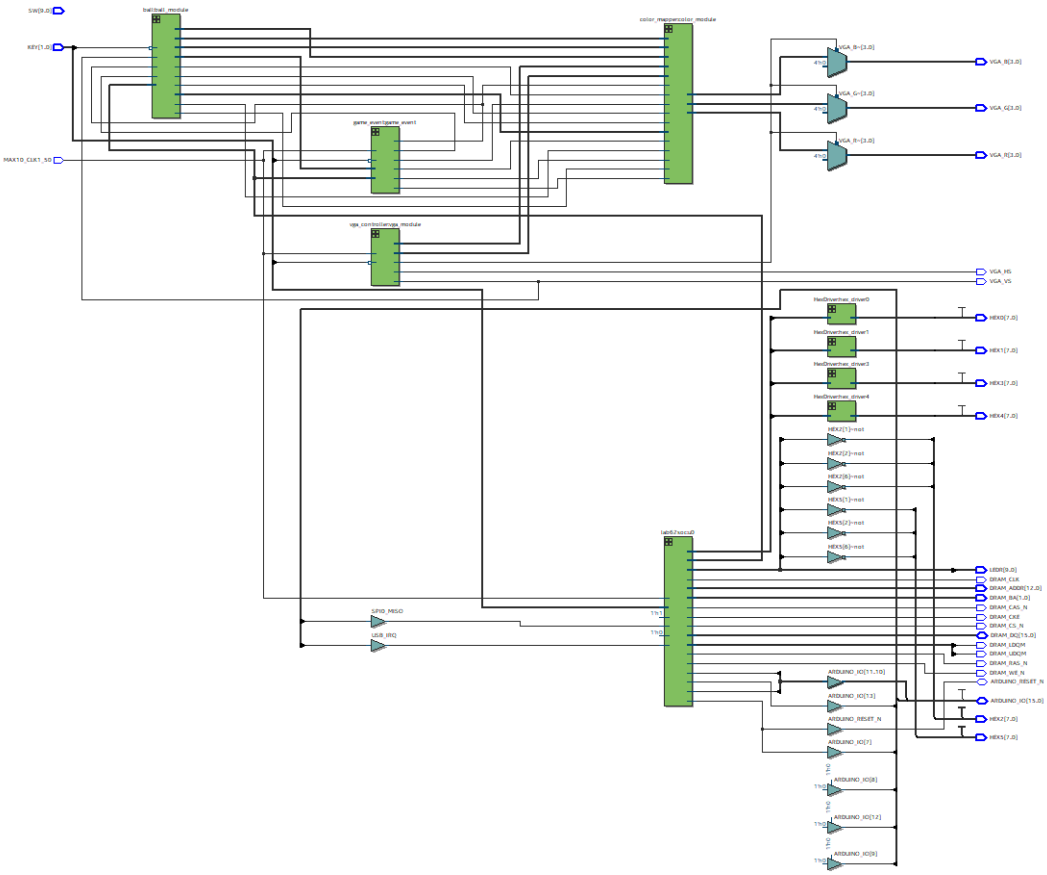
Contents

1 Introduction

For our final project, we developed a voice-controlled simplified Super Mario game. A video of our demo has been uploaded on Youtube. The project is partially based on lab6.2 "movement of the ball" and we almost kept the same platform designers and change the ball into our Mario. In addition, we added 26 sprites into the ROM to create the gaming environment as well as the texts/scores/signs. And we also involved a state machine to make the game make transitions between reset. start, pause, win and lose states. Moreover, a voice-controlled system has been included to control Mario do simple movements.

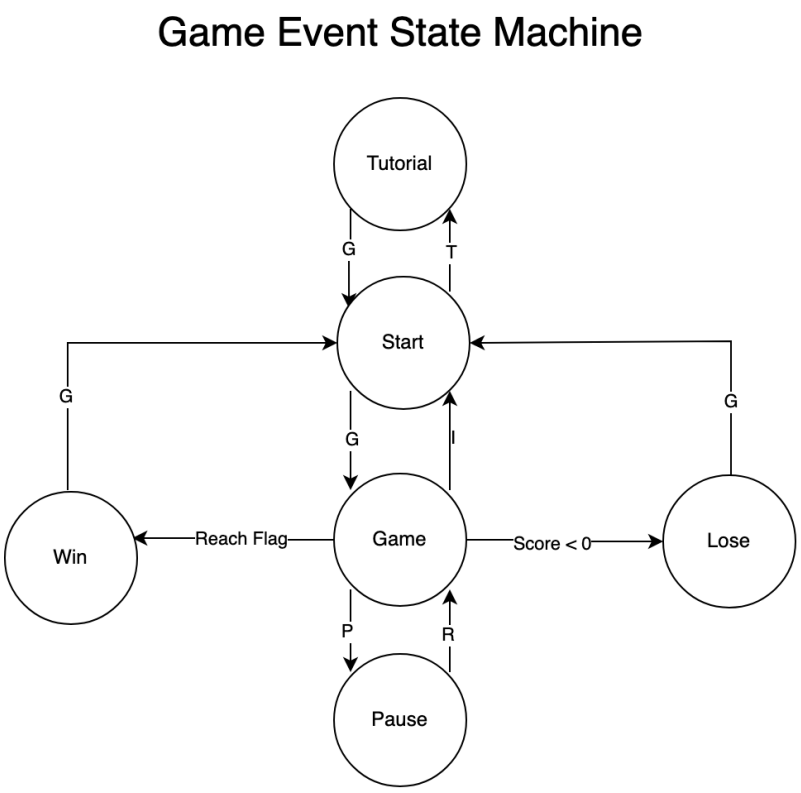
2 Overview(Diagram)

2.1 Top-Level Block Diagram



2.2 Game Event State Machine

After compilation and boot-up, players will first enter a start interface(Section 2.6.2). From there, they can go into the main game interface or a tutorial interface depending on the different keypress they choose. In the main game interface, the players have the option to pause/resume the game, they will also enter different interfaces based on the condition(win/lose) of the game.



2.3 Rules

We have implemented a simplified version of the Super Mario game. As also indicated by the above state diagram, the winning condition is that we have reached the flag(bottom right corner in Section 2.6.1). The losing condition is that the current game score is less than 0.

The player initially has a base score of 0, and will be awarded or deducted points as follows:

- **Coin: +8**

  This coin will appear at the upper left corner of the game interface (Section 2.6.1) from the start of the game and can only be collected once. (It will not re-appear after the player has reached and got points from that). This collection will increase the score by 8.

- **Mushroom: +1**

  When the player hits the right question block in the game interface (Section 2.6.1), a mushroom will appear at the top of that question block. Whenever the player collides with the mushroom, the score will be increased by 1. After the player collects the mushroom, he can hit the right question block again for another mushroom reward. We will go over the detailed logic in (Section 3.4)

- **Turtle: -1**

  When the player hits the left question block in the game interface (Section 2.6.1), a turtle will appear at the top of that question block. Whenever the player collides with the turtle, the

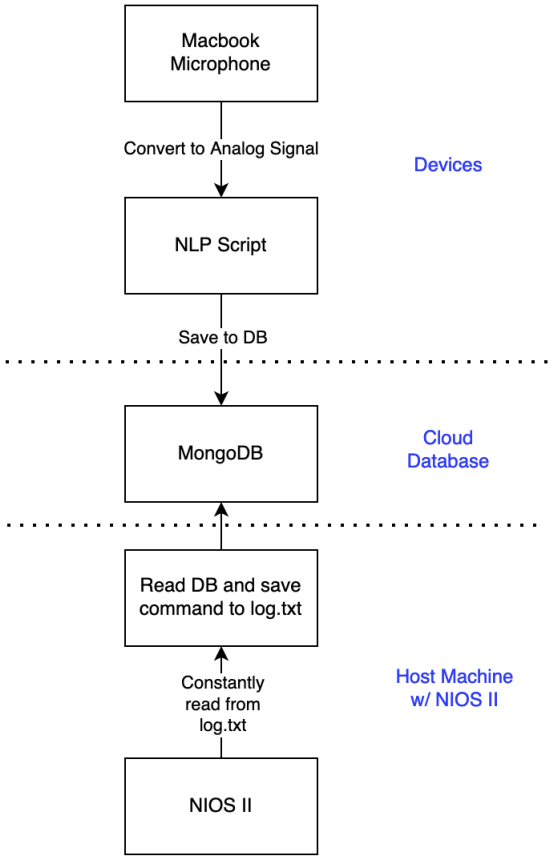
score will be decreased by 1. After the player collects the turtle, he can hit the left question block again for another turtle penalty. We will go over the detailed logic in (Section 3.4)

2.4 Keypress Instruction

- **W** : Jump up
- **A** : Move Left
- **D** : Move Right
- **G** : Start Game
- **I** : Restart Game
- **P** : Pause Game
- **R** : Resume Game
- **V** : Enable Voice Control
- **M** : Disable Voice Control

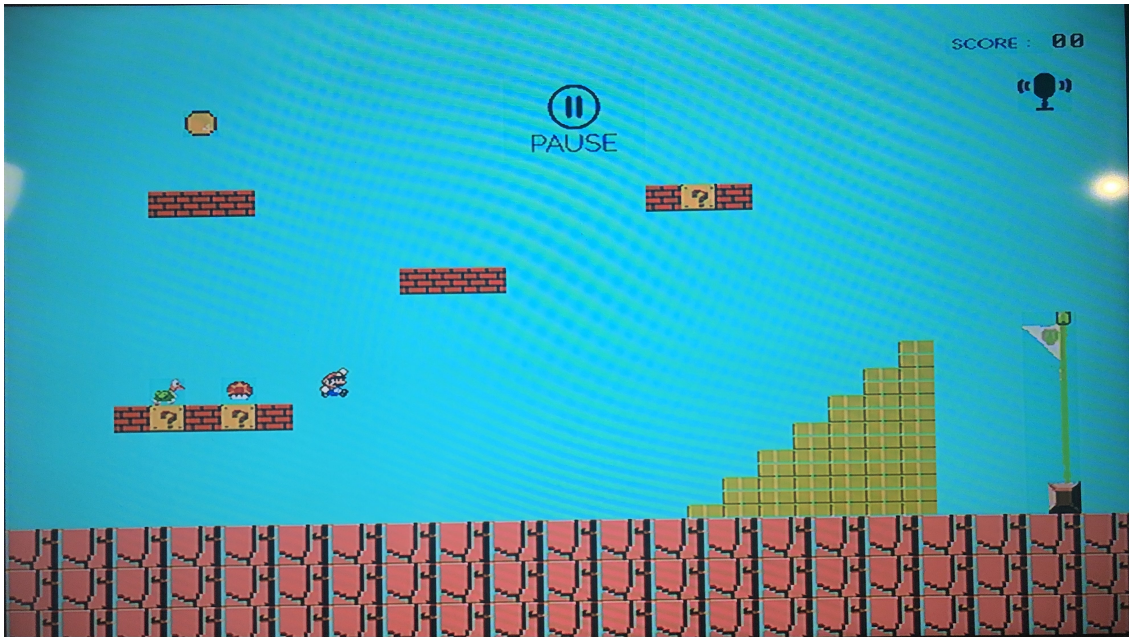
2.5 Vocal Control Diagram

When voice control is enabled, we used Yan's Macbook built-in Microphone as our voice-input device. With the NLP Script and Yan's Mongo-DB connection(check Section 3.8 for implementation details), we pass the verbal command as a digital signal(variable) to our NIOS-II Processor.

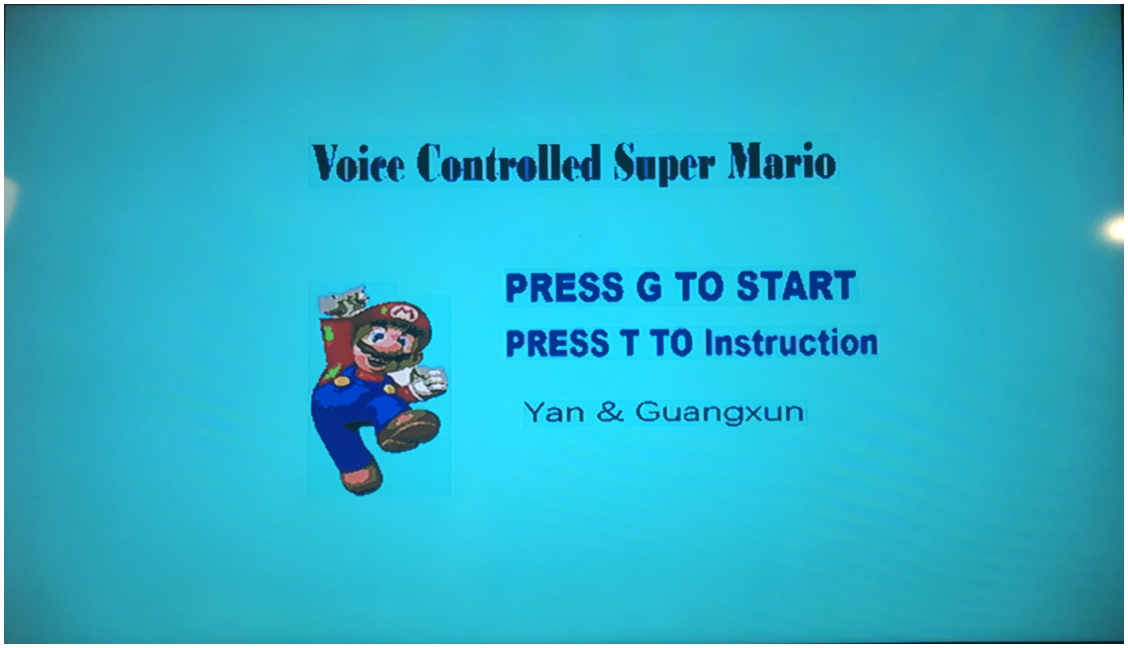


2.6 Interfaces

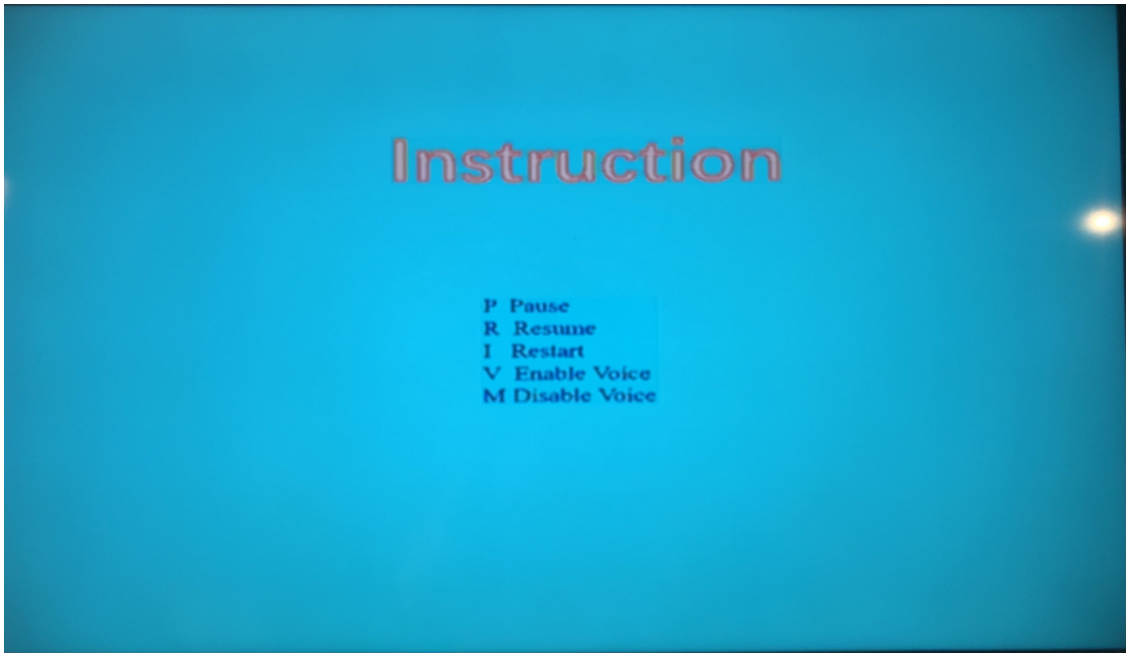
2.6.1 Main Game Interface



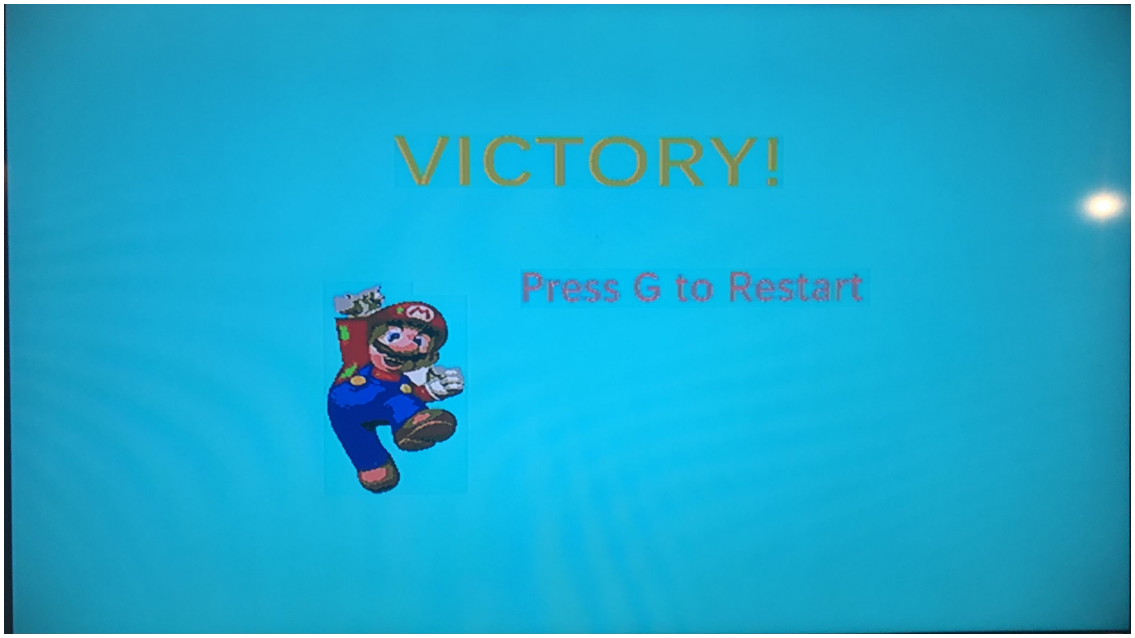
2.6.2 Start Interface



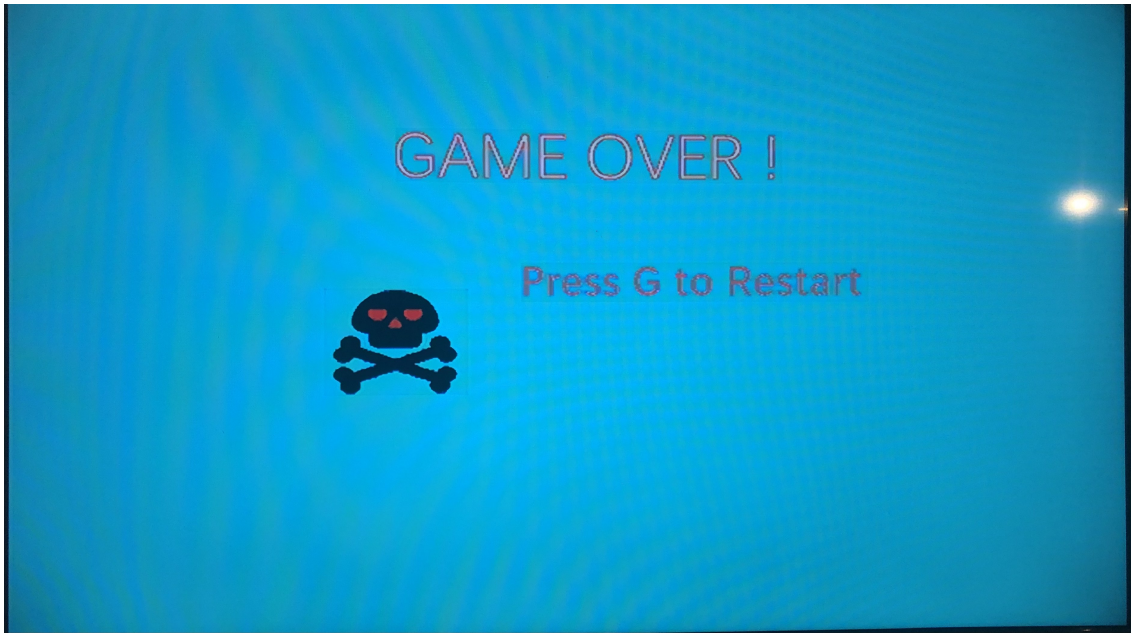
2.6.3 Tutorial Interface



2.6.4 Win Interface



2.6.5 Lose Interface



3 Feature

3.1 Mario Movement

The movement of Mario is similar to the movement of the ball done in lab 6.2 except that we deleted the horizontal bouncing and constant moving and replaced that with "moving only when key pressed". We also deleted the key "s" since Mario does not need to move downwards. Besides, the

ball was replaced by a Mario sprite converted by a helper tool from the ECE 385 website.

```
if (pause_on_mario == 1'b0) begin

    // A -- Left
    if (keycode == 8'h04) begin
        if (
            Ball_X_Pos > Ball_X_Min + 11'd8 &&  // Left Boundary
            !(Ball_X_Pos == 11'd520 && Ball_Y_Pos > 11'd240) // Right Flag Boundary
        )
            Ball_X_Pos <= Ball_X_Pos - 1'b1;
    end
```

3.2 Jumping & Free Fall

We attempted to simulate a more realistic free fall with gravitational acceleration at first, but we encountered so many bugs so we changed our falling into a constant falling with one pixel per clock cycle. We set a variable called "horizontal_level" to detect whether Mario is standing on the ground. And when Mario is not above the hard-coded "horizontal_level" and finishing the jumping movement, he can begin free fall.

```
// W -- Up
if (keycode == 8'h1A) begin
    up_key_pressed = 1'b1; // to make sure mario keeps jumping even if key not pressed
    if (no_jump != 1'b1) begin
        jump_counter = jump_counter + 6'b1;
        Ball_Y_Pos = Ball_Y_Pos - 2'd2;
        if (jump_counter == 6'b0)
            no_jump = 1'b1;
    end
end

// Continue freefall even after up key is release
if (up_key_pressed == 1'b1) begin
    if (
        jump_counter == 6'd55 ||  // Jump Height is reached
        Ball_Y_Pos <= 10'd4   ||  // Jump over the ceiling
        (Ball_X_Pos >= 10'd44 && Ball_X_Pos < 10'd160 && Ball_Y_Pos <= 10'd315 && Ball_Y_Pos >= 10'd310) || // Left bottom Bounceback
        (Ball_X_Pos >= 10'd64 && Ball_X_Pos < 10'd140 && Ball_Y_Pos <= 10'd145 && Ball_Y_Pos >= 10'd140) || // Left Upper Bounceback
        (Ball_X_Pos >= 10'd204 && Ball_X_Pos < 10'd280 && Ball_Y_Pos <= 10'd205 && Ball_Y_Pos >= 10'd200) || // Middle Bounceback
        (Ball_X_Pos >= 10'd344 && Ball_X_Pos < 10'd420 && Ball_Y_Pos <= 10'd145 && Ball_Y_Pos >= 10'd140)  // Right Upper Bounceback
    )begin
        no_jump = 1'b1;
        up_key_pressed = 1'b0;

        // check the questions block being touched
        if (Ball_X_Pos >= 10'd64 && Ball_X_Pos < 10'd100 && Ball_Y_Pos <= 10'd315 && Ball_Y_Pos >= 10'd310 )
            turtle_on_mario = 1'b1;

        if (Ball_X_Pos >= 10'd104 && Ball_X_Pos < 10'd140 && Ball_Y_Pos <= 10'd315 && Ball_Y_Pos >= 10'd310 )
            mushroom_on_mario = 1'b1;

    end else begin
        jump_counter = jump_counter + 6'b1;
        Ball_Y_Pos = Ball_Y_Pos - 2'd2;
    end
end

// When jump is done, begin freefall
else if (Ball_Y_Pos < horizontal_level && no_jump == 1'b1) begin
    Ball_Y_Pos = Ball_Y_Pos + 1'b1;
end

// reset after freefall
else begin
    if (Ball_Y_Pos < horizontal_level)
        Ball_Y_Pos = Ball_Y_Pos + 1'b1;
    no_jump <= 1'b0;
    up_key_pressed <= 1'b0;
end
```

Our Jump & Freefall Logic: When a jump is initiated by the keypress W, it always jumps up 120 pixels even if the player releases the key early. But the jump will be interrupted if it reached the maximum height or collide with a block above; In that case, our Mario will begin an automatic freefall motion.

3.3 Detect Obstacles

"Detect Obstacles" in our project means Mario cannot go through the obstacle. For detecting the obstacles above Mario, we wrote a feature called "Bounce Back" so that Mario won't pass through the bricks while jumping up. For detecting the obstacles beneath Mario, we wrote an if statement to adjust the "horizontal_level" when Mario is in a specific zone. (More precisely, Mario is standing on the bricks). The "Detect Obstacles" horizontal feature also follows the same logic as the one detecting barriers beneath Mario.

```
// Gound block level
horizontal_level = 10'd358;

// Bottom Left
if (Ball_X_Pos >= 10'd44 && Ball_X_Pos < 10'd160 && Ball_Y_Pos >= 10'd140 && Ball_Y_Pos <= 10'd300)
    horizontal_level = 10'd268;

// Upper Left
else if (Ball_X_Pos >= 10'd64 && Ball_X_Pos < 10'd140 && Ball_Y_Pos >= 10'd0 && Ball_Y_Pos <= 10'd140)
    horizontal_level = 10'd108;

// Middle
else if (Ball_X_Pos >= 10'd204 && Ball_X_Pos < 10'd280 && Ball_Y_Pos >= 10'd0 && Ball_Y_Pos <= 10'd200)
    horizontal_level = 10'd168;

// Upper Right
else if (Ball_X_Pos >= 10'd344 && Ball_X_Pos < 10'd420 && Ball_Y_Pos >= 10'd0 && Ball_Y_Pos <= 10'd140)
    horizontal_level = 10'd108;
```

Our Manually Coded Ground Level: based on block and ground.

3.4 Multiple Appearance of Mushroom & Turtle

First, we determine the collision logic: whenever the Mario reached the question block from below, and collide with the box, the turtle/mushroom will appear on display(right above the question blocks).

```
// check the questions block being touched
if (Ball_X_Pos >= 10'd64 && Ball_X_Pos < 10'd100 && Ball_Y_Pos <= 10'd315 && Ball_Y_Pos >= 10'd310 )
    turtle_on_mario = 1'b1;

if (Ball_X_Pos >= 10'd104 && Ball_X_Pos < 10'd140 && Ball_Y_Pos <= 10'd315 && Ball_Y_Pos >= 10'd310 )
    mushroom_on_mario = 1'b1;
```

Next, whenever the turtle/mushroom is on display and our Mario reaches its position, the point is awarded/deducted based on the different object, and we immediately take the object off the display and set the proper flags to make sure the objects can re-appear.

```
// check if turtle/mushroom has been touched
if (Ball_X_Pos >= 10'd64 && Ball_X_Pos < 10'd100 && Ball_Y_Pos <= 10'd268 && Ball_Y_Pos >= 10'd248 ) begin
    if (score > 7'b0 && turtle_on_mario == 1'b1)
        score = score - 7'b1;
    else if (score == 7'b0 && turtle_on_mario == 1'b1)
        game_condition = 2'd2; // lose the game
    // ensure score only change and turtle will disappear on first collision
    turtle_on_mario = 1'b0;
end

if (Ball_X_Pos >= 10'd104 && Ball_X_Pos < 10'd140 && Ball_Y_Pos <= 10'd268 && Ball_Y_Pos >= 10'd248 )begin
    if (mushroom_on_mario == 1'b1)
        score = score + 7'b1;
    // ensure score only change and mushroom will disappear on first collision
    mushroom_on_mario = 1'b0;
end
```

3.5 Bounce Back

"Bounce Back" happens when Mario doing jumping up, and its position reaches the height of the brick or other obstacles, and we force Mario to do free fall here.

```
if (
    jump_counter == 6'd55 ||  // Jump Height is reached
    Ball_Y_Pos <= 10'd4   ||  // Jump over the ceiling
    (Ball_X_Pos >= 10'd44 && Ball_X_Pos < 10'd160 && Ball_Y_Pos <= 10'd315 && Ball_Y_Pos >= 10'd310) || // Left bottom Bounceback
    (Ball_X_Pos >= 10'd64 && Ball_X_Pos < 10'd140 && Ball_Y_Pos <= 10'd145 && Ball_Y_Pos >= 10'd140) || // Left Upper Bounceback
    (Ball_X_Pos >= 10'd204 && Ball_X_Pos < 10'd280 && Ball_Y_Pos <= 10'd205 && Ball_Y_Pos >= 10'd200) || // Middle Bounceback
    (Ball_X_Pos >= 10'd344 && Ball_X_Pos < 10'd420 && Ball_Y_Pos <= 10'd145 && Ball_Y_Pos >= 10'd140)  // Right Upper Bounceback
)begin
    no_jump = 1'b1;
    up_key_pressed = 1'b0;
```

3.6 Winning/Losing Scores

The rules for scores in our game are when Mario hits the mushroom, the score shown on the upright corner + 1; when Mario hits the turtle, the score - 1; when Mario hits the coin, the score + 8. Since we have the coordinate information of mushroom, turtle, and coin, we wrote the code that whenever Mario reaches these positions and those obstacles are shown on the screen, those obstacles will disappear and the scores will be influenced.

```
// Check if reached the flag: WIN THE GAME
if (Ball_X_Pos >= 10'd554 && Ball_X_Pos < 10'd600 && Ball_Y_Pos <= 10'd300 && Ball_Y_Pos >= 10'd208 )begin
    game_condition = 2'd1;
end
```

3.7 Switching Interfaces

There are in total 5 pages in our game: start, win, lose, instruction, and an interface for playing Mario. Most of the "Page Switching" is implemented by pressing a key and the state machine. The winning and losing interfaces are achieved by the scores and reaching the flag.

We mainly used the keycodes as our inputs to switch between interfaces, the only exception is that the win game signal and the lose game signal come from the main game(ball.sv) to the game event.

```
// Assign outputs based on state
always_comb begin: next_state_logic
    next_state = state;
    unique case (state)
        start:  begin
            if (keycode == 8'd10 ) // if G is pressed, Game On
                next_state = game;
            if(keycode == 8'h17) // If T is Pressed, go to instruction
                next_state = tutorial;
        end

        game: begin
            if (keycode == 8'd12 ) begin// if I is pressed
                next_state = start;
            end

            if(game_condition == 2'd1)
                next_state = win;
            else if (game_condition == 2'd2)
                next_state = lose;
        end

        // Win & Lose & Tutorial going back to Start
        win,
        lose,
        tutorial: if(keycode == 8'd11) // if H is pressed
            next_state = start;

        default: next_state <= start;
    endcase
end
```

Since we developed a Moore State Machine, the output is purely decided by the current state.

```
always_comb begin: state_actions
    start_interface = 1'b0;
    game_mode = 1'b0;
    win_interface = 1'b0;
    lose_interface  = 1'b0;
    game_reset = 1'b0;
    tutorial_interface = 1'b0;

    // Assign outputs based on 'state', Moore Machine
    case (state)
        start: begin
            start_interface = 1'b1;
            game_reset = 1'b1;     // to reset the game variables in ball.sv
        end

        game:  game_mode = 1'b1;

        win:  win_interface = 1'b1;

        lose:  lose_interface = 1'b1;

        tutorial:  tutorial_interface = 1'b1;
    endcase
end
```

And the output of the state machine will be connected to the color mapper and the main game through our top-level file lab62.sv.

```
game_event game_event(
    .Clk(MAX10_CLK1_50),
    .Reset(Reset_h),
    .keycode,
    .game_condition, // Inputs coming from ball.sv
                     //    0: Game Going on or Not started yet
                     //    1: Win Game
                     //    2: Lose Game
    .game_reset,

    .start_interface, // Outputs going to color mapper & ball
    .game_mode,       // Outputs going to color mapper & ball
    .win_interface,   // Outputs going to color mapper & ball
    .lose_interface,  // Outputs going to color mapper & ball

    .tutorial_interface

);
```

We color the interfaces/background according to the signal sent from the game event state machine.

```
// Win Interface
end else if (win_interface == 1'b1) begin
    if(win_interface_victory_on == 1'b1) begin
        Red = win_interface_victory_data[23:16];
        Green = win_interface_victory_data[15:8];
        Blue = win_interface_victory_data[7:0];
    end
    if(win_interface_mario_on == 1'b1) begin
        Red = interface_mario_data[23:16];
        Green = interface_mario_data[15:8];
        Blue = interface_mario_data[7:0];
    end
    if(win_interface_restart_on == 1'b1) begin
        Red = interface_restart_data[23:16];
        Green = interface_restart_data[15:8];
        Blue = interface_restart_data[7:0];
    end
```

3.8 Voice Control

In the voice-controlled part, we first set up the connection between Yan's Macbook with his MongoDB Cloud Database.

```python
r = s_r.Recognizer()
my_mic = s_r.Microphone(device_index=0)    # Use device0 -- MacBook built-in Microphone
while True:
    with my_mic as source:
        print("Say now!!!!")
        voice.update({}, {"direction": 66}) # default value to dingtinguish from command value
        r.adjust_for_ambient_noise(source)  # reduce noise
        audio = r.listen(source)            # take voice input from the microphone

    sentence = r.recognize_google(audio)
    print(sentence)        # Debugging: to print voice into text
    words = sentence.split()
    todo = []
    for word in words:
        if word in ["jump", "pause", "resume", "restart"]:
            todo.append(word)
    for word in todo:
        # 0: Jump
        # 1: Pause
        # 2: Resume
        # 3: Restart
        # 66: Default
        if word == "jump":
            voice.update({}, {"direction": 0})
            system('say Now jumping up')
        elif word == "pause":
            voice.update({}, {"direction": 1})
            system('say Now pausing the game')
        elif word == "resume":
            voice.update({}, {"direction": 2})
            system('say Now resuming the game')
        elif word == "restart":
            voice.update({}, {"direction": 3})
            system('say Now restarting the game')
```

Then we established the voice recognizer script on Yan's Macbook with the help of speech recognizer API. Then we assign the certain command received from Yan's Macbook built-in microphone with a specific value and send that value to the MongoDB Cloud Database.

```python
import pymongo
from pymongo import MongoClient

# temporary PWD expired 5/7/2021
client = MongoClient(
    "mongodb+srv://yanmiao2:123456@yan-personal.klzbu.mongodb.net/"
)
db = client.get_database("385")
voice = db.voice_control
```

When the value is stored in the database, a script on Guangxun's host computer will constantly read from that database and store the value corresponding to the command in a local txt file.

```python
while True:
    temp = voice.find_one({})
    with open("C:\\/Users/Alfoul/Desktop/shared_385/Lab62/software/usb_kb/log.txt", "w") as f:
        f.write(str(temp['direction']))
```

Then in our NIOS-II processor, we mounted the file system with the help of this File System document and read from that local txt file all the time.

```c
// Only read from DB if voice is enabled
if (voice_enabled == 1)
{
    if ((fptr = fopen("/mnt/host/log.txt", "r")) == NULL)
    {
        printf("FILE READ FAILED \n");
        return 0;
    }
    fscanf(fptr, "%d", &num);

    fclose(fptr);
    printf("FILE IS %d \n", num);
    if (num == 0)
    {
        kbdbuf.keycode[0] = 0x1A;
    }
    else if (num == 1)
    {
        kbdbuf.keycode[0] = 0x13;
    }
    else if (num == 2)
    {
        kbdbuf.keycode[0] = 0x15;
    }
    else if (num == 3)
    {
        kbdbuf.keycode[0] = 0x0C;
        voice_enabled = 0;
    }
}
```

3.9 ROM for Sprite

We used on-chip memory to store all our sprites since we didn't require too many storage spaces. We make the memory read-only because we are only using static sprites so there's no need to modify the sprite itself.

```
initial begin
    $readmemh("C:\/Users/Alfoul/Desktop/shared_385/Lab62/ECE385-HelperTools-master/PNG To Hex/On-Chip Memory/sprite_bytes/mario1.txt", mem);
end

always_comb begin
    data = palette[mem[addr]];
end
```

We used the provided script from the helper tools on ECE385 Website.

4 Module Descriptions

1. **Module: coin_ROM**
    - Inputs: addr.
    - Outputs: data.
    - Description: This is the ROM file which utilize the .txt files to gain the address information and convert them into color data. We have 26 ROM files in our project, all of them followed the same structure of code and share the same purpose, so I will just write the description of "coin_ROM" as an example.
    - Purpose: convert the image PNG files into data that can print on the screen.
2. **Module: lab62.sv**
    - Inputs: MAX10_CLK1_50, KEY, SW, DRAM_DQ, ARDUINO_IO, ARDUINO_RESET_N.
    - Outputs: LEDR, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, DRAM_CLK, DRAM_CKE, DRAM_ADDR, DRAM_BA, DRAM_LDQM, DRAMUDQM, DRAM_CS_N, DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N, VGA_HS, VGS_VS, VGA_R, VGA_G, VGA_B, ARDUINO_IO, ARDUINO_RESET_N.
    - Description: this module is the top-level file of our final project and remains the same as lab 6.2. It basically connects all the components together to run the design.
    - Purpose: serves as an interface for connecting FGPA and the c code.
3. **Module: lab62soc.v**
    - Inputs: clk_clk, key_external_connection_export, reset_reset_n, sdram_wire_dq, spi0_MISO, usb_gpx_export, usb_irq_export.
    - Outputs: hex_digits_export, keycode_export, leds_export, sdram_clk_clk, sdram_wire_addr, sdram_wire_ba, sdram_wire_cas_n, sdram_wire_cke, sdram_wire_cs_n, sdram_wire_dqm, sdram_wire_ras_n, sdram_wire_we_n, spi0_MOSI, spi0_SCLK, spi0_SS_n, usb_rst_export.
    - Description: this file is generated by the platform designer for all the hardware components used in the lab 6.2 design. It includes all the connections of the wires that we made in the platform designer.
    - Purpose: provide all the wires and hardware for the top-level module to use and interact.
4. **Module: HexDriver.sv**
    - Inputs: In0.
    - Outputs: Out0.

- Description: Each hex value has its unique corresponding 7-bit binary value. (7 bits because we modify the HEX driver to not display decimal point by adding another bit).
- Purpose: Hex module is used to output the hex value on the board screen.

5. **Module: VGA_controller.sv**
    - Inputs: Clk, Reset
    - Outputs: hs, vs, pixel_clk, blank, sync, DrawX, DrawY
    - Description: This module keeps the same as lab6.2. It generates the Horizontal Sync(HS) and the Vertical Sync(VS) signals.
    - Purpose: Generate the coordinates corresponding to HS and VS, so that over modules like Color_Mapper() can draw to the coordinate and display it on screen.
6. **Module: Color_Mapper.sv**
    - Inputs: BallX, BallY, DrawX, DrawY, Ball_size, score, turtle_on_mario, mushroom_on_mario, left_coin_touched, pause_on_mario, start_interface, game_mode, win_interface, lose_interface, tutorial_interface, voice_enabled.
    - Outputs: Red, Green, Blue.
    - Description: This module determines the RGB values for the ball and the background, and we wrote most of our codes here. We designed all the features and game rules here.
    - Purpose: serves as a color settings for our game on the screen.
7. **game_event.sv**
    - Inputs: Reset, Clk, keycode, game_condition.
    - Outputs: start_interface, game_mode, win_interface, lose_interface, tutorial_interface, game_reset.
    - Description: this module is a state machine that controls "Page Switching". Each interface is controlled by entering a keycode or fulfilling some conditions.
    - Purpose: control the game entering different interfaces.
8. **ball.sv**
    - Inputs: Reset, frame_clk, keycode, game_mode, game_reset.
    - Outputs: game_condition, BallX, BallY, BallS, turtle_on_mario, mushroom_on_mario, score, left_coin_touched, pause_on_mario, voice_enabled.
    - Description: this module controls the movement of Mario after pressing the keyboard as well as the logic related to Mario's hitting those obstacles.
    - Purpose: define the movement and rule of Mario and some winning/losing conditions.

5 Future Optimizations

The graph quality of our game is still far from satisfactory, the organization of the game image could be improved if we spend more time on planing the organization of each obstacle.

Currently, the character in our game is a motionless sprite. We could optimize the character image into a Dynamic Sprite, and make Mario doing different gestures when moving or jumping.

The delay time for our vocal system is too long. When the voice system is enabled, any movements of Mario will have a 3-second-delay after pressing the keys. Besides, it takes around 3 seconds for the AI system to recognize the voice instruction including "keywords". Up till now, we do not have a clue on how to solve this issue.

As written in our proposal, we planned to add a second character so that one character can depend on the key button to move and another depends on vocal control. The logic of add a second character would be the same as what we have implemented for the Mario character, just require double time and effort. The only difficulty that we might encounter for the double character would be how to receive two instructions simultaneously from players.

6 Conclusion

We really enjoy the whole process of our doing the final project, and also learn much about what we are interested in this course. It is our first time designing a game though many of the game concepts and features are borrowed from existed old games. We had a tough time connecting the vocal system to our project, but fortunately it worked at last. We both believe the knowledge of adding sprites and display them on the screen are very essential and interesting, which should definitely be considered to add into one lab to learn and implement ahead of the final project.