CS207 Project Report: ELECTRONIC PIANO LEARNING MACHINE

Group members:
Zerhouni Khal Jaouhara (12211456)
Hok Layheng (12210736)
刘淦 (12210729)

CS207, Southern University of Science and Technology
2023 FALL- PROJECT A

I. DEVELOPMENT SCHEDULE

- **Team division of labor:**
    a) 刘淦 : basic functionalities (AutoPlay Mode, Learning Mode), bonus points (real-time reaction rating, speed control)
    b) Hok Layheng : basic functionalities (Free Mode), VGA, videographer
    c) Zerhouni Khal Jaouhara : basic functionalities (octave adjustment, LED), code specification (comments, constants file), project document (report), video recording (demonstration of bonus points)
- **Contribution ratio:**
    a) 刘淦 : ⅓
    b) Hok Layheng : ⅓
    c) Zerhouni Khal Jaouhara : ⅓
- **Development plan schedule:**

We first started with implementing the Basic functions:

- Support switching between three basic modes (free mode, learning mode and autoplay mode).
- Being able to play the different notes by pressing the corresponding button/ switch.
- Automatically playing a song after entering the AutoPlay mode (initially we had “Little Star” and “Two Tigers” only).
- Designing Learning mode to guide the user to play a song by following the order of the LED lights.

Moved on to the Standard functions:

- Octave adjustment is available not only for free mode but for all modes.
- Adding a third song “Ode To Joy” to the music library.
- Adjusting the LED lights for the Learning mode accordingly (light on above the required note and once played, the light will be turned off until the song ends).
- Improve the overall performance of rating.

Targeting the Additional Creativity section:

- Real-time update of ratings in Learning mode.
- VGA display.
- Speed control of the songs in AutoPlay mode.

Handling the Code specification:

- Constant file (Verilog Header).
- Adding necessary comments.
- Checking and testing the codes multiple times.

Project Documentation:

- After gathering and assembling all the pieces of the project, we started working on the report.

Video Recording:

- Two members took part in recording the video to demonstrate the bonus points of the project.

NOTE: We were able to stay in touch by keeping each other updated on the progress of the project and meeting from time to time.

II. USER DOCUMENTATION

1. Introduction:

For this project, our group has decided to take " Electronic Piano Learning Machine" as our implementation. This involves implementing a mini electronic piano to help play music, as it supports mode selection and playing functions, as well as the use of VGA as an output device. There are three available modes: Free Mode, AutoPlay Mode and Learning Mode. In this project, we will be able to use the EGO1 FPGA board to play the different notes and the suggested songs through mode switching.

2. System Function List:

The functionality of our project combines simplicity, efficiency and creativity. The general idea of the project utilizes the FPGA EGO-1 as a device to represent an electronic piano. Before getting started, by pressing the mode selector button, the user should be able to choose between three different modes available (Free Mode, Learning Mode and AutoPlay Mode). Once done, our piano journey begins!
Now let's dive into the details of each mode:

- ***Free Mode:*** once the user has entered the free mode, they will be able to play the 7 piano notes (do, re, mi, fa, so, la, si). We designed the 7 switches in the bottom left of the FPGA board for this purpose. Once the switch is on, the corresponding note will be played and the LED light will be lit. The note is set to be played continuously until the switch is turned off. The user also can change the octave of the notes. Pressing the corresponding button "Octave adjustment" will lead to either a high octave or a low octave and according to the user's preference, the selected frequency/octave will be held for the upcoming notes.

- ***Learning Mode:*** Once done with the free mode, pressing the "Mode selector" button again will take the user to the learning mode. This mode is designed for the user to evaluate their piano playing skills. First, the user has to choose a user by pressing the "User selector" button. There are three users available with indexes 0, 1 and 2 that can be seen in the left-hand side of the 7-segment. After selecting the user, the player can choose the song they want to play; the choice is limited to three songs as in the music library, which contains (Little Star, Two Tigers and Ode To Joy); the user can then begin the performance. According to the selected song, the light on indicates the note that should be played and the corresponding key below it should be turned on. After doing so, the LED light will turn off and the following light will designate the note that should be played next and so on. With every correct move that the user does, and based on their speed/ reaction time to the light on, the user will get a score which can be seen in the LED lights (refer to the control diagram of the FPGA board), the maximum score for each correct move is 8. Once the user completes the selected song, and based on their performance, they will be able to see their score on the right-hand side of the 7-segment display (maximum 8) which will also be saved and can be checked after pressing the "Reset" button of the FPGA board and back to the learning mode. With the remaining two users, the user can either play the two other songs to compare his performance in each one of them or if they have a sense of competition, they can use them to compete with 2 other players, where, following the same demonstration and instructions as before, they will be able to get their scores and check their final results after resetting the FPGA board and then decide who is the winner of the game! Back to the octave adjustment functionality, the user can not only switch between low, normal and high octaves in the free mode, but also in the learning mode.

- ***AutoPlay Mode:*** Once entering the autoplay mode, the user will be able to hear a song and the LED of the corresponding played note will be on, thus, helping the user to keep track of the rhythm of the song. Additionally, by pressing the “Song selector” button on the left, the user should be able to choose a song from the music library (Little Star, Two Tigers and Ode To Joy). If the selected song is ‘Two Tigers’, the 7-segment will display number ‘1’, if it is ‘Little Star’, the 7-segment will show number ‘2’, else; if the song selected is ‘Ode To Joy’; number ‘3’ will be displayed. And just like other modes, the octaves can be adjusted in autoplay mode as well.
  Another creative feature that the user can enjoy in this mode is the functionality of changing speed! Yes, the user should be able to control the speed of the songs played by setting it either to slow, normal or fast simply by pressing the “Speed control” button.

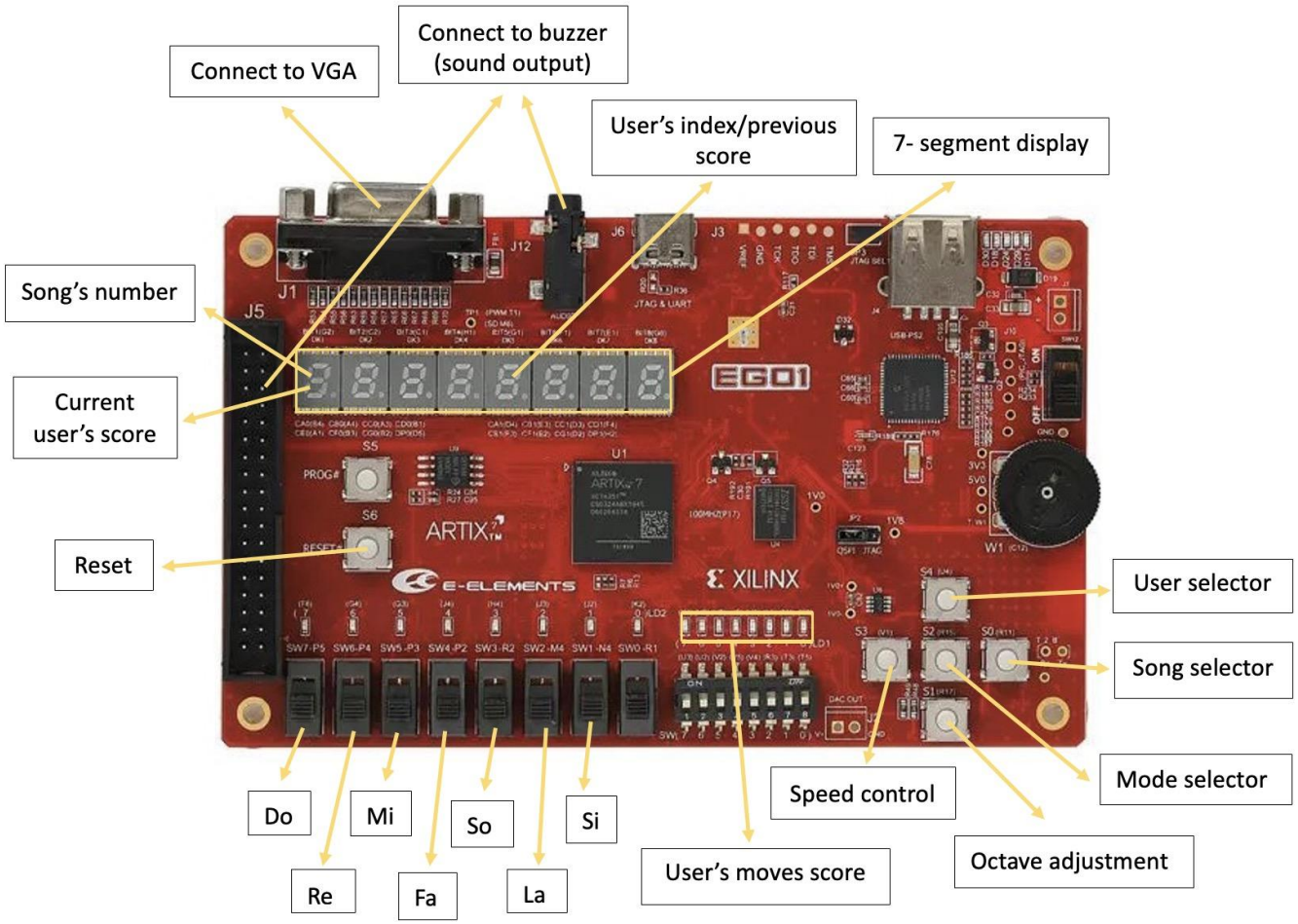
Now let’s get into the fun part of our project, the VGA display!

Once we get started, we will be able to see a piano with 7 “natural” keys (white) and 5 “sharp” or “flat” keys (black). After entering a mode; by following the steps provided above; a little colored square can be seen on the top right corner of the piano, indicating which mode has the user entered; blue for ‘free mode’, green for ‘learning mode’ and purple for ‘autoplay mode’. According to which mode the user is in, the pressed “natural” key will change color to the corresponding mode color. The user can also see which octave they are using in the bottom left corner of the piano with the words LOW, NORMAL and HIGH. The name of the song will also be displayed on the top left corner of the piano (LITTLE STAR, TWO TIGERS and ODE TO JOY). Regarding the learning mode, the score of the user after each correct move will be displayed on the bottom right corner of the piano, with the maximum score set to 8, and after successfully completing the song, two smiley emojis will appear in the top middle of the piano.

That is, we can reveal that our piano implementation triggers in its users both challenge and excitement!
The detailed guidelines and the functionalities of the FPGA board will be demonstrated below:

3. System usage instructions:

a) Control Diagram of FPGA Board:



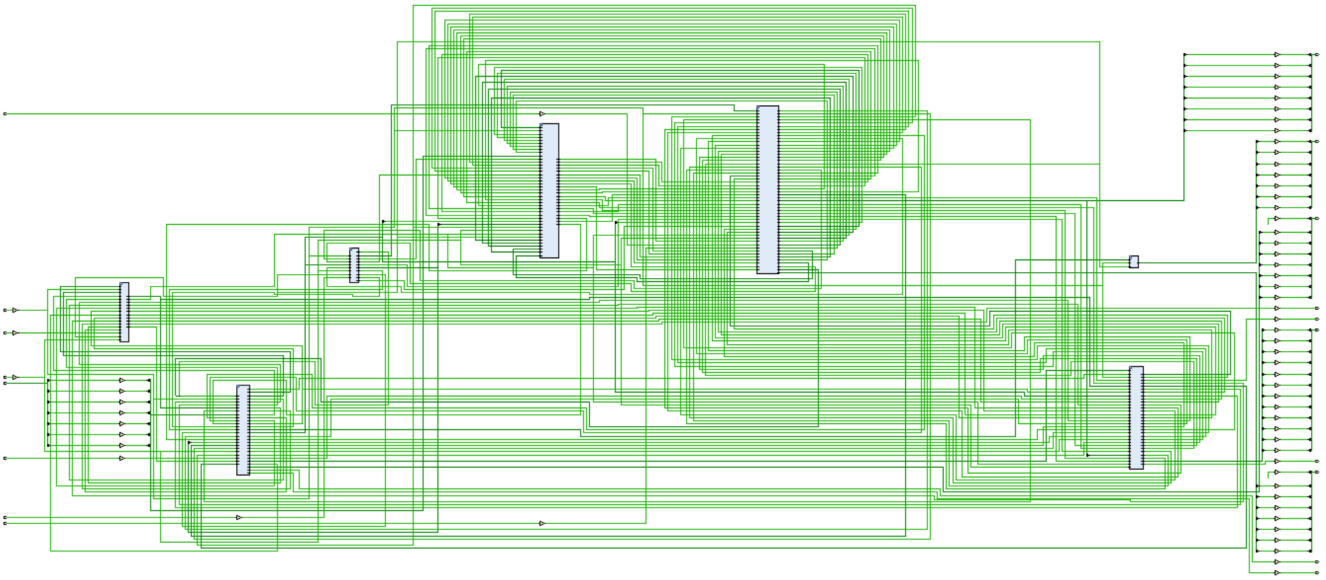
b) Guide to use FPGA board to play the piano:

- First, choose a mode using the '*Mode selector'* button:
    1. *Free Mode selected:*
        - Use any switch and the electronic keyboard will play the corresponding notes. (do re mi …)
        - The user can choose between the octaves (low, normal, high) by pressing the '*Octave adjustment'* button.
    2. *AutoPlay Mode selected:*
        - The user can choose one of the songs available (Little Star, Two Tigers and Ode To Joy) through the *'Song selector'* button and check the song's number in *'Song's number'.*
        - The user can also control the speed of songs (slow, normal, fast) using the *'Speed control'* button.
        - The user should be able to switch between octaves using the *'Octave adjustment'* button.
    3. *Learning Mode selected:*
        - *The user can use the 'User selector' button to select a user and check their index in 'User's index/ previous score'.*
        - The user can select the song they want to play using the *'Song selector'* button.

    - The user should be able to see the LED light corresponding to their moves' scores in '*User's move score'.*
    - The final score can be seen in *'Current user's score'* and can be checked again in *'User's index/ previous score'* after resetting the game using the *'Reset'* button.
    - The user should be able to switch between octaves using the *'Octave adjustment'* button.
- The 7-segment display:
    - *Two tigers:* display '1'
    - *Little star:* display '2'
    - Ode To Joy: display '3'
- VGA:
    - Connect to the VGA to display the electronic piano on a monitor.
- Buzzer:
    - Connect to the Buzzer to be able to hear the sound.

3. Architecture diagram of the circuit:

Schematic Design:



III. GAME IMPLEMENTATION

Piano module (Top Module):

Our top module for this project is "Piano". It describes the top-level module of the electronic piano implementation, which also links all the major modules (KeyController, AutoPlay, LearningMode, ModeSelector, LED, Buzzer). This includes determining which module will be activated/ generated.

The following explanation will give a clear and detailed approach of the module's components and functionality:

Inputs:

- clk: Clock signal.
- rst_n: Active_low reset signal.
- set_mode: Signal to set the mode.
- octave_button: Signal from the octave button.
- set_speed: Signal to set the speed.
- set_song: Signal to set the song.
- key_notes: 7-bit input for the key notes.
- set_user: Signal to set the user.

```
module Piano(
    input wire clk,
    input wire rst_n,
    input wire set_mode,
    input wire octave_button,
    input wire set_speed,
    input wire set_song,
    input wire [`KEY_NOTES_WIDTH -1:0] key_notes,
    input set_user,
```

Outputs:

- speaker: Output signal for the speaker.
- light: 7- bit output signal for lights.
- tub_sel: Output signal for tub selection.
- tub_sel_user: Output signal for user-specific tub selection.
- tub_control: 8-bit Output signal for tub control.
- tub_control_user: Output signal for user-specific tub control.
- hsync: Output signal for horizontal synchronization.
- vsync: Output signal for vertical synchronization.
- vga_rgb: Output signal for VGA RGB.
- reaction_rate: Output signal for reaction rate.

```
output wire speaker,
output wire [`LIGHT_WIDTH -1:0] light,
output wire tub_sel,
output wire tub_sel_user,
output wire [`TUB_CONTROL -1:0] tub_control,
output wire [`TUB_CONTROL_USER -1:0] tub_control_user,
output hsync,
output vsync,
output [`VGA_RGB_WIDTH -1:0] vga_rgb,
output [`REACTION_RATE_WIDTH -1:0] reaction_rate
);
```

Wires:

- key_on_A, key_off_A, note_A (6-bit): Output signals from the KeyController module
- key_on_B, key_off_B, note_B (6-bit), tub_control_B(8-bit): Outputs signals from the AutoPlay module.
- key_on_C, key_off_C, note_C (6-bit), tub_sel_C, tub_control_C (8-bit): Output signals from the Learning Mode module
- key_on, key_off, note (6-bit), mode (2-bit): Output signals from the ModeSelector module.
- selected_note (6-bit), current_octave (2-bit): Helper output signals from the Buzzer module.
- track_number_auto (2-bit), track_number_learning (2-bit): Signals fromthe Music module and LearningMode module.
- finish: Signal from the LearningMode module.

```
//output from keyController module
wire key_on_A;
wire key_off_A;
wire[`NOTE_A_WIDTH -1:0] note_A;

// output from autoPlay module
wire key_on_B;
wire key_off_B;
wire[`NOTE_B_WIDTH -1:0] note_B;
wire [`TUB_CONTROL_B -1:0] tub_control_B;

// output from learning mode module
wire key_on_C;
wire key_off_C;
wire[`NOTE_C_WIDTH -1:0] note_C;
wire tub_sel_C;
wire[`TUB_CONTROL -1:0] tub_control_C;

// after selection from ModeSelector module
wire key_on;
wire key_off;
wire [`NOTE_WIDTH -1:0] note;
wire [`MODE_WIDTH -1:0] mode;

// helper output from buzzer
wire [`SELECTED_NOTE_WIDTH -1:0] selected_note;
wire [`CURRENT_OCTAVE_WIDTH -1:0] current_octave;

// from music module
wire [`TRACK_NUM_AUTO_WIDTH -1:0] track_number_auto;
wire [`TRACK_NUM_LEARNING_WIDTH -1:0] track_number_learning;

// from learning_mode
wire finish;
```

The instantiation of the submodules is as follow:

```
//instantiating submodules
    KeyController key_controller (
        .clk(clk),
        .rst_n(rst_n),
        .key_notes(key_notes),
        .key_on_out(key_on_A),
        .key_off_out(key_off_A),
        .note_out(note_A)

    );
    AutoPlay auto_play(
        .clk(clk),
        .rst_n(rst_n),
        .set_speed(set_speed),
        .set_song(set_song),
        .key_on_out(key_on_B),
        .key_off_out(key_off_B),
        .note(note_B),
        .seg(tub_control_B),
        .track_number_auto(track_number_auto)
    );
 LearningMode learning_mode(
     .clk(clk),
     .rst_n(rst_n),
     .set_song(set_song),
     .set_user(set_user),
     .key_notes(key_notes),
     .key_on_out(key_on_C),
     .key_off_out(key_off_C),
     .notes(note_C),
     .track_number_learning(track_number_learning),
     .reaction_rate(reaction_rate),
     .tub_sel(tub_sel_C),
     .tub_control(tub_control_C),
     .tub_control_user(tub_control_user),
     .finish(finish)
 );
```

```
    ModeSelector(
        .clk(clk),
        .rst_n(rst_n),
        .set_mode(set_mode),
        .key_on_A(key_on_A),
        .key_off_A(key_off_A),
        .note_A(note_A),
        .key_on_B(key_on_B),
        .key_off_B(key_off_B),
        .note_B(note_B),
        .tub_control_B(tub_control_B),
        .key_on_C(key_on_C),
        .key_off_C(key_off_C),
        .note_C(note_C),
        .tub_sel_C(tub_sel_C),
        .tub_control_C(tub_control_C),
        .key_on_out(key_on),
        .key_off_out(key_off),
        .tub_control(tub_control),
        .tub_sel(tub_sel),
        .tub_sel_user(tub_sel_user),
        .note_out(note),
        .mode(mode)
    );

//output modules
    LED led(
        .clk(clk),
        .rst_n(rst_n),
        .note(note),
        .key_notes(key_notes),
        .light(light)
    );
```

```
Buzzer buzzer (
    .clk(clk),
    .rst_n(rst_n),
    .key_on_in(key_on),
    .key_off_in(key_off),
    .note_in(note),
    .note_out(selected_note),
    .octave_button(octave_button),
    .octave_out(current_octave),
    .speaker(speaker)
);

 VGADisplay vga_display(
    .clk(clk),
    .sys_rst_n(rst_n),
    .mode(mode),
    .key_on_in(key_on),
    .key_off_in(key_off),
    .note_in(note),
    .selected_note(selected_note),
    .track_number_auto(track_number_auto),
    .track_number_learning(track_number_learning),
    .current_octave(current_octave),
    .reaction_rate(reaction_rate),
    .finish(finish),
    .hsync(hsync),
    .vsync(vsync),
    .vga_rgb(vga_rgb)
);

endmodule
```

1. **KeyController Module**

This module is responsible for managing the key inputs of the electronic piano. It receives signals representing the state of individual keys (key_notes) and produces outputs indicating when a key is pressed (key_on_out), when a key is released (key_off_out), and the corresponding note value (note_out). The module uses instances of the Key module (which will be introduced next) to handle each individual key.

Inputs:

- clk: Clock signal.
- rst_n: Active-low reset signal.

- key_notes: 7-bits input vector representing the state of individual keys.

```
module KeyController(
    input wire clk,
    input wire rst_n,
    input wire [`KEY_NOTES_WIDTH - 1:0] key_notes,
```

Outputs:

- key_on_out: Output indicating when a key is pressed.
- key_off_out: Output indicating when a key is released.
- note_out: 7-bits output representing the decoded note value based on the pressed key.

```
    output wire key_on_out,
    output wire key_off_out,
    output reg [`NOTE_OUT_WIDTH -1:0] note_out
    );
```

Wires:

- key_on: 7-bits vector of signals indicating which keys are pressed.
- key_off: 7-bits vector of signals indicating which keys are released.

```
    wire [`KEY_ON_WIDTH - 1:0] key_on;
    wire [`KEY_OFF_WIDTH - 1:0] key_off;
```

The Key module is instantiated multiple times to handle each individual key. It takes the clock signal, reset signal, and the state of a specific key (key_note) as inputs, and provides outputs indicating when the key is pressed (key_on_out) or released (key_off_out).

```
// instantiate each key note
Key key_note_1 (
    .clk(clk),
    .rst_n(rst_n),
    .key_note(key_notes[0]),
    .key_on_out(key_on[0]),
    .key_off_out(key_off[0])
);

Key key_note_2 (
    .clk(clk),
    .rst_n(rst_n),
    .key_note(key_notes[1]),
    .key_on_out(key_on[1]),
    .key_off_out(key_off[1])
);

Key key_note_3 (
    .clk(clk),
    .rst_n(rst_n),
    .key_note(key_notes[2]),
    .key_on_out(key_on[2]),
    .key_off_out(key_off[2])
);

Key key_note_4 (
    .clk(clk),
    .rst_n(rst_n),
    .key_note(key_notes[3]),
    .key_on_out(key_on[3]),
    .key_off_out(key_off[3])
);
```

wire [(7 - 1):0]

```
Key key_note_5 (
    .clk(clk),
    .rst_n(rst_n),
    .key_note(key_notes[4]),
    .key_on_out(key_on[4]),
    .key_off_out(key_off[4])
);

Key key_note_6 (
    .clk(clk),
    .rst_n(rst_n),
    .key_note(key_notes[5]),
    .key_on_out(key_on[5]),
    .key_off_out(key_off[5])
);

Key key_note_7 (
    .clk(clk),
    .rst_n(rst_n),
    .key_note(key_notes[6]),
    .key_on_out(key_on[6]),
    .key_off_out(key_off[6])
);
```

Note Decoding Logic:

The module includes a case statement to decode the correct note based on the combination of pressed keys (key_on). The resulting note value is assigned to note_out.

```
// decode the correct note
always @(*) begin
    case (key_on)
        7'b000_0001: note_out = 6'd1;
        7'b000_0010: note_out = 6'd2;
        7'b000_0100: note_out = 6'd3;
        7'b000_1000: note_out = 6'd4;
        7'b001_0000: note_out = 6'd5;
        7'b010_0000: note_out = 6'd6;
        7'b100_0000: note_out = 6'd7;
        default: note_out = 6'd0;
    endcase
end
```

Output assignment:

- key_on_out is assigned 1 if any key is pressed, 0 otherwise
- key_off_out is assigned 1 if any key is released, 0 otherwise.

```
assign key_on_out = (key_on == 7'd0) ? 0 : 1;
assign key_off_out = (key_off == 7'd0) ? 0 : 1;
```

1.1. Key Module

This module is designed to handle the state transitions and edge detection for an individual key in the electronic piano. It includes a state machine to determine whether the key is pressed or released, and it employs clock synchronization and edge detection logic.

Inputs:

- clk: Clock signal.
- rst_n: Active-low reset signal.
- key_note: Input representing the state of an individual key.

```
module Key(
    input wire clk,
    input wire rst_n,
    input wire key_note,
```

Outputs:

- key_on_out: Output indicating when the key is pressed.
- key_off_out: Output indicating when the key is released.

```
output reg key_on_out,
output reg key_off_out
);
```

Wires:

- pos_edge: Signal indicating the positive edge of the input key signal.
- neg_edge: Signal indicating the negative edge of the input key signal.

```
wire pos_edge;  // record edge value for switching status
wire neg_edge;
```

Regs:

- state: 2-bits state variable for the state machine.
- key_sig1, key_sig2, key_sig3: Synchronized key signals for edge detection.
- counter_en: Signal to enable the counter for clock delay.
- counter: 32-bits counter for clock delay.

```
reg [`STATE_WIDTH -1:0] state;
reg key_sig1, key_sig2, key_sig3;  // edge change is asynchronous signal
reg counter_en;
reg [`KEY_COUNTER_WIDTH -1:0] counter;  // for clock delay
```

Synchronization and Edge Detection Logic:

- key_sig1, key_sig2, and key_sig3 are used to synchronize the key signal with the clock. Edge detection is performed by checking for transitions in these synchronized signals.
- pos_edge and neg_edge are determined based on the synchronized key signals.

```
// synchronize key signal with the clock
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        key_sig1 <= 1'b0;
        key_sig2 <= 1'b0;
        key_sig3 <= 1'b0;
    end
    else begin
        key_sig1 <= key_note;
        key_sig2 <= key_sig1;
        key_sig3 <= key_sig2;
    end
end

assign pos_edge = key_sig2 & (!key_sig3);
assign neg_edge = (!key_sig2) & key_sig3;
```

Counter for Clock Delay:

- A counter (counter) is used for introducing a clock delay. It is incremented or reset based on the clock and reset signals.

```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n)
        counter <= `KEY_COUNTER_INIT_STATE;
    else if (counter_en)
        counter <= counter + `KEY_COUNTER_INIT_STATE +1;
    else
        counter <= `KEY_COUNTER_INIT_STATE;
end
```

State Machine:

- The module employs a state machine to handle the transitions between different states (IDLE_KEY, FILTER0_KEY, DOWN_KEY, FILTER1_KEY). The states control the output signals (key_on_out and key_off_out) based on key presses and releases.
- The state transitions are determined by edge conditions and counter values.

```
// state machine
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        state <= `IDLE_KEY;
        counter_en <= 1'b0;
        key_on_out <= 1'b0;
        key_off_out <= 1'b0;
    end
    else begin
        case (state)
            `IDLE_KEY:
                begin
                    key_on_out <= 1'b0;
                    key_off_out <= 1'b0;
                    if (neg_edge) begin
                        counter_en <= 1'b1;
                        state <= `FILTER0_KEY;
                    end
                    else
                        state <= `IDLE_KEY;
                end
            `FILTER0_KEY:
                begin
                    if (counter == `MAX_COUNT) begin
                        key_on_out <= 1'b1;
                        counter_en <= 1'b0;
                        state <= `DOWN_KEY;
                    end
                    else if (pos_edge) begin
                        counter_en <= 1'b0;
                        state <= `IDLE_KEY;
                    end
                    else
                        state <= `FILTER0_KEY;
                end
```

```
            `DOWN_KEY:
                begin
                    key_on_out <= 1'b0; // output one clock pulse, change later
                    if (pos_edge) begin
                        counter_en <= 1'b1;
                        state <= `FILTER1_KEY;
                    end
                    else
                        state <= `DOWN_KEY;
                end
            `FILTER1_KEY:
                begin
                    if (counter == `MAX_COUNT) begin
                        key_off_out <= 1'b1;
                        counter_en <= 1'b0;
                        state <= `IDLE_KEY;
                    end
                    else if (neg_edge) begin
                        counter_en <= 1'b0;
                        state <= `DOWN_KEY;
                    end
                    else
                        state <= `FILTER1_KEY;
                end
            default:
                begin
                    state <= `IDLE_KEY;
                    counter_en <= 1'b0;
                    key_on_out <= 1'b0;
                    key_off_out <= 1'b0;
                end
        endcase
    end
end

endmodule
```

2. AutoPLay Module

The AutoPlay module is responsible for the automatic playback functionality of the electronic piano. It integrates key controllers for speed and song selection, a beat controller to manage the rhythm, a music playback module, and a BCD to 7-segment converter for display.

Inputs:

- clk: Clock signal.
- rst_n: Active-low reset signal.

- set_speed: Input signal for setting the playback speed.
- set_song: Input signal for selecting the song.

```
module AutoPlay(
    input wire clk,
    input wire rst_n,
    input wire set_speed,
    input wire set_song,
```

Outputs:

- key_on_out: Output indicating when a key is pressed during automatic playback.
- key_off_out: Output indicating when a key is released during automatic playback.
- note: 6-bit output representing the note being played during automatic playback.
- seg: 8-bit output for the 7-segment display.
- track_number_auto: 2-bit output indicating the track number being played during automatic playback.

```
output wire key_on_out,
output wire key_off_out,
output wire[`NOTE_WIDTH - 1:0] note,
output wire[`SEG_WIDTH -1:0] seg,
output wire [`TRACK_NUM_AUTO_WIDTH -1:0] track_number_auto
```

Wires:

- finish: Signal indicating the completion of music playback.
- sameNote: Signal indicating if the note is the same as the previous one.
- index: 6-bit signal for managing the index of the selected song.
- song_num: 2-bit constant representing the number of songs.
- speed: 2-bit constant representing the speed setting for playback.
- key_on_speed, key_off_speed: Signals from the speed key controller.
- key_on_song, key_off_song: Signals from the song key controller.
- song_BCD_song: 4-bit BCD code representing the selected song for display.

```
wire finish;
wire sameNote;
wire[`NOTE_WIDTH - 1:0] index;
wire[`SONG_NUM_WIDTH -1:0] song_num = `SONGNM;
wire[`SPEED_WIDTH -1:0] speed = `SPEED;
wire key_on_speed;
wire key_off_speed;
wire key_on_song;
wire key_off_song;
wire[`SONG_BCD_CODE_WIDTH -1:0] song_BCD_code;
```

Instantiated Modules:

- Key (speed and song key controllers): Modules for controlling the speed and song selection keys.
- BeatController: Module for managing the beat and rhythm during automatic playback.
- Music: Module for playing back the selected song.
- BCDto7Segment: Module for converting the BCD code to a 7-segment display.

```
Key key_speed(clk, rst_n, set_speed, key_on_speed, key_off_speed);  //speed key controller
Key key_song(clk, rst_n, set_song, key_on_song, key_off_song);      //song key controller
BeatController bc(clk, rst_n, key_on_song, key_off_song, key_on_speed, key_off_speed, key_on_out, key_off_out, index);  //beat controller
Music music1(clk, rst_n, key_on_song, key_off_song, index, song_BCD_code, note, track_number_auto, finish);  //music playback module
BCDto7Segment bcd_7segment(song_BCD_code, seg);  // BCD to 7-segment converter
```

2.1. BeatController Module

The BeatController module is responsible for managing the beat and rhythm during the automatic playback of the electronic piano. It controls the key press and release signals based on the specified speed modes (normal, fast, slow) and transitions between idle and set states.

Inputs:

- clk: Clock signal.
- rst_n: Active-low reset signal.
- key_on_song: Input indicating when a song is selected.
- key_off_song: Input indicating when a song is deselected.
- key_on: Input indicating when a key is pressed.
- key_off: Input indicating when a key is released.

```
module BeatController(
    input wire clk,
    input wire rst_n,
    input wire key_on_song,
    input wire key_off_song,
    input wire key_on,
    input wire key_off,
```

Outputs:

- key_on_out: Output indicating when a key is pressed during the beat cycle.
- key_off_out: Output indicating when a key is released during the beat cycle.
- index: 10-bit output indicating the index of the note during automatic playback.

```
    output reg key_on_out,
    output reg key_off_out,
    output reg[`INDEX_WIDTH -1:0] index    //index of note
    );
```

Regs:

- count: 32-bit counter for managing the beat cycle.
- max_count: 32-bit maximum count for the beat cycle based on the speed mode.
- max_IDLE_count: 32-bit maximum count for the idle state between beat cycles.
- speed_mode: 2-bit mode representing the speed of the beat cycle.
- state: State variable for the state machine controlling transitions between idle and set states.

```
reg[`COUNT_WIDTH -1:0] count;
reg[`MAX_COUNT_WIDTH -1:0] max_count = `NORMAL;
reg[`MAX_IDLE_COUNT_WIDTH -1:0] max_IDLE_count = `NORMAL_IDLE;
reg[`SPEED_MODE_WIDTH -1:0] speed_mode = `NORMAL_MODE;
```

Beat cycle:

- The beat cycle is managed using the count variable, which increments during the idle state and resets after completing a beat cycle.
- The duration is determined by max_count and max_IDLE_count, which are set based on the selected speed mode.

```
always @ (posedge clk, negedge rst_n,posedge key_on_song) begin
        if(~rst_n||key_on_song) begin
            count <= 32'd0;
            key_on_out <= 1'b0;
            key_off_out <= 1'b0;
            index <= 10'd0;
        end
        else begin
             if(count<max_IDLE_count) begin
                key_on_out <= 1'b0;
                key_off_out <= 1'b1;
                count <= count+32'd1;

            end
            else if(count >= max_IDLE_count && count < max_count+max_IDLE_count)begin
                key_on_out <= 1'b1;
                key_off_out<= 1'b0;
                count <= count+32'd1;
            end
            else if(count >= max_count+max_IDLE_count) begin
                index <= index + 10'd1;
                count <= 32'd0;
            end

        end
    end
```

State machine Logic:

- The module includes a state machine (state) with states IDLE and SET.
- In the IDLE state, the module waits for the selection of a song (key_on_song). Once a song is selected, it transitions to the SET state and determines the speed mode based on the selected song.
- In the SET state, the module manages the beat cycle. It controls the key press and release signals (key_on_out and key_off_out) based on the count and transitions back to the IDLE state after completing a beat cycle.

```
    reg state;
    always @(posedge clk or negedge rst_n) begin
            if (!rst_n) begin
                state <= `IDLE;
            end
            else begin
                case (state)
                    `IDLE:
                       begin
                           if (key_on) begin
                               state <= `SET;
                               case (speed_mode)
                                   2'd0: begin speed_mode <= `NORMAL_MODE;max_count <= `NORMAL; max_IDLE_count <= `NORMAL_IDLE; end
                                   2'd1: begin speed_mode <= `FAST_MODE;max_count <= `FAST; max_IDLE_count <= `FAST_IDLE; end
                                   2'd2: begin speed_mode <= `SLOW_MODE;max_count <= `SLOW; max_IDLE_count <= `SLOW_IDLE; end
                                   default: begin speed_mode <= `NORMAL_MODE;max_count <= `NORMAL; max_IDLE_count <= `NORMAL_IDLE; end
                               endcase
                           end
                           else
                               state <= `IDLE;
                       end
                    `SET:
                       begin
                           if (key_off) begin
                               state <= `IDLE;
                           end
                           else
                               state <= `SET;
                       end
                   default:
                       state <= `IDLE;
                endcase
            end
        end
endmodule
```

2.2. Music Module

This module is responsible for managing the music playback during the automatic mode. It handles the selection of songs, note sequencing, and tracks the progress of the playback.

Inputs:

- clk: Clock signal.
- rst_n: Active-low reset signal.
- key_on: Input indicating when a key is pressed.
- key_off: Input indicating when a key is released.
- index: 10-bit input indicating the current index of the note being played during automatic playback.

```
module Music(
    input wire clk,
    input wire rst_n,
    input wire key_on,
    input wire key_off,
    input wire[`INDEX_WIDTH -1:0] index,
```

Outputs:

- song_BCD_code: 4-bit output representing the BCD code corresponding to the selected song.
- note_out: 6-bit output representing the current note being played.
- track_number: 2-bit output indicating the current track (song) being played.
- finish: Output indicating the completion of music playback for the selected song.

```
output reg[`SONG_BCD_CODE_WIDTH -1:0] song_BCD_code,
output reg[`NOTE_OUT_WIDTH -1:0] note_out,
output wire [`TRACK_NUM_WIDTH-1:0] track_number,
output reg finish
);
```

Regs:

- song_mode: 2-bit mode representing the currently selected song.
- note_sequence: 301-bit sequence of notes for the selected song.
- state: State variable for the state machine controlling transitions between idle and set states.

```
reg[`SONG_MODE_WIDTH -1:0] song_mode = `LITTLE_STAR_MODE;
reg[`NOTE_SEQUENCE_WIDTH -1:0] note_sequence = 0;
```

Song and note handling:

- The module includes a note_sequence for each song, and the note_out output is determined based on the index and the selected song's note sequence.
- The song_mode variable represents the currently selected song, and the track_number output is set accordingly.
- The song_BCD_code output is set based on the selected song.

```
assign track_number = song_mode;
always @(*)begin
        case(song_mode)
            `LITTLE_STAR_MODE: begin note_sequence = `LITTLE_STAR; song_BCD_code = 4'd2; end
            `ODE_TO_JOY_MODE: begin note_sequence = `ODE_TO_JOY; song_BCD_code = 4'd3; end
            `TWO_TIGERS_MODE: begin note_sequence= `TWO_TIGERS; song_BCD_code = 4'd1; end
        endcase
        note_out = note_sequence[index * 6 +5 -:6];
end
```

State machine:

- The module includes a state machine (state) with states IDLE and SET.
- In the IDLE state, the module waits for a key press (key_on) to transition to the SET state. It then cycles through the available songs in a predetermined order.
- In the SET state, the module waits for a key release (key_off) to transition back to the IDLE state.

```
reg state;
always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            state <= `IDLE;
        end
        else begin
            case (state)
                `IDLE:
                    begin
                        if (key_on) begin
                            state <= `SET;
                            case (song_mode)
                                `LITTLE_STAR_MODE: song_mode <= `ODE_TO_JOY_MODE;
                                `TWO_TIGERS_MODE: song_mode <= `LITTLE_STAR_MODE;
                                `ODE_TO_JOY_MODE: song_mode <= `TWO_TIGERS_MODE;
                                default: song_mode <= `LITTLE_STAR_MODE;
                            endcase
                        end
                        else
                            state <= `IDLE;
                    end
```

```
                `SET:
                    begin
                        if (key_off) begin
                            state <= `IDLE;
                        end
                        else
                            state <= `SET;
                    end
                default:
                    state <= `IDLE;
            endcase
        end
    end
```

Completion detection:

- The finish output is set based on the current index and the length of the note sequence for the selected song. It indicates the completion of music playback for the selected song.

```
    always@(*)begin
        case(song_mode)
            `LITTLE_STAR_MODE:
            begin
                  if( index == `LITTLE_STAR_LENGTH+1)finish = 1;
            else finish = 0;
            end
            `TWO_TIGERS_MODE:
            begin
                  if( index == `TWO_TIGERS_LENGTH-1)finish = 1;
                  else finish = 0;
             end
             `ODE_TO_JOY_MODE:
             begin
                  if( index == `ODE_TO_JOY_LENGTH-1)finish = 1;
                  else finish = 0;
             end
        endcase
    end
endmodule
```

2.3. BCDto7Segment Module

The BCDto7Segment module is responsible for converting a 4-bit Binary-Coded Decimal (BCD) input into a 7-segment display format. It maps each BCD digit to the corresponding 7-segment display pattern.

Input:

- BCD: 4-bit Binary-Coded Decimal input representing a digit (0 to 9).

Output:

- segment: 8-bit output representing the 7-segment display pattern for the input BCD digit.

```
module BCDto7Segment(
    input[`BCD_WIDTH -1:0] BCD,
    output reg[`SEG_WIDTH -1:0] segment
    );
```

7-Segment Display Mapping:

- The module uses a case statement to map each possible BCD input (0 to 9) to the corresponding 7-segment display pattern.
- The segment output is set based on the BCD input, where each bit in the 8-bit output represents a segment of the 7-segment display.

```
always @ * begin
case(BCD)
4'b0000: segment = 8'b1111_1100; //"0" : abcdef_ _
4'b0001: segment = 8'b0110_0000; //"1": _bc_ _ _ _ _ _
4'b0010: segment = 8'b1101_1010; //"2": ab_de_g_
4'b0011: segment = 8'b1111_0010; //"3": abcd_ _ g _
4'b0100: segment = 8'b0110_0Z110; //"4": _bc _ _fg_
4'b0101: segment = 8'b1011_0110; //"5": a_cd_fg_
4'b0110: segment = 8'b1011_1110; //"6": a_cdefg_
4'b0111: segment = 8'b1110_0000; //"7": abc_ _ _ _ _
4'b1000: segment = 8'b1111_1110; //"8": abcdefg_
4'b1001: segment = 8'b1110_0110; //"9": abc_ _ fg_
default:
segment = 8'b1001_1110; //"E": a_ _ defg_
endcase
end
endmodule
```

3. ModeSelector Module

This module is responsible for managing the selection of different modes (Free Mode, Auto Mode, Learning Mode). It interfaces with the key controller and coordinates the transition between modes based on user input.

Inputs:

- clk: Clock signal.

- rst_n: Active-low reset signal.
- set_mode: Input signal indicating the request to change the mode.
- key_on_A, key_off_A, note_A (6-bit): Input signals from the Auto Mode.
- key_on_B, key_off_B, note_B (6-bit), tub_control_B (8-bit): Input signals from the Free Mode (AutoPlay module).
- key_on_C, key_off_C, note_C (6-bit), tub_sel_C, tub_control_C (8-bit): Input signals from the Learning Mode.

```
module ModeSelector(
    input clk,
    input rst_n,
    input set_mode,
    input key_on_A,
    input key_off_A,
    input[`NOTE_A_WIDTH -1:0] note_A,
    input key_on_B,
    input key_off_B,
    input[`NOTE_B_WIDTH -1:0] note_B,
    input [`TUB_CONTROL_B -1:0]tub_control_B,
    input key_on_C,
    input key_off_C,
    input[`NOTE_C_WIDTH -1:0] note_C,
    input tub_sel_C,
    input [`TUB_CONTROL_C -1:0]tub_control_C,
```

Outputs:

- key_on_out, key_off_out, note_out (6-bit): Outputs for key controller signals based on the selected mode.
- tub_control: 8-bit output for tube control based on the selected mode.
- tub_sel: Output indicating tube selection based on the selected mode.
- tub_sel_user: Output indicating user-selected tube in Learning Mode.
- mode : 2-bit output indicating the current mode.

```
output reg key_on_out,
output reg key_off_out,
output reg[`TUB_CONTROL -1:0]tub_control,
output reg tub_sel,
output reg tub_sel_user,
output reg[`NOTE_OUT_WIDTH -1:0] note_out,
output reg[`MODE_MODE_SELECTOR_WIDTH -1:0] mode
);
```

Regs:

- state: Internal state variable for the state machine controlling mode transitions.

Wires:

- key_on, key_off: Internal signals for key on and key off.

Mode handling:

- The mode variable represents the current mode, and it transitions between Free Mode, Auto Mode, and Learning Mode.
- Based on the selected mode, the module sets the output signals (key_on_out, key_off_out, note_out, tub_control, tub_sel, tub_sel_user) accordingly.

```
Key key(clk,rst_n,set_mode,key_on,key_off); //key controller instance
always @(posedge clk,posedge key_on)begin
    if(key_on) begin
        key_on_out = 1'b0;
        key_off_out = 1'b1;
        note_out = 6'd0;
        tub_sel = 1'b0;
        tub_control = 8'd0;
    end
    else begin
        case(mode)
        `FREE_MODE: begin key_on_out = key_on_B; key_off_out = key_off_B; note_out = note_B; tub_sel = 1'b1;tub_control = tub_control_B;tub_sel_user = 0;end
        `AUTO_MODE: begin key_on_out = key_on_A; key_off_out = key_off_A; note_out = note_A; tub_sel = 1'b0;tub_control = tub_control_B;tub_sel_user = 0;end
        `LEARNING_MODE: begin  key_on_out = key_on_C; key_off_out = key_off_C; note_out = note_C; tub_sel = tub_sel_C;tub_control = tub_control_C; tub_sel_user = 1;end
        endcase
    end
end
```

Mode transition:

- IDLE and SET states.
- In the IDLE state, the module waits for a key press (key_on) and changes the mode based on the current mode.
- The SET state is entered when a key release (key_off) is detected, and it transitions back to IDLE.

```
reg state;
always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            state <= `IDLE;
        end
        else begin
            case (state)
                `IDLE:
                    begin
                        if (key_on) begin
                            state <= `SET;
                            case (song_mode)
                                `LITTLE_STAR_MODE: song_mode <= `ODE_TO_JOY_MODE;
                                `TWO_TIGERS_MODE: song_mode <= `LITTLE_STAR_MODE;
                                `ODE_TO_JOY_MODE: song_mode <= `TWO_TIGERS_MODE;
                                default: song_mode <= `LITTLE_STAR_MODE;
                            endcase
                        end
                        else
                            state <= `IDLE;
                    end

                `SET:
                    begin
                        if (key_off) begin
                            state <= `IDLE;
                        end
                        else
                            state <= `SET;
                    end
                default:
                    state <= `IDLE;
            endcase
        end
    end
```

4. LED Module

The LED module is responsible for controlling the LED lights. The LEDs represent the keys that are pressed, and their illumination is based on the notes played or the buttons pressed.

Inputs:

- clk: Clock signal.
- rst_n: Active-low reset signal.
- note: 6-bit input signal indicating the note being played.
- key_notes: 7-bit input signal representing the functionality of buttons (button functionality).

Output:

- light: 7-bit output signal representing the LED lights in one-hot code format.

```
module LED(
    input wire clk,
    inout wire rst_n,
    input wire[`NOTE_WIDTH -1:0] note,
    input wire [`KEY_NOTES_WIDTH -1:0] key_notes, //button functionality
    output reg[`LIGHT_WIDTH -1:0] light //one_hot code (LED)
    );
```

LED Illumination logic:

- The module uses a case statement to determine the illumination pattern of the LEDs based on the note signal.
- If a note is played (note is not equal to 0), the LEDs corresponding to the played note are illuminated in a one-hot pattern.

```
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            light <= 7'b0000000;  // reset the lights when rst_n is inactive
        end else begin
            if (note != 0) begin
                case (note)
                    6'd1: light <= 7'b0000001;
                    6'd2: light <= 7'b0000010;
                    6'd3: light <= 7'b0000100;
                    6'd4: light <= 7'b0001000;
                    6'd5: light <= 7'b0010000;
                    6'd6: light <= 7'b0100000;
                    6'd7: light <= 7'b1000000;

                    6'd8: light <= 7'b0000001;
                    6'd9: light <= 7'b0000010;
                    6'd10: light <= 7'b0000100;
                    6'd11: light <= 7'b0001000;
                    6'd12: light <= 7'b0010000;
                    6'd13: light <= 7'b0100000;
                    6'd14: light <= 7'b1000000;

                    6'd15: light <= 7'b0000001;
                    6'd16: light <= 7'b0000010;
                    6'd17: light <= 7'b0000100;
                    6'd18: light <= 7'b0001000;
                    6'd19: light <= 7'b0010000;
                    6'd20: light <= 7'b0100000;
                    6'd21: light <= 7'b1000000;
                    default: light <= 7'b0000000;
                endcase
            end else begin
                // Free mode
                light <= ~key_notes;
            end
        end
    end
```

5. Buzzer Module

This module is responsible for generating sound output through a buzzer. It takes input signals related to key presses and controls the frequency of the sound produced.

Inputs:

- clk: Clock signal.
- rst_n: Active-low reset signal.
- key_on_in: Input signal indicating the activation of a key.
- key_off_in: Input signal indicating the deactivation of a key.
- note_in: 6-bit input signal representing the note to be played.
- octave_button - Input signal indicating the octave change button.

```
module Buzzer (
    input wire clk,
    input wire rst_n,
    input wire key_on_in,
    input wire key_off_in,
    input wire [`BUZZER_NOTE_IN_WIDTH -1:0] note_in,  // note (Input 1 outputs a signal for 'do, 2 for 're, 3 for 'mi, 4, and so on)
    input wire octave_button, // octave button to switch between octaves
```

Outputs:

- note_out: 6-bit output signal representing the note being played.
- octave_out: 2-bit output signal representing the current octave setting.
- speaker: Output signal controlling the buzzer to produce sound.

```
    output reg [`NOTE_OUT_WIDTH -1:0] note_out,
    output wire [`CURRENT_OCTAVE_WIDTH -1:0] octave_out,
    output wire speaker     // buzzer output signal
    );
```

Wires and Regs:

- currentOctave: Internal signal representing the current octave setting.
- state: Internal signal representing the state of the state machine.
- counter_en: Internal signal controlling the counter enable.
- counter_max: Internal signal representing the maximum counter value for PWM generation.
- counter: Internal signal representing the current counter value for PWM generation.
- pwm: Internal signal representing the PWM signal for sound generation.
- octave_button_prev: Internal signal representing the previous state of the octave button.
- key_on_tmp, key_off_tmp: Internal signals for key on and key off, synchronized with the clock.

```
reg[`CURRENT_OCTAVE_WIDTH -1:0] currentOctave;
assign octave_out = currentOctave;
localparam IDLE = 1'b0, BUZZ = 1'b1;
wire [`NOTES_WIDTH -1:0] notes [`NOTES_SIZE -1:0];
reg state;
reg counter_en;
reg [`COUNTER_MAX_WIDTH -1:0] counter_max;
reg [`COUNT_WIDTH -1:0] counter;
reg pwm;
reg octave_button_prev;
wire key_on_tmp,key_off_tmp;
```

Note frequencies:

The module uses an array “notes” to store the frequencies corresponding to different notes in different octaves.

```
assign notes[0] = `BUZZER_NOTE_0;
// frequencies of do, re, mi, fa, so, la, si
assign notes[1] = `BUZZER_NOTE_1;
assign notes[2] = `BUZZER_NOTE_2;
assign notes[3] = `BUZZER_NOTE_3;
assign notes[4] = `BUZZER_NOTE_4;
assign notes[5] = `BUZZER_NOTE_5;
assign notes[6] = `BUZZER_NOTE_6;
assign notes[7] = `BUZZER_NOTE_7;

//frequencies of low do, low re, low mi...
assign notes[8] = `BUZZER_NOTE_8;
assign notes[9] = `BUZZER_NOTE_9;
assign notes[10] = `BUZZER_NOTE_10;
assign notes[11] = `BUZZER_NOTE_11;
assign notes[12] = `BUZZER_NOTE_12;
assign notes[13] = `BUZZER_NOTE_13;
assign notes[14] = `BUZZER_NOTE_14;

//fequencies of high do, high re, high mi...
assign notes[15] = `BUZZER_NOTE_15;
assign notes[16] = `BUZZER_NOTE_16;
assign notes[17] = `BUZZER_NOTE_17;
assign notes[18] = `BUZZER_NOTE_18;
assign notes[19] = `BUZZER_NOTE_19;
assign notes[20] = `BUZZER_NOTE_20;
assign notes[21] = `BUZZER_NOTE_21;
```

Clock divider and PWM generation:

- The module includes a clock divider to control the PWM generation for sound.
- The pwm signal is toggled based on the counter value, creating a PWM waveform.

```
// clock divider
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        pwm <= 1'b0;
        counter <= 32'd0;
    end
    else begin
        if (counter == counter_max) begin
            pwm <= ~pwm;
            counter <= 32'd0;
        end
        else if (counter_en)
            counter <= counter + 32'd1;
        else
            counter <= 32'd0;
      end
      end
```

State machine:

- The module uses a state machine with states IDLE and BUZZ to control the sound generation process.

```
always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            state <= IDLE;
            counter_en <= 1'b0;
        end
        else begin
            case (state)
                IDLE:
                    begin
                        if (key_on_in) begin
                            state <= BUZZ;
                            note_out <= note_in;
                            case ({currentOctave, note_in})
                                  {`OCTAVE_LOW, 6'd1}: counter_max <= notes[8];
                                  {`OCTAVE_LOW, 6'd2}: counter_max <= notes[9];
                                  {`OCTAVE_LOW, 6'd3}: counter_max <= notes[10];
                                  {`OCTAVE_LOW, 6'd4}: counter_max <= notes[11];
                                  {`OCTAVE_LOW, 6'd5}: counter_max <= notes[12];
                                  {`OCTAVE_LOW, 6'd6}: counter_max <= notes[13];
                                  {`OCTAVE_LOW, 6'd7}: counter_max <= notes[14];

                                  {`OCTAVE_NORMAL, 6'd1}: counter_max <= notes[1];
                                  {`OCTAVE_NORMAL, 6'd2}: counter_max <= notes[2];
                                  {`OCTAVE_NORMAL, 6'd3}: counter_max <= notes[3];
                                  {`OCTAVE_NORMAL, 6'd4}: counter_max <= notes[4];
                                  {`OCTAVE_NORMAL, 6'd5}: counter_max <= notes[5];
                                  {`OCTAVE_NORMAL, 6'd6}: counter_max <= notes[6];
                                  {`OCTAVE_NORMAL, 6'd7}: counter_max <= notes[7];


                                  {`OCTAVE_HIGH, 6'd1}: counter_max <= notes[15];
                                  {`OCTAVE_HIGH, 6'd2}: counter_max <= notes[16];
                                  {`OCTAVE_HIGH, 6'd3}: counter_max <= notes[17];
                                  {`OCTAVE_HIGH, 6'd4}: counter_max <= notes[18];
                                  {`OCTAVE_HIGH, 6'd5}: counter_max <= notes[19];
                                  {`OCTAVE_HIGH, 6'd6}: counter_max <= notes[20];
                                  {`OCTAVE_HIGH, 6'd7}: counter_max <= notes[21];
                                 default: counter_max <= notes[0];
                            endcase
                        end
                        else
                            state <= IDLE;
                    end
                BUZZ:
                    begin
                        counter_en <= 1'b1;
                        if (key_off_in) begin
                            counter_en <= 1'b0;
                            state <= IDLE;
                        end
                        else
                            state <= BUZZ;
                    end
                default:
                    state <= IDLE;
            endcase
        end
    end
```

Octave change logic:

- The module includes logic to change the current octave based on the octave_button input.

```
    assign speaker = pwm;
    reg statel;
        localparam idle = 1'b0, set = 1'b1;
        always @(posedge clk or negedge rst_n) begin
            if (!rst_n) begin
                statel <= idle;
                currentOctave <= `OCTAVE_NORMAL;
            end
            else begin
                case (statel)
                    idle:
                        begin
                            if (key_on_tmp) begin
                                case (currentOctave)
                                    `OCTAVE_NORMAL: begin currentOctave <= `OCTAVE_HIGH; end
                                    `OCTAVE_HIGH: begin currentOctave <= `OCTAVE_LOW; end
                                    `OCTAVE_LOW: begin currentOctave <= `OCTAVE_NORMAL; end
                                endcase
                                statel <= set;
                            end
                            else
                                statel <= idle;
                        end
                    set:
                        begin
                            if (key_off_tmp) begin
                                statel <= idle;
                            end
                            else
                                statel <= set;
                        end
                    default:
                        statel <= idle;
                endcase
            end
        end
endmodule
```

6. **Counting Module**

This module is a simple counter that increments its output value on each rising edge of the clock signal when the enable signal is active. It has the ability to reset to an initial value when the enable signal is inactive.

Inputs:

- clk: Clock signal.
- enable: Input signal that controls the counting operation.

Output:

- outputCount: 32-bit output signal representing the current count value.

```
module Counting(
     input clk,
     input enable,
     output reg[`OUTPUTCOUNT_WIDTH -1:0] outputCount
     );
```

Operation:

- On each rising edge of the clock signal (posedge clk), the module checks the enable signal.
- If enable is inactive (!enable), the outputCount is reset to the initial value defined by OUTPUTCOUNT_INIT_VALUE.
- If enable is active, the outputCount is incremented by the value defined by OUTPUT_INCR.

```
    always @(posedge clk)begin
        if(!enable) begin
            outputCount <= `OUTPUTCOUNT_INIT_VALUE; //reset
        end
        else begin
            outputCount <= outputCount + `OUTPUT_INCR; //increment
        end
    end
endmodule
```

7. **FrequencyDivider Module**

This module is designed to divide the frequency of an input clock signal (clk) to generate a slower clock signal (slow_clk). It utilizes a counter to achieve the frequency division.

Inputs:

- clk: Clock signal.
- rst_n: Active-low asynchronous reset signal.

Outputs:

- slow_clk - Output clock signal with a frequency lower than the input clock.

Reg:

- counter: 28-bit counter used for frequency division.

```
module FrequencyDivider(
    input clk,
    input rst_n,
    output reg slow_clk
    );
    reg [`FREQUENCY_DIVIDER_COUNT -1:0] counter;
```

Operations:

- On each rising edge of the input clock signal (posedge clk), the module checks the active-low asynchronous reset signal rst_n.
- If rst_n is inactive (~rst_n), the slow_clk is initialized to the value specified by SLOW_CLK_INIT_VALUE, and the counter is reset to the value specified by COUNTER_INIT_VALUE.
- If rst_n is active, the counter is incremented by 1.
- When the counter reaches the value specified by FREQUENCY_DIVIDER, the counter is reset, and the slow_clk is toggled (slow_clk <= ~slow_clk).

```
always @(posedge clk, negedge rst_n) begin
    if(~rst_n)begin
        slow_clk <= `SLOW_CLK_INIT_VALUE; //initialize slow clk
        counter <= `COUNTER_INIT_VALUE; //reset counter
    end
    else begin
        counter <= counter + 1;
        if (counter == `FREQUENCY_DIVIDER)
            begin
                counter <= `COUNTER_INIT_VALUE; //reset
                slow_clk <= ~slow_clk; //toggle
            end
    end
end
endmodule
```

8. **LearningMode module**

This module is designed to operate in the learning mode of the system. It involves key input handling, music checking, music playback, key controller, reaction rating, scoring, and recording functionalities.

Inputs:

- clk: Clock signal.
- rst_n: Active-low asynchronous reset signal.
- set_song: Signal to set the current song.
- set_user: Signal to set the user record.
- key_notes: 7-bit input signal representing the state of key notes.

```
module LearningMode(
    input clk,
    input rst_n,
    input set_song,
    input set_user,
    input [`KEY_NOTES_WIDTH -1:0] key_notes,
```

Outputs:

- key_on_out: Output signal indicating a key press.
- key_off_out: Output signal indicating a key release.
- notes: 6-bit output signal representing the notes being played.
- track_number_learning: 2-bit output signal representing the current track number in learning mode.
- reaction_rate: 8-bit output signal representing the reaction rate.
- tub_sel: Output signal indicating the tub selected.
- tub_control: 8-bit output signal controlling the tub in automatic mode.
- tub_control_user: 8-bit output signal controlling the tub in user mode.
- finish: Output signal indicating the completion of the learning mode.

```
output key_on_out,
output key_off_out,
output wire[`NOTE_WIDTH -1:0] notes,
output wire[1:0] track_number_learning,
output wire[`REACTION_RATE_WIDTH -1:0] reaction_rate,
output tub_sel,
output [`TUB_CONTROL -1:0] tub_control,
output [`TUB_CONTROL_USER -1:0] tub_control_user,
output wire finish
);
```

Wires:

- index:10-bit wire representing the index of the note.
- key_on_song, key_off_song: Wires for key press and release in the song.
- key_on_speed, key_off_speed: Wires for key press and release related to speed.

- dummy: 6-bit Wire for a dummy signal.
- sameNote: Wire indicating if the current note is the same as the expected note.
- mistake_on, mistake_off: Wires indicating key press and release mistakes.
- dummy_wire: 4-bit wire for a dummy signal.
- key_on_LED: Wire indicating a key press for LED control.
- state: 3-bit wire representing the state of the learning mode.
- new_record: 4-bit wire representing a new record in BCD format.

```
wire[`INDEX_WIDTH -1:0] index;
wire key_on_song,key_off_song;
wire key_on_speed,key_off_speed;
wire [`NOTE_OUT_WIDTH -1:0] dummy;
wire sameNote;
wire mistake_on,mistake_off;
wire [`DUMMY_WIRE_WIDTH -1:0] dummy_wire;
wire key_on_LED;
wire[`STATE_LEARNING_MODE_WIDTH -1:0] state;
wire[`BCD_WIDTH -1:0] new_record;
```

Submodule Instantiations:

```
// submodule instantiations
    Key key(clk, rst_n, set_song, key_on_song, key_off_song);
    MusicChecker music_checker(clk, rst_n, set_song, key_notes, index, state, mistake_on, mistake_off);
    Music music(clk, rst_n, key_on_song, key_off_song, index, dummy_wire, notes, track_number_learning, finish);
    KeyController key_controller(clk, rst_n, key_notes, key_on_out, key_off_out, dummy);
    ReactionRating rating(clk, rst_n, state, reaction_rate);
    Score score(clk, rst_n, mistake_on, mistake_off, finish, tub_sel, tub_control, new_record);
    Record record(clk, rst_n, set_user, finish, new_record, tub_control_user);
endmodule
```

7.1. MusicChecker module

The MusicChecker module is designed to check the correctness of the user's key presses during music playback. It monitors the user's key presses, compares them with the expected notes, and handles the state transitions accordingly.

Inputs:

- clk: Clock signal
- rst_n: Active-low asynchronous reset signal.
- set_song: Signal to set the current song.
- key_notes: 7-bit input signal representing the state of key notes.

```
module MusicChecker(
    input clk,
    input rst_n,
    input wire set_song,
    input wire [`KEY_NOTES_WIDTH -1:0] key_notes,
```

Outputs:

- index: 10-bit output signal representing the index of the note.
- state: 3-bit output signal representing the state of the MusicChecker.
- mistake_on: Output signal indicating a mistake in key press.
- mistake_off: Output signal indicating a mistake in key release.

```
output reg[`INDEX_WIDTH -1:0] index,
output reg[`STATE_MUSIC_CHECKER_WIDTH -1:0] state,
output reg mistake_on,
output reg mistake_off
);
```

Wires:

- dummyWire: Wire for a dummy signal.
- note_input: 6-bit wire representing the input note.
- note_current: 6-bit wire representing the current note being played.
- key_on, key_off: Wires indicating key press and release.
- key_on_song, key_off_song: Wires indicating key press and release specific to the song.
- finish: Wire indicating the completion of music playback.

```
wire dummyWire;
wire[`NOTE_INPUT_WIDTH -1:0] note_input;
wire[`NOTE_CURRENT_WIDTH -1:0] note_current;
wire key_on,key_off;
wire key_on_song,key_off_song;
wire finish;
```

Submodule Instantiations:

```
//submodules' initialization
Key key(clk, rst_n, set_song, key_on_song, key_off_song);
KeyController key_controller(clk, rst_n, key_notes, key_on, key_off, note_input);
Music music(clk, rst_n, key_on_song, key_off_song, index, dummyWire, note_current, finish);
```

Mistake handling and state machine:

```
always @(posedge clk,negedge rst_n)begin
    if(~rst_n)begin
        index <= 10'd0;
        state <= 2'd0;
        count <= 0;
        mistake_on <=0;
        mistake_off <=1;
    end
    else begin
    /*state explanation
    0:when the user have not press the correct button
    1:the button is on
    2:the button is off, idle then enter state 0 */
        case(state)
            2'd0:
                if(note_current == note_input)begin
                    state<= 2'd1;
                    mistake_on<=0;
                    mistake_off<=1;
                end
                else begin
                if(key_on)begin
                    mistake_on <=1;
                    mistake_off <=0;
                end
                end
            2'd1:
                if(key_off)begin
                    state <= 2'd2;
                end
```

```
                2'd2:
                    if(count<4*max_IDLE_count) begin
                        count <= count + 1;
                    end
                    else begin
                        index <= index + 1;
                        count <= 0;
                        state <=2'd3;
                    end
                 2'd3:
                    state <= 2'd0;
            endcase
        end
    end
endmodule
```

7.2. Record module

The Record module is meant to manage and record new scores for the three different users.

Inputs:

- clk: Clock signal.
- rst_n: Active-low asynchronous reset signal.
- set_user: Signal indicating a user switch.
- set_record: Signal indicating a new record should be set.
- new_record: 4-bit input representing the new record value.

Outputs:

- tub_control_user: 8-bit 7-segment display control signal based on the BCD representation of the current record.

```
module Record(
    input clk,
    input rst_n,
    input set_user,
    input set_record,
    input [`NEW_RECORD_WIDTH -1:0] new_record,  // new record input
    output [`TUB_CONTROL_USER -1:0] tub_control_user
);
```

Internal registers:

- current_record: 4-bit register storing the BCD-encoded current record.
- current_user: 2-bit register storing the current user (2-bit value).
- record1, record2, record3: 4-bit registers storing the records for three users (BCD-encoded).

Modules instantiated:

- Key: Module handling key-related signals (key_on and key_off).
- BCDto7Segment: Converts the BCD representation of the current record to 7-segment display control signals.

```
wire key_on,key_off;
reg[`CURRENT_RECORD_WIDTH -1:0] current_record;
Key key(clk,rst_n,set_user,key_on,key_off);
reg[`CURRENT_USER_WIDTH -1:0] current_user;
reg[`RECORD123_WIDTH -1:0] record1 = 0,record2 = 1,record3 = 2;
BCDto7Segment bcd(current_record,tub_control_user);
```

Record updates:

Update the records (record1, record2, record3) based on the current user (current_user) when the set_record signal is active.

```
always@(posedge clk)begin
    if(set_record)begin
        case (current_user)
        2'd1:record1 <= new_record;
        2'd2:record2 <= new_record;
        2'd3:record3<=new_record;
        endcase
    end
end
```

State machine:

The module includes a state machine to manage user switching:

- IDLE (IDLE): Waits for a key press (key_on) to initiate a user switch. Transitions to the SET state.
- SET (SET): Waits for a key release (key_off) to complete the user switch. Transitions back to the IDLE state.

```
// state machine for managing user switching
    reg state;
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            state <= `IDLE;
        end
        else begin
            case (state)
                `IDLE:
                    begin
                        if (key_on) begin
                            case (current_user)
                                2'd1: begin current_user <= 2'd2; end
                                2'd2: begin current_user <= 2'd3; end
                                2'd3: begin current_user <= 2'd1; end
                                default: begin current_user <= 2'd1;end
                            endcase
                            state <= `SET;
                        end
                        else
                            state <= `IDLE;
                    end
                `SET:
                    begin
                        if (key_off) begin
                            state <= `IDLE;
                        end
                        else
                            state <= `SET;
                    end
                default:
                    state <= `IDLE;
            endcase
        end
    end
```

Record update:

The module updates the current record based on the current user:

- If set_record is active, it updates the record for the current user with the value of new_record.

```
    // logic for updating current record based on current user
            always @(*) begin
                case (current_user)
                    2'd1: current_record = record1;
                    2'd2: current_record = record2;
                    2'd3: current_record = record3;
                    default: current_record = record1;
                endcase
            end
endmodule
```

IV. Implementation instructions for BONUS:

7.3. ReactingRating module

This module is responsible for calculating the reaction rate based on certain conditions and input signals.

Inputs:

- clk: Clock signal.
- rst_n: Active-low asynchronous reset signal.
- state: 3-bit input signal representing the state information.

Outputs:

- reaction_rate: 8-bit output signal representing the calculated reaction rate.

```
module ReactionRating(
    input clk,
    input rst_n,
    input [`STATE_REACTION_RATING_WIDTH -1:0] state,
    output reg[`REACTION_RATE_WIDTH -1:0] reaction_rate
    );
```

Regs:

- count_diff1: 32-bit register storing the count difference 1.
- count_diff2: 32-bit register storing the count difference 2.
- count1: 32-bit register storing count 1.
- count2: 32-bit register storing count 2.

```
reg[`OUTPUTCOUNT_WIDTH -1:0] count_diff1;
reg[`OUTPUTCOUNT_WIDTH -1:0] count_diff2;
reg[`OUTPUTCOUNT_WIDTH -1:0] count1;
reg[`OUTPUTCOUNT_WIDTH -1:0] count2;
```

State machine:

The module utilizes a simple state machine with the following states:

- 2'd0: Increment count1 until it reaches a predefined value (COUNT1).
- 2'd1: Increment count2.
- 2'd2: Calculate the reaction rate based on the sum of count_diff1 and count_diff2.
- 2'd3: Reset count1 and count2 to 0.

```
  // state machine logic
always @(posedge clk or negedge rst_n) begin
          if (!rst_n) begin
              count1 <= 0;
              count2 <= 0;
              reaction_rate <= 0;
          end
          else begin
              case (state)
                  2'd0:
                      begin
                          if (count1 <= `COUNT1)
                              count1 <= count1 + 32'd1;
                          reaction_rate <= 8'd0;
                          count2 <= 32'd0;
                      end
                  2'd1:
                      begin
                          if (count2 <= `COUNT2)
                              count2 <= count2 + 32'd1;
                      end
```

```
                2'd2:
                    if (count_diff1 + count_diff2 <= `COUNT_DIFF_12_1)
                        reaction_rate <= 8'b11111111;
                    else if (count_diff1 + count_diff2 <= `COUNT_DIFF_12_2)
                        reaction_rate <= 8'b01111111;
                    else if (count_diff1 + count_diff2 <= `COUNT_DIFF_12_3)
                        reaction_rate <= 8'b00111111;
                    else if (count_diff1 + count_diff2 <= `COUNT_DIFF_12_4)
                        reaction_rate <= 8'b00011111;
                    else if (count_diff1 + count_diff2 <= `COUNT_DIFF_12_5)
                        reaction_rate <= 8'b00001111;
                    else if (count_diff1 + count_diff2 <= `COUNT_DIFF_12_6)
                        reaction_rate <= 8'b00000111;
                    else if (count_diff1 + count_diff2 <= `COUNT_DIFF_12_7)
                        reaction_rate <= 8'b00000011;
                    else
                        reaction_rate <= 8'b00000001;
                2'd3:
                    begin
                        count1 <= 32'd0;
                        count2 <= 32'd0;
                    end
            endcase
        end
    end
```

Count Difference Calculation:

- count_diff1 is assigned the value of count1.
- count_diff2 is calculated based on the difference between count2 and a predefined threshold (COUNT2_GREATER).

```
    // count difference calculation
    always @(*) begin
        count_diff1 = count1;
        if (count2 > `COUNT2_GREATER)
            count_diff2 = count2 - 32'd20000000;
        else
            count_diff2 = 32'd20000000 - count2;
    end

endmodule
```

7.4. Score module

This module is a simple scoring mechanism that counts mistakes and outputs the score in BCD format.

Inputs:

- clk: Clock signal.
- rst_n - Active-low asynchronous reset signal.
- count_on - Signal indicating the occurrence of a count increment event.
- count_off - Signal indicating the occurrence of a count decrement event.
- finish - Signal indicating the end of the task (finish condition).

```
module Score(
        input clk,
        input rst_n,
        input count_on,
        input count_off,
        input finish,
```

Outputs:

- tub_sel: Signal indicating whether the task is finished.
- tub_control: 8-bit 7-segment display control signal based on the BCD score.
- new_record: 4-bit BCD-encoded output representing the mistakes counter.

```
    output tub_sel,
    output [`SEG_WIDTH -1:0] tub_control,
    output [`BCD_WIDTH -1:0] new_record   // new record BCD output
);
```

Regs, assign and initialization of submodule (BCDto7Segment):

- BCD: 4-bit register storing the BCD-encoded score.
- state: 3-bit register representing the state of the state machine.
- mistakes: 9-bit register counting the number of mistakes.
- assign tub_sel = finish
- BCDto7Segment bcd(BCD, tub_control)

```
reg[`BCD_WIDTH -1:0] BCD;
reg[`STATE_SCORE_WIDTH -1:0] state; //state for controlling score counting
reg[`MISTAKES_WIDTH -1:0] mistakes; //mistakes counter
BCDto7Segment bcd(BCD, tub_control);
assign tub_sel = finish;
```

State machine:

- The module uses a simple state machine with the following states:
- 2'd0: Wait for a count increment event (count_on).
- 2'd1: Increment the mistakes counter (mistakes) on each count increment event.
- 2'd2: Wait for a count decrement event (count_off).

```
// state machine for counting mistakes
  always @(posedge clk or negedge rst_n) begin
      if (!rst_n) begin
          state <= 2'd0;
          mistakes <= 0;
      end
      else begin
          case (state)
              2'd0:
                  begin
                      if (count_on) state <= state + 1;
                  end
              2'd1:
                  begin
                      state <= state + 1;
                      mistakes <= mistakes + 1;
                  end
              2'd2:
                  begin
                      if (count_off) state <= 2'd0;
                  end
          endcase
      end
  end
```

BCD encoding logic:

The mistakes counter (mistakes) is used to determine the BCD-encoded score. The BCD values are assigned based on different ranges of mistakes:

- 0 to 2 mistakes: BCD value 8
- 3 mistakes: BCD value 7
- 4 mistakes: BCD value 6

- 5 mistakes: BCD value 5
- 6 mistakes: BCD value 4
- 7 mistakes: BCD value 3
- 8 or more mistakes: BCD value 2

```
    always@(*)begin
        if(mistakes <= 9'd2)BCD = 8;
        else if(mistakes <= 9'd3)BCD = 7;
        else if(mistakes <= 9'd4)BCD = 6;
        else if(mistakes <= 9'd5)BCD = 5;
        else if(mistakes <= 9'd6)BCD = 4;
        else if(mistakes <= 9'd7)BCD = 3;
        else BCD = 2;
    end

    assign new_record = BCD;
endmodule
```

8. **Speed control:**

The user should be able to control the speed of the song in AutoPlay mode. The detailed description of the module that handles this functionality can be found in the BeatController module.

```
reg state;
always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            state <= `IDLE;
        end
        else begin
            case (state)
                `IDLE:
                   begin
                       if (key_on) begin
                           state <= `SET;
                           case (speed_mode)
                               2'd0: begin speed_mode <= `NORMAL_MODE;max_count <= `NORMAL; max_IDLE_count <= `NORMAL_IDLE; end
                               2'd1: begin speed_mode <= `FAST_MODE;max_count <= `FAST; max_IDLE_count <= `FAST_IDLE; end
                               2'd2: begin speed_mode <= `SLOW_MODE;max_count <= `SLOW; max_IDLE_count <= `SLOW_IDLE; end
                               default: begin speed_mode <= `NORMAL_MODE;max_count <= `NORMAL; max_IDLE_count <= `NORMAL_IDLE; end
                           endcase
                       end
                       else
                           state <= `IDLE;
                   end
```

```
                `SET:
                    begin
                        if (key_off) begin
                            state <= `IDLE;
                        end
                        else
                            state <= `SET;
                    end
                default:
                    state <= `IDLE;
            endcase
        end
    end
endmodule
```

The detailed implementation of the inputs, outputs and the internal signals of the module can be found in the implementation details of the BeatController module. (Please refer to page 22)

VGA

9. VGADisplay module

The VGADisplay module is designed to interface with a VGA display system, generating signals for horizontal synchronization (hsync), vertical synchronization (vsync), and pixel RGB data (vga_rgb). It receives input signals related to musical notes, key events, and other parameters to drive the VGA display.

Inputs:

- clk: Clock signal.
- sys_rst_n: Active-low system reset signal.
- mode: VGA display mode.
- key_on_in: Signal indicating a key press event.
- key_off_in: Signal indicating a key release event.
- note_in: Input representing musical notes.
- selected_note: Input representing selected musical notes.
- track_number_auto: Input representing the track number in auto mode.
- track_number_learning: Input representing the track number in learning mode.
- current_octave: Input representing the current octave.
- reaction_rate: Input representing the reaction rate.
- finish: Signal indicating a task or operation completion.

Outputs:

- hsync: Horizontal synchronization signal.
- vsync: Vertical synchronization signal.
- vga_rgb: Pixel RGB data for the VGA display.

```
module VGADisplay(
    input wire clk,
    input wire sys_rst_n,
    input wire [`MODE_WIDTH -1:0] mode,
    input wire key_on_in,
    input wire key_off_in,
    input wire [`NOTE_IN_WIDTH -1:0] note_in,
    input wire [`SELECTED_NOTE_WIDTH -1:0] selected_note,
    input wire [`TRACK_NUM_AUTO_WIDTH -1:0]  track_number_auto,
    input wire [`TRACK_NUM_LEARNING_WIDTH -1:0] track_number_learning,
    input wire [`CURRENT_OCTAVE_WIDTH -1:0] current_octave,
    input wire [`REACTION_RATE_WIDTH -1:0] reaction_rate,
    input wire finish,
    output wire hsync,
    output wire vsync,
    output wire [`VGA_RGB_WIDTH -1:0] vga_rgb
);
```

Internal signals and rst_n assign:

- vga_clk: Clock signal for the VGA system.
- locked: Signal indicating whether the VGA clock is locked.
- rst_n: Active-low reset signal for internal use.

```
//clock and reset signals
    wire vga_clk;
    wire locked;
    wire rst_n;
//pixel-related signals
    wire [`PIX_X_WIDTH -1:0] pix_x;
    wire [`PIX_Y_WIDTH -1:0] pix_y;
    wire [`PIX_DATA_WIDTH -1:0] pix_data;

    assign rst_n = (sys_rst_n && locked);
```

Instantiated modules:

- clk_wiz_0: Clock wizard module for generating the VGA clock signal.
- PixelMapper: Module responsible for mapping musical input signals to pixel-related signals.
- VGADriver: Module responsible for driving the VGA display based on pixel-related signals.

```
clk_wiz_0 clk_25
(
    .clk_in1(clk),
    .reset(~sys_rst_n),
    .clk_out1(vga_clk),
    .locked(locked)
);

PixelMapper pixel_mapper
(
    .clk(clk),
    .vga_clk(vga_clk),
    .rst_n(rst_n),
    .pix_x(pix_x),
    .pix_y(pix_y),
    .pix_data(pix_data),
    .mode(mode),
    .key_on_in(key_on_in),
    .key_off_in(key_off_in),
    .note_in(note_in),
    .selected_note(selected_note),
    .track_number_auto(track_number_auto),
    .track_number_learning(track_number_learning),
    .current_octave(current_octave),
    .reaction_rate(reaction_rate),
    .finish(finish)
);
```

```
    VGADriver vga_driver
    (
        .vga_clk(vga_clk),
        .rst_n(rst_n),
        .pix_data(pix_data) ,
        .pix_x(pix_x),
        .pix_y(pix_y),
        .hsync(hsync),
        .vsync(vsync),
        .vga_rgb(vga_rgb)
    );

endmodule
```

9.1. PixelMapper module

This module is responsible for generating pixel data for the VGA display.

Inputs:

- clk: Clock signal.
- vga_clk: Another clock signal, possibly related to the VGA display.
- rst_n: Active-low reset signal.
- pix_x: Pixel x-coordinate.
- pix_y: Pixel y-coordinate.
- mode: Mode signal, possibly indicating the operation mode (auto-play, free mode, learning mode).
- key_on_in, key_off_in: Signals related to key events.
- note_in: Input related to musical notes.
- selected_note: Selected musical note.
- track_number_auto, track_number_learning: Track numbers for auto-play and learning modes.
- current_octave, reaction_rate: Parameters related to musical aspects.
- finish: A signal indicating the completion of a task.

Output:

- pix_data: Output representing pixel data to be displayed.

```
module PixelMapper (
    input wire clk,
    input wire vga_clk,
    input wire rst_n,
    input wire [`PIX_X_WIDTH -1:0] pix_x,
    input wire [`PIX_Y_WIDTH -1:0] pix_y,
    input wire [`MODE_WIDTH -1:0] mode,
    input wire key_on_in,
    input wire key_off_in,
    input wire [`NOTE_IN_WIDTH -1:0] note_in,
    input wire [`SELECTED_NOTE_WIDTH -1:0] selected_note,
    input wire [`TRACK_NUM_AUTO_WIDTH -1:0] track_number_auto,
    input wire [`TRACK_NUM_LEARNING_WIDTH -1:0] track_number_learning,
    input wire [`CURRENT_OCTAVE_WIDTH -1:0] current_octave,
    input wire [`REACTION_RATE_WIDTH -1:0] reaction_rate,
    input wire finish,
    output reg [`PIX_DATA_WIDTH -1:0] pix_data
);
```

Internal signals and components:

- state: State variable.
- key_color: Color determined based on the mode.
- key_on: A register storing a key-on state.
- Various wires and registers related to ASCII characters and their display on the VGA screen.
- Instances of an ASCII_ROM module for each ASCII character display.

```
reg [`STATE_PIXEL_MAPPEL_WIDTH -1:0] state;
wire [`KEY_COLOR_WIDTH -1:0] key_color = (mode == `AUTO_PLAY_PIXEL_MAPPEL ) ? `MAGENTA :
(mode == `FREE_MODE_PIXEL_MAPPEL  ) ? `BLUE : (mode == `LEARNING_PIXEL_MAPPEL) ?  `GREEN : `WHITE;
reg key_on;
reg [`TRACK_NUM_WIDTH -1:0]  track_number;
wire [`ROM_ADDR_TWO_TIGERS_WIDTH -1:0] rom_addr_two_tigers;          // 11-bit text ROM address
wire [`ASCII_CHAR_TWO_TIGERS_WIDTH -1:0] ascii_char_two_tigers;        // 7-bit ASCII character code
wire [`CHAR_ROW_TWO_TIGERS_WIDTH -1:0] char_row_two_tigers;          // 4-bit row of ASCII character
wire [`BIT_ADDR_TWO_TIGERS_WIDTH -1:0] bit_addr_two_tigers;          // column number of ROM data
wire [`ROM_DATA_TWO_TIGERS_WIDTH -1:0] rom_data_two_tigers;          // 8-bit row data from text ROM
wire ascii_bit_two_tigers, ascii_bit_on_two_tigers;     // ROM bit and status signal
```

Purpose:

The module includes multiple instances of the ASCII_ROM module (which will be presented next), each responsible for fetching ASCII character data based on the pixel coordinates. The ASCII characters are then displayed on the VGA screen, and the color will vary depending on the mode.

The module also handles track selection based on the mode, and it has different sections for displaying ASCII characters corresponding to the musical tracks (Little Star, Two Tigers and Ode to Joy).

Code example:

```
ASCII_ROM ascii_rom_two_tigers(
    .clk(vga_clk),
    .addr(rom_addr_two_tigers),
    .data(rom_data_two_tigers)
);

assign rom_addr_two_tigers = {ascii_char_two_tigers, char_row_two_tigers};   // ROM address is ascii code + row
assign ascii_bit_two_tigers = rom_data_two_tigers[~bit_addr_two_tigers];     // reverse bit order

assign ascii_char_two_tigers = (pix_x >= 51 && pix_x <= 59) ? 7'h54 :  // 'T'
                   (pix_x >= 59 && pix_x <= 67) ? 7'h57 :  // 'W'
                   (pix_x >= 67  && pix_x <= 75) ? 7'h4F :  // 'O'
                   (pix_x >= 75  && pix_x <= 83) ? 7'h20 :  // ' '
                   (pix_x >= 83  && pix_x <= 91) ? 7'h54 :  // 'T'
                   (pix_x >= 91  && pix_x <= 99) ? 7'h49 :  // 'I'
                   (pix_x >=99  && pix_x <= 107) ? 7'h47 :  // 'G'
                   (pix_x >=107  && pix_x <= 115) ? 7'h45 :  // 'E'
                   (pix_x >=115  && pix_x <= 123) ? 7'h52 :  // 'R'
                   (pix_x >=123  && pix_x <= 134) ? 7'h53 :  // 'S'
                  7'h20;  // ' '

assign char_row_two_tigers = pix_y[3:0];              // row number of ascii character rom
assign bit_addr_two_tigers = pix_x[2:0];              // column number of ascii character rom

assign ascii_bit_on_two_tigers = (pix_x >= 51 && pix_x <= 147 && pix_y >= 110 && pix_y <= 127) ? ascii_bit_two_tigers : 1'b0;
```

and so on…

9.1.1. ASCII_ROM module

Inputs:

- clk: Clock signal.
- addr: Address input for accessing the ROM.

Outputs:

- data: Output representing the ASCII character pattern corresponding to the given address.

```
module ASCII_ROM(
    input clk,
    input wire [`ADDR_WIDTH -1:0] addr,
    output reg [`DATA_WIDTH -1:0] data
    );
```

Internal signals and components:

- addr_reg: A register storing the address input for synchronization with the clock.

- Cases for various ASCII character codes with corresponding 8-bit patterns.

```
// infer BRAM
(* rom_style = "block" *)
reg [`ADDR_REG_WIDTH -1:0] addr_reg;
```

Purpose:

This module serves as a ROM that stores ASCII character patterns. The addr input is used to access specific locations in the ROM, and the corresponding ASCII character pattern is provided at the data output. The ASCII characters are represented by 8-bit patterns, and the module includes cases for handling specific ASCII character codes.

Example:

For instance, when addr is 11'h012, the output data will be 8'b01111110, which represents a smiley face.

Code example:

```
always @*
    case(addr_reg)
        // Begin non-printable ASCII characters (00 - 1f)
        // code x00 (nul) null byte, which is the all-zero pattern
        11'h000: data = 8'b00000000;    //
        11'h001: data = 8'b00000000;    //
        11'h002: data = 8'b00000000;    //
        11'h003: data = 8'b00000000;    //
        11'h004: data = 8'b00000000;    //
        11'h005: data = 8'b00000000;    //
        11'h006: data = 8'b00000000;    //
        11'h007: data = 8'b00000000;    //
        11'h008: data = 8'b00000000;    //
        11'h009: data = 8'b00000000;    //
        11'h00a: data = 8'b00000000;    //
        11'h00b: data = 8'b00000000;    //
        11'h00c: data = 8'b00000000;    //
        11'h00d: data = 8'b00000000;    //
        11'h00e: data = 8'b00000000;    //
        11'h00f: data = 8'b00000000;    //
        // code x01 smilet face
        11'h010: data = 8'b00000000;    //
        11'h011: data = 8'b00000000;    //
        11'h012: data = 8'b01111110;    // ******
        11'h013: data = 8'b10000001;    //*      *
        11'h014: data = 8'b10100101;    //* *  * *
        11'h015: data = 8'b10000001;    //*      *
        11'h016: data = 8'b10000001;    //*      *
        11'h017: data = 8'b10111101;    //* **** *
        11'h018: data = 8'b10011001;    //*  **  *
        11'h019: data = 8'b10000001;    //*      *
        11'h01a: data = 8'b10000001;    //*      *
        11'h01b: data = 8'b01111110;    // ******
        11'h01c: data = 8'b00000000;    //
        11'h01d: data = 8'b00000000;    //
        11'h01e: data = 8'b00000000;    //
        11'h01f: data = 8'b00000000;    //
```

and so on…

9.2. VGADriver Module

Inputs:

- vga_clk: VGA clock input.

- rst_n: System reset signal (active low).
- pix_data: Pixel data input.

Outputs:

- pix_x: Pixel X-coordinate output.
- pix_y: Pixel Y-coordinate output.
- hsync: Horizontal sync signal output.
- vsync: Vertical sync signal output.
- vga_rgb: VGA RGB data output.

```
module VGADriver(
    input wire vga_clk,           // VGA clock input
    input wire rst_n,             // system reset signal (active low)
    input wire [`PIX_DATA_WIDTH -1:0] pix_data,  // pixel data input
    output wire [`PIX_X_WIDTH -1:0] pix_x,    // pixel X coordinate output
    output wire [`PIX_Y_WIDTH -1:0] pix_y,    // pixel Y coordinate output
    output wire hsync,           // horizontal sync signal output
    output wire vsync,           // vertical sync signal output
    output wire [`VGA_RGB_WIDTH -1:0] vga_rgb  // VGA RGB data output
);
```

Internal signals:

- counter_h: Horizontal counter.
- counter_v: Vertical counter.
- rgb_valid: RGB valid signal.
- pix_data_req: Pixel data request signal.

```
// horizontal and vertical counters
reg [`COUNTER_H_WIDTH -1:0] counter_h;
reg [`COUNTER_V_WIDTH -1:0] counter_v;

// RGB valid and pixel data request signals
wire rgb_valid;     // output pix_data to VGA when rgb_valid is high (1'b1)
wire pix_data_req;  // request new pix_data when pix_data_req is high (1'b1)
```

Implementation details:

Horizontal Counter (counter_h):

```
// horizontal counter logic
always @(posedge vga_clk or negedge rst_n) begin
    if (rst_n == 1'b0) begin
        counter_h <= 10'd0;
    end
    else if (counter_h == (`H_TOTAL - 1'b1)) begin
        counter_h <= 10'd0;  // reset counter at end of line
    end
    else begin
        counter_h <= counter_h + 10'd1;
    end
end
```

Vertical Counter (counter_v):

```
// vertical counter logic
always @(posedge vga_clk or negedge rst_n) begin
    if (rst_n == 1'b0) begin
        counter_v <= 10'd0;
    end
    else if ((counter_h == (`H_TOTAL - 1'b1)) && (counter_v == (`V_TOTAL - 1'b1))) begin
        counter_v <= 10'd0;  // reset counter at end of frame
    end
    else if (counter_h == (`H_TOTAL - 1'b1)) begin
        counter_v <= counter_v + 10'd1;  // increment counter at the end of line
    end
    else begin
        counter_v <= counter_v; // hold the counter
    end
end
```

RGB Valid Signal (rgb_valid):

```
// RGB valid signal logic
assign rgb_valid = ((counter_h >= `H_SYNC + `H_BACK + `H_LEFT)
                    && (counter_h < `H_SYNC + `H_BACK + `H_LEFT + `H_VALID)
                    && (counter_v >= `V_SYNC + `V_BACK + `V_TOP)
                    && (counter_v < `V_SYNC + `V_BACK + `V_TOP + `V_VALID))
                    ?  1'b1 : 1'b0;  // RGB valid during addresable video
```

Pixel Data Request Signal (pix_data_req):

```
// pixel data request signal logic
assign pix_data_req = ((counter_h >= `H_SYNC + `H_BACK + `H_LEFT - 1'b1)
                    && (counter_h < `H_SYNC + `H_BACK + `H_LEFT + `H_VALID - 1'b1)
                    && (counter_v >= `V_SYNC + `V_BACK + `V_TOP)
                    && (counter_v < `V_SYNC + `V_BACK + `V_TOP + `V_VALID))
                    ?  1'b1 : 1'b0;  // pixel data request during addressable video
```

Pixel X and Y Coordinate (pix_x and pix_y):

```
// pixel X and Y coordinate logic
assign pix_x = (pix_data_req == 1'b1) ? (counter_h - (`H_SYNC + `H_BACK + `H_LEFT) - 1'b1) : 10'd0;
assign pix_y = (pix_data_req == 1'b1) ? (counter_v - (`V_SYNC + `V_BACK + `V_TOP)) : 10'd0;
```

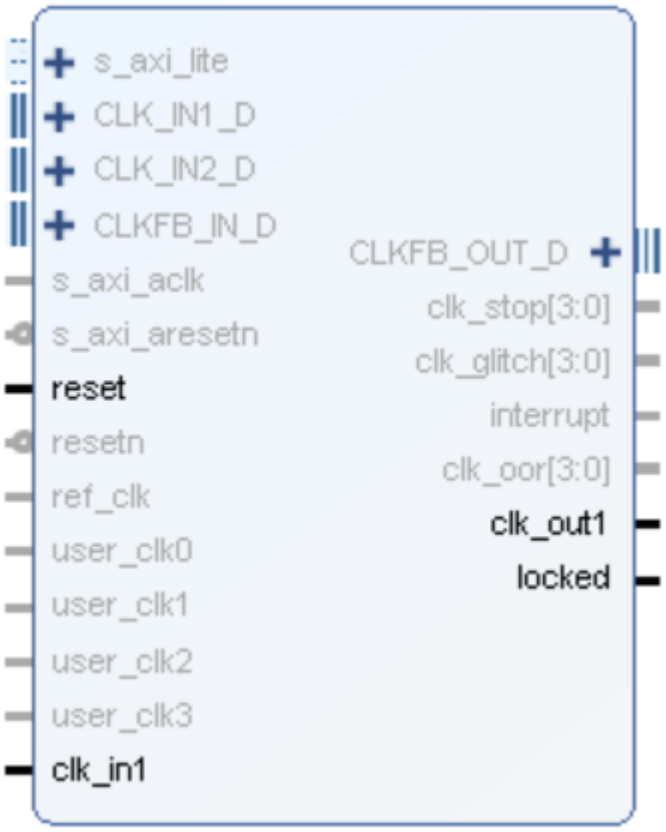
Sync Signal (hsync and vsync):

```
// sync signal logic
assign hsync = (counter_h <= `H_SYNC - 1'b1) ? 1'b1 : 1'b0;  // HSYNC during sync pulse
assign vsync = (counter_v <= `V_SYNC - 1'b1) ? 1'b1 : 1'b0;  // VSYNC during sync pulse
```

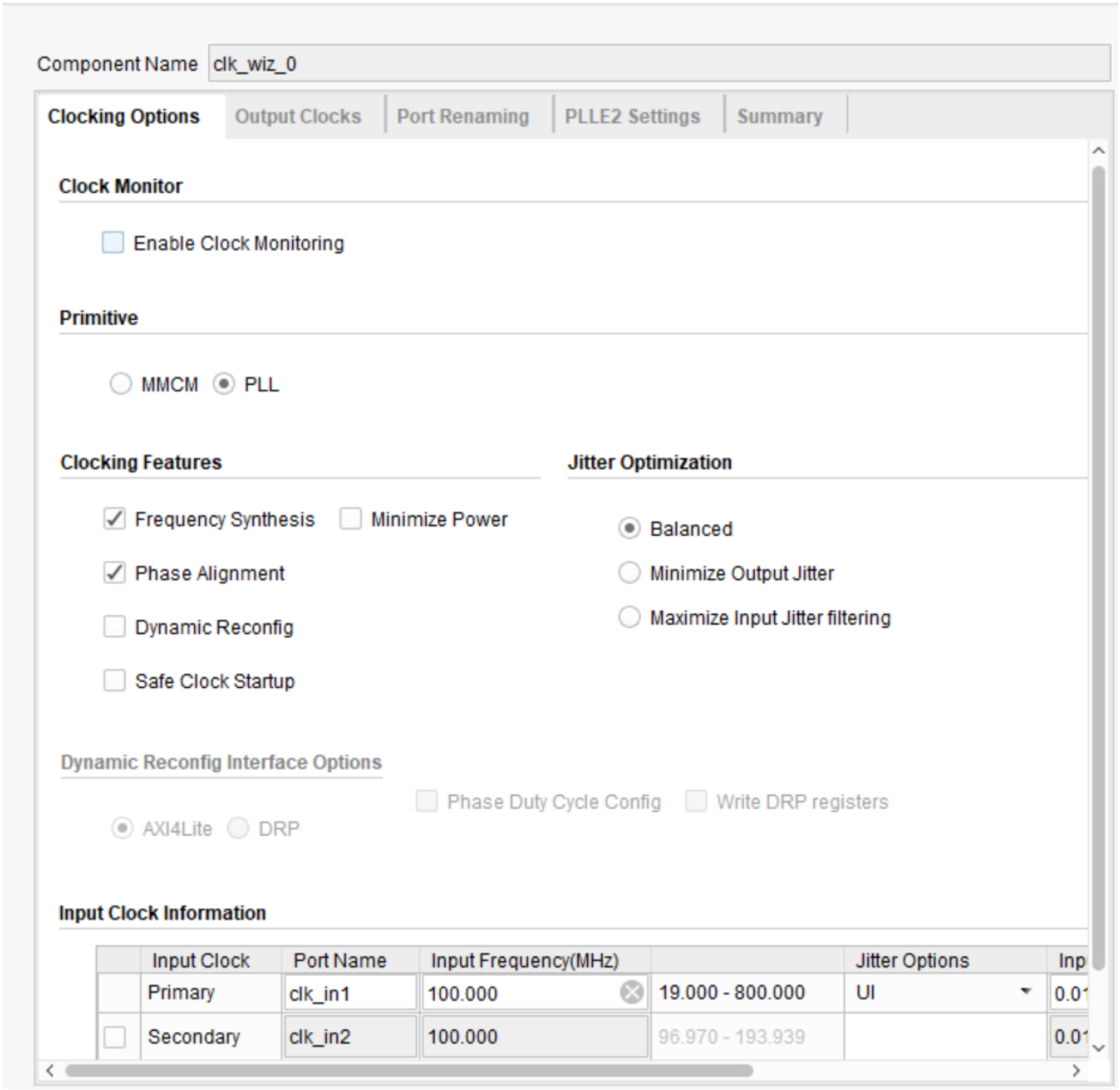
VGA RGB Data (vga_rgb):

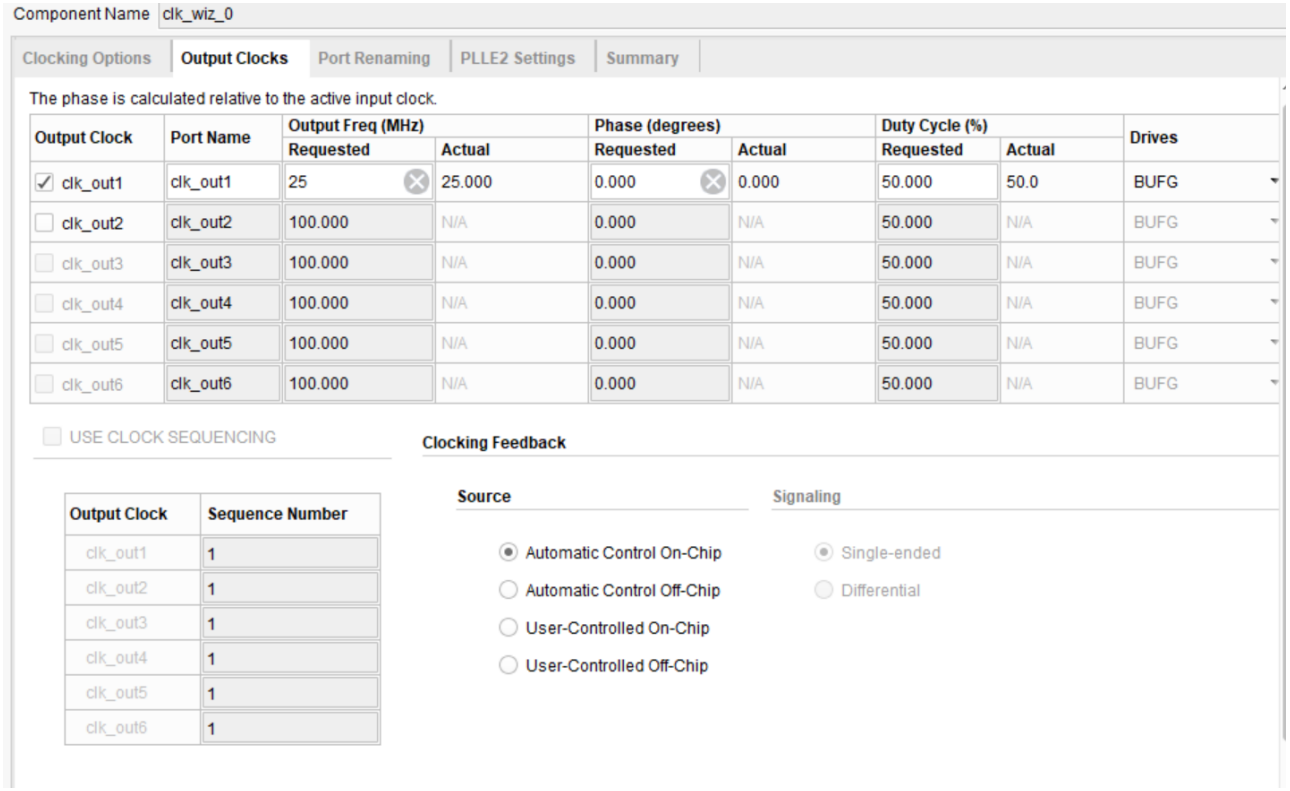
```
// VGA RGB data logic
assign vga_rgb = (rgb_valid == 1'b1) ? pix_data : 12'h000;  // output pixel data when RGB valid
```

9.3. clk_wiz_0:



Clocking wizard:


Component Name
clk_wiz_0
Clocking Options
Output Clocks
Port Renaming
PLLE2 Settings
Summary
Clock Monitor
Enable Clock Monitoring
Primitive
MMCM
PLL
Clocking Features
Frequency Synthesis
Minimize Power
Phase Alignment
Dynamic Reconfig
Safe Clock Startup
Jitter Optimization
Balanced
Minimize Output Jitter
Maximize Input Jitter filtering
Dynamic Reconfig Interface Options
Phase Duty Cycle Config
Write DRP registers
AXI4Lite
DRP
Input Clock Information
Input Clock
Port Name
Input Frequency(MHz)
Jitter Options
Primary
clk_in1
100.000
19.000 - 800.000
UI
Secondary
clk_in2
100.000
96.970 - 193.939



Component Name clk_wiz_0

Clocking Options | Output Clocks | Port Renaming | PLLE2 Settings | Summary

The phase is calculated relative to the active input clock.

Output Clock	Port Name	Output Freq (MHz)		Phase (degrees)		Duty Cycle (%)		Drives
		Requested	Actual	Requested	Actual	Requested	Actual	
☑ clk_out1	clk_out1	25	25.000	0.000	0.000	50.000	50.0	BUFG
☐ clk_out2	clk_out2	100.000	N/A	0.000	N/A	50.000	N/A	BUFG
☐ clk_out3	clk_out3	100.000	N/A	0.000	N/A	50.000	N/A	BUFG
☐ clk_out4	clk_out4	100.000	N/A	0.000	N/A	50.000	N/A	BUFG
☐ clk_out5	clk_out5	100.000	N/A	0.000	N/A	50.000	N/A	BUFG
☐ clk_out6	clk_out6	100.000	N/A	0.000	N/A	50.000	N/A	BUFG

☐ USE CLOCK SEQUENCING

Output Clock	Sequence Number
clk_out1	1
clk_out2	1
clk_out3	1
clk_out4	1
clk_out5	1
clk_out6	1

Clocking Feedback

Source

- ◉ Automatic Control On-Chip
- ○ Automatic Control Off-Chip
- ○ User-Controlled On-Chip
- ○ User-Controlled Off-Chip

Signaling

- ◉ Single-ended
- ○ Differential

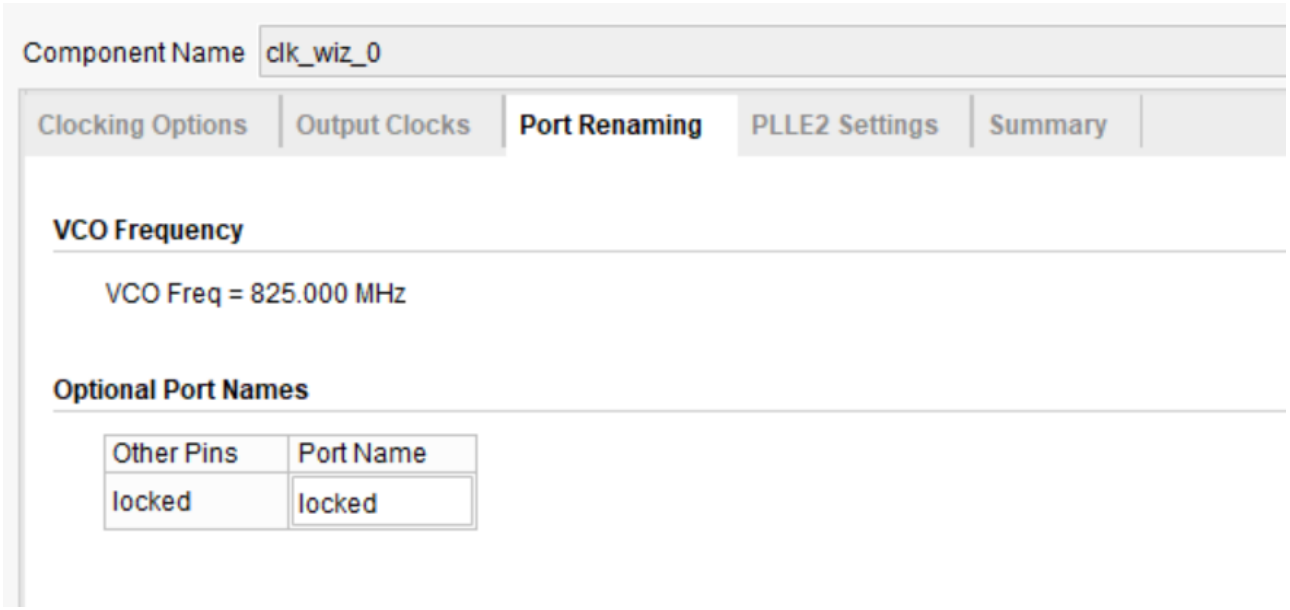
Enable Optional Inputs / Outputs for MMCM/PLL

☑ reset ☐ power_down

☑ locked

Reset Type

◉ Active High ○ Active Low



Component Name clk_wiz_0

Clocking Options | Output Clocks | **Port Renaming** | PLLE2 Settings | Summary

VCO Frequency

VCO Freq = 825.000 MHz

Optional Port Names

Other Pins	Port Name
locked	locked

Component Name clk_wiz_0

Clocking Options | Output Clocks | Port Renaming | **PLLE2 Settings** | Summary

These are the settings based on inputs from previous pages. Any update on this page will override the optimal settings calculated by the wizard

☐ Allow Override Mode

Attribute	Value
BANDWIDTH	OPTIMIZED
CLKFBOUT_MULT	33
CLKFBOUT_PHASE	0.000
CLKIN1_PERIOD	10.000
CLKIN2_PERIOD	10.000
COMPENSATION	ZHOLD
DIVCLK_DIVIDE	4
REF_JITTER1	0.010
REF_JITTER2	0.010
STARTUP_WAIT	☐

Clk Wizard Port	Renamed Port	MMCM/PLL Port	Divide	Duty Cycle	Phase
clk_out1	clk_out1	CLKOUT0	33	0.500	0.000

Component Name clk_wiz_0

Clocking Options | Output Clocks | Port Renaming | PLLE2 Settings | **Summary**

Primary Input Clock Attributes

Input Clock Frequency (MHz)	100.000
Clock Source	Single_ended_clock_capable_pin
Jitter	0.010

Clocking Primitive Attributes

Primitive Instantiated : PLL
Divide Counter : 4
Mult Counter : 33
Clock Phase Shift : None

Clock Wiz O/p Pins	Source	Divider Value	Tspread (ps)	Pk-to-Pk Jitter (ps)	Phase Error (ps)
clk_out1	PLL CLKOUT0	33	OFF	352.369	261.747
clk_out2	OFF	OFF	OFF	OFF	OFF
clk_out3	OFF	OFF	OFF	OFF	OFF
clk_out4	OFF	OFF	OFF	OFF	OFF
clk_out5	OFF	OFF	OFF	OFF	OFF
clk_out6	OFF	OFF	OFF	OFF	OFF
clk_out7	OFF	OFF	OFF	OFF	OFF

10. Verilog Header (constants.vh)

This module helps improve code readability and maintainability by assigning names to specific values or widths.

- The `define directives are used for creating named constants in the Header file as follow:

```
`define FREE_MODE          0
`define AUTO_MODE          1
`define LEARNING_MODE      2

`define NOTE_A_WIDTH          6
`define NOTE_B_WIDTH          6
`define NOTE_C_WIDTH          6
`define KEY_ON_OUT_WIDTH      1
`define KEY_OFF_OUT_WIDTH     1
`define NOTE_WIDTH            6
`define NOTE_OUT_WIDTH        6
`define NOTE_INPUT_WIDTH      6
`define NOTE_CURRENT_WIDTH    6
`define KEY_NOTES_WIDTH       7

`define CURRENT_STATE_WIDTH          2
`define OUTPUTCOUNT_WIDTH            32
`define FREQUENCY_DIVIDER_COUNT      28
```

- The file can be accessed later by other modules within the project through the syntax:

```
`include "constants.vh"
```

Purpose and usage:

- If the project is meant to be used for larger purposes, the numerical constants in the code, as well as all the symbolic constants can be accessed and modified through the “constants.vh” file.
- In the Verilog code, instead of using numeric values directly, we can simply use these symbolic names for clarity. For example, instead of using 0 for free mode, `FREE_MODE can be used.

V. PROJECT SUMMARY

Conclusion:

We cannot deny that this project was a challenge for us as we never dealt with hardware programming before. At first, we were all confused about how to get started, however, after discussing and deciding on the logic of our project, things got easier and manageable. We made good use of the available tools in our disposition to generate some ideas, which helped us reach a general plan of what we would be working on.

We based our project on simplicity, efficiency, and creativity to create a real approach to a digital piano to make the user enjoy the experience.

Acquired skills from our project experience:

By the end of this project, we acquired several necessary skills that extend beyond the technical aspects and that will benefit us in our upcoming projects, including:

- **Teamwork and collaboration:** communication (convey ideas, discuss solutions and address challenges), as well as team building (understanding different working styles) are the key for a successful and a robust project.
- **Time management:** managing time effectively through prioritization.
- **Problem-solving:** through consistently facing problems and overcoming challenges, we were able to enhance our critical thinking skills.
- **Planification, organization…**

After all this hard work and sleepless nights, we can proudly say that “We made it!”