CS207 Project Report

Name: Huimin LIANG
StudentID: 12210161
Class#: Lab Mon 5-6

Name: HARROLD TOK KWAN HANG
StudentID: 12212025
Class#: Lab Mon 5-6

CS207 Digital Logic
(Fall, 2023)

Southern University of Science and Technology(SUSTech), China
Students of Department of Computer Science

December 30, 2023

Contents

1 Contribution

The basic information and contribution of our members are as follow

Name & SID	Contribution	Rate
Huimin LIANG 12210161	auto-play + Recording	50%
	top module	
	Report Composition	
HARROLD TOK KWAN HANG 12212025	Learn mode	50%
	Free play	
	Report Composition	

2 Project Introduction

2.1 Project Description

This project involves the development of a digital piano emulation using a Field-Programmable Gate Array (FPGA) board. The goal is to replicate the authentic piano-playing experience by combining FPGA's programmable logic with advanced digital signal processing. The emulation includes real-time synthesis of lifelike piano sounds, responsive key interactions, and visual features, providing users with a versatile and fun digital piano alternative.

2.2 Project Background

Relevance:

In today's day and age, technology is integrated in all walks on life from healthcare to business to any form of entertainment, including music. This project combines the musical element of a traditional piano and the programmability of FPGA technology.

Objectives:

1. Accurate emulation of piano key interactions and dynamics.
2. Real-time synthesis of authentic piano sounds through FPGA-based digital signal processing.
3. Provide the user with a bug-free experience
4. Variability in terms of different modes to play.

Considerations:

1. Sound Synthesis: Implement high-quality sound synthesis algorithms that correctly reproduce the characteristics of traditional piano tones, including accurate modeling of the lasting time, period and pitch.
2. Resource Optimization: Optimize the utilization of FPGA resources to ensure efficient implementation of piano emulation algorithms.
3. User Experience (UX): Prioritize the user experience to make the piano emulation accessible and enjoyable. Consider factors such as ease of use, visual feedback, and button placement to enhance the overall user experience.
4. Responsiveness: Ensure responsive feedback to key presses, including considerations for unnecessary bounces of the button. Minimize latency in sound generation and key responsiveness to enhance the experience of the piano emulation.

2.3 Independent Design Declaration and Project Journey

The project team, leveraging their expertise and creative vision, has conceived and planned the design, then executed on the general framework while adding additional features of their own. This approach encompasses the entire development process, from the conceptualization of accurate piano key interactions to the implementation of FPGA-based real-time processing. This independent design implementation ensures that the project is driven by the team's unique perspective, goals, and technical proficiency, resulting in a customized and innovative digital piano emulation solution.

Week	Project journey
10	generate_melody and music_to_frequency modules
11	free_play and start of auto_play. Clock divider.
12	free_play_top. Buzzer started making noise. Change note pitch. Led for free play and auto play. Button debouncer. Clock debouncer.
13	learn_mode started. Auto play state machine
14	Seven segment display. Interval countdown. User accounts. Learn mode state machine
15	Finish learn_mode and auto_play. Top module. Recording.

Table 1: Timeline of Events

3 Hardware Design Approach

3.1 Ego1 Development Board And Other Hardware

Hardware	Model Type	Company Name
FPGA board	xc7a35tcsg324-1 A	XILINX
Microphone	N/A	N/A

Table 2: Hardware Models

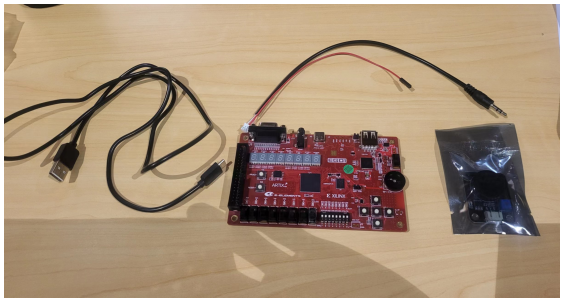
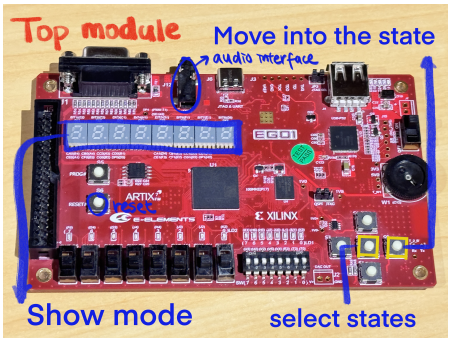


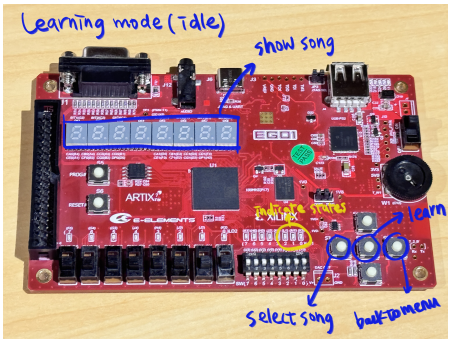
Figure 1: Hardware

3.1.1 Input/Output Value Assignment

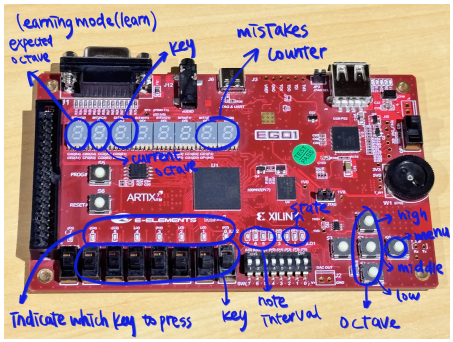


(a) Top module

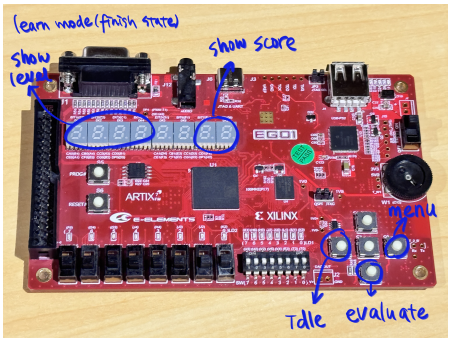
(b) auto



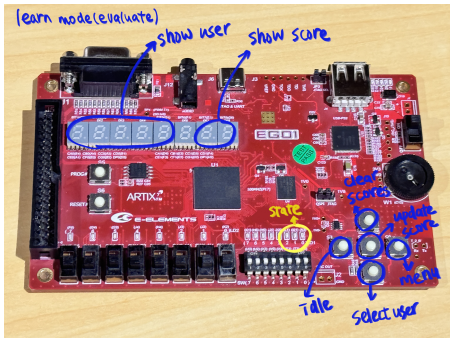
(c) learn mode idle



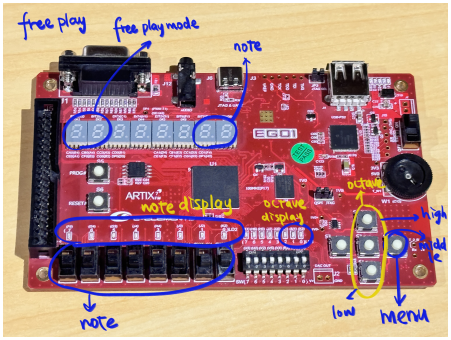
(d) learn mode learning



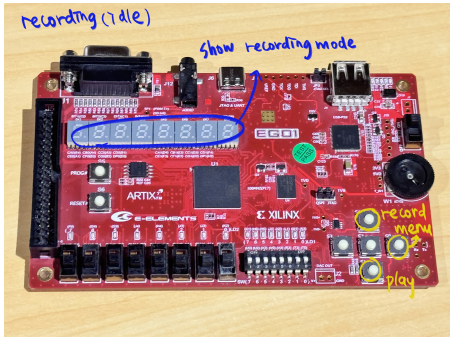
(e) learn mode finish



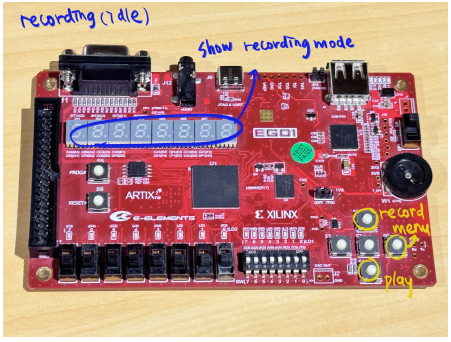
(f) learn mode evaluate



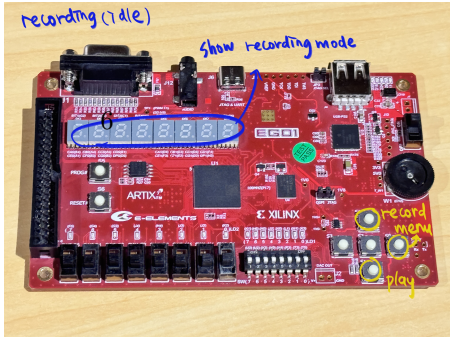
(g) free mode



(h) rcd idle



(i) rcd record



(j) rcd play

Figure 2: Input/Output Placement

4 Code Implementation Approach

4.0.1 Key Input and Output Signals

4.0.2 RTL Circuit Diagram of Top-Level Module

4.0.3 Top-Level Module

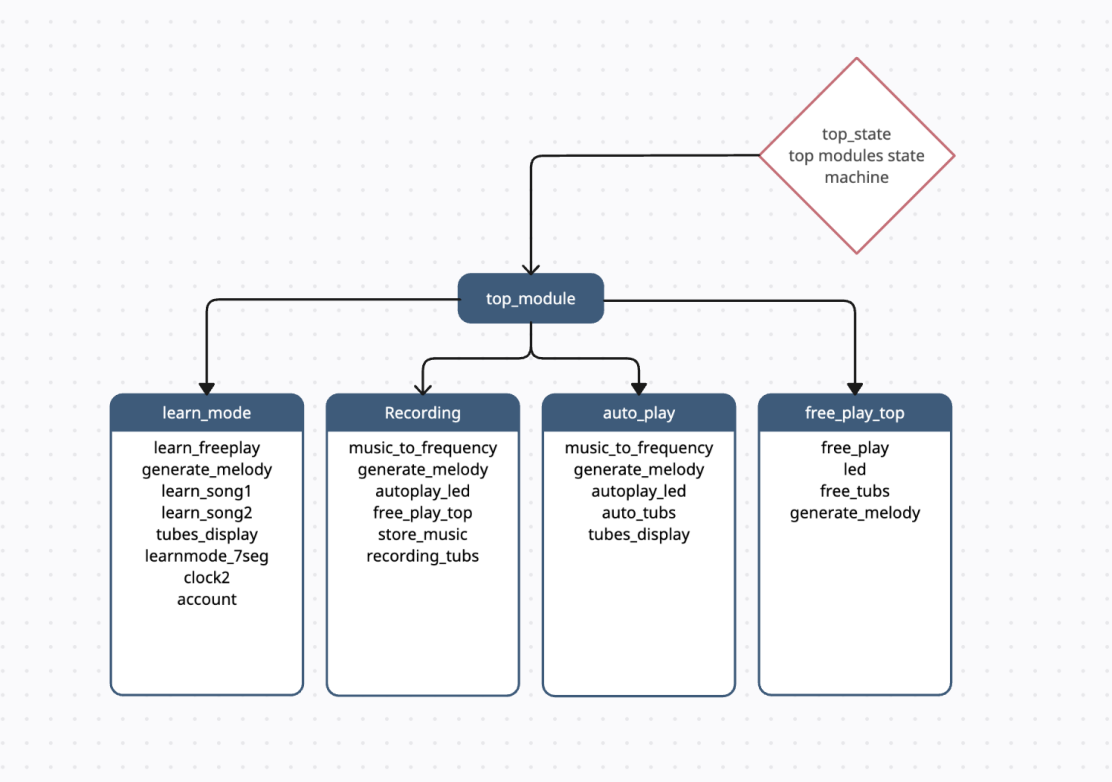


Figure 3: top module state diagram with the submodules they use

```
module top_module(
    input clk, rst,
    input btn_up,
    input btn_center,
    input btn_left,
    input btn_right,
    input btn_down,
    input [7:0] bg_sw,
    input [7:0] sm_sw,

    output [7:0] bg_led,
    output [7:0] sm_led,
    output [7:0] seg_en,
    output [7:0] seg_sel_l,
```

```
output [7:0] seg_sel_r,
output pwm,
output sd
);
```

Inputs:

- `clk`: Clock signal (100MHz).
- `reset`: Reset signal.
- `sw[7:0]`: 8-bit input for switch values.
- `btn_up`: Input for the top button.
- `btn_center`: Input for the middle button.
- `btn_left`: Input for the left button.
- `btn_right`: Input for the right button.
- `btn_down`: Input for the down button.
- `bg_sw`: 8-bit switches.
- `sm_sw`: 8-bit small switch to the right of switches (small).

Outputs:

- `[7:0] bg_led`: 8-bit output for LED above switches.
- `[7:0] sm_led`: 8-bit output for LED next to switches.
- `[7:0] seg_en`: 8-bit output for segment enables.
- `[7:0] seg_sel_l`: 4-bit outputs for 4 segments on the left.
- `[7:0] seg_sel_r`: 4-bit output for 4 segments on the right.
- `pwm`: Output for melody, creates the buzzing sound.
- `sd`: Amplifies pwm.

Signal inputs

```
wire en_auto, en_free, en_learn, en_record, en_top;

wire [7:0] seg_en1, seg_sel_l1, seg_sel_r1,
seg_en2, seg_sel_l2, seg_sel_r2,
seg_en3, seg_sel_l3, seg_sel_r3,
seg_en4, seg_sel_l4, seg_sel_r4,
seg_en5, seg_sel_l5, seg_sel_r5;

wire [7:0] bg_led1, sm_led1,bg_led3, sm_led3,bg_led2, sm_led2,bg_led4, sm_led4, bg_led5, sm_l
wire pwm1, pwm2, pwm3, pwm4, pwm5;
```

- `wire en_auto, en_learn, en_record, en_top`
  enables the functions of the different modules when they equal 1.
- `wire [7:0] seg_en1, seg_sel_l1, seg_sel_r1,`
  `seg_en2, seg_sel_l2, seg_sel_r2,`
  `seg_en3, seg_sel_l3, seg_sel_r3,`
  `seg_en4, seg_sel_l4, seg_sel_r4;`
  7-segment tube for all the different modes output.
- `wire [7:0] bg_led1, sm_led1, bg_led3, sm_led3,`
  `bg_led2, sm_led2, bg_led4, sm_led4;`
  LED for all states
- `wire pwm1, pwm2, pwm3, pwm4;`
  PWM for different states

  When in a certain state, for example learn mode, top_module's pwm = pwm4. This is the same with the seven segment tubes and led.

Top Module State Machine

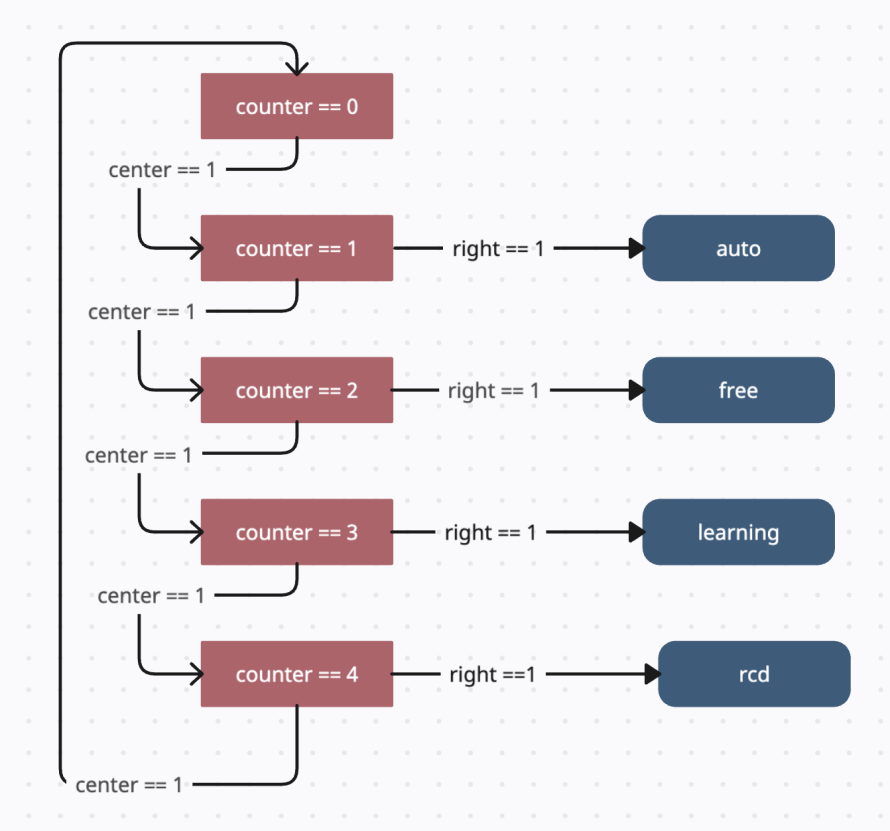


Figure 4: Top module's state machine. Center button flips through the modes. When you click right button, you enter that mode.

When a mode is selected, depending on the enable signals in the state machine, we give the output of the submodules to the output of the top module. For example, if en_auto == 1, the top module's led, pwm and seven segment gets its values from auto_play. Code is shown below.

```
    if(counter == 1) begin
        state <= auto;
        en_auto <= 1'b1;
        en_free <= 1'b0;
        en_learn <= 1'b0;
        en_record <= 1'b0;
    end
////////////////////////////////////////////////////////////////
```

```
auto: begin
    bg_led <= bg_led2;
    sm_led <= sm_led2;
    seg_en <= seg_en2;
    seg_sel_l <= seg_sel_l2;
    seg_sel_r <= seg_sel_r2;
    pwm <= pwm2;
    end
```

4.1 Detailed Principles

4.1.1 System functions list

Mode	Description
Free play	users have the freedom to play notes without any constraints. Users can play using 7 switches and change the pitch of the note with buttons, further enhacing user playability.
Auto play	Auto Play mode introduces pre-determined songs to the users, offering an automated musical experience. Users can enjoy listening to 3 musical pieces without actively playing the notes themselves. This mode provides an alternative and passive way to experience the capabilities of the FPGA-based piano.
Learn mode	Learn Mode serves as the educational part of the project, guiding users through structured songs to foster musical learning and skill development. Users follow predefined musical lessons upon successfully completing a song, users have the option to update their scores in individual accounts, encouraging a sense of accomplishment and providing a means for progress tracking.
Recording	Recording Mode allows users to become composers by allowing them to input unique combinations of notes and create their own musical sequences. Users can experiment with different melodies, rhythms, and harmonies to produce original compositions. The recorded compositions can be played back anytime, providing a platform for users to listen to and enjoy their own creations.

Table 3: Modes and their description

4.1.2 Audio Output

music to frequency The `music_to_frequency` module converts a 5-bit music input into an 11-bit frequency output. The mapping from music notes to frequencies is defined using a case statement. Each music note corresponds to a specific frequency, derived from a parameterized list. For example:

```
always @* begin
    case (music)
        // ... Other cases ...
        5'd1 : frequency = low1;
        // ... More cases ...
    endcase
end
```

The frequencies are calculated based on predefined parameters like `low1`, `middle1`, `high1`, etc. The resulting frequency is then output as an 11-bit value.

Generate Melody Module The `generate_melody` module takes clock (`clk`) and frequency (`frequency`) inputs and produces a melody output. The module calculates the period (`period`) of the corresponding PWM wave using the formula $T = \frac{1}{\text{frequency}}$.

```
always@(frequency) begin
    if(frequency == 1)
        period = 1;
    else
        period = N / (2 * frequency);
end

always @(posedge clk) begin
    if (counter < period || period == 1) begin
        counter <= counter + 1;
    end
    else begin
        pwm = ~pwm;
        counter <= 0;
    end
end
```

The module utilizes a counter to generate a PWM signal by toggling `pwm` based on the calculated period. The resulting melody is obtained by controlling the PWM wave output based on the input frequency.

4.1.3 Seven-Segment Display

In auto_play mode and learn_mode, we use the module tubes_display module to display our seven segment tubes. This section explain how we do it.

```
module tubes_display(
input clk, rst,
input [7:0] data7, data6, data5, data4, data3, data2, data1, data0,
output reg [7:0]seg_en,
```

```
    output reg [7:0]seg_out0, //left
    output reg [7:0]seg_out1 //right
    );
    `include "ppppparameters.v"

    reg clkout;
    reg [17:0] cnt;
    reg [2:0] scan_cnt;
    reg [7:0] datain;
```

Inputs:

- `clk`: Clock signal (100MHZ).
- `reset`: Reset signal.
- `data7, data6, data5, data4, data3, data2, data1, data0`: 8 bit inputs to display on the the seven segment tubes

Outputs:

- `[7:0] seg_en`: Enables the individual tubes
- `[7:0] seg_out0`: Outputs the left 4 segment tubes
- `[7:0] seg_out1`: Outputs the right 4 segment tubes

Internal signals:

- `clkout`: 1ms clock
- `cnt`: To match half the period
- `scan_cnt`: Used to light up different segment tubes
- `datain`: The data the tubes display

```
////////////////////////////////////////////////////
    always @(posedge clk, negedge rst)
    begin
        if(~rst) begin
            cnt<= 0;
            clkout <= 0;
        end else if(cnt == (period2 >>1) -1) begin
            clkout <= ~clkout;
            cnt <= 0;
        end else
```

```
        cnt <= cnt +1;
    end

    always @(posedge clkout, negedge rst) begin
        if(~rst)
            scan_cnt <= 0;
        else begin
            if(scan_cnt == 3'd7)
                scan_cnt <= 0;
            else
                scan_cnt <= scan_cnt +1;
            end
        end

    always@(scan_cnt) begin
        case(scan_cnt)
            3'b000: begin seg_en = 8'h01; datain = data0;end
            3'b001: begin seg_en = 8'h02; datain = data1;end
            3'b010: begin seg_en = 8'h04; datain = data2;end
            3'b011: begin seg_en = 8'h08; datain = data3; end
            3'b100: begin seg_en = 8'h10; datain = data4; end
            3'b101: begin seg_en = 8'h20; datain = data5; end
            3'b110: begin seg_en = 8'h40; datain = data6; end
            3'b111: begin seg_en = 8'h80; datain = data7; end
            default: begin seg_en = 8'h00; datain = 4'b0; end
        endcase
    end
```

Description:

We use a clock divider to make it slower (1ms). Then, the second always block will increment scan_cnt. The third always clock will enable each individual segment tubes and light them up. Since its 1ms, it is fast enough to display all of them to the human eye.

Example of how we display tubes:

```
    module learnmode_7seg(input [5:0] cnt,
input clk, reset, [1:0] state,  [2:0] octave, [2:0] octave_sw, [2:0] interval, [1:0] score, [3:0]
input [3:0] user0_r1, user0_r2, user1_r1, user1_r2, user2_r1, user2_r2, user3_r1, user3_r2,
input [3:0] rating1, rating2,
input counter_for_sss,
output reg [7:0] data7, data6, data5, data4, data3, data2, data1, data0
    );
    `include "ppppparameters.v"

    always @(posedge clk) begin
        if(state == learn) begin
            case(octave)
                low: data7 <= ll;
                middle: data7 <= mm;
```

```
        high: data7 <= hh;
    default: data7 <= null;
    endcase

    case(octave_sw)
        low: data6 <= ll;
        middle: data6 <= mm;
        high: data6 <= hh;
    default: data6 <= null;
    endcase
```

In this example, we use learn mode's 7 segment module learnmode_7seg. The code is as follows: When state is learn, we take the input octave, depending on the octave, we give data7 the corresponding letter H (in the parameters file, we give hh = 8'b01101100, which displays H). This can be repeated for data 6,5,4,3,2,1 and 0. We simply wire an input from learn mode top module into this module and give give data a letter we want to display. We then feed data7,6,5,4,3,2,1,0 into tube_display which allows us to display all tubes individually.

Autoplay implements a similar approach (with different inputs), where we have one module storing all the imformation to be displayed, and feed it into tubes_display to show it on the FPGA board.

4.1.4 Debouncer

In an ideal situation, when an input signal, for example a button, is pressed, it should execute its intended command once. Unfortunately, this does not play out in real life. The bouncing of a button or switch happens when it is toggled, where the contacts have to physically move from one position to another. As the components of the switch settle into their new position, they mechanically bounce, causing the underlying circuit to be opened and closed several times. This causes multiple executions of the same command, causing unintended results. To mitigate this issue, we implement a (button) debouncer.

debounce:

```
module debounce(
    input clk, rst,
    input key_in,
    output reg key_out
);
    reg pos_key;
    reg neg_key;
    reg cnt_start;
    reg [21:0] cnt;
    reg [1:0] state_key;
    reg [1:0] count; //to make the time period of 1 to be 1.5periods
    reg [1:0] count_start;
```

How debounce works:

```
always @(posedge clk)
```

```
    state_key = {state_key[0], key_out};

// Only when 20ms passed, will we assign keyin to keyout
always @(posedge clk, negedge rst) begin
    if (~rst) begin
        key_out <= 0;
        state_key = 0;
    end else if (cnt == TIME_20MS - 1)
        key_out <= key_in;
end

always @(posedge clk, negedge rst) begin
    if (~rst)
        cnt <= 22'b0;
    else if (cnt_start == 0)
        cnt <= 22'b0;
    else
        cnt <= cnt + 1;
end

always @(posedge clk, negedge rst) begin
    if (~rst)
        cnt_start <= 0;
    else if (cnt_start == 0 && key_in != key_out)
        cnt_start <= 1;
    else if (cnt == TIME_20MS - 1)
        cnt_start <= 0;
end

always @(posedge clk, negedge rst) begin
    if (~rst) begin
        pos_key = 0;
        neg_key = 0;
        count = 0;
        count_start = 0;
    end else if (state_key == 2'b10)
        count_start = 1;
    else if (state_key == 2'b01)
        count_start = 1;
end
```

Description:

the delayed output from debounce emerges precisely 20 ms following the positive edge of the clock signal. This deliberate delay is strategically introduced to synchronize with the inherent characteristics of mechanical switches, ensuring that the output reflects a stable state after the bouncing phenomenon has subsided within the initial 20-millisecond timeframe.

By orchestrating this nuanced temporal alignment, we secure a reliable and steadfast output of 1, signifying a stable switch state.

debounce2:

```
module debounce2(
    input clk,
    input rst,
    input key,
    output reg key_pflag // when the key is pressed
);
    wire key_flag; // when the key is pressed/released
    reg key_state; // the wave after debouncing
    reg key_rflag; // when the key is released
    assign key_flag = (key_pflag | key_rflag);
    reg [1:0] r_key; //

    wire pos_key;
    wire neg_key;
    assign neg_key = (r_key == 2'b10);
    assign pos_key = (r_key == 2'b01);
    reg[21:0] cnt;
    reg[1:0] state;
    parameter idle = 2'b00, p_filter = 2'b01, wait_r = 2'b10, n_filter = 2'b11;
    parameter TIME = 2_000_000;

    // every posclk, rkey1 is the last state, rkey0 is the current key(state
    always @(posedge clk)
        r_key <= {r_key[0], key};


    always @(posedge clk, negedge rst) begin
        if (~rst) begin
            state <= idle;
            cnt <= 0;
            key_pflag <= 0;
            key_rflag <= 0;
            key_state <= 0;
        end else begin
            case(state)
                idle: begin
                    key_rflag <= 0;
                    if (pos_key)
                        state <= p_filter;
                    else
                        state <= idle;
                end
                p_filter: begin
                    if (cnt < TIME - 1) begin
                        if (neg_key) begin
                            state <= idle;
                            cnt <= 0;
```

```
                    end else
                        cnt <= cnt + 1;
                end else if (cnt >= TIME - 1) begin
                    key_pflag <= 1;
                    state <= wait_r;
                    cnt <= 0;
                    key_state <= 1;
                end
            end
            wait_r: begin
                key_pflag <= 0;
                if (neg_key)
                    state <= n_filter;
                else
                    state <= wait_r;
            end
            n_filter: begin
                if (cnt < TIME - 1) begin
                    if (pos_key) begin
                        state <= wait_r;
                        cnt <= 0;
                    end else
                        cnt <= cnt + 1;
                end else if (cnt >= TIME - 1) begin
                    key_rflag <= 1;
                    state <= idle;
                    cnt <= 0;
                    key_state <= 0;
                end
            end
        endcase
    end
end

endmodule
```

Description:

Operates as a rising-edge (posedge) clock signal, strategically designed to yield a singular output within a single clock cycle. This particular configuration proves invaluable in scenarios where the objective is to discern a switch change with precision, requiring only a singular assessment during the course of the entire clock cycle. This is used in functions such as learn_mode's mistake counter to determine if the user has made a mistake.

4.1.5 Clock

```
module clock2(
input wire clk,rst,
output reg clk_1
    );
    `include "ppppparameters.v"
    reg [31:0] div_cnt;

    always @(posedge clk, negedge rst) begin
        if(~rst)
            div_cnt <= 0;
        else if(div_cnt == TIME_075SEC)
            div_cnt <= 0;
        else
            div_cnt <= div_cnt + 1;
    end

    always @(posedge clk, negedge rst) begin
        if(~rst)
            clk_1 <= 0;
        else if( div_cnt == (TIME_075SEC>>1) -1)
            clk_1 <= 1;
        else if(div_cnt == TIME_075SEC -1 )
            clk_1 <= 0;
    end
endmodule
```

Whenever we need a clock that is slower than the 100MHZ one that is built into the FPGA board, we can build a clock divider. In this case, this is a 0.75s clock divider we used to count the interval that the user has to play. 1 neat $= 0.75$s. After 0.75s has passed, the led light will switch off. We can use this powerful program to create a clock which times to our liking, thus vastly improving our options when building this project.

4.1.6 Free Play

The first mode we implemented into our design of a degital piano was free play. It is a simple design which allows the user to play freely whatever tune they want. Functions include flipping the switches to play music, pressing buttons to change the octave (pitch) and displaying the note played on the seven segment tube.

```
    module free_play_top(
 input en,
 input clk, input reset,
 input wire [2:0] button,
 input [7:0] sw,
 output melody,
 output [7:0] ledsw,
 output [7:0] ledrange,
 output [7:0] seg_en,
 output [7:0]seg_outl, //left
 output [7:0]seg_outr //right
    );


    wire [10:0] frequency;
    wire [1:0] state;
    wire [2:0] note;


    free_play play(en, clk, reset, button, sw, frequency, state, note);

    led ll(en,clk, reset, sw, state, ledsw, ledrange);

    free_tubs ft(en, clk, reset, state, note, seg_en, seg_outl, seg_outr);


    generate_melody gm(.clk(clk), .frequency(frequency), .melody(melody));
endmodule
```

Inputs:

- `clk`: Clock signal (100MHZ).
- `reset`: Reset signal.
- `sw[7:0]`: 8-bit input for switch values.
- `[2:0] button`: Selects high octave in learn state and clears user score in evaluate state.

Outputs:

- `melody`: Output signal to the buzzer to make piano tunes.
- `[7:0] ledsw`: 8-bit output for guide lights. Specifies where to play next.
- `[7:0] ledrange`: 8-bit output for segment enables.

- `seg_en`: Enables the seven segment tubes]
- `seg_outl, seg_outr`: 8-bit outputs for 7-segment data. Left 4 and Right 4.

Free Play submodule description:

In a nutshell, the module free_play outputs a frequency depending on the switch played and the pitch (high, middle, low). This frequency is wired into generate_melody where it is converted into the buzzing sound.

4.1.7 Auto Play

The second mode programmed was the autoplay. This robust module allows the user to listen to up to three different songs. The led lights up the current note that is playing which allows the user to understand how the music is played.

```
module auto_play(
    input en,
    input wire clk, rst,
    input selectsongbtn, pausebtn,
    input stopbtn, playbtn,
    output wire melody,
    output [7:0] seg_en,
    output [7:0] seg_outl, // left
    output [7:0] seg_outr, // right
    output [7:0] led, // the led to indicate note
    output [7:0] range // led to indicate range
);
    `include "ppppparameters.v"
    wire [5:0] cnt;
    reg [4:0] music;
    wire [4:0] music1;
    wire [4:0] music2;
    wire [4:0] music3;
    wire [10:0] frequency;
    reg [1:0] counter;
    reg [1:0] state; // 00 is stop, 01 is play, and 10 is pause
    // reg spause, splay, sstop; // these are states;
    wire clk1;
    wire pause, play, stop; // these are the btn after debouncing
    wire select;
    reg [1:0] music_flag;
    wire [7:0] data7, data6, data5, data4, data3, data2, data1, data0;

        clock1 cl1(clk, rst, clk1);
    song1 s1(clk1, rst,state, cnt, music1);
    song2 s2(clk1, rst, music2);
    song3 s3(clk1, rst, music3);
    autoplay_led ledd(clk, rst, music, led, range);
    auto_tubs getdata(clk, rst, state, counter, music,
```

```
data7, data6, data5, data4, data3, data2, data1, data0);
tubes_display td(clk, ~rst, data7, data6, data5, data4, data3, data2, data1, data0,
seg_en, seg_outl, seg_outr);
music_to_frequency mf(clk, music, frequency);
generate_melody gm(clk, frequency, melody);
```

Inputs:

- `en`: Enables the functions in auto_play such that it won't trigger other functions in other modes.
- `clk`: Clock signal (100MHZ).
- `reset`: Reset signal.
- `selectsongbtn`: Swaps song to play.
- `pausebutton`: Swaps state to pause.
- `stopbtn`: Swaps state to stop.
- `playbtn`: Swaps state to play.

Outputs:

- `seg_en`: enables the seven segment tubes
- `[7:0] seg_outl`: 8-bit output for seven segment tubes. 4 tubes on the left.
- `[7:0] seg_outr`: 8-bit output for seven segment tubes. 4 tubes on the right
- `led`: displays which switch is playing
- `range`: indicates high, middle, low using led

Internal Signals:

- `cnt`: Input to determine current position of music in song modules. Output is music value in that specific cnt position.
- `music`: This wire gets value from either song1, song2 or song3. Depending on counter.
- `music1, music2, music3`: Music values for song1, song2, song3.
- `frequency`: A frequency converted from music to be generated into a melody
- `counter`: to cycle through the songs.
- `state`: state for autoplay
- `clk1`: Generates a 0.3second period.
- `pause, play, stop`: Buttons to enter the corresponding state.
- `data7, data6, data5, data4, data3, data2, data1, data0`: Each data corresponds to an individual segment tube. It contains information on what to display. Refer to 7 segment tube section.

Autoplay State Machine

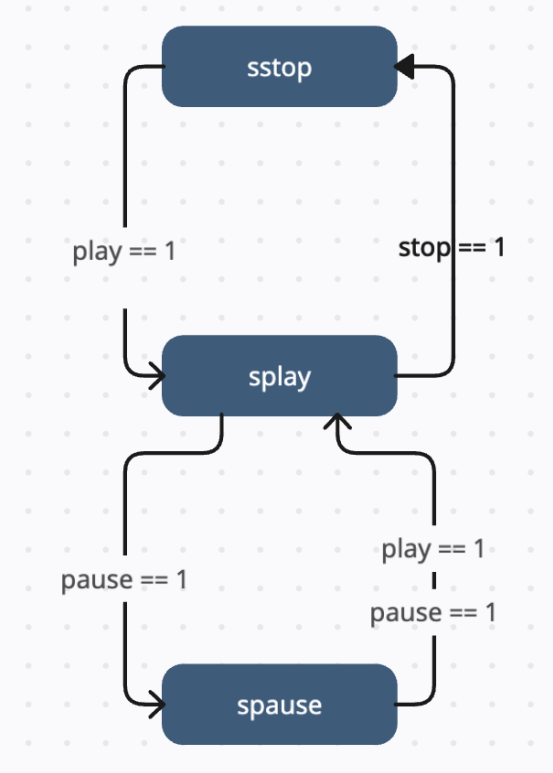


Figure 5: State Machine for auto play

Autoplay submodule description:

- `song1, song2, song3`: creates the songs to be played. Depending on the counter, music either gets music1, music2 or music3. We input the cnt and output a music to be converted to frequency then to a melody in generate_melody. Refer to music section to learn more about the topic.
- `autoplay_led`: Displays which note is currently being played as well as which octave (high, middle, low).
- `auto_tubs`: Contains information to displays on the seven segment tube including song name and note playing. We are showing data7 to data0.
- `music_to_frequency and generate_melody`: Produces a frequency depending on the music that is input. Then the frequency is converted into a melody to buzz.

4.1.8 Learn Mode

Learn mode was created with the goal to be able to teach users certain songs. There are a variety or functions built in to fully guide the user such as guide_lights, which show the user which note to play next, a mistake counter, rating and score as well as accounts to store their scores.

```
    module learn_mode(
input en, //when en is 1, the functions of learn mode are exihited in the top module
input clk, reset,
input [7:0] sw,
input topbtn, midbtn, botbtn, leftbtn,
output melody,
output reg [7:0] guide_lights,
output [7:0] seg_en,
output [7:0] seg_out0, seg_out1,
output reg [3:0] interval_led,
output reg [2:0] state_led
    );
    `inc
```

Inputs:

- `en`: Enables the functions in learn_mode such that it won't trigger other functions in other modes.
- `clk`: Clock signal (100MHZ).
- `reset`: Reset signal.
- `sw[7:0]`: 8-bit input for switch values.
- `topbtn`: Selects high octave in learn state and clears user score in evaluate state.
- `midbtn`: Transitions to learn state and selects middle octave in learn state
- `botbtn`: Transitions to evaluate state and flips through users and selects low octave in learn state.
- `leftbtn`: Returns to menu (idlelearn) and flips through songs to learn in idlelearn state.

Outputs:

- `melody`: Output signal to the buzzer to make piano tunes.
- `guide_lights[7:0]`: 8-bit output for guide lights. Specifies where to play next.
- `seg_en[7:0]`: 8-bit output for segment tubes enables. Allows us to display.
- `seg_out0, seg_out1`: 8-bit outputs for 7-segment data. seg_out0 is for the 4 on the right. seg_out1 is for the 4 on the left.

- `interval_led[3:0]`: 4-bit output for interval LED above small switches. Guides user on how long to play each note.
- `state_led[2:0]`: 3-bit output for state on the LED above the small button. Tells us which state the user is in.

Internal Signals:

```
37  reg [1:0] state; //learn mode state
38  reg [5:0] cnt; //music count
39  wire [10:0] frequency; //frequency converted from music
40  reg [4:0] music; //music either gets music1 or music2, depending on song user selected
41  wire [4:0] music1, music2; //music values of song1 and song2
42  reg [7:0] guide_lights_holder;//holds previous combination of guide_lights
43  reg [7:0] prev = 0; //previous combination of switches
44  wire [7:0] sw_d, sw_d2; //sw_d is debounced key, sw_d2 is to debounce it so that it only returns one period of the clock cycle
45  reg [2:0] interval; //length of each note to play
46  wire [2:0] interval1, interval2; //Interval value of song1 and song2
47  reg [2:0] octave; //pitch of the note
48  reg [2:0] octave_sw; //Current user octave (pitch)
49  wire clk_1sec; //a one second clock to time the interval
50  reg [2:0] countdown; //countdown the interval to 0
51  reg counter_for_sss; //To determine which song is played
52  reg [1:0] sss_shift; //To shift between the two songs
53  wire [7:0] data7, data6, data5, data4, data3, data2, data1, data0; //seven segment tube display
54  reg [3:0] digit1 = 0, digit2 = 0; //digit1 is the mistake in the ones position, digit2 is mistakes in the tens position
55  reg [1:0] score; //will display "fail", "okay", "good" and "perfect" on 7 segment tubes
56  reg [5:0] mistakes;//increments everytime a wrong key is played
57  reg [3:0] rating1, rating2;//rating1 is the rating in the ones position, rating2 is rating in the tens position
58  wire [3:0] user0_r1, user0_r2, user1_r1, user1_r2, user2_r1, user2_r2, user3_r1, user3_r2; //4 users and their ratings
59  wire [1:0] user; //The current user
60  reg [5:0] song_length; //A value that determines the end of the song
61  reg idlelearn_btn, evaluate_btn, learn_btn, userbtn, updatebtn, clearbtn; //Depending on the state, these buttons will get input button values
62  wire idlelearn_button, evaluate_button, userbtnd, updatebtnd, clearbtnd, learn_button; // Debounced buttons
63  wire hb, mb, lb; //Button for high, middle and low octave
64  reg hb2, mb2, lb2; //Debounced octave buttons
65  reg select_song_btn; //Press to change songs
66  wire select; // Debounced select_song_btn
```

Figure 6: Learn mode internal signals- An exhaustive list of the signal lines learn mode uses to express all its functionality. The figure explains the role of each signal. Therefore, due to brevity, we exclude the re-iteration of each signal line.

Learn mode submodules

```
learn_freeplay play1(clk, reset, octave_sw, sw, frequency);

generate_melody gm(.clk(clk), .frequency(frequency), .melody(melody));

learn_song1 song1(cnt, music1, interval1);
learn_song2 song2(cnt, music2, interval2);

tubes_display tb(clk, ~reset, data7, data6, data5, data4, data3, data2, data1, data0,
seg_en, seg_out1, seg_out0);

learnmode_7seg l7(cnt, clk, reset, state, octave, octave_sw, interval, score, digit1,
digit2, user, user0_r1, user0_r2, user1_r1, user1_r2, user2_r1, user2_r2, user3_r1,
user3_r2, rating1, rating2, counter_for_sss, data7, data6, data5, data4, data3, data2,
data1, data0);

clock2 c2(clk, reset, clk_1sec);

account acc(clk, reset, state, rating1, rating2, userbtnd, updatebtnd, clearbtnd,
user, user0_r1, user0_r2, user1_r1, user1_r2, user2_r1, user2_r2, user3_r1, user3_r2);
```

- learn_freeplay and generate_melody module allows us to flip the switch and play the note. Refer to description in free play and generate melody section.
- learn_song1 and learn_song2 module are the songs that we can play from. We input the cnt and output a music value and frequency, which is given to learn_mode's music and frequency to be outputted.
- tubes_display allows us to individually manipulate all 8 segment tubes to display whatever we want. learnmode_7seg contains all the information about what we want to display. Refer to seven segment section for a comprehensive description.
- clock2 is a module that produces a specific second clock period. Description is in the section clock.
- account contains all the operations regarding the users such as update rating and clear rating. Account module is describes below.

account:

```
33          always@(posedge clk, negedge reset) begin
34                  if(~reset) begin
35                      user = 0;
36                  end
37                  else if(state == evaluate) begin
38                  if(userbtn == 1) begin
39                      case(user)
40                          0: user = 1;
41                          1: user = 2;
42                          2: user = 3;
43                          3: user = 0;
44                          default: begin user = 0; end
45                      endcase
46                  end else
47                     user <= user;
48                if(updatebtn == 1) begin
49                      case(user)
50                          0: begin user0_rating1 <= rating1; user0_rating2 <= rating2; end
51                          1: begin user1_rating1 <= rating1; user1_rating2 <= rating2; end
52                          2: begin user2_rating1 <= rating1; user2_rating2 <= rating2; end
53                          3: begin user3_rating1 <= rating1; user3_rating2 <= rating2; end
54                          endcase
55                      end
56                if(clearbtn == 1) begin
57                      case(user)
58                          0: begin user0_rating1 <= 0; user0_rating2 <= 0; end
59                          1: begin user1_rating1 <= 0; user1_rating2 <= 0; end
60                          2: begin user2_rating1 <= 0; user2_rating2 <= 0; end
61                          3: begin user3_rating1 <= 0; user3_rating2 <= 0; end
62                          endcase
63                      end
64                end
65                end
```

Figure 7: account submodule

When reset, we set user to 0. When we are in the evaluate state and we press the userbtn, we scroll through the different users. If we press the updatebtn, depending on the user we are at, we change their rating. When clearbtn is pressed, it resets the users score to 0. We can only enter the evaluate state from the finish state, such that we can only update the score after the song is played.

Learn mode functions:

Octave Switch:

```
always @(posedge clk) begin
            if(hb == 1) begin octave_sw <= high; end
            else if(mb == 1) begin octave_sw <= middle; end
            else if (lb ==1 ) begin octave_sw <= low; end
       end
```

Description:

If hb is pressed, the octave played will be high, thus playing a higher pitch. Same for mb and lb.

Swap Song:

```
always @(posedge clk, negedge reset) begin
            if(~reset)
                sss_shift <= 2'b00;
            else
                sss_shift <= {sss_shift[0], counter_for_sss};
        end
```

Description:

A shift register which detects a change in counter-for-sss.

State Machine:

```
always @(posedge clk, negedge reset) begin
    if (~reset) begin
        state_led <= 3'b111;
        state <= idle_learn;
    end
    else begin
        case (state)
            idle_learn: begin
                state_led <= 3'b111;
                select_song_btn <= leftbtn;
                learn_btn <= midbtn;
                if (learn_button == 1) begin
                    state <= learn;
                end
            end
            learn: begin
                state_led <= 3'b001;
                hb2 <= topbtn;
                mb2 <= midbtn;
                lb2 <= botbtn;
                if (cnt >= song_length) begin
                    state <= finish;
                end
            end
            finish: begin
                state_led <= 3'b010;
                idlelearn_btn <= leftbtn;
                evaluate_btn <= botbtn;
                if (idlelearn_button == 1) begin
                    state <= idle_learn;
                end
                if (evaluate_button == 1) begin
                    state <= evaluate;
                end
            end
            evaluate: begin
                state_led <= 3'b100;
                userbtn <= botbtn;
                updatebtn <= midbtn;
                idlelearn_btn <= leftbtn;
                clearbtn <= topbtn;
                if (idlelearn_button == 1) begin
                    state <= idle_learn;
                end
            end
            default: state <= idle_learn;
        endcase
```

```
    end
end
```

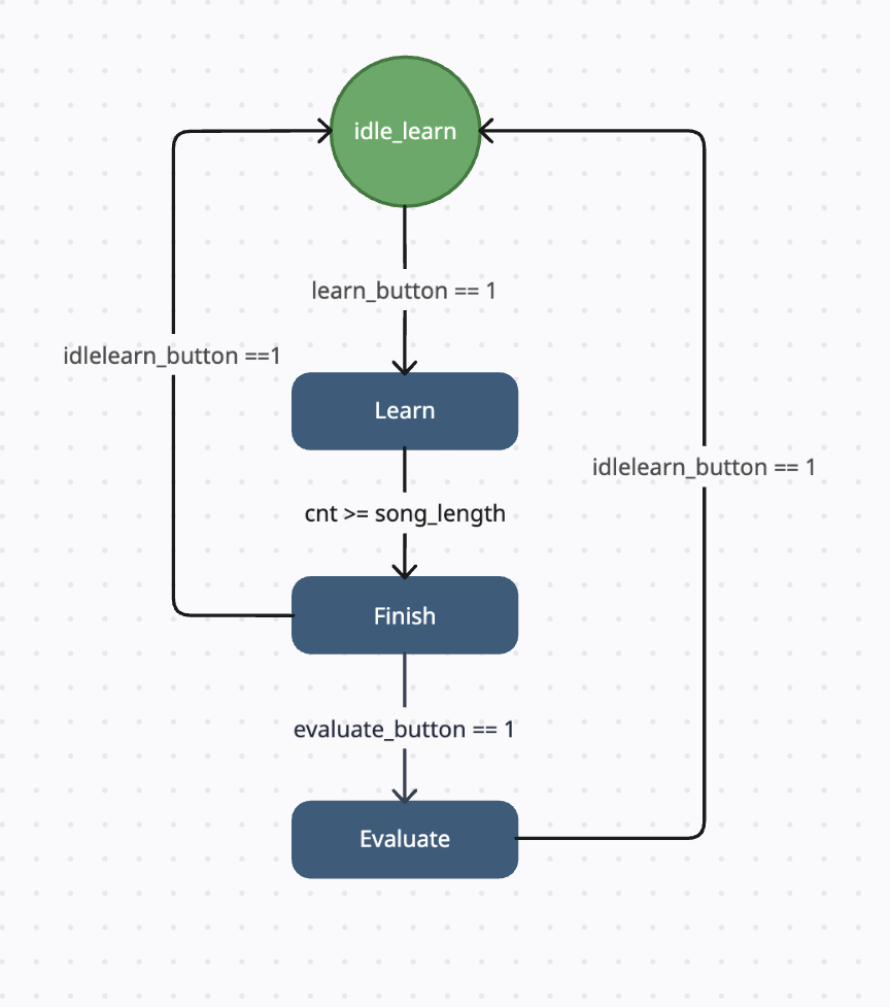


Figure 8: Learn mode state diagram - For ease of the eye, we decided to provide an explanation in terms of a diagram. We start in idle_learn to select our song. Once selected, we press learn_button to enter learn mode. After we finish playing the song, we transition to finish. From there, we can either go back to the menu and play another song or we can enter evaluate state to update the user account score. From evaluate, we press the ide_learn button to return back.

guide-lights function

```
always @(posedge clk, negedge reset) begin
    if (~reset) begin
        cnt <= 0;
    end
    else if (state == idle_learn) begin
        if (sss_shift[1] ^ sss_shift[0]) begin
            cnt <= 0;
            guide_lights <= 8'b0000_0000;
        end
    end
    else if (state == learn) begin
        guide_lights <= 8'b0000_0000;
        case (music)
            5'd1: begin guide_lights[0] <= 1'b1; octave <= low; end
            5'd2: begin guide_lights[1] <= 1'b1; octave <= low; end
            5'd3: begin guide_lights[2] <= 1'b1; octave <= low; end
            5'd4: begin guide_lights[3] <= 1'b1; octave <= low; end
            5'd5: begin guide_lights[4] <= 1'b1; octave <= low; end
            5'd6: begin guide_lights[5] <= 1'b1; octave <= low; end
            5'd7: begin guide_lights[6] <= 1'b1; octave <= low; end
            5'd8: begin guide_lights[0] <= 1'b1; octave <= middle; end
            5'd9: begin guide_lights[1] <= 1'b1; octave <= middle; end
            5'd10: begin guide_lights[2] <= 1'b1; octave <= middle; end
            5'd11: begin guide_lights[3] <= 1'b1; octave <= middle; end
            5'd12: begin guide_lights[4] <= 1'b1; octave <= middle; end
            5'd13: begin guide_lights[5] <= 1'b1; octave <= middle; end
            5'd14: begin guide_lights[6] <= 1'b1; octave <= middle; end
            5'd15: begin guide_lights[0] <= 1'b1; octave <= high; end
            5'd16: begin guide_lights[1] <= 1'b1; octave <= high; end
            5'd17: begin guide_lights[2] <= 1'b1; octave <= high; end
            5'd18: begin guide_lights[3] <= 1'b1; octave <= high; end
            5'd19: begin guide_lights[4] <= 1'b1; octave <= high; end
            5'd20: begin guide_lights[5] <= 1'b1; octave <= high; end
            5'd21: begin guide_lights[6] <= 1'b1; octave <= high; end
            default: guide_lights <= 8'b0000_0000;
        endcase

        // LED will switch to the next light when the correct note is played with the right frequ
        if (guide_lights[0] == 1'b1) begin
            if (prev == not_playing_note) begin
                if ((sw_d[0] == 1'b1 & octave_sw == low & music == 5'd1) |
                    (sw_d[0] == 1'b1 & octave_sw == middle & music == 5'd8) |
                    (sw_d[0] == 1'b1 & octave_sw == high & music == 5'd15)) begin
                    cnt <= cnt + 1;
                    guide_lights[0] <= 1'b0;
                end
            end
        end
```

```
        // This is repeated for sw[1] to sw[6]

        prev <= sw_d;
    end
end
```

Description:

When we reset. We set cnt to 0 and reset the guide-lights. If the state is in idle-learn and we detect a shift in the sss-shift value, the song will change and we set the count back to 0 and reset guide-lights. We need to do this because we always want the song to start from the very beginning

In the case block, the guide-lights are determined by the music, which gets its value from either music1 or music2. For example, regardless of the octave, the note 'do' will always be at the first switch. Therefore, if music == 1, 8 or 15, we light up switch 1.

After that, we determine if the correct switch (note) is played. We do this by detecting a few conditions.

1. The guide-light in that position is on (== 1).
2. our initial switch position is all 0 (sw = 8'b0000-0000 (not_playing_note).
3. octave is the same combination as octave_sw. (e.g music == 1, octave_sw == low we have to play a low octave.)

Once all these conditions are met, we increment the count and set the guide-light in that position to 0. Then the guide-light moves on.

Mistake Counter:

```
always @(posedge clk or negedge reset) begin
    if (~reset) begin
        digit1 <= 0;
        digit2 <= 0;
        guide_lights_holder <= guide_lights;
    end
    else begin
        if (sw_d2 == 8'b00000001 | sw_d2 == 8'b00000010 | sw_d2 == 8'b00000100 |
        sw_d2 == 8'b00001000 | sw_d2 == 8'b00010000 | sw_d2 == 8'b00100000 |
            sw_d2 == 8'b01000000) begin
            if ((sw_d !== guide_lights_holder |
            (sw_d == guide_lights_holder & octave !== octave_sw))
            & sw_d !== not_playing_note) begin
                if (guide_lights_holder == guide_lights) begin
                    if (digit1 == 4'd9) begin
                        digit1 <= 4'd0;
```

```
                    digit2 <= digit2 + 1;
                end
                else begin
                    digit1 <= digit1 + 1;
                end
                guide_lights_holder <= 8'b0000_0000;
            end
        end
    end
    if (sw_d == not_playing_note) begin
        guide_lights_holder <= guide_lights;
    end
end
```

Description:

When we reset, we set digit1 and digit2 to 0. digit1 is the mistake in the ones position and digit2 is the mistake in the tens position(digit2 == 2, digit1 == 6, total mistakes == 26) If we detect a change in the switch, meaning the user has played a note, we check for these conditions.

1. if the switch combination is not equal guide_lights_ holder.
2. If they are equal but the octave is not equal to the octave user has set.

If either condition (1) or (2) is met, we increment digit1. guide_light_holder simply holds the current combination of guide_light, even after guide_lights has changed. This is because if the switch is played correctly, guide-light changes, this will unnecessarily increment the number of mistakes.

Change Song

```
always @(posedge clk, negedge reset) begin
    if (~reset) begin
        counter_for_sss <= 0;
    end
    else if (select == 1) begin
        case (counter_for_sss)
            0: counter_for_sss <= 1;
            1: counter_for_sss <= 0;
        endcase
    end
    else
        counter_for_sss <= counter_for_sss;
end

always @(posedge clk, negedge reset) begin
    if (~reset) begin
        music <= 5'b0;
    end
    else begin
```

```
        case (counter_for_sss)
            0: begin music <= music1; interval <= interval1; song_length <= s1_length; end
            1: begin music <= music2; interval <= interval2; song_length <= s2_length; end
            default: begin music <= music1; interval <= interval1; song_length <= s1_length; end
        endcase
    end
end
```

Description

When we press reset. counter-for-ss is set to 0. When we press select, it switches counter-for-sss between 0 and 1. When counter-for-sss is 0, we set music ¡- music1 and interval ¡- interval1 and song-length ¡- song-length1. When counter-for-sss is 1, we set music ¡- music2 and interval ¡- interval2 and song-length ¡- song-length2.

Countdown

```
always @(posedge clk_1sec) begin
    if (sw_d == not_playing_note) begin
        countdown <= interval;
        case (countdown)
            1: interval_led <= 4'b1000;
            2: interval_led <= 4'b1100;
            3: interval_led <= 4'b1110;
            4: interval_led <= 4'b1111;
            default: interval_led <= 4'b0000;
        endcase
    end
    else begin
        countdown <= countdown - 1;
        case (countdown)
            1: interval_led <= 4'b1000;
            2: interval_led <= 4'b1100;
            3: interval_led <= 4'b1110;
            4: interval_led <= 4'b1111;
            default: interval_led <= 4'b0000;
        endcase
        if (countdown == 0) begin
            countdown <= interval;
        end
    end
end
```

Description:

This function utilizes the 1 second clk. In actual fact, we set each clock period to roughly 0.75seconds, this is to emulate length of 1 beat on a piano, otherwise called a crotchet in music theory. When we are not playing any note, the lights will just light up and not go out. When we play an note, the countdown decrements every clock cycle. This gives the user a feel of how long to play each note.

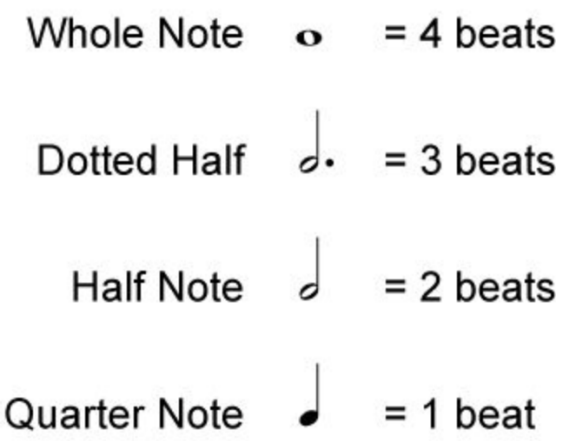


Figure 9: 1 clock cycle of clk_1sec represents 1 beat

Mistake and Rating calculator

```
always @(posedge clk or negedge reset) begin
    if (~reset) begin
        rating1 <= 0;
        rating2 <= 0;
    end
    else begin
        mistakes <= 10 * digit2 + digit1;
        if (14 - digit1 < 10) begin
            rating1 <= 14 - digit1;
        end
        else
            rating1 <= 4 - digit1;

        if (digit1 > 4) begin
            rating2 <= 6 - digit2 - 1;
        end
        else
            rating2 <= 6 - digit2;

        if (mistakes == 6'd0) begin
            score <= 3;
        end
        else if (mistakes == 5'd1 | mistakes == 5'd2 | mistakes == 5'd3 | mistakes == 5'd4 | mist
            score <= 2;
        end
        else if (mistakes == 5'd6  | mistakes == 5'd7 | mistakes == 5'd8 | mistakes == 5'd9) begi
            score <= 1;
        end
        else begin
            score <= 0;
        end
```

```
    end
end
```

Description:

Our rating system is simply 64 minus the number of mistakes made. 64 was chosen because cnt has 6 bits and 2 to the power of 6 is 64. This function simply converts digit1 and digit2 to the number of mistakes as well as the final score (64 - mistakes). It is done in this way because digit1 and digit2 are seperate numbers. So we first calculate the ones position then calculate the two's position. Depending on the mistakes made, the user is also given a score value which displays how well the user played the song ("Perfect" means 0 mistakes made).

5 Bonus Implementations

In our project, we decided to implement two bonus modules which we feel enhance the user experience the most in terms of a piano.

5.1 Variable Duration for Each Note

```
case(cnt)
  6'd0    : begin  music <= 5'd13;  end
  6'd1    :   begin music <= 5'd14;  end
  6'd2    :   begin music <= 5'd13;  end
  6'd3    :   begin music <= 5'd12;  end
  6'd4    :   begin music <= 5'd11;  end
  6'd5    :   begin music <= 5'd14;  end
  6'd6    :   begin music <= 5'd13;  end
  6'd7    :   begin music <= 5'd12;  end

  6'd8    :   begin music <= 5'd11;  end
  6'd9    :   begin music <= 5'd11;  end
```

How it works:

As section song describes, as cnt increments at a rate of 1 cnt per 0.5seconds, the music will play according to its value at cnt. Our way of making the notes longer is simply to string many of the same music values together, such as in cnt == 8 and 9. This creates the effect of longer lasting notes as music == 11 is played 2 * 0.5 = 1second long.

If we want to play the same music value next to each other but we want it to be distinct, all that is required is to add a music == 0 between them, which allows for a 0.5second pause before the same note is played again, creating the effect of playing the same note multiple times, at 0.5seconds each. Below is an example taken from song3 module.

```
case(cnt)
  6'd0    : begin  music <= 5'd1;  end
  6'd1    :   begin music <= 5'd0;  end
  6'd2    :   begin music <= 5'd1;  end
  6'd3    :   begin music <= 5'd0;  end
```

```
6'd4    :   begin music <= 5'd5;  end
6'd5    :   begin music <= 5'd0;  end
6'd6    :   begin music <= 5'd5;   end
```

5.2 Recording Mode

In recording mode, we utilize Vivado's Block RAM (BRAM) Configuration feature

```
module recording(
input en,
input clk, rst,
input [7:0] sw,
input btn_up,
input btn_center,
input btn_left,
input btn_down,

output reg [7:0] led,
output reg [7:0] range,
output reg pwm,
output reg [7:0] seg_en,
output reg [7:0]seg_outl, //left
output reg [7:0]seg_outr //right
);
```

Inputs:

- `en`: Output signal to the buzzer to make piano tunes.
- `clk`: 100MHZ clock.
- `rst`: reset the cnt
- `[7:0] sw`: 8-bit switches to play notes
- `btn_up`: Plays high octave in play mode. Also the button to enter play mode of record module.
- `btn_center`: Plays the middle octave.
- `btn_left`: button to go to record menu state.
- `btn_down`: button to enter record state rcd and change to low octave.

Outputs:

- `[7:0] led`: 8-bit output for led above big switches
- `[7:0] range`: 8-bit output for led above small switches
- `pwm`: output the melody to make a noise
- `seg_en`: Allows seven segment to display data
- `seg_outl, seg_outr`: 8-bit outputs for seven-segment data. seg_outr is for the 4 on the right. seg_outl is for the 4 on the left.

Internal Signals

```
wire up, center, down, left;
wire [11:0]frequency;

reg [3:0] counter;//0.1
reg [31:0] count;
reg count_start;

wire [7:0]led1,led2, ledrange1,ledrange2;

wire pwm1, pwm2;

wire [7:0] seg_en1, seg_outl1, seg_outr1,
seg_en2, seg_outl2, seg_outr2,
seg_en0, seg_outl0, seg_outr0;

reg clkout;
 reg [31:0] cnt_clk; //the get the clkout
 reg [5:0] cnt_note;
 reg [3:0] cnt_interval;

 reg [4:0] music0;
```

- `up, center, down, left`: buttons
- `[3:0] counter`: to count the length of time the player plays
- `frequency`: converted from music in the recording mode to be generated into a melody.
- `[31:0] count`: to count 0.1 second
- `count_start`: a signal allow the count to start to count
- `led1,led2`: the led of play and record
- `ledrange1, ledrange2`: the octave led of play and record
- `pwm1, pwm2`: the sound wave of play and record
- `[7:0] seg_en1, seg_outl1, seg_outr1, seg_en2, seg_outl2, seg_outr2, seg_en0, seg_outl0, seg_outr0;`: the 7 segment tubes of menu, record and play states
- `cnt_clk`: to generate a 0.1s clock
- `cnt_note`: which note in the music array the machine is playing
- `cnt_interval`: to count how many 0.1s has the the recorder plays one note
- `[4:0] music0`: the note the machine is playing

Submodules

```
autoplay_led al(clk, rst, music0, led2, ledrange2);

free_play_top fpt(1,clk,rst,{btn_up, btn_center, btn_down},sw,pwm1,led1,ledrange1,seg_en
store_music sm(clk, sw, octave, music1);

recording_tubs(clk, rst, state, music0,seg_en0,seg_outl0,seg_outr0, seg_en2,seg_outl2, s
```

We only include these submodules for simplicity. Other submodules used in recording such as generate_melody is explained elsewhere in this report.

- `autoplay_led`: Utilizes autoplay's led since playing a recorded song is the same function.
- `free_play_top`: Uses free play's note playing capabilities
- `store_music`: stores the music from the record mode to be played during play mode.
- `recording_tubs`: Stored information to display on seven segment tubes.

Recording the music:

```
always @(posedge clk, negedge rst) begin
    if(~rst)
        cnt <= 7'b0;
    else if( state == record)begin
        if(note > 0)
        cnt <= cnt + 2;
        else cnt <= cnt;
    end
end

always@(note_de1) begin
    if(note_de1 == 8'b0) begin //when note goes back to 0000_oooo
        count_start = 1'b0;
    end else if(note_de1 > 0 && state == record)begin
        count_start = 1'b1;
    end
end

integer i;
 always @(posedge clk, negedge rst) begin
    if(~rst) begin
        count <= 32'b0;
        counter <= 0;
        interval[0] <= 1;
        for( i = 0; i < 100; i = i+1) begin
            music[i] <= 0;
            interval[i] <= 0;
```

```
            end
        end else if(en) begin
        if(state == idle && up == 1) begin
            for( i = 0; i < 100; i = i+1) begin
                        music[i] <= 0;
                        interval[i] <= 0;
                    end
        end
        if(state == record) begin
            if(count >= 100_000_00) begin
                counter <= counter +1;
                count <= 0;
            end else if(count_start)
            count <= count + 1;
            end

            if(count_start) begin
            music[cnt] <= music1;
            music[cnt + 1] <= 0; end
             else if(note == 8'b0)
            counter <= 16'b0;


            if(note_de2)begin
                if(count < 100_000_00 & counter == 0) begin
                    interval[cnt] <= 1;
                end else begin
                    interval[cnt] <= counter;
                end
                counter <= 0;
            end
        end
    end
```

Description:

Recording consist of three blocks.

1. Counter Initialization for Recording: The second block initializes a counter cnt to zero during a reset. During the recording state, if the note signal is greater than zero, the counter is incremented by 2 to store the musical notes inputted by the user into an array. We increment by two to put a music == 0 between two notes to create the effect of a pause.

2.Start/Stop Counter Based on Note Signals: The third block, triggered by changes in the note_de1 signal, controls the start and stop of a counter (count_start). When note_de1 becomes zero, count_start is set to 0. If note_de1 is greater than zero and the system is in the recording state, count_start is set to 1.

3.Music Initialization and Recording Logic: The final block initializes various music-related vari-

ables during a reset, including count, counter, interval array, and music array. It handles the recording of musical notes music during the recording state, updating the music and interval arrays based on the count and counter values. We can store a maximum of 100 notes before playing.

It also manages the counter count used to keep track of time during recording, incrementing the counter and updating the counter variable based on specific conditions. The recorded notes and their durations are stored in the music and interval arrays.

Reading the music:

```
//build a .1s clk
    always @(posedge clk, negedge rst)
begin
    if(~rst) begin
        cnt_clk<= 0;
        clkout <= 0;
    end else if(cnt_clk == (period3 >>1) -1) begin
        clkout <= ~clkout;
        cnt_clk <= 0;
    end else
        cnt_clk <= cnt_clk +1;
end

always @(posedge clkout, negedge rst) begin
    if(~rst) begin
        music0 <= 5'b0;
        cnt_note <= 0;
        cnt_interval <= 0;
    end else if(en) begin
       if(state == play) begin
       music0 <= music[cnt_note];
        if(music[cnt_note] == 0) begin
            cnt_interval <= cnt_interval + 2;
            end
        if(cnt_interval < interval[cnt_note] & music[cnt_note] !== 0)begin
            cnt_interval <= cnt_interval + 1;
        end else if(cnt_interval == interval[cnt_note])begin
           cnt_note <= cnt_note + 1;
           cnt_interval <= 0;
        end
    end
    else if(state == idle | state == record) begin
        music0 <= 5'b0;
    end
    end
end
```

Description

Reading has two blocks with two different functions to make the whole thing work.

1.Clock Generation: The first part of the code generates a clock signal clkout with a period of 0.1 seconds. This is achieved by counting clock cycles cnt_clk and toggling clkout when the condition is met. The clock generation block also includes a reset mechanism rst, initializing the counters and output to their initial states when we reset.

2. Music Player: The second part of the code implements a 'music player' that plays a sequence of musical notes. The musical notes are stored in an array music. The player advances through the notes based on a counter cnt_note and we control the duration of each note using another counter cnt_interval, this allows the player to play the song to the exact input length that the user played. As we increment through the array, we give the value at the position in the array to music0 to output as melody.

6 Project Summary and Conclusion

Summary of team collaboration, development, and testing work

This project group consist on two people. Throughout the project, collaboration was smooth and no disagreements were made. Majority of the time spent on the project was spent together, this facilitated discussion among us and allowed two minds to solve the same problem, thus speeding up the completion of the project.

Our method for testing was having one test on simulation while the other tests on board which provided us with much needed data on where our mistakes were which allows swift correction. If we were ever stuck on an issue, we would generally turn to the teaching assistants for advice.

Development had a slow start as we were figuring out the intricacies of the FPGA board with initial limited knowledge. As the weeks pass and we learned more from the course, development was a lot faster and more work was done. You could think of the development as an exponential curve.

Conclusion

The aim of our project revolves around emulating the classical piano experience on an FPGA board, offering users a multi-functional musical experience through the implementation of four distinctive modes. These modes, labeled as "free_play," "auto_play," "learn_mode," and "recording," are strategically crafted to cater to diverse musical preferences and learning approaches.

In "free_play," users are have the freedom to play notes however they want. This mode encourages unrestrained musical exploration, providing a platform for users to experiment with different melodies and harmonies. "auto_play" introduces pre-determined songs, presenting users with an automated musical experience.

The educational facet of our project is encapsulated in the "learn_mode," which guides users through structured songs, fostering musical learning and skill development. Upon successfully completing a song, users have the option to update their scores in individual accounts, promoting a sense of accomplishment and progress tracking.

The "recording" mode empowers users to compose their own musical sequences by inputting unique combinations of notes. This recorded composition can be played back seamlessly, allowing users to experiment with and enjoy their original creations.

While additional features such as VGA integration and customizable key selections could potentially enhance the user experience, practical constraints, including limited time and busy schedules, necessitated a focused scope for our project. Despite these challenges, the project has served as an enriching experience, refining our teamwork, time management, and critical thinking skills. Overall, our endeavor underscores a commitment to delivering a comprehensive and enjoyable musical platform on FPGA, providing users with a diverse and immersive piano experience.

6.1 Future Project Ideas

1. Remote control car
2. Pokemon Battle (provided with VGA)
3. Whack-A-Mole game
4. Minesweeper
5. Solitaire
6. tetris

7 References

State Machine Diagram Maker

https://creately.com/

General Project References

https://blog.csdn.net/m0_37921318/article/details/105890194
https://blog.csdn.net/Winter233333333/article/details/127818948
https://blog.csdn.net/ricky_hust/article/details/9766385
https://www.eetree.cn/project/detail/506
https://blog.csdn.net/weixin_44181627/article/details/106637511
https://blog.csdn.net/qq_39507748/article/details/108770459