CDA 4203/4203L
Spring 2024
Computer System Design

Final Project Report
Due by 11:59PM, 1 May

Today's Date:	04/16/2024	
Team Member Names:	Aidan Khaliil Christian Lovetere Evan Inge	
Your U Numbers:	U9240-8495 U4648-9387 U6421-9670	
Work Distribution	Briefly explain each team member's contribution. Aidan Khalil – Worked to ensure lab report responses' completeness and correctness. Christian Lovetere – Worked to ensure interface operated appropriately on FPGA board Evan Inge – Worked to combine all codes into working interface	

Feedback: Your feedback is extremely important to improve the mini-project for future course offerings.

Total Number of Person Hours Spent:	*Estimate the number of hours spent by each team member and add the three numbers.* *20hrs+*
Exercise Difficulty: (Easy, Average, Hard)	Hard
Issues You ran into:	*List all problem/issues you faced while doing this project.* *(please use bulletized list)* • *Microphone issues recording audio* • *Combining all the working codes into the necessary interface* • *Determining if the FPGA board was working as expected* • *Recording / playback / volume issues.*
Any Suggestions to improve this project:	*Clearer steps to be followed. Though the struggle of doing it without clear instructions forces learning – it also creates a barrier for most and takes up much more time than we'd originally predicted.*
Any Other Feedback:	Some small discrepancies between instructions and the required steps taken, such as mentions of .xise files when the file type was different, causing confusion for our team.

Put the link to your demonstration Video here (on YouTube, etc.). Make sure you show as much as you could achieve in the Video.

https://drive.google.com/file/d/1pl-FMf7zIBKo58JOKrZ5B6Jc4t_Iqqqy/view?usp=drivesdk

Overview

Describe the overall functionality of the system. Describe any relevant interfaces. Maximum 1 page.

This project constructs a versatile audio management system that capitalizes on the combined functionality of a picoBlaze microcontroller, a high-fidelity audio codec, and dynamic RAM storage to handle and manipulate audio data. The system's primary capabilities encompass capturing audio through a microphone, storing these recordings, facilitating playback, and providing user-driven options to manage audio files, such as deletion and volume adjustment.

There are 3 buttons used: BTN0, BTN1, BTN2. These buttons are for scroll up, scroll down, and confirm, respectively. So, each menu displays the options for itself, for example, when you want to play a track, the options are 1.) track 1, 2.) track 2 and so on. You can use BTN1 to scroll down to 2 and then press BTN2 to confirm, and play track 2 this way.

The menu begins in a master menu with options to select between play, record, delete, delete all, and volume settings. You can play, record, and delete up to 5 tracks. You can set the volume to low, medium, or high, which are all intervals of 10 decibels, from 0 to 20. To close the program, simply close the terminal tab being utilized to interface with the FPGA.

Interactions with the system are mediated through a sophisticated user interface that leverages a serial terminal for command input and menu navigation, augmented by intuitive hardware controls including buttons and visual indicators like LEDs. This setup ensures a user-friendly experience, allowing effortless access to the system's features. The interface design carefully considers user ergonomics, ensuring that all functionalities are easily accessible through logical command sequences and clear visual feedback, making the system both powerful and approachable for users of varying technical proficiency.

System Block Diagram

Description:

Microphone & Speaker: The primary devices for audio input and output.

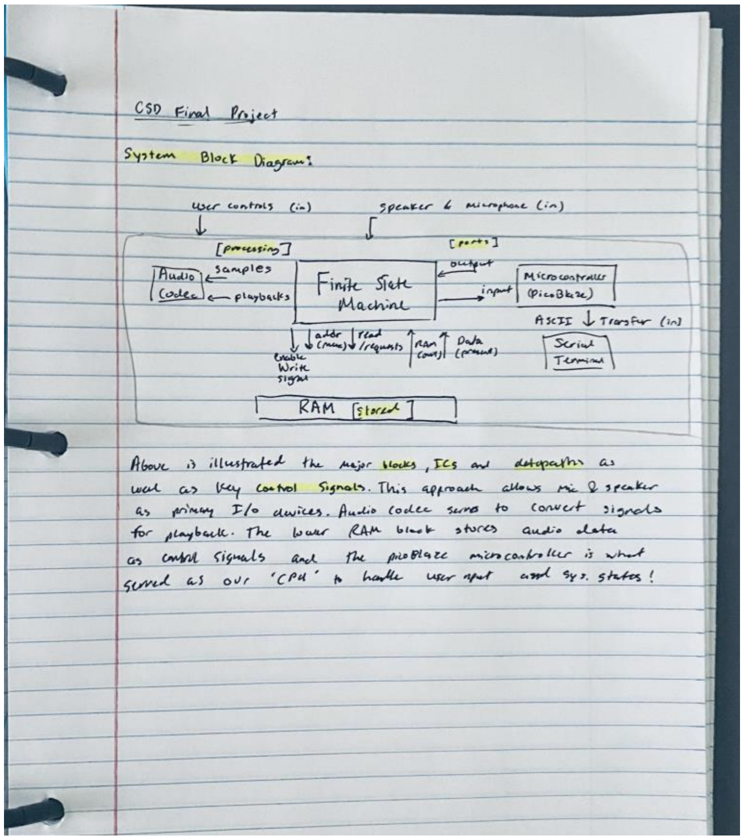
Audio Codec: Converts analog signals from the microphone to digital for processing and storage, and digital back to analog for playback through the speaker.

picoBlaze Microcontroller: Serves as the central processing unit, handling user inputs, system states, and controlling other components.

RAM: Stores audio data as digital signals that can be accessed for playback or deletion.

User Interface: Includes a serial terminal and physical buttons/LEDs for direct user interaction. Each component should be linked by data paths showing the flow of audio and control signals, indicating directions for audio data, user commands, and system feedback.

Illustration:



CSD Final Project

System Block Diagram:

Above is illustrated the major blocks, ICs and datapaths as well as key control signals. This approach allows mic & speaker as primary I/O devices. Audio codec serves to convert signals for playback. The lower RAM block stores audio data as control signals and the picoBlaze microcontroller is what served as our 'CPU' to handle user input and sys. states!

Codes

Include source codes. The length of this will vary widely; depending on how much of your design’s functionality is in software.

For our project, we had one top module called `Controller` that we used. It initializes instances of the Ram, Sockit Top module, UART, Picoblaze, and two clock wizards. This selection will include the Controller.v and our program’s .psm file since all of the other code in the project was given. (Starts on next page)

Controller.v

```
input   reset;
input up, down;
output wire STATLED3, STATLED4;
output reg highLED, medLED, lowLED;
output reg LEDRAM;
reg [7:0] state;



// ram inputs and outputs
output            hw_ram_rasn;
output            hw_ram_casn;
output            hw_ram_wen;
output [2:0]  hw_ram_ba;
inout             hw_ram_udqs_p;
inout             hw_ram_udqs_n;
inout             hw_ram_ldqs_p;
inout             hw_ram_ldqs_n;
output            hw_ram_udm;
output            hw_ram_ldm;
output            hw_ram_ck;
output            hw_ram_ckn;
output            hw_ram_cke;
output            hw_ram_odt;
output [12:0] hw_ram_ad;
inout  [15:0] hw_ram_dq;
inout             hw_rzq_pin;
inout             hw_zio_pin;
input  [3:0]  switches;                  // address
wire                 status;
reg    [15:0] RAMin;
wire     [15:0] RAMout;
reg    [25:0] address;
reg                       enableWrite;
reg                  reqRead;
reg                       ackRead;
wire                      dataPresent;
wire   [25:0] max_ram_address;
reg    [15:0] mem_in;
reg    [15:0] mem_out;
reg    [29:0] count;


// Audio Codec Wires
inout             AUD_ADCLRCK;
input             AUD_ADCDAT;
inout             AUD_DACLRCK;
output            AUD_DACDAT;
output            AUD_XCK;
inout             AUD_BCLK;
output            AUD_I2C_SCLK;
inout             AUD_I2C_SDAT;
```

```
output            AUD_MUTE;
wire                      PLL_LOCKED;
reg     [1:0]     volume_control;
reg                       playback;
wire  [15:0]  audio_output;
wire  [1:0]   sample_avaliable;
wire  [1:0]   sample_request;
reg   [15:0]  tmpDATA;
reg                       sample_a_check;
reg                       sample_req_check;


// RS232 Lines
input                     rs232_rx;
output            rs232_tx;


// PicoBlaze Data Lines
wire    [7:0]     pb_port_id;
wire    [7:0]     pb_out_port;
reg     [7:0]     pb_in_port;
wire                      pb_read_strobe;
wire                      pb_write_strobe;
// PicoBlaze CPU Control Wires
wire                      pb_reset;
wire                      pb_interrupt;
wire                      pb_int_ack;


// UART wires
wire                    write_to_uart;
wire                    uart_buffer_full;
wire                    uart_data_present;
reg                     read_from_uart;
wire                    uart_reset;
wire    [7:0]   uart_rx_data;


//Clock wires
wire                    clk50;
wire                    clk11;
wizard myclock (
    .CLK_IN1(clk37),
    .CLK_OUT1(clk100mhz_pico),
         .CLK_OUT2(clk100mhz_audio),
         .CLK_OUT3(clk50),
         .CLK_OUT4(clk11)
);
```

```
//FSM registers and wires
reg force_start;
reg delete_all, pause, change_v, skip, skip_p, skip_ack;
reg deleted_all;
reg [2:0] curr_track;
reg high, med, low;
reg [25:0]max_ram_address_track1, max_ram_address_track2, max_ram_address_track3, max_ram_address_track4, max_ram_address_track5;
reg [25:0]address_track1, address_track2, address_track3, address_track4, address_track5;
reg [25:0] max_address;
reg record_track1, record_track2, record_track3, record_track4, record_track5;
reg play_track1, play_track2, play_track3, play_track4, play_track5;
reg delete_track1, delete_track2, delete_track3, delete_track4, delete_track5;

initial begin
        record_track1 <= 0; //set all recordings to 0
        record_track2 <= 0;
        record_track3 <= 0;
        record_track4 <= 0;
        record_track5 <= 0;
        play_track1 <= 0;   //set the play track values to 0
        play_track2 <= 0;
        play_track3 <= 0;
        play_track4 <= 0;
        play_track5 <= 0;
        delete_track1 <= 0;  //set the delete track values to 0
        delete_track2 <= 0;
        delete_track3 <= 0;
        delete_track4 <= 0;
        delete_track5 <= 0;
        delete_all <= 0;
        pause <= 0;
        count <= 0;
        volume_control <= 1;
        change_v <= 0;
        playback <= 0;
        address <= 0;     //reset the address of the ram
        state <= 8'h00;   //reset the state to 0
        enableWrite <= 0;
        LEDRAM <= 1'b0;
        max_address <= 26'h0FFFFFF;  //max address is the max 26 bit hex value
        max_ram_address_track1 <= 26'h0325AA0;  // max address for the first track
        max_ram_address_track2 <= 26'd6700000;  // max address for the second track
        max_ram_address_track3 <= 26'd10200000; //max address for the third track
        max_ram_address_track4 <= 26'd13600000; //max address for the fourth track
        max_ram_address_track5 <= 26'd16700000; //max address for the fifth track
        address_track1 <= 26'd0; //starting address for the first track
        address_track2 <= 26'd3400000; //starting address for the second track
        address_track3 <= 26'd6900000; //starting address for the third track
        address_track4 <= 26'd10300000; //starting address for the fourth track
        address_track5 <= 26'd13700000; //starting address for the fifth track
        sample_a_check <= 0;
   sample_req_check <= 0;
        tmpDATA <= 0;
        high <= 0;
        med <= 0;
        low <= 0;
        skip <= 0;
        skip_p <= 0;
        curr_track <= 0;
        skip_ack <= 0;
        highLED <= 0;
        medLED <= 0;
        lowLED <= 1;
        force_start <= 0;
end


//assign the resets, interrupts, and write signals to uart and state reg
assign pb_reset = ~reset;
assign uart_reset =  ~reset;
assign pb_interrupt = 1'b0;
assign write_to_uart = pb_write_strobe & (pb_port_id == 8'h03);
assign write_to_state_reg = pb_write_strobe & (pb_port_id == 8'h08);



//every clock signal (from the main picoblaze 100Mhz clock) take the input values into the picoblaze ports
always @(posedge clk100mhz_pico or posedge pb_reset)
        begin
                if(pb_reset) begin
                        pb_in_port <= 0;
                        read_from_uart <= 0;
                end else begin
                        // Set pb input port to appropriate value
                        case(pb_port_id)
                                8'h00: pb_in_port <= {7'b0000000,up};
                                8'h01: pb_in_port <= {7'b0000000,down};
                                8'h06: pb_in_port <= {7'b0000000,sel};
                                8'h02: pb_in_port <= uart_rx_data;
                                8'h04: pb_in_port <= {7'b0000000,uart_data_present};
                                8'h05: pb_in_port <= {7'b0000000,uart_buffer_full};
                                8'h09: pb_in_port <= {7'b0000000,deleted_all};
                                8'h0A: pb_in_port <= {7'b0000000,skip_ack};
                                8'h0B: pb_in_port <= {7'b0000000,force_start};
                                default: pb_in_port <= 8'h00;
                        endcase
                        // Set up acknowledge/enable signals.
                        //
                        // Some modules, such as the UART, need confirmation that the data
                        // has been read, since it needs to push it off the queue and make
                        // [illegible]
                        // the next byte available. This logic will set the 'read_from'
                        // signal high for corresponding ports, as needed. Most input
                        // ports will not need this.
                        read_from_uart <= pb_read_strobe & (pb_port_id == 8'h04);
                        if(write_to_state_reg) begin
                                record_track1 <= (pb_out_port == 8'h01);
                                record_track2 <= (pb_out_port == 8'h09);
                                record_track3 <= (pb_out_port == 8'h0A);
                                record_track4 <= (pb_out_port == 8'h0B);
                                record_track5 <= (pb_out_port == 8'h0C);
                                play_track1 <= (pb_out_port == 8'h00);
                                play_track2 <= (pb_out_port == 8'h05);
                                play_track3 <= (pb_out_port == 8'h06);
                                play_track4 <= (pb_out_port == 8'h07);
                                play_track5 <= (pb_out_port == 8'h08);
                                delete_track1 <= (pb_out_port == 8'h0D);
                                delete_track2 <= (pb_out_port == 8'h0E);
                                delete_track3 <= (pb_out_port == 8'h0F);
                                delete_track4 <= (pb_out_port == 8'h10);
                                delete_track5 <= (pb_out_port == 8'h11);
                                pause <= (pb_out_port == 8'h03);
                                delete_all <= (pb_out_port == 8'h04);
                                change_v <= (pb_out_port == 8'h12);
                                high <= (pb_out_port == 8'h13);
                                med <= (pb_out_port == 8'h14);
                                low <= (pb_out_port == 8'h15);
                                skip <= (pb_out_port == 8'h16);
                                skip_p <= (pb_out_port == 8'h17);
                        end
                end
        end


//instantiate the picoblaze module using the 100Mhz clock)
picoblaze pblaze (
        .port_id(pb_port_id),
        .read_strobe(pb_read_strobe),
        .in_port(pb_in_port),
        .write_strobe(pb_write_strobe),
        .out_port(pb_out_port),
        .interrupt(pb_interrupt),
        .interrupt_ack(),
        .reset(pb_reset),
        .clk(clk100mhz_pico)
 );

//instantiate the ram wrapper.
ram_interface_wrapper RAMRapper (
        .address(address),                              // input of the address
        .data_in(RAMin),                                // 16 bit input to pass into the ram from the audio codec
        .write_enable(enableWrite),                     // input, enable when recording
        .read_request(reqRead),                         // input, should be high to be 'playing'
        .read_ack(ackRead),                             //
        .data_out(RAMout),                              // output from ram to wire RAMout
        .reset(0),
        .clk(clk),
        .hw_ram_rasn(hw_ram_rasn),
        .hw_ram_casn(hw_ram_casn),
        .hw_ram_wen(hw_ram_wen),
        .hw_ram_ba(hw_ram_ba),
        .hw_ram_udqs_p(hw_ram_udqs_p),
        .hw_ram_udqs_n(hw_ram_udqs_n),
        .hw_ram_ldqs_p(hw_ram_ldqs_p),
        .hw_ram_ldqs_n(hw_ram_ldqs_n),
        .hw_ram_udm(hw_ram_udm),
        .hw_ram_ldm(hw_ram_ldm),
        .hw_ram_ck(hw_ram_ck),
        .hw_ram_ckn(hw_ram_ckn),
        .hw_ram_cke(hw_ram_cke),
        .hw_ram_odt(hw_ram_odt),
        .hw_ram_ad(hw_ram_ad),
        .hw_ram_dq(hw_ram_dq),
        .hw_rzq_pin(hw_rzq_pin),
        .hw_zio_pin(hw_zio_pin),
        .clkout(clk37),                      //uses the 37.5Mhz clock
        .sys_clk(clk37),
        .rdy(status),
        .rd_data_pres(dataPresent),
        .max_ram_address(max_ram_address),
        .STATLED3(STATLED3),
        .STATLED4(STATLED4)
);


//instantiate the uart using the 100Mhz clock
rs232_uart urt (
                .tx_data_in(pb_out_port),
                .write_tx_data(write_to_uart),
                .tx_buffer_full(uart_buffer_full),
                .rx_data_out(uart_rx_data),
                .read_rx_data_ack(read_from_uart),
                .rx_data_present(uart_data_present),
                .rs232_tx(rs232_tx),
                .rs232_rx(rs232_rx),
                .reset(uart_reset),
                .clk(clk100mhz_pico)
);
```

```
// Audio Codec Interface Instantiation that uses the 100Mhz clock
sockit_top audio (
    .clk(clk100mhz_audio),
         .audio_clk(clk11),
         .main_clk(clk50),
         .playback(playback),
         .volume_control(volume_control),
    .AUD_ADCLRCK(AUD_ADCLRCK),
    .AUD_ADCDAT(AUD_ADCDAT),
    .AUD_DACLRCK(AUD_DACLRCK),
    .AUD_DACDAT(AUD_DACDAT),
    .AUD_XCK(AUD_XCK),
    .AUD_BCLK(AUD_BCLK),
    .AUD_I2C_SCLK(AUD_I2C_SCLK),
    .AUD_I2C_SDAT(AUD_I2C_SDAT),
    .AUD_MUTE(AUD_MUTE),
    .PLL_LOCKED(PLL_LOCKED),
    .KEY(1),
    .SW(switches),
         .audio_in(mem_out),    //input
         .audio_out(audio_output), //output
         .sample_end(sample_avaliable),
         .sample_req(sample_request)
);


//name the states for the FSM
parameter main_menu = 8'h00;
parameter main_record = 8'h01;
parameter main_play = 8'h02;
parameter main_pause = 8'h03;
parameter delete_all_messages = 8'h04;
parameter record_part2 = 8'h05;
parameter record_part3 = 8'h06;
parameter play_part2 = 8'h07;
parameter play_part3 = 8'h08;
parameter play_part4 = 8'h09;
parameter play_part5 = 8'h0A;
parameter del_all_part2 = 8'h0B;
parameter del_all_part3 = 8'h0C;
parameter del_all_part4 = 8'h0D;
parameter delete_message = 8'h0F;
parameter delete_message_part2 = 8'h10;
parameter delete_message_part3 = 8'h11;
parameter delete_message_part4 = 8'h12;
parameter v_control = 8'h13;
parameter skip_play = 8'h14;
```

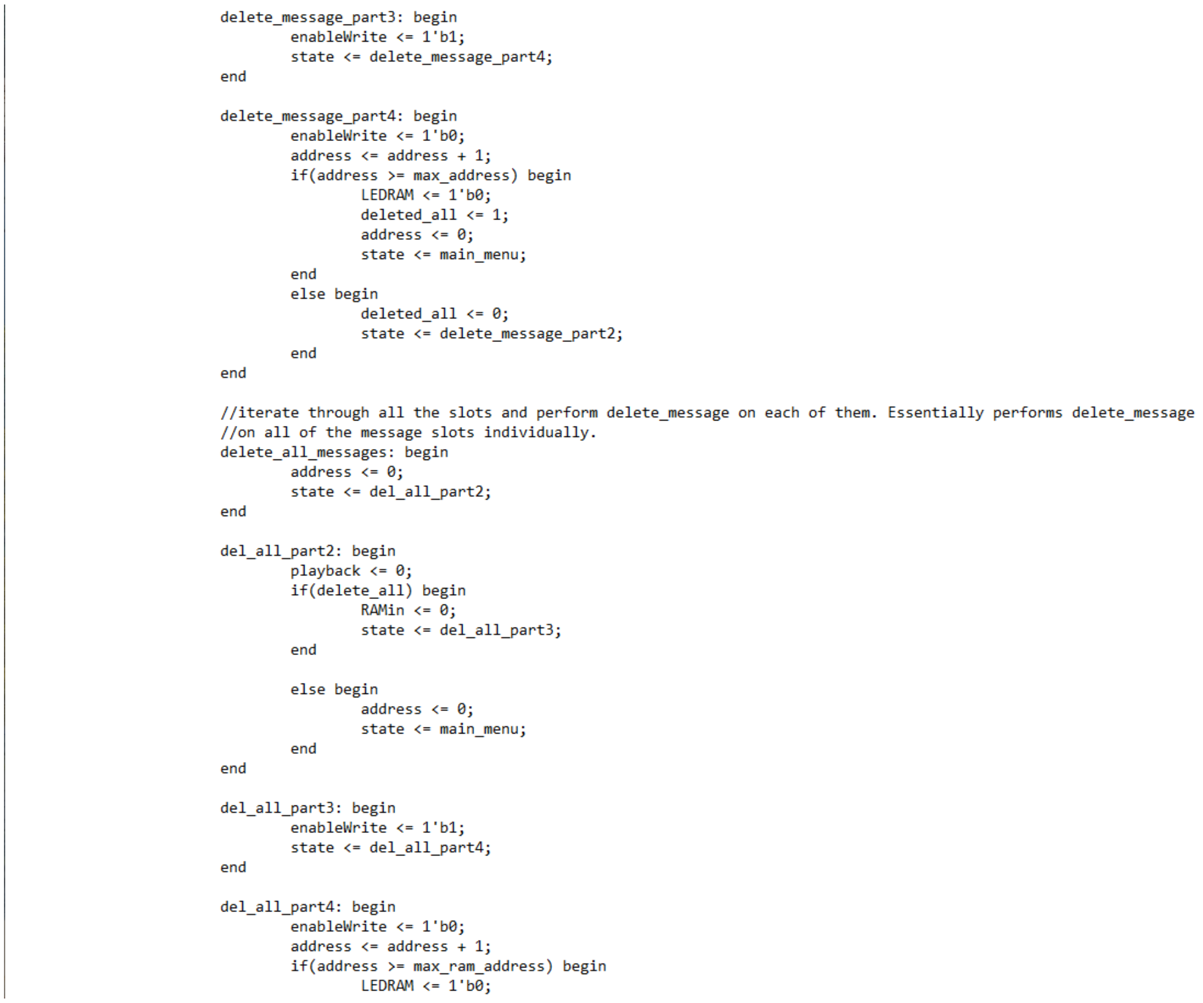
```
// Memory FSMD
always@(posedge clk37) begin
        if(~reset) begin
                address <= 0;
        end

        //5 second welcome message. Counts to 5 and then force starts if nothing is selected
        else if(~force_start) begin
                if(count == 30'd500000000 || sel || up || down) begin
                        count <= 30'd500000000;
                        force_start <= 1;
                end
                else begin
                        count <= count + 1;
                end
        end

        else if(status)begin

                //states for the FSM
                case (state)
                        //MAIN STATE: changes the address and max address values for the selected track to record and then moves to the main_record state
                        main_menu: begin
                                if(record_track1) begin
                                        address <= address_track1;
                                        max_address <=  max_ram_address_track1;
                                        state <= main_record;
                                end

                                else if(record_track2) begin
                                        address <= address_track2;
                                        max_address <= max_ram_address_track2;
                                        state <= main_record;
                                end

                                else if(record_track3) begin
                                        address <= address_track3;
                                        max_address <= max_ram_address_track3;
                                        state <= main_record;
                                end

                                else if(record_track4) begin
                                        address <= address_track4;
                                        max_address <= max_ram_address_track4;
                                        state <= main_record;
                                end

                                else if(record_track5) begin
                                        address <= address_track5;
                                        max_address <= max_ram_address_track5;
        max_address <= max_ram_address_track5;
        state <= main_record;
end


//if the option to play a track is selected and the address is not at the max ram address, set the current track, change the address and
else if(play_track1 && (address < max_ram_address)) begin
        curr_track <= 3'b001;
        address <= address_track1;
        max_address <= max_ram_address_track1;
        state <= main_play;
end
else if(play_track2 && (address < max_ram_address)) begin
        curr_track <= 3'b010;
        address <= address_track2;
        max_address <= max_ram_address_track2;
        state <= main_play;
end
else if(play_track3 && (address < max_ram_address)) begin
        curr_track <= 3'b011;
        address <= address_track3;
        max_address <= max_ram_address_track3;
        state <= main_play;
end
else if(play_track4 && (address < max_ram_address)) begin
        curr_track <= 3'b100;
        address <= address_track4;
        max_address <= max_ram_address_track4;
        state <= main_play;
end
else if(play_track5 && (address < max_ram_address)) begin
        curr_track <= 3'b101;
        address <= address_track5;
        max_address <= max_ram_address_track5;
        state <= main_play;
end


//if the option to delete the track is selected, set the addresses and move to the delete message state
else if(delete_track1) begin
        address <= address_track1;
        max_address <=  max_ram_address_track1;
        state <= delete_message;
end

else if(delete_track2) begin
        address <= address_track2;
        max_address <= max_ram_address_track2;
        state <= delete_message;
end

else if(delete_track3) begin
        address <= address_track3;
```

```
else if(delete_track3) begin
        address <= address_track3;
        max_address <= max_ram_address_track3;
        state <= delete_message;
end

else if(delete_track4) begin
        address <= address_track4;
        max_address <= max_ram_address_track4;
        state <= delete_message;
end

else if(delete_track5) begin
        address <= address_track5;
        max_address <= max_ram_address_track5;
        state <= delete_message;
end

//if pause is selected move to the main_pause method

else if(pause) begin
        state <= main_pause;
end


//if the volume change option is selected, go to the volume control state
else if(change_v) begin
        state <= v_control;
end

//if delete all is selected move to the delete all messages state
else if(delete_all) begin
        address <= 0;
        state <= delete_all_messages;
end


// if skip is selected, check the track and move the addresses accordingly, and then go to the skip_play state
else if(skip) begin
        if (curr_track == 3'b001) begin
                curr_track <= 3'b010;
                address <= address_track2;
                max_address <= max_ram_address_track2;
                state <= skip_play;
        end

        if (curr_track == 3'b010) begin
                curr_track <= 3'b011;
                address <= address_track3;
                max_address <= max_ram_address_track3;
                state <= skip_play;
        end
```

```
                if (curr_track == 3'b011) begin
                        curr_track <= 3'b100;
                        address <= address_track4;
                        max_address <= max_ram_address_track4;
                        state <= skip_play;
                end

                if (curr_track == 3'b100) begin
                        curr_track <= 3'b101;
                        address <= address_track5;
                        max_address <= max_ram_address_track5;
                        state <= skip_play;
                end

                if (curr_track == 3'b101) begin
                        curr_track <= 3'b001;
                        address <= address_track1;
                        max_address <= max_ram_address_track1;
                        state <= skip_play;
                end
        end

        else begin
                state <= main_menu;
        end
        playback <= 0;
end

//the skip play case sets the acknowledge for skipping.
skip_play: begin
   skip_ack = 1;
        if(skip_p) begin
                skip_ack = 0;
                state <= main_play;
        end
        else begin
                state <= skip_play;
        end
end


//volume control sets the volume based on the input from the menu and sets the LEDs, and then goes back to the main menu
v_control: begin
                if(high) begin
                        volume_control <= 3;
                        highLED = 1;
                        medLED = 1;
                        lowLED = 1;
                        state <= main_menu;
                end
                else if(med) begin
                        volume_control <= 2;
                        highLED = 0;
                        medLED = 1;
                        lowLED = 1;
                        state <= main_menu;
                end
                else if(low) begin
                        volume_control <= 1;
                        highLED = 0;
                        medLED = 0;
                        lowLED = 1;
                        state <= main_menu;
                end
                else begin
                        state <= v_control;
                end
        end

//Recording state - here we check if recording has been requested. If so, record to the first memory address, then loop and increment mem
//address until we reach the end of the memory or 'stop' or 'done' is asserted.
main_record: begin
        playback <= 0;
        LEDRAM <= 1'b0;
        if(record_track1 || record_track2 || record_track3  || record_track4 || record_track5) begin
                if(sample_avaliable == 0) begin
                        sample_a_check <= 1;
                end
                if(sample_avaliable && sample_a_check) begin
                        sample_a_check <= 0;
                        address <= address + 1;
                        enableWrite <= 1'b1;
                        RAMin <= audio_output;
                        state <= record_part2;
                end
                else begin
                        state <= main_record;
                end
        end
        else begin
                state <= main_menu;
                address <= 0;
        end
end

record_part2: begin
        enableWrite <= 1'b0;
        if(address >= max_address) begin
                LEDRAM <= 1'b1;
                state <= main_menu;
        end
```

```
	else begin
		state <= main_record;
	end
end


//Playback state - If any play slot is asserted, move audio data from RAM to memory, and play each address one by one in sequential order.
//This plays all of the audio in the same order it was recorded in. Audio will loop if 'done' or 'pause' is not asserted.
main_play: begin
	playback <= 1;
	if(play_track1 || play_track2 || play_track3 || play_track4 || play_track5 || skip_p) begin
		state <= play_part2;
	end

	else begin
		state <= main_menu;
	end
end


play_part2: begin
	enableWrite <= 1'b0;
	reqRead <= 1'b1;
	ackRead <= 1'b0;
	address <= address + 1;
	state <= play_part3;
end

play_part3: begin
	reqRead <= 1'b0;
	if(dataPresent) begin
		tmpDATA <= RAMout;
		ackRead <= 1'b1;
		state <= play_part4;
	end

	else begin
		reqRead <= 1'b1;
		state <= play_part3;
	end
end

play_part4: begin
	ackRead <= 1'b0;
	if(sample_request == 0) begin
		sample_req_check <= 1;
	end
	if(sample_request && sample_req_check) begin
		sample_req_check <= 0;
		mem_out <= tmpDATA;
		state <= play_part5;
```

```
        end
        else begin
                state <= play_part4;
        end
end

play_part5: begin
        ackRead <= 1'b0;
        if (address >= max_address) begin
                state <= main_menu;
        end

        else begin
                state <= main_play;
        end
end

//When pause is asserted, the recording pauses and the place in memory is maintained by looping this state until pause is not asserted
main_pause: begin
        if(pause) begin
                state <= main_pause;
        end

        else if (play_track1 || play_track2 || play_track3 || play_track4 || play_track5) begin
                state <= main_play;
        end

        else begin
                address <= 0;
                state <= main_play;
        end
end

//loop through all of the memory addresses for the specified message and set all values to 0. stop when we are outside the bounds of the
//memory for that slot.
delete_message: begin
        state <= delete_message_part2;
end

delete_message_part2: begin
        playback <= 0;
        if(delete_track1 || delete_track2 || delete_track3  || delete_track4 || delete_track5) begin
                RAMin <= 0;
                state <= delete_message_part3;
        end

        else begin
                address <= 0;
                state <= main_menu;
        end
end
```



```
            delete_message_part3: begin
                enableWrite <= 1'b1;
                state <= delete_message_part4;
            end

            delete_message_part4: begin
                enableWrite <= 1'b0;
                address <= address + 1;
                if(address >= max_address) begin
                    LEDRAM <= 1'b0;
                    deleted_all <= 1;
                    address <= 0;
                    state <= main_menu;
                end
                else begin
                    deleted_all <= 0;
                    state <= delete_message_part2;
                end
            end

            //iterate through all the slots and perform delete_message on each of them. Essentially performs delete_message
            //on all of the message slots individually.
            delete_all_messages: begin
                address <= 0;
                state <= del_all_part2;
            end

            del_all_part2: begin
                playback <= 0;
                if(delete_all) begin
                    RAMin <= 0;
                    state <= del_all_part3;
                end

                else begin
                    address <= 0;
                    state <= main_menu;
                end
            end

            del_all_part3: begin
                enableWrite <= 1'b1;
                state <= del_all_part4;
            end

            del_all_part4: begin
                enableWrite <= 1'b0;
                address <= address + 1;
                if(address >= max_ram_address) begin
                    LEDRAM <= 1'b0;
```

```
                    LEDRAM <= 1'b0;
                    deleted_all <= 1;
                    address <= 0;
                    state <= main_menu;
                end
                else begin
                    deleted_all <= 0;
                    state <= del_all_part2;
                end
            end

        endcase
    end
end


endmodule
```

.psm file:

Because we have a lot of different menus in our psm file, we have provided a pastebin link to our psm code. Our file has the implementation of mapping the buttons on the Anvyl board to specific functions in our code. We have a button to reset, one to increment the option numbers in the menu, and one to select an option. This is passed from the UART and connects with our verilog in order to run with the verilog FSMD.

https://pastebin.com/b6A11Cza

Discussion and Conclusions

In this section, discuss how you arrived at your major decisions. Discuss any aspects of your design that you would do differently if you had it all to do over again, and why. Include any thoughts you have on the project, how it could have been done differently or better, etc.

In developing this project, several key decisions were based on the requirements for functionality, ease of use, and reliable performance. The choice to use a picoBlaze microcontroller was driven by its suitability for managing I/O operations and its compatibility with our FPGA platform. The integration of an audio codec was essential for achieving high-quality audio signal processing.

If given another opportunity to approach this design, considerations would be made to simplify the system architecture to enhance maintainability and reduce potential for errors. For example, consolidating some of the control logic could minimize the complexity of state management. Additionally, optimizing clock synchronization across different components could improve system efficiency and reduce latency issues observed with multiple clock domains.

Overall, the project provided valuable insights into system design and the challenges of integrating multiple hardware and software components. Future iterations could benefit from a more streamlined design and perhaps the inclusion of advanced features like wireless control or expanded memory capacity. Also as discussed on the cover page, we believe more straightforward instructions would have streamlined this project, but because it was more open to interpretation of groups – more problems had arisen than originally expected.