6.111 Final Project:
Gesture Controlled Music Player

Elaine Pham & Tiffany Tran

Table of Contents

I. Overview:

The purpose of our project is to create a music player that can be controlled using hand gestures. By simply tilting their wrist, the user will be able to pause and play songs, change tracks, and increase or decrease the volume.

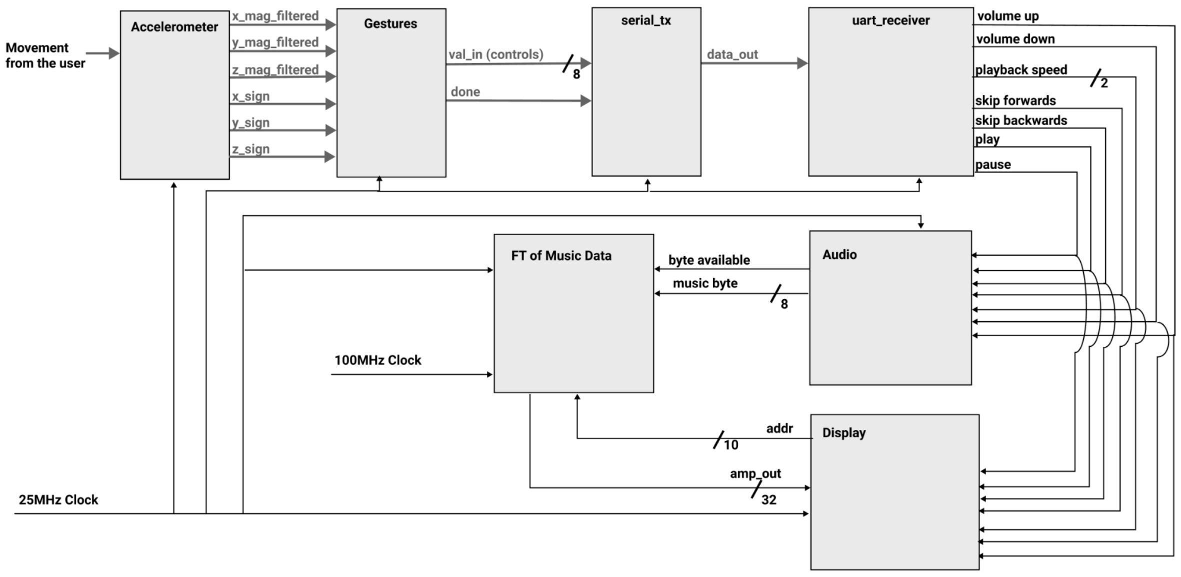
Using the accelerometer built in the FPGA, we can identify what movements the user makes in the x, y, and z directions. After the accelerometer detects what gesture was made, the controls that the gesture corresponds with will be set to high and sent to a HC-05 Bluetooth Module which will transmit that data to a receiver Bluetooth Module. The receiver HC-05 Bluetooth Module is connected to a different FPGA that will control music playback and graphics displayed on the monitor.

When the music control signal is received, the FPGA updates the graphics and audio playback correspondingly. If a play signal is received, 8-bit music data from an SD card will be played at 48kHz. The graphics on the monitor will change as the song is playing. In addition to displaying the song title, gesture instructions, volume, speed, music bar, and time elapsed in the song, the graphics will showcase a Fourier Transform (FT) of the music data using vertical colored bars of varying heights.

II. Other Devices: Bluetooth Modules

To send the data from one FPGA to another wireless, I used two HC-05 Bluetooth modules and paired them using an Arduino Micro (Board was set to Arduino Leonardo in Arduino). Using the SerialPassthrough example code, I ran the code for both modules. I set the role of one of them to master while I set the role of the other to slave. Then, I binded them (with both of them on at the same time).

III. Total Block Diagram



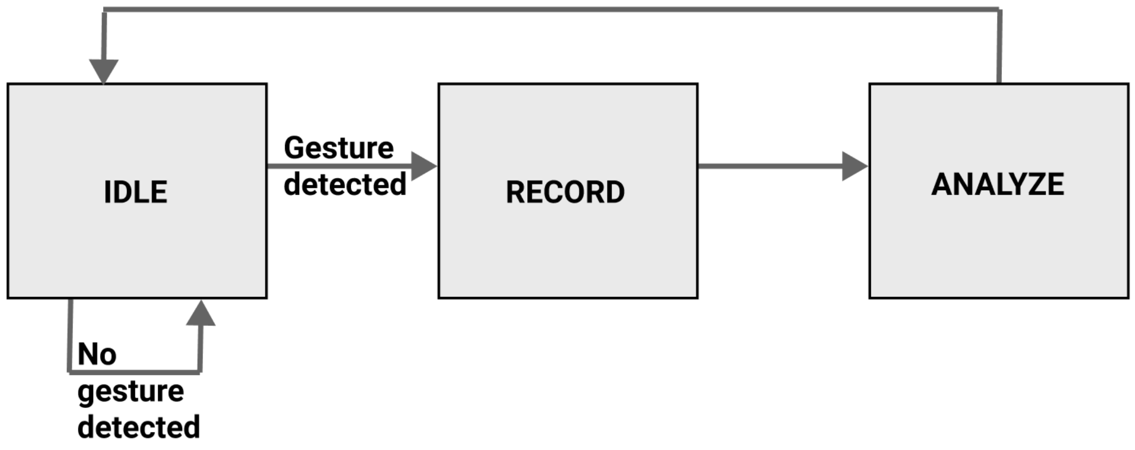
IV. Description of Modules

A. Accelerometer - Tiffany Tran

Using the resources given to us on the course website about the accelerometer, we were to get the values in the x, y, and z directions from the accelerometer on the FPGA. After the acceleration in each of the 3 directions is received, each of the accelerations is then filtered using an infinite impulse response (IIR) filter. The direction in which the FPGA is moving (positive or negative) is also stored. The filtered magnitude of the acceleration along with the sign of the direction is sent to the gestures module.

B. Gestures - Tiffany Tran

After receiving the filtered magnitudes and the sign of the directions, the data is then put through a finite state machine (FSM) to process each of the accelerations to see if and what kind of gesture was made.



The FSM first starts in the IDLE state where it will keep checking to see if the magnitude of the acceleration in the x, y, or z direction has passed a certain threshold, indicating that a gesture has been made. If none of them have passed the threshold, it will stay in IDLE. Otherwise, it will move onto the RECORD state where it will identify which gesture was made.

In the RECORD state, it will see whether the gesture was made in the x, y, or z direction as well as whether it was made in a positive or negative direction. In the x-direction, if the sign of the direction was positive, the control would be increasing the volume while if it was negative, it would decrease the volume. Along the y-axis, the positive direction would skip to the next song while the negative direction would go to the previous song. Along the z-axis, the positive direction would play the song while the negative direction would pause the song. After the gesture has been determined, it will move onto the next state which is the ANALYZE state.

In the ANALYZE state, all the controls (each of which is 1 bit long) will be put together in 1 byte, and the done signal will be set to high. These variables will be sent to the serial transmitter (serial_tx) while the state will return to IDLE until the next gesture is made.

C. UART Serial Transmitter - Tiffany Tran

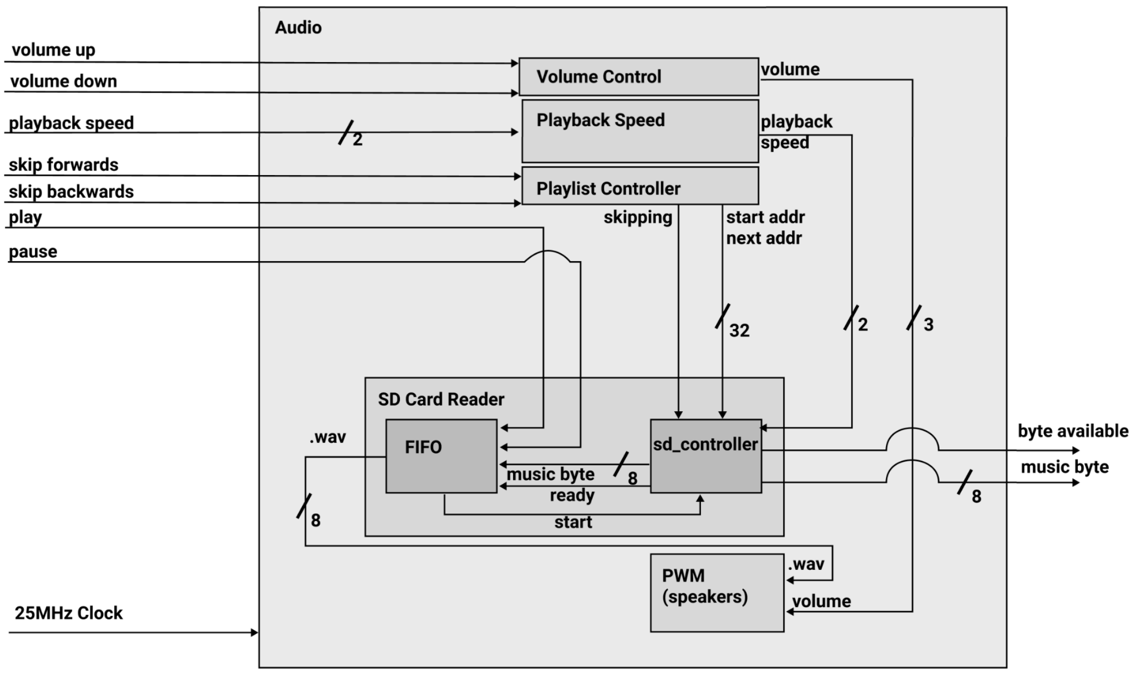
After receiving the byte from the gestures module, the data will then be sent to the transmitting Bluetooth module. Using the serial transmitter written in Lab 2, the module will send the byte bit by bit until all the bits in the byte have been sent. The data will be sent to a transmitter Bluetooth module connected to the FPGA and received by a receiver Bluetooth module connected to the different FPGA. For both the transmitter and the receiver, the divisor (the rate at which the bits were being sent and received) needed to be calculated for the 25 mHz clock we were using and for a baud rate of 38400 (baud rate needed for Bluetooth modules).

D. UART Serial Receiver - Tiffany Tran

After the receiver Bluetooth module has received the data, the receiver module will process the data to get the original byte of data. Afterwards, it will send the rearranged byte to the audio and the display module.

E. Audio - Elaine Pham

Block Diagram



The audio module performs the following functions:

- Read 8-bit unsigned mono sound music data stored in the form of a .wav file from the SD card.
- Output the 8-bit music data at a desired frequency to the speaker.
- Change the speed of the playback, volume, and track being played based on user input.

Read 8-bit unsigned mono sound music data stored in the form of a .wav file from an SD card.

In the SD Card Reader module, 8-bit unsigned mono sound music data from the SD card is read into a 8-bit-write-depth, 1024-length First In First Out (FIFO) data structure until the FIFO is full. On the start signal from the FIFO, the sd_controller module will output 512 bytes of music data from the SD card, one byte at a time, to send to the FIFO. Therefore, to start reading from the SD card, the FIFO must have at least 512 empty spaces to store the data output from the SD card, and the SD card must be ready to start a read operation. The SD card also sends out a ready signal and the data read from the SD card to the Fourier Transform Logic module in order to compute the Fourier Transform of the music data and display the Fourier Transform graphics in real time.

Output the 8-bit mono music data at a desired frequency to the speaker.

When the Audio module receives a play signal, the data from the FIFO is outputted at a rate of 48kHz, 24kHz or 96kHz depending on user input. This can be done by waiting a specific number of clock cycles corresponding to the desired frequency before reading a new byte of data from the FIFO. Once the wait period is over, a new byte of music data is read from the FIFO and is sent to the PWM module, which plays the audio through the speaker. The PWM module outputs sound by keeping track of a 8-bit counter (to output 8-bit music). The PWM module outputs a high signal to the speaker when the counter is smaller than the byte of music data and a low signal otherwise. This will create a square wave tone at the desired frequency in the music.

Change the speed of the playback, volume, and track being played based on user input.

The Audio module has three modules with finite state machines (FSM) to keep track of the current volume, playback speed, and track being played.

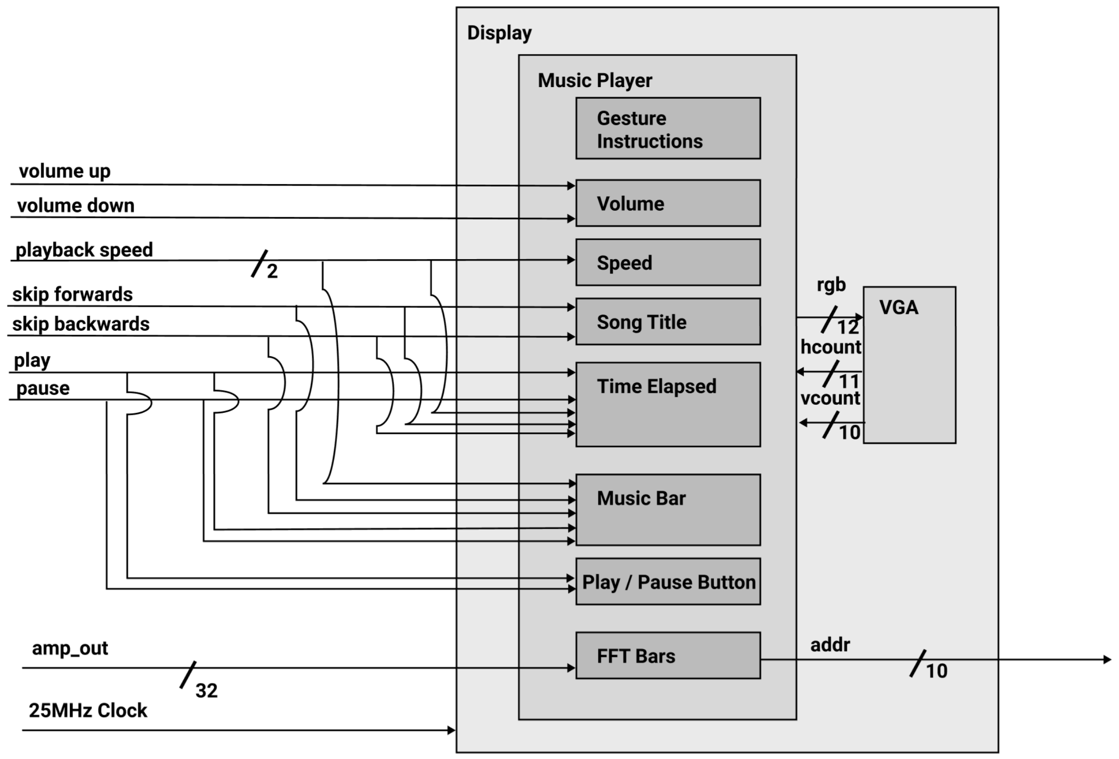
When the Audio module receives a volume down signal, the Volume Control module will switch from its current state to the state immediately before it, which will perform an unsigned right shift on the byte of music data to attenuate it. An unsigned left shift is performed on the music data for a volume up.

The Playback Speed will change the number of clock cycles the FIFO waits to output new data in order to speed up or slow down the music. The fewer clock cycles waited, the faster the music outputted and vice versa.

The Playlist Controller keeps track of the starting address in the SD card for each song. When the Playlist Controller receives a skip forward signal, the module will switch from its current state to the state immediately after it, which contains the starting address of the next two songs. After the Playlist Controller switches state, the Playlist Controller will notify the SD Card Reader module that a skip command has been received and pass the new addresses to the SD Card Reader Module using the skipping signal. This signal will prompt the SD Card Module to stop outputting data, clear the FIFO, and start reading in music data from the new address. The FSM for the Playlist Controller will switch from its current state to the state immediately before it in the case of a skip backward signal.

F. Display - Elaine Pham

Block Diagram



The Display module performs the following functions:

- create a 640x480 pixel VGA display
- draw pictures, words, and numbers on screen
- update the buttons, volume, speed, and music bar displayed based on user input
- draw the progress bar and show how much time has elapsed in the song the user is listening to
- draw a colorful visual representation of the Fourier Transform (FT) of the music in real time

Create a 640x480 pixel VGA display

Using the XVGA module provided from the Ping Pong Lab, we can create a 640x480 pixel screen by slowing the clock frequency from 100MHz to 25MHz, changing the display width and height from 1024 and 768 to 640 and 480, the horizontal and vertical front porch from 24 and 3 to 16 and 11, the horizontal and vertical sync from 136 and 6 to 96 and 2, and the horizontal and vertical back porch from 160 and 29 to 48 and 31. Because the specs are changed, the module is renamed to VGA.

Draw pictures, words, and numbers on screen

To draw a PNG image on the screen, we generate two .coe files, one to encode information about which pixels are used to draw the image and one to determine which color each pixel should be. We would then create a ROM for each .coe file. We pass in the vcount and hcount returned by the VGA module into the image ROM and pass the output of the image ROM into the colormap ROM. When the vcount and hcount are within the desired x and y locations, we use the output of the colormap ROM to determine what values to set the 4-bit pins corresponding to red, green, and blue pixels in the VGA connector.

Numbers are needed to display the volume level, speed, and time elapsed on the screen. To accomplish this, ten pre‑drawn 16x16 pixel images of the digits 0 through 9 are used to generate .coe files which are then used to generate a Read Only Memory (ROM). To display a specific digit, #, at a location on the screen, the volume, speed, and time elapsed module must wait until the hcount and vcount returned from the VGA module equal the desired x and y locations on the screen. When this condition is satisfied, the modules multiplies the digit, #, by 16x16 or 256 and passes that as the address to start reading from in the ROM module.

Similarly, words and characters could be displayed to the screen in the same manner. We did not use this method to display words in our demo because the .coe files available had too big of a font and we did not want to go through the trouble of recreating the files with a specific font and font size. We did not need to display a lot of text on the screen and thought it wasn't worth the hassle. So we displayed the remaining text on the screen (the instructions, song title, and the labels for the speed and volume) by saving PNG images of the text, generating .coes files from the images, and creating ROMs for each image displayed on the screen.

Update the buttons, volume, speed, and music bar displayed based on user input

Inside the Play / Pause Button module, there is a simple FSM which keeps track of what button is displayed currently. When the FPGA is programmed, the FSM starts by displaying the play button. When the module receives a play signal from the user, the pause button is displayed. When the module receives a pause signal from the user, the play button is displayed.

The volume and speed module are similar. The speed FSM encodes 3 states to display the digits and characters 1.0x for normal audio playback speed, 0.5x for half the audio playback speed, and 2.0x for twice the audio playback speed. The speed FSM would change depending on the playback speed signal received from the user. The volume FSM encodes 4 states to display

"100%" for the loudest volume, "75%", "50%", and "25%" for softer volumes, and 0% for no volume. Similarly, the volume FSM would change depending on the volume up or volume down signal received from the user. Note: we did not finish the volume display in time, so it did not appear in the demo.

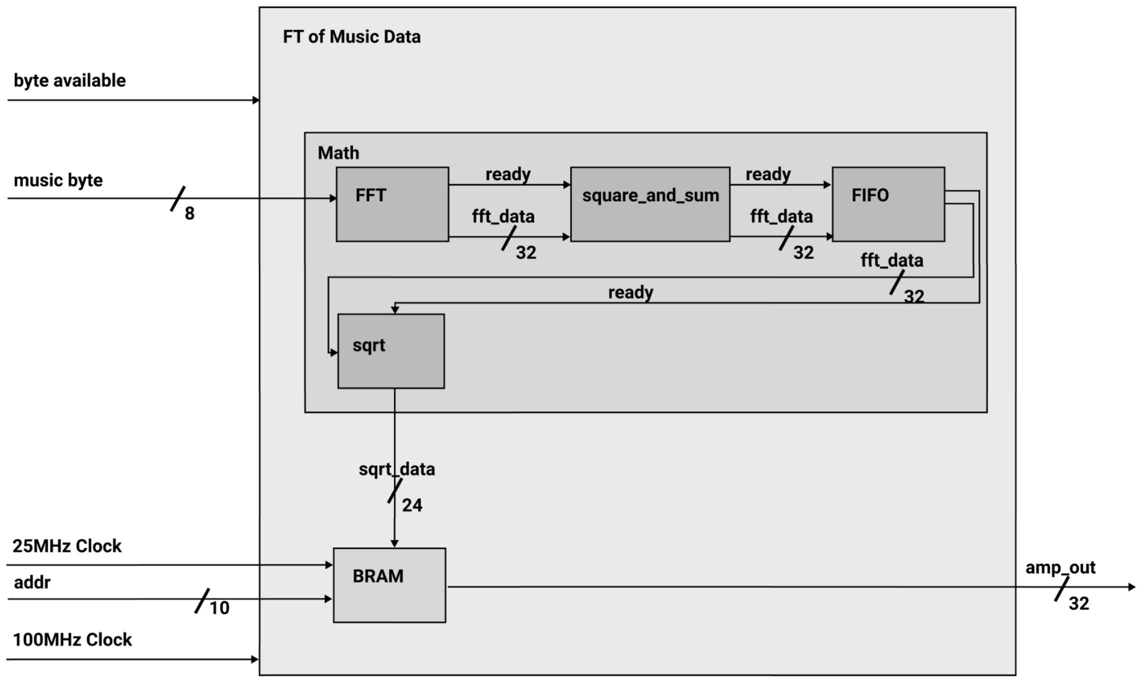
Draw the progress bar and show how much time has elapsed in the song the user is listening to

The progress bar is exactly 512 pixels in length. This length being a power of 2 is important as it enables us to perform division by simply bit shifting to the right. The clock frequency for this module is 25MHz. If we take the clock frequency and divide by the length of the music bar, the constant, k, that is returned is the number of clock cycles that the module must wait in order to color in a new pixel horizontally in the music bar. When the music bar is completely colored in, 25MHz or 1 second will have passed. Thus, if we take k and multiply that by the length of the song in seconds, we will be able to keep track of how much of the music the user has listened to and display that in the form of a white and grey music progress bar.

The logic behind updating how much time has passed in the song is similar. For the rightmost digit, the counter simply needs to wait for 25,000,000 clock cycles to increment. The number would increment from 0 to 9 and cycle back to 0. For the middle digit, the counter would wait until the rightmost digit transitions from a 9 to a 0 to increment. The number would increment from 0 to 5 and cycle back to 0. Similarly, the leftmost digit would wait until the middle digit transitions from a 5 to a 0 to increment. The number would only increment in this case from 0 to a maximum of 9 as all of our songs stored on the SD card are under 10 minutes.

Draw a colorful visual representation of the Fourier Transform (FT) of the music in real time

Block Diagram



The module to compute the Fourier Transform (FT) of the music data is provided on the course website. A simplified version of the FT logic is displayed above in the block diagram. The main functionalities are the following:

- on a high byte available signal, compute the FT for 1024 frequency bins of the music data passed in from the Audio module.
- take the magnitude of data returned from the FFT module by squaring the real and imaginary parts of the coefficients for each frequency bin, summing them, and then taking the square root.
- store the magnitude in a dual port Block Random Access Memory (BRAM) of size 1024 and write-depth 32-bits.
- return the magnitude in the BRAM at a specified address passed in from the Display module to create the FT graphics. The Display module runs on a 25MHz clock, hence, we pass in a 25MHz clock to the BRAM to ensure the data from the BRAM is returned in sync with the Display module. This avoids explicitly crossing clock domains.

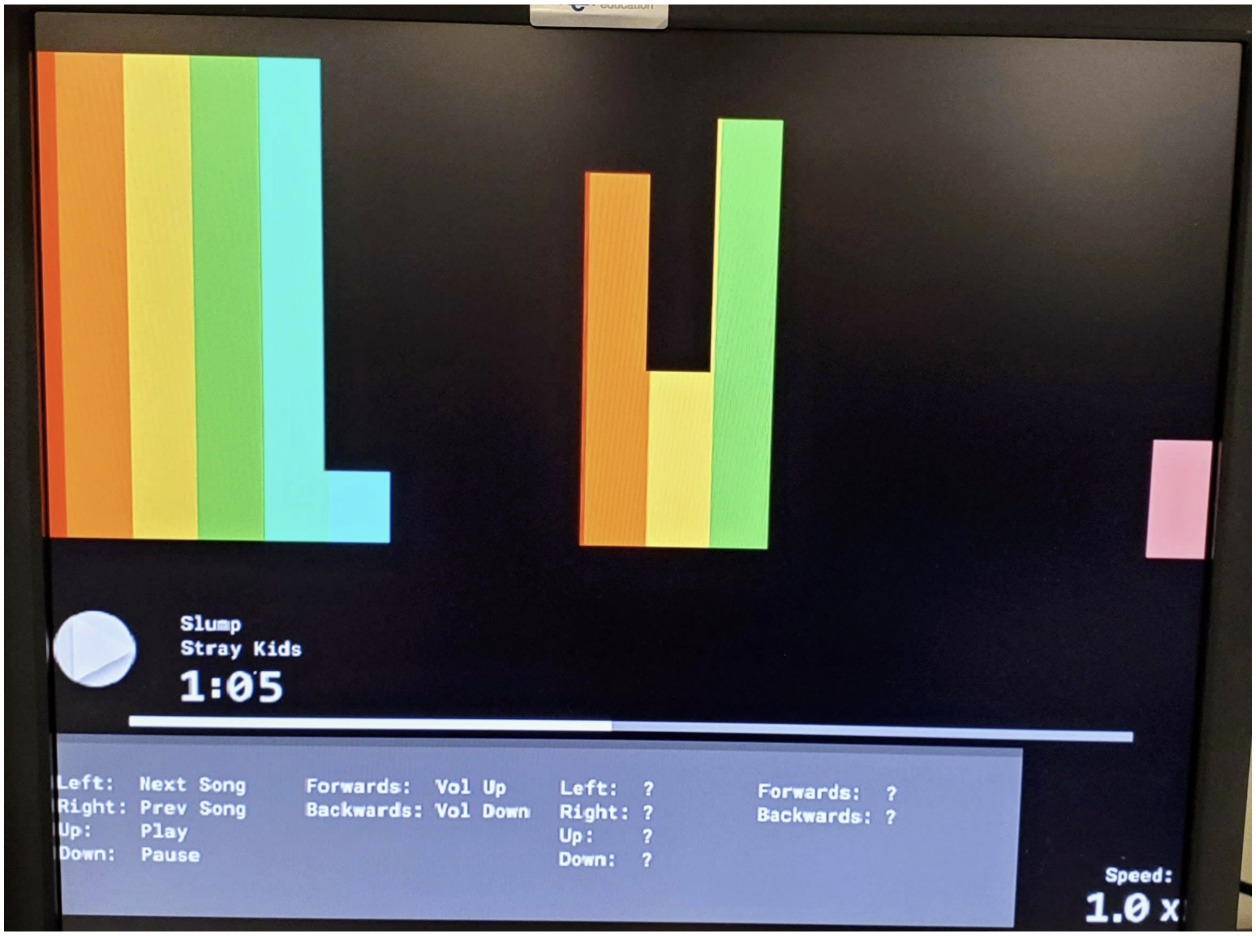
Display the Fourier Transform Graphics

When the hcount and vcount returned from the VGA module is in the top half of the screen, the FT graphic is drawn. The hcount determines what address is passed into the BRAM. The code provided on the course website originally uses a 1024x768 pixel display. When the hcount increments, the new hcount is passed into the BRAM from the Display module and a new magnitude is read out. This magnitude is passed back to the Display module and scaled to create 1 pixel wide vertical bars. The magnitude is scaled such that the maximum magnitude corresponds to the tallest colored bar. Everything below the scaled magnitude is colored in and everything above it is black. Each horizontal pixel on the screen corresponds to the different frequency and the vertical pixel determines how big the amplitude or how loud the frequency is in the byte of music read in from the SD card.

We observed that the music we were using had mostly lower frequencies out of the 1024 frequency bins and decided to crop out the higher frequency bars from the screen. We do this by dividing hcount by 32 (by bit shifting to the right) and passing that into the BRAM to only display the lowest 32 frequency amplitudes.

To make the bars colorful, I used a case statement that checked for whether hcount was in between a certain range and if so, set the RGB bits to a certain color.

Graphics Demo



IV. Testing

Read 8-bit unsigned mono sound music data stored in the form of a .wav file from an SD card.

To debug reading from the SD card, I needed to set up an Integrated Logic Analyzer (ILA).

Output the 8-bit mono music data at a desired frequency to the speaker.

Once I was confident that I was reading from the SD card correctly, I made a fake SD module that mimicked the SD Controller module provided by the course site. In the fake SD module, I programmed the byte output to be the numbers 0 to 511 repeatedly. This helped me figure out whether I was skipping bytes while reading in and outputting data from the FIFO. I also saved time by not having to manually write data to my SD card and then write back my song data after I finished debugging.

Finite State Machines (FSMs) in all modules

I wrote test benches that stepped through all possible state transitions to test all of the modules that required FSMs.

V. Challenges

A. Tiffany Tran

Some of the biggest challenges were trying to get the bluetooth modules connected, as well as working properly, and combining my code with Elaine's code.

For the bluetooth modules, I was having trouble pairing them properly and . I did not know this until later, but not only did both devices have to be on at the same time when binding, but the LED's on the modules would blink twice every two seconds when paired properly. I also had to set the board to Arduino Leonardo instead of Arduino Micro when coding in Arduino. Quite a bit of research needed to be done on the HC-05 Bluetooth module as there were many websites about how to pair them, but there were very few websites that were helpful in actually pairing the devices. Some required code to pair them, which was not necessary (except for setting Serial to a baud rate of 9600 and Serial1 to a baud rate of 38400. I am unsure if the Serial can be set higher as when we tried to set it to 38400, the Arduino Micro's we were using broke.)

The other challenge was combining the code I wrote for the receiver Bluetooth module with Elaine's code. Since we had coded separately, Elaine had been using other variables (such as the switches, buttons, and LED's) to make sure her parts were working properly. We had to go through each of her modules where she had to change the inputs to the controls that were sent across the Bluetooth. The debugging took longer than expected since we not only had to check her code, but my code as well to see if there was something wrong with how I sent the data.

B. Elaine Pham

Read 8-bit unsigned mono sound music data stored in the form of a .wav file from an SD card.

To debug reading from the SD card, I needed to set up an Integrated Logic Analyzer (ILA). However, my ILA encountered problems such as not triggering correctly. The instructors and I did not figure out why my ILA was malfunctioning so I had to work with a janky ILA to debug the SD card. My workaround for the ILA was just to keep hitting the play button until a new waveform appeared. My main problem with reading data from the SD card and outputting from the FIFO was finding the right condition for when to start reading from the SD card.

Output the 8-bit mono music data at a desired frequency to the speaker.

The quality of the 8-bit music data was really bad at first and it took a while for me to figure out that the volume module was doing a signed shift on unsigned music data. Therefore, the music outputted through the speakers was extra staticy. This took a while to debug as well. I first tried to pass the music data through a filter, but the music was still rather noisy. I then tried to upgrade from 8-bit to 16-bit music data, but found out that it was not possible with a 25MHz clock.

Create a 640x480 pixel VGA display

The clock frequency needed for a 640x480 pixel display cannot be exactly 25.175MHz as described in the XVGA module. Therefore, the screen occasionally encountered problems with alignment. The screen sometimes would shift to the left or right or appear zoomed in. Our workaround for this problem was to repeatedly hit the alignment button on the computer monitor or reset the display by reprogramming the FPGA until the screen looked correct.

Draw pictures, words, and numbers on screen

Debugging why an image was not outputting correctly was also difficult in that I could not easily write a test bench and simulate why the pixel output was wrong. I debugged mostly by looking at the display and rereading my code, which at times was very time consuming for relatively small tasks such as displaying a number on the screen.

Draw the progress bar and show how much time has elapsed in the song the user is listening to

This part of the project was also tricky in that it required a clever solution. The most straightforward way of making a music progress bar was to take the current time elapsed in the song, divide by the length of the song, and multiply by the length of the bar in pixels to figure out how much of the progress bar was filled in at a time. However, division is hard to do in hardware. I first tried to make a division module to compute the division in several clock cycles. I asked Gim about a more efficient solution that didn't require extra clock cycles and he told me about the method described above in the display module.

VI. Conclusion/Reflection

We learned a lot from this project, such as using the accelerometer to convert user hand movements into digital signals, bluetooth to send data, SD card to store music data, graphics, FIFOs, BRAMs, and a lot of math modules from the IP Wizard to handle the complex Fourier Transform calculations efficiently and pass data to the right modules at the right time.

In the future, we could work on improving the accuracy of the gestures detected from the user. To accomplish this, we could insert a time buffer in between when new signals can be processed so that multiple signals are not sent in a short period of time accidentally when the user makes a movement. We can also work on improving the graphics for the FT by slowing down how often the FT bars update to avoid the flashing effect from our demo.

We can also add additional cool effects to our program such as implementing a filter for the bass and treble frequencies of the song and allowing the user to control how loud or soft these frequencies are in the song using their hand.

Overall, we think the resources provided on the course website were very helpful. While we ran into some bumps due to some things not working as they should and took longer to debug than we thought, such as the bluetooth modules and reading music stored on the SD card, we were able to figure out the issues and learn from them.

VII. Appendix

A. References

- Python script to generate .coe files from PNG or JPEG
- 16x16 Pixels Digits ROM
- 8x12 Pixels Alpha Characters ROM
- SD Starter Code
- FFT Starter Code
- Pairing Bluetooth Modules
- Transmitting and Receiving Serial Data

B. Code

// accelerometer.sv

```
`timescale 1ns / 1ps

module accelerometer(
   input clk_100mhz,
   input[15:0] sw,
   input btnc,btnr,
   input jb_in, // jb[0]
   output logic [15:0] led,
   output acl_sclk,
   output acl_mosi,
   input acl_miso,
   output acl_csn,
   output logic jb_out // jb[1]
   );

   logic clk_25mhz;
   clk_wiz_0 clocks(.clk_in1(clk_100mhz), .clk_out1(clk_25mhz));

   logic [31:0] data;


// ---------------------------------------------------------------------------------------
// Process data from the ADXL
//
    logic [11:0] accel_x, accel_y, accel_z, accel_temp;
    logic [11:0] accel_x_latch, accel_y_latch, accel_z_latch;
    logic [11:0] x_mag, y_mag, z_mag;
    logic [3:0] x_sign, y_sign, z_sign;

    logic [11:0] x_mag_filtered, y_mag_filtered, z_mag_filtered;

    logic data_ready;

    always_ff @(posedge clk_25mhz) begin
        accel_x_latch <= data_ready ? accel_x : accel_x_latch;
        accel_y_latch <= data_ready ? accel_y : accel_y_latch;
        accel_z_latch <= data_ready ? accel_z : accel_z_latch;
    end

   // this section displays twos complement in + - format
   assign x_mag = accel_x_latch[11] ? (~accel_x_latch +1)  : accel_x_latch;
   assign y_mag = accel_y_latch[11] ? (~accel_y_latch +1)  : accel_y_latch;
   assign z_mag = accel_z_latch[11] ? (~accel_z_latch +1)  : accel_z_latch;

   assign x_sign = accel_x_latch[11] ? 4'hF : 4'h0;
   assign y_sign = accel_y_latch[11] ? 4'hF : 4'h0;

 // average  samples
 iir_filter x_filter(.clk(clk_25mhz), .data_ready(data_ready), .data_in(x_mag),
.data_filtered(x_mag_filtered));
 iir_filter y_filter(.clk(clk_25mhz), .data_ready(data_ready), .data_in(y_mag),
.data_filtered(y_mag_filtered));
```

```
 iir_filter z_filter(.clk(clk_25mhz), .data_ready(data_ready), .data_in(z_mag),
.data_filtered(z_mag_filtered));


   assign data = {x_sign, x_mag_filtered, y_sign, y_mag_filtered}; // Display the x and y values

// acl_miso  acl_mosi  acl_sclk  acl_csn

// btnc button is user reset
    logic reset;
    debounce db1(.reset_in(0),.clock_in(clk_25mhz),.noisy_in(btnc),.clean_out(reset));

    logic switch;
    logic x_light, y_light, z_light, send_light;

    assign led[0] = x_light;
    assign led[1] = y_light;
    assign led[2] = z_light;
    assign led[3] = send_light;

    assign switch = sw[0];

    logic play;
    logic pause;
    logic next_song; // skip forwards
    logic prev_song; // skip backwards
    logic vol_up;
    logic vol_down;

    logic [7:0] val_in;
    logic done;

    // Send the x, y, and z values to gestures to get the controls

    gestures my_gest(.clk_25mhz(clk_25mhz), .rst_in(btnc), .x_mag_filtered(x_mag_filtered),
                 .y_mag_filtered(y_mag_filtered), .z_mag_filtered(z_mag_filtered),
                 .x_sign(x_sign), .y_sign(y_sign), .z_sign(z_sign),
                 .x_light(x_light), .y_light(y_light), .z_light(z_light),
.send_light(send_light),
                 .switch(switch),
                 .val_in(val_in), .done(done));

    serial_tx my_tx (.clk_in(clk_25mhz), .rst_in(btnc), .trigger_in(done), //btnr&~old_button),
                     .val_in(val_in), .data_out(jb_out));

ADXL362Ctrl   my_accel(
    .SYSCLK     (clk_25mhz),   // in
    .RESET      (reset),        // in

    .ACCEL_X    (accel_x),      // out STD_LOGIC_VECTOR (11 downto 0);
    .ACCEL_Y    (accel_y),      // out STD_LOGIC_VECTOR (11 downto 0);
    .ACCEL_Z    (accel_z),      // out STD_LOGIC_VECTOR (11 downto 0);
    .ACCEL_TMP  (accel_temp),   // out STD_LOGIC_VECTOR (11 downto 0);
    .Data_Ready (data_ready),   // out STD_LOGIC;

    .SCLK       (acl_sclk),     // out
    .MOSI       (acl_mosi),     // out
    .MISO       (acl_miso),     // in
    .SS         (acl_csn)       // out STD_LOGIC
    );

endmodule


module iir_filter(
    input clk, data_ready,
    input[11:0] data_in,
    output [11:0] data_filtered
    );

    logic old_ready;
    logic [14:0] data_average;

    assign data_filtered = data_average[12:1]; // scale it

    always_ff @(posedge clk) begin
        old_ready <= data_ready;
        data_average <= !(old_ready && data_ready) ?
          ((data_in >> 3) + ((7*data_average)>>3)) : data_average;
        end


endmodule
```

// gestures.sv

```
`timescale 1ns / 1ps

module gestures(
    input clk_25mhz,
    input rst_in,
    input switch,
    input [11:0] x_mag_filtered, y_mag_filtered, z_mag_filtered,
    input x_sign, y_sign, z_sign,
    output logic x_light, y_light, z_light, send_light,
    output logic [7:0] val_in,
    output logic done
    );

    parameter X = 2'b00;
    parameter Y = 2'b01;
    parameter Z = 2'b10;

    parameter IDLE = 2'b00;
    parameter RECORD = 2'b01;
    parameter ANALYZE = 2'b10;

    logic [1:0] axis;
    logic [1:0] state;

    logic play;
    logic pause;
    logic next_song; // skip forwards
    logic prev_song; // skip backwards
    logic vol_up;
    logic vol_down;
    logic zsign;

    always_ff @ (posedge clk_25mhz) begin
        if (rst_in) begin
            state <= IDLE;
        end else begin
            case(state)
                // Wait until motion is detected
                IDLE:   begin
                            x_light <= 1'b0;
                            y_light <= 1'b0;
                            z_light <= 1'b0;

                            val_in <= 8'b0;
                            done <= 1'b0;

                            vol_up <= 1'b0;
                            vol_down <= 1'b0;

                            // Reset all controls to 0
                            play <= 1'b0;
                            pause <= 1'b0;
                            next_song <= 1'b0;
                            prev_song <= 1'b0;

                            if (x_mag_filtered >= 12'hA0) begin
                                state <= RECORD;
                                axis <= X;
                            end

                            if (y_mag_filtered >= 12'hA0) begin
                                state <= RECORD;
                                axis <= Y;
                            end

                            if (z_mag_filtered >= 12'h245) begin
                                state <= RECORD;
                                axis <= Z;
                                zsign <= ~zsign;
                            end
                        end
                // Get the gesture
                RECORD: begin
                            send_light <= 1'b0;

                            // Check the axis, then the sign to assign the gesture
                            // first set of gestures
                            if (axis == X & (x_mag_filtered < 12'hA0)) begin
                                if (x_sign == 4'h0) begin
                                    vol_up <= 1'b1;
                                end else begin
                                    vol_down <= 1'b1;
                                end
                                x_light <= 1'b1;
                                state <= ANALYZE;
                            end

                            if (axis == Y & (y_mag_filtered < 12'hA0)) begin
                                if (y_sign == 4'h0) begin
                                    next_song <= 1'b1;
                                end else begin
```

```
                                        prev_song <= 1'b1;
                                    end
                                    y_light <= 1'b1;
                                    state <= ANALYZE;
                                end

                                if (axis == Z & (z_mag_filtered < 12'h245)) begin
                                    if (zsign) begin
                                        play <= 1'b1;
                                    end else begin
                                        pause <= 1'b1;
                                    end
                                    z_light <= 1'b1;
                                    state <= ANALYZE;
                                end

                            end

                // Send the data to bluetooth modules
                ANALYZE:begin
                                val_in <=
{1'b0,1'b0,play,pause,next_song,prev_song,vol_up,vol_down};
                                // X: volume
                                // Y: prev/next song
                                // Z: play/pause

                                done <= 1'b1;

                                state <= IDLE;
                            end
                default:begin
                                state <= IDLE;
                                x_light <= 1'b0;
                            end
            endcase
        end
    end

endmodule
```

// serial.tx

```
`timescale 1ns / 1ps

`default_nettype none

module serial_tx(   input wire      clk_in,
                    input wire      rst_in,
                    input wire      trigger_in,
                    input wire [7:0]     val_in,
                    output logic    data_out);
    parameter   DIVISOR = 651; //2604; //treat this like a constant!!
                               // baud rate of bluetooth is 9600 (Should be 38400, but didn't
work)

    logic [9:0]         shift_buffer; //10 bits...interesting
    logic [31:0]        count;


    always_ff @(posedge clk_in)begin
```

```
        if (rst_in) begin
            count <= 32'b0;
        end else begin
            if (trigger_in) begin
                shift_buffer <= {1'b1, val_in, 1'b0};
                count <= count +1;
            end
            if (count === DIVISOR) begin
                data_out <= shift_buffer[0];
                shift_buffer <= shift_buffer >> 1;
                shift_buffer[9] <= 1;
                count <= 32'b0;
            end else begin
                count <= count + 1;
            end
        end
    end
endmodule
`default_nettype wire
```

// uart_receiver.sv

```
`timescale 1ns / 1ps

module uart_receiver (input clk,
                      input val_input,
                      output [7:0] byte_out,
                      output clean,
                      // elaine's edit
                      output byte_available_out);

    parameter [15:0] divisor = 651;

    parameter IDLE = 3'b000;
    parameter START = 3'b001;
    parameter DATA = 3'b010;
    parameter STOP = 3'b011;

    logic [2:0] state;
    logic [15:0] cycle_counter;

    logic [7:0] byteout;
    logic [3:0] bit_index;
    logic starting;

    assign clean = starting;
    assign byte_out = byteout;

    //this module uses an fsm that gets triggered when it
    //reads a start bit . it then waits divisor , cycles per bit ,
    //and samples the input to get the following bits.
    //Once it starts reading, it also sends a high signal
    //for one clock cycle to signal to other modules that it is reading.

    // elaine's edit
    logic old_starting;
    // byte_available = HI only when starting first becomes HI (ie after finished reading new
byte)
    assign byte_available_out = (starting && (~old_starting))? 1: 0 ;
    always @(posedge clk) begin
        starting <= 0;
        case(state)
            IDLE: if (val_input == 0) begin
                    starting <= 0;
                    state <= START;
                    cycle_counter <= 0;
                  end
            START: begin
                    starting <= 0;
                    if (cycle_counter == divisor / 2) begin
                        cycle_counter <= 0;
                        state <= DATA;
                    end

                    else if (val_input == 0) cycle_counter <= cycle_counter + 1;

                    else begin
                        state <= IDLE;
                        cycle_counter <= 0;
                    end
                   end

            DATA: begin
                starting <= 0;
                if (cycle_counter == divisor) begin
                    cycle_counter <= 0;
                    if (bit_index <= 7) begin
                        byteout[bit_index] <= val_input;
                        bit_index <= bit_index + 1;
                    end

                    else begin
                        state <= STOP;
                        bit_index <= 0; end
                    end

                    else cycle_counter <= cycle_counter + 1;
                end

            STOP: begin
                    starting <= 0;
                    if (cycle_counter == divisor/4) begin
                        cycle_counter <= 0;
                        state <= IDLE;
                        starting <= 1;
                    end else cycle_counter <= cycle_counter + 1;
                  end
        endcase
        // elaine's edit
        old_starting <= starting;
    end
endmodule
```

// top level for controlling audio and display on Elaine's side

```
`timescale 1ns / 1ps
`default_nettype none


module top_level(
        input wire clk_100mhz,
        input wire btnd, // system reset
        // TODO: replace with gestures!
        input wire[15:0] sw, // controls volume

        // sd card inputs
        input wire sd_cd,
        // TODO: replace these with gestures!
        input wire btnc, // play
        input wire btnu, // pause
        input wire btnl, // skip backwards
        input wire btnr, // skip forwards
        inout wire [3:0] sd_dat,

        // bluetooth receiver input
        input wire jb_in,

        // bluetooth receiver output
        output logic jb_out,

        // sd card outputs
        output logic sd_reset,
        output logic sd_sck,
        output logic sd_cmd,

        // audio outputs
        output logic aud_pwm,
        output logic aud_sd,

        // vga outputs
        output logic[3:0] vga_r,
        output logic[3:0] vga_b,
        output logic[3:0] vga_g,
        output logic vga_hs,
        output logic vga_vs,

        // not being used
        output logic [15:0] led,

        // hex display outputs for debugging
        output logic ca, cb, cc, cd, ce, cf, cg,
        output logic [7:0] an
    );

    // SD, VGA need to use 25 mhz clock
    logic clk_25Mhz; //65 MHz clock!
    clk_wiz_0 clkdivider(.clk_in1(clk_100mhz), .clk_out1(clk_25Mhz));

    // GLOBAL VARIABLES
    logic sys_reset_in;
    assign sys_reset_in = btnd;
//    logic [6:0] cat;
//    assign {cg,cf,ce,cd,cc,cb,ca} = cat;


//    // play/pause, volume up/down, skip forwards/backwards are all 1 clk cycle pulse
//    // TODO: replace buttons with signals coming from accelerometer
//    logic play_in, pause_in, volume_up, volume_down, skip_forward_in, skip_backward_in;
//    debounce d1( .reset_in(sys_reset_in), .clock_in(clk_25Mhz),
//                .noisy_in(btnc), .clean_out(play_in));
//    debounce d2( .reset_in(sys_reset_in), .clock_in(clk_25Mhz),
//                .noisy_in(btnu), .clean_out(pause_in));
//    debounce d3( .reset_in(sys_reset_in), .clock_in(clk_25Mhz),
//                .noisy_in(btnr), .clean_out(skip_forward_in));
//    debounce d4( .reset_in(sys_reset_in), .clock_in(clk_25Mhz),
//                .noisy_in(btnl), .clean_out(skip_backward_in));


//    audio my_audio (
//        .clk_25Mhz_in(clk_25Mhz),
//        .sys_reset_in(sys_reset_in),
//        .sw_in(sw),

//        .sd_cd_in(sd_cd),
//        .play_in(play_in),
//        .pause_in(pause_in),
//        .skip_forward_in(skip_forward_in),
//        .skip_backward_in(skip_backward_in),
//        .sd_dat_in(sd_dat),

//        .sd_reset_out(sd_reset),
//        .sd_sck_out(sd_sck),
//        .sd_cmd_out(sd_cmd),

//        .aud_pwm_out(aud_pwm),
//        .aud_sd_out(aud_sd),

//        .cat_out(cat),
//        .an_out(an));
    logic byte_available;
    logic [7:0] fft_data;
    audio my_audio (
        .clk_25Mhz(clk_25Mhz),
        .btnd(sys_reset_in),
        .sw(sw),

        .sd_cd(sd_cd),
        .sd_dat(sd_dat),
        .btnc(play_in),
        .btnu(pause_in),
        .btnl(prev_song_in),
        .btnr(next_song_in),
        .vol_up(vol_up_in),
        .vol_down(vol_down_in),

        .sd_reset(sd_reset),
        .sd_sck(sd_sck),
        .sd_cmd(sd_cmd),

        .byte_available_out(byte_available),
        .fifo_data(fft_data),

        .aud_pwm(aud_pwm),
        .aud_sd(aud_sd),

        //.ca(ca), .cb(cb), .cc(cc), .cd(cd), .ce(ce), .cf(cf), .cg(cg),
        .an(an));


//    display my_display(.clk_25Mhz(clk_25Mhz),
//            .sys_reset_in(sys_reset_in),
////            .amp_in(),
//            .sw_in(sw),
//            .play_in(play_in),
//            .pause_in(pause_in),
//            .skip_forward_in(skip_forward_in),
//            .skip_backward_in(skip_backward_in),
////            .draw_addr_out(),
//            .vga_r_out(vga_r),
//            .vga_b_out(vga_b),
//            .vga_g_out(vga_g),
//            .vga_hs_out(vga_hs),
//            .vga_vs_out(vga_vs));

        logic [31:0] amp;
        logic [9:0] draw_addr;
        display my_display(
            .clk_25Mhz(clk_25Mhz),
            .sw(sw),
            .btnc(play_in),
            .btnu(pause_in),
            .btnl(prev_song_in),
            .btnr(next_song_in),
            .btnd(sys_reset_in),
            .vol_up(play_in),
            .vol_down(pause_in),

            .amp_in(amp),
            .draw_addr_out(draw_addr),

            .vga_r(vga_r),
            .vga_b(vga_b),
            .vga_g(vga_g),
            .vga_hs(vga_hs),
            .vga_vs(vga_vs));

        fft my_fft(
            .clk_100mhz(clk_100mhz),
            .byte_available_in(byte_available),
            .byte_in(fft_data),
            .draw_addr_in(draw_addr),
            .amp_out(amp));

        // debouned buttons to control music player and graphics
        logic btn_play, btn_pause, btn_volume_up, btn_volume_down, btn_skip_forward,
btn_skip_backward;
        debounce d1( .reset_in(sys_reset_in), .clock_in(clk_25Mhz),
            .noisy_in(btnc), .clean_out(btn_play));
        debounce d2( .reset_in(sys_reset_in), .clock_in(clk_25Mhz),
                    .noisy_in(btnu), .clean_out(btn_pause));
        debounce d3( .reset_in(sys_reset_in), .clock_in(clk_25Mhz),
                    .noisy_in(btnr), .clean_out(btn_skip_forward));
        debounce d4( .reset_in(sys_reset_in), .clock_in(clk_25Mhz),
                    .noisy_in(btnl), .clean_out(btn_skip_backward));

        // val_in <= {2'b0,play,pause,next_song,prev_song,vol_up,vol_down};
        logic uart_available, uart_clean;
        logic [7:0] byte_out;
        logic [31:0] num_to_display;
        uart_receiver my_urt(.clk(clk_25Mhz),
                       .val_input(jb_in),
                       .byte_out(byte_out),
                       .clean(uart_clean),
                       .byte_available_out(uart_available));
        logic play_in, pause_in, next_song_in, prev_song_in, vol_up_in, vol_down_in;
        assign num_to_display = {4'b1111,
                                {3'b000, byte_out[5]},
                                {3'b000, byte_out[4]},
                                {3'b000, byte_out[3]},
                                {3'b000, byte_out[2]},
                                {3'b000, byte_out[1]},
                                {3'b000, byte_out[0]}};


        parameter use_gestures = 1; // change to 1 if using gestures, otherwise use buttons to
control music player
        always_comb begin
            if (use_gestures) begin
                // use gestures
                if (uart_available) begin // wait for new byte received from bluetooth
                    // uart_available only HI for one clk cycle
                    vol_down_in = byte_out[1];
                    vol_up_in = byte_out[0];
                    prev_song_in = byte_out[3];//byte_out[2];
                    next_song_in = byte_out[2]; //byte_out[3];
                    pause_in = byte_out[4];
                    play_in = byte_out[5];
                end else begin
                    // ensures inputs return to LO after going HI = 1 clk cycle pulses
                    vol_down_in = 0;
                    vol_up_in = 0;
                    prev_song_in = 0;
                    next_song_in = 0;
                    pause_in = 0;
                    play_in = 0;
                end
            end else begin
                // use buttons for play_in pause_in next_song_in, prev_song_in
                // no buttons for vol_up_in, vol_down_in
                vol_down_in = 0;
                vol_up_in = 0;
                prev_song_in = btn_skip_backward;
                next_song_in = btn_skip_forward;
                pause_in = btn_pause;
                play_in = btn_play;
            end
        end

        // for debugging
        // display what is being read from the SD card and what FIFO is outputting
        parameter ONE_HZ_PERIOD = 25_000_000;          // 1 sec
        seven_seg_controller #(.ONE_HZ_PERIOD(ONE_HZ_PERIOD)) ssc
                (.clk_in(clk_25Mhz), .val_in(num_to_display),
                .rst_in(sys_reset_in),
                .cat_out({cg,cf,ce,cd,cc,cb,ca}), .an_out(an));

endmodule


// button debouncer
module debounce (input wire reset_in, clock_in, noisy_in,
                 output logic clean_out);
   logic [19:0] count;
   logic new_input;

   always_ff @(posedge clock_in)
     if (reset_in) begin
        new_input <= noisy_in;
        clean_out <= noisy_in;
        count <= 0; end
     else if (noisy_in != new_input) begin new_input<=noisy_in; count <= 0; end
     else if (count == 1000000) clean_out <= new_input;
     else count <= count+1;
endmodule


// display to the hex leds
module seven_seg_controller(input wire         clk_in,
                            input wire         rst_in,
                            input wire [31:0]  val_in,
                            output logic[6:0]   cat_out,
                            output logic[7:0]   an_out
    );

    parameter ONE_HZ_PERIOD = 25_000_000;          // 1 sec
    parameter REFRESH_RATE = ONE_HZ_PERIOD >> 10;

    logic[7:0]      segment_state;
    logic[31:0]     segment_counter;
    logic [3:0]     routed_vals;
    logic [6:0]     led_out;

    binary_to_seven_seg my_converter ( .bin_in(routed_vals), .hex_out(led_out));
    assign cat_out = ~led_out;
    assign an_out = ~segment_state;



    always_comb begin
        case(segment_state)
            8'b0000_0001:   routed_vals = val_in[3:0];
            8'b0000_0010:   routed_vals = val_in[7:4];
            8'b0000_0100:   routed_vals = val_in[11:8];
            8'b0000_1000:   routed_vals = val_in[15:12];
            8'b0001_0000:   routed_vals = val_in[19:16];
            8'b0010_0000:   routed_vals = val_in[23:20];
            8'b0100_0000:   routed_vals = val_in[27:24];
            8'b1000_0000:   routed_vals = val_in[31:28];
            default:        routed_vals = val_in[3:0];
        endcase
    end

    always_ff @(posedge clk_in)begin
        if (rst_in)begin
            segment_state <= 8'b0000_0001;
            segment_counter <= 32'b0;
        end else begin
            if (segment_counter == REFRESH_RATE)begin
                segment_counter <= 32'd0;
                segment_state <= {segment_state[6:0],segment_state[7]};
            end else begin
                segment_counter <= segment_counter +1;
            end
        end
    end

endmodule //seven_seg_controller


module binary_to_seven_seg(
        input wire [3:0]       bin_in,  //declaring input explicitely
        output logic [6:0]      hex_out);  //declaring output explicitely


    // Get the hex value from 4 bits
    logic [15:0] num;
    genvar i;
    generate
      for (i = 0; i < 16; i=i+1) begin:m
        assign num[i] = bin_in==i;
      end
    endgenerate

    // Get the corresponding leds to draw a hex num
    assign hex_out[0] = num[0] || num[2] || num[3] || num[5] || num[6] || num[7] || num[8] ||
                     num[9] || num[10] || num[12] || num[14] || num[15];

    assign hex_out[1] = num[0] || num[1] || num[2] || num[3] || num[4] || num[7] || num[8] ||
                     num[9] || num[10] || num[13];

    assign hex_out[2] = num[0] || num[1] || num[3] || num[4] || num[5] || num[6] || num[7] ||
                     num[8] || num[9] || num[10] || num[11] || num[13];

    assign hex_out[3] = num[0] || num[2] || num[3] || num[5] || num[6] || num[8] || num[9] ||
                     num[11] || num[12] || num[13] || num[14];

    assign hex_out[4] = num[0] || num[2] || num[6] || num[8] || num[10] || num[11] || num[12]
                     || num[13] || num[14] || num[15];

    assign hex_out[5] = num[0] || num[4] || num[5] || num[6] || num[8] || num[9] || num[10] ||
                     num[11] || num[12] || num[14] || num[15];

    assign hex_out[6] = num[2] || num[3] || num[4] || num[5] || num[6] || num[8] || num[9] ||
     num[10] || num[11] || num[13] || num[14] || num[15];


endmodule //binary_to_seven_seg


`default_nettype wire
```

// audio

```
`timescale 1ns / 1ps
`default_nettype none

module audio(
         input wire clk_25Mhz,
         input wire btnd, // system reset
         // TODO: replace with gestures!
         input wire[15:0] sw, // controls volume

         // sd card inputs
         input wire sd_cd,
         // TODO: replace these with gestures!
         input wire btnc, // play
         input wire btnu, // pause
         input wire btnl, // skip backwards
         input wire btnr, // skip forwards
         input wire vol_up,
         input wire vol_down,
         inout wire [3:0] sd_dat,

         // sd card outputs
         output logic sd_reset,
         output logic sd_sck,
         output logic sd_cmd,

         // my outputs (for graphics)
         output logic byte_available_out,
         output logic [7:0] fifo_data,
//        output logic [15:0] fifo_data,

         // audio outputs
         output logic aud_pwm,
         output logic aud_sd,

         // not being used
         output logic [15:0] led,

         // hex display outputs for debugging
         //output logic ca, cb, cc, cd, ce, cf, cg,
         output logic [7:0] an

);

    // GLOBAL VARIABLES
    logic sys_reset_in; // s
    assign sys_reset_in = btnd;
    // play/pause, volume up/down, skip forwards/backwards are all 1 clk cycle pulse
    // TODO: replace buttons with signals coming from accelerometer
    logic play, pause, volume_up, volume_down, skip_forward, skip_backward;
    assign play = btnc;
    assign pause = btnu;
    assign skip_forward = btnr;
    assign skip_backward = btnl;
    // TODO: you can use leds for the switches however you'd like
    assign led = sw; // turns on led above corresponding switches


    // outputs correct song addr after skipping
    logic skipping;
    logic [31:0] sd_start_addr, sd_next_addr;
    playlist_controller(
            .clk_in(clk_25Mhz), .sys_reset_in(sys_reset_in),
            .skip_forward_in(skip_forward), .skip_backward_in(skip_backward),
            .skip_out(skipping),
            .sd_start_addr_out(sd_start_addr), .sd_next_addr_out(sd_next_addr));


    // SD card reader + FIFO at 48kHz
    // logic [7:0] sd_data;
    logic [31:0] num_to_display; // for debugging
    sd_card_reader my_sd(.clk_in(clk_25Mhz), .sys_reset_in(sys_reset_in),
            .speed_in(sw[10:9]),
            .play_in(play), .pause_in(pause), .skip_in(skipping),
            .start_addr_in(sd_start_addr), .next_addr_in(sd_next_addr),
            .sd_cd_in(sd_cd), .sd_dat_in(sd_dat),
            .byte_available_out(byte_available_out),
            .sd_reset_out(sd_reset), .sd_sck_out(sd_sck), .sd_cmd_out(sd_cmd),
            .num_to_display_out(num_to_display),
            .fifo_data_out(fifo_data));


    // from lab 5
    // controlls the volume based on switches
    // TODO: change module to use gestures instead of switches
    assign aud_sd = 1;
    logic [7:0] vol_out;
//    logic [15:0] vol_out;
    // if sw[7] HIGH then use gestures as input, otherwise use switche
    volume_control vc (.clk_in(clk_25Mhz), .sys_reset_in(sys_reset_in),
                    .vol_in(sw[15:13]), .which_vol(sw[7]),
                    .vol_up_in(vol_up), .vol_down_in(vol_down),
                    .signal_in(fifo_data), .signal_out(vol_out));


    // from lab 5
    // outputs data to the speakers
    logic pwm_val; //pwm signal (HI/LO)
    pwm pwmthing(.clk_in(clk_25Mhz), .rst_in(sys_reset_in),
                .level_in(vol_out[7:0]), .pwm_out(pwm_val));
//                .level_in(vol_out[15:0]), .pwm_out(pwm_val));
    assign aud_pwm = pwm_val?1'bZ:1'b0;


//    // for debugging
//    // display what is being read from the SD card and what FIFO is outputting
//    parameter ONE_HZ_PERIOD = 25_000_000;          // 1 sec
//    seven_seg_controller #(.ONE_HZ_PERIOD(ONE_HZ_PERIOD)) ssc
//            (.clk_in(clk_25Mhz), .val_in(num_to_display),
//            .rst_in(sys_reset_in),
//            .cat_out({cg,cf,ce,cd,cc,cb,ca}), .an_out(an));

endmodule


module sd_card_reader(
            // sd card inputs
            input wire clk_in, // 25MHz
            input wire sd_cd_in,
            input wire sys_reset_in, // replace w/ your system reset
            inout wire [3:0] sd_dat_in,
            // my inputs
            // these are all 1clk cycle pulses
            input wire play_in, // start playing audio
            input wire pause_in, // stop playing audio
            input wire skip_in, // HIGH if skip buttons are pressed
            input wire [1:0] speed_in, // sw[10:9]
            // TODO: input wire [1:0] speed,
            input wire [31:0] start_addr_in, // addr to start reading from
            input wire [31:0] next_addr_in, // addr for next song

            // sd card outputs
            output logic sd_reset_out,
            output logic sd_sck_out,
            output logic sd_cmd_out,
            // my outputs
            output logic byte_available_out,
            output logic [31:0] num_to_display_out, // for debugging
            output logic [7:0] fifo_data_out // data from FIFO
//            output logic [15:0] fifo_data_out // data from FIFO
    );

    // sd controller module stuff, don't modify
    assign sd_dat_in[2:1] = 2'b11;
    assign sd_reset_out = 0;

    // sd_controller inputs
    logic sd_rd;                    // read enable
    logic sd_wr;                    // write enable
    logic [7:0] sd_din;             // data to sd card
    logic [31:0] sd_addr;           // starting address for read/write operation

    // sd_controller outputs
    logic ready;                 // high when ready for new read/write operation
    logic [7:0] sd_dout;            // data from sd card
    logic byte_available;        // high when byte available for read
    logic ready_for_next_byte;   // high when ready for new byte to be written

    // for graphics
    assign byte_available_out = byte_available;

    // use real or fake SD card module
    parameter WHICH_SD = 1;
    assign sd_wr = 0; // never write to SD
    generate
        if(WHICH_SD) begin
            sd_controller sd(.reset(sys_reset_in), .clk(clk_in), .cs(sd_dat_in[3]),
.miso(sd_dat_in[0]),
                         .mosi(sd_cmd_out), .sclk(sd_sck_out), .ready(ready),
.address(sd_addr),
                         .rd(sd_rd), .dout(sd_dout), .byte_available(byte_available),
                         .wr(sd_wr), .din(sd_din),
.ready_for_next_byte(ready_for_next_byte));
        end else begin
            // for debugging
            fake_sd sd(.reset(sys_reset_in), .clk(clk_in),
                         .mosi(sd_cmd_out), .sclk(sd_sck_out), .ready(ready),
.address(sd_addr),
                         .rd(sd_rd), .dout(sd_dout), .byte_available(byte_available),
                         .wr(sd_wr), .din(sd_din),
.ready_for_next_byte(ready_for_next_byte));
        end
    endgenerate


    // fifo module
    logic fifo_full, fifo_empty, fifo_wr_en, fifo_rd_en;
    logic [7:0] fifo_din, fifo_dout;
    logic [10:0] fifo_data_count;
    fifo_generator_0 my_fifo(.clk(clk_in), .full(fifo_full), .din(fifo_din),
.srst(fifo_reset),
                .wr_en(fifo_wr_en), .empty(fifo_empty), .dout(fifo_dout),
                .rd_en(fifo_rd_en), .data_count(fifo_data_count));


    // filter for removing static and improving audio quality
    logic [7:0] filter_in;
    logic signed [17:0] filter_out;
    fir31 my_fir31(.clk_in(clk_in), .rst_in(sys_reset_in),
            .ready_in(sample_ready_in), .x_in(filter_in), .y_out(filter_out));


    // produce sample_ready_in pulse for fifo to know when to output data
    // sample_count = 520 ~ 48kHz
    // TODO: change speed
    logic [10:0] sample_count;
    parameter SAMPLE_COUNT = 520;//gets approximately (will generate audio at approx 48
kHz sample rate.
    always_comb begin
        case(speed_in)
            2'b00: sample_count = 520;
            2'b01: sample_count = 260; // speed up
            2'b10: sample_count = 1040; // slow up
            default: sample_count = 520;
        endcase
    end
    logic [15:0] sample_counter;
    logic sample_ready_in;
    assign sample_ready_in = (sample_counter == sample_count);
    always_ff @(posedge clk_in)begin
        if(sys_reset_in) begin
            sample_counter <= 0;
        end else begin
            if (sample_counter == sample_count)begin
                sample_counter <= 16'b0;
            end else begin
                sample_counter <= sample_counter + 16'b1;
            end
        end
    end

    // controls when to start and stop reading from sd card
    logic old_byte_available;
    logic [8:0] counter_512;
    always_comb begin
        if(fifo_data_count<1024 && ready) begin
            sd_rd = 1; // 2048 max - 512/section = 1536 =
        end else begin
            sd_rd = 0;
        end
    end

    // SD and FIFO logic
//    logic [7:0] fifo_1_data_out, fifo_2_data_out;
    enum {PLAY, PAUSE} music_player_state;
//    enum {BYTE1, BYTE2, SEND} two_byte_aud_fsm;
    logic fifo_reset; //assign  = (sys_reset_in || skip_in);
    always_ff @(posedge clk_in)begin
        if(sys_reset_in || skip_in) begin
            // sd card
            old_byte_available <= 0;
            sd_addr <= start_addr_in;
            // fifo
            fifo_reset <= 1;
            fifo_wr_en <= 0;
            fifo_rd_en <= 0;
            counter_512 <= 0;
            // fsm
            music_player_state <= PAUSE;
        end else begin
            fifo_reset <= 0;
            // writing data from SD to FIFO
            old_byte_available <= byte_available;
            if(byte_available & ~(old_byte_available)) begin // rising edge of
byte_available
                fifo_wr_en <= 1;
                fifo_din <= sd_dout; // enqueue FIFO with new SD data
//                fifo_din <= sd_dout + 15'b100_0000_0000_0000; // make unsigned
                counter_512 <= counter_512 + 1;
                if(counter_512==(512-1)) begin // finished reading section in SD
                    if((sd_addr+512)<next_addr_in) begin // check if reading the same song
                        sd_addr <= sd_addr + 512; // increment to next section in current
song
                    end
                    // TODO: at the end of a song, next song will not play automatically
                    // need to create corresponding signals to update graphics
                end
            end else begin
                fifo_wr_en <= 0;
            end
            // outputing data from FIFO every sample_ready_in pulse
            // TODO implement fir filtering for better audio quality
            case(music_player_state)
                PLAY: begin
                    if(pause_in)
                        music_player_state <= PAUSE;
                    else begin
//                        case(two_byte_aud_fsm)
//                            BYTE1: begin
//                                if(sample_ready_in) begin
//                                    fifo_rd_en <= 1; // read data from FIFO
//                                    fifo_1_data_out <= fifo_dout;
//                                    two_byte_aud_fsm <= BYTE2;
//                                end else begin
//                                    fifo_rd_en <= 0;
//                                end
//                            end
//                            BYTE2: begin
//                                fifo_rd_en <= 1; // read data from FIFO
//                                fifo_2_data_out <= fifo_dout;
//                                two_byte_aud_fsm <= SEND;
//                            end
//                            SEND: begin
//                                fifo_rd_en <= 0;
//                                fifo_data_out <= {fifo_2_data_out, fifo_1_data_out};
//                                two_byte_aud_fsm <= BYTE1;
//                            end
//                        endcase
```

```
                        if(sample_ready_in) begin
                            fifo_rd_en <= 1; // read data from FIFO
                            fifo_data_out <= fifo_dout;
//                            filter sounds worse than unfiltered... is the amp too high?
//                            filter_in <= fifo_dout;
//                            fifo_data_out <= filter_out[14:7];
                        end else begin
                            fifo_rd_en <= 0;
                        end
                    end
                end
                PAUSE: begin
                    fifo_rd_en <= 0;
                    if(play_in) music_player_state <= PLAY;
                end
            endcase
        end
    end

    // debugging purposes
//    fifo_ila my_ila(.clk(clk_in), .probe0(fifo_din), .probe1(fifo_data_out),
.probe2(two_byte_aud_fsm), .probe3(sample_ready_in));
    // fifo_ila my_ila(.clk(clk_in), .probe0(fifo_din), .probe1(fifo_data_out),
.probe2(byte_available), .probe3(sd_rd));
    // ila_0 my_ila2(.clk(clk_in), .probe0(next_addr_in[7:0]), .probe1(fifo_data_out),
.probe2(sd_addr), .probe3(byte_available), .probe4(sd_rd));
    // ila_1 my_ila3(.clk(clk_in), .probe0(next_addr_in), .probe1(sd_addr));
    assign num_to_display_out = {fifo_data_count, {3'b000, play_in}, fifo_data_out};
endmodule


//Skipping songs logic
module playlist_controller(
            input wire clk_in, // 25MHz
            input wire sys_reset_in,
            input wire skip_forward_in,
            input wire skip_backward_in,

            output logic skip_out,
            output logic [31:0] sd_start_addr_out,
            output logic [31:0] sd_next_addr_out
    );
    // LIST OF SONGS 8-bit music
    logic [31:0] titanic_addr; assign titanic_addr = 32'h200; // decimal: 512
    logic [31:0] slump_addr; assign slump_addr = 32'hBB8200; // decimal: 12,288,400
    logic [31:0] end_day_addr; assign end_day_addr = 32'h11FEC00;
    logic [31:0] empty_addr; assign empty_addr = 32'h1F68200;
    // LIST OF SONGS 16-bit music
//    logic [31:0] titanic_addr; assign titanic_addr = 32'h1770000; // decimal: 512
//    logic [31:0] slump_addr; assign slump_addr = 32'h23fd000; // decimal: 12,288,400
// 3ecfa00


    enum {TITANIC, SLUMP, END_DAY, BUTTON_TRANSITION} current_addr, temp_current_addr;
    logic [31:0] temp_sd_start_addr_out, temp_sd_next_addr_out;
    always_ff @(posedge clk_in)begin
        if(sys_reset_in) begin
            current_addr <= TITANIC;
            sd_start_addr_out <= titanic_addr;
            sd_next_addr_out <= slump_addr;
            skip_out <= 0;
        end else begin
            // detect any skipping action to tell sd_card_reader to reset sd_addr and fifo
elements
//            skip_out <= (skip_forward_in || skip_backward_in)? 1: 0;
            skip_out <= 0;
            case(current_addr)
                BUTTON_TRANSITION: begin
                    if((~skip_forward_in) && (~skip_backward_in)) begin // release button
press
                        current_addr <= temp_current_addr;
                        sd_start_addr_out <= temp_sd_start_addr_out;
                        sd_next_addr_out <= temp_sd_next_addr_out;
                        skip_out <= 1;
                    end
                end
                TITANIC: begin
                    if(skip_backward_in) begin
                        temp_current_addr <= TITANIC;
                        temp_sd_start_addr_out <= titanic_addr;
                        temp_sd_next_addr_out <= slump_addr;
                        current_addr <= BUTTON_TRANSITION;
                    end else begin
                        if(skip_forward_in) begin
                            temp_current_addr <= SLUMP;
                            temp_sd_start_addr_out <= slump_addr;
                            temp_sd_next_addr_out <= end_day_addr;
                            current_addr <= BUTTON_TRANSITION;
                        end
                    end
                end
                SLUMP: begin
                    if(skip_forward_in) begin
                        temp_current_addr <= END_DAY;
                        temp_sd_start_addr_out <= end_day_addr;
                        temp_sd_next_addr_out <= empty_addr; // for now
                        current_addr <= BUTTON_TRANSITION;
                    end else begin
                        if(skip_backward_in) begin
                            temp_current_addr <= TITANIC;
                            temp_sd_start_addr_out <= titanic_addr;
                            temp_sd_next_addr_out <= slump_addr;
                            current_addr <= BUTTON_TRANSITION;
                        end
                    end
                end
                END_DAY: begin
                    if(skip_backward_in) begin
                        temp_current_addr <= SLUMP;
                        temp_sd_start_addr_out <= slump_addr;
                        temp_sd_next_addr_out <= end_day_addr;
                        current_addr <= BUTTON_TRANSITION;
                    end else begin
                        if(skip_forward_in) begin
                            temp_current_addr <= END_DAY;
                            temp_sd_start_addr_out <= end_day_addr;
                            temp_sd_next_addr_out <= empty_addr; // for now
                            current_addr <= BUTTON_TRANSITION;
                        end
                    end
                end
            endcase
        end
    end
    endmodule


    ////Volume Control
    //// TODO: change volume control to be based off of user gestures
    ////module volume_control (input wire [2:0] vol_in, input wire signed [7:0] signal_in,
    output logic signed[7:0] signal_out);
    //module volume_control (input wire [2:0] vol_in, input wire signed [15:0] signal_in,
    output logic signed[15:0] signal_out);
    //    logic [2:0] shift;
    //    assign shift = 3'd7 - vol_in;
    //    assign signal_out = signal_in>>>shift;
    //endmodule


    //Volume Control
    // TODO: change volume control to be based off of user gestures
    //module volume_control (input wire [2:0] vol_in, input wire [15:0] signal_in, output
    logic [15:0] signal_out);
    module volume_control (input wire clk_in, input wire sys_reset_in, input wire [2:0]
    vol_in,
                        input wire [7:0] signal_in, output logic [7:0] signal_out,
                        // my inputs
                        input wire vol_up_in, input wire vol_down_in, input wire
    which_vol);

        logic [2:0] shift;
        enum {ZERO, ONE_FOURTH, ONE_HALF, THREE_FOURTHS, FULL} vol_fsm;

        always_ff @(posedge clk_in)begin
            if (sys_reset_in) begin
                signal_out <= 0;
                vol_fsm <= FULL;
            end else begin
                if (~which_vol) begin // switch 7 off, switches control volume
                shift <= 3'd7 - vol_in;
                signal_out <= signal_in>>shift;
                end else begin // switch 7 on, gestures control volume
                case(vol_fsm)
                    ZERO: begin
                        signal_out <= signal_in>>7;
                        if (vol_up_in) begin
                            vol_fsm <= ONE_FOURTH;
                        end
                    end
                    ONE_FOURTH: begin
                        signal_out <= signal_in>>5;
                        if (vol_up_in) begin
                            vol_fsm <= ONE_HALF;
                        end else begin
                            if (vol_down_in) begin
                                vol_fsm <= ZERO;
                            end
                        end
                    end
                    ONE_HALF: begin
                        signal_out <= signal_in>>3;
                        if (vol_up_in) begin
                            vol_fsm <= THREE_FOURTHS;
                        end else begin
                            if (vol_down_in) begin
                                vol_fsm <= ONE_FOURTH;
                            end
                        end
                    end
                    THREE_FOURTHS: begin
                        signal_out <= signal_in>>1;
                        if (vol_up_in) begin
                            vol_fsm <= FULL;
                        end else begin
                            if (vol_down_in) begin
                                vol_fsm <= ONE_HALF;
                            end
                        end
                    end
                    FULL: begin
                        signal_out <= signal_in;
                        if (vol_down_in) begin
                            vol_fsm <= THREE_FOURTHS;
                        end
                    end
                endcase
            end
        end
    end
endmodule


//PWM generator for audio generation!
module pwm (input wire clk_in, input wire rst_in, input wire [7:0] level_in, output logic
pwm_out);
//module pwm (input wire clk_in, input wire rst_in, input wire [15:0] level_in, output
logic pwm_out);
    logic [7:0] count;
//    logic [15:0] count;
    assign pwm_out = count<level_in;
    always_ff @(posedge clk_in)begin
        if (rst_in)begin
            count <= 8'b0;
        end else begin
            count <= count+8'b1;
        end
    end
endmodule
```

```
`default_nettype wire

/* SD Card controller module. Allows reading from and writing to a microSD card
through SPI mode. */
module sd_controller(
    output reg cs, // Connect to SD_DAT[3].
    output mosi, // Connect to SD_CMD.
    input miso, // Connect to SD_DAT[0].
    output sclk, // Connect to SD_SCK.
                // For SPI mode, SD_DAT[2] and SD_DAT[1] should be held HIGH.
                // SD_RESET should be held LOW.

    input rd,   // Read-enable. When [ready] is HIGH, asseting [rd] will
                // begin a 512-byte READ operation at [address].
                // [byte_available] will transition HIGH as a new byte has been
                // read from the SD card. The byte is presented on [dout].
    output reg [7:0] dout, // Data output for READ operation.
    output reg byte_available, // A new byte has been presented on [dout].

    input wr,   // Write-enable. When [ready] is HIGH, asserting [wr] will
                // begin a 512-byte WRITE operation at [address].
                // [ready_for_next_byte] will transition HIGH to request that
                // the next byte to be written should be presentaed on [din].
    input [7:0] din, // Data input for WRITE operation.
    output reg ready_for_next_byte, // A new byte should be presented on [din].

    input reset, // Resets controller on assertion.
    output ready, // HIGH if the SD card is ready for a read or write operation.
    input [31:0] address,   // Memory address for read/write operation. This MUST
                            // be a multiple of 512 bytes, due to SD sectoring.
    input clk,  // 25 MHz clock.
    output [4:0] status // For debug purposes: Current state of controller.
);

    parameter RST = 0;
    parameter INIT = 1;
    parameter CMD0 = 2;
    parameter CMD55 = 3;
    parameter CMD41 = 4;
    parameter POLL_CMD = 5;

    parameter IDLE = 6;
    parameter READ_BLOCK = 7;
    parameter READ_BLOCK_WAIT = 8;
    parameter READ_BLOCK_DATA = 9;
    parameter READ_BLOCK_CRC = 10;
    parameter SEND_CMD = 11;
    parameter RECEIVE_BYTE_WAIT = 12;
    parameter RECEIVE_BYTE = 13;
    parameter WRITE_BLOCK_CMD = 14;
    parameter WRITE_BLOCK_INIT = 15;
    parameter WRITE_BLOCK_DATA = 16;
    parameter WRITE_BLOCK_BYTE = 17;
    parameter WRITE_BLOCK_WAIT = 18;

    parameter WRITE_DATA_SIZE = 515;

    reg [4:0] state = RST;
```

```
assign status = state;
reg [4:0] return_state;
reg sclk_sig = 0;
reg [55:0] cmd_out;
reg [7:0] recv_data;
reg cmd_mode = 1;
reg [7:0] data_sig = 8'hFF;

reg [9:0] byte_counter;
reg [9:0] bit_counter;

reg [26:0] boot_counter = 27'd100_000_000;
always @(posedge clk) begin
    if(reset == 1) begin
        state <= RST;
        sclk_sig <= 0;
        boot_counter <= 27'd100_000_000;
    end
    else begin
        case(state)
            RST: begin
                if(boot_counter == 0) begin
                    sclk_sig <= 0;
                    cmd_out <= {56{1'b1}};
                    byte_counter <= 0;
                    byte_available <= 0;
                    ready_for_next_byte <= 0;
                    cmd_mode <= 1;
                    bit_counter <= 160;
                    cs <= 1;
                    state <= INIT;
                end
                else begin
                    boot_counter <= boot_counter - 1;
                end
            end
            INIT: begin
                if(bit_counter == 0) begin
                    cs <= 0;
                    state <= CMD0;
                end
                else begin
                    bit_counter <= bit_counter - 1;
                    sclk_sig <= ~sclk_sig;
                end
            end
            CMD0: begin
                cmd_out <= 56'hFF_40_00_00_00_00_95;
                bit_counter <= 55;
                return_state <= CMD55;
                state <= SEND_CMD;
            end
            CMD55: begin
                cmd_out <= 56'hFF_77_00_00_00_00_01;
                bit_counter <= 55;
                return_state <= CMD41;
                state <= SEND_CMD;
            end
```

```
CMD41: begin
    cmd_out <= 56'hFF_69_00_00_00_00_01;
    bit_counter <= 55;
    return_state <= POLL_CMD;
    state <= SEND_CMD;
end
POLL_CMD: begin
    if(recv_data[0] == 0) begin
        state <= IDLE;
    end
    else begin
        state <= CMD55;
    end
end
IDLE: begin
    if(rd == 1) begin
        state <= READ_BLOCK;
    end
    else if(wr == 1) begin
        state <= WRITE_BLOCK_CMD;
    end
    else begin
        state <= IDLE;
    end
end
READ_BLOCK: begin
    cmd_out <= {16'hFF_51, address, 8'hFF};
    bit_counter <= 55;
    return_state <= READ_BLOCK_WAIT;
    state <= SEND_CMD;
end
READ_BLOCK_WAIT: begin
    if(sclk_sig == 1 && miso == 0) begin
        byte_counter <= 511;
        bit_counter <= 7;
        return_state <= READ_BLOCK_DATA;
        state <= RECEIVE_BYTE;
    end
    sclk_sig <= ~sclk_sig;
end
READ_BLOCK_DATA: begin
    dout <= recv_data;
    byte_available <= 1;
    if (byte_counter == 0) begin
        bit_counter <= 7;
        return_state <= READ_BLOCK_CRC;
        state <= RECEIVE_BYTE;
    end
    else begin
        byte_counter <= byte_counter - 1;
        return_state <= READ_BLOCK_DATA;
        bit_counter <= 7;
        state <= RECEIVE_BYTE;
    end
end
READ_BLOCK_CRC: begin
    bit_counter <= 7;
    return_state <= IDLE;
    state <= RECEIVE_BYTE;
end
SEND_CMD: begin
    if (sclk_sig == 1) begin
        if (bit_counter == 0) begin
            state <= RECEIVE_BYTE_WAIT;
        end
        else begin
            bit_counter <= bit_counter - 1;
            cmd_out <= {cmd_out[54:0], 1'b1};
        end
    end
    sclk_sig <= ~sclk_sig;
end
RECEIVE_BYTE_WAIT: begin
    if (sclk_sig == 1) begin
        if (miso == 0) begin
            recv_data <= 0;
            bit_counter <= 6;
            state <= RECEIVE_BYTE;
        end
    end
    sclk_sig <= ~sclk_sig;
end
RECEIVE_BYTE: begin
    byte_available <= 0;
    if (sclk_sig == 1) begin
        recv_data <= {recv_data[6:0], miso};
        if (bit_counter == 0) begin
            state <= return_state;
        end
        else begin
            bit_counter <= bit_counter - 1;
        end
    end
    sclk_sig <= ~sclk_sig;
end
WRITE_BLOCK_CMD: begin
    cmd_out <= {16'hFF_58, address, 8'hFF};
    bit_counter <= 55;
    return_state <= WRITE_BLOCK_INIT;
    state <= SEND_CMD;
   ready_for_next_byte <= 1;
end
WRITE_BLOCK_INIT: begin
    cmd_mode <= 0;
    byte_counter <= WRITE_DATA_SIZE;
    state <= WRITE_BLOCK_DATA;
    ready_for_next_byte <= 0;
end
WRITE_BLOCK_DATA: begin
    if (byte_counter == 0) begin
        state <= RECEIVE_BYTE_WAIT;
        return_state <= WRITE_BLOCK_WAIT;
    end
    else begin
        if ((byte_counter == 2) || (byte_counter == 1)) begin
            data_sig <= 8'hFF;
```

```
                    end
                    else if (byte_counter == WRITE_DATA_SIZE) begin
                        data_sig <= 8'hFE;
                    end
                    else begin
                        data_sig <= din;
                        ready_for_next_byte <= 1;
                    end
                    bit_counter <= 7;
                    state <= WRITE_BLOCK_BYTE;
                    byte_counter <= byte_counter - 1;
                end
            end
            WRITE_BLOCK_BYTE: begin
                if (sclk_sig == 1) begin
                    if (bit_counter == 0) begin
                        state <= WRITE_BLOCK_DATA;
                        ready_for_next_byte <= 0;
                    end
                    else begin
                        data_sig <= {data_sig[6:0], 1'b1};
                        bit_counter <= bit_counter - 1;
                    end;
                end;
                sclk_sig <= ~sclk_sig;
            end
            WRITE_BLOCK_WAIT: begin
                if (sclk_sig == 1) begin
                    if (miso == 1) begin
                        state <= IDLE;
                        cmd_mode <= 1;
                    end
                end
                sclk_sig = ~sclk_sig;
            end
        endcase
    end
end

assign sclk = sclk_sig;
assign mosi = cmd_mode ? cmd_out[55] : data_sig[7];
assign ready = (state == IDLE);
endmodule


// Imitate SD_controller module for testbenching purposes
module fake_sd(
    output mosi, // Connect to SD_CMD.

    output sclk, // Connect to SD_SCK.
                // For SPI mode, SD_DAT[2] and SD_DAT[1] should be held HIGH.
                // SD_RESET should be held LOW.

    input rd,   // Read-enable. When [ready] is HIGH, asseting [rd] will
                // begin a 512-byte READ operation at [address].
                // [byte_available] will transition HIGH as a new byte has been
                // read from the SD card. The byte is presented on [dout].
    output reg [7:0] dout, // Data output for READ operation.
    output reg byte_available, // A new byte has been presented on [dout].

    input wr,   // Write-enable. When [ready] is HIGH, asserting [wr] will
                // begin a 512-byte WRITE operation at [address].
                // [ready_for_next_byte] will transition HIGH to request that
                // the next byte to be written should be presentaed on [din].
    input [7:0] din, // Data input for WRITE operation.
    output reg ready_for_next_byte, // A new byte should be presented on [din].

    input reset, // Resets controller on assertion.
    output ready, // HIGH if the SD card is ready for a read or write operation.
    input [31:0] address,   // Memory address for read/write operation. This MUST
                            // be a multiple of 512 bytes, due to SD sectoring.
    input clk,  // 25 MHz clock.
    output [4:0] status // For debug purposes: Current state of controller.
);

    // timers
    logic [4:0] counter_is_ready, counter_byte_available;
    logic [8:0] counter_data;
    // states
    enum {NOT_READY, READY, READING, DONE} tb_state;
    logic [9:0] data_out;

    // vivado didn't let me assign to ready in always_ff
    logic is_ready;
    assign ready = is_ready;
    assign dout = data_out;

    logic [31:0] addr;
    always_ff @(posedge clk)begin
        if(reset) begin
            addr <= address;
            is_ready <= 0;
            byte_available <= 0;
            data_out <= 0;
            counter_is_ready <= 0;
            counter_byte_available <= 0;
            counter_data <= 0;
            tb_state <= NOT_READY;
        end else begin
            case(tb_state)
                // wait a lil before ready HIGH
                NOT_READY: begin
                    if(counter_is_ready == (2-1)) begin
                        counter_is_ready <= 0;
                        is_ready <= 1;
                        tb_state <= READY;
                    end else begin
                        counter_is_ready <= counter_is_ready + 1;
                    end
                end
                // ok ready HIGH, wait for rd HIGH
                READY: begin
                    if(rd) begin
                        is_ready <= 0;
                        tb_state <= READING;
                    end
                end
                // pass out byte, wait a lil, pass out more
                READING: begin
                    if(counter_byte_available == (5-1)) begin
                        counter_byte_available <= 0;
                        byte_available <= 1;
                        data_out <= data_out + 1;
                        counter_data <= counter_data + 1;
                        // stop after output 512 data samples, go back to starting
tb_state
                        if(counter_data == (512-1)) begin
                            counter_data <= 0;
                            tb_state <= DONE;
                        end
                    end else begin
                        counter_byte_available <= counter_byte_available + 1;
                        byte_available <= 0;
                    end
                end
                DONE: begin
                    byte_available <= 0;
                    data_out <= 0;
                    tb_state <= NOT_READY;
                end

            endcase
        end
    end

endmodule


///////////////////////////////////////////////////////////////////////////////
//
// 31-tap FIR filter, 8-bit signed data, 10-bit signed coefficients.
// ready is asserted whenever there is a new sample on the X input,
// the Y output should also be sampled at the same time.  Assumes at
// least 32 clocks between ready assertions.  Note that since the
// coefficients have been scaled by 2**10, so has the output (it's
// expanded from 8 bits to 18 bits).  To get an 8-bit result from the
// filter just divide by 2**10, ie, use Y[17:10].
//
///////////////////////////////////////////////////////////////////////////////

module fir31(
  input  wire clk_in,rst_in,ready_in,
  input wire signed [7:0] x_in,
  output logic signed [17:0] y_out
);

    logic signed [7:0] sample [31:0];      // 32 element array each 8 bits wide
    logic [4:0] offset;             //pointer for the array!
    logic signed [17:0] accumulator;       // running sum
    logic [5:0] cycle_31;           // count 31 cycles
    logic signed [9:0] coeff;
    logic [4:0] index;
    coeffs31 c (.index_in(cycle_31-1), .coeff_out(coeff));
    // adina
//    logic signed [17:0] multiplier;
//    always_comb begin
//        multiplier = (coeff * sample[offset-cycle_31]);
//    end

    logic signed [17:0] multiply, sum;

    assign index = offset - cycle_31;

    // for now just pass data through
    always_ff @(posedge clk_in) begin
        if (rst_in) begin
            sample <= '{default:0};
            offset <= 0;
            accumulator <= 0;
            cycle_31 <= 0;
            y_out <= 0;
        end else begin
            // if (ready_in) y_out <= {x_in,10'd0};
            if (ready_in) begin
                offset <= offset + 1;
                sample[offset] <= x_in;
                cycle_31 <= 0;
                accumulator <= 0;
            end else begin
                if (cycle_31 < 33) begin
                    cycle_31 <= cycle_31 + 1;
                    multiply <= sample[index];
                    sum <= coeff * multiply;
                    if ( cycle_31 >=2 ) begin
                        accumulator <= accumulator + sum;
                    end
                    // running sum
//                    // adina
//                    if (cycle_31 < 31) begin
//                        accumulator <= accumulator + multiplier;
//                        if (cycle_31 <= 29) cycle_31 <= cycle_31 + 1;
//                        if (cycle_31 == 30) y_out <= accumulator + multiplier;
//                    end
                    // done calculating sum

                end
                if (cycle_31 == 33) y_out <= accumulator;
           end
        end

    end
endmodule


///////////////////////////////////////////////////////////////////////////////
//
// Coefficients for a 31-tap low-pass FIR filter with Wn=.125 (eg, 3kHz for a
// 48kHz sample rate).  Since we're doing integer arithmetic, we've scaled
// the coefficients by 2**10
// Matlab command: round(fir1(30,.125)*1024)
//
```

```
////////////////////////////////////////////////////////////////////////////////

module coeffs31(
  input  wire [4:0] index_in,
  output logic signed [9:0] coeff_out
);
  logic signed [9:0] coeff;
  assign coeff_out = coeff;
  // tools will turn this into a 31x10 ROM
  always_comb begin
    case (index_in)
      5'd0:  coeff = -10'sd1;
      5'd1:  coeff = -10'sd1;
      5'd2:  coeff = -10'sd3;
      5'd3:  coeff = -10'sd5;
      5'd4:  coeff = -10'sd6;
      5'd5:  coeff = -10'sd7;
      5'd6:  coeff = -10'sd5;
      5'd7:  coeff = 10'sd0;
      5'd8:  coeff = 10'sd10;
      5'd9:  coeff = 10'sd26;
      5'd10: coeff = 10'sd46;
      5'd11: coeff = 10'sd69;
      5'd12: coeff = 10'sd91;
      5'd13: coeff = 10'sd110;
      5'd14: coeff = 10'sd123;
      5'd15: coeff = 10'sd128;
      5'd16: coeff = 10'sd123;
      5'd17: coeff = 10'sd110;
      5'd18: coeff = 10'sd91;
      5'd19: coeff = 10'sd69;
      5'd20: coeff = 10'sd46;
      5'd21: coeff = 10'sd26;
      5'd22: coeff = 10'sd10;
      5'd23: coeff = 10'sd0;
      5'd24: coeff = -10'sd5;
      5'd25: coeff = -10'sd7;
      5'd26: coeff = -10'sd6;
      5'd27: coeff = -10'sd5;
      5'd28: coeff = -10'sd3;
      5'd29: coeff = -10'sd1;
      5'd30: coeff = -10'sd1;
      default: coeff = 10'hXXX;
    endcase
  end
endmodule
```

// display

```
`timescale 1ns / 1ps
`default_nettype none

module display(
        input wire clk_25Mhz,
        input wire [15:0] sw,
        input wire btnc, btnu, btnl, btnr, btnd,
        input wire vol_up, vol_down,

        // my inputs
```

```
    input wire [31:0] amp_in,
    // my outputs
    output logic [9:0] draw_addr_out,

    output logic[3:0] vga_r,
    output logic[3:0] vga_b,
    output logic[3:0] vga_g,
    output logic vga_hs,
    output logic vga_vs
);

// sys_reset
logic sys_reset_in;
assign sys_reset_in = btnd;
// play button for now
logic play;
assign play = btnc;
logic pause;
assign pause = btnu;
// skip button for now
logic skip_forward;
logic skip_backward;
assign skip_forward = btnr;
assign skip_backward = btnl;




logic [10:0] hcount;    // pixel on current line
logic [9:0] vcount;     // line number
logic hsync, vsync, blank; //control signals for vga
logic [11:0] pixel;
logic [11:0] rgb;
vga vga1(.vclock_in(clk_25Mhz),.hcount_out(hcount),.vcount_out(vcount),
      .hsync_out(hsync),.vsync_out(vsync),.blank_out(blank));


logic phsync,pvsync,pblank;
music_player_display my_display (.vclock_in(clk_25Mhz),.sys_reset_in(sys_reset_in),
            .hcount_in(hcount),.vcount_in(vcount),
            .hsync_in(hsync),.vsync_in(vsync),.blank_in(blank),
            .play_in(play), .pause_in(pause), .speed_in(sw[10:9]),
            .skip_forward_in(skip_forward), .skip_backward_in(skip_backward),
            .vol_up_in(vol_up), .vol_down_in(vol_down),
            .shift_in(sw[15:13]), .which_vol_in(sw[7]),
            .alpha_in(sw[11:9]), .image_set_in(sw[8]),
            .phsync_out(phsync),.pvsync_out(pvsync),.pblank_out(pblank),
            .pixel_out(pixel));

logic border = (hcount==0 | hcount==639 | vcount==0 | vcount==479 |
               hcount == 319 | vcount == 239);

logic b,hs,vs;
always_ff @(posedge clk_25Mhz) begin
  if (sw[1:0] == 2'b01) begin
     // 1 pixel outline of visible area (white)
     hs <= hsync;
     vs <= vsync;
     b <= blank;
     rgb <= {12{border}};
  end else if (sw[1:0] == 2'b10) begin
     // color bars
```

```
         hs <= hsync;
         vs <= vsync;
         b <= blank;
          rgb <= {{4{hcount[8]}}, {4{hcount[7]}}, {4{hcount[6]}}} ;
      end else begin
         // default: music display
         hs <= phsync;
         vs <= pvsync;
         b <= pblank;

        // fft display on the top half of the screen
        if ( (hcount>25) && (hcount<620) && (vcount>20) && (vcount<265) ) begin
            draw_addr_out <= hcount >> 5; // 1024 / 2^5 = 32 num of bins
            if (((amp_in+250)>>sw[3:0])>=768-vcount)begin
                // rgb <= sw[15:4];
                // split up into diff colors
                if (hcount < 32+4) rgb <= 12'b1111_0000_0000; // red
                if (32+4 <= hcount && hcount < 64+4) rgb <= 12'b1110_0110_0000; // orange
                if (64+4 <= hcount && hcount < 96+4) rgb <= 12'b1111_1100_0000; // yellow
                if (96+4 <= hcount && hcount < 128+4) rgb <= 12'b0111_1100_0000; // green
                if (129+4 <= hcount && hcount < 160+4) rgb <= 12'b0000_1111_1000; // teal
                if (160+4 <= hcount && hcount < 192+4) rgb <= 12'b0000_1111_1111; // blue
                if (192+4 <= hcount && hcount < 224+4) rgb <= 12'b1000_0000_1111; // purple
                if (224+4 <= hcount && hcount < 256+4) rgb <= 12'b1111_0111_0111; // pink
                if (256+4 <= hcount && hcount < 288+4) rgb <= 12'b1111_0000_0000; // red
                if (288+4 <= hcount && hcount < 320+4) rgb <= 12'b1110_0110_0000; // orange
                if (320+4 <= hcount && hcount < 352+4) rgb <= 12'b1111_1100_0000; // yellow
                if (352+4 <= hcount && hcount < 384+4) rgb <= 12'b0111_1100_0000; // green
                if (384+4 <= hcount && hcount < 416+4) rgb <= 12'b0000_1111_1000; // teal
                if (416+4 <= hcount && hcount < 448+4) rgb <= 12'b0000_1111_1111; // blue
                if (448+4 <= hcount && hcount < 480+4) rgb <= 12'b1000_0000_1111; // purple
                if (480+4 <= hcount) rgb <= 12'b1111_0111_0111; // pink
            end else begin
                rgb <= 12'b0000_0000_0000;
            end
        end else begin
            // play button, volume, speed, song title, music bar, instructions on the bottom
            rgb <= pixel;
        end
      end
    end


    // the following lines are required for the Nexys4 VGA circuit - do not change
    assign vga_r = ~b ? rgb[11:8]: 0;
    assign vga_g = ~b ? rgb[7:4] : 0;
    assign vga_b = ~b ? rgb[3:0] : 0;

    assign vga_hs = ~hs;
    assign vga_vs = ~vs;

endmodule






module music_player_display (
   input wire vclock_in,        // 25MHz clock
   input wire sys_reset_in,         // 1 to initialize module
   input wire [10:0] hcount_in, // horizontal index of current pixel (0..1023)
   input wire [9:0]  vcount_in, // vertical index of current pixel (0..767)
   input wire hsync_in,         // XVGA horizontal sync signal (active low)
   input wire vsync_in,         // XVGA vertical sync signal (active low)
   input wire blank_in,         // XVGA blanking (1 means output black pixel)
   input wire [2:0] alpha_in,      //alpha flavor,
   input wire image_set_in,     //when 1, puck becomes image, when 0, puck is plain box
   // my inputs
   input wire play_in,
   input wire pause_in,
   input wire skip_forward_in,
   input wire skip_backward_in,
   input wire [1:0] speed_in,
   input wire vol_up_in,
   input wire vol_down_in,
   input wire [2:0] shift_in,
   input wire which_vol_in,

   output logic phsync_out,       // pong game's horizontal sync
   output logic pvsync_out,       // pong game's vertical sync
   output logic pblank_out,       // pong game's blanking
   output logic [11:0] temp_pixel_out, pixel_out  // pong game's pixel  // r=11:8, g=7:4,
b=3:0
   );

    assign phsync_out = hsync_in;
    assign pvsync_out = vsync_in;
    assign pblank_out = blank_in;

//////////////////////////////////////////////////////////////////
// INSTRUCTIONS + LABELS
//////////////////////////////////////////////////////////////////
    // instructions
    logic [11:0] instructions_pixel;
    instructions_blob #(.WIDTH(11'd520), .HEIGHT(11'd100))
        my_instructions (.pixel_clk_in(vclock_in),
            .x_in(11'd0),.y_in(11'd370),
            .hcount_in(hcount_in),.vcount_in(vcount_in),
            .pixel_out(instructions_pixel));
//    // volume text
//    logic [11:0] volume_pixel;
//    volume_blob #(.WIDTH(11'd50), .HEIGHT(11'd20))
//        my_volume (.pixel_clk_in(vclock_in),
//            .x_in(11'd555),.y_in(11'd384),
//            .hcount_in(hcount_in),.vcount_in(vcount_in),
//            .pixel_out(volume_pixel));
    // speed text
    logic [11:0] speed_pixel;
    speed_blob #(.WIDTH(11'd50), .HEIGHT(11'd20))
        my_speed (.pixel_clk_in(vclock_in),
            .x_in(11'd558),.y_in(11'd430),
            .hcount_in(hcount_in),.vcount_in(vcount_in),
            .pixel_out(speed_pixel));
    // letter x after speed digits
    logic [11:0] letter_x_pixel;
    letter_x_blob #(.WIDTH(11'd21), .HEIGHT(11'd24))
        my_letter_x (.pixel_clk_in(vclock_in),
            .x_in(11'd590),.y_in(11'd448),
            .hcount_in(hcount_in),.vcount_in(vcount_in),
            .pixel_out(letter_x_pixel));
//    // percentage after volume digits
//    logic [11:0] percentage_pixel;
//    percentage_blob #(.WIDTH(11'd21), .HEIGHT(11'd24))
//        my_percentage (.pixel_clk_in(vclock_in),
//            .x_in(11'd590),.y_in(11'd405),
//            .hcount_in(hcount_in),.vcount_in(vcount_in),
//            .pixel_out(percentage_pixel));
////////////////////////////////////////////////////////////////
// BUTTONS
////////////////////////////////////////////////////////////////
    // play btn
    logic [11:0] btn_x, btn_y, play_btn_pixel, pause_btn_pixel;
    assign btn_x = 11'd20; assign btn_y = 11'd300;
    play_blob #(.WIDTH(11'd48), .HEIGHT(11'd48))
        my_play_blob (.pixel_clk_in(vclock_in),
            .x_in(btn_x),.y_in(btn_y),
            .hcount_in(hcount_in),.vcount_in(vcount_in),
            .pixel_out(play_btn_pixel));
    // pause btn
    pause_blob #(.WIDTH(11'd48), .HEIGHT(11'd48))
        my_pause_blob (.pixel_clk_in(vclock_in),
            .x_in(btn_x),.y_in(btn_y),
            .hcount_in(hcount_in),.vcount_in(vcount_in),
            .pixel_out(pause_btn_pixel));
////////////////////////////////////////////////////////////////////////////////
// MUSIC BAR
////////////////////////////////////////////////////////////////////////////////
    // music bar
    logic [11:0] music_bar_pixel;
    logic [8:0] music_length; // in seconds
    music_bar_blob #(.WIDTH(11'd512), .HEIGHT(11'd5))
       my_music_bar (.pixel_clk_in(vclock_in),
           .x_in(11'd64),.y_in(11'd360),
           .music_len_in(music_length), .play_in(play_in), .pause_in(pause_in),
           .skip_forward_in(skip_forward_in), .skip_backward_in(skip_backward_in),
           .sys_reset_in(sys_reset_in),
           .hcount_in(hcount_in),.vcount_in(vcount_in),
           .pixel_out(music_bar_pixel));
//////////////////////////////////////////////////////////////////////////////
// DIGITS
//////////////////////////////////////////////////////////////////////////////
    logic [11:0] speed_0_digit_pixel;
    speed_0_digit_blob #(.WIDTH(11'd16), .HEIGHT(11'd16))
       my_speed_0_digit (.pixel_clk_in(vclock_in),
           .x_in(11'd575),.y_in(11'd450),
           .hcount_in(hcount_in),.vcount_in(vcount_in),
           .speed_in(speed_in),
           .pixel_out(speed_0_digit_pixel));
    logic [11:0] speed_1_digit_pixel;
    speed_1_digit_blob #(.WIDTH(11'd16), .HEIGHT(11'd16))
       my_speed_1_digit (.pixel_clk_in(vclock_in),
           .x_in(11'd555),.y_in(11'd450),
           .hcount_in(hcount_in),.vcount_in(vcount_in),
           .speed_in(speed_in),
           .pixel_out(speed_1_digit_pixel));
    // inbetween speed digits
    logic [11:0] period_pixel;
    period_blob #(.WIDTH(11'd4), .HEIGHT(11'd4))
       my_period (.pixel_clk_in(vclock_in),
           .x_in(11'd572),.y_in(11'd462),
           .hcount_in(hcount_in),.vcount_in(vcount_in),
           .pixel_out(period_pixel));
    logic [11:0] music_0_digit_pixel;
    music_0_digit_blob #(.WIDTH(11'd16), .HEIGHT(11'd16))
       my_music_0_digit (.pixel_clk_in(vclock_in),
           .x_in(11'd87),.y_in(11'd335),
           .music_len_in(music_length), .play_in(play_in), .pause_in(pause_in),
           .skip_forward_in(skip_forward_in), .skip_backward_in(skip_backward_in),
           .sys_reset_in(sys_reset_in),
           .hcount_in(hcount_in),.vcount_in(vcount_in),
           .pixel_out(music_0_digit_pixel));
    logic [11:0] music_1_digit_pixel;
    music_1_digit_blob #(.WIDTH(11'd16), .HEIGHT(11'd16))
       my_music_1_digit (.pixel_clk_in(vclock_in),
           .x_in(11'd110),.y_in(11'd335),
           .music_len_in(music_length), .play_in(play_in), .pause_in(pause_in),
           .skip_forward_in(skip_forward_in), .skip_backward_in(skip_backward_in),
           .sys_reset_in(sys_reset_in),
           .hcount_in(hcount_in),.vcount_in(vcount_in),
           .pixel_out(music_1_digit_pixel));
    logic [11:0] music_2_digit_pixel;
    music_2_digit_blob #(.WIDTH(11'd16), .HEIGHT(11'd16))
       my_music_2_digit (.pixel_clk_in(vclock_in),
           .x_in(11'd126),.y_in(11'd335),
           .music_len_in(music_length), .play_in(play_in), .pause_in(pause_in),
           .skip_forward_in(skip_forward_in), .skip_backward_in(skip_backward_in),
           .sys_reset_in(sys_reset_in),
           .hcount_in(hcount_in),.vcount_in(vcount_in),
           .pixel_out(music_2_digit_pixel));
    // inbetween the time digits
    logic [11:0] semicolon_0_pixel;
    semicolon_0_blob #(.WIDTH(11'd4), .HEIGHT(11'd4))
       my_semicolon_1 (.pixel_clk_in(vclock_in),
           .x_in(11'd105),.y_in(11'd339),
           .hcount_in(hcount_in),.vcount_in(vcount_in),
           .pixel_out(semicolon_0_pixel));
    logic [11:0] semicolon_1_pixel;
    semicolon_1_blob #(.WIDTH(11'd4), .HEIGHT(11'd4))
       my_semicolon_2 (.pixel_clk_in(vclock_in),
           .x_in(11'd105),.y_in(11'd345),
           .hcount_in(hcount_in),.vcount_in(vcount_in),
           .pixel_out(semicolon_1_pixel));
//    logic [11:0] volume_0_digit_pixel;
//    volume_0_digit_blob #(.WIDTH(11'd16), .HEIGHT(11'd16))
//       my_volume_0_digit (.pixel_clk_in(vclock_in),
//            .x_in(11'd574),.y_in(11'd410),
//            .vol_up_in(vol_up_in), .vol_down_in(vol_down_in),
//            .shift_in(shift_in), .which_vol_in(which_vol_in),
//            .hcount_in(hcount_in),.vcount_in(vcount_in),
//            .pixel_out(volume_0_digit_pixel));
//    logic [11:0] volume_1_digit_pixel;
//    volume_1_digit_blob #(.WIDTH(11'd16), .HEIGHT(11'd16))
//       my_volume_1_digit (.pixel_clk_in(vclock_in),
//            .x_in(11'd559),.y_in(11'd410),
//            .vol_up_in(vol_up_in), .vol_down_in(vol_down_in),
//            .shift_in(shift_in), .which_vol_in(which_vol_in),
//            .hcount_in(hcount_in),.vcount_in(vcount_in),
//            .pixel_out(volume_1_digit_pixel));
//    logic [11:0] volume_2_digit_pixel;
//    volume_2_digit_blob #(.WIDTH(11'd16), .HEIGHT(11'd16))
//       my_volume_2_digit (.pixel_clk_in(vclock_in),
//            .x_in(11'd543),.y_in(11'd410),
//            .vol_up_in(vol_up_in), .vol_down_in(vol_down_in),
//            .shift_in(shift_in), .which_vol_in(which_vol_in),
//            .hcount_in(hcount_in),.vcount_in(vcount_in),
//            .pixel_out(volume_2_digit_pixel));
////////////////////////////////////////////////////////////////////////////
// SONG TITLES
```

```
/////////////////////////////////////////////////////////////////////////////
    logic [11:0] titanic_title_pixel;
    titanic_blob #(.WIDTH(11'd200), .HEIGHT(11'd30))
        my_titanic (.pixel_clk_in(vclock_in),
            .x_in(11'd80),.y_in(11'd300),
            .hcount_in(hcount_in),.vcount_in(vcount_in),
            .pixel_out(titanic_title_pixel));
    // volume digits - 0, 12, 25, 37, 50, 62, 75, 87, 100
//    vol_0_digit_blob #(.WIDTH(11'd16), .HEIGHT(11'd16))
//       my_vol_0_digit (.pixel_clk_in(vclock_in),
//           .x_in(11'd575),.y_in(11'd450),
//           .hcount_in(hcount_in),.vcount_in(vcount_in),
//           .vol_in(vol_in),
//           .pixel_out(vol_0_digit_pixel));
    // slump song
    logic [11:0] slump_title_pixel;
    slump_blob #(.WIDTH(11'd200), .HEIGHT(11'd30))
        my_slump (.pixel_clk_in(vclock_in),
            .x_in(11'd80),.y_in(11'd300),
            .hcount_in(hcount_in),.vcount_in(vcount_in),
            .pixel_out(slump_title_pixel));
   // end of the day song
   logic [11:0] end_day_title_pixel;
   end_day_blob #(.WIDTH(11'd200), .HEIGHT(11'd30))
       my_end_day (.pixel_clk_in(vclock_in),
           .x_in(11'd80),.y_in(11'd300),
           .hcount_in(hcount_in),.vcount_in(vcount_in),
           .pixel_out(end_day_title_pixel));


    logic [11:0] title_pixel, btn_pixel;
    assign pixel_out = btn_pixel | title_pixel | instructions_pixel
//                       | music_bar_pixel | volume_pixel | speed_pixel
                        | music_bar_pixel | speed_pixel
                       | letter_x_pixel
                       | speed_0_digit_pixel | speed_1_digit_pixel
                       | period_pixel | music_0_digit_pixel
                       | music_1_digit_pixel | music_2_digit_pixel
                       | semicolon_0_pixel | semicolon_1_pixel
                       //| volume_0_digit_pixel | volume_1_digit_pixel
                       //| volume_2_digit_pixel
                       ;

    enum {PLAY, PAUSE} btn_state;
    enum {TITANIC, SLUMP, END_DAY, BUTTON_TRANSITION} temp_song_state, cur_song_state;

    always_ff @ (posedge vclock_in) begin
        if(sys_reset_in)begin
            btn_state <= PAUSE;
            cur_song_state <= TITANIC;
        end else begin
            // play or pause button
            case(btn_state)
                PLAY: begin
                    btn_pixel <= pause_btn_pixel;
                    if (pause_in || skip_forward_in || skip_backward_in) btn_state <= PAUSE;
                end
                PAUSE: begin
                    btn_pixel <= play_btn_pixel;
                    if (play_in) btn_state <= PLAY;
                end
            endcase
            // music title
            case(cur_song_state)
                BUTTON_TRANSITION: begin
                    if((~skip_forward_in) && (~skip_backward_in)) begin // release button
press
                        cur_song_state <= temp_song_state;
                    end
                end
                TITANIC: begin
                    title_pixel <= titanic_title_pixel;
                    music_length <= 9'd255;
                    if(skip_backward_in) begin
                        temp_song_state <= TITANIC;
                        cur_song_state <= BUTTON_TRANSITION;
                    end else begin
                        if(skip_forward_in) begin
                            temp_song_state <= SLUMP;
                            cur_song_state <= BUTTON_TRANSITION;
                        end
                    end
                end
                SLUMP: begin
                    title_pixel <= slump_title_pixel;
                    music_length <= 9'd137;
                    if(skip_forward_in) begin
                        temp_song_state <= END_DAY;
                        cur_song_state <= BUTTON_TRANSITION;
                    end else begin
                        if(skip_backward_in) begin
                            temp_song_state <= TITANIC;
                            cur_song_state <= BUTTON_TRANSITION;
                        end
                    end
                end
                END_DAY: begin
                    title_pixel <= end_day_title_pixel;
                    music_length <= 9'd292;
                    if(skip_backward_in) begin
                        temp_song_state <= SLUMP;
                        cur_song_state <= BUTTON_TRANSITION;
                    end else begin
                        if(skip_forward_in) begin
                            temp_song_state <= END_DAY;
                            cur_song_state <= BUTTON_TRANSITION;
                        end
                    end
                end
            endcase
        end
    end

endmodule




////////////////////////////////////////////////////
//
// picture_blobs to display a pictures
//
//////////////////////////////////////////////////
```

```
module play_blob
   #(parameter WIDTH = 256,     // default picture width
               HEIGHT = 240)    // default picture height
   (input wire pixel_clk_in,
    input wire [10:0] x_in,hcount_in,
    input wire [9:0] y_in,vcount_in,
    output logic [11:0] pixel_out);

   logic [15:0] image_addr;   // num of bits for 256*240 ROM
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
   play_btn_rom rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));

   // use color map to create 4 bits R, 4 bits G, 4 bits B
   play_btn_red rcm (.clka(pixel_clk_in), .addra(image_bits), .douta(red_mapped));
   // note the one clock cycle delay in pixel!
   always_ff @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        // use MSB 4 bits
        pixel_out <= {red_mapped[7:4], red_mapped[7:4], red_mapped[7:4]}; // greyscale
//        pixel_out <= {red_mapped[7:4], green_mapped[7:4], blue_mapped[7:4]}; // only red
hues
        else pixel_out <= 0;
   end
endmodule


module pause_blob
   #(parameter WIDTH = 256,     // default picture width
               HEIGHT = 240)    // default picture height
   (input wire pixel_clk_in,
    input wire [10:0] x_in,hcount_in,
    input wire [9:0] y_in,vcount_in,
    output logic [11:0] pixel_out);

   logic [15:0] image_addr;   // num of bits for 256*240 ROM
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
   pause_btn_rom  rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));

   // use color map to create 4 bits R, 4 bits G, 4 bits B
   pause_btn_red rcm (.clka(pixel_clk_in), .addra(image_bits), .douta(red_mapped));
   // note the one clock cycle delay in pixel!
   always_ff @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        // use MSB 4 bits
         pixel_out <= {red_mapped[7:4], red_mapped[7:4], red_mapped[7:4]}; // greyscale
//        pixel_out <= {red_mapped[7:4], green_mapped[7:4], blue_mapped[7:4]}; // only red
hues
        else pixel_out <= 0;
   end
endmodule


module instructions_blob
```

```
   #(parameter WIDTH = 256,     // default picture width
               HEIGHT = 240)    // default picture height
   (input wire pixel_clk_in,
    input wire [10:0] x_in,hcount_in,
    input wire [9:0] y_in,vcount_in,
    output logic [11:0] pixel_out);

   logic [15:0] image_addr;   // num of bits for 256*240 ROM
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
   instructions_rom  rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));

   // use color map to create 4 bits R, 4 bits G, 4 bits B
   instructions_red rcm (.clka(pixel_clk_in), .addra(image_bits), .douta(red_mapped));
   // note the one clock cycle delay in pixel!
   always_ff @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        // use MSB 4 bits
         pixel_out <= {red_mapped[7:4], red_mapped[7:4], red_mapped[7:4]}; // greyscale
        else pixel_out <= 0;
   end
endmodule


module volume_blob
   #(parameter WIDTH = 256,     // default picture width
               HEIGHT = 240)    // default picture height
   (input wire pixel_clk_in,
    input wire [10:0] x_in,hcount_in,
    input wire [9:0] y_in,vcount_in,
    output logic [11:0] pixel_out);

   logic [15:0] image_addr;   // num of bits for 256*240 ROM
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
   volume_rom  rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));
   volume_red rcm (.clka(pixel_clk_in), .addra(image_bits), .douta(red_mapped));
   // note the one clock cycle delay in pixel!
   always_ff @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        pixel_out <= {red_mapped[7:4], red_mapped[7:4], red_mapped[7:4]}; // greyscale
        else pixel_out <= 0;
   end
endmodule


module percentage_blob
#(parameter WIDTH = 256,     // default picture width
               HEIGHT = 240)    // default picture height
   (input wire pixel_clk_in,
    input wire [10:0] x_in,hcount_in,
    input wire [9:0] y_in,vcount_in,
    output logic [11:0] pixel_out);

   logic [15:0] image_addr;   // num of bits for 256*240 ROM
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;
   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
   percentage_rom  rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));
   percentage_red rcm (.clka(pixel_clk_in), .addra(image_bits), .douta(red_mapped));
   // note the one clock cycle delay in pixel!
   always_ff @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        pixel_out <= {red_mapped[7:4], red_mapped[7:4], red_mapped[7:4]}; // greyscale
        else pixel_out <= 0;
   end
endmodule


module speed_blob
   #(parameter WIDTH = 256,     // default picture width
               HEIGHT = 240)    // default picture height
   (input wire pixel_clk_in,
    input wire [10:0] x_in,hcount_in,
    input wire [9:0] y_in,vcount_in,
    output logic [11:0] pixel_out);

   logic [15:0] image_addr;   // num of bits for 256*240 ROM
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
   speed_rom  rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));
   speed_red rcm (.clka(pixel_clk_in), .addra(image_bits), .douta(red_mapped));
   // note the one clock cycle delay in pixel!
   always_ff @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        pixel_out <= {red_mapped[7:4], red_mapped[7:4], red_mapped[7:4]}; // greyscale
        else pixel_out <= 0;
   end
endmodule


module letter_x_blob
#(parameter WIDTH = 256,     // default picture width
               HEIGHT = 240)    // default picture height
   (input wire pixel_clk_in,
    input wire [10:0] x_in,hcount_in,
    input wire [9:0] y_in,vcount_in,
    output logic [11:0] pixel_out);

   logic [15:0] image_addr;   // num of bits for 256*240 ROM
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
   letter_x_rom  rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));
   letter_x_red rcm (.clka(pixel_clk_in), .addra(image_bits), .douta(red_mapped));
   // note the one clock cycle delay in pixel!
   always_ff @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        pixel_out <= {red_mapped[7:4], red_mapped[7:4], red_mapped[7:4]}; // greyscale
        else pixel_out <= 0;
   end
```

```
endmodule


module music_bar_blob
   #(parameter WIDTH = 256,     // default picture width
               HEIGHT = 240,    // default picture height
               k = 48_828)      // 25_000_000 / 512
   (input wire pixel_clk_in,
    input wire [10:0] x_in,hcount_in,
    input wire [9:0] y_in,vcount_in,
    input wire [8:0] music_len_in,
    input wire play_in,
    input wire pause_in,
    input wire skip_forward_in,
    input wire skip_backward_in,
    input wire sys_reset_in,
    output logic [11:0] pixel_out);

   ;
   logic [15:0] image_addr;   // num of bits for 256*240 ROM
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;

   logic [31:0] k_song, k_counter;
   assign k_song = (music_len_in == 0)? k: k * music_len_in;
   logic [9:0] music_bar_pixel; // 0 ... 511
   enum {RESET, PLAY, PAUSE, END_SONG} music_bar_fsm;

   always_ff @ (posedge pixel_clk_in) begin
        if (sys_reset_in) music_bar_fsm <= RESET;
        // grey bar on new song or sys_reset
        case(music_bar_fsm)
            RESET: begin
                music_bar_pixel <= x_in;
                k_counter <= 0;
                music_bar_fsm <= PAUSE;
            end
            PLAY: begin
                // fill in section of bar with white when music playing
                k_counter <= (k_counter==k_song)? 0: k_counter + 1;
                music_bar_pixel <= ( (k_counter==k_song)
                                    && (music_bar_pixel<(WIDTH+x_in)) )?
                        music_bar_pixel + 1 : music_bar_pixel;
                if (pause_in) music_bar_fsm <= PAUSE;
                if (skip_backward_in || skip_forward_in) music_bar_fsm <= RESET;
                if (music_bar_pixel >= (WIDTH+x_in)) music_bar_fsm <= END_SONG;
            end
            PAUSE: begin
                if (play_in) music_bar_fsm <= PLAY;
                if (skip_backward_in || skip_forward_in) music_bar_fsm <= RESET;
            end
            // bar is filled wait for user to go to next song
            END_SONG: begin
                if (skip_backward_in || skip_forward_in) music_bar_fsm <= RESET;
            end
        endcase

        if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT))) begin
            // use MSB 4 bits
            if ( hcount_in < music_bar_pixel ) begin
                pixel_out <= {4'b1111, 4'b1111, 4'b1111}; // white
            end else begin
                pixel_out <= {4'b1000, 4'b1000, 4'b1000}; // grey
            end
        end else pixel_out <= 0;
   end
endmodule


module speed_0_digit_blob
#(parameter WIDTH = 256,     // default picture width
               HEIGHT = 240)    // default picture height
   (input wire pixel_clk_in,
    input wire [10:0] x_in,hcount_in,
    input wire [9:0] y_in,vcount_in,
    input wire [1:0] speed_in,
    output logic [11:0] pixel_out);

   logic [15:0] image_addr;   // num of bits for 256*240 ROM
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

    logic [15:0] addr_offset;
    always_comb begin
        case(speed_in)
            2'b00: addr_offset = 0; // regular speed, display 1.0
            2'b01: addr_offset = 0; // display 2.0
            2'b10: addr_offset = 5*256; // display 0.5
            default: addr_offset = 0;
        endcase
    end

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + ((vcount_in-y_in) * WIDTH) + addr_offset;
   digits_rom  rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));
   // note the one clock cycle delay in pixel!
   always_ff @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        pixel_out <= {3{image_bits[3:0]}}; // white
        else pixel_out <= 0;
   end
endmodule


module speed_1_digit_blob
#(parameter WIDTH = 256,     // default picture width
               HEIGHT = 240)    // default picture height
   (input wire pixel_clk_in,
    input wire [10:0] x_in,hcount_in,
    input wire [9:0] y_in,vcount_in,
    input wire [1:0] speed_in,
    output logic [11:0] pixel_out);

   logic [15:0] image_addr;   // num of bits for 256*240 ROM
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

    logic [15:0] addr_offset;
    always_comb begin
        case(speed_in)
            2'b00: addr_offset = 256; // regular speed, display 1.0
            2'b01: addr_offset = 2*256; // display 2.0
            2'b10: addr_offset = 0; // display 0.5
            default: addr_offset = 0;
        endcase
    end

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + ((vcount_in-y_in) * WIDTH) + addr_offset;
   digits_rom  rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));
   // note the one clock cycle delay in pixel!
   always_ff @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        pixel_out <= {3{image_bits[3:0]}}; // white
        else pixel_out <= 0;
   end
endmodule


module period_blob
   #(parameter WIDTH = 256,     // default picture width
               HEIGHT = 240,    // default picture height
               k = 48_828)      // 25_000_000 / 512
   (input wire pixel_clk_in,
    input wire [10:0] x_in,hcount_in,
    input wire [9:0] y_in,vcount_in,
    output logic [11:0] pixel_out);

   ;
   logic [15:0] image_addr;   // num of bits for 256*240 ROM
   logic [7:0] image_bits;

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;

   always_ff @ (posedge pixel_clk_in) begin
        if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT))) begin
            // use MSB 4 bits
                pixel_out <= {4'b1111, 4'b1111, 4'b1111}; // white
        end else pixel_out <= 0;
   end
endmodule


module music_0_digit_blob
#(parameter WIDTH = 256,     // default picture width
               HEIGHT = 240)    // default picture height
   (input wire pixel_clk_in,
    input wire [10:0] x_in,hcount_in,
    input wire [9:0] y_in,vcount_in,
    input wire [8:0] music_len_in,
    input wire play_in,
    input wire pause_in,
    input wire skip_forward_in,
    input wire skip_backward_in,
    input wire sys_reset_in,
    output logic [11:0] pixel_out);

   logic [15:0] image_addr;   // num of bits for 256*240 ROM
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

    logic [15:0] addr_offset;
    logic [31:0] second_counter; // 0 ... 25_000_000
    logic [3:0] seconds_counter; // 0 ... 9
    logic [3:0] seconds2_counter; // 0 ... 9
    logic [3:0] display_min; // 0 ... 9
    logic [8:0] end_song_counter; // 0 ... song len
    enum {RESET, PLAY, PAUSE, END_SONG} music_fsm;
    always_ff @ (posedge pixel_clk_in) begin
        if (sys_reset_in) music_fsm <= RESET;
        case (music_fsm)
            RESET: begin
                second_counter <= 0;
                seconds_counter <= 0;
                end_song_counter <= 0;
                seconds2_counter <= 0;
                display_min <= 0;
                music_fsm <= PAUSE;
            end
            PLAY: begin
                second_counter <= (second_counter == 31'd25_000_000)? 0: second_counter + 1;
                if (second_counter == 31'd25_000_000) begin
                    if (seconds_counter == 4'd9) begin
                        seconds_counter <= 0;
                        if (seconds2_counter == 4'd5) begin
                            seconds2_counter <= 0;
                            display_min <= (display_min==4'd5)? 0: display_min+1;
                        end else begin
                            seconds2_counter <= seconds2_counter+1;
                        end
                    end else begin
                        seconds_counter <= seconds_counter + 1;
                    end
                end
                end_song_counter <= (second_counter == 31'd25_000_000)? end_song_counter + 1:
end_song_counter;
                if (pause_in) music_fsm <= PAUSE;
                if (skip_backward_in || skip_forward_in) music_fsm <= RESET;
                if (end_song_counter >= music_len_in) music_fsm <= END_SONG;
            end
            PAUSE: begin
                if (play_in) music_fsm <= PLAY;
                if (skip_backward_in || skip_forward_in) music_fsm <= RESET;
            end
            END_SONG: begin
                if (skip_backward_in || skip_forward_in) music_fsm <= RESET;
            end
        endcase
    end
    assign addr_offset = display_min * 256;
   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + ((vcount_in-y_in) * WIDTH) + addr_offset;
   digits_rom  rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));
   // note the one clock cycle delay in pixel!
   always_ff @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        pixel_out <= {3{image_bits[3:0]}}; // white
        else pixel_out <= 0;
   end
endmodule


module music_1_digit_blob
```

```
#(parameter WIDTH = 256,     // default picture width
               HEIGHT = 240)    // default picture height
   (input wire pixel_clk_in,
    input wire [10:0] x_in,hcount_in,
    input wire [9:0] y_in,vcount_in,
    input wire [8:0] music_len_in,
    input wire play_in,
    input wire pause_in,
    input wire skip_forward_in,
    input wire skip_backward_in,
    input wire sys_reset_in,
    output logic [11:0] pixel_out);

   logic [15:0] image_addr;   // num of bits for 256*240 ROM
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

    logic [15:0] addr_offset;
    logic [31:0] second_counter; // 0 ... 25_000_000
    logic [3:0] seconds_counter; // 0 ... 9
    logic [3:0] display_sec; // 0 ... 5
    logic [8:0] end_song_counter; // 0 ... song len
    enum {RESET, PLAY, PAUSE, END_SONG} music_fsm;
    always_ff @ (posedge pixel_clk_in) begin
        if (sys_reset_in) music_fsm <= RESET;
        case (music_fsm)
            RESET: begin
                second_counter <= 0;
                seconds_counter <= 0;
                end_song_counter <= 0;
                display_sec <= 0;
                music_fsm <= PAUSE;
            end
            PLAY: begin
                second_counter <= (second_counter == 31'd25_000_000)? 0: second_counter + 1;
                if (second_counter == 31'd25_000_000) begin
                    if (seconds_counter == 4'd9) begin
                        seconds_counter <= 0;
                        display_sec <= (display_sec == 4'd5)? 0: display_sec+1;
                    end else begin
                        seconds_counter <= seconds_counter + 1;
                    end
                end
                end_song_counter <= (second_counter == 31'd25_000_000)? end_song_counter + 1:
end_song_counter;
                if (pause_in) music_fsm <= PAUSE;
                if (skip_backward_in || skip_forward_in) music_fsm <= RESET;
                if (end_song_counter >= music_len_in) music_fsm <= END_SONG;
            end
            PAUSE: begin
                if (play_in) music_fsm <= PLAY;
                if (skip_backward_in || skip_forward_in) music_fsm <= RESET;
            end
            END_SONG: begin
                if (skip_backward_in || skip_forward_in) music_fsm <= RESET;
            end
        endcase
    end
    assign addr_offset = display_sec * 256;
   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + ((vcount_in-y_in) * WIDTH) + addr_offset;
   digits_rom  rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));
   // note the one clock cycle delay in pixel!
   always_ff @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        pixel_out <= {3{image_bits[3:0]}}; // white
        else pixel_out <= 0;
   end
endmodule


module music_2_digit_blob
#(parameter WIDTH = 256,     // default picture width
               HEIGHT = 240)    // default picture height
   (input wire pixel_clk_in,
    input wire [10:0] x_in,hcount_in,
    input wire [9:0] y_in,vcount_in,
    input wire [8:0] music_len_in,
    input wire play_in,
    input wire pause_in,
    input wire skip_forward_in,
    input wire skip_backward_in,
    input wire sys_reset_in,
    output logic [11:0] pixel_out);

   logic [15:0] image_addr;   // num of bits for 256*240 ROM
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

    logic [15:0] addr_offset;
    logic [31:0] second_counter; // 0 ... 25_000_000
    logic [3:0] display_sec; // 0 ... 9
    logic [8:0] end_song_counter; // 0 ... song len
    enum {RESET, PLAY, PAUSE, END_SONG} music_fsm;
    always_ff @ (posedge pixel_clk_in) begin
        if (sys_reset_in) music_fsm <= RESET;
        case (music_fsm)
            RESET: begin
                second_counter <= 0;
                display_sec <= 0;
                end_song_counter <= 0;
                music_fsm <= PAUSE;
            end
            PLAY: begin
                second_counter <= (second_counter == 31'd25_000_000)? 0: second_counter + 1;
                if (second_counter == 31'd25_000_000) begin
                    if (display_sec == 4'd9) begin
                        display_sec <= 0;
                    end else begin
                        display_sec <= display_sec + 1;
                    end
                end
                end_song_counter <= (second_counter == 31'd25_000_000)? end_song_counter + 1:
end_song_counter;
                if (pause_in) music_fsm <= PAUSE;
                if (skip_backward_in || skip_forward_in) music_fsm <= RESET;
                if (end_song_counter >= music_len_in) music_fsm <= END_SONG;
            end
            PAUSE: begin
                if (play_in) music_fsm <= PLAY;
                if (skip_backward_in || skip_forward_in) music_fsm <= RESET;
            end
            END_SONG: begin
                if (skip_backward_in || skip_forward_in) music_fsm <= RESET;
            end
```

```
         endcase
     end
     assign addr_offset = display_sec * 256;
    // calculate rom address and read the location
    assign image_addr = (hcount_in-x_in) + ((vcount_in-y_in) * WIDTH) + addr_offset;
    digits_rom  rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));
    // note the one clock cycle delay in pixel!
    always_ff @ (posedge pixel_clk_in) begin
      if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
           (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
         pixel_out <= {3{image_bits[3:0]}}; // white
         else pixel_out <= 0;
    end
endmodule


module semicolon_0_blob
    #(parameter WIDTH = 256,     // default picture width
                HEIGHT = 240,    // default picture height
                k = 48_828)      // 25_000_000 / 512
    (input wire pixel_clk_in,
     input wire [10:0] x_in,hcount_in,
     input wire [9:0] y_in,vcount_in,
     output logic [11:0] pixel_out);

    ;
    logic [15:0] image_addr;   // num of bits for 256*240 ROM
    logic [7:0] image_bits;

    // calculate rom address and read the location
    assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;

    always_ff @ (posedge pixel_clk_in) begin
         if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
           (vcount_in >= y_in && vcount_in < (y_in+HEIGHT))) begin
             // use MSB 4 bits
                 pixel_out <= {4'b1111, 4'b1111, 4'b1111}; // white
         end else pixel_out <= 0;
    end
endmodule


module semicolon_1_blob
    #(parameter WIDTH = 256,     // default picture width
                HEIGHT = 240,    // default picture height
                k = 48_828)      // 25_000_000 / 512
    (input wire pixel_clk_in,
     input wire [10:0] x_in,hcount_in,
     input wire [9:0] y_in,vcount_in,
     output logic [11:0] pixel_out);

    ;
    logic [15:0] image_addr;   // num of bits for 256*240 ROM
    logic [7:0] image_bits;

    // calculate rom address and read the location
    assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;

    always_ff @ (posedge pixel_clk_in) begin
         if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
           (vcount_in >= y_in && vcount_in < (y_in+HEIGHT))) begin
             // use MSB 4 bits
```

```
                pixel_out <= {4'b1111, 4'b1111, 4'b1111}; // white
        end else pixel_out <= 0;
   end
endmodule


module volume_0_digit_blob
   #(parameter WIDTH = 256,     // default picture width
               HEIGHT = 240,    // default picture height
               k = 48_828)      // 25_000_000 / 512
   (input wire pixel_clk_in,
    input wire [10:0] x_in,hcount_in,
    input wire [9:0] y_in,vcount_in,
    input wire which_vol_in,
    input wire [2:0] shift_in,
    input wire vol_up_in,
    input wire vol_down_in,
    input wire sys_reset_in,
    output logic [11:0] pixel_out);
   ;

    logic [15:0] addr_offset;
    logic [2:0] display_vol; // 0...7
    assign addr_offset = display_vol*256;

    logic [15:0] image_addr;   // num of bits for 256*240 ROM
    logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH + addr_offset;
   digits_rom  rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));

   enum {FULL, THREE_FOURTHS, ONE_HALF,ONE_FOURTH, ZERO} vol_fsm;

    always_ff @ (posedge pixel_clk_in) begin
        if (sys_reset_in) begin
            vol_fsm <= FULL;
        end else begin
            if (~which_vol_in) begin // switch 7 off, switches control volume
                display_vol <= shift_in;
                // use MSB 4 bits
            end else begin // switch 7 on, gestures control volume
                case(vol_fsm)
                    ZERO: begin // display the 0 in 0%
                        display_vol <= 0;
                        if (vol_up_in) vol_fsm <= ONE_FOURTH;
                    end
                    ONE_FOURTH: begin // display the 5 in 25%
                        display_vol <= 5;
                        if (vol_up_in) vol_fsm <= ONE_HALF;
                        if (vol_down_in) vol_fsm <= ZERO;
                    end
                    ONE_HALF: begin // display the 0 in 50%
                        display_vol <= 0;
                        if (vol_up_in) vol_fsm <= THREE_FOURTHS;
                        if (vol_down_in) vol_fsm <= ONE_FOURTH;
                    end
                    THREE_FOURTHS: begin // display the 5 in 75%
                        display_vol <= 5;
                        if (vol_up_in) vol_fsm <= FULL;
                        if (vol_down_in) vol_fsm <= ONE_HALF;
                    end
                    FULL: begin // display the 0 in 100%
                        display_vol <= 0;
                        if (vol_down_in) vol_fsm <= THREE_FOURTHS;
                    end
                endcase
            end
        end
    end

   always_ff @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        pixel_out <= {3{image_bits[3:0]}}; // white
        else pixel_out <= 0;
   end
endmodule


//module volume_1_digit_blob
//   #(parameter WIDTH = 256,     // default picture width
//               HEIGHT = 240,    // default picture height
//               k = 48_828)      // 25_000_000 / 512
//   (input wire pixel_clk_in,
//    input wire [10:0] x_in,hcount_in,
//    input wire [9:0] y_in,vcount_in,
//    input wire which_vol_in,
//    input wire [2:0] shift_in,
//    input wire vol_up_in,
//    input wire vol_down_in,
//    input wire sys_reset_in,
//    output logic [11:0] pixel_out);
//   ;

//    logic [15:0] addr_offset;
//    logic [2:0] display_vol; // 0...7
//    assign addr_offset = display_vol*256;

//    logic [15:0] image_addr;   // num of bits for 256*240 ROM
//    logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

//   // calculate rom address and read the location
//   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH + addr_offset;
//   digits_rom  rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));

//   enum {FULL, THREE_FOURTHS, ONE_HALF, ONE_FOURTH, ZERO} vol_fsm;
//    always_ff @ (posedge pixel_clk_in) begin
//        if (sys_reset_in) begin
//            vol_fsm <= FULL;
//        end else begin
//            if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
//                (vcount_in >= y_in && vcount_in < (y_in+HEIGHT))) begin
//                if (~which_vol_in) begin // switch 7 off, switches control volume
//                    // use MSB 4 bits
//                    pixel_out <= 0; // black
//                end else begin // switch 7 on, gestures control volume
//                    case(vol_fsm)
//                        ZERO: begin // display nothing for 0%
//                            display_vol <= 0;
//                            pixel_out <= 12'b1111_0000_0000;
//                            if (vol_up_in) vol_fsm <= ONE_FOURTH;
//                        end
//                        ONE_FOURTH: begin // display the 2 in 25%
//                              display_vol <= 2;
//                              pixel_out <= {3{image_bits[3:0]}}; // white
//                              if (vol_up_in) vol_fsm <= ONE_HALF;
//                              if (vol_down_in) vol_fsm <= ZERO;
//                          end
//                          ONE_HALF: begin // display the 5 in 50%
//                              display_vol <= 5;
//                              pixel_out <= {3{image_bits[3:0]}}; // white
//                              if (vol_up_in) vol_fsm <= THREE_FOURTHS;
//                              if (vol_down_in) vol_fsm <= ONE_FOURTH;
//                          end
//                          THREE_FOURTHS: begin // display the 7 in 75%
//                              display_vol <= 7;
//                              pixel_out <= {3{image_bits[3:0]}}; // white
//                              if (vol_up_in) vol_fsm <= FULL;
//                              if (vol_down_in) vol_fsm <= ONE_HALF;
//                          end
//                          FULL: begin // display the 0 in 100%
//                              display_vol <= 0;
//                              pixel_out <= {3{image_bits[3:0]}}; // white
//                              if (vol_down_in) vol_fsm <= THREE_FOURTHS;
//                          end
//                      endcase
//                  end
//              end else pixel_out <= 0;
//          end
//      end
//endmodule


//module volume_2_digit_blob
//   #(parameter WIDTH = 256,     // default picture width
//               HEIGHT = 240,    // default picture height
//               k = 48_828)      // 25_000_000 / 512
//   (input wire pixel_clk_in,
//    input wire [10:0] x_in,hcount_in,
//    input wire [9:0] y_in,vcount_in,
//    input wire which_vol_in,
//    input wire [2:0] shift_in,
//    input wire vol_up_in,
//    input wire vol_down_in,
//    input wire sys_reset_in,
//    output logic [11:0] pixel_out);
//   ;

//    logic [15:0] addr_offset;
//    logic [2:0] display_vol; // 0...7
//    assign addr_offset = display_vol*256;

//    logic [15:0] image_addr;   // num of bits for 256*240 ROM
//    logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

//   // calculate rom address and read the location
//   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH + addr_offset;
//   digits_rom  rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));

//   enum {FULL, THREE_FOURTHS, ONE_HALF,ONE_FOURTH, ZERO} vol_fsm;
//    always_ff @ (posedge pixel_clk_in) begin
//        if (sys_reset_in) begin
//            vol_fsm <= FULL;
//        end else begin
//            if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
//                (vcount_in >= y_in && vcount_in < (y_in+HEIGHT))) begin
//                if (~which_vol_in) begin // switch 7 off, switches control volume
//                    pixel_out <= 0; // black
//                end else begin // switch 7 on, gestures control volume
//                    case(vol_fsm)
//                        ZERO: begin // display nothing for 0%
//                            pixel_out <= {3{image_bits[3:0]}}; // black
//                            if (vol_up_in) vol_fsm <= ONE_FOURTH;
//                        end
//                        ONE_FOURTH: begin // display nothing for 25%
//                            pixel_out <= 12'b0000_0000_0000; // black
//                            if (vol_up_in) vol_fsm <= ONE_HALF;
//                            if (vol_down_in) vol_fsm <= ZERO;
//                        end
//                        ONE_HALF: begin // display the 5 in 50%
//                            pixel_out <= 12'b0000_0000_0000; // black
//                            if (vol_up_in) vol_fsm <= THREE_FOURTHS;
//                            if (vol_down_in) vol_fsm <= ONE_FOURTH;
//                        end
//                        THREE_FOURTHS: begin // display the 7 in 75%
//                            pixel_out <= 12'b0000_0000_0000; // black
//                            if (vol_up_in) vol_fsm <= FULL;
//                            if (vol_down_in) vol_fsm <= ONE_HALF;
//                        end
//                        FULL: begin // display the 1 in 100%
//                            display_vol <= 1;
//                            pixel_out <= {3{image_bits[3:0]}}; // white
//                            if (vol_down_in) vol_fsm <= THREE_FOURTHS;
//                        end
//                    endcase
//                end
//            end else pixel_out <= 0;
//        end
//    end
//endmodule


// ALL THE SONGS TITLES
module titanic_blob
   #(parameter WIDTH = 256,     // default picture width
               HEIGHT = 240)    // default picture height
   (input wire pixel_clk_in,
    input wire [10:0] x_in,hcount_in,
    input wire [9:0] y_in,vcount_in,
    output logic [11:0] pixel_out);

   logic [15:0] image_addr;   // num of bits for 256*240 ROM
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
   title_titanic_rom  rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));
   title_titanic_red  rcm (.clka(pixel_clk_in), .addra(image_bits), .douta(red_mapped));
   // note the one clock cycle delay in pixel!
   always_ff @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        // use MSB 4 bits
        pixel_out <= {red_mapped[7:4], red_mapped[7:4], red_mapped[7:4]}; // greyscale
        else pixel_out <= 0;
   end
endmodule
```

```
module slump_blob
   #(parameter WIDTH = 256,     // default picture width
               HEIGHT = 240)    // default picture height
   (input wire pixel_clk_in,
    input wire [10:0] x_in,hcount_in,
    input wire [9:0] y_in,vcount_in,
    output logic [11:0] pixel_out);

   logic [15:0] image_addr;   // num of bits for 256*240 ROM
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
   title_slump_rom  rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));
   title_slump_red  rcm (.clka(pixel_clk_in), .addra(image_bits), .douta(red_mapped));
   // note the one clock cycle delay in pixel!
   always_ff @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        // use MSB 4 bits
        pixel_out <= {red_mapped[7:4], red_mapped[7:4], red_mapped[7:4]}; // greyscale
        else pixel_out <= 0;
   end
endmodule


module end_day_blob
    #(parameter WIDTH = 256,     // default picture width
               HEIGHT = 240)    // default picture height
   (input wire pixel_clk_in,
    input wire [10:0] x_in,hcount_in,
    input wire [9:0] y_in,vcount_in,
    output logic [11:0] pixel_out);

   logic [15:0] image_addr;   // num of bits for 256*240 ROM
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
   end_day_rom  rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));
   title_end_day_red  rcm (.clka(pixel_clk_in), .addra(image_bits), .douta(red_mapped));
   // note the one clock cycle delay in pixel!
   always_ff @ (posedge pixel_clk_in) begin
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        // use MSB 4 bits
        pixel_out <= {red_mapped[7:4], red_mapped[7:4], red_mapped[7:4]}; // greyscale
        else pixel_out <= 0;
   end
endmodule

////////////////////////////////////////////////////////////////////////////////
// Update: 8/8/2019 GH
// Create Date: 10/02/2015 02:05:19 AM
// Module Name: xvga
//
// xvga: Generate VGA display signals (1024 x 768 @ 60Hz)
//
//                              ---- HORIZONTAL -----     ------VERTICAL -----
//                              Active                    Active
//                    Freq      Video   FP  Sync   BP      Video   FP  Sync  BP
//   640x480, 60Hz    25.175    640     16    96   48        480    11   2    31
//   800x600, 60Hz    40.000    800     40   128   88        600     1   4    23
//   1024x768, 60Hz   65.000    1024    24   136  160        768     3   6    29
//   1280x1024, 60Hz  108.00    1280    48   112  248        768     1   3    38
//   1280x720p 60Hz   75.25     1280    72    80  216        720     3   5    30
//   1920x1080 60Hz   148.5     1920    88    44  148       1080     4   5    36
//
// change the clock frequency, front porches, sync's, and back porches to create
// other screen resolutions
////////////////////////////////////////////////////////////////////////////////

module vga(input wire vclock_in,
           output logic [10:0] hcount_out,    // pixel number on current line
           output logic [9:0] vcount_out,     // line number
           output logic vsync_out, hsync_out,
           output logic blank_out);

   parameter DISPLAY_WIDTH  = 640;      // display width
   parameter DISPLAY_HEIGHT = 480;       // number of lines

   parameter  H_FP = 16;                 // horizontal front porch
   parameter  H_SYNC_PULSE = 96;        // horizontal sync
   parameter  H_BP = 48;                // horizontal back porch

   parameter  V_FP = 11;                  // vertical front porch
   parameter  V_SYNC_PULSE = 2;          // vertical sync
   parameter  V_BP = 31;                 // vertical back porch

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   logic hblank,vblank;
   logic hsyncon,hsyncoff,hreset,hblankon;
   assign hblankon = (hcount_out == (DISPLAY_WIDTH -1));
   assign hsyncon = (hcount_out == (DISPLAY_WIDTH + H_FP - 1));  //1047
   assign hsyncoff = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE - 1));  // 1183
   assign hreset = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE + H_BP - 1));  //1343

   // vertical: 806 lines total
   // display 768 lines
   logic vsyncon,vsyncoff,vreset,vblankon;
   assign vblankon = hreset & (vcount_out == (DISPLAY_HEIGHT - 1));   // 767
   assign vsyncon = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP - 1));  // 771
   assign vsyncoff = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP + V_SYNC_PULSE - 1));  //
777
   assign vreset = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP + V_SYNC_PULSE + V_BP - 1));
// 805

   // sync and blanking
   logic next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always_ff @(posedge vclock_in) begin
      hcount_out <= hreset ? 0 : hcount_out + 1;
      hblank <= next_hblank;
      hsync_out <= hsyncon ? 0 : hsyncoff ? 1 : hsync_out;  // active low

      vcount_out <= hreset ? (vreset ? 0 : vcount_out + 1) : vcount_out;
      vblank <= next_vblank;
      vsync_out <= vsyncon ? 0 : vsyncoff ? 1 : vsync_out;  // active low

      blank_out <= next_vblank | (next_hblank & ~hreset);
   end
endmodule


`default_nettype wire
```

// FFT Logic

```
`timescale 1ns / 1ps
`default_nettype none

module fft(
        input wire clk_100mhz,
        input wire byte_available_in,
        input wire [7:0] byte_in,
        input wire [9:0] draw_addr_in,
        output wire [31:0] amp_out
    );

    logic [11:0] adc_data;
    logic [11:0] sampled_adc_data;
    logic adc_ready;
    logic enable;
    logic [7:0] recorder_data;
    logic [7:0] vol_out;
    logic pwm_val; //pwm signal (HI/LO)
    logic [15:0] scaled_adc_data;
    logic [15:0] scaled_signed_adc_data;
    logic [15:0] fft_data;
    logic       fft_ready;
    logic       fft_valid;
    logic       fft_last;
    logic [9:0] fft_data_counter;

    logic fft_out_ready;
    logic fft_out_valid;
    logic fft_out_last;
    logic [31:0] fft_out_data;

    logic sqsum_valid;
    logic sqsum_last;
    logic sqsum_ready;
    logic [31:0] sqsum_data;

    logic fifo_valid;
    logic fifo_last;
    logic fifo_ready;
    logic [31:0] fifo_data;

    logic [23:0] sqrt_data;
    logic sqrt_valid;
    logic sqrt_last;

    // SD, VGA need to use 25 mhz clock
    logic clk_25mhz;
    clk_wiz_0 clocks(.clk_in1(clk_100mhz), .clk_out1(clk_25mhz));
    always_ff @(posedge clk_100mhz)begin
        if (byte_available_in) begin
//            scaled_adc_data <= 16*adc_data;
//            scaled_signed_adc_data <= {~scaled_adc_data[15],scaled_adc_data[14:0]};
//            sampled_adc_data <= {~adc_data[11],adc_data[10:0]}; //convert to signed. incoming
data is offset binary
            if (fft_ready)begin
                fft_data_counter <= fft_data_counter +1;
                fft_last <= fft_data_counter==1023;
                fft_valid <= 1'b1;
//                fft_data <= {~scaled_adc_data[15],scaled_adc_data[14:0]}; //set the FFT DATA
here!
                fft_data <= {~byte_in[7], byte_in[6:0], 8'b0000_0000};
            end
            //https://en.wikipedia.org/wiki/Offset_binary
        end else begin
            fft_data <= 0;
            fft_last <= 0;
            fft_valid <= 0;
        end
    end


    //FFT module:
    //CONFIGURATION:
    //1 channel
    //transform length: 1024
    //target clock frequency: 100 MHz
    //target Data throughput: 50 Msps
    //Auto-select architecture
    //IMPLEMENTATION:
    //Fixed Point, Scaled, Truncation
    //MAKE SURE TO SET NATURAL ORDER FOR OUTPUT ORDERING
    //Input Data Width, Phase Factor Width: Both 16 bits
    //Result uses 12 DSP48 Slices and 6 Block RAMs (under Impl Details)
    xfft_0 my_fft (.aclk(clk_100mhz), .s_axis_data_tdata(fft_data),
                    .s_axis_data_tvalid(fft_valid),
                    .s_axis_data_tlast(fft_last), .s_axis_data_tready(fft_ready),
                    .s_axis_config_tdata(0),
                     .s_axis_config_tvalid(0),
                     .s_axis_config_tready(),
                    .m_axis_data_tdata(fft_out_data), .m_axis_data_tvalid(fft_out_valid),
                    .m_axis_data_tlast(fft_out_last), .m_axis_data_tready(fft_out_ready));

    //for debugging commented out, make this whatever size,detail you want:
    //ila_0 myila (.clk(clk_100mhz), .probe0(fifo_data), .probe1(sqrt_data), .probe2(sqsum_data),
.probe3(fft_out_data));

    //custom module (was written with a Vivado AXI-Streaming Wizard so format looks inhuman
    //this is because it was a template I customized.
    square_and_sum_v1_0 mysq(.s00_axis_aclk(clk_100mhz), .s00_axis_aresetn(1'b1),
                            .s00_axis_tready(fft_out_ready),
                            .s00_axis_tdata(fft_out_data),.s00_axis_tlast(fft_out_last),
                            .s00_axis_tvalid(fft_out_valid),.m00_axis_aclk(clk_100mhz),
                            .m00_axis_aresetn(1'b1),. m00_axis_tvalid(sqsum_valid),
                            .m00_axis_tdata(sqsum_data),.m00_axis_tlast(sqsum_last),
                            .m00_axis_tready(sqsum_ready));
    //Didn't really need this fifo but put it in for because I felt like it and for practice:
    //This is an AXI4-Stream Data FIFO
    //FIFO Depth: 1024
    //No packet mode, no async clock, 2 sycn stages for clock domain crossing // CAN'T CHANGE
FROM 3
    //no aclken conversion
    //TDATA Width: 4 bytes
    //Enable TSTRB: No...isn't needed
    //Enable TKEEP: No...isn't needed
    //Enable TLAST: Yes...use this for frame alignment
    //TID Width, TDEST Width, and TUSER width: all 0
    axis_data_fifo_0 myfifo (.s_axis_aclk(clk_100mhz), .s_axis_aresetn(1'b1),
                             .s_axis_tvalid(sqsum_valid), .s_axis_tready(sqsum_ready),
                             .s_axis_tdata(sqsum_data), .s_axis_tlast(sqsum_last),
                             .m_axis_tvalid(fifo_valid), .m_axis_tdata(fifo_data),
                             .m_axis_tready(fifo_ready), .m_axis_tlast(fifo_last));


    //AXI4-STREAMING Square Root Calculator:
    //CONFIGUATION OPTIONS:
    // Functional Selection: Square Root
    //Architec Config: Parallel (can't change anyways)
    //Pipelining: Max
    //Data Format: UnsignedInteger
    //Phase Format: Radians, the way God intended.
    //Input Width: 32
    //Output Width: 17 // CAN'T SET THIS PARAM
    //Round Mode: Truncate
    //0 on the others, and no scale compensation
    //AXI4 STREAM OPTIONS:
    //Has TLAST!!! need to propagate that
    //Don't need a TUSER
    //Flow Control: Blocking
    //optimize Goal: Performance
    //leave other things unchecked.
    cordic_0 mysqrt (.aclk(clk_100mhz), .s_axis_cartesian_tdata(fifo_data),
                     .s_axis_cartesian_tvalid(fifo_valid), .s_axis_cartesian_tlast(fifo_last),
                     .s_axis_cartesian_tready(fifo_ready),.m_axis_dout_tdata(sqrt_data),
                     .m_axis_dout_tvalid(sqrt_valid), .m_axis_dout_tlast(sqrt_last));

    logic [9:0] addr_count;
    always_ff @(posedge clk_100mhz)begin
        if (sqrt_valid)begin
            if (sqrt_last)begin
                addr_count <= 'd1023; //allign
            end else begin
                addr_count <= addr_count + 1'b1;
            end
        end

    end

    //Two Port BRAM: The FFT pipeline files values inot this and the VGA side of things
    //reads the values out as needed!  Separate clocks on both sides so we don't need to
    //worry about clock domain crossing!! (at least not directly)
    //BRAM Generator (v. 8.4)
    //BASIC:
    //Interface Type: Native
    //Memory Type: True Dual Port RAM (leave common clock unticked...since using100 and 65 MHz)
    //leave ECC as is
    //leave Write enable as is (unchecked Byte Write Enabe)
    //Algorithm Options: Minimum Area (not too important anyways)
    //PORT A OPTIONS:
    //Write Width: 32
    //Read Width: 32
    //Write Depth: 1024
    //Read Depth: 1024
    //Operating Mode; Write First (not too important here)
    //Enable Port Type: Use ENA Pin
    //Keep Primitives Output Register checked
    //leave other stuff unchecked
    //PORT B OPTIONS:
    //Should mimic Port A (and should auto-inheret most anyways)
    //leave other tabs as is. the summary tab should report one 36K BRAM being used
    value_bram mvb (.addra(addr_count+3), .clka(clk_100mhz), .dina({8'b0,sqrt_data}),
                    .douta(), .ena(1'b1), .wea(sqrt_valid),.dinb(0),
                    .addrb(draw_addr_in), .clkb(clk_25mhz), .doutb(amp_out),
                    .web(1'b0), .enb(1'b1));



endmodule


module square_and_sum_v1_0 #
    (
        // Users to add parameters here

        // User parameters ends
        // Do not modify the parameters beyond this line


        // Parameters of Axi Slave Bus Interface S00_AXIS
        parameter integer C_S00_AXIS_TDATA_WIDTH    = 32,

        // Parameters of Axi Master Bus Interface M00_AXIS
        parameter integer C_M00_AXIS_TDATA_WIDTH    = 32,
        parameter integer C_M00_AXIS_START_COUNT    = 32
    )
    (
        // Users to add ports here

        // User ports ends
        // Do not modify the ports beyond this line


        // Ports of Axi Slave Bus Interface S00_AXIS
        input wire  s00_axis_aclk,
        input wire  s00_axis_aresetn,
        output wire  s00_axis_tready,
        input wire [C_S00_AXIS_TDATA_WIDTH-1 : 0] s00_axis_tdata,
        input wire  s00_axis_tlast,
        input wire  s00_axis_tvalid,

        // Ports of Axi Master Bus Interface M00_AXIS
        input wire  m00_axis_aclk,
        input wire  m00_axis_aresetn,
        output wire  m00_axis_tvalid,
        output wire [C_M00_AXIS_TDATA_WIDTH-1 : 0] m00_axis_tdata,
        output wire  m00_axis_tlast,
        input wire  m00_axis_tready
    );

    reg m00_axis_tvalid_reg_pre;
    reg m00_axis_tlast_reg_pre;
    reg m00_axis_tvalid_reg;
    reg m00_axis_tlast_reg;
    reg [C_M00_AXIS_TDATA_WIDTH-1 : 0] m00_axis_tdata_reg;

    reg s00_axis_tready_reg;
    reg signed [31:0] real_square;
    reg signed [31:0] imag_square;

    wire signed [15:0] real_in;
    wire signed [15:0] imag_in;
    assign real_in = s00_axis_tdata[31:16];
    assign imag_in = s00_axis_tdata[15:0];

    assign m00_axis_tvalid = m00_axis_tvalid_reg;
    assign m00_axis_tlast = m00_axis_tlast_reg;
    assign m00_axis_tdata = m00_axis_tdata_reg;
    assign s00_axis_tready = s00_axis_tready_reg;

    always @(posedge s00_axis_aclk)begin
        if (s00_axis_aresetn==0)begin
            s00_axis_tready_reg <= 0;
        end else begin
            s00_axis_tready_reg <= m00_axis_tready; //if what you're feeding data to is ready,
then you're ready.
        end
    end

    always @(posedge m00_axis_aclk)begin
        if (m00_axis_aresetn==0)begin
            m00_axis_tvalid_reg <= 0;
            m00_axis_tlast_reg <= 0;
            m00_axis_tdata_reg <= 0;
        end else begin
            m00_axis_tvalid_reg_pre <= s00_axis_tvalid; //when new data is coming, you've got new
data to put out
            m00_axis_tlast_reg_pre <= s00_axis_tlast; //
            real_square <= real_in*real_in;
            imag_square <= imag_in*imag_in;

            m00_axis_tvalid_reg <= m00_axis_tvalid_reg_pre; //when new data is coming, you've got
new data to put out
            m00_axis_tlast_reg <= m00_axis_tlast_reg_pre; //
            m00_axis_tdata_reg <= real_square + imag_square;
        end
    end
endmodule
```

```
`default_nettype wire
```