Virtual Piano

6.111 Final Project

Lisa Liu
Sheldon Trotman

December 11, 2013

Abstract

We aim to making piano practice more portable than before. The Virtual Piano aims to imitate the experience of playing a piano with a camera, a 6.111 labkit, and a set of color tags. The camera tracks fingers which have been tagged with colored pieces of paper. We use the images to determine whether a gesture counts as a piano note being played. Once we determine a note was indeed played, we play the proper note at the proper volume, and display it on the screen. The Virtual Piano aims to recreate the experience as much as possible by adjusting the volume and decay of the sound.

I. Overview

Who says you need a piano or keyboard to play piano? For our final project, we plan to play and display music based on hand gestures that imitate piano playing. The project can be divided into three parts: video capture and processing, sound processing, and display.

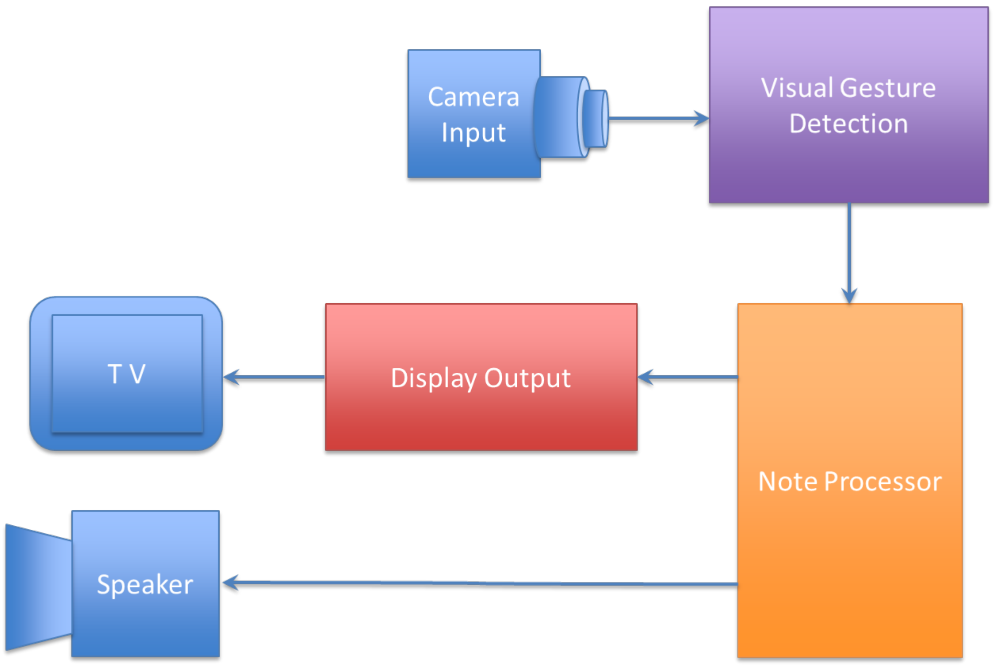

Figure 1 General Block Diagram: Graphical explanation of project design. We take in a camera input and after visual and logic processing we output display and audio data.

The Visual Gesture Detection Module takes an input from the camera. We have Verilog code to detect a gesture by analyzing frames, store them in metadata storage, and compare it to the next frame in order to determine the change in position of a finger. Individual fingers on a single hand are tagged with colors to aid tracking and detection. A spatial analyzer will determine if a change in position represents a pressed key. It will output the name of the note pressed (note_name) and how quickly that not was pressed (attack) to the Note Processor.

At the most basic level, the Virtual Piano replays recorded piano notes. The sound data for the piano notes are stored in BRAM and are recalled and played when needed. In response to information about the speed of the finger, the volume of the note will increase or decrease. We may adjust the attack and decay of the note as well.

Lastly, we would like to display the notes we play. An outline of piano keys is displayed on the screen. When a virtual key is pressed by the user, the corresponding key on the display will light up. The display also shows the last eight notes played on a bar staff. The notes should accurately show the note name and note length.

Given a note name, the Note Processor can play that note directly, adjusting the volume and decay for attack. However, it can also output the ticking of a metronome.

II. Visual Gesture Detection

The Visual Gesture Detection module was responsible for using a camera input and recognizing the user's desired actions and sending these actions to the Note Processor. Input would pass through NTSC to RGB conversion. Then to a center of mass detector then this would be fed to a module that maps the camera input coordinates coordinates to on screen coordinates because the dimensions of the two were not the same.

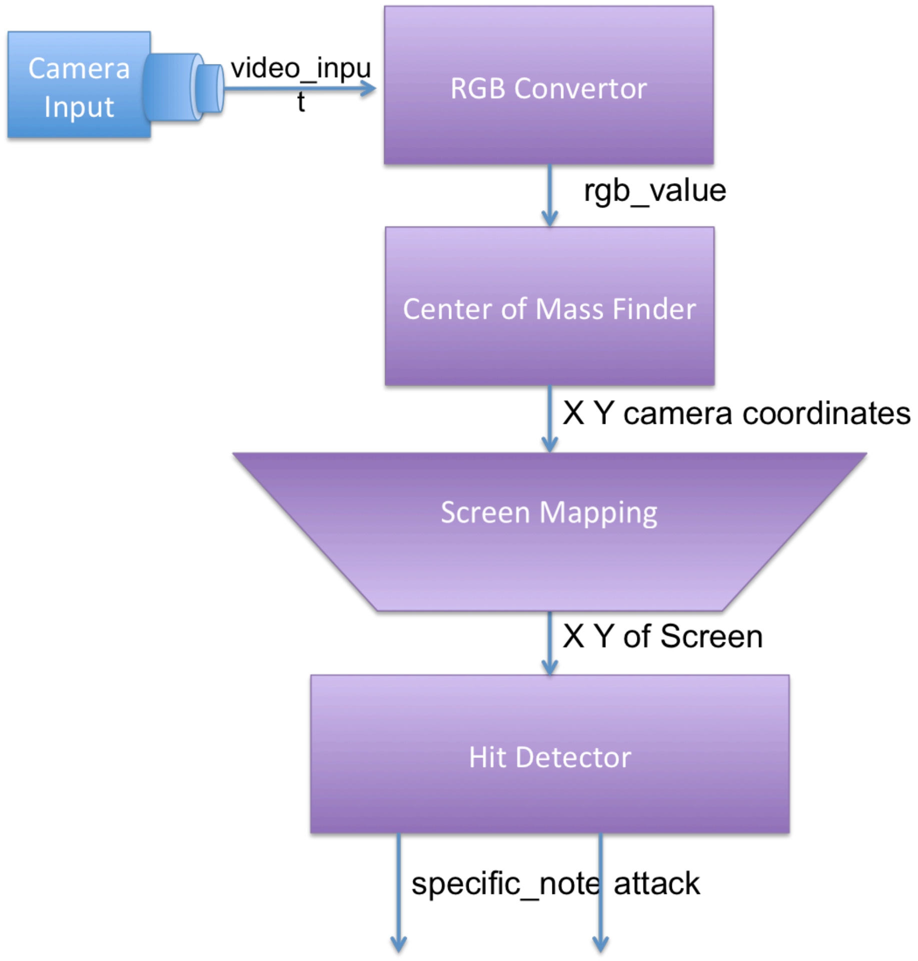

Figure 2 Visual Gesture Detection Breakdown. The specific sub-modules of the Visual Gesture detection module

The design was split between receiving the camera data and recognizing the fingers and then mapping the center of mass of each finger to a visual on the screen. Primarily because the camera was NTSC, there had to be some conversion to RGB values for each pixel. Secondly after this was found, certain groupings of pixels were fed to specific "Hit Detectors".

Rgb Convertor

Input: video input pixel
Ouput: RGB value

Takes NTSC input and converts it to RGB values for each pixel

Center of Mass Finder

Input: RGB value and x y value of that pixel
Output: X Y Coordinates

First detects if the RGB value is within a certain range and then accumulates how many of that specific color has been found and does a continuous average.

Screen Mapping

Input: X Y coordinates of camera
Output: X Y coordinate of screen

Converts the 479x680 camera input to 1023x767.

Hit Detector

Input: X Y coordinates
Output: Attack and specific note

Receives coordinates and detects if the Y coordinate is below a certain threshold specified as a note hit. The note and the attack is passed to the note processor.

Detecting multiple objects on the screen has been a long standing problem for visual detection but this module was success in doing so. The input from the screen was split into partitions in which specific fingers were detected. Instead of trying to use the full input, four different detection modules were created to watch a specific quarter of the screen. As shown below, there are the main quarter is shown and the threshold region is specified.

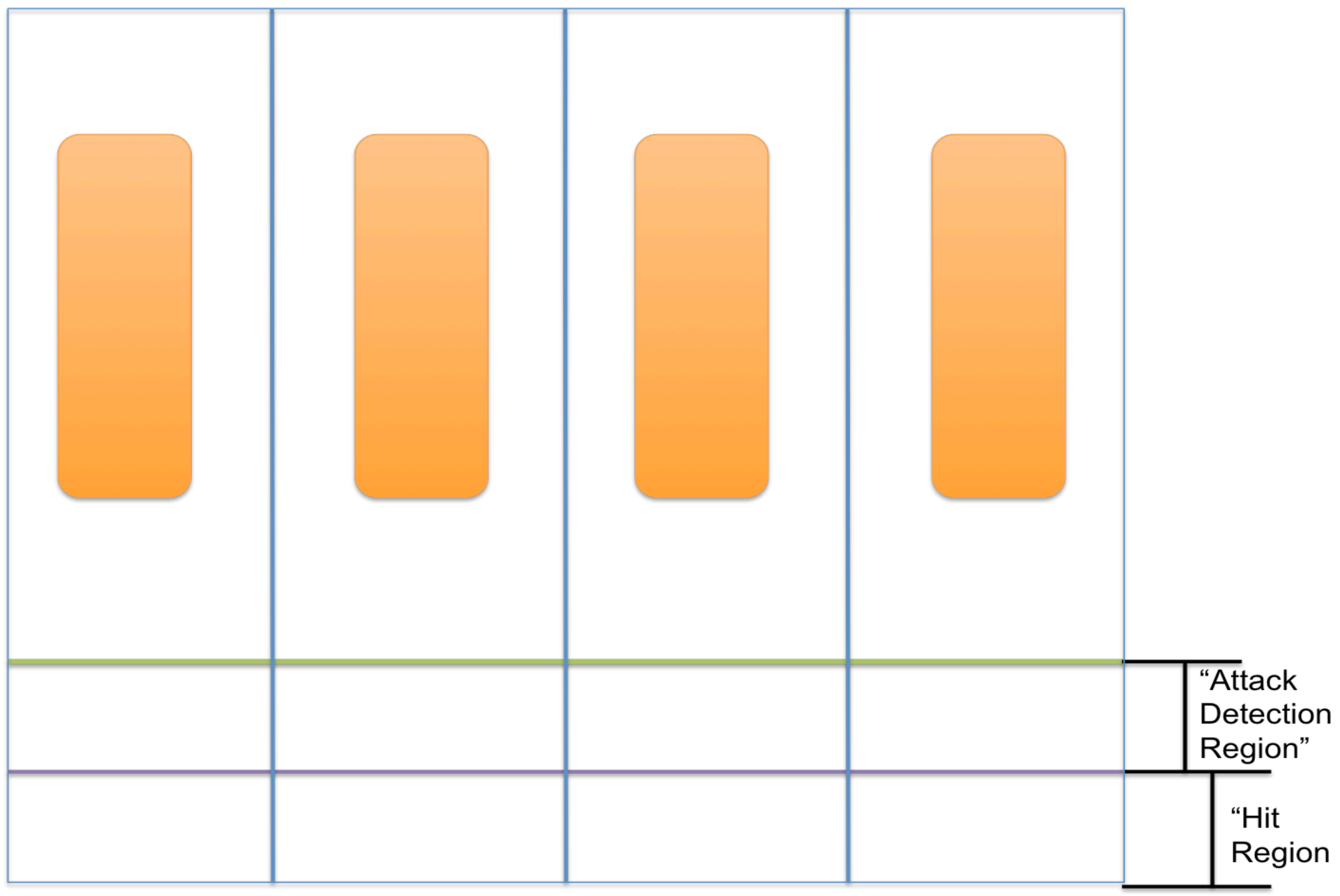

Figure 3 Screen Partitioning. The screen is split into subsections to make detection within the specified quarter of the screen easier. The orange is an example of a finger within the region. The green is where a timer starts and the purple is where it ends to calculate speed through this region for attack calculations. The purple also functions to declare when a key is hit.

For each detection module, once a center of mass is detected, and the bottom of the center of mass has crossed a threshold, a "hit" is sent for that key. Also what is show is an "attack detection" region in which there is a measurement of how long it takes the bottom of the center of mass to cross from the start of the region into the "hit" region.

III. Note Processor

The note processor module is responsible for the metronome and sound of the Virtual Piano. The metronome is used to help the user maintain a tempo by outputting a periodic tick sound. It also times the length of the notes played. The tempo of the metronome is set by switches on the labkit.

Sound data is sent to the ac97 chip in response to inputs from the gesture detection module. The sound adjusts to the note played, the volume set by user input, and how quickly a note is hit.

Metronome

inputs: labkit switches

output: metronome_tick_enable, metronome_sound_enable, note_length

The metronome can help the user keep track of the tempo by playing a periodic tick. Using the labkit switches, the tempo can be set to 16 different values, between 60 beats per minute to 120 beats per minute.

The metronome is also used to measure the length of a note played. However, some notes are a fraction of a beat, so we must be able to subdivide beats. The shortest note we aim to measure is an eighth note, which is half of one beat. Therefore, we make a signal called metronome_tick_enable, which is enabled every quarter of a beat for one clock cycle. Therefore, an eighth note would be 2 metronome ticks.

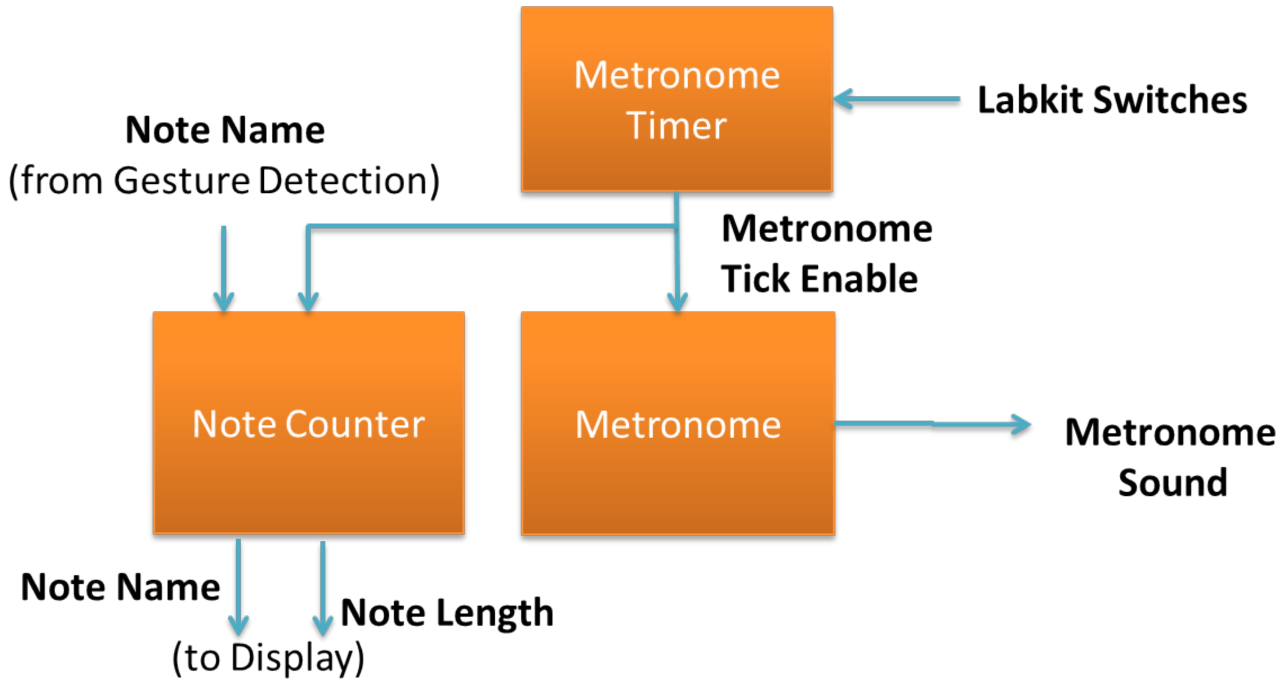

Figure 3 Metronome submodules. The labkit switches set the tempo in metronome timer. Metronome converts the metronome tick enables to moments when the metronome sound is enabled. The metronome ticks are also fed into a note counter which tracks how long each note lasts and sends that data to the display.

<u>metronome_timer</u>

We generate the metronome_tick_enable signal in the metronome_timer module, which is a variation on the divider from the Car Alarm divider. The module counts clock cycles until it reaches a maximum number, which is set by the labkit switches. The maximum number is calculated by

```
assign MAX = (CLK_CYCLES – CLK_CYCLES*SWITCHES[3:0]/32)/4
```

<u>metronome</u>

The metronome_tick_enable signal is also sent to the metronome module, which determines when a metronome sound actually has to play. The user doesn't want the metronome to be on all the time, so we use switch[7] to toggle the metronome. If the metronome is on, after every fourth high metronome_tick_enable, the metronome_sound_enable is set to high for one clock cycle.

<u>note_counter</u>

The note_counter module also receives the metronome_tick_enable signal, in addition to note_name. Th module uses the tick enables to determine the length of a note. We assume that the user isn't able to perfectly time his notes, so the note_counter module will assign the note length based on number of metronome ticks according to table 1. The note counter will continue to update the note_length register after the end of a note, which is detected when note_name changes from one non-undefined note to another value.

The counter is fed back up to the labkit file and output on the 3rd hex display LED, in order to help with debugging and ensure that the expected note is displayed.

Note ID	Note Length Parameter	Metronome Ticks	Picture
0	EIGHTH	1-2	
1	QUARTER	3-5	
2	HALF	6-9	
3	THREE	10-13	
4	WHOLE	14-20	

5	UNDEF	else	n/a

Table 1. Note parameters. Each length of note is assigned an ID number. A note will be assigned an ID number based on the number of metronome ticks that pass while it plays. We show how each note length will be represented on the display.

Sound Transformer

Inputs: attack, note_name, metronome_tick_enable, metronome_sound_enable
Outputs: to_ac97_data

The sound transformer determines which note to play and how to play it. In addition, it plays a metronome sound when the metronome module sends a high on metronome_sound_enable. We are able to use lab5audio from the Voice Recorder lab to handle the data going to the ac97 chip. All we need to do is determine what PCM data to send, and what volume we want to send it at.

Ideally, we receive note_name and attack from the Visual Gesture Detection module, which tells the sound transformer module which note to play and how to play it. However, we mapped each of the 7 notes to buttons on the labkit for testing purposes.

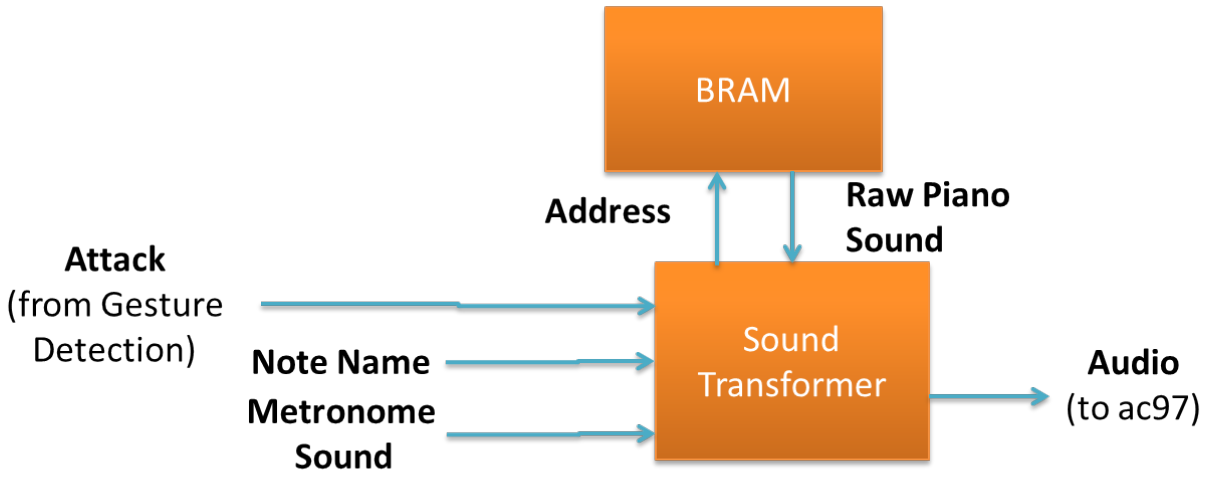

Figure 4 The sound transformer received the note name from the Visual Gesture Detection module. It called for the data for that note stored in BRAM and would transform it based on the attack. It would combine the metronome sound, if needed.

Sound storage

Each piano note lasts about 1-2 seconds. The metronome is even shorter. If we down-sample to 6 kHz, we have enough space in BRAM, which contains 2,592 kb of memory. Each audio sample requires 8 bits of data. 2 seconds of audio corresponds to 12000 samples, or 12000 * 8 bits is 96 kb per note. For 8 audio samples, this represents a maximum of 584 kb. These 8 notes fit well within the BRAM memory constraint, but only when down-sampled.

To load the files onto BRAM, had to down-sample the audio file, convert the down-sampled file into a coe file, and load the coe file into BRAM. To down-sample the wav files, we used MATLAB code provided by Dylan Sherry from a past 6.111 project. We then used Yuta Kuboyama's MATLAB code from a past 6.111 project to convert that file to a coe file. For each audio file, we used ISE's CoreGen & Architecture to create the BRAM and load the coe file into memory. The write width was 8, and the write depth was about 9000 for most of the notes.

8 separate BRAM modules were created, one for each note.
<u>Addressing and looping</u>

All the piano note BRAM modules were fed the same address, but output different audio data. We used a case statement to decide which audio data we wanted to send to the filter and ac97 chip. This limits us to playing one note at a time. If we wanted to play multiple, perhaps we would have used a different address variable for each note.

Each note piano note had a different number of addresses, since each note lasted for a slightly different amount of time. However, we would like to play a note for as long as a note is held down. We did this by looping back to an earlier part of the note. However, some fiddling was required, since the piano note had its own natural decay, if we looped back too far, it would be weird, short cycles when the piano sound would alternate between loud and quiet. If we don't cycle back far enough, then the noise would sound somewhat like static. When we pick the note we want to output, we also had to reset the variables max_note_addr. When we looped back, we jump 100 addresses back from the maximum note address.

While a note is played, the addresses loops around the maximum note address. Once it was done playing, the note address is reset to 0, so that the next new note can start from the beginning of the sound.

However, the metronome addresses had to be handled differently, since the metronome might play a couple times per note, we used a separate address variable for the metronome. First, we needed a new variable to determine whether we wanted to play a metronome tick on the next ac97 sample, since sampling may not occur on the same clock cycle as metronome_sound_enable, but metronome_sound_enable only lasts one clock cycle. When we determine we play a new metronome sound, we initialize the address to 0 and increment it until we hit the end. We do not reset the address to zero until we metronome_sound_enable is high again.

Filter

Since our sound data was downsampled to 6 kHz, but the ac97 is designed to play at 48 kHz, we needed to zero-expand our data samples and low-pass filter the audio. We implemented the same filter used in the Voice Recorder lab.

Attack and Decay

The sound of the notes adjusts to the amount of attack the note is hit with. Ideally, the Visual Gesture Detection system gives us a number between 0 and 4 to tell us how hard the note was hit. A 0 represents a soft hit, a 4 represents a hard hit. However, the attack variable was also set by labkit switch[5:4], for debugging purposes.

At the beginning of each note, we reset the volume to base_volume + 2 * attack and let the volume decrease each time we received a high from metronome_tick_enable. We use the volume input to the lab 5 audio module to control the decay. The user can change the base volume by using the up and down switches on the labkit.

IV. Graphics

inputs: note_name, note_length, attack

output: vga signal

The graphics module output a vga signal based on the note being played. It drew a keyboard to show the current note being played and a bar staff to show the last 8 notes being played.

It receives note_name directly from the Visual Gesture Detection module or from the labkit buttons. It receives note_length from the note processor module, which outputs its data one clock cycle after the note was done playing. Therefore, we put note_name in a register for one clock cycle in order to correctly associate it with the right note length.

Since all the shapes were fairly geometric, we are able to specify conditions for drawing, rather than use an image file. We use separate modules to determine pixel assignments for the keyboard, bar staff, and note. The final pixel assignment is created by adding all three of those together.

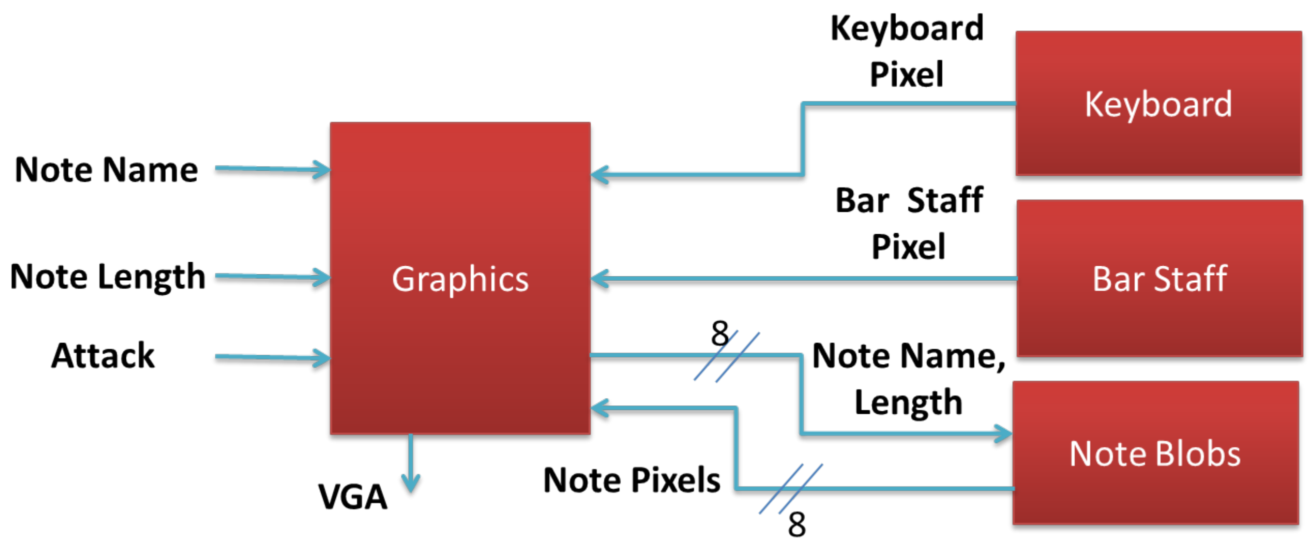

Figure 5. The graphics module draws a bar staff and a keyboard. It is fed note the note name, note length, and attack. From this it shows the current not playing by outputting the proper keyboard pixel. The graphics module also tells the note blob module what note it wants to show, and the note blob sends the graphics module the proper pixel assignments.

<u>Keyboard</u>

A 7-note keyboard was displayed on the bottom half of the screen. The outline of the keyboard was colored in white. The keyboard module also received the current note being played, so it could color in that note. Else all other pixels were black. This module outputs the keyboard pixel assignment.

<u>Bar Staff</u>

This module only needed to create five, evenly spaced horizontal white lines. This module outputs the staff pixel assignment.

<u>Note Blobs</u>

The notes were fairly geometric, so it was easy to specify conditions for coloring them in. For example, the whole note was the area between a large ellipse and a smaller ellipse. A quarter note was an ellipse, vertical line, and diagonal line with a slope dy/dx = 2 attached to the top of the vertical line.

The x and y location of the note blobs were calculated from the information about the note. The y location is determined by which note was played, allowing us to map the note to a specific line or space on the bar staff. The x location was determined by how many notes had been displayed on the screen. Each new note would have the x value increased in order to display the note to the right of the previous note, to show the order in which notes have been played.

Note Data

We want to display 8 notes, so we need to store data for the 8 most recent notes. A counter keeps track of the current note we're writing. When the counter reaches 8, it restarts the counter and clears all the note data.

When a note has finished, we store all the information needed to play the note back later. We store whether the note exists, the name of the note, the length of the note, and the attack. Only the first three pieces of data are needed for display. The last would be used if we implemented a playback feature, and wanted to hear the note exactly as we played it the first time.

The note data is fed into the note blob module to determine the note pixel assignment.

V. Conclusion

In the end we were unable to integrate the visual detection part with the audio and sound display. However, we were able to implement some of the basic features such as vision detection, velocity measurements, audio transformed by attack, and display which reflected the name and length of notes.

This is in part because our systems were designed with slightly different assumptions. The Visual Gesture Detection module was designed to play four notes, whereas the Sound Transformer and Display was designed to play up to 7. Furthermore the Visual Gesture Detection module can detect multiple notes at the same time. In order to integrate, further work must be done to connect the outputs (the Boolean if a certain note has been hit, and the attack value relating to that note) of the Visual Gesture Detection module to the Note Processor. Although we were not able to connect the two pieces, separately, each were able to hit the desired requirements.

VI. Appendix – Code

```
module labkit (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
               ac97_bit_clock,

               vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
               vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
               vga_out_vsync,

               tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
               tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
               tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

               tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
               tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
               tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
               tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

               ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
               ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

               ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
               ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

               clock_feedback_out, clock_feedback_in,

               flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
               flash_reset_b, flash_sts, flash_byte_b,

               rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

               mouse_clock, mouse_data, keyboard_clock, keyboard_data,

               clock_27mhz, clock1, clock2,

               disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
               disp_reset_b, disp_data_in,

               button0, button1, button2, button3, button_enter, button_right,
               button_left, button_down, button_up,

               switch,

               led,

               user1, user2, user3, user4,

               daughtercard,

               systemace_data, systemace_address, systemace_ce_b,
               systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

               analyzer1_data, analyzer1_clock,
               analyzer2_data, analyzer2_clock,
               analyzer3_data, analyzer3_clock,
               analyzer4_data, analyzer4_clock);

output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
input  ac97_bit_clock, ac97_sdata_in;

output [7:0] vga_out_red, vga_out_green, vga_out_blue;
output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
           vga_out_hsync, vga_out_vsync;

output [9:0] tv_out_ycrcb;
output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
           tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
           tv_out_subcar_reset;

input  [19:0] tv_in_ycrcb;
input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef,
           tv_in_hff, tv_in_aff;
output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
           tv_in_reset_b, tv_in_clock;
inout  tv_in_i2c_data;

inout  [35:0] ram0_data;
output [18:0] ram0_address;
output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b;
output [3:0] ram0_bwe_b;

inout  [35:0] ram1_data;
output [18:0] ram1_address;
output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b;
output [3:0] ram1_bwe_b;

input  clock_feedback_in;
output clock_feedback_out;

inout  [15:0] flash_data;
output [23:0] flash_address;
output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
input  flash_sts;

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
input  disp_data_in;
output  disp_data_out;

input  button0, button1, button2, button3, button_enter, button_right,
           button_left, button_down, button_up;
input  [7:0] switch;
```

```
output [7:0] led;

inout [31:0] user1, user2, user3, user4;

inout [43:0] daughtercard;

inout  [15:0] systemace_data;
output [6:0]  systemace_address;
output systemace_ce_b, systemace_we_b, systemace_oe_b;
input  systemace_irq, systemace_mpbrdy;

output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
                         analyzer4_data;
output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;

////////////////////////////////////////////////////////////////////////////
//
// I/O Assignments
//
////////////////////////////////////////////////////////////////////////////

// Audio Input and Output
assign beep= 1'b0;
          //////////// the following 3 lines are commented out so we can use ac97
//assign audio_reset_b = 1'b0;
//assign ac97_synch = 1'b0;
//assign ac97_sdata_out = 1'b1;
// ac97_sdata_in is an input

// VGA Output
          /*
assign vga_out_red = 8'h0;
assign vga_out_green = 8'h0;
assign vga_out_blue = 8'h0;
assign vga_out_sync_b = 1'b1;
assign vga_out_blank_b = 1'b1;
assign vga_out_pixel_clock = 1'b0;
assign vga_out_hsync = 1'b0;
assign vga_out_vsync = 1'b0;
          */

// Video Output
assign tv_out_ycrcb = 10'h0;
assign tv_out_reset_b = 1'b0;
assign tv_out_clock = 1'b0;
assign tv_out_i2c_clock = 1'b0;
assign tv_out_i2c_data = 1'b0;
assign tv_out_pal_ntsc = 1'b0;
assign tv_out_hsync_b = 1'b1;
assign tv_out_vsync_b = 1'b1;
assign tv_out_blank_b = 1'b1;
assign tv_out_subcar_reset = 1'b0;

// Video Input
assign tv_in_i2c_clock = 1'b0;
assign tv_in_fifo_read = 1'b0;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b0;
assign tv_in_reset_b = 1'b0;
assign tv_in_clock = 1'b0;
assign tv_in_i2c_data = 1'bZ;
// tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
// tv_in_aef, tv_in_hff, and tv_in_aff are inputs

// SRAMs
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1;
assign ram0_ce_b = 1'b1;
assign ram0_oe_b = 1'b1;
assign ram0_we_b = 1'b1;
assign ram0_bwe_b = 4'hF;
assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
assign ram1_adv_ld = 1'b0;
assign ram1_clk = 1'b0;
assign ram1_cen_b = 1'b1;
assign ram1_ce_b = 1'b1;
assign ram1_oe_b = 1'b1;
assign ram1_we_b = 1'b1;
assign ram1_bwe_b = 4'hF;
assign clock_feedback_out = 1'b0;
// clock_feedback_in is an input

// RS-232 Interface
assign rs232_txd = 1'b1;
assign rs232_rts = 1'b1;
// rs232_rxd and rs232_cts are inputs

// PS/2 Ports
// mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

// LED Displays
/*assign disp_blank = 1'b1;
assign disp_clock = 1'b0;
assign disp_rs = 1'b0;
assign disp_ce_b = 1'b1;
assign disp_reset_b = 1'b0;
assign disp_data_out = 1'b0;*/
// disp_data_in is an input

// Buttons, Switches, and Individual LEDs
//assign led = 8'hFF;
// button0, button1, button2, button3, button_enter, button_right,
// button_left, button_down, button_up, and switches are inputs

// User I/Os
assign user1 = 32'hZ;
```

```
  assign user2 = 32'hZ;
  assign user3 = 32'hZ;
  assign user4 = 32'hZ;

            // the following are assigned as part of the instantiation of the flash_manager module
  assign flash_data = 16'hZ;                              //direct passthrough from labkit to low-level
                                                                                              //
modules (flash_int and test_fsm)
  assign flash_address = 24'h0;
  assign flash_ce_b = 1'b1;
  assign flash_oe_b = 1'b1;
  assign flash_we_b = 1'b1;
  assign flash_reset_b = 1'b0;
  assign flash_byte_b = 1'b1;
  // flash_sts is an input

  // Daughtercard Connectors
  assign daughtercard = 44'hZ;

  // SystemACE Microprocessor Port
  assign systemace_data = 16'hZ;
  assign systemace_address = 7'h0;
  assign systemace_ce_b = 1'b1;
  assign systemace_we_b = 1'b1;
  assign systemace_oe_b = 1'b1;
  // systemace_irq and systemace_mpbrdy are inputs

  // Logic Analyzer
  assign analyzer1_data = 16'h0;
  assign analyzer1_clock = 1'b1;
  assign analyzer2_data = 16'h0;
  assign analyzer2_clock = 1'b1;
  //assign analyzer3_data = 16'h0;
  //assign analyzer3_clock = 1'b1;
  assign analyzer4_data = 16'h0;
  assign analyzer4_clock = 1'b1;

////////////////////////////////////////////////////////////////////////////
 //
 // Reset Generation
 //
 // A shift register primitive is used to generate an active-high reset
 // signal that remains high for 16 clock cycles after configuration finishes
 // and the FPGA's internal clocks begin toggling.
 //
 ////////////////////////////////////////////////////////////////////////////
 wire reset;
 SRL16 reset_sr(.D(1'b0), .CLK(clock_27mhz), .Q(reset),
                 .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
 defparam reset_sr.INIT = 16'hFFFF;

/////////////////////////////////////////////////////
            // PARAMETERS
            parameter SOUND_ADDR_SIZE = 24;
            parameter SOUND_DATA_SIZE = 16;
		parameter NONE = 0;			// no note played
		parameter A = 1;
		parameter B = 2;
		parameter C = 3;
		parameter D = 4;
		parameter E = 5;
		parameter F = 6;
		parameter G = 7;

//////////////////////////////////////////////////

//////////////////////////////////////////////////
//		VARIABLES
//////////////////////////////////////////////////
		wire [3:0] tempo_switches;
		assign tempo_switches = switch[3:0];
			// pick screen mode
		wire screen_mode;
		assign screen_mode = switch[6];

		wire metronome_on;
		assign metronome_on = switch[7];		// use switch[7] to toggle metronome

		wire [1:0] attack;
		assign attack = switch[5:4];

		wire clock;
		assign clock = clock_27mhz;

		wire [2:0] note_length;
		reg [2:0] note_name;
		wire [63:0] disp_data;		// hex display


//////////////////////////////////////////////////
//			Pick Note
/////////////////////////////////////////////////

		wire a_but_left, b_but_right, c_but_enter,
					d_but_3, e_but_2, f_but_1, g_but_0;
		debounce but_a(.reset(reset),.clock(clock),.noisy(~button_left),.clean(a_but_left));
		debounce but_b(.reset(reset),.clock(clock),.noisy(~button_right),.clean(b_but_right));
		debounce but_c(.reset(reset),.clock(clock),.noisy(~button_enter),.clean(c_but_enter));
		debounce but_d(.reset(reset),.clock(clock),.noisy(~button3),.clean(d_but_3));
		debounce but_e(.reset(reset),.clock(clock),.noisy(~button2),.clean(e_but_2));
		debounce but_f(.reset(reset),.clock(clock),.noisy(~button1),.clean(f_but_1));
		debounce but_g(.reset(reset),.clock(clock),.noisy(~button0),.clean(g_but_0));

		always @(posedge clock) begin
			if (a_but_left) note_name <= A;
			else if (b_but_right) note_name <= B;
			else if (c_but_enter) note_name <= C;
			else if (d_but_3) note_name <=D;
			else if (e_but_2) note_name <= E;
```

```
                else if (f_but_1) note_name <= F;
                else if (g_but_0) note_name <= G;
                else note_name <= NONE;
        end

////////////////////////////////////////////////////
//                      DEBUGGING
////////////////////////////////////////////////////

        // debugging note counter counter
        wire [4:0] note_counter_counter;

        // debug - with leds
        wire metronome_sound_enable;
        reg met_toggle = 1;
        always @(posedge clock) begin
                        if (metronome_sound_enable) met_toggle <= ~met_toggle;
        end
        wire [4:0] volume;
        assign led[4:0] = ~volume;
        assign led[6:5] = 3'b111;
        assign led[7] = met_toggle;
        wire metronome_tick_enable;         // keep this after debugging, for hex display


        // debug w/ logic analyzer
        assign analyzer3_clock = clock;
        assign analyzer3_data[15:0] = 0;

        // Display (for debugging too)
        // first hex is note length
        // third hex shows counter for metronome_tick_enable, to count note length
        assign disp_data = {1'b0, note_length, 7'b0,
                note_counter_counter, 48'b0};
        display_16hex display1(.reset(reset), .clock_27mhz(clock), .data(disp_data),
                .disp_blank(disp_blank), .disp_clock(disp_clock),
                .disp_data_out(disp_data_out), .disp_rs(disp_rs), .disp_ce_b(disp_ce_b),
                .disp_reset_b(disp_reset_b));



////////////////////////////////////////////////////////////
//              NOTE PROCESSOR
////////////////////////////////////////////////////////////

        note_processor note_processor1(.clock(clock), .reset(reset),
                .metronome_tempo(tempo_switches), .attack(attack),
                .note_name(note_name), .note_length(note_length),
                .button_up(button_up), .button_down(button_down),
                .ac97_sdata_in(ac97_sdata_in), .ac97_bit_clock(ac97_bit_clock),
                .ac97_sdata_out(ac97_sdata_out), .audio_reset_b(audio_reset_b),
                .ac97_synch(ac97_synch), .metronome_on(metronome_on),
                .note_counter_counter(note_counter_counter),
                .metronome_tick_enable(metronome_tick_enable),
                .volume(volume),
				.metronome_sound_enable(metronome_sound_enable)
				);

//////////////////////////////////////////////////////////////
//				GRAPHICS
//////////////////////////////////////////////////////////////

		graphics graphics1(.clock(clock), .reset(reset), .note_name(note_name),
				.note_length(note_length), .attack(attack),
				.screen_mode(screen_mode), .vga_out_red(vga_out_red),
				.vga_out_green(vga_out_green), .vga_out_blue(vga_out_blue),
				.vga_out_sync_b(vga_out_sync_b), .vga_out_blank_b(vga_out_blank_b),
				.vga_out_pixel_clock(vga_out_pixel_clock), .vga_out_hsync(vga_out_hsync),
				.vga_out_vsync(vga_out_vsync));

endmodule


module graphics(
		 input clock, reset,
		 input [2:0] note_name, note_length,
		 input [1:0] attack,
		 input screen_mode,
		 output [7:0] vga_out_red, vga_out_green, vga_out_blue,
		 output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
					vga_out_hsync, vga_out_vsync
    );
////////////////////////////////////////
//		PARAMETERS
////////////////////////////////////////

				// for the notes
		parameter NONE = 0;			// no note played
		parameter A = 1;
		parameter B = 2;
		parameter C = 3;
		parameter D = 4;
		parameter E = 5;
		parameter F = 6;
		parameter G = 7;
		parameter EIGHTH = 0;
		parameter QUARTER = 1;
		parameter HALF = 2;
		parameter THREE = 3;
		parameter WHOLE = 4;

				// for the display
		parameter XRES = 11'd1024;
		parameter YRES = 10'd768;

////////////////////////////////////////
//				VARIABLES
////////////////////////////////////////

				// general variables
```

```
            reg [2:0] counter = 0;
            reg [23:0] rgb;
                        // outputs from xvga
            wire [10:0] hcount;
    wire [9:0] vcount;
            wire hsync, vsync, blank;
                        // clock
            wire clock_65mhz_unbuf,clock_65mhz;
    DCM vclk1(.CLKIN(clock),.CLKFX(clock_65mhz_unbuf));
            // synthesis attribute CLKFX_DIVIDE of vclk1 is 10
    // synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
    // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
    // synthesis attribute CLKIN_PERIOD of vclk1 is 37
    BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));
            wire vclock;
            assign vclock = clock_65mhz;
                        // pixel
            wire [23:0] staff_pixel, keyboard_pixel, note0_pixel, note1_pixel,
                        note2_pixel, note3_pixel, note4_pixel, note5_pixel, note6_pixel,
                        note7_pixel;
            // notedata[8] exists, notedata[7:5] note_name, notedata[4:2] note_length,
                        // notedata[1:0] attack
            reg [8:0] note0_data = 0;
            reg [8:0] note1_data = 0;
            reg [8:0] note2_data = 0;
            reg [8:0] note3_data = 0;
            reg [8:0] note4_data = 0;
            reg [8:0] note5_data = 0;
            reg [8:0] note6_data = 0;
            reg [8:0] note7_data = 0;


//////////////////////////////////////
//                      XVGA
//////////////////////////////////////
    xvga xvga1(.vclock(clock_65mhz),.hcount(hcount),.vcount(vcount),
            .hsync(hsync),.vsync(vsync),.blank(blank));


//////////////////////////////////////
//                      DRAWING/pixel assignment
//////////////////////////////////////

            bar_staff staff1(.vclock(vclock), .reset(reset), .hcount(hcount),
                                    .vcount(vcount), .pixel(staff_pixel));

            keyboard keyboard1(.vclock(vclock), .reset(reset), .hcount(hcount),
                                    .vcount(vcount), .note_name(note_name), .pixel(keyboard_pixel));

            note_blob note0(.hcount(hcount), .vcount(vcount), .note_length(note0_data[4:2]),
                                    .note_name(note0_data[7:5]), .exists(note0_data[8]), .note_count(3'd0),
                                    .pixel(note0_pixel));
            note_blob note1(.hcount(hcount), .vcount(vcount), .note_length(note1_data[4:2]),
                                    .note_name(note1_data[7:5]), .exists(note1_data[8]), .note_count(3'd1),
                                    .pixel(note1_pixel));
```

```
	note_blob note2(.hcount(hcount), .vcount(vcount), .note_length(note2_data[4:2]),
				.note_name(note2_data[7:5]), .exists(note2_data[8]), .note_count(3'd2),
				.pixel(note2_pixel));
	note_blob note3(.hcount(hcount), .vcount(vcount), .note_length(note3_data[4:2]),
				.note_name(note3_data[7:5]), .exists(note3_data[8]), .note_count(3'd3),
				.pixel(note3_pixel));
	note_blob note4(.hcount(hcount), .vcount(vcount), .note_length(note4_data[4:2]),
				.note_name(note4_data[7:5]), .exists(note4_data[8]), .note_count(3'd4),
				.pixel(note4_pixel));
	note_blob note5(.hcount(hcount), .vcount(vcount), .note_length(note5_data[4:2]),
				.note_name(note5_data[7:5]), .exists(note5_data[8]), .note_count(3'd5),
				.pixel(note5_pixel));
	note_blob note6(.hcount(hcount), .vcount(vcount), .note_length(note6_data[4:2]),
				.note_name(note6_data[7:5]), .exists(note6_data[8]), .note_count(3'd6),
				.pixel(note6_pixel));
	note_blob note7(.hcount(hcount), .vcount(vcount), .note_length(note7_data[4:2]),
				.note_name(note7_data[7:5]), .exists(note7_data[8]), .note_count(3'd7),
				.pixel(note7_pixel));
	wire [23:0] note_pixels;
	assign note_pixels = note0_pixel + note1_pixel + note2_pixel + note3_pixel +
			note4_pixel + note5_pixel + note6_pixel + note7_pixel;

	// assign note data on 27 MHz clock cycle
	reg [2:0] last_note_name;
	always @ (posedge clock) begin
				last_note_name <= note_name;
			if (/*(note_name == NONE) && (last_note_name != NONE)*/
				(last_note_name != note_name) && last_note_name != NONE) begin
				counter <= counter + 1;
				case (counter)
					0: note0_data <= {1'b1, last_note_name, note_length, attack};
					1: note1_data <= {1'b1, last_note_name, note_length, attack};
					2: note2_data <= {1'b1, last_note_name, note_length, attack};
					3: note3_data <= {1'b1, last_note_name, note_length, attack};
					4: note4_data <= {1'b1, last_note_name, note_length, attack};
					5: note5_data <= {1'b1, last_note_name, note_length, attack};
					6: note6_data <= {1'b1, last_note_name, note_length, attack};
					7: note7_data <= {1'b1, last_note_name, note_length, attack};
				endcase
				if (counter == 0) begin		// clear data if counter == 0
					note1_data <= 0;
					note2_data <= 0;
					note3_data <= 0;
					note4_data <= 0;
					note5_data <= 0;
					note6_data <= 0;
					note7_data <= 0;
				end
			end

	end

	// switch[6] selects which video generator to use:
//  0: blank screen
	//  1: color bars
```

```
   reg b,hs,vs;
   always @(posedge clock_65mhz) begin

      if (screen_mode) begin
                                          // color bars
                                          hs <= hsync;
                                          vs <= vsync;
                                          b <= blank;
                                          rgb <= {{8{hcount[8]}}, {8{hcount[7]}}, {8{hcount[6]}}} ;
      end else begin
                              // replace this with actual pixel data for drawing
                                          hs <= hsync;
                                          vs <= vsync;
                                          b <= blank;
                                          rgb <= staff_pixel + keyboard_pixel + note_pixels;
      end
   end


////////////////////////////////////////
//                          VGA output
///////////////////////////////////////
               assign vga_out_red = rgb[23:16];
   assign vga_out_green = rgb[15:8];
   assign vga_out_blue = rgb[7:0];
   assign vga_out_sync_b = 1'b1;
   assign vga_out_blank_b = ~b;
   assign vga_out_pixel_clock = ~clock_65mhz;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;

endmodule

module bar_staff(input vclock, reset,
                                                      input [10:0] hcount,
                                                      input [9:0] vcount,
                                                      output reg [23:0] pixel);

               parameter SPACE_HEIGHT = 10'd40;                           // space between each bar
               parameter LINE_WIDTH = 4'd8;                               // width of line
               parameter TOP_OF_STAFF = 10'd150;          // location of top bar
               parameter WHITE = 24'hFF_FF_FF;
               parameter BLACK = 24'h00_00_00;

               always @(posedge vclock) begin
                              if (((vcount >= TOP_OF_STAFF) && (vcount <= (TOP_OF_STAFF+LINE_WIDTH))) ||               // first line
                                          ((vcount >= TOP_OF_STAFF+SPACE_HEIGHT) && (vcount <= (TOP_OF_STAFF+
SPACE_HEIGHT+LINE_WIDTH))) ||
                                          ((vcount >= TOP_OF_STAFF+SPACE_HEIGHT*2) && (vcount <= (TOP_OF_STAFF+
SPACE_HEIGHT*2+LINE_WIDTH))) ||
                                          ((vcount >= TOP_OF_STAFF+SPACE_HEIGHT*3) && (vcount <= (TOP_OF_STAFF+
SPACE_HEIGHT*3+LINE_WIDTH))) ||
                                          ((vcount >= TOP_OF_STAFF+SPACE_HEIGHT*4) && (vcount <= (TOP_OF_STAFF+
SPACE_HEIGHT*4+LINE_WIDTH))))
                                          pixel <= WHITE;
			else pixel <= BLACK;
		end

endmodule

module keyboard(input vclock, reset,
					input [10:0] hcount,
					input [9:0] vcount,
					input [2:0] note_name,
					output reg [23:0] pixel);

		// parameters
			// geometry
		parameter XRES = 1024;
		parameter X_OFFSET = 282;
		parameter LINE_WIDTH = 5; // this leaves 60 for key width
		parameter KEY_WIDTH = 60;
		parameter KEY_HEIGHT = 200;
		parameter KEYBOARD_TOP = 480;  // top line of keyboard
			// colors
		parameter BLACK = 24'h00_00_00;
		parameter WHITE = 24'hFF_FF_FF;
		parameter RED = 24'hEB4528;
		parameter ORANGE = 24'hE8A327;
		parameter YELLOW = 24'hCBE527;
		parameter GREEN = 24'h26E03C;
		parameter TEAL = 24'h2639DA;
		parameter BLUE = 24'h256FD8;
		parameter PURPLE = 24'h8725D2;
			// for the notes
		parameter NONE = 0;			// no note played
		parameter A = 1;
		parameter B = 2;
		parameter C = 3;
		parameter D = 4;
		parameter E = 5;
		parameter F = 6;
		parameter G = 7;

		always @(posedge vclock) begin

			// keyboard outline
			if ((vcount >= KEYBOARD_TOP) && (vcount <= KEYBOARD_TOP + KEY_HEIGHT) &&
					(hcount >= X_OFFSET) && (hcount <= XRES-X_OFFSET)) begin
				// color in outline
				if ((vcount <= KEYBOARD_TOP+LINE_WIDTH) ||						//
top line
					(vcount >= KEYBOARD_TOP + KEY_HEIGHT- LINE_WIDTH) ||	// bottom line
					(hcount <= X_OFFSET + LINE_WIDTH) ||
					(hcount >= XRES-X_OFFSET-LINE_WIDTH)) begin
		//right line
					pixel <= WHITE;
				// color in key outline
				end else if (((hcount >= X_OFFSET + LINE_WIDTH + KEY_WIDTH) && (hcount <= X_OFFSET +
2*LINE_WIDTH + KEY_WIDTH))  ||
                                        ((hcount >= X_OFFSET + 2*LINE_WIDTH + 2*KEY_WIDTH) && (hcount <= X_OFFSET
+ 3*LINE_WIDTH + 2*KEY_WIDTH))  ||
                                        ((hcount >= X_OFFSET + 3*LINE_WIDTH + 3*KEY_WIDTH) && (hcount <= X_OFFSET
+ 4*LINE_WIDTH + 3*KEY_WIDTH))  ||
                                        ((hcount >= X_OFFSET + 4*LINE_WIDTH + 4*KEY_WIDTH) && (hcount <= X_OFFSET
+ 5*LINE_WIDTH + 4*KEY_WIDTH))  ||
                                        ((hcount >= X_OFFSET + 5*LINE_WIDTH + 5*KEY_WIDTH) && (hcount <= X_OFFSET
+ 6*LINE_WIDTH + 5*KEY_WIDTH))  ||
                                        ((hcount >= X_OFFSET + 6*LINE_WIDTH + 6*KEY_WIDTH) && (hcount <= X_OFFSET
+ 7*LINE_WIDTH +6*KEY_WIDTH))
                                                ) begin
                                        pixel <= WHITE;
                                // color in the keys
                                end else begin
                                        case (note_name)
                                                A: begin
                                                        if ((hcount > X_OFFSET + LINE_WIDTH) && (hcount < X_OFFSET +
LINE_WIDTH + KEY_WIDTH) )
                                                                pixel <= RED;
                                                        else pixel <= BLACK;
                                                end
                                                B:  begin
                                                        if ((hcount > X_OFFSET + 2*LINE_WIDTH + KEY_WIDTH) &&(hcount
< X_OFFSET + 2*LINE_WIDTH + 2*KEY_WIDTH) )
                                                                pixel <= ORANGE;
                                                        else pixel <= BLACK;
                                                end
                                                C:  begin
                                                        if ((hcount > X_OFFSET + 3*LINE_WIDTH + 2*KEY_WIDTH)
&&(hcount < X_OFFSET + 3*LINE_WIDTH + 3*KEY_WIDTH) )
                                                                pixel <= YELLOW;
                                                        else pixel <= BLACK;
                                                end
                                                D: begin
                                                        if ((hcount > X_OFFSET + 4*LINE_WIDTH + 3*KEY_WIDTH)
&&(hcount < X_OFFSET + 4*LINE_WIDTH + 4*KEY_WIDTH) )
                                                                pixel <= GREEN;
                                                        else pixel <= BLACK;
                                                end
                                                E: begin
                                                        if ((hcount > X_OFFSET + 5*LINE_WIDTH + 4*KEY_WIDTH)
&&(hcount < X_OFFSET + 5*LINE_WIDTH + 5*KEY_WIDTH) )
                                                                pixel <= TEAL;
                                                        else pixel <= BLACK;
                                                end
                                                F: begin
                                                        if ((hcount > X_OFFSET + 6*LINE_WIDTH + 5*KEY_WIDTH)
&&(hcount < X_OFFSET + 6*LINE_WIDTH + 6*KEY_WIDTH) )
                                                                pixel <= BLUE;
                                                        else pixel <= BLACK;
                                                end
                                                G: begin
                                                        if (( hcount > X_OFFSET + 7*LINE_WIDTH +6*KEY_WIDTH)
&&(hcount <X_OFFSET + 7*LINE_WIDTH +7*KEY_WIDTH) )
                                                                pixel <= PURPLE;
                                                        else pixel <= BLACK;
                                                end
                                                default: pixel <= BLACK;
                                        endcase
                                end
                                //end else pixel <= BLACK;
                        end else pixel <= BLACK;
                end

endmodule


module note_blob(input [10:0] hcount,
                                        input [9:0] vcount,
                                        output reg [23:0] pixel,
                                        input [2:0] note_length, note_name,
                                        input exists,
                                        input [2:0] note_count);

                // parameters
                parameter EIGHTH = 0;
                parameter QUARTER = 1;
                parameter HALF = 2;
                parameter THREE = 3;
                parameter WHOLE = 4;
                parameter UNDEF = 5;
                parameter X_OFFSET = 150;
                parameter X_SPACING = 110;
                parameter Y_TOP_LINE = 150;
                parameter Y_SPACING = 22;

                parameter STEM_LENGTH = 100;

                parameter NONE = 0;             // no note played
                parameter A = 1;
                parameter B = 2;
                parameter C = 3;
                parameter D = 4;
                parameter E = 5;
                parameter F = 6;
                parameter G = 7;

                parameter BLACK = 24'h00_00_00;
                parameter WHITE = 24'hFF_FF_FF;

                parameter RADIUS = 36;
                parameter INNER_RADIUS = 32;

                wire [10:0] x;
                wire [9:0] y;

                assign x = X_OFFSET + X_SPACING*note_count;

                assign y = (note_name == A) ? Y_TOP_LINE + Y_SPACING*5:
                        (note_name == B) ? Y_TOP_LINE + Y_SPACING*4:
```

```
                (note_name == C) ? Y_TOP_LINE + Y_SPACING*3:
                (note_name == D) ? Y_TOP_LINE + Y_SPACING*2:
                (note_name == E) ? Y_TOP_LINE + Y_SPACING*1:
                (note_name == F) ? Y_TOP_LINE :
                (note_name == G) ? Y_TOP_LINE - Y_SPACING: 0;

        always @ * begin

                if (~exists) begin
                        pixel <= BLACK;
                end else begin
                        case (note_length)

                                EIGHTH: begin
                                        if ((x-hcount)*(x-hcount) + 2*(y-vcount)*(y-vcount) < RADIUS*RADIUS)
                                                pixel <= WHITE;
                                        else if ((vcount > y-STEM_LENGTH) && (vcount < y)
                                                        && (hcount > x + RADIUS) && (hcount < x + RADIUS + 4))
                                                pixel <= WHITE;
                                        else if ((hcount < x + RADIUS +15 )&& (hcount > x+ RADIUS)
                                                        && (vcount > y-STEM_LENGTH-3+2*(hcount-x-RADIUS))
                                                        && (vcount < y-STEM_LENGTH+3+2*(hcount-x-RADIUS)))
                                                pixel <= WHITE;
                                        else pixel <= BLACK;
                                end
                                QUARTER: begin
                                        if ((x-hcount)*(x-hcount) + 2*(y-vcount)*(y-vcount) < RADIUS*RADIUS)
                                                pixel <= WHITE;
                                        else if ((vcount > y-STEM_LENGTH) && (vcount < y)
                                                        && (hcount > x + RADIUS) && (hcount < x + RADIUS + 4))
                                                pixel <= WHITE;
                                        else pixel <= BLACK;
                                end
                                HALF: begin
                                        if (((x-hcount)*(x-hcount) + 2*(y-vcount)*(y-vcount) < RADIUS*RADIUS)
                                                        && ((x-hcount)*(x-hcount) + 2*(y-vcount)*(y-vcount) > IN-
NER_RADIUS*INNER_RADIUS))
                                                pixel <= WHITE;
                                        else if ((vcount > y-STEM_LENGTH) && (vcount < y)
                                                        && (hcount > x + RADIUS) && (hcount < x + RADIUS + 4))
                                                pixel <= WHITE;
                                        else pixel <= BLACK;
                                end
                                THREE: begin
                                        if (((x-hcount)*(x-hcount) + 2*(y-vcount)*(y-vcount) < RADIUS*RADIUS)
                                                        && ((x-hcount)*(x-hcount) + 2*(y-vcount)*(y-vcount) > IN-
NER_RADIUS*INNER_RADIUS))
                                                pixel <= WHITE;
                                        else if ((vcount > y-STEM_LENGTH) && (vcount < y)
                                                        && (hcount > x + RADIUS) && (hcount < x + RADIUS + 4))
                                                pixel <= WHITE;
                                        else if (((x+2*RADIUS-5-hcount)*(x+2*RADIUS-5-hcount) + 2*(y-vcount)*(y-
vcount) < 16))
                                                pixel <= WHITE;
                                        else pixel <= BLACK;
                                        end
                                        WHOLE: begin
                                                if (((x-hcount)*(x-hcount) + 2*(y-vcount)*(y-vcount) < RADIUS*RADIUS)
                                                                && ((x-hcount)*(x-hcount) + 2*(y-vcount)*(y-vcount) > IN-
NER_RADIUS*INNER_RADIUS))
                                                        pixel <= WHITE;
                                                else pixel <= BLACK;
                                        end
                                        default: pixel <= BLACK;
                                endcase
                        end
                end


endmodule




module metronome_timer(
    input [3:0] metronome_switches,
            input clock,
            input reset,
    output metronome_tick_enable
    );

            // PARAMETERS
            parameter CYCLES = 25'd27_000_000;

            // variable definition
            wire [24:0] tempo4;                     // tempo4 is 1/4 of an actual metronome beat

            // counter counts clock cycles until it hits tempo, which is set by labkit switches
            reg [24:0] counter = 0;

            // labkit switches [3:0] sets tempo, for 16 unique tempos
            assign tempo4 = (CYCLES - CYCLES/32*metronome_switches[3:0])/4;

            // increment counter until tempo is hit
            always @ (posedge clock) begin
                    counter <= (counter == tempo4) ? 0 : counter + 1;
            end

            //assing metronome_tick_enable here to avoid 1 cycle delay
            assign metronome_tick_enable = (counter == tempo4) ? 1 : 0;

endmodule


module metronome(
    input metronome_tick_enable,
            input clock,
            input reset,
    output reg metronome_sound_enable,
            input metronome_on
    );

            // there are 4 enables from metronome_tick_enable for each
            // tick of the metronome
            reg [1:0] counter = 0;

            always @(posedge clock) begin
                    if (reset) begin
                            counter <= 0;
                    end else if (metronome_tick_enable) begin
                            counter <= counter + 1;
                    end

                    if ((counter == 0) && (metronome_on) && (metronome_sound_enable == 0)) begin
                            metronome_sound_enable <= 1;
                    end else begin
                            metronome_sound_enable <= 0;
                    end

            end

endmodule


module note_counter(
    input metronome_tick_enable,
            input clock,
    input [2:0] note_name,
    output [2:0] note_length,
            output reg [4:0] counter // remove after debug
    );

            // Parameters
            parameter NONE = 0;                     // no note played
            parameter A = 1;
            parameter B = 2;
            parameter C = 3;
            parameter D = 4;
            parameter E = 5;
            parameter F = 6;
            parameter G = 7;

            // parameters
            parameter EIGHTH = 0;
            parameter QUARTER = 1;
            parameter HALF = 2;
            parameter THREE = 3;
            parameter WHOLE = 4;
            parameter UNDEF = 5;

            // variables
            reg [2:0] last_note;
            //reg [4:0] counter = 0;                // counts the quarter-metronome ticks, return after debug

            // while current note is a note, begin count
```

```
	// if the current note is different from note on previous cycle,
	// reset counter
	always @(posedge clock) begin

		last_note <= note_name;

		// I think this logic can be simplified
		if ((note_name != NONE) && metronome_tick_enable && (last_note == note_name)) begin
			counter <=  counter + 1;
		end else if (last_note != note_name) counter <= 0;

	end

	assign note_length = ((counter > 0) && (counter <= 2)) ? EIGHTH :
								((counter >= 3) && (counter <= 5)) ? QUARTER :
								((counter >= 6) && (counter <= 9)) ? HALF :
								((counter >= 10) && (counter <= 13)) ? THREE :
								((counter >= 14) && (counter <= 20)) ? WHOLE : UNDEF;


endmodule

module note_processor(
		input clock,
		input reset,
   input [3:0] metronome_tempo,
   input [3:0] attack,
   input [2:0] note_name,
   output [2:0] note_length,
		input button_up, button_down,
		input ac97_sdata_in, ac97_bit_clock,
		output ac97_sdata_out,audio_reset_b,ac97_synch,
		input metronome_on,
		output [4:0] note_counter_counter, // debugging to display
		output metronome_tick_enable,
		output [4:0] volume,
		output metronome_sound_enable
   );

	// PARAMETERS

	// VARIABLES
	//wire metronome_tick_enable;	// four per beat return after debugging
	//wire metronome_sound_enable;		// one per beat

	// metronome timer - input: switches
								// output: metronome_tick_enable
	metronome_timer m_timer1(.metronome_switches(metronome_tempo),
				.clock(clock), .reset(reset),
				.metronome_tick_enable(metronome_tick_enable));

	// metronome - input: metronome_tick_enable
						// output: metronome_sound_enable
	metronome metronome1(.clock(clock),
				.metronome_tick_enable(metronome_tick_enable), .reset(reset),
                                .metronome_sound_enable(metronome_sound_enable),
                                .metronome_on(metronome_on));

            // note counter - uses inputs to figure out how long each note lasts
                                                            // input: metronome_tick_enable, note_name
                                                            // output: note_length
            //wire [5:0] note_counter_counter;
            note_counter note_counter1(.metronome_tick_enable(metronome_tick_enable),
                                .clock(clock), .note_name(note_name), .note_length(note_length),
                                .counter(note_counter_counter));

            // sound_transformer - adjusts sound output
                                                            // input: attack, note_name
                                                            // output: sound to ac97 module
            sound_transformer sound_transformer1(.clock(clock), .reset(reset),
                      .attack(attack), .note_name(note_name), .button_up(button_up),
                      .button_down(button_down), .metronome_sound_enable(metronome_sound_enable),
                      .metronome_tick_enable(metronome_tick_enable),
                      .ac97_sdata_in(ac97_sdata_in), .ac97_sdata_out(ac97_sdata_out),
                      .ac97_bit_clock(ac97_bit_clock),
                      .audio_reset_b(audio_reset_b), .ac97_synch(ac97_synch),
                      .volume(volume));


endmodule


module sound_transformer(
   input clock,
   input reset,
   input [1:0] attack,
            input [2:0] note_name,
            input button_up,             // for volume
            input button_down,                     // for volume
            input metronome_sound_enable,
            input metronome_tick_enable,
            input ac97_sdata_in,         // inputs to lab5audio
            input ac97_bit_clock, // inputs to lab5audio
            output ac97_sdata_out,
            output audio_reset_b,   // ac97 interface signals
            output ac97_synch,
            output reg [4:0] volume = 8
   );

            // PARAMETERS
                      // BRAM parameters
            parameter NOTE_A_MAX_ADDR = 14'd8367;
            parameter NOTE_B_MAX_ADDR = 14'd8603;
            parameter NOTE_C_MAX_ADDR = 14'd8677;
            parameter NOTE_D_MAX_ADDR = 14'd8803;
            parameter NOTE_E_MAX_ADDR = 14'd9016;
            parameter NOTE_F_MAX_ADDR = 14'd9100;
            parameter NOTE_G_MAX_ADDR = 14'd9100;
            parameter MET_TICK_MAX_ADDR = 10'd900;
            parameter WE = 1'd0;                                            // don't enable writing for BRAM
			// note name assignments
		parameter NONE = 0;			// no note played
		parameter A = 1;
		parameter B = 2;
		parameter C = 3;
		parameter D = 4;
		parameter E = 5;
		parameter F = 6;
		parameter G = 7;

		// VARIABLES
		wire [7:0] from_ac97_data;
  wire ready;
		wire vup,vdown;
  reg old_vup,old_vdown;
		reg [4:0] base_volume = 8;
		//reg [4:0] volume = 8;
		reg [7:0] to_ac97_data;
		reg [2:0] old_note_name;

		reg new_metronome_sound = 0;
		reg last_metronome_sound_enable;

			// sound variables
		reg [13:0] note_addr;
		reg [9:0] met_addr = 0;
		wire signed [7:0] A_audio, B_audio, C_audio, D_audio,
							E_audio, F_audio, G_audio, metronome_audio;
		reg [7:0] note_audio, combined_audio;
		wire signed [17:0] filtered_audio;
		wire [7:0] din = 0; // don't need to write to bram, just set 0
		reg [13:0] max_note_addr = 900;
		wire [13:0] midnote;
		assign midnote = max_note_addr - 250; // from here on this as the note decays -- adjust this to work well!!!!!!!
		reg [2:0] counter = 0;


		// debounce buttons
		debounce bup(.reset(reset),.clock(clock),.noisy(~button_up),.clean(vup));
  debounce bdown(.reset(reset),.clock(clock),.noisy(~button_down),.clean(vdown));


///////////////////////////////////////////////////
//					END OF VARIABLE DECLARATIONS, START SOUND PROCESSING
///////////////////////////////////////////////////

		// INSTANTIATE BRAM
		met_tick met_tick1(.clka(clock), .dina(din),
					.addra(met_addr), .wea(WE), .douta(metronome_audio));
		note_A note_A1(.clka(clock), .dina(din),
					.addra(note_addr), .wea(WE), .douta(A_audio));
		note_B note_B1(.clka(clock), .dina(din),
					.addra(note_addr), .wea(WE), .douta(B_audio));
		note_C note_C1(.clka(clock), .dina(din),
                        .addra(note_addr), .wea(WE), .douta(C_audio));
note_D note_D1(.clka(clock), .dina(din),
                        .addra(note_addr), .wea(WE), .douta(D_audio));
note_E note_E1(.clka(clock), .dina(din),
                        .addra(note_addr), .wea(WE), .douta(E_audio));
note_F note_F1(.clka(clock), .dina(din),
                        .addra(note_addr), .wea(WE), .douta(F_audio));
note_G note_G1(.clka(clock), .dina(din),
                        .addra(note_addr), .wea(WE), .douta(G_audio));

// FILTER UNIT
fir31 filter1(.clock(clock), .reset(reset), .ready(ready),
                        .x(combined_audio), .y(filtered_audio));

// GENERATE AUDIO
// piano sound should be put in to_ac97_data, use volume to change attack
lab5audio piano_sound(clock, reset, volume, from_ac97_data, to_ac97_data,
                                    ready, audio_reset_b, ac97_sdata_out, ac97_sdata_in,
                                    ac97_synch, ac97_bit_clock);

// for each clock cycle (1) chose note
// (2) determine if base volume needs changing
always @ (posedge clock) begin

            last_metronome_sound_enable <= metronome_sound_enable;
            // if we have a new metronome tik
            if ((metronome_sound_enable != last_metronome_sound_enable) &&
                        metronome_sound_enable) begin
                        new_metronome_sound <= 1;

            end

            if (metronome_tick_enable) volume <= (volume == 0) ? 0 :volume - 1;

// VOLUME CONTROL - 11/2 currently taken from lab 5
            // adjusted so to be between 5 and 27, so that there is room for
            // decay in the sound
            if (reset) base_volume <= 5'd8;
            else begin
                        if (vup & ~old_vup & base_volume != 27'd26) base_volume <= base_volume+1;
                        if (vdown & ~old_vdown & base_volume != 5'd3) base_volume <= base_volume-1;
            end
            old_vup <= vup;
            old_vdown <= vdown;

 // when ac97 is ready for new sample
            if(ready) begin
                        counter <= counter + 1;

            // CHOOSE NOTE
                        case (note_name)
                                    NONE: note_addr <= 0;             // no note played, reset addr to 0
                                    A: begin
                                                note_audio <= A_audio;
                                                max_note_addr <= NOTE_A_MAX_ADDR;
                        end
                        B: begin
                                note_audio <= B_audio;
                                max_note_addr <= NOTE_B_MAX_ADDR;
                        end
                        C: begin
                                note_audio <= C_audio;
                                max_note_addr <= NOTE_C_MAX_ADDR;
                        end
                        D: begin
                                note_audio <= D_audio;
                                max_note_addr <= NOTE_D_MAX_ADDR;
                        end
                        E: begin
                                note_audio <= E_audio;
                                max_note_addr <= NOTE_E_MAX_ADDR;
                        end
                        F: begin
                                note_audio <= F_audio;
                                max_note_addr <= NOTE_F_MAX_ADDR;
                        end
                        G: begin
                                note_audio <= G_audio;
                                max_note_addr <= NOTE_G_MAX_ADDR;
                        end
                        default: note_addr <= 0;
                endcase

        // INCREMENT NOTE AND METRONOME ADDRESSES
                if (counter == 0) begin                 // every 8th sample
                        old_note_name <= note_name;
                        if (note_name != old_note_name) begin // new note
                                note_addr <= 0;
                                volume <= (base_volume + attack*2 <31) ?
                                        base_volume + attack*2 : 31;
                        end else if (note_addr != max_note_addr) begin
                                note_addr <= note_addr + 1;
                        end else if (note_addr == max_note_addr) begin
                                note_addr <= midnote;     // loops back until note stops
                        end
                        if (new_metronome_sound) begin
                                met_addr <= 0;
                                new_metronome_sound <= 0;
                        end
                        else if (met_addr < MET_TICK_MAX_ADDR) met_addr <= met_addr + 1;
                end

                // COMBINE METRONOME AND NOTE SOUND for input to sound filter
                if (counter == 0) begin         // every 8th sample
                        combined_audio <= note_audio + metronome_audio;
                end else begin
                        combined_audio <= 0;                    // zero expand the 7 other samples
                end

                // FILTER
```

```
				// do I have the right range of filtered audio????? I think so b/c of downsampling
				to_ac97_data <= filtered_audio[14:7];
			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 clock,reset,ready,
  input wire signed [7:0] x,
  output reg signed [17:0] y
);
  /*// for now just pass data through
  always @(posedge clock) begin
    if (ready) y <= {x,10'd0};
  end*/

  // variables
  reg signed [7:0] sample [31:0];					// 32 element arry, each 8 bits wide
  reg [4:0] index = 0;										// points to the nth element of
sample
  reg [4:0] offset = 0;
  wire signed [9:0] coeff;									// coefficient
  reg signed [17:0] accumulator = 0;

  // instantiate a coeff31 module, to give us coeffs
  coeffs31 coeffs31_1(.index(index), .coeff(coeff));

  // accumulator logic
  always @(posedge clock) begin
			if (ready) begin
				offset <= offset + 1;
				sample[offset] <= x;										// sample
storage
				index <= 0;
				accumulator <= 0;
			end

			if (index < 31) begin
				accumulator <= accumulator + coeff*sample[offset-index];
				index <= index + 1;
```

```
                              end

                              if (index == 31) y <= accumulator;
  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,
  output reg signed [9:0] coeff
);
  // tools will turn this into a 31x10 ROM
  always @(index)
    case (index)
      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
```

```
endmodule
```