FINAL PROJECT: DIGITAL TUNER

ROSHNI COOPER, LINDA FANG
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
6.111 INTRODUCTION TO DIGITAL SYSTEMS LABORATORY
PROFS: TERMAN AND CHUANG
TA: JAVIER CASTRO
DECEMBER 14, 2005

ABSTRACT

This final project has built a digital tuner. It has a variety of independent functions and modules, utilizing a microphone, speaker, keyboard, mouse and FPGA. It can record audio input and play the most recently recorded audio sample back to the user. It can also play a pure tone selected by the user with either the switches, mouse, or keyboard connected to an FPGA. At the same time, the tuner can display the note information (location on the staff, note name, note number, and frequency) of the user selected tone on the VGFA display.

TABLE OF CONTENTS

Abstract i

Table of Contents ii

List of Figures ii

List of Tables iii

Overview 1

Description 3

Testing and Debugging 32

Conclusion 38

Appendices 41

Appendices A – W (By Roshni Cooper) 41

Appendices 1-3 (By Linda Fang) 74

LIST OF FIGURES

Figure 1: Block Diagram of Digital Tuner 1

Figure 2: Input/Output Audio/Video Module Block Diagram 3

Figure 3: Speaker Module Block Diagram 7

Figure 4: Video Module Block Diagram 10

Figure 5: Note2 Module Block Diagram 11

Figure 6: Get_y Module Block Diagram 12

Figure 7: Note Module Block Diagram 13

Figure 8: Sharp Module Block Diagram..14

Figure 9: Mouse_disp Module Block Diagram 15

Figure 10: Staff_disp Module Block Diagram...15

Figure 11: Get_strings Module Block Diagram...16

Figure 12: Get_letter Module Block Diagram 17

Figure 13: Note Number Finder Block Diagram 19

Figure 14: Filter Block Diagram...20

Figure 15: Version 1 Matlab Ouput..21

Figure 16: Version 1 Convolution FSM 23

Figure 17: Version 2 Matlab Output...25

Figure 18: Version 2 & 3 Note Number FSM 26

Figure 19: Version 2 & 3 Algorithm Read/Write FSM 27

Figure 20: Version 3 Matlab Output...29

Figure 22: Sine Waves in Version 2 35

LIST OF TABLES

Table 1: Signals to Be Convolved 24

Table 2: Convolution Output 24

OVERVIEW

This project implements a digital tuner. It's basically used for people who want to tune instruments or their voices. It has three major functions – record, playback and play a tone (see Figure 1). The user will select the functions via the buttons on the FPGA. Record takes a user microphone input and stores it in a BRAM. Playback plays back the last recorded tone and display its position on a music staff, its frequency and its note number (the numeric value corresponding to the key of the tone on a piano). Play tone takes one of three user inputs (switches[6:0], mouse, or keyboard) and generates a tone using a DDS, outputting it on the video display same as if it were a playback.

A typical piano has 88 keys, ranging in frequency from 27.5Hz to 4.186KHz. These are the values that will be stored into the tonecheck ROM and will be the valid range for testing.

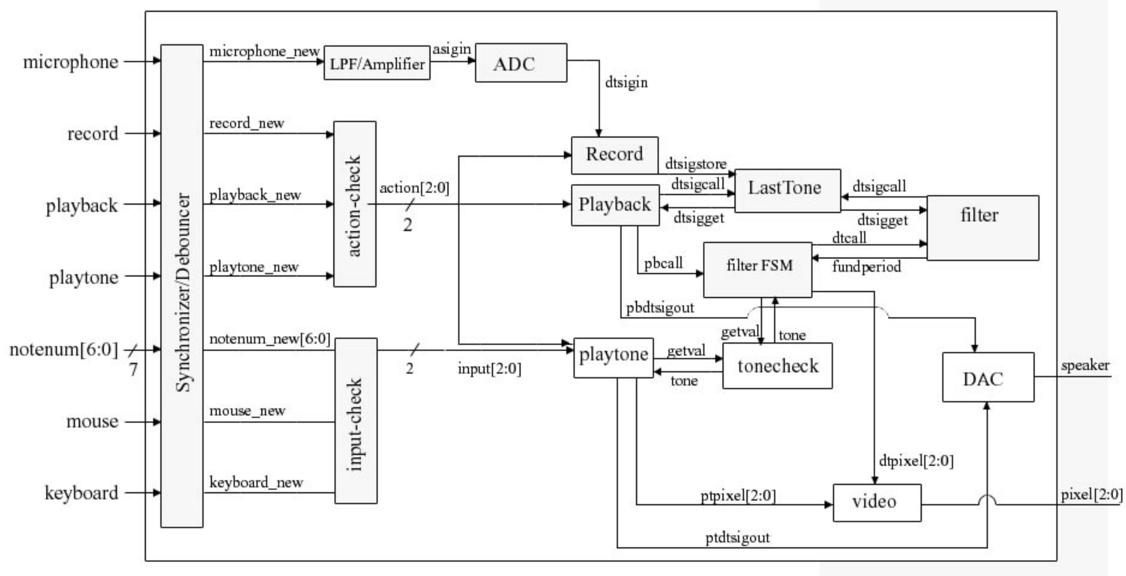

Figure 1: The block diagram that this project implements.

The filter FSM and filter are the modules that will find the frequency of the input tone. It generates signals using the frequencies prestored into the tonecheck ROM, and compares those with the input signal until it finds one it matches with. These were implemented using three different methods, none of which worked.

DESCRIPTION

LABKIT (BY ROSHNI COOPER)

The labkit is the top level module for the digital tuner. It assigns all of the inputs and outputs of the FPGA, and within it the submodules which make up the digital tuner are instantiated.

The entire digital tuner is run using a 65MHz clock, which needs to be defined in the Labkit module using the FPGA's digital clock manager from the 27MHz clock. This is implemented by arithmetically manipulating the frequency of the clock.

In addition, a global reset is defined in the top level module. This reset combines a reseting of the FPGA when both the user reset button is selected and when the FPGA is restarted.

Figure 2 shows a block diagram of the Labkit module, which the important inputs and outputs emphasized.

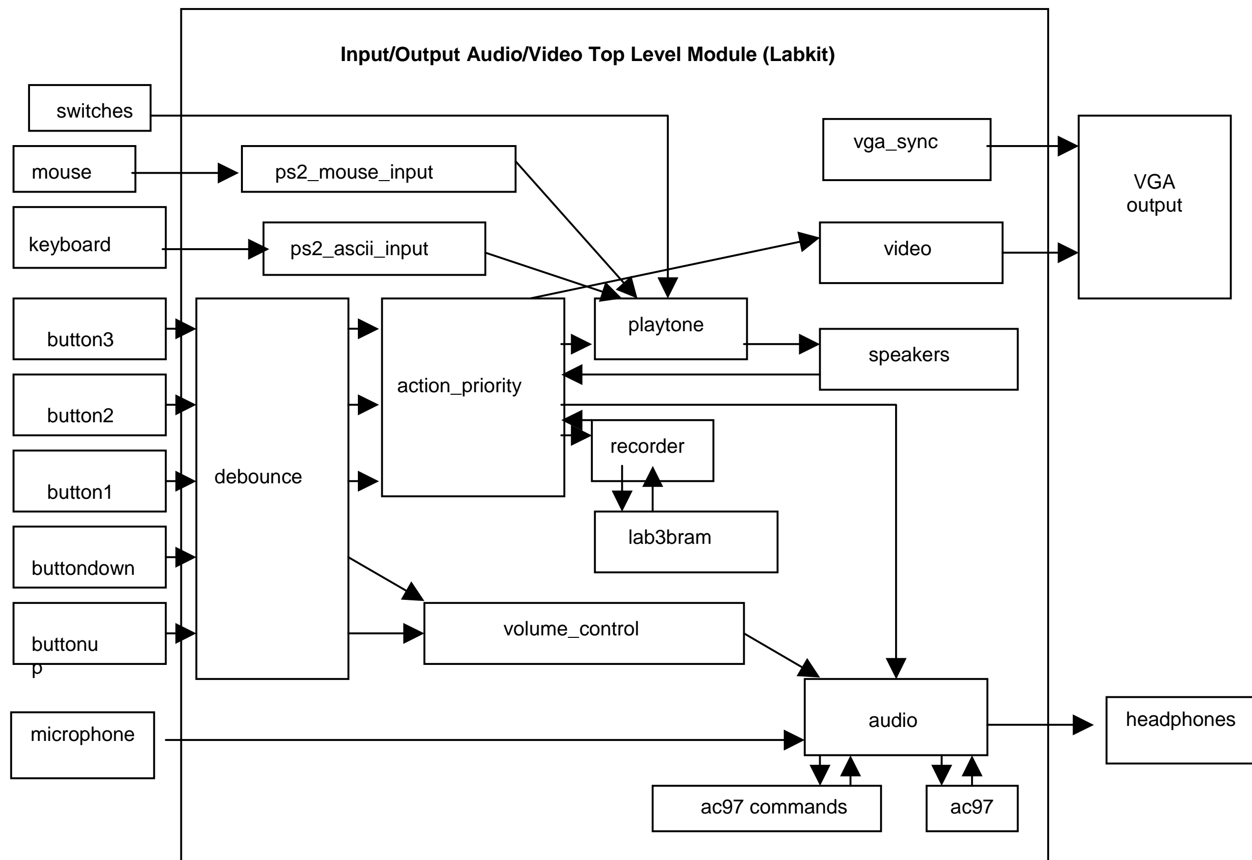

Figure 2: Block diagram of top level of input/output audio/video module

The digital tuner makes use of several of the buttons on the FPGA, and those must all be debounced to ensure the most accurate button inputs.

Three of the buttons control which action the digital tuner is taking at a particular time. The module action_priority determines which of the three actions the user is specifying based on a predetermined priorization. Record mode is selected by pushing button3 of the FPGA, playback by pushing button2, and playtone with button1.

The possible actions of the digital tuner are record, playback, and playtone. The recorder module both records and plays back a user inputted tone, which is stored and accessed from a BRAM titled lab3bram. The playtone module determines the note number of a note based on user selected inputs, either the FPGA's switches, the mouse, or the keyboard.

The mouse and keyboard modules (ps2_mouse_xy and ps2_ascii_input) are instantiated in this top level module, since they convert the inputs into the Labkit of mouse_data, mouse_clock, keyboard_data, and keyboard_clock into information usable by the video and audio outputs of the FPGA. The Labkit module assigns the audio and video outputs for the FPGA, both of which are crucial to the proper functionality of the digital tuner.

The audio of the FPGA is controlled by the AC97. The audio module creates an interface between the Labkit module and the AC97 using the ac97 and ac97_commands modules. The volume of the output is set by the volume_control module which allows the user to utilize the up and down buttons of the FPGA to adjust the volume. The playtone module interacts with the speakers module to use the DDS of the FPGA to output a signal based on the frequency corresponding to the note number of the note the user has selected to hear.

The video display of the FPGA is controlled by the vga_output pins. These are assigned using the vga_sync module and the video module. The vga_sync module sets the hsync, vsync, and blank of the VGA outputs, and the video module sets the pixel value for the red, blue, and green outputs of the VGA. The video module displays a background image of a musical staff, a mouse cursor, the latest keyboard input, the current note and a number of strings (including the current note letter and number, frequency, action, and method of note selection).

The following few pages describe in detail each submodule of the Labkit module. The code for the Labkit module and the following modules can be found in the appendices.

Debounce

The debounce module stabilizes the button inputs of the FPGA. Since the buttons are critical in determining the function of the digital tuner, it is imperative that they are stable.

Action_Priority

The action_priority module has multiple functions. The primary function is to provide the output action to a number of other modules in the top level module. The output action is a two bit value that represents record mode when it has a value of 3 (or 2'b11), playback at a value of 2 (or 2'b10), and playtone at 1 (or 2'b01). If action is equal to 0, no button to select an action has been pushed yet, so the entire system is waiting. Once an action button is pressed, the action remains the same until another action button is selected. If two buttons are pushed simultaneously, the action takes the highest possible value (i.e., if both the button for record and playtone buttons are pushed at once, the action will become record).

The other role of the action_priority module is to tell the audio and video subsystems which note to play and display. Hence, action_priority takes two note numbers and two to_ac97_datas as inputs, one of each from playback and one from playtone. Based on the determined action, the outputted note_num and to_ac97_data are assigned to the inputs from either playtone or playback.

Recorder

The recorder module performs two very important functions: record and playback. Action is sent to the recorder module, and based on its value, the recorder module either records, plays back, or does neither. The recorder module also takes from_ac97_data, which it sends to the BRAM at address din in record mode, and it takes the address dout from which to read the data from the BRAM which is then outputted to_ac97_data to be played back to the user. The recorder must reset the address from which to read or write whenever the digital turner transitions into the record or playback modes. Hence registers maintain the most recent values of action to determine if the address must be reset. During record, the we sent to the BRAM must be high.

Lab3bram

This BRAM stores the most recently recorded data. The BRAM is accessed using the inputs and outputs of the recorder module. The BRAM is 20 bits wide and 65536 addresses deep, allowing for approximately a second worth of recording.

Playtone

The playtone module is only functional if the action input is equal to 1 (2'b01). Assuming the action is indeed playtone, the playtone module outputs a note number which corresponds to the note selected by the user in one of three ways. The user can set a specific note number (from 1 to 88) using the switches of the FPGA, or select a note on the musical staff by clicking a location on the staff with the mouse, or finally input a numerical value between 1 and 88 on the keyboard. If more than one of these inputs is active at once, the playtone module prioritizes in the order switches, then mouse, then keyboard.

The output of the note number is straightforward when the input is from the switches; the note number is simply the value of the switches. When the user is using the mouse input, however, the y location of the mouse click determines the note number. From the code you can see an elaborate selection statement to determine which note number a mouse click corresponds too. If the user clicks off of the staff, then the mouse click has no effect. The keyboard input, which is in the form of two ascii values, must be converted to a number by subtracting the ascii value of "0" from each of the digits. If either of these digits is not a number, then the keyboard input is considered invalid. If none of the inputs is valid, the playtone module will revert to the most recent valid input.

The playtone module also outputs ninput, which is a value of 3 (for switches), 2 (for mouse), 1 (for keyboard), or 0 (for waiting) which is used by the video module to display the current input form on the VGA display.

Mouse

The mouse_data and mouse_clock are sent to the prewritten ps2_mouse_xy module, and the following outputs are received: mx and my, which are twelve-bit outputs which store the current x and y location of the mouse cursor, and btn_click which is three bits wide and is 0 if no click has occurred, and nonzero when a click has occurred. The actual value of btn_click changes based on whether there is a right, left, or middle mouse click, but for the purposes of the digital tuner, the only fact of importance is the presence of any mouse click at all.

Keyboard

The keyboard module is essentially the prewritten ps2_ascii_input module, which takes the keyboard_data and keyboard_clock inputs from the FPGA, as well as a few extra registers to maintain the last two inputted ascii values from the keyboard. The output ascii_ready must be high if a key has been pressed on the keyboard. A concatenation of the two lastest ascii values is sent to the playback module. Pressing the carriage return once on the keyboard resets the keyboard input values.

Speakers (Figure 3)

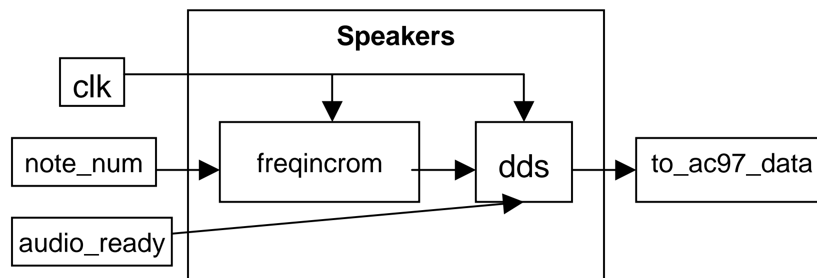

Figure 3: Speakers submodule

This module takes a note number and uses the built in DDS to send a corresponding audio signal to the output to_ac97_data.

DDS

The DDS module takes a frequency increment and, based on this value, outputs a sine wave. This sine wave is then used as an audio output. The frequency increment value is a function of the desired output frequency:

frequency increment = f_{out} * (2^(number of bits in increment)) / clk_dds

The output frequency is unique for each of the 88 notes. The frequency increment is described with eighteen bits. The clk_dds is set using the audio_ready signal from the AC97, and is run at 48KHz.

Frequency Increment ROM

Since Verilog and the FPGA cannot easily handle division by numbers that are not powers of two, a ROM was instantiated in which each of the entries is a frequency increment corresponding to the note number, which is the location of that frequency increment in the ROM.

Audio

The audio module is basically the same as the prewritten module, with only the following change: the volume, instead of being a parameter of the audio module, is now an input to the module, so that the volume can be controlled by the user. This module forms an interface between the labkit and the AC97 and AC97_commands modules.

AC97

This module accesses the AC97, which is the audio output module of the FPGA.

AC97_commands

This module communicates with the audio module to control the current functions of the AC97.

Volume_control

This module takes the signals from the up and down buttons of the FPGA and allows the user to change the current volume level. The volume ranges from 0 to 15, and is initially set to a value of 10. Every time a button is pushed, the volume changes. Registers keep track of the "rising" edges of the up and down buttons, and on these "edges" change the volume. Once the volume reaches the limits of 0 or 15, it can no longer be brought down or up, respectively.

Vga_sync

This module is the same as the prewritten module, and it sets the hsync, vsync, and blank signals for the VGA outputs of the FPGA. The numbers in this module are all set specifically for the 1024 x 768 screen resolution.

Video (Figure 4)

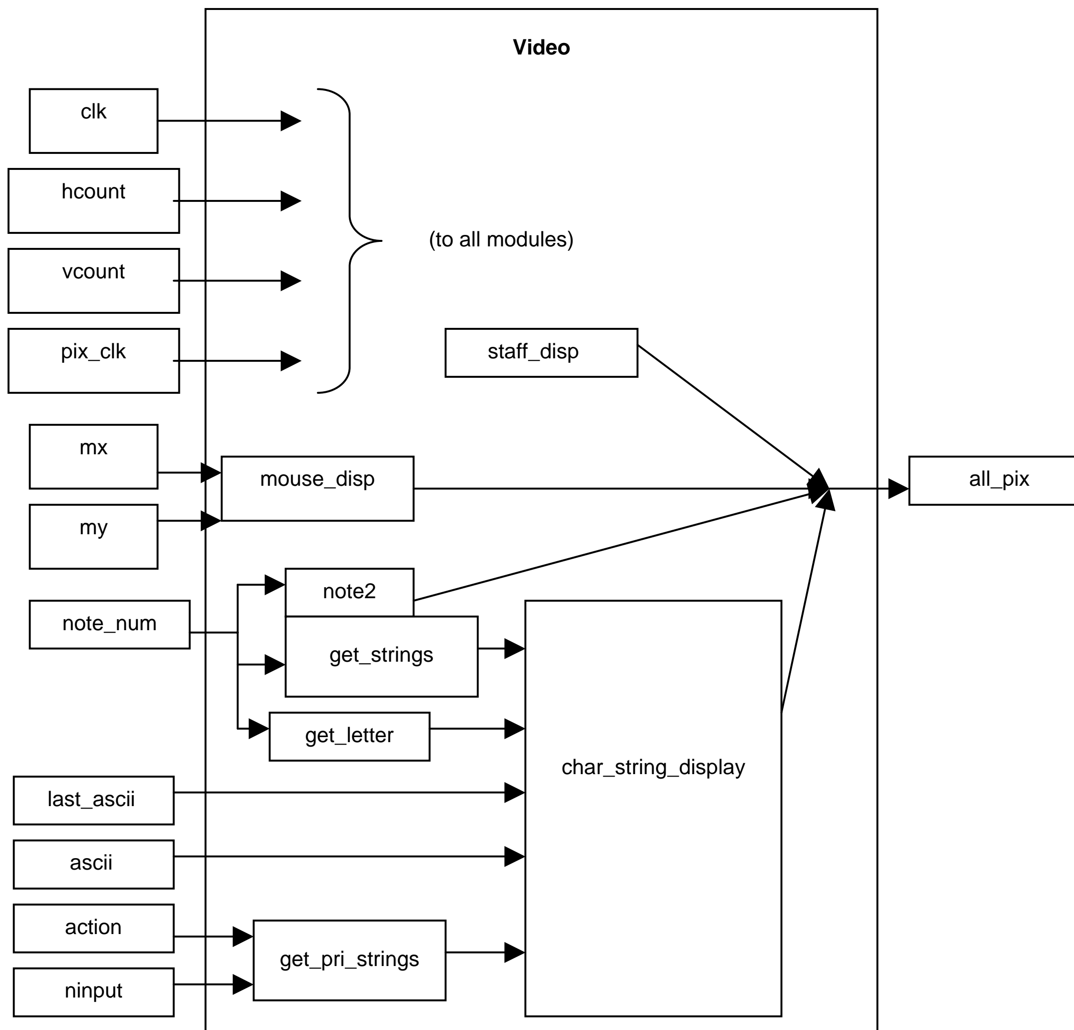

Figure 4: Video submodule

The video module has a number of submodules. It creates the pixels of the staff, the mouse cursor, the note, and the strings of the note number and frequency, they keyboard input, and the current action and user selected input. Each of these individual module's pixels are one bit: either 0 for black or 1 for white. Since the background of the overall video display is white, at any location of hcount and vcount, if any of the individual pixels is black, the overall pixel outputted by the video module must also be black, so the individual pixels are all combined using an and gate.

Note2 (Figure 5)

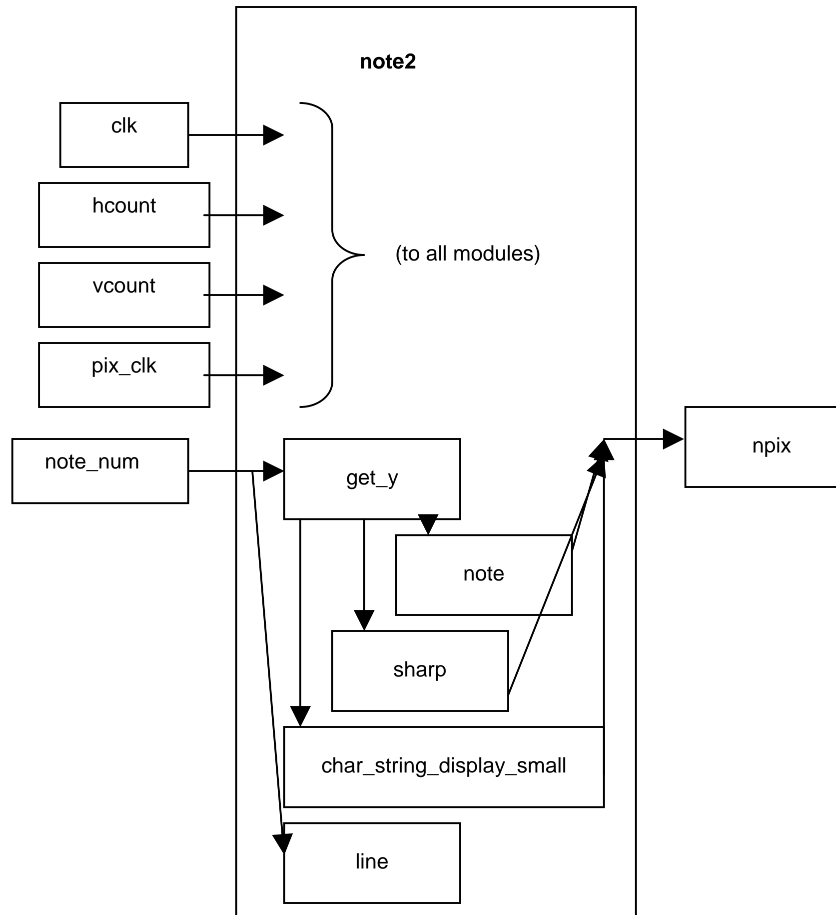

Figure 5: Note2 submodule. Creates the note and additional note drawing information

This module provides the pixels for the note. The note consists of the circle to display on the staff, possibly an accidental (sharp or flat symbol), and perhaps some sort of va notation (where 8va indicates the same note, but an octave higher or lower, 16va denotes two octaves higher or lower, etc.). In addition, if the note is middle C, it needs an additional line through it since it sits in between the treble and bass staves. Each of these components generates its own pixels and then combines them using an and gate.

Get_y (Figure 6)

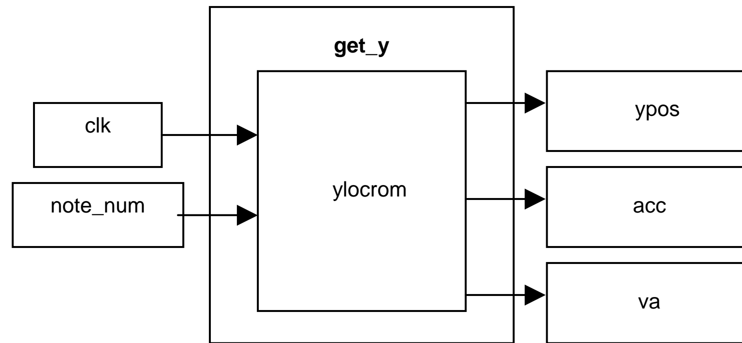

Figure 6: get_y module; returns information about the location of the note

This submodule looks up the given note number in a ROM and returns the y location on the screen of the given note, the presence and type of an accidental, and the presence and type of va notation.

Ylocrom

This ROM has 88 locations, each of which are 14 bits wide. The 14 bits can be interpreted as follows:

Bits 13:4 → the y position of the note in question

Bits 3:2 → the presence and type of an accidental:

2'b00 for no accidental

2'b01 for a sharp

2'b10 for a flat

Bits 1:0 → the presence and type of va notation

2'b00 for no va

2'b01 for 8va

2'b10 for 16va

2'b11 for 24 va

Note (Figure 7)

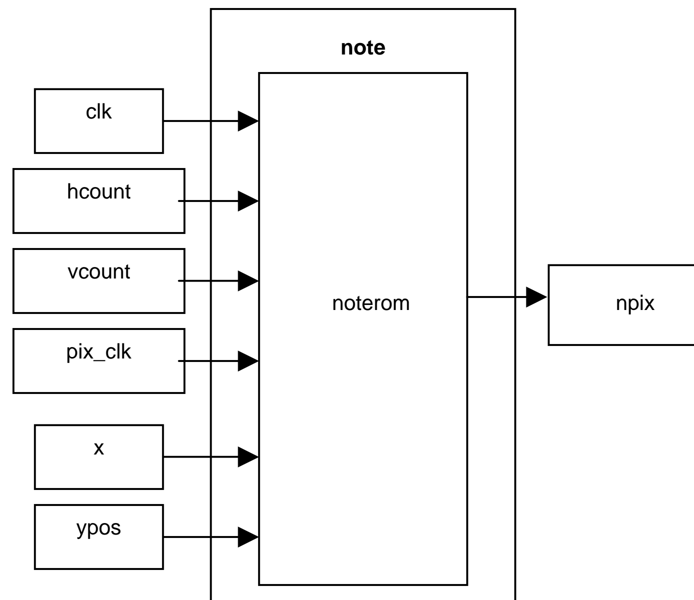

Figure 7: Note submodule

This module simply draws the image in noterom at the location passed in from the Note2 module.

<u>*Noterom*</u>

This ROM stores the image of the note—a simple circle. Storing the circle in the ROM is more efficient than drawing the circle as a sprite during every clock cycle.

Sharp (Figure 8)

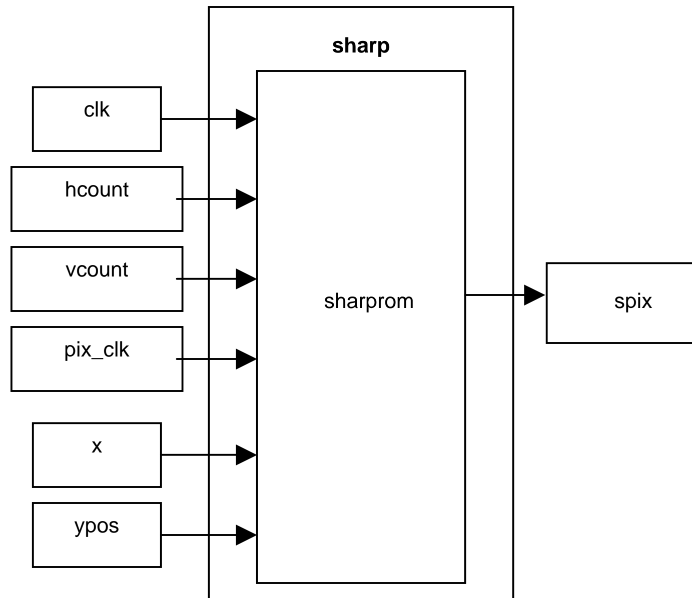

Figure 8: Sharp submodule

This module draws a sharp if necessary at the location it is passed.

Sharprom

This ROM stores the image of a sharp. Currently there is no ROM for the image of a flat, because any note can be a sharp or a flat.

Line

This simple module draws a short line around the location of middle C.

Char_string_display_small

This module is very similar to the char_string_display, only with changes in a few numbers to make the display size of the font smaller. This module is used to create the va notation which is placed above the staff for notes above middle C and below the staff for notes below middle C.

Mouse_disp (Figure 9)

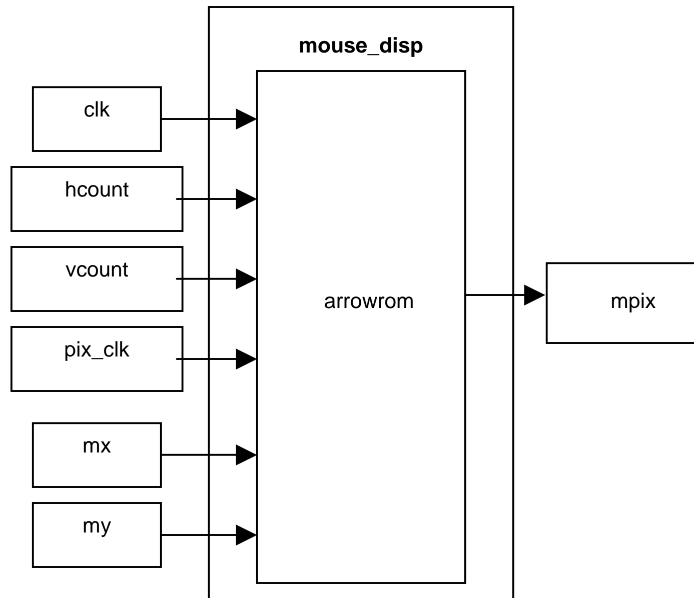

Figure9: Mouse_disp submodule

This module takes in the location of the mouse, and uses it to draw the cursor from the arrowrom.

Arrowrom

This ROM contains the image of an arrow to be used for the mouse cursor.

Staff_disp (Figure 10)

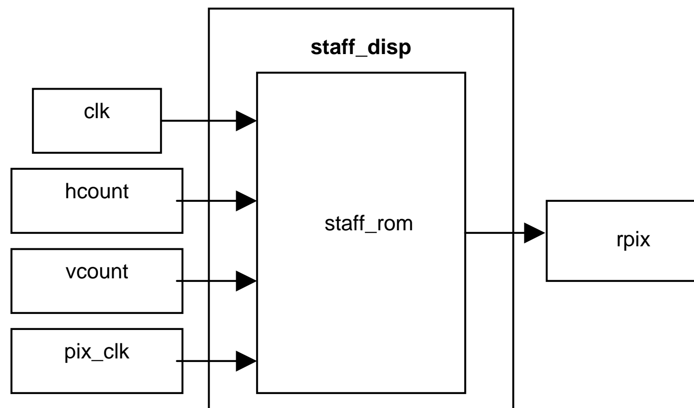

Figure 10: Staff_disp submodule

This module displays the background image of the staff. It concatenates the hcount and vcount to determine the current location in the ROM of the image.

Staff_rom

ROM containing the background staff image.

Get_strings (Figure 11)

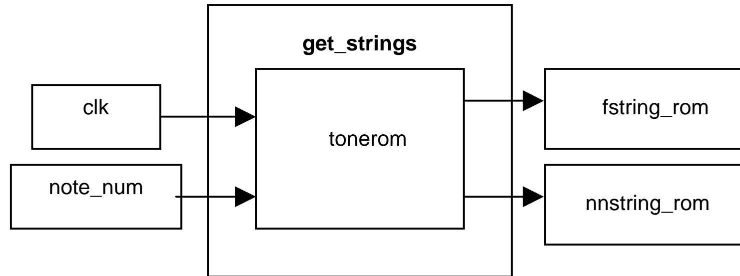

Figure 11: get_strings submodule

This module takes in a note number and uses it to find the frequency of the note in the tonerom. It then converts both the note number and the frequency into strings to be displayed on the video display. In order to convert the binary number into a string, we must first convert the binary number into a "decimal" number, which can then be converted into an ascii value digit by digit. This conversion occurs by subtracting first a certain number of numbers in the thousands, then the hundreds, etc. Since the note number is at most 88, we can first check to see if the number is greater than 80, in which case the "tens" digit is 8, and then ones digit is note_num-80. Else if the number is greater than 70, the "tens" digit becomes 7, and the "ones" digit note_num-70, etc. Similarly, the frequency is no larger than 4186, so we can place restrictions on the number of cases which need to be checked to determine the thousands, hundreds, tens, and ones digits of the frequency.

Tonerom

The ROM which contains the note numbers' corresponding frequencies.

Get_letter (Figure 12)

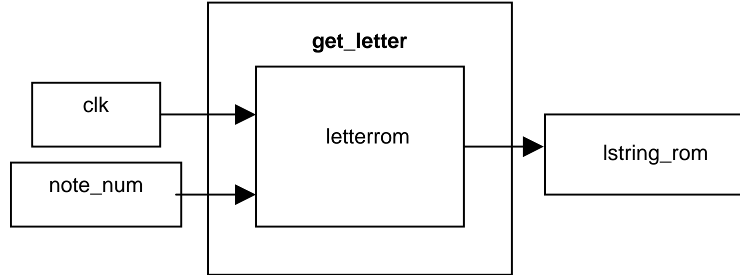

Figure 12: get_letter submodule

This module takes the note number and looks up the corresponding letter in the letterrom, and then converts the data in the letterrom into a string which describes the note of interest: "A", "G Sharp", etc. The ROM gives the values of the letters as their ascii code minus the ascii code of "A," i.e. the note A has value in the ROM of 3'b000, B has 3'b001, etc., since 3'b000 + 65 = "A", 3'b001 + 65 = "B", etc. The ROM also provides information about the accidentals for a given note number (sharp, flat).

Letterrom

This ROM has an entry corresponding to each note number, where the data is 5 bits:

Bits [4:2] → (ascii value of the letter) – "A"

Bits [1:0] → accidental?

2'b00 for no accidental ("   ")

2'b01 for "Sharp"

2'b10 for "Flat "

Get_pri_strings

Converts the action and ninput values into strings to be displayed:

Action:2'b11 → "Record "

2'b10 → "Playback"

2'b01 → "Playtone"

2'b00 → "Waiting"

Ninput:2'b11 → "Switches"

2'b10 → "Mouse   "

2'b01 → "Keyboard"

2'b00 → "Waiting"

Char_string_display

This prewritten code takes in a character string (a string where each character is represented by its 8-bit ascii value) and outputs the pixel for that string using the font_rom.

Font_rom

The ROM containing all characters represented by ascii values.

NOTE NUMBER FINDER (BY LINDA FANG)

There were three versions written for this part. The main idea behind this part is to find the note number corresponding to the input signal. All three will essentially start checking at some given note number, and change that as see fit (see Figure 13). They also implement different algorithms that compare the input signal with the signal generated from the note number, and filter the calculated signal with a low pass filter to see if the two signals match (see Figure 14).

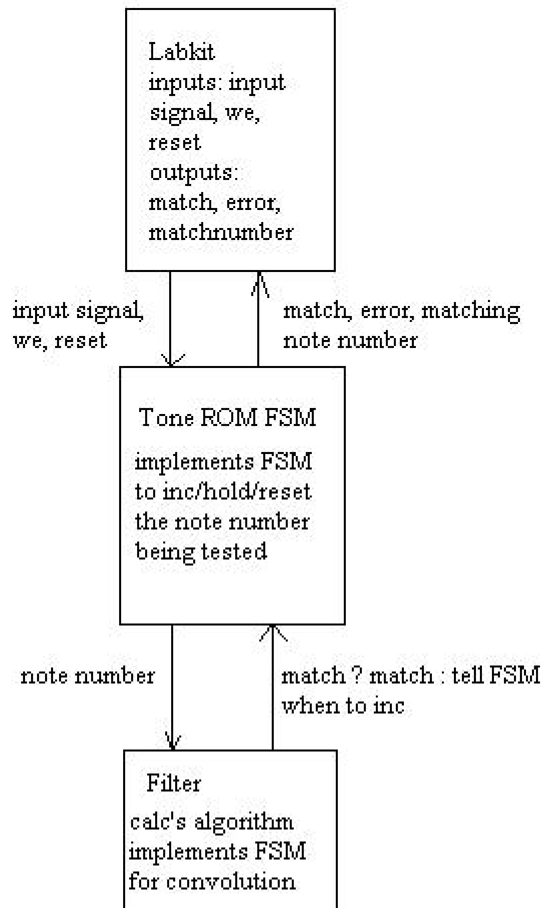

Figure 13: Block diagram for the entire part

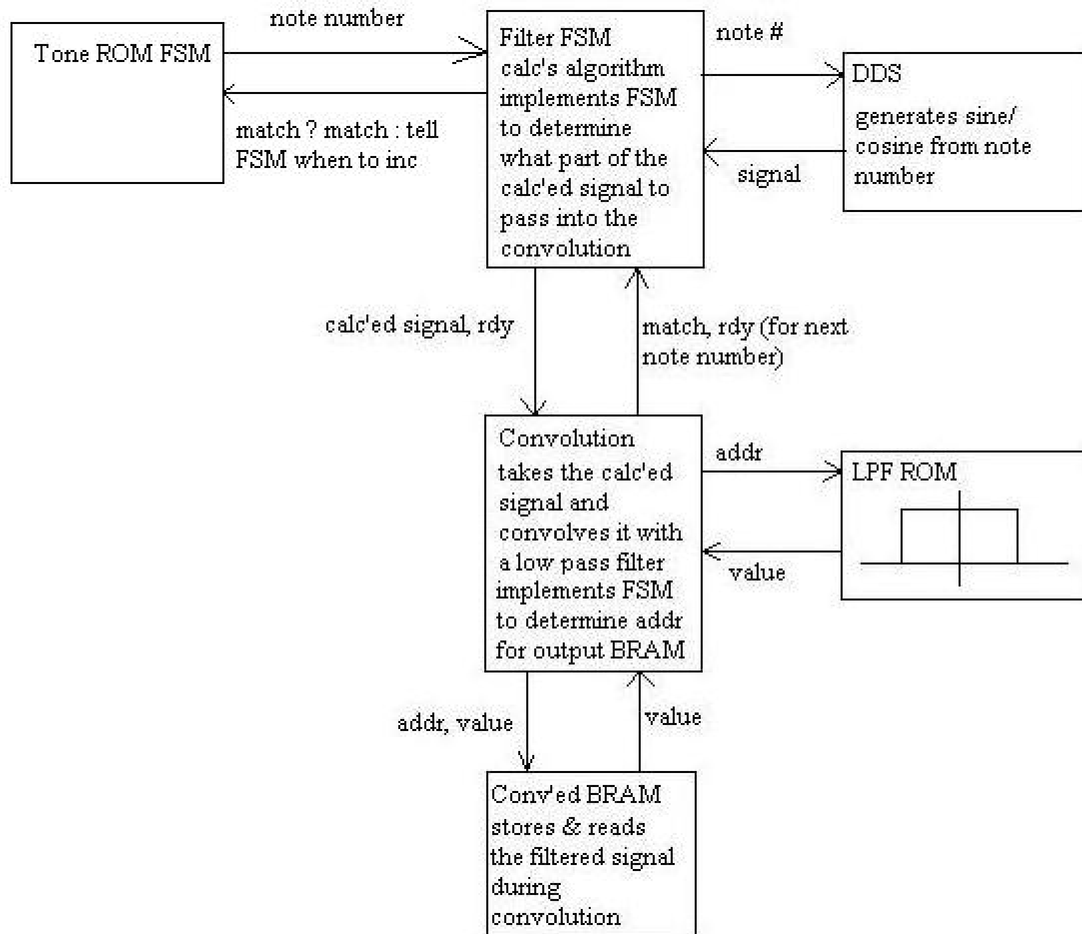

Figure 14: block diagram for the filter part

VERSION 1: {V(T)[SIN(W1-W2)T + SIN(W1+W2)T]} *LPF

The ideal for this version was the equation above. The Matlab outputs Figure 15, which essentially shows us that when there is a match, the output is critically damped, and should be testable. All the parts are incomplete and do not fully function, as parts left to be added were not implemented due to thinking the algorithm could not test user input signals. See Appendix 1.3.i-iii for the Matlab code.

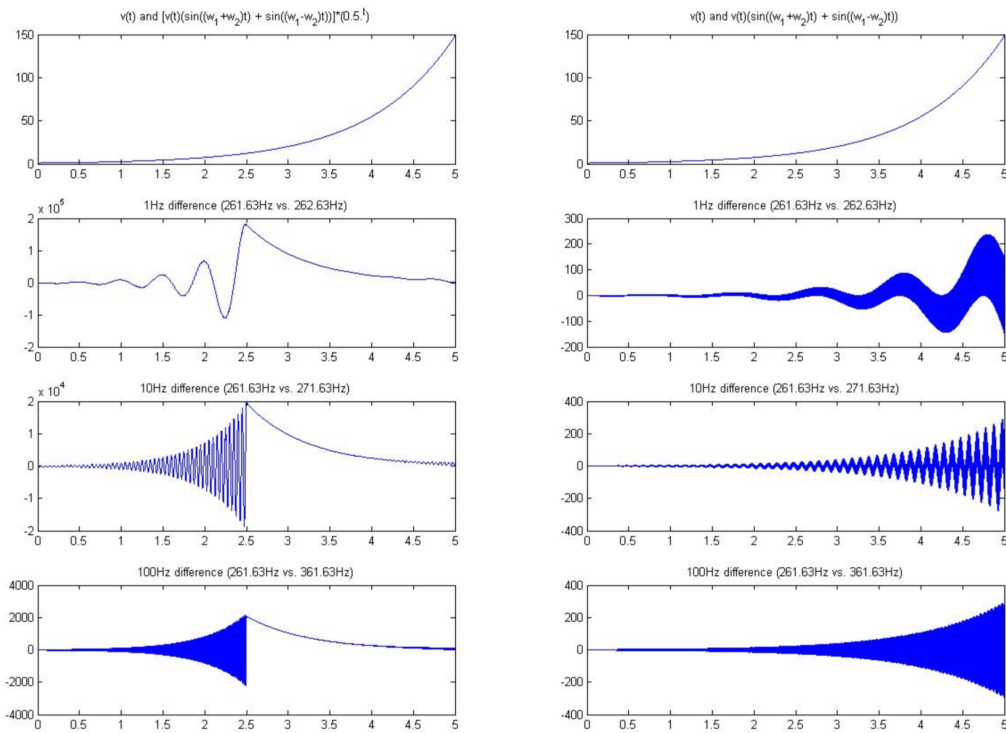

Figure 15: (l-r) z(t) = {v(t)[sin(w1-w2)t + sin(w1+w2)t]} ∗LPF and z(t) = {v(t)[sin(w1-w2)t + sin(w1+w2)t]}, where w1 is a test frequency against which the modules will try to match the input signal of frequency w2. (t-b) v(t), the modulation signal and z(t) with w1 and w2 having a difference of 1, 10, and 100Hz, top to bottom.

Labkit

As this was tested separately from the labkit part Roshni made, the same labkit as in lab 3 was reused. The clock speed was changed to 65MHz to avoid problems when combining. It borrows the DDS to make a test signal, which can be switched from middle C to one octave above. It also called the recorder, so that instead of a test signal, the user can input a signal. It also calls the filter FSM module with the output from the DDS. If the low pass filter has not completed, there will be no output. If it has, it will display that output.

DDS

This module is called twice, once in the labkit and once in the filter FSM. It takes in a note number and outputs both the sine and cosine signals for the matching frequency, working in much the same way as speaker, except its outputs are only 6 bits wide for testing purposes.

Filter FSM

FSM

This module is used to determine the current position of testing. It also calls the DDS again to create test signals against which the input signal can be compared, and the tone ROM to get the matching frequency value. It also does the calculations specified in the Matlab code in Appendix 1.3.iii, excepting the modulation signal.

The main FSM in this module has four states: initial check (initial point), check up, check down, and matched, which were to switch based on a match wire that was never implemented. When it's in the initial check state, it calls the filter module at middle C, which is note number and tone ROM address 40. Depending on whether it was too high or low, it would increase or decrease the frequency. When there was a match, it would turn on a ready signal and stop all other functions.

Convolution

This module implements a convolution with a low pass filter, also with an FSM. There are three states to this FSM, now, new and done, as seen in Figure 16.

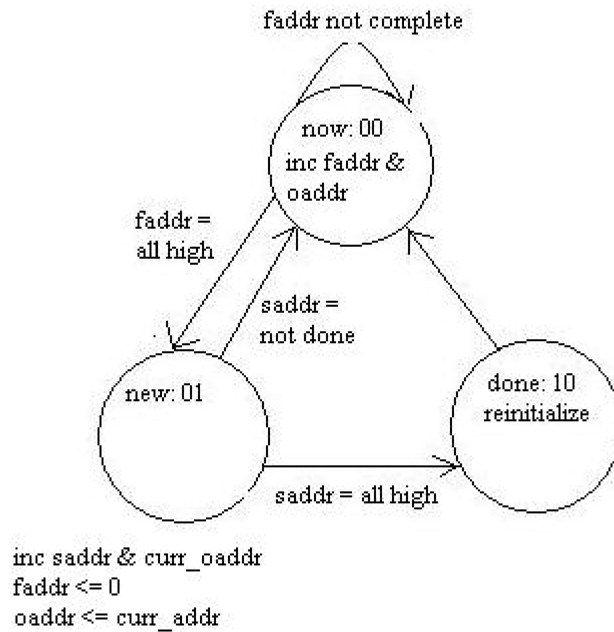

Figure 16: FSM for how the filter should work

The actual convolution process itself is shown in Tables 1 and 2. The lowpass filter was essentially a ROM with 16 high bits. When a reset is passed in or it is done, the address for the signal, the filter ROM, and the output signal BRAM are cleared. Otherwise, if the filter has not reached the end, keep the input signal address where it is, but increment the filter signal address (this is the equivalent of moving down a column). If the signal address has not finished, but the filter has, the module implements the equivalent of moving onto the next column in Table 2. Therefore, everything except the output addresses are re-initialized. The "current output address" keeps track of where the output address is (since it does not start at 0 every column), and the "output address" is reset to that value. If both the signal address and the filter address are done, then it signals a ready high back to filter_write so that the next note number can be passed in to restart the filtering.

While it's changing addresses, it's also writing into and reading from the output_sig BRAM. It takes the product of the filter and input signals at the determined addressed and adds it to the output from the BRAM at that address, then writes it back in, sort of like a feedback system.

	0	**1**	**2**	**3**
Input signal	1	2	3	4
Filter signal	1	1	1	

Table 1: values and positions of two to-be convolved signals

Output signal	**Column 0**	**Column 1**	**Column 2**	**Column 3**	**Output total**
0	I[0]*F[0]				1*1 = 1
1	I[0]*F[1]	I[1]*F[0]			1*1 + 2*1 = 3
2	I[0]*F[2]	I[1]*F[1]	I[2]*F[0]		1*1 + 2*1 + 3* 1 = 6
3		I[1]*F[2]	I[2]*F[1]	I[3]*F[0]	2*1 + 3*1 + 4*1 = 9
4			I[2]*F[2]	I[3]*F[1]	3*1 + 4*1 = 7
5				I[4]*F[2]	4*1 = 4

Table 2: the convolution and output, step-by-step

Filter ROM

This is a ROM that stores the low pass filter, which is just a series of high bits.

Output RAM

This is a BRAM that stores the convolved signal and outputs what is already in the BRAM, so that it can be added to the new product and stored back in, similar to the Output total column in Table 2.

VERSION 2: {SIN(W1)T-SIN(W2)T} ∗LPF

The idea behind this version was that taking a difference of the values of the input signal and test signal should come out as a constant if the two frequencies are relatively close together. See Figure 17 for the Matlab output.

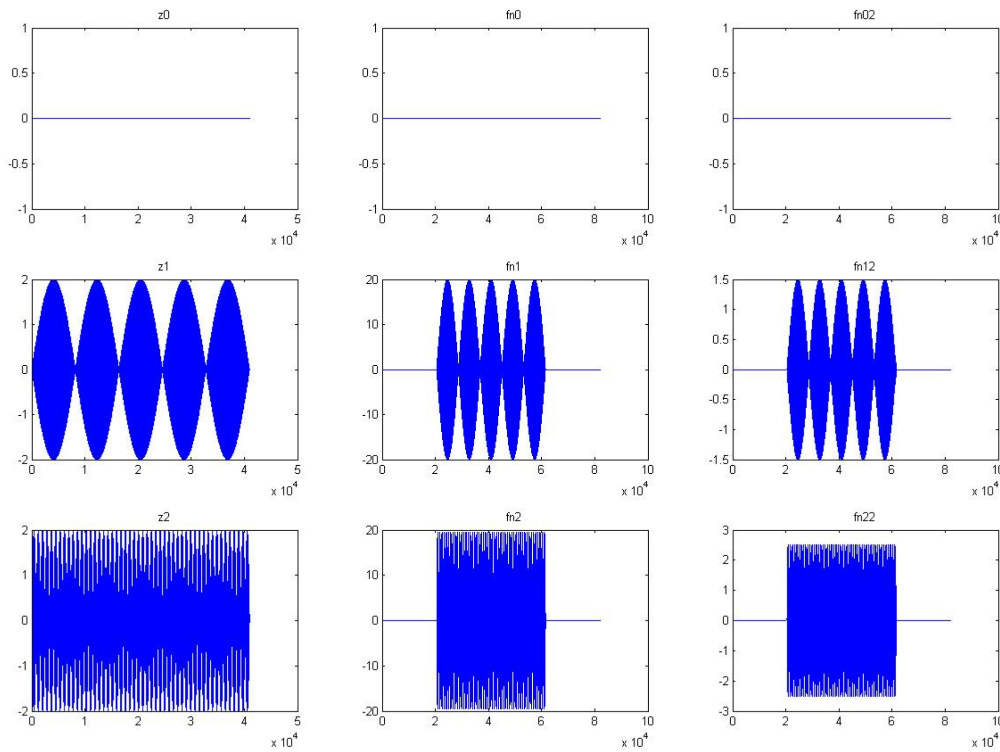

Figure 17: (l-r) z(t) = sin(w1t)-sin(w2t), fn(t) = z(t) ∗ LPF1, fn1(t) = z(t) ∗ LPF2
(t-b) w1-w2 = 0Hz, 1Hz, 10Hz. Again, w1 is a test frequency against which the modules are testing the input signal, with frequency w2.

Labkit

This was similar to the Labkit that Roshni made; it has the options of switching between a six bit signal and a 20 bit signal, as the 20 bit signal sounds better, but the six bit signal is easier to test with the logic analyzers. This option is only done by changing the code; it's not something the

user can change. It allows the user to either record something or to record a signal of a selected frequency. The buttons are set so that the user can select when the calculation process should be reset or started. The logic analyzer data, LEDs, and 16-hex display are also all assigned for purposes of testing; the 16-hex display should display the value of the matching note number.

Tone ROM FSM

This module was originally an FSM that was changed to be sequential for easier testing. It has three states: reset, increment, and hold, as can be seen in Figure 18. It starts with the lowest tone ROM address, which is the equivalent to the lowest key on a piano, and increments it when the filter modules have determined that particular frequency as not a match with the input signal. When there is a match, it should hold the same address until there is a reset or the user passes in a new input signal. If the addresses have reached 88 – the highest key on a piano – without a match, it should also stop and output an error.

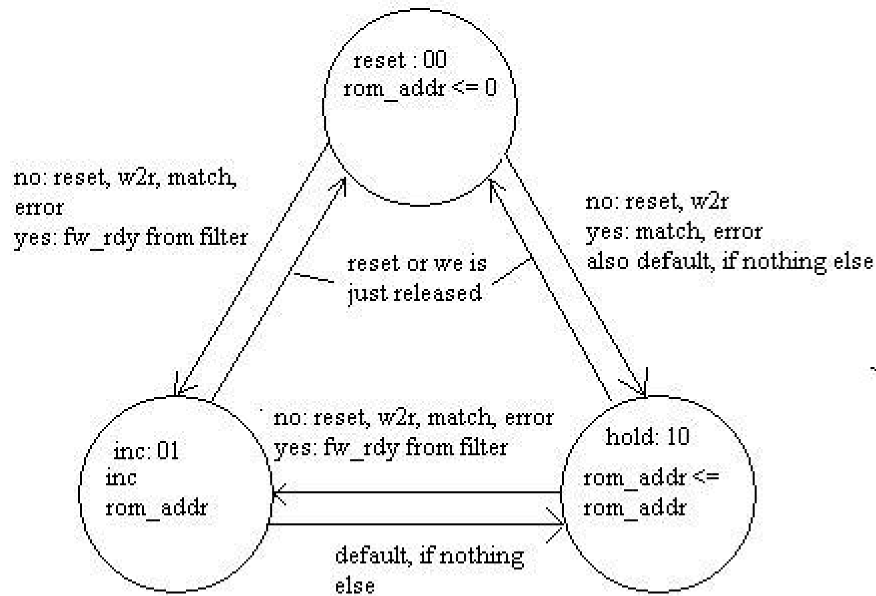

Figure 18: Tone ROM address FSM

This module also stores two useful impulses and two step functions. When the user first calls for the checking process to start, an impulse and a step function are created. The step function lasts as long as the button is held down. When the user releases that button, there is another impulse and step function.

Filtering

Read-Write

This module mainly uses an FSM to determine when to storing the differences of the input signal and the signal it's being tested against into a BRAM, and when to just read from the BRAM. It first generates a signal from the note number being tested, then follows the FSM in Figure 19 to determine what it should be doing. This module also tests for and holds the matching note number if there is a match, until there is a reset.

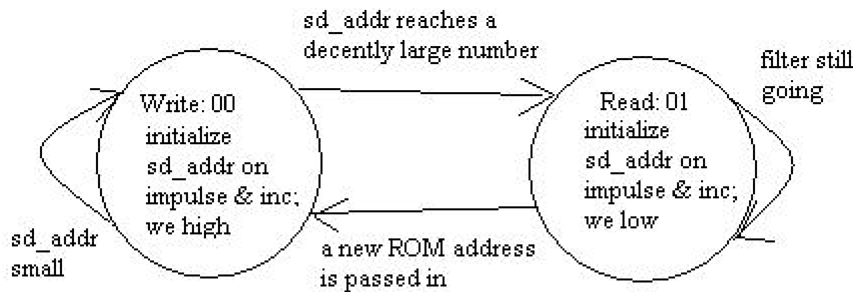

Figure 19: FSM for read-write process of the BRAM for the signal differences

Signal Difference RAM

This is a BRAM that stores the difference of the input signal and the signal it is being tested against.

Convolution

This module is better tested than its counterpart in Version 1, but it essentially does the same operations, with one exception. After the convolution process is done with a particular difference of signals, it goes through and samples various points along the filtered signal. If the difference of those points were relatively small (the exact number would be tightened during debugging), it would latch onto the ROM address and pass that out as the matching note number.

Filter ROM

This is a ROM that stores the low pass filter, which is just a series of high bits.

Output RAM

This is a BRAM that stores the convolved signal and outputs what is already in the BRAM, so that it can be added to the new product and stored back in, similar to the Output total column in Table 2.

*Version 3 {V(T)[SIN(W1T) *COS(W2T)]} *LPF*

This version is much the same as version 1, except it cancels out the sine of one frequency and cosine of another frequency. Therefore, if the input signal is called a sine, the project just has to create cosine waves (in practice, sine or cosine waves should be OK, since they are essentially the same and only differ with a phase shift). Matlab outputs Figure 20 as what we should be getting after passing the product of the sine, cosine, and a modulation signal through a low pass filter. See Appendix 3.3.i-iii.

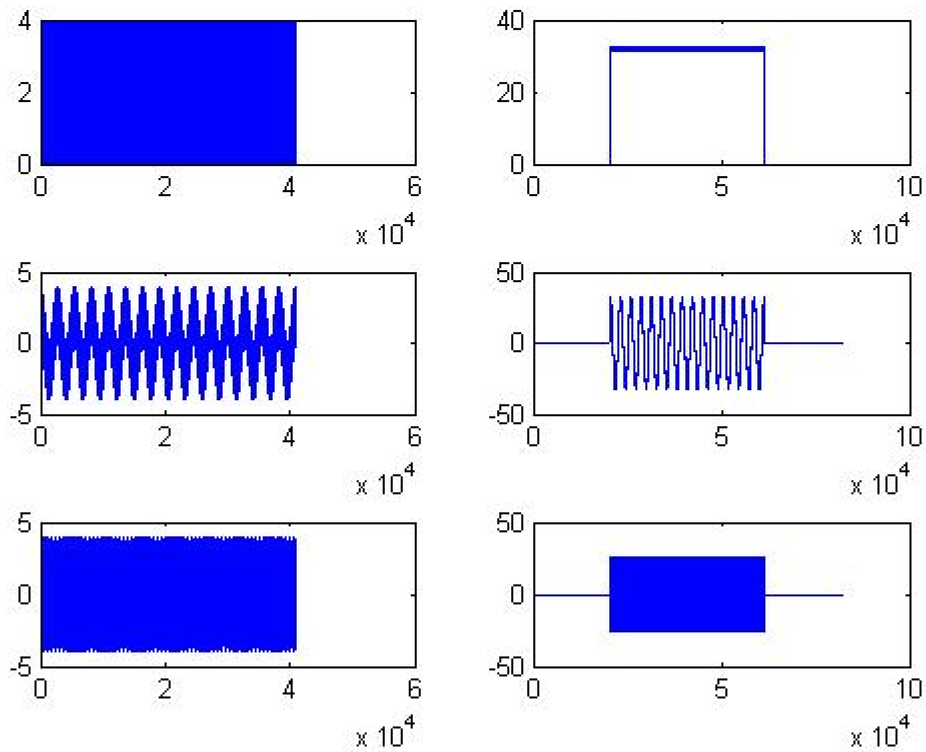

Figure 20: (l-r) {2v(t)[(sin(w1t)cos(w2t)}, {2v(t)[(sin(w1t)cos(w2t)} *LPF
(t-b) w1 and w2 differ by 0Hz, 3Hz, and w2 = 2(w1).

Labkit

This used the same labkit as in version 2, except that the DDS outputs both cosine and sine. The filtering modules take in the sine as the input signal.

DDS

This is similar to the DDS in version 1, except it outputs a 20 bit signal.

Tone ROM Control

This is identical to its counterpart in version 2, except it receives match and matching note number calls from the filter submodules.

Filtering

Read-Write FSM

This is very similar to its counterpart in version 2. Instead of subtracting the signals and storing their differences, this module multiplies the two signals and a constant, and stores product, which is why the product BRAM is larger than the difference BRAM. After the convolution is finished, it implements a count register. This count takes six samples from the filtered signal and checks if the sum of three samples is about the same as the sum of the other three samples. As the entire filtered signal will be a constant if the two signals match, it should be possible to tailor this part to be very accurate. This part should actually be a separate module by itself.

Product RAM

This is a BRAM that stores the product of the input signal with the signal it is being tested against and a constant. It is larger than the its counterpart Difference RAM in version 2 because the product is larger than the difference.

Convolution

This part is identical to its counterpart in version 2, except it doesn't sample the completed filtered sample

Filter ROM

This is a ROM that stores the low pass filter, which is just a series of high bits.

Output RAM

This is a BRAM that stores the convolved signal and outputs what is already in the BRAM, so that it can be added to the new product and stored back in, similar to the Output total column in Table 2.

TESTING & DEBUGGING

The audio/video components of the digital tuner were attacked by first completing the video portion, then the audio, and finally the entire Labkit was assembled.

The video component was very challenging initially, and behaved in an entirely unpredictable manner. The background staff (the first subcomponent of the video module which was tested) was extremely glitchy for no apparent reason. Even though this should have been enough of a sign to reevaluate the entire set up of the video component, more subcomponents of the video were added. A simple note sprite was added, and then a mouse cursor. Initially, the mouse cursor would not appear from the correctly generated ROM, so the arrow was replaced with a square sprite. Next, some of the character display strings were added. Every time a new component was added, however, the video seemed very flaky, and would suddenly stop working and then suddenly behave again. For example, after adding letters to the character display, the addition of strings of numbers caused the video to disappear. For two hours, the code was combed through, and suddenly upon recompilation, the numbers appeared.

This strange behavior of the video component eventually became entirely unacceptable (as it should have been from the beginning), when the tiniest changes, for example, to inputs to the video module which had no effect on the pixels, would cause the video signal to disappear. After discussing the situation with the teaching staff, the suggestion to slow down the clock of the system arose. The video was running at a clock of 130MHz, while the VGA was running at 65MHz, in order to allow two full cycles for reading from the ROMs. Once the entire system was changed to a 65MHz clock, some unnoticeable delays might occur when reading the ROMs, but the overall behavior of the system became much more stable.

After this challenge of the video clock was overcome, the addition of the remaining video code was added relatively smoothly. The only debugging necessary was the correct renaming of wires, the generation of ROMs from valid .coe files (ones that began with "memory_initialization_radix" as opposed to "memory_init*itit*ialization_radix"), and the proper declaration of inputs and outputs.

The video was easy to test, since one can easily see the screen under inspection. By playing with the video to ensure that the changes in the display were always the same as those anticipated by the tester, it was easy to determine whether or not the video was working correctly.

The audio portion of the recorder module was very straightforward, but was originally only recording and playing back every eighth sample, creating a very noisy input. This error was determined by simply looking more closely at the code. Also, originally the address of the BRAM was not being properly reset in the recorder module, which upon inspection was because the registers which kept track of transitions into record or playback were not included. Similarly, in the volume_control module, the code originally incremented or decremented the outputted volume every clock cycle that the button was pushed, causing drastic increases and decreases in the volume, since a human hand obviously keeps the buttons pushed for multiple clock cycles.

The remaining components of the labkit module, for example the action_priority, were relatively simple to test using test bench simulations. After assembling the entire labkit module, the only problems encountered dealt once again with the incorrect naming of wires and registers.

There were a lot of problems with the math behind the Verilog that has not been resolved. The original plan was to use a Fast Fourier Transform. Then we were given the suggestion of

using the method shown in Figure 21. After it was written out and being tested, we discovered that it could not work. {v(t)[sin(w1-w2)t + sin(w1+w2)t]} could not be implemented because it would require knowing the frequencies of both the test signal and input signal. Its expanded form, {v(t)[(sin(w1t)cos(w2t)-sin(w2t)cos(w1t)) + (sin(w1t)cos(w2t)+sin(w2t)cos(w1t))]}also could not be implemented because while the sines of both functions could be found, only the cosine of the test frequency could be found. Furthermore, this would only work if the two signals started and ended in phase; were they to be off, it would not match.

This was to have been implemented with an FSM as seen in Figure 21, which would have seen tweaking as necessary. That never happened, however, and the FSM was left incomplete.

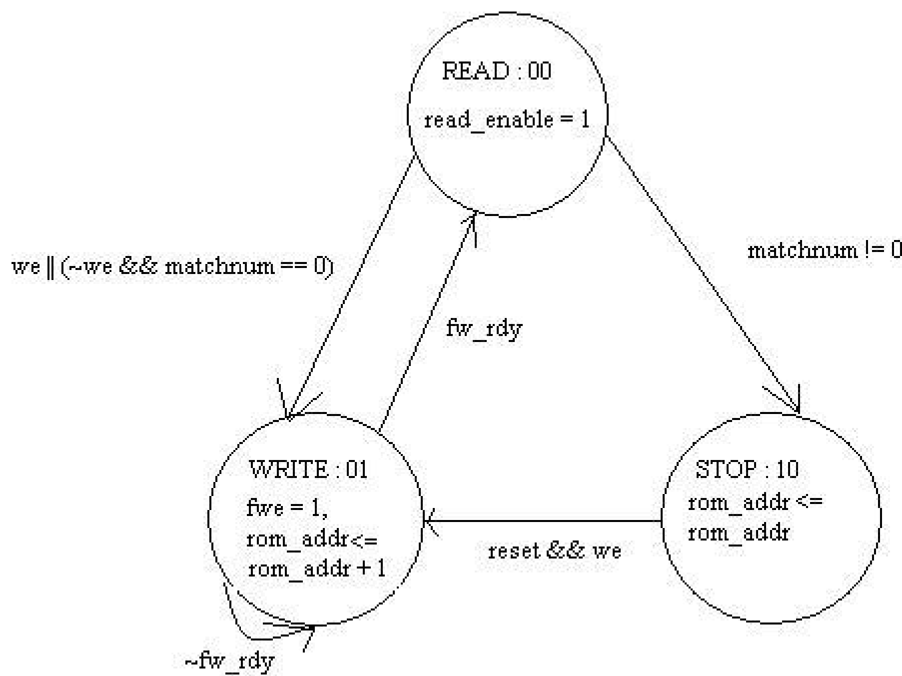

Figure 21: the FSM first implemented in the module.

A new method was suggested, as shown in the lower graph in Figure 22. It was completely mistaken until the last day – we thought it meant finding the difference of the values

of the input signal and test signal. After changing filter_write2 as described, the calculations were looked into more carefully and it was discovered that the difference that would be constant should be the difference of the *time* when the signal *values* were the same, not the difference of the signal values at the same time. See Figure 22. That had previously been determined unstable, because the values varied too much too be adequate.

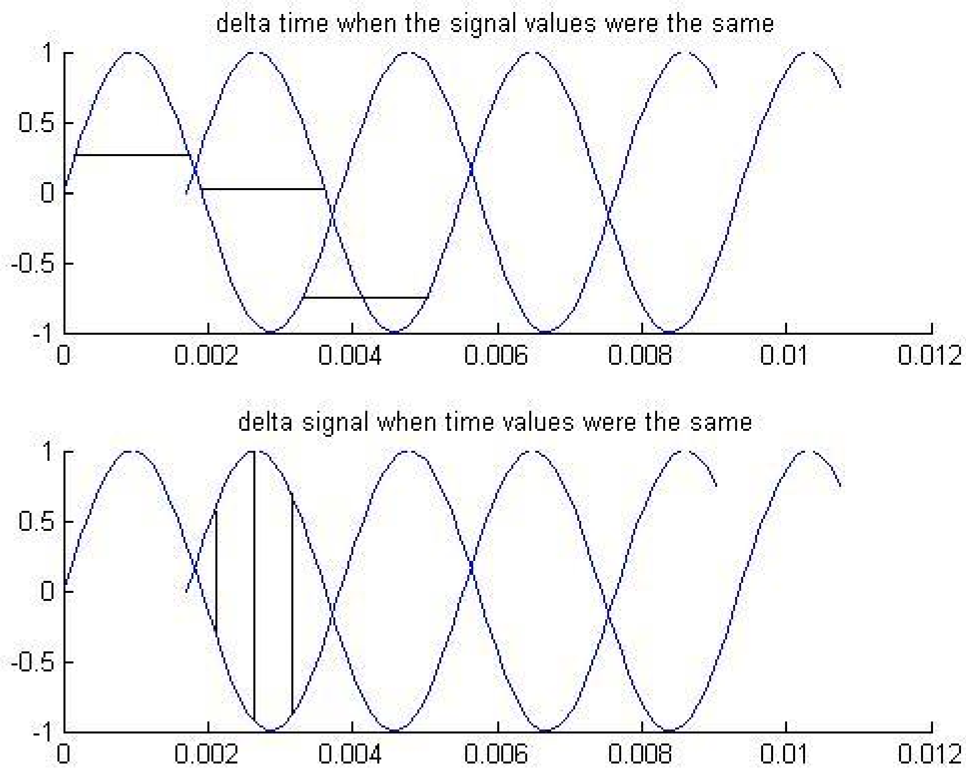

Figure 22: The logic behind Figure 17 – the top is the correct logic; the bottom is the incorrect logic we tried to implement. Note how the horizontal lines are all the same in length, whereas the vertical ones vary greatly in length.

Initially, the ROM address was passed into the logic analyzer and it was discovered that it was not incrementing, due to some sequential logic error. After it started incrementing, it neither stopped for an error or match signal – those were also passed into the logic analyzer, only to find

they were showing impulses instead of step functions cleared by reset. That was fixed with a latch on both wires.

While this paper was being written, an initial mistake was discovered. The original signal could in fact be simplified. {v(t)[(sin(w1t)cos(w2t)-sin(w2t)cos(w1t)) + (sin(w1t)cos(w2t)+sin(w2t)cos(w1t))]} = {2v(t)[(sin(w1t)cos(w2t)}, which was finally attempted but never completed. This should theoretically mean that after filtering, the difference of various samples in the output should be very small when the signals are of similar frequency. Unfortunately, there was not enough time to do more than draw a block diagram and write the initial code for it; no debugging was done. That can be found in Appendix 3.

Other than the incorrect use of the algorithms, the testing and debugging for everything else, including the FSMs, were fairly standard. Remnants are still in the code in the Appendix. Getting a note number to show was a big period – the logic analyzer showed that while the match and error values were going high as expected, they did not stay. That was easily fixed with a latch that was reset only when the user called a reset or a new input signal was passed in. The various FSMs also did not always operate the right way – instead of FSMs, a lot of them were changed into sequential logic so that it could be implemented stage by stage. This also made passing values into the logic analyzer and changing break points for individual variables easier.

In order for signals to be valid when passed into the logic analyzer, instead of the 20 bit signals used elsewhere in the project, six bit signals were generated instead to fit onto one eight bit probe. However, the 20 bit signals were still possible for when the two parts would be connected. In addition, to fix some errors that showed when the two parts ran at different clock speeds, the labkit clock was changed to also run at 65MHz.

Finally, for the algorithm, Matlab helped to determine exactly what kind of filter worked, how much of a difference was needed, and other such important factors.

CONCLUSION

The digital tuner was originally split into two components to be completed by two students. One such component was the audio/video input and output interface, and the other component was the internal playback module to determine the note number corresponding to an inputted (recorded) tone. The goal of the project was to have a tuner with the following functionality:

1. Audio:
    a. Record a tone inputted through a microphone connected to the FPGA
    b. Playback the most recently recorded tone through speakers connected to the FPGA
    c. Play a pure tone selected by the user using either the switches, the mouse, or the keyboard input
2. Video:
    a. Display the note information (location on the staff, note name, note number, and frequency) of a tone selected by the user using the switches, mouse, or keyboard
    b. Display the note information (determined by the internal playback module) of the most recently recorded tone on the VGA display

Currently, the tuner has all of the above functionality except for that described by 2b. The tuner will perform the record, playback, and playtone actions but cannot determine the note number or frequency of an inputted tone.

The most intuitive way of determining the frequency of a recorded tone would be by using the Fast Fourier Transform module on the FPGPAs. However, this tuner was aiming to determine the frequency of notes from 27Hz to 4186Hz, i.e., some very low frequencies, which

would not necessarily be easily discerned using the FFT. Instead, a manipulation of sine and cosine functions using basic trigonometric properties was considered to compare signals with two different frequencies:

$$\sin(w1-w2) + \sin(w1+w1) = \sin(w1)\cos(w2) +/- \sin(w2)\cos(w1)$$

However, implementing this sort of function would require knowing the frequencies beforehand, which is contradictory to the goal of determining the frequency. The other method of determining the frequency would involve taking the difference in addresses with the same data in order to determine if there was a constant phase shift between two signals or instead a frequency difference. This method was discovered at a time too late to implement it, but it could be implemented to complete the functionality of the digital tuner.

Even if this functionality had been present, another problem would have arisen when integrating the two separate components of the digital tuner. The audio/video input/output module used a significant number of ROMs, and the internal module used a reasonable number of RAMs. Together, there would not have been enough BRAMs on the FPGA to implement the entire design. The input/output module used 142 and the internal used 84 out of a total possible 144 BRAMs. The total usage of the BRAMs would have been over 30% more than could have been accommodated. A possible solution to this problem would have been to eliminate some of the ROMs used in creating the video display by using sprites instead of images from ROMs for the mouse cursor and the note, or perhaps recalculating the frequency increment and the ylocations for each note number. This would have sacrificed time, but would have saved space.

Once the basic functionality of the digital tuner is secured, various extensions can be made to the tuner. For example, it can analyze simultaneous notes (for example a chord) or a string of notes (perhaps an arpeggio or another melody). In addition, the tuner could be extended

to determine tempo. Also, the tuner could take user input of which note they expected to record and give feedback about the comparison of the actual frequency versus the desired frequency.

APPENDICES

Appendix A: labkit.v

```
module labkit(...)
   ...


   // use FPGA's digital clock manager to produce a 65 Mhz clock from 27 Mhz
   wire clock_65mhz_unbuf,clock_65MHz;
   DCM vclk1(.CLKIN(clock_27mhz),.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   clk = clock_65MHz;
   wire power_on_reset;
   SRL16 reset_sr (.D(1'b0), .CLK(clk), .Q(power_on_reset),
               .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
   defparam reset_sr.INIT = 16'hFFFF;
   wire user_reset;
   debounce dbreset(1'b0,clk,~button_enter,user_reset);
   wire reset = power_on_reset | user_reset;

   wire [7:0] pixel;
   wire       blank;
   wire       pix_clk = 1;
   wire [10:0] hcount;
   wire [9:0] vcount;
   wire all_pix;
   wire [11:0] mx,my; // mouse locations
   wire [2:0] btn_click;  // mouse button click left-middle-right
   wire [7:0] ascii;  // keyboard output
   wire ascii_ready;  // keyboard output is ready
   wire [6:0] note_num;  // the note number of the current note being
analyzed
   wire [1:0] action;  // the current action of the tuner:
                       // 11=record,10=playback,01=playtone
                       // determined by the buttons

   wire [1:0] ninput;  // the method of note input:
                       // 11=switches,10=mouse,01=keyboard
   wire [6:0] note_num_pt;  // note_num from playtone
   wire [6:0] note_num_pb;  // note_num from playback
   wire [6:0] nswitch;  // note num determined by switches
   wire [15:0] nkbd_ascii; //note num in ascii determined by keyboard
   wire audio_ready;  // signal from ac97
   wire [19:0] to_ac97_data_pt,to_ac97_data_pb, to_ac97_data,from_ac97_data;
   wire rec,pb,pt,up,down;
   wire we;
   wire [19:0] din;
   wire [19:0] dout;
```

```
   wire [15:0] addr;
   wire [3:0] vol;

   // set up the VGA display outputs
   assign     vga_out_red = pixel;  // black and white display
   assign     vga_out_green = pixel;
   assign     vga_out_blue = pixel;
   assign     vga_out_blank_b = ~blank;
   assign     vga_out_pixel_clock = clk;   // vga pixel clock = 65mhz

   wire pix_clk2;  //dummy pixel clock generated by vga_sync
   vga_sync
vga1(clk,vga_out_hsync,vga_out_vsync,hcount,vcount,pix_clk2,blank);

   assign nswitch = switch[6:0];

   // keyboard input
   ps2_ascii_input kbd(clk,reset,keyboard_clock,keyboard_data,
                       ascii,ascii_ready);
   // keep track of two most recent keyboard inputs
   // pressing “enter” resets the inputs
   reg [7:0] curr_ascii;
   reg [7:0] old_ascii;
   always @ (posedge clk) begin
      if (ascii_ready && (ascii == 8'h0D)) begin
         old_ascii <= " ";
          curr_ascii <= " ";
      end
       else if (ascii_ready) begin
          old_ascii <= curr_ascii;
          curr_ascii <= ascii;
       end
   end
   assign nkbd_ascii = {old_ascii,curr_ascii};

   // mouse input

   ps2_mouse_xy m1(clk, reset, mouse_clock, mouse_data, mx, my, btn_click);


   debounce drec(reset,clk,~button3,rec);
   debounce dpb(reset,clk,~button2,pb);
   debounce dpt(reset,clk,~button1,pt);
   debounce dup(reset,clk,~button_up,up);
   debounce ddown(reset,clk,~button_down,down);

   // determine the current action using the buttons
   // also select which note_num to display and which
   // to_ac97_data to send

   action_priority ap(clk,rec,pb,pt,action,
                      note_num_pb,note_num_pt,note_num,
                      to_ac97_data_pb,to_ac97_data_pt,to_ac97_data);
   // record/playback
   recorder r(clk, reset, action, audio_ready, from_ac97_data,
to_ac97_data_pb,
                addr, din, dout, we);
```

```
   // bram with most recently recorded tone

   lab3bram b(addr,clk,din,dout,we);

   // play a user selected tone

   playtone playt(clk,action,ninput,nswitch,my,
                  btn_click,nkbd_ascii,note_num_pt);

   // display the currently desired note

   video v(clk,reset,action,ninput,hcount,vcount,pix_clk,
           note_num,old_ascii,curr_ascii,mx,my,all_pix);

   // allow the user to control the volume of the note using the
   // up and down buttons

   volume_control vc(clk,reset,4'd10,up,down,vol);

   // interface with the ac97

   audio myaudio(clk, reset, from_ac97_data, to_ac97_data,
             audio_ready, audio_reset_b, ac97_sdata_out, ac97_sdata_in,
             ac97_synch, ac97_bit_clock,vol);

   // get a tone made by the dds from the note_num

   speakers s(clk,note_num,audio_ready,to_ac97_data_pt);

   assign     pixel = blank ? 8'b0
                  : ({8{all_pix}});
endmodule
```

Appendix B: debounce.v

```
module debounce (reset, clock_65mhz, noisy, clean);
   input reset, clock_65mhz, noisy;
   output clean;

   reg [19:0] count;
   reg new, clean;

   always @(posedge clock_65mhz)
     if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
     else if (noisy != new) begin new <= noisy; count <= 0; end
     else if (count == 650000) clean <= new;
     else count <= count+1;

endmodule
```

Appendix C: vga_sync.v

```
//
```

```
// File:   vga_sync.v
// Date:   04-Nov-05
// Author: C. Terman / I. Chuang
//
// MIT 6.111 Fall 2005
//
// Verilog code to produce VGA sync signals (and blanking) for 1024x768
screen
//

module vga_sync(clk,hsync,vsync,hcount,vcount,pix_clk,blank);

   input clk;     // 65Mhz
   output hsync;
   output vsync;
   output [10:0] hcount;
   output [9:0] vcount;
   output   pix_clk;
   output   blank;

   wire     en = 1;
   wire     pix_clk = ~en;

   //****************************************************************
   //****************************************************************
   //***
   //***  Sync and Blanking Signals
   //***
   //****************************************************************
   //****************************************************************

   reg            hsync,vsync,hblank,vblank;
   reg [10:0]     hcount;      // pixel number on current line
   reg [9:0]      vcount;      // line number

   // display 1024 pixels per line
   wire     hsyncon,hsyncoff,hreset,hblankon;
   assign   hblankon = en & (hcount == 1023);
   assign   hsyncon = en & (hcount == 1047);
   assign   hsyncoff = en & (hcount == 1183);
   assign   hreset = en & (hcount == 1343);

   wire     blank =  (vblank | (hblank & ~hreset));    // blanking => black
   //wire   blank = vblank | hblank;

   // display 768 lines
   wire     vsyncon,vsyncoff,vreset,vblankon;
   assign   vblankon = hreset & (vcount == 767);
   assign   vsyncon = hreset & (vcount == 776);
   assign   vsyncoff = hreset & (vcount == 782);
   assign   vreset = hreset & (vcount == 805);

   // sync and blanking
   always @(posedge clk) begin
      hcount <= en ? (hreset ? 0 : hcount + 1) : hcount;
      hblank <= hreset ? 0 : hblankon ? 1 : hblank;
      hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;   // hsync is active low
```

```
      vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
      vblank <= vreset ? 0 : vblankon ? 1 : vblank;
      vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;   // vsync is active low
   end

endmodule
```

Appendix D: recorder.v

```
module recorder(clk, reset, action, ready, from_ac97_data, to_ac97_data,
                addr, din, dout, we);
   input clk;              // 27mhz system clock
   input reset;                    // 1 to reset to initial state
   input [1:0] action;                 // 1 for playback, 0 for record
   input ready;                    // 1 when AC97 data is available
   input [19:0] from_ac97_data;     // 8-bit PCM data from mic
   output [19:0] to_ac97_data;      // 8-bit PCM data to headphone

   output [15:0] addr;              // address to be sent to the bram
   output [19:0] din;               // output to write into the bram
   input [19:0] dout;               // data received from the bram
   output we;                             // write-enable for the bram

   reg [15:0] addr;
   reg [19:0] din;
   reg we;

   // detect clock cycle when READY goes 0 -> 1

   // f(READY) = 48khz
   wire new_frame;
   reg old_ready;
   always @ (posedge clk) old_ready <= reset ? 0 : ready;
   assign new_frame = ready & ~old_ready;

   // use reset_addr to decide whether or not playback has changed
   // if it has, reset the address inside the big always block
   wire record,playback;
   wire reset_addr_p;
   wire reset_addr_r;
   reg old_playback;
   reg old_record;
   assign   record = (action==2'b11) ? 1 : 0;
   assign   playback = (action==2'b10) ? 1 : 0;
   always @ (posedge clk)  begin
      old_record <= reset ? 0 : record;
      old_playback <= reset ? 0 : playback;
   end
   assign reset_addr_p = (playback != old_playback);
   assign reset_addr_r = (record != old_record);

   // recorder counter which counts from 0 to 7 and reg to keep track of
   // the maximum address
   reg [15:0] max_addr = 16'b0;
   reg [19:0] to_ac97_data;
```

```
   always @ (posedge clk) begin
      if (reset_addr_p || reset_addr_r) addr <= 16'b0;
      else if (new_frame) begin
       // just received new data from the AC97
          if (action == 2'b10) begin // we're in playback mode
              we <= 0;
                    addr <= addr + 16'b1;
                  // the data from the bram is sent to the ac97
                 to_ac97_data <= dout;
                  // if we're at the maximum address, start over again
                 if (addr == max_addr) begin
                     addr <= 16'b0;
              end
          end
          else if (action == 2'b11) begin // we're in record mode
                  max_addr <= addr;
                  we <= 1;
                  din <= from_ac97_data;
                  addr <= addr + 16'b1;
          end
      end
   end

endmodule
```

Appendix E: playtone.v

```
module
playtone(clk,action,ninput,nswitch,nmy,nbtn_click,nkbd_ascii,note_num);
   input clk;
   input [1:0] action;  // the tuner's current action
   output [1:0] ninput;  // output the highest priority input for video
display
   input [6:0] nswitch;  // note selected by switches
   input [11:0] nmy;  // the location selected by the mouse
   input [2:0] nbtn_click;  // is there a mouse click?
   input [15:0] nkbd_ascii;  // ascii keyboard input
   output [6:0] note_num;  //

   reg [6:0] note_num;
   reg [1:0] ninput = 2'b0;

   wire [6:0] nkbd;  // keyboard ascii input converted to a number
                     // after first ensuring that the keyboard inputs are
                     // in fact numbers

   assign nkbd = ((nkbd_ascii[15:8] >= 8'd48) && (nkbd_ascii[15:8] <= 8'd57))
&&
                 ((nkbd_ascii[7:0] >= 8'd48) && (nkbd_ascii[7:0] <= 8'd57)) ?
                 ((nkbd_ascii[15:8] - 8'd48) * 10 + (nkbd_ascii[7:0] -
8'd48)) :
                 0;

   // at every posedge, determine which input is highest priority

   always @ (posedge clk) begin
```

```
      if (action==2'b01) begin // only bother to continue if we want to
playtone
          if (nswitch) begin
              note_num <= nswitch;
              ninput <= 2'b11;
          end
          else if (nbtn_click) begin
              // if we want the mouse click input,
              // convert the mouse click to a note number based on
              // current location

              note_num <= (nmy <= 91) ? 7'd59 :
                          (nmy >= 91 && nmy <= 95) ? 7'd57 :
                          (nmy >= 95 && nmy <= 131) ? 7'd56 :
                          (nmy >= 131 && nmy <= 135) ? 7'd54 :
                          (nmy >= 135 && nmy <= 168) ? 7'd52 :
                          (nmy >= 168 && nmy <= 172) ? 7'd51 :
                          (nmy >= 172 && nmy <= 206) ? 7'd49 :
                          (nmy >= 206 && nmy <= 210) ? 7'd47 :
                          (nmy >= 210 && nmy <= 244) ? 7'd45 :
                          (nmy >= 244 && nmy <= 248) ? 7'd44 :
                          (nmy >= 248 && nmy <= 288) ? 7'd42 :
                          (nmy >= 288 && nmy <= 328) ? 7'd40 :
                          (nmy >= 328 && nmy <= 366) ? 7'd39 :
                          (nmy >= 366 && nmy <= 370) ? 7'd37 :
                          (nmy >= 370 && nmy <= 406) ? 7'd35 :
                          (nmy >= 406 && nmy <= 410) ? 7'd33 :
                          (nmy >= 410 && nmy <= 444) ? 7'd32 :
                          (nmy >= 444 && nmy <= 448) ? 7'd30 :
                          (nmy >= 448 && nmy <= 482) ? 7'd28 :
                          (nmy >= 482 && nmy <= 486) ? 7'd27 :
                          (nmy >= 486 && nmy <= 520) ? 7'd25 :
                          (nmy >= 520 && nmy <= 524) ? 7'd23 :
                          (nmy >= 524) ? 7'd21 :
                          7'd0;
            ninput <= 2'b10;
          end
          else if (nkbd) begin

              note_num <= nkbd;
              ninput <= 2'b01;
          end
       end
   end
endmodule
```

Appendix F: speakers.v

```
// for playtone, convert the selected note number into data for the ac97
module speakers(clk,note_num,audio_ready,to_ac97_data);
   input clk;  //65 MHz
   input [6:0] note_num;  // user selected note number
   input audio_ready;  // high if the ac97 is ready
   output [19:0] to_ac97_data;  // data for ac97

   wire [17:0] freq;          // phase increment value (sets output freq)
```

```
   wire        we;            // write enable
   wire [4:0]  dds_a = 5'b0;  // used for multiple output channels (not here)
   wire        rfd;           // not used by DDS, always high
   wire        rdy;           // high when output samples ready
   //wire [15:0]  sine,cosine;      // quadrature outputs of DDS
   wire [16:0] sine;
   reg             clk_dds;         // DDS output clock

   // the dds

   dds_2 dds1(freq,we,dds_a,clk,clk_dds,rdy,sine);

   reg [17:0]  old_freq;
   always @(posedge clk) old_freq <= freq;
   assign      we = ~(freq == old_freq);

   wire [19:0] to_ac97_data;

   assign to_ac97_data = {sine,3'b0};
   // generate DDS clock from ac'97 audio_ready signal
   reg      old_ready;
   always @(posedge clk)
     begin
      old_ready <= audio_ready;
      clk_dds <= audio_ready & ~old_ready;      // one cycle delayed ok
     end
   wire [14:0] freqinc;
   // look up the correct frequency increment for that note number

   freqincrom fi(note_num,clk,freqinc);
   assign freq = {3'b000,freqinc};
endmodule
```

Appendix G: video.v

```
// take the notenumber, action, user selected in put,
// ascii values from keyboard, and mouse location to
// create the pixel for the vga display
module video(clk,reset,action,ninput,hcount,vcount,pix_clk,
             note_num,last_ascii,ascii,mx,my,all_pix);
   input clk;  // 65MHz
   input reset;
   input [1:0] action;  // current mode of tuner
   input [1:0] ninput;  // where the note input is from
   input [10:0] hcount;
   input [9:0] vcount;
   input pix_clk;
   input [6:0] note_num;  //note num of note’s info to display
   input [7:0] last_ascii; // last keyboard input
   input [7:0] ascii;  // current keyboard input
   input [11:0] mx;  // mouse x location
   input [11:0] my;  // mouse y location
   output all_pix;

   // generate the mouse display

   wire mpix;
```

```
mouse_disp2 m(clk,mx,my,hcount,vcount,pix_clk,mpix);

// generate the note display, including the note itself, an accidental
// symbol, and any "va" notation
wire npix;
note2 n1(clk,note_num,hcount,vcount,pix_clk,npix);

// the background image of the staff
wire rpix;
staff_disp vr(clk,hcount,vcount,pix_clk,rpix);

// now add the character displays
// there are six:
// the note name
// the note number
// the note frequency
// the current action
// the current user selected input
// the latest keyboard input

// in order to turn the number,letter,action,input,freq,etc into strings,
// call several submodules to read from roms and output strings, meaning
// outputs in ascii
wire [15:0] nnstring_rom;
wire [31:0] fstring_rom;
wire [55:0] lstring_rom;
wire [63:0] actionstr,ninputstr;
get_pri_strings gp(action,ninput,actionstr,ninputstr);
get_strings g(clk,note_num,nnstring_rom,fstring_rom);
get_letter gl(clk,note_num,lstring_rom);

// display the note letter
wire lpix;
wire [10:0] lx = 11'd0;
wire [9:0] ly = 10'd600;
wire [159:0] lstring = {"Note Letter: ",lstring_rom};
char_string_display notelet(clk,hcount,vcount,lpix,lstring,lx,ly);
defparam notelet.NCHAR = 20;
defparam notelet.NCHAR_BITS = 5;

// display the note number
wire numpix;
wire [10:0] numx = 11'd0;
wire [9:0] numy = 10'd650;
wire [119:0] numstring = {"Note Number: ",nnstring_rom};
char_string_display notenum(clk,hcount,vcount,numpix,numstring,numx,numy);
defparam notenum.NCHAR = 15;
defparam notenum.NCHAR_BITS = 4;

// display the note frequency
wire fpix;
wire [10:0] fx = 11'd0;
wire [9:0] fy = 10'd700;
wire [159:0] fstring = {"Frequency (Hz): ",fstring_rom};
char_string_display freqdisp(clk,hcount,vcount,fpix,fstring,fx,fy);
defparam freqdisp.NCHAR = 20;
defparam freqdisp.NCHAR_BITS = 5;
```

```
   // display the action
   wire actpix;
   wire [10:0] actx = 11'd512;
   wire [9:0] acty = 10'd600;
   wire [127:0] actstring = {"Action: ",actionstr};
   char_string_display actdisp(clk,hcount,vcount,actpix,actstring,actx,acty);
   defparam actdisp.NCHAR = 16;
   defparam actdisp.NCHAR_BITS = 5;

   // display the input
   wire inpix;
   wire [10:0] inx = 11'd512;
   wire [9:0] iny = 10'd650;
   wire [119:0] instring = {"Input: ",ninputstr};
   char_string_display indisp(clk,hcount,vcount,inpix,instring,inx,iny);
   defparam indisp.NCHAR = 15;
   defparam indisp.NCHAR_BITS = 4;

   // display the keyboard input
   wire kbdpix;
   wire [10:0] kbdx = 11'd512;
   wire [9:0] kbdy = 10'd700;
   wire [143:0] kbdstring = {"Keyboard Input: ",last_ascii,ascii};
   char_string_display kbddisp(clk,hcount,vcount,kbdpix,kbdstring,kbdx,kbdy);
   defparam kbddisp.NCHAR = 18;
   defparam kbddisp.NCHAR_BITS = 5;

   // now combine all of the pixels generated by the various components of
the
   // video display:  the mouse, staff, note, strings, etc.
   wire all_pix;
   assign all_pix = (rpix && npix && mpix && fpix && numpix && kbdpix &&
                     lpix && actpix && inpix);
endmodule
```

Appendix H: note2.v

```
module note2(clk,note_num,hcount,vcount,pix_clk,pixel);
   input clk;           // video clock
   input [6:0] note_num;
   input [10:0] hcount; // current x,y location of pixel
   input [9:0] vcount;
   input       pix_clk; // pixel clock
   output pixel;  // pixel value output

   // using submodule get_y, we retrieve the following information:
   // the ypos on the vga display,
   // is there an accidental?  If so, what kind?
   // is there "va" notation needed?  If so, how many octaves?

   wire [9:0] ypos;

   wire [1:0] acc;
   wire [1:0] va;
   get_y gy(clk,note_num,ypos,acc,va);
```

```
   // draw the actual circle of the note from a rom

   wire npix;
   note n(clk,11'd512,ypos,hcount,vcount,pix_clk,npix);

   // draw a sharp symbol from a rom

   wire spix;
   wire srompix;
   wire [9:0] yposs;
   assign yposs = ypos - 16;
   sharp s(clk,11'd492,yposs,hcount,vcount,pix_clk,srompix);
   assign spix = acc ? srompix : 1;

   // make the character display of the va notation

   wire [15:0] vanum;
   assign vanum = (va == 2'b11) ? "24" :
                  (va == 2'b10) ? "16" :
                     (va == 2'b01) ? " 8" :
                     "00";
   wire vapix;
   wire vastrpix;
   wire [10:0] vax = 11'd512;
   wire [9:0] vay = (note_num > 44) ? 10'd40 : 10'd560;
   wire [31:0] vastring = {vanum,"va"};
   char_string_display_small vatext(clk,hcount,vcount,vastrpix,
                                    vastring,vax,vay);
   defparam vatext.NCHAR = 4;
   defparam vatext.NCHAR_BITS = 3;
   assign vapix = va ? vastrpix : 1;

   // if the note is middle c, add a line through the note

   wire mclinepixtemp,mclinepix;
   line mcline(clk,hcount,vcount,mclinepixtemp);
   assign mclinepix = (note_num == 7'd40) ? mclinepixtemp : 1;

   wire temppix;
   assign temppix = (npix && spix && vapix && mclinepix);
   // only output the pixels anded together if the note_num is valid

   assign pixel = (note_num > 0 && note_num < 89) ? temppix : 1;
endmodule
```

Appendix I: note.v

```
// draw a note from a rom
module note(clk,x,y,hcount,vcount,pix_clk,pixel);
   input clk;            // video clock
   input [10:0] x,hcount;      // current x,y location of pixel
   input [9:0] y,vcount;
   input       pix_clk; // pixel clock
   output pixel;  // pixel value output

   // the memory address is hcount + vcount * 32
   reg in_rom = 0;
```

```
   reg [9:0] raddr;  //row address
                     // adds the correct number to adjust for vcount
   always @(posedge clk) begin
     if(pix_clk) begin
        if (((hcount >= x) && (hcount <= (x+32))) &&
            ((vcount >= y) && (vcount <= (y+32)))) begin
           in_rom <= 1;
           raddr <= (hcount==x & vcount==y) ? 0
                    : (hcount==x & pix_clk) ? raddr + 32 : raddr;
        end
        else
           in_rom <= 0;
     end
   end
   //current location in rom
   wire [9:0] addr = in_rom ? (hcount - x + raddr) : 0;
   // latch the addr value

   reg [9:0]      addr_reg;
   always @(posedge clk) addr_reg <= pix_clk ? addr : addr_reg;
   wire rom_pix;

   // instantiate the image rom
   noterom the_note(addr_reg,clk,rom_pix);
   wire pixel;
   assign pixel = in_rom ? rom_pix : 1;
endmodule
```

Appendix J: sharp.v

```
module sharp(clk,x,y,hcount,vcount,pix_clk,pixel);
   input clk;           // video clock
   input [10:0] x,hcount;     // current x,y location of pixel
   input [9:0] y,vcount;
   input       pix_clk; // pixel clock
   output pixel;  // pixel value output

   // the memory address is hcount + vcount * 16
   reg in_rom = 0;
   reg [9:0]      raddr;  //row address?
                          //adds the correct number to adjust for vcount
   always @(posedge clk) begin
     if(pix_clk) begin
        if (((hcount >= x) && (hcount <= (x+16))) &&
            ((vcount >= y) && (vcount <= (y+64)))) begin
            in_rom <= 1;
            raddr <= (hcount==x & vcount==y) ? 0
                    : (hcount==x & pix_clk) ? raddr + 16 : raddr;
        end
        else
           in_rom <= 0;
     end
   end
   //current location in rom
```

```
   wire [9:0]     addr = in_rom ? (hcount - x + raddr) : 0;
   reg [9:0]      addr_reg;

   always @(posedge clk) addr_reg <= pix_clk ? addr : addr_reg;
   wire rom_pix;
   // instantiate the image rom
   sharprom the_sharp(addr_reg,clk,rom_pix);
   wire pixel;
   assign pixel = in_rom ? rom_pix : 1;
endmodule
```

Appendix K: mouse_disp.v

```
module mouse_disp2(clk,mx,my,hcount,vcount,pix_clk,pixel);
   input clk;           // video clock
   input [10:0] hcount; // current x,y location of pixel
   input [9:0] vcount;
   input       pix_clk; // pixel clock
   output pixel;  // pixel value output
   input [11:0] mx, my;
   // the memory address is hcount + vcount * 12
   reg in_rom = 0;
   reg [7:0]      raddr;  // row address
                          // adds the correct number to adjust for vcount
   always @(posedge clk) begin
     if(pix_clk) begin
        if (((hcount >= mx) && (hcount <= (mx+12))) &&
            ((vcount >= my) && (vcount <= (my+20)))) begin
            in_rom <= 1;
            raddr <= (hcount==mx & vcount==my) ? 0
                     : (hcount==mx & pix_clk) ? raddr + 12 : raddr;
        end
        else
           in_rom <= 0;
     end
   end

   //current location in rom
   wire [7:0]     addr = in_rom ? (hcount - mx + raddr) : 0;
   reg [7:0]      addr_reg;
   always @(posedge clk) addr_reg <= pix_clk ? addr : addr_reg;
   wire rom_pix;
   // instantiate the image rom
   arrowrom the_cursor(addr_reg,clk,rom_pix);
   wire pixel;
   assign pixel = in_rom ? rom_pix : 1;
endmodule
```

Appendix L: staff_display.v

```
module staff_disp(clk,hcount,vcount,pix_clk,pixel);
   input clk;           // video clock
   input [10:0] hcount; // current x,y location of pixel
   input [9:0] vcount;
   input       pix_clk; // pixel clock
   output pixel;  // pixel value output
   // the memory address is hcount + vcount * 1024
   // since hcount is never bigger than 1023,
```

```
   wire [19:0]    addr;
   assign addr = {vcount,hcount[9:0]};  //current location in rom
   reg [19:0]     addr_reg;
   always @(posedge clk) addr_reg <= pix_clk ? addr : addr_reg;
   // instantiate the image rom
   staff_rom drawing(addr_reg,clk,pixel);
endmodule
```

Appendix M: get_strings.v

```
// determine the ascii values of the note_num and
// corresponding frequency by determining the value
// of each of the base 10 digits of the numbers and then
// converting each digit separately to ascii
module get_strings(clk,note_num,nnstring,fstring);
   input clk;
   input [6:0] note_num;
   output [31:0] fstring;  // string of frequency, at most 4 base 10 digits

   output [15:0] nnstring;  // string of note_num, at most 2 base 10 digits

   wire [15:0] nnstring;

   reg [6:0] nn_tens = 0;
   reg [6:0] nn_ones = 0;
   reg [6:0] nn_temp_t;
   reg [6:0] nn_temp_o;

   always @ (posedge clk) begin
      nn_temp_t <= note_num;
      // determine the “tens” digit of the note_num

       if(nn_temp_t >= 80) begin
          nn_tens <= 8;
          nn_temp_o <= nn_temp_t - 80;
       end
       else if(nn_temp_t >= 70) begin
          nn_tens <= 7;
          nn_temp_o <= nn_temp_t - 70;
       end
       else if(nn_temp_t >= 60) begin
          nn_tens <= 6;
          nn_temp_o <= nn_temp_t - 60;
       end

       else if(nn_temp_t >= 50) begin
          nn_tens <= 5;
          nn_temp_o <= nn_temp_t - 50;
       end
       else if(nn_temp_t >= 40) begin
          nn_tens <= 4;
          nn_temp_o <= nn_temp_t - 40;
       end
       else if(nn_temp_t >= 30) begin
          nn_tens <= 3;
          nn_temp_o <= nn_temp_t - 30;
       end
```

```
    else if(nn_temp_t >= 20) begin
       nn_tens <= 2;
       nn_temp_o <= nn_temp_t - 20;
    end
    else if(nn_temp_t >= 10) begin
       nn_tens <= 1;
       nn_temp_o <= nn_temp_t - 10;
    end
    else begin
       nn_tens <= 0;
       nn_temp_o <= nn_temp_t;
    end

    //determine the “ones” digit

    if(nn_temp_o >= 9)
       nn_ones <= 9;
    else if(nn_temp_o >= 8)
       nn_ones <= 8;
    else if(nn_temp_o >= 7)
       nn_ones <= 7;
    else if(nn_temp_o >= 6)
       nn_ones <= 6;
    else if(nn_temp_o >= 5)
       nn_ones <= 5;
    else if(nn_temp_o >= 4)
       nn_ones <= 4;
    else if(nn_temp_o >= 3)
       nn_ones <= 3;
    else if(nn_temp_o >= 2)
       nn_ones <= 2;
    else if(nn_temp_o >= 1)
       nn_ones <= 1;
    else begin
       nn_ones <= 0;
    end
end

// as long as the note number is valid, convert the digits to ascii

assign nnstring = (note_num > 0 && note_num < 89) ?
                  {(nn_tens+8'h30),(nn_ones+8'h30)} : "  ";

// get the frequency from the tonerom

wire [12:0] freq;
tonerom tr(note_num,clk,freq);

wire [31:0] fstring;

reg [5:0] ff_thous = 0;
reg [5:0] ff_hunds = 0;
reg [5:0] ff_tens = 0;
reg [5:0] ff_ones = 0;
reg [31:0] ff_temp_th;
reg [31:0] ff_temp_h;
reg [31:0] ff_temp_t;
```

```
reg [31:0] ff_temp_o;

always @ (posedge clk) begin
   ff_temp_th <= freq;

    if(ff_temp_th >= 4000) begin
       ff_thous <= 4;
       ff_temp_h <= ff_temp_th - 4000;
    end
    else if(ff_temp_th >= 3000) begin
       ff_thous <= 3;
       ff_temp_h <= ff_temp_th - 3000;
    end
    else if(ff_temp_th >= 2000) begin
       ff_thous <= 2;
       ff_temp_h <= ff_temp_th - 2000;
    end
    else if(ff_temp_th >= 1000) begin
       ff_thous <= 1;
       ff_temp_h <= ff_temp_th - 1000;
    end
    else begin
       ff_thous <= 000;
       ff_temp_h <= ff_temp_th;
    end

    if(ff_temp_h >= 900) begin
       ff_hunds <= 9;
       ff_temp_t <= ff_temp_h - 900;
    end
    else if(ff_temp_h >= 800) begin
       ff_hunds <= 8;
       ff_temp_t <= ff_temp_h - 800;
    end
    else if(ff_temp_h >= 700) begin
       ff_hunds <= 7;
       ff_temp_t <= ff_temp_h - 700;
    end
    else if(ff_temp_h >= 600) begin
       ff_hunds <= 6;
       ff_temp_t <= ff_temp_h - 600;
    end
    else if(ff_temp_h >= 500) begin
       ff_hunds <= 5;
       ff_temp_t <= ff_temp_h - 500;
    end
    else if(ff_temp_h >= 400) begin
       ff_hunds <= 4;
       ff_temp_t <= ff_temp_h - 400;

    end
    else if(ff_temp_h >= 300) begin
       ff_hunds <= 3;
       ff_temp_t <= ff_temp_h - 300;
    end
    else if(ff_temp_h >= 200) begin
       ff_hunds <= 2;
```

```
   ff_temp_t <= ff_temp_h - 200;
end
else if(ff_temp_h >= 100) begin
   ff_hunds <= 1;
   ff_temp_t <= ff_temp_h - 100;
end
else begin
   ff_hunds <= 00;
   ff_temp_t <= ff_temp_h;
end

if(ff_temp_t >= 90) begin
   ff_tens <= 9;
   ff_temp_o <= ff_temp_t - 90;
end
else if(ff_temp_t >= 80) begin
   ff_tens <= 8;
   ff_temp_o <= ff_temp_t - 80;
end
else if(ff_temp_t >= 70) begin
   ff_tens <= 7;
   ff_temp_o <= ff_temp_t - 70;
end
else if(ff_temp_t >= 60) begin
   ff_tens <= 6;
   ff_temp_o <= ff_temp_t - 60;
end
else if(ff_temp_t >= 50) begin
   ff_tens <= 5;
   ff_temp_o <= ff_temp_t - 50;
end
else if(ff_temp_t >= 40) begin
   ff_tens <= 4;
   ff_temp_o <= ff_temp_t - 40;
end
else if(ff_temp_t >= 30) begin
   ff_tens <= 3;
   ff_temp_o <= ff_temp_t - 30;
end
else if(ff_temp_t >= 20) begin
   ff_tens <= 2;
   ff_temp_o <= ff_temp_t - 20;
end
else if(ff_temp_t >= 10) begin
   ff_tens <= 1;
   ff_temp_o <= ff_temp_t - 10;
end
else begin
   ff_tens <= 0;
   ff_temp_o <= ff_temp_t;
end

if(ff_temp_o >= 9)
   ff_ones <= 9;
else if(ff_temp_o >= 8)
   ff_ones <= 8;
else if(ff_temp_o >= 7)
```

```
          ff_ones <= 7;
       else if(ff_temp_o >= 6)
          ff_ones <= 6;
       else if(ff_temp_o >= 5)
          ff_ones <= 5;
       else if(ff_temp_o >= 4)
          ff_ones <= 4;
       else if(ff_temp_o >= 3)
          ff_ones <= 3;
       else if(ff_temp_o >= 2)
          ff_ones <= 2;
       else if(ff_temp_o >= 1)
          ff_ones <= 1;
       else begin
          ff_ones <= 0;
       end
   end
   assign fstring = (note_num > 0 && note_num < 89) ?
                     {(ff_thous+8'h30),(ff_hunds+8'h30),
                      (ff_tens+8'h30),(ff_ones+8'h30)} :
                     "    ";
endmodule
```

Appendix N: get_y.v

```
// find the y location, and the existence of an accidental and/or va
// for a given note_number
module get_y(clk,note_num,y,acc,va);
   input clk;
   input [6:0] note_num;
   output [9:0] y;
   output [1:0] acc;
   output [1:0] va;
   wire [9:0] y;
   wire [13:0] rom_info;
   // access the ylocrom, where each entry has 14 bits:
   // [13:4] yposition,
   // [3:2] acc?  If so, then what type?
   // [1:0] va?  If so, how many octaves?

   ylocrom ylocations(note_num,clk,rom_info);
   assign y = rom_info[13:4];
   assign acc = rom_info[3:2];
   assign va = rom_info[1:0];
endmodule
```

Appendix O: get_letter.v

```
//access the letterrom to determine the letter name of the current note
//if it's a sharp or flat, include that information in the output string
module get_letter(clk,note_num,lstring);
   input clk;
   input [6:0] note_num;
   output [55:0] lstring;

   wire [55:0] lstring;
   wire [4:0] lrom;
   letterrom l(note_num,clk,lrom);
```

```
   wire [39:0] acc;
   assign acc = (lrom[1:0]==2'b10) ? "Flat " :
                (lrom[1:0]==2'b01) ? "Sharp" :
                   "     ";
   assign lstring = (note_num > 0 && note_num < 89) ?
                      {(lrom[4:2] + 8'd65)," ",acc} : "       ";
endmodule
```

Appendix P: get_pri_strings.v

```
//conver the action and user selected form of input into strings
// based on their numerical values
module get_pri_strings(action,ninput,actionstr,ninputstr);
   input [1:0] action;
   input [1:0] ninput;
   output [63:0] actionstr;
   output [63:0] ninputstr;

   wire [63:0] actionstr;
   wire [63:0] ninputstr;
   assign actionstr = (action==2'b11) ? "Record  " :
                      (action==2'b10) ? "Playback" :
                          (action==2'b01) ? "Playtone" :
                          "Waiting ";  //no action selected yet
   assign ninputstr = (ninput==2'b11) ? "Switches" :
                      (ninput==2'b10) ? "Mouse   " :
                          (ninput==2'b01) ? "Keyboard" :
                          "Waiting ";  // no input selected yet
endmodule
```

Appendix Q: action_priority.v

```
//using the button inputs, determine which action the user has selected
//based on this input, determine what note num and to_ac97_data should be
//processed
module action_priority(clk,record_new,playback_new,playtone_new,action,
                       note_num_pb,note_num_pt,note_num,
                       to_ac97_data_pb,to_ac97_data_pt,to_ac97_data);
   input clk;
   input record_new;
   input playback_new;
   input playtone_new;
   input [6:0] note_num_pb;
   input [6:0] note_num_pt;
   output [1:0] action;
   output [6:0] note_num;
   input [19:0] to_ac97_data_pb;
   input [19:0] to_ac97_data_pt;
   output [19:0] to_ac97_data;

   reg [1:0] action = 2'b0;
   always @ (posedge clk) begin
       action <= record_new ? 2'b11 :
                 playback_new ? 2'b10 :
                 playtone_new ? 2'b01 : action;
   end
   assign note_num = (action == 2'b10) ? note_num_pb : note_num_pt;
   assign to_ac97_data = (action == 2'b11) ? 19'b0 :
```

```
                         (action == 2'b10) ? to_ac97_data_pb :
to_ac97_data_pt;
endmodule
```

Appendix R: line.v

```
// draw a line for middle c at the arbitrary location
module line(clk,hcount,vcount,pix);
   input clk;
   input [10:0] hcount;
   input [9:0] vcount;
   output pix;
   reg pix;
   always @ (hcount or vcount) begin
       if (((hcount >= 496) && (hcount <= 560)) &&
           ((vcount >= 305) && (vcount <= 309)))
          pix = 0;
       else
          pix = 1;
   end
endmodule
```

Appendix S: ps2_mouse.v

```
// directly from the class website
```

Appendix T: ps2_kbd.v

```
// directly from the class website
```

Appendix U: ac97_audio.v

```
//directly from the class website, with one change:
//change VOLUME from a parameter to an input so it can be controlled
//by the user
module audio (clock_27mhz, reset, audio_in_data, audio_out_data, ready,
            audio_reset_b, ac97_sdata_out, ac97_sdata_in,
              ac97_synch, ac97_bit_clock,VOLUME);

   input clock_27mhz;
   input reset;
   output [19:0] audio_in_data;
   input [19:0] audio_out_data;
   output ready;

   //ac97 interface signals
   output audio_reset_b;
   output ac97_sdata_out;
   input ac97_sdata_in;
   output ac97_synch;
   input ac97_bit_clock;

   input [3:0] VOLUME;
   ...
endmodule
```

Appendix V: volume_control.v

```
//allow the user to set the volume using the up and down buttons
```

```
module volume_control(clk,reset,initval,up,down,vol);
   input clk;
   input reset;
   input [3:0] initval;
   input up;
   input down;
   output [3:0] vol;

   reg [3:0] vol;
   reg up_change;
   reg old_up;
   reg down_change;
   reg old_down;
   // change the volume everytime up or down is pushed, so keep track of
   // changes in up and down
   always @ (posedge clk) begin
      old_up <= up;
       old_down <= down;
       up_change <= up & ~old_up;
       down_change <= down & ~old_down;
   end
   always @ (posedge clk) begin
       vol <= reset ? initval :
              (up_change ? ((vol < 15) ? (vol + 1) : vol) :
                (down_change ? ((vol > 0) ? (vol - 1) : vol) : vol));
   end

endmodule
```

Appendix W: ROM .v files

```
module tonerom (
      addr,
      clk,
      dout);    // synthesis black_box

input [6 : 0] addr;
input clk;
output [12 : 0] dout;

// synopsys translate_off

      BLKMEMSP_V6_1 #(
            7,    // c_addr_width
            "0",  // c_default_data
            89,   // c_depth
            0,    // c_enable_rlocs
            0,    // c_has_default_data
            0,    // c_has_din
            0,    // c_has_en
            0,    // c_has_limit_data_pitch
            0,    // c_has_nd
            0,    // c_has_rdy
            0,    // c_has_rfd
            0,    // c_has_sinit
            0,    // c_has_we
            18,   // c_limit_data_pitch
```

```
            "tonerom.mif",    // c_mem_init_file
            0,    // c_pipe_stages
            0,    // c_reg_inputs
            "0",  // c_sinit_value
            13,   // c_width
            0,    // c_write_mode
            "0",  // c_ybottom_addr
            1,    // c_yclk_is_rising
            1,    // c_yen_is_high
            "hierarchy1",     // c_yhierarchy
            0,    // c_ymake_bmm
            "16kx1",    // c_yprimitive_type
            1,    // c_ysinit_is_high
            "1024",     // c_ytop_addr
            0,    // c_yuse_single_primitive
            1,    // c_ywe_is_high
            1)    // c_yydisable_warnings
      inst (
            .ADDR(addr),
            .CLK(clk),
            .DOUT(dout),
            .DIN(),
            .EN(),
            .ND(),
            .RFD(),
            .RDY(),
            .SINIT(),
            .WE());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of tonerom is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of tonerom is "black_box"

endmodule

module ylocrom (
      addr,
      clk,
      dout);    // synthesis black_box

input [6 : 0] addr;
input clk;
output [13 : 0] dout;

// synopsys translate_off

      BLKMEMSP_V6_1 #(
            7,    // c_addr_width
            "0",  // c_default_data
```

```
            89,   // c_depth
            0,    // c_enable_rlocs
            0,    // c_has_default_data
            0,    // c_has_din
            0,    // c_has_en
            0,    // c_has_limit_data_pitch
            0,    // c_has_nd
            0,    // c_has_rdy
            0,    // c_has_rfd
            0,    // c_has_sinit
            0,    // c_has_we
            18,   // c_limit_data_pitch
            "ylocrom.mif",    // c_mem_init_file
            0,    // c_pipe_stages
            0,    // c_reg_inputs
            "0",  // c_sinit_value
            14,   // c_width
            0,    // c_write_mode
            "0",  // c_ybottom_addr
            1,    // c_yclk_is_rising
            1,    // c_yen_is_high
            "hierarchy1",     // c_yhierarchy
            0,    // c_ymake_bmm
            "16kx1",    // c_yprimitive_type
            1,    // c_ysinit_is_high
            "1024",     // c_ytop_addr
            0,    // c_yuse_single_primitive
            1,    // c_ywe_is_high
            1)    // c_yydisable_warnings
      inst (
            .ADDR(addr),
            .CLK(clk),
            .DOUT(dout),
            .DIN(),
            .EN(),
            .ND(),
            .RFD(),
            .RDY(),
            .SINIT(),
            .WE());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of ylocrom is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of ylocrom is "black_box"

endmodule

module noterom (
      addr,
      clk,
```

```
      dout);    // synthesis black_box

input [9 : 0] addr;
input clk;
output [0 : 0] dout;

// synopsys translate_off

      BLKMEMSP_V6_1 #(
            10,   // c_addr_width
            "0",  // c_default_data
            1024, // c_depth
            0,    // c_enable_rlocs
            0,    // c_has_default_data
            0,    // c_has_din
            0,    // c_has_en
            0,    // c_has_limit_data_pitch
            0,    // c_has_nd
            0,    // c_has_rdy
            0,    // c_has_rfd
            0,    // c_has_sinit
            0,    // c_has_we
            18,   // c_limit_data_pitch
            "noterom.mif",    // c_mem_init_file
            0,    // c_pipe_stages
            0,    // c_reg_inputs
            "0",  // c_sinit_value
            1,    // c_width
            0,    // c_write_mode
            "0",  // c_ybottom_addr
            1,    // c_yclk_is_rising
            1,    // c_yen_is_high
            "hierarchy1",     // c_yhierarchy
            0,    // c_ymake_bmm
            "16kx1",    // c_yprimitive_type
            1,    // c_ysinit_is_high
            "1024",     // c_ytop_addr
            0,    // c_yuse_single_primitive
            1,    // c_ywe_is_high
            1)    // c_yydisable_warnings
      inst (
            .ADDR(addr),
            .CLK(clk),
            .DOUT(dout),
            .DIN(),
            .EN(),
            .ND(),
            .RFD(),
            .RDY(),
            .SINIT(),
            .WE());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
```

```
// synthesis attribute fpga_dont_touch of noterom is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of noterom is "black_box"

endmodule

module sharprom (
      addr,
      clk,
      dout);    // synthesis black_box

input [9 : 0] addr;
input clk;
output [0 : 0] dout;

// synopsys translate_off

      BLKMEMSP_V6_1 #(
            10,   // c_addr_width
            "0",  // c_default_data
            1024, // c_depth
            0,    // c_enable_rlocs
            0,    // c_has_default_data
            0,    // c_has_din
            0,    // c_has_en
            0,    // c_has_limit_data_pitch
            0,    // c_has_nd
            0,    // c_has_rdy
            0,    // c_has_rfd
            0,    // c_has_sinit
            0,    // c_has_we
            18,   // c_limit_data_pitch
            "sharprom.mif",   // c_mem_init_file
            0,    // c_pipe_stages
            0,    // c_reg_inputs
            "0",  // c_sinit_value
            1,    // c_width
            0,    // c_write_mode
            "0",  // c_ybottom_addr
            1,    // c_yclk_is_rising
            1,    // c_yen_is_high
            "hierarchy1",     // c_yhierarchy
            0,    // c_ymake_bmm
            "16kx1",    // c_yprimitive_type
            1,    // c_ysinit_is_high
            "1024",     // c_ytop_addr
            0,    // c_yuse_single_primitive
            1,    // c_ywe_is_high
            1)    // c_yydisable_warnings
      inst (
            .ADDR(addr),
            .CLK(clk),
            .DOUT(dout),
            .DIN(),
            .EN(),
```

```
            .ND(),
            .RFD(),
            .RDY(),
            .SINIT(),
            .WE());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of sharprom is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of sharprom is "black_box"

endmodule

module staff_rom (
      addr,
      clk,
      dout);    // synthesis black_box

input [19 : 0] addr;
input clk;
output [0 : 0] dout;

// synopsys translate_off

      BLKMEMSP_V6_1 #(
            20,   // c_addr_width
            "0",  // c_default_data
            786432,     // c_depth
            0,    // c_enable_rlocs
            0,    // c_has_default_data
            0,    // c_has_din
            0,    // c_has_en
            0,    // c_has_limit_data_pitch
            0,    // c_has_nd
            0,    // c_has_rdy
            0,    // c_has_rfd
            0,    // c_has_sinit
            0,    // c_has_we
            18,   // c_limit_data_pitch
            "staff_rom.mif",  // c_mem_init_file
            0,    // c_pipe_stages
            0,    // c_reg_inputs
            "0",  // c_sinit_value
            1,    // c_width
            0,    // c_write_mode
            "0",  // c_ybottom_addr
            1,    // c_yclk_is_rising
            1,    // c_yen_is_high
            "hierarchy1",     // c_yhierarchy
            0,    // c_ymake_bmm
            "16kx1",    // c_yprimitive_type
```

```
            1,    // c_ysinit_is_high
            "1024",     // c_ytop_addr
            0,    // c_yuse_single_primitive
            1,    // c_ywe_is_high
            1)    // c_yydisable_warnings
      inst (
            .ADDR(addr),
            .CLK(clk),
            .DOUT(dout),
            .DIN(),
            .EN(),
            .ND(),
            .RFD(),
            .RDY(),

            .SINIT(),
            .WE());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of staff_rom is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of staff_rom is "black_box"

endmodule

module lab3bram (
      addr,
      clk,
      din,
      dout,
      we);    // synthesis black_box

input [15 : 0] addr;
input clk;
input [19 : 0] din;
output [19 : 0] dout;
input we;

// synopsys translate_off

      BLKMEMSP_V6_1 #(
            16,   // c_addr_width
            "0",  // c_default_data
            65536,      // c_depth
            0,    // c_enable_rlocs
            1,    // c_has_default_data
            1,    // c_has_din
            0,    // c_has_en
            0,    // c_has_limit_data_pitch
            0,    // c_has_nd
            0,    // c_has_rdy
```

```
            0,    // c_has_rfd
            0,    // c_has_sinit
            1,    // c_has_we
            18,   // c_limit_data_pitch
            "mif_file_16_1",  // c_mem_init_file
            0,    // c_pipe_stages
            0,    // c_reg_inputs
            "0",  // c_sinit_value
            20,   // c_width
            0,    // c_write_mode
            "0",  // c_ybottom_addr
            1,    // c_yclk_is_rising
            1,    // c_yen_is_high
            "hierarchy1",     // c_yhierarchy
            0,    // c_ymake_bmm
            "16kx1",    // c_yprimitive_type
            1,    // c_ysinit_is_high
            "1024",     // c_ytop_addr
            0,    // c_yuse_single_primitive
            1,    // c_ywe_is_high
            1)    // c_yydisable_warnings
      inst (
            .ADDR(addr),
            .CLK(clk),
            .DIN(din),
            .DOUT(dout),
            .WE(we),
            .EN(),
            .ND(),
            .RFD(),
            .RDY(),
            .SINIT());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of lab3bram is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of lab3bram is "black_box"

endmodule

module letterrom (
      addr,
      clk,
      dout);    // synthesis black_box

input [6 : 0] addr;
input clk;
output [4 : 0] dout;

// synopsys translate_off
```

```
	BLKMEMSP_V6_1 #(
		7,	// c_addr_width
		"0",	// c_default_data
		89,	// c_depth
		0,	// c_enable_rlocs
		0,	// c_has_default_data
		0,	// c_has_din
		0,	// c_has_en
		0,	// c_has_limit_data_pitch
		0,	// c_has_nd
		0,	// c_has_rdy
		0,	// c_has_rfd
		0,	// c_has_sinit
		0,	// c_has_we
		18,	// c_limit_data_pitch
		"letterrom.mif",	// c_mem_init_file
		0,	// c_pipe_stages
		0,	// c_reg_inputs
		"0",	// c_sinit_value
		5,	// c_width
		0,	// c_write_mode
		"0",	// c_ybottom_addr
		1,	// c_yclk_is_rising
		1,	// c_yen_is_high
		"hierarchy1",	// c_yhierarchy
		0,	// c_ymake_bmm
		"16kx1",	// c_yprimitive_type
		1,	// c_ysinit_is_high
		"1024",	// c_ytop_addr
		0,	// c_yuse_single_primitive
		1,	// c_ywe_is_high
		1)	// c_yydisable_warnings
	inst (
		.ADDR(addr),
		.CLK(clk),
		.DOUT(dout),
		.DIN(),
		.EN(),
		.ND(),
		.RFD(),
		.RDY(),
		.SINIT(),
		.WE());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of letterrom is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of letterrom is "black_box"

endmodule
```

```
module arrowrom (
      addr,
      clk,
      dout);    // synthesis black_box

input [7 : 0] addr;
input clk;
output [0 : 0] dout;

// synopsys translate_off

      BLKMEMSP_V6_1 #(
            8,    // c_addr_width
            "0",  // c_default_data
            252,  // c_depth
            0,    // c_enable_rlocs
            0,    // c_has_default_data
            0,    // c_has_din
            0,    // c_has_en
            0,    // c_has_limit_data_pitch
            0,    // c_has_nd
            0,    // c_has_rdy
            0,    // c_has_rfd
            0,    // c_has_sinit
            0,    // c_has_we
            18,   // c_limit_data_pitch
            "arrowrom.mif",   // c_mem_init_file
            0,    // c_pipe_stages
            0,    // c_reg_inputs
            "0",  // c_sinit_value
            1,    // c_width
            0,    // c_write_mode
            "0",  // c_ybottom_addr
            1,    // c_yclk_is_rising
            1,    // c_yen_is_high
            "hierarchy1",     // c_yhierarchy
            0,    // c_ymake_bmm
            "16kx1",    // c_yprimitive_type
            1,    // c_ysinit_is_high
            "1024",     // c_ytop_addr
            0,    // c_yuse_single_primitive
            1,    // c_ywe_is_high
            1)    // c_yydisable_warnings
      inst (
            .ADDR(addr),
            .CLK(clk),
            .DOUT(dout),
            .DIN(),
            .EN(),
            .ND(),
            .RFD(),
            .RDY(),
            .SINIT(),
            .WE());


// synopsys translate_on
```

```
// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of arrowrom is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of arrowrom is "black_box"

endmodule

module font_rom(
      addr,
      clk,
      dout);


input [10 : 0] addr;
input clk;
output [7 : 0] dout;

// synopsys translate_off

      BLKMEMSP_V6_1 #(
            11,   // c_addr_width
            "0",  // c_default_data
            1536, // c_depth
            0,    // c_enable_rlocs
            0,    // c_has_default_data
            0,    // c_has_din
            0,    // c_has_en
            0,    // c_has_limit_data_pitch
            0,    // c_has_nd
            0,    // c_has_rdy
            0,    // c_has_rfd
            0,    // c_has_sinit
            0,    // c_has_we
            18,   // c_limit_data_pitch
            "font_rom.mif",   // c_mem_init_file
            0,    // c_pipe_stages
            0,    // c_reg_inputs
            "0",  // c_sinit_value
            8,    // c_width
            0,    // c_write_mode
            "0",  // c_ybottom_addr
            1,    // c_yclk_is_rising
            1,    // c_yen_is_high
            "hierarchy1",     // c_yhierarchy
            0,    // c_ymake_bmm
            "16kx1",    // c_yprimitive_type
            1,    // c_ysinit_is_high
            "1024",     // c_ytop_addr
            0,    // c_yuse_single_primitive
            1,    // c_ywe_is_high
            1)    // c_yydisable_warnings
      inst (
            .ADDR(addr),
```

```
            .CLK(clk),
            .DOUT(dout),
            .DIN(),
            .EN(),
            .ND(),
            .RFD(),
            .RDY(),
            .SINIT(),
            .WE());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of font_rom is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of font_rom is "black_box"

endmodule

module freqincrom (
      addr,
      clk,
      dout);    // synthesis black_box

input [6 : 0] addr;
input clk;
output [14 : 0] dout;

// synopsys translate_off

      BLKMEMSP_V6_1 #(
            7,    // c_addr_width
            "0",  // c_default_data
            89,   // c_depth
            0,    // c_enable_rlocs
            0,    // c_has_default_data
            0,    // c_has_din
            0,    // c_has_en
            0,    // c_has_limit_data_pitch
            0,    // c_has_nd
            0,    // c_has_rdy
            0,    // c_has_rfd
            0,    // c_has_sinit
            0,    // c_has_we
            18,   // c_limit_data_pitch
            "freqincrom.mif", // c_mem_init_file
            0,    // c_pipe_stages
            0,    // c_reg_inputs
            "0",  // c_sinit_value
            15,   // c_width
            0,    // c_write_mode
            "0",  // c_ybottom_addr
            1,    // c_yclk_is_rising
```

```
            1,    // c_yen_is_high
            "hierarchy1",     // c_yhierarchy
            0,    // c_ymake_bmm
            "16kx1",    // c_yprimitive_type
            1,    // c_ysinit_is_high
            "1024",     // c_ytop_addr
            0,    // c_yuse_single_primitive
            1,    // c_ywe_is_high
            1)    // c_yydisable_warnings
      inst (
            .ADDR(addr),
            .CLK(clk),
            .DOUT(dout),
            .DIN(),
            .EN(),
            .ND(),
            .RFD(),
            .RDY(),

            .SINIT(),
            .WE());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of freqincrom is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of freqincrom is "black_box"

endmodule
```

APPENDIX 1: {v(t)[sin(w1-w2)t + sin(w1+w2)t]} *LPF

Appendix 1.1: lab3.v

```
// Clock at 65MHz
   // use FPGA's digital clock manager to produce a
   // 65MHz clock (actually 64.8MHz)
   wire clock_65mhz_unbuf;//,clock_65mhz;
   DCM vclk1(.CLKIN(clock_27mhz),.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 [7:0] from_ac97_data, to_ac97_data;
   wire ready;

   // AC97 driver
   audio a(clock_65mhz, reset, from_ac97_data, to_ac97_data, ready,
         audio_reset_b, ac97_sdata_out, ac97_sdata_in,
         ac97_synch, ac97_bit_clock);

   // push ENTER button to record, release to playback
   wire playback;
   debounce benter(reset, clock_65mhz, button_enter, playback);

/////////////
//// DDS ////
/////////////

   wire [6:0] chknum1,chknum2,chknum;
   wire [5:0] chksin, chkcos;
   assign chknum1 = 6'd40;
   assign chknum2 = 6'd52;
   wire which;
   assign which = switch[0];
   assign chknum = (which ? chknum1 : chknum2);

   freq2sig chksig(clock_65mhz,1,chknum,rdy,chksin,chkcos);
//   assign to_ac97_data = {2'b0,chksin};

////////////////
//// Record ////
////////////////

   wire we;
   wire [7:0] din;
   wire [7:0] dout;
   wire [15:0] addr;

   // recorder r(clock_65mhz, reset, playback, ready, {2'b0,chksin}, dout,
to_ac97_data, we, din, addr);

   // call to bram
   lab3ram b(addr,clock_65mhz,din,dout,we);
```

```
////////////////
//// Filter ////
////////////////

   wire [12:0] fout;
   wire [12:0] match_freq;
   filt_fsm
check(clock_65mhz,reset,ready,chksin,chkcos,ff_rdy,fout,match_freq);
   assign fout = ((ff_rdy == 0) ? 12'b0 : fout);
   // assign to_ac97_data = fout;

///////////////
//// Tests ////
///////////////

   display_16hex d(reset, clock_65mhz, {51'b0,match_freq}, disp_blank,
disp_clock, disp_rs, disp_ce_b, disp_reset_b, disp_data_out);
   assign led = ~we;
   assign analyzer2_clock = ready;
   assign analyzer2_data = {3'b0,match_freq};
   assign analyzer4_clock = 1'b1;
   assign analyzer4_data = {3'b0,fout};
```

Appendix 1.2: freq2sig.v

```
// inaptly named, this module takes a note number and converts it to sin/cos
// signals
module freq2sig(clock,reset,audio_ready,note_num,rdy,sin,cos);
      input clock, reset, audio_ready;
      input [6:0] note_num;
      output [5:0] sin, cos;
      output rdy;

// output
      wire [5:0] sin,cos;

// input
      // phase increment value (sets output freq)
      wire [20:0] freq;
      wire [14:0] freqinc;
      freqincrom inc(note_num,clk,freqinc);
      assign freq = {6'b0,freqinc};

      wire we;
      reg [20:0] old_freq;
      always @(posedge clock) old_freq  <= freq;
      assign we = ~(freq == old_freq);

      // generate DDS clock from ac'97 audio_ready signal
      reg old_ready;
      reg clock_dds;
      always @(posedge clock) begin
      old_ready <= audio_ready;
      clock_dds <= audio_ready & ~old_ready; end      // one cycle delayed ok
```

```
    wire [4:0] A;
    assign A = 5'b0;

  sine sine1(freq,we,A,clock,clock_dds,rdy,sin,cos);

endmodule
```

Appendix 1.2.i: sine.v

```
// just the i/o ports are included, since it's just a coregen BRAM
module sine(
   DATA,
   WE,
   A,
   CLK,
   RFD,
   RDY,
   SINE,
   COSINE
   ); // synthesis black_box

   input [20 : 0] DATA;
   input WE;
   input [4 : 0] A;
   input CLK;
   output RFD;
   output RDY;
   output [5 : 0] SINE;
   output [5 : 0] COSINE;
```

Appendix 1.3: filt_fsm.v

```
// Take the input from the AC97 and run it through the filter with values
// from the 88freq ROM until there is a match
module filt_fsm(clock,reset,audio_ready,sin_in,cos_in,rdy,fout,match_freq);

   input clock,reset,audio_ready;          // make sure clocking at same time
   input [7:0] sin_in, cos_in;             // should be input sig from ac97;
                                           // test with predetermined signals
   output rdy;
   output [12:0] fout;

   wire [12:0] match_freq;
   wire rdy;

   wire [6:0] rom_addr;                    // equivalent to note number
   wire [12:0] rom_freq;                   // frequency of that number
   wire [5:0] rom_sin,rom_cos;

   tonerom rom88(rom_addr,clock,rom_freq);
   freq2sig romchk(clock,reset,audio_ready,rom_addr,rom_rdy,rom_sin,rom_cos);

// needs to be made
   wire [1:0] match;           // 2 if freq match, 1 if too low, 0 if too high

   wire error;
   assign error = (rom_addr >= 7'd88) ? 1 : 0;
```

```
////////////////////////////
//// From Matlab
////////////////////////////
   wire [7:0] sig_add, sig_sub, sum_sigs, sig_in;

   assign sig_add = sin_in*rom_cos + cos_in*rom_sin;
   assign sig_sub = sin_in*rom_cos - cos_in*rom_sin;
   assign sum_sigs = sig_add + sig_sub;

   filter fthis(clock,reset,f_we,sum_sigs,addr,fout);
   //haha fthis; i'm funny aren't i. life's sad.

   reg [1:0] state = 2'b0;
   parameter init_chk = 0;
   parameter chk_up = 1;
   parameter chk_down = 2;
   parameter matched = 3;

   always @(posedge clock)
      case(state)
      init_chk:   state <= ((match == 2'd2) ? matched : ((match == 2'd1) ?
chk_up : chk_down));
      chk_up:     state <= ((match == 2'd2) ? matched : ((match == 2'd1) ?
chk_up : chk_down));
      chk_down:   state <= ((match == 2'd2) ? matched : ((match == 2'd1) ?
chk_up : chk_down));
      matched:    state <= init_chk;
      default:    state <= init_chk;
      endcase

   assign rdy = ((state == 2'd3) ? 1 : 0);
   assign match_freq = ((state == 2'd3) ? rom_freq : 0);
   assign rom_addr = ((state == 0 || state == 3) ? 6'd40 : ((state == 1'd1) ?
(rom_addr+1) : (rom_addr-1)));

endmodule
```

Appendix 1.3.i: sine.m

```
function [x t] = sinusoid(omega0, dur, omegaS)


T = 2*pi/omegaS;
nmax = fix(dur/T);
n = [0:nmax];
t = n*T;

x = sin (omega0*t);
```

Appendix 1.3.ii: cosine.m

```
function [x t] = cosine(omega0, dur, omegaS)


T = 2*pi/omegaS;
nmax = fix(dur/T);
n = [0:nmax];
```

```
t = n*T;

x = cos (omega0*t);
```

Appendix 1.3.iii: v1.m

```
omegaS = 2*pi*8192;
[c40 t] = sine(2*pi*261.63, 2, omegaS);
v = 2.71828183.^t;

dtfilter1 = 0.5.^t;
%dtfilter2 = sinc (t);

diff1 = sine((2*pi*262.63-2*pi*261.63), 2, omegaS);
sum1 = sine((2*pi*262.63+2*pi*261.63), 2, omegaS);
z1 = v.*(diff1 + sum1);
fn1 = conv(z1,dtfilter1);

diff2 = sine((2*pi*271.63-2*pi*261.63), 2, omegaS);
sum2 = sine((2*pi*271.63+2*pi*261.63), 2, omegaS);
z2 = v.*(diff2 + sum2);
fn2 = conv(z2,dtfilter1);

diff3 = sine((2*pi*361.63-2*pi*261.63), 2, omegaS);
sum3 = sine((2*pi*361.63+2*pi*261.63), 2, omegaS);
z3 = v.*(diff3 + sum3);
fn3 = conv(z3,dtfilter1);


t_in = [0:pi/omegaS:2];
subplot(4,2,1),plot(t, v);title('v(t) and [v(t)(sin((w_1+w_2)t) +
sin((w_1-w_2)t))]*(0.5.^t)');
subplot(4,2,3),plot(t_in, fn1);title('1Hz difference (261.63Hz vs.
262.63Hz)');
subplot(4,2,5),plot(t_in, fn2);title('10Hz difference (261.63Hz vs.
271.63Hz)');
subplot(4,2,7),plot(t_in, fn3);title('100Hz difference (261.63Hz vs.
361.63Hz)');

subplot(4,2,2),plot(t, v);title('v(t) and v(t)(sin((w_1+w_2)t) +
sin((w_1-w_2)t))');
subplot(4,2,4),plot(t, z1);title('1Hz difference (261.63Hz vs.
262.63Hz)');
subplot(4,2,6),plot(t, z2);title('10Hz difference (261.63Hz vs.
271.63Hz)');
subplot(4,2,8),plot(t, z3);title('100Hz difference (261.63Hz vs.
361.63Hz)');
```

Appendix 1.4: filter.v

```
// takes the calculated & modulated signal and passes it through a LPF
module filter(clock, reset, we, sig_in, saddr, oout);
	input clock,reset,we;
	input [5:0] sig_in;
	output [15:0] saddr;
	output [7:0] oout;
```

```
	wire [3:0] faddr;
	wire [12:0] fout;
	filter_sig1 filt(faddr,clock,fout);

	wire [15:0] mout;
	assign mout = fout*sig_in;

	wire [15:0] oaddr;
	wire [15:0] curr_oaddr;
	wire [7:0] oin, oout;
	wire owe;

	output_sig out(oaddr,clock,oin,oout,owe);
	// RAM to keep the filtered signal

	// This is the FSM;
	reg [1:0] state;
	parameter now = 0; // keep same saddr, increase faddr
	parameter new = 1; // increase saddr, restart faddr
	parameter done = 2;

	wire [15:0] saddr;

	always @(posedge clock)
	case (state)
		now: state <= (faddr == 4'd15) ? new : now;
		new: state <= (saddr == 16'hFF) ? done : now;
		done: state <= new;
		default: state <= new;
	endcase


// need to pass saddr back into filter_fsm; store signal?
	assign saddr = ((reset) ? 16'b0 : ((state == 2'b0) ? saddr : ((state ==
2'b1) ? (saddr + 1) : 16'b0)));
	// now: keep saddr, else new, inc it, else 0
	assign faddr = ((reset) ? 16'b0 : ((state == 2'b0) ? (faddr + 1) :
16'b0));
	// now: inc it, else 0.

	assign oaddr = ((reset) ? 16'b0 : ((state == 2'b0) ? (oaddr + 1) :
((state == 2'b1) ? (curr_oaddr + 1) : 16'b0)));
	// now: inc it, else new: inc the current initial point, else 0.
	assign curr_oaddr = ((reset) ? 16'b0 : ((state == 2'b0) ? curr_oaddr :
((state == 2'b1) ? (curr_oaddr + 1) : 16'b0)));
	// keep track of the initial point to start the convolution; inc only
at new, keep at now.

	assign oin = mout + oout;
	// add the current product back into the RAM.


endmodule
```

Appendix 1.4.i: filter_sig1.coe

```
//coe file for the filter_sig1 ROM
memory_initialization_radix=2;
memory_initialization_vector=

1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
```

Appendix 1.4.ii: outputsig.v

```
// just the i/o ports are included; it's a coregen BRAM
module output_sig (
	addr,
	clk,
	din,
	dout,
	we);	// synthesis black_box

input [15 : 0] addr;
input clk;
input [7 : 0] din;
output [7 : 0] dout;
input we;
```

APPENDIX 2: {sin(w1)t – sin(w2)t]} ∗LPF

Note: Some of the numbers/wires may not match up. This was compiled from many various versions I have, and I'm not positive they were all testing the same thing.

Appendix 2.1: labkit.v

```
// Clock at 65MHz
   // use FPGA's digital clock manager to produce a
   // 65MHz clock (actually 64.8MHz)
   wire clock_65mhz_unbuf;//,clock_65mhz;
   DCM vclk1(.CLKIN(clock_27mhz),.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 [7:0] from_ac97_data, to_ac97_data;
   wire ready;

   // AC97 driver
   audio a(clock_65mhz, reset, from_ac97_data, to_ac97_data, ready,
         audio_reset_b, ac97_sdata_out, ac97_sdata_in,
         ac97_synch, ac97_bit_clock);

   // push ENTER button to record, release to playback
   wire playback;
   debounce benter(reset, clock_65mhz, button_enter, playback);

/////////////
//// DDS ////
/////////////

   wire [6:0] chknum;
   assign chknum = switch[6:0];
   wire [19:0] chksig;

   speakers romchk(clock_65mhz,chknum,ready,chksig);

////////////////
//// Record ////
////////////////
   wire [19:0] din;
   wire [19:0] dout;
   wire [15:0] addr;

   wire rec;
   wire [19:0] from_ac97_data1;

   assign rec = switch[7];
   assign from_ac97_data1 = (rec) ? chksig : from_ac97_data;

   recorder r(clock_65mhz, reset, playback, ready, from_ac97_data1,
to_ac97_data, addr, din, dout, we);
```

```
   //recorder r(clock_65mhz, reset, playback, ready, chksig, to_ac97_data,
addr, din, dout, we);

   // call to bram

   lab3ram b(addr,clock_65mhz,din,dout,we);

////////////////
//// Filter ////
////////////////
   wire reset_1;
   assign reset_1 = ~button0;

   wire fwe;                  // start calculation
   assign fwe = ~button1;     // press = fwe high, else low

   wire error;
   wire match;
   wire [6:0] notenum;

   wire [7:0] rom_addr;
   wire fw_rdy;
   wire [13:0] sd_addr;

   rw_fsm fsm(clock_65mhz, ready, reset_1, fwe, to_ac97_data,notenum, error,
match, rom_addr);


///////////////
//// Tests ////
///////////////

   assign analyzer2_clock = ready;
   assign analyzer2_data[15:8] = {2'b0,sd_addr[13:8]};
   assign analyzer2_data[7:0] = sd_addr[7:0];

   assign analyzer4_clock = ready;
   assign analyzer4_data[15:8] = 0;
   assign analyzer4_data[7:0] = {1'b1,notenum};

   //assign to_ac97_data = chksig;


   display_16hex d(reset, clock_65mhz, {3'b0,match,3'b0,error,49'b0,notenum},
disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b, disp_data_out);

   // LED
   assign led[7:3] = 5'b11111;
   assign led[0] = ~match;
   assign led[1] = ~error;
   assign led[2] = ~fwe;
```

Appendix 2.2: rw_fsm.v

```
// resets/holds/increments rom_addr as needed for testing
module rw_fsm(clock, audio_ready, reset, fwe, sig_in, notenum, error, match,
rom_addr);
```

```
	input clock;		//65MHz clock
	input audio_ready;	//ready for the DDS
	input reset;		//call a reset
	input fwe;		//call for the filter to start
	input [19:0] sig_in;	//input signal

	output [6:0] notenum;	//output the matched notenum
	output error;		//output if there's an error
	output match;		// to check; output when there's a match
	output [6:0] rom_addr;	// to check; the rom addr we're cycling through


	reg [6:0] rom_addr;

	// some registers that will let us know where we're at
	// could be instantiated with wires, but didn't work o.0

	reg s2w;		//impulse when user first presses fwe
	reg w2r;		//impulse when user first releases fwe
	reg old_fwe;
	always @(posedge clock)
		begin
		if (reset) begin old_fwe <= 0; s2w <= 0; w2r <= 0; end
		else if (~old_fwe && fwe) begin old_fwe <= 1; s2w <= 1; end
		else if (old_fwe && ~fwe) begin old_fwe <= 0; w2r <= 1; end
		else begin s2w <= 0; w2r <= 0; end
		end

	wire calc;		//step fn after user has pressed fwe
	assign calc = (reset || match) ? 0 : ((calc) ? 1 : fwe);
	wire re_mode;		//step fn when user has pressed and released fwe
	assign re_mode = (reset || match) ? 0 : ((re_mode) ? 1 : w2r);

	wire match;

// the when the filter's done checking with one tone, it needs to let us know
// so we can increment the rom and go on to the next tone
// was done with an FSM; changed into sequential logic for easier debugging
// and was to have been changed back.

	wire fw_rdy,fw_rdy1;
	assign fw_rdy = (reset) ? 1 : fw_rdy1;

	always @(posedge clock) begin
		if (reset || w2r) rom_addr <= 0;
		else if (match || error) rom_addr <= rom_addr;
		else if (fw_rdy) rom_addr <= (rom_addr + 1'b1);
		else rom_addr <= rom_addr;
		end


////////////////////////////////////////////////
/////call to the filter and filtering stuff/////
////////////////////////////////////////////////
// the variables are oddly named, sorry!
	wire [6:0] matchnum;
```

```
	filter_write writeit1(clock, reset, audio_ready, fwe, s2w, w2r, sig_in,
rom_addr, fw_rdy1, matchnum);

	//when matchnum is not the initial value, there is a match.
	assign match = (reset) ? 0 : ((match) ? match : ((matchnum == 0) ? 0 :
1));

	//at the same time, get the matching value for your output
	wire [6:0] notenum;
	assign notenum = (reset) ? 0 : ((notenum) ? notenum : matchnum);

	//// caveat for the error ////
	wire error;
	assign error = (reset) ? 0 : ((rom_addr >= 7'd88) ? 1 : 0);
	  // if we've reached the 88th note and no match, there is an error.

endmodule
```

Appendix 2.3: filter_write.v

```
// finds the differences of the signals
// messy because was still in testing process when scrapped, sorry!
module filter_write(clock, reset, audio_ready, we, s2w, w2r, sig_in,
rom_addr, rdy, matchnum);
	input clock,reset,audio_ready,we,s2w,w2r;
	input [6:0] rom_addr;
	input [7:0] sig_in;
	output [6:0] matchnum;
	output rdy;

// freq2sig makes a 6bit signal for testing; speakers make a 20 bit signal
	wire [5:0] rom_out;
	freq2sig romchk(clock, reset, audio_ready, rom_addr, rom_rdy, rom_out,
rom_cos);
//	speakers romchk(clock,rom_addr,audio_ready,rom_out); // convert #
into signal

// RAM for the difference of test & input signals; FSM-able
	reg [15:0] sd_addr;
	always @(posedge clock)
		begin
		if (reset || s2w || w2r) sd_addr <= 0;
			// after WE is released, we switch to read mode,
			// so reinitialize the addr.
		else sd_addr <= (sd_addr + 1);
			// otherwise, hey keep read/write -ing the bram
		end

	wire [7:0] sd_in, sd_out;
	assign sd_in = (reset) ? 0 : (sig_in-rom_out);
		// this will be stored into a BRAM when WE is high

/* for use with filter
	wire [15:0] sd_addr1,fsd_addr;
	assign sd_addr1 = (reset || we) ? sd_addr : fsd_addr;
	sig_diff sd(sd_addr1,clock,sd_in,sd_out,we);
*/
```

```
      sig_diff sd(sd_addr,clock,sd_in,sd_out,we);

// TEST without filter
      wire [7:0] oout1, oout2;
      assign oout1 = (reset || we) ? 2'd3 : ((sd_addr == 2'd2) ? sd_out :
2'd3);
      assign oout2 = (reset || we) ? 0 : ((sd_addr == 4'd12) ? sd_out : 0);

/*TEST with filter; change RAM addr stuff
filter fthis(clock, reset, w2r, re_mode, sd_out, rom_addr, fsd_addr, fw_rdy,
matchnum, oout);
*/
      wire rdy;
      assign rdy = (reset) ? 0 : ((sd_addr <= 5'd16) ? 0 : 1);

      wire [6:0] matchnum;
      assign matchnum = (reset || we) ? 0 : ((oout1 - oout2 < 1) ? rom_addr :
0);

// test if rom_addr is incrementing
      assign matchnum = (reset || we) ? 0 : ((rom_addr == 6'd40) ? rom_addr :
0);

endmodule
```

Appendix 2.3.i: sig_diff.v

```
      // i/o ports for sig_diff RAM
      module sig_diff (
            addr,
            clk,
            din,
            dout,
            we);    // synthesis black_box

      input [15 : 0] addr;
      input clk;
      input [7 : 0] din;
      output [7 : 0] dout;
      input we;
```

Appendix 2.4: filter.v

```
// filters
module filter(clock, reset, w2r, re_mode, sig_in, rom_addr, saddr, fw_rdy,
matchnum, oout);

      input clock,reset,w2r, re_mode;
      input [19:0] sig_in;
      input [6:0] rom_addr;
      output [8:0] saddr;
      output fw_rdy;
      output [6:0] matchnum;
      output [20:0] oout;


      wire fout;  // value in the filter signal
```

```
      reg [8:0] saddr;  // address for the signal (diff signal)
      reg [3:0] faddr;  // address for the filter signal
      reg  [7:0] oaddr;
      reg [7:0] curr_oaddr;
      reg fw_rdy;

      reg [6:0] new_rom_addr;
      reg [6:0] old_rom_addr;
      always @(posedge clock)
            begin
            old_rom_addr <= reset ? 0 : rom_addr;
            new_rom_addr <= (old_rom_addr != rom_addr);
            end

// was FSM; changed to this for easier debugging.
      always @(posedge clock) begin
            if (reset) begin
                  // initialized to 0
                  saddr <= 9'b0;
                  faddr <= 4'b0;
                  oaddr <= 8'b0;
                  curr_oaddr <= 8'b0;
                  fw_rdy <= 0; end
            else if (new_rom_addr) begin
                  // initialized to 0
                  saddr <= 9'b0;
                  faddr <= 4'b0;
                  oaddr <= 8'b0;
                  curr_oaddr <= curr_oaddr;
                  fw_rdy <= 0; end
            else if (re_mode && (faddr < 3'b111)) begin
                  // once the write is done, and if we haven't filtered all
of faddr
                  // keep saddr stable, inc faddr
                  // and write into subsequent values of oaddr
                  saddr <= saddr;
                  faddr <= faddr + 1;
                  oaddr <= oaddr + 1;
                  curr_oaddr <= curr_oaddr; end
            else if (re_mode &&(saddr < 5'b11111) && (faddr == 3'b111)) begin
                  // after we've finished one cycle of filter,
                  // move on to the next saddr, and start a new cycle of
faddr
                  // and ADD them into the output signal
                  // ONE addr after the one prior
                  saddr <= saddr + 1;
                  faddr <= 4'b0;
                  oaddr <= curr_oaddr + 1;
                  curr_oaddr <= curr_oaddr + 1; end
            else if (re_mode && (saddr >= 5'b11111) && (faddr >= 3'b111))
begin
                  // once we've filtered the whole thing, it's ready for the
next step
                  fw_rdy <= 1; end
            end

      filter_sig1 f(faddr,clock,fout); // ROM of the filter
```

```
//////////////////////////////////////////////////////////////////////
///////////multiply the right parts of the signal and filter//////////
//////////////////////////////////////////////////////////////////////

      wire [20:0] mout;
      assign mout = fout*sig_in;

/////////////////////////////////////////////////
///////////adding back into the output//////////
/////////////////////////////////////////////////

      wire [20:0] oin, oout;
      assign oin = mout + oout;
      // see? put back into the input, the original value in that input and
      // the next section of the convolution.

      wire owe;
      assign owe = (reset || fw_rdy) ? 0 : ((re_mode) ? 1 : 0);
      // when there's a reset or we're done convolving, stop write
      // when there's an fwe release impulse, change to write;
      // keep writing buddy. Or y'know, keep not.

      output_sig write(oaddr,clock,oin,oout,owe); // And the BRAM goes too...

      // TEST
      wire [20:0] oout1, oout2;
      assign oout1 = (reset) ? 8'd160 : (((re_mode) && (oaddr == 4'd12)) ?
oout : 8'd160);
      assign oout2 = (reset) ? 0 : (((re_mode) && (oaddr == 9'h100)) ? oout :
0);

//    wire rdy;
//    assign rdy = (reset) ? 0 : ((oaddr <= 6'd40) ? 0 : 1);

      wire [6:0] matchnum;
      //assign matchnum = (reset || w2r) ? 0 : ((matchnum) ? matchnum :
((oout1 - oout2 <= 7'd125) ? rom_addr : 0));
      assign matchnum = (reset) ? 0 : ((re_mode) ? ((matchnum) ? matchnum :
((rom_addr == 6'd40) ? rom_addr : 0)) : 0);

endmodule
```

Appendix 2.4.i: filter_sig1.coe

```
//coe file for the filter_sig1 ROM
memory_initialization_radix=2;
memory_initialization_vector=

1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
```

Appendix 2.4.ii: outputsig.v

```
// i/o ports for filtered signal RAM
// the width & depth weren't too carefully calculated yet
module output_sig (
      addr,
      clk,
```

```
            din,
            dout,
            we);    // synthesis black_box

      input [7 : 0] addr;
      input clk;
      input [20 : 0] din;
      output [20 : 0] dout;
      input we;
```

APPENDIX 3: {v(t)[sin(w1t)cos(w2t)]} *LPF

Note: This was never completed or tested; it was just designed and written.

Appendix 3.1: lab3.v

```
// Clock at 65MHz
   // use FPGA's digital clock manager to produce a
   // 65MHz clock (actually 64.8MHz)
   wire clock_65mhz_unbuf;//,clock_65mhz;
   DCM vclk1(.CLKIN(clock_27mhz),.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 [7:0] from_ac97_data, to_ac97_data;
   wire ready;

   // AC97 driver
   audio a(clock_65mhz, reset, from_ac97_data, to_ac97_data, ready,
         audio_reset_b, ac97_sdata_out, ac97_sdata_in,
         ac97_synch, ac97_bit_clock);

   // push ENTER button to record, release to playback
   wire playback;
   debounce benter(reset, clock_65mhz, button_enter, playback);

/////////////
//// DDS ////
/////////////

   wire [6:0] chknum1,chknum2,chknum;
   wire [19:0] chksig, chkcos;
   assign chknum1 = 6'd40;
   assign chknum2 = 6'd52;
   wire which;
   assign which = switch[0];
   assign chknum = (which ? chknum1 : chknum2);

   speaker1 chksig(clock_65mhz,1,chknum,rdy,chksig,chkcos);
//   assign to_ac97_data = chksig;

////////////////
//// Record ////
```

```
////////////////
   wire [19:0] din;
   wire [19:0] dout;
   wire [15:0] addr;

   wire rec;
   wire [19:0] from_ac97_data1;

   assign rec = switch[7];
   assign from_ac97_data1 = (rec) ? chksig : from_ac97_data;

   recorder r(clock_65mhz, reset, playback, ready, from_ac97_data1,
to_ac97_data, addr, din, dout, we);
   //recorder r(clock_65mhz, reset, playback, ready, chksig, to_ac97_data,
addr, din, dout, we);

   // call to bram

   lab3ram b(addr,clock_65mhz,din,dout,we);

////////////////
//// Filter ////
////////////////
   wire reset_1;
   assign reset_1 = ~button0;

   wire fwe;                  // start calculation
   assign fwe = ~button1;     // press = fwe high, else low

   wire error;
   wire match;
   wire [6:0] notenum;

   wire [7:0] rom_addr;
   wire fw_rdy;
   wire [13:0] sd_addr;

   rw_fsm fsm(clock_65mhz, ready, reset_1, fwe, to_ac97_data,notenum, error,
match, rom_addr);


///////////////
//// Tests ////
///////////////

   assign analyzer2_clock = ready;
   assign analyzer2_data[15:8] = {fwe, 1'b0, error, 1'b0, match, 4'b0};
   assign analyzer2_data[7:0] = rom_addr[7:0];

   assign analyzer4_clock = ready;
   assign analyzer4_data[15:8] = 0;
   assign analyzer4_data[7:0] = {1'b1,notenum};

   display_16hex d(reset, clock_65mhz, {3'b0,match,3'b0,error,49'b0,notenum},
disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b, disp_data_out);

   // LED
```

```
   assign led[7:3] = 5'b11111;
   assign led[0] = ~match;
   assign led[1] = ~error;
   assign led[2] = ~fwe;
```

Appendix 3.2: speaker1.v

```
module speakers1(clk,note_num,audio_ready,rom_sin,rom_cos);
   input clk;
   input [6:0] note_num;
   input audio_ready;
   output [19:0] rom_sin,rom_cos;

   wire [17:0] freq;          // phase increment value (sets output freq)
   wire        we;            // write enable
   wire [4:0]  dds_a = 5'b0;  // used for multiple output channels (not here)
   wire        rfd;           // not used by DDS, always high
   wire        rdy;           // high when output samples ready
   //wire [15:0]  sine,cosine;      // quadrature outputs of DDS
   wire [16:0] sine,cos;
   reg             clk_dds;         // DDS output clock

   dds_2 dds1(freq,we,dds_a,clk,clk_dds,rdy,sine,cos);

   reg [17:0]  old_freq;
   always @(posedge clk) old_freq <= freq;
   assign      we = ~(freq == old_freq);

   wire [19:0] rom_sin,rom_cos;

   assign rom_sin = {sine,3'b0};
   assign rom_cos = {cos,3'b0};

   // generate DDS clock from ac'97 audio_ready signal
   reg      old_ready;
   always @(posedge clk)
     begin
      old_ready <= audio_ready;
      clk_dds <= audio_ready & ~old_ready;      // one cycle delayed ok
     end
   wire [14:0] freqinc;
   freqincrom fi(note_num,clk,freqinc);
   assign freq = {3'b000,freqinc};
endmodule
```

Appendix 3.3: rw_fsm.v

```
// resets/holds/increments rom_addr as needed for testing
module rw_fsm(clock, audio_ready, reset, fwe, sig_in, notenum, error, match,
rom_addr);
      input clock;            //65MHz clock
      input audio_ready;      //ready for the DDS
      input reset;            //call a reset
      input fwe;              //call for the filter to start
      input [19:0] sig_in;    //input signal

      output [6:0] notenum;   //output the matched notenum
      output error;           //output if there's an error
```

```
	output match;		// to check; output when there's a match
	output [6:0] rom_addr;	// to check; the rom addr we're cycling through


	reg [6:0] rom_addr;

	// some registers that will let us know where we're at
	// could be instantiated with wires, but didn't work o.0

	reg s2w;		//impulse when user first presses fwe
	reg w2r;		//impulse when user first releases fwe
	reg old_fwe;
	always @(posedge clock)
		begin
		if (reset) begin old_fwe <= 0; s2w <= 0; w2r <= 0; end
		else if (~old_fwe && fwe) begin old_fwe <= 1; s2w <= 1; end
		else if (old_fwe && ~fwe) begin old_fwe <= 0; w2r <= 1; end
		else begin s2w <= 0; w2r <= 0; end
		end

	wire calc;		//step fn after user has pressed fwe
	assign calc = (reset || match) ? 0 : ((calc) ? 1 : fwe);
	wire re_mode;		//step fn when user has pressed and released fwe
	assign re_mode = (reset || match) ? 0 : ((re_mode) ? 1 : w2r);

	wire match;

// the when the filter's done checking with one tone, it needs to let us know
// so we can increment the rom and go on to the next tone
// was done with an FSM; changed into sequential logic for easier debugging
// and was to have been changed back.

	wire fw_rdy,fw_rdy1;
	assign fw_rdy = (reset) ? 1 : fw_rdy1;

	always @(posedge clock) begin
		if (reset || w2r) rom_addr <= 0;
		else if (match || error) rom_addr <= rom_addr;
		else if (fw_rdy) rom_addr <= (rom_addr + 1'b1);
		else rom_addr <= rom_addr;
		end


////////////////////////////////////////////////
/////call to the filter and filtering stuff/////
////////////////////////////////////////////////

// call to the filter
	wire [6:0] matchnum;
	filter_write2 writeit2(clock, audio_ready, reset, fwe, s2w, w2r,
re_mode, sig_in, rom_addr, match, matchnum);



endmodule
```

Appendix 3.3: filter_write2.v

```
//multiply the sine & cosine waves
//take samples and determine match
//totally incomplete; not even tested yet
module
filter_write2(clock,audio_ready,reset,we,s2w,w2r,re_mode,sig_in,rom_addr,matc
h,matchnum);
      input clock,audio_ready,reset,we,s2w,w2r,re_mode;
      input [19:0] sig_in;
      input [6:0] rom_addr;
      output match;
      output [6:0] matchnum;

// get cosine with currently testing rom_addr
      wire [19:0] rom_cos;
      speakers1 romchk1(clock,rom_addr,audio_ready,rom_sin,rom_cos);
      wire [3:0] v;
      assign v = 4'd10;

      wire [23:0] sig_product,p_out;
      assign sig_product = v*2*sig_in*rom_cos;

// determine address for the product; store product into BRAM
      wire [15:0] addr,addr1;
      assign addr = (reset || w2r || s2w) ? 0 : ((we) ? (addr + 1) : addr1);
      product go(addr,clock,sig_product,pout,we);

// call filter
      wire [23:0] oout;
      filter fthis(clock, reset, w2r, re_mode, sig_in, rom_addr, addr1,
fw_rdy, oout);

// after filter is done, go through the filtered signal and pick 6 values.
// They should all be around the same value.
      reg [20:0] count;
      always @(posedge clock)
      begin if (reset) count <= 0;
      else if (fw_rdy) count <= count + 1; end


      wire [23:0] val1,val2,val3,val4,val5,val6;
      assign val1 = (reset || we) ? 0 : ((count == 3'b110) ? oout : val1);
      assign val2 = (reset || we) ? 20'd524288 : ((count == 4'd1100) ? oout :
val2);
      assign val3 = (reset || we) ? 1 : ((count == 5'd11000) ? oout : val3);
      assign val4 = (reset || we) ? 20'd524288 : ((count == 6'd110000) ? oout
: val4);
      assign val5 = (reset || we) ? 1 : ((count == 7'd1100000) ? oout :
val5);
      assign val6 = (reset || we) ? 20'd524288 : ((count == 8'd11000000) ?
oout : val6);

      wire [23:0] difference;
      assign difference = (reset || (count < 8'd11000001)) ? 20'd524288 :
(val1 + val3 + val5 - val2 - val4 - val6);
```

```
// determine matches & errors
      wire match,error;
      assign match = (reset || we) ? 0 :
                                      ((match) ? match :
                                      ((rom_addr == 6'd40) ? 1 : 0));
                                      //((difference <= 10'd1000) ? 1 : 0));
      assign error = (reset || we || match) ? 0 :     ((count >= 8'd11000010)
? 1 : 0);

      assign matchnum = (reset || we) ? 0 : ((rom_addr == 6'd40) ? rom_addr :
0);

endmodule
```

Appendix 3.3.i: sine.m

```
function [x t] = sinusoid(omega0, dur, omegaS)


T = 2*pi/omegaS;
nmax = fix(dur/T);
n = [0:nmax];
t = n*T;

x = sin (omega0*t);
```

Appendix 3.3.ii: cosine.m

```
function [x t] = cosine(omega0, dur, omegaS)


T = 2*pi/omegaS;
nmax = fix(dur/T);
n = [0:nmax];
t = n*T;

x = cos (omega0*t);
```

Appendix 3.3.iii: v4.m

```
omegaS = 2*pi*8192;
[sint t] = sine(2*pi*261.63, 5, omegaS);
[cost t] = sine(2*pi*261.63, 5, omegaS);
v = 2;

dtfilter1 = [zeros(1,20472) ones(1,16) zeros(1,20472)];

[sint0 t] = sine(2*pi*261.63, 5, omegaS);
z0 = (v*2).*cost.*sint0;
fn0 = conv(z0,dtfilter1);

[sint1 t] = sine(2*pi*264.63, 5, omegaS);
z1 = (v*2).*cost.*sint1;
fn1 = conv(z1,dtfilter1);

[sint2 t] = sine(2*pi*529.26, 5, omegaS);
z2 = (v*2).*cost.*sint2;
fn2 = conv(z2,dtfilter1);
```

```
subplot(3,2,1),plot(z0);
subplot(3,2,2),plot(fn0);
subplot(3,2,3),plot(z1);
subplot(3,2,4),plot(fn1);
subplot(3,2,5),plot(z2);
subplot(3,2,6),plot(fn2);
```

Appendix 3.3.iv: product.v

```
// i/o ports for the product BRAM
module product (
      addr,
      clk,
      din,
      dout,
      we);    // synthesis black_box

input [15 : 0] addr;
input clk;
input [23 : 0] din;
output [23 : 0] dout;
input we;
```

Appendix 3.4: filter.v

```
// convolve product with LPF
module filter(clock, reset, w2r, re_mode, sig_in, rom_addr, addr1, fw_rdy,
oout);

      input clock,reset,w2r, re_mode;
      input [19:0] sig_in;
      input [6:0] rom_addr;
      output [15:0] addr1;
      output fw_rdy;
      output [23:0] oout;

      wire fout;  // value in the filter signal

      reg [15:0] addr1; // address for the signal (diff signal)
      reg [3:0] faddr;  // address for the filter signal
      reg [13:0] oaddr;
      reg [13:0] curr_oaddr;
      reg fw_rdy;

      reg [6:0] new_rom_addr;
      reg [6:0] old_rom_addr;
      always @(posedge clock)
            begin
            old_rom_addr <= reset ? 0 : rom_addr;
            new_rom_addr <= (old_rom_addr != rom_addr);
            end


      always @(posedge clock) begin
            if (reset) begin
                  // initialized to 0
                  addr1 <= 16'b0;
```

```
                  faddr <= 4'b0;
                  oaddr <= 14'b0;
                  curr_oaddr <= 16'b0;
                  fw_rdy <= 0; end
            else if (new_rom_addr) begin
                  // initialized to 0
                  addr1 <= 16'b0;
                  faddr <= 4'b0;
                  oaddr <= 14'b0;
                  curr_oaddr <= curr_oaddr;
                  fw_rdy <= 0; end
            else if (re_mode && (faddr < 4'b1100)) begin
                  // once the write is done, and if we haven't filtered all
of faddr
                  // keep addr1 stable, inc faddr
                  // and write into subsequent values of oaddr
                  addr1 <= addr1;
                  faddr <= faddr + 1;
                  oaddr <= oaddr + 1;
                  curr_oaddr <= curr_oaddr; end
            else if (re_mode && (addr1 < 16'b1111111111111100) && (faddr ==
4'b1100)) begin
                  // after we've finished one cycle of filter,
                  // move on to the next addr1, and start a new cycle of
faddr
                  // and ADD them into the output signal
                  // ONE addr after the one prior
                  addr1 <= addr1 + 1;
                  faddr <= 4'b0;
                  oaddr <= curr_oaddr + 1;
                  curr_oaddr <= curr_oaddr + 1; end
            else if (re_mode && (addr1 >= 16'b1111111111111100) && (faddr >=
4'b1100)) begin
                  // once we've filtered the whole thing, it's ready for the
next step
                  fw_rdy <= 1; end
            end

      filter_sig1 f(faddr,clock,fout); // ROM of the filter


///////////////////////////////////////////////////////////////////////
//////////multiply the right parts of the signal and filter//////////
///////////////////////////////////////////////////////////////////////
      wire [23:0] oin;
      assign oin = fout*sig_in;

      reg [13:0] oaddr1;
      always @(posedge clock) begin
      if (reset || w2r) oaddr1 <= 0;
      else if (~fw_rdy) oaddr1 <= oaddr;
      else oaddr1 <= oaddr1 + 1; end

      output_sig store(oaddr1,clock,oin,oout,~fw_rdy);

endmodule
```

Appendix 3.4.i: filter_sig1.coe

```
//coe file for the filter_sig1 ROM
memory_initialization_radix=2;
memory_initialization_vector=

1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
```

Appendix 3.4.ii: output_sig.v

```
// i/o ports for output_sig BRAM
module output_sig (
	addr,
	clk,
	din,
	dout,
	we);    // synthesis black_box

input [13 : 0] addr;
input clk;
input [23 : 0] din;
output [23 : 0] dout;
input we;
```