Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.111 Introductory Digital System Laboratory

"Bird's" Music Transcriber

Authors: Alessandro Yamhure, Roberto Carli
T.A: Javier Castro

This report describes the design, implementation and testing of a digital music transcriber. Our system allows musicians to pick a tempo, play any instrument and get a transcription of the notes played on the screen. The system recognizes the value and duration of the notes played, and transforms them into a music sheet. The device is divided into two main subsystems. The first is the *note recognizer* system, which inputs the audio signal, and detects the pitch and duration of the notes played in units of "beats". A pitch detector calculates the pitch of the note and a counter detects its duration by counting tempo units up to a change in pitch. The system also detects rests. The note value and duration are the inputs to the *video display* system, which converts them to the final music sheets on video by use of movable sprites on a non-erasable video memory. First, the system decides which note font to use and the position it must have on screen, then it feeds coordinates and font address to the movable sprites, which finally draw the symbols on screen

Contents

Figures

Tables

1 Overview

It is often quite a hassle and a burden to have to write music, and musicians often express the desire to have a system that will write music for them. Music composers and musicians would be much better off with a system that allows them to play music on an instrument of their choice and have that system analyze and display the music being played in proper musical format, and thus liberate the composer himself from that responsibility. This involves recognizing and displaying the note **pitch** being played (A, F#, B*b* etc) and measuring and displaying the **duration** of the note being played (quarter, half, whole etc). In addition it is convenient for the composer to have the option of selecting a **tempo** from a range of predefined tempos (largo, allegro, presto etc.). The intent of this project is to design and build a system which performs these functions. Moreover the aim of this project is to build a fully functioning music transcriber that writes music it receives in a tempo defined by the user.

From the user's perspective, the composer can communicate with the transcriber in two ways; either he plugs his electric instrument directly into the audio input of the system or communicates to the system through a microphone that is held near the instrument. The musician will see a metronome flashing at the desired tempo on the LEDs of the labkit. He can also hear the beats in headphones in the form of beeps sounding at the previously defined tempo. The user can select a specific tempo from a list of predefined tempos using the switches on the labkit.

Finally, the composer observes the music he is playing in real-time on the monitor connected to the labkit. He observes notes of the right pitch and duration on a background of staves and treble clefs. When the whole screen is filled with music and the last stave has been used, the composer has the option of refreshing the page and starting from the top stave by pressing the *reset* button.

A key element of the project is modularity. The system is made up of three main submodules. First, we have the *Note Recognizer* module, which is responsible for detecting the pitch and duration of the current note being played. It takes in the music audio and the user's choice of tempo as its only two inputs and outputs the *pitch* and *duration* of the note that has just been played along with a *new-note* pulse that signals that a new note is being played. Additionally, it outputs a *sharp* signal which is high when the note being played is an accidental. Finally, it outputs a *beat* pulse which signals the expiration of a beat in the selected tempo.

Next we have the *Video Display* module that takes in the *pitch, duration, sharp* and *new-note* data from the *Note Recognizer* and outputs a video output of appropriate notes on a screen. The position of the note on the stave is determined by the *pitch* and its shape is determined by the value of *duration* and *sharp.* A new note is drawn when *new-note* pulses high.

Finally, we have the *metronome* submodule, which simply takes in the tempo chosen by the user and causes an LED to flash at that tempo and the headphones to sound a beep at that tempo. This is meant to make it easier for the musician to keep tempo while playing his instrument and allows him to accurately communicate the rhythm of the notes to the transcriber system.

This document describes, the design, development, testing and debugging of a digital music transcriber.

2 Description

The obvious way to break down the system is into two main submodules; a **Note Recognizer** module which analyses the audio and recognizes pitch and duration characteristics and a **Video Display** module that uses pitch and duration to draw an appropriate note on a screen equipped with a background of staves and treble clefs. Additionally, a small **Metronome** submodule is responsible for helping the user stay in tempo with the system.

The submodules communicate with each other to form the Music Transcriber as shown in Figure 1.

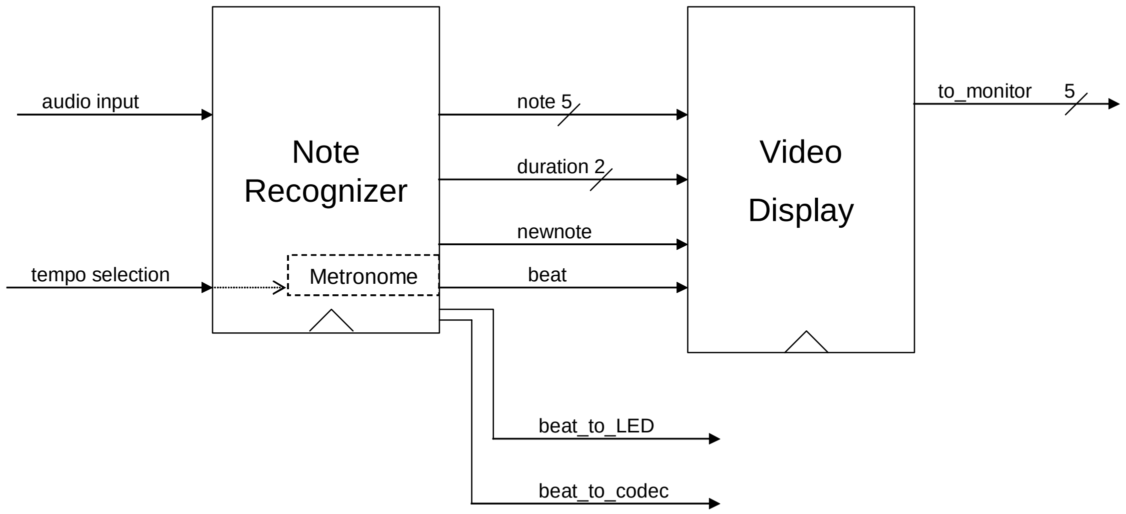

Figure 1: Overall Block Diagram

2.1 Note Recognizer (by Alessandro Yamhure)

The note recognizer module is itself divided into six modules organized into the data path as shown in figure 1.

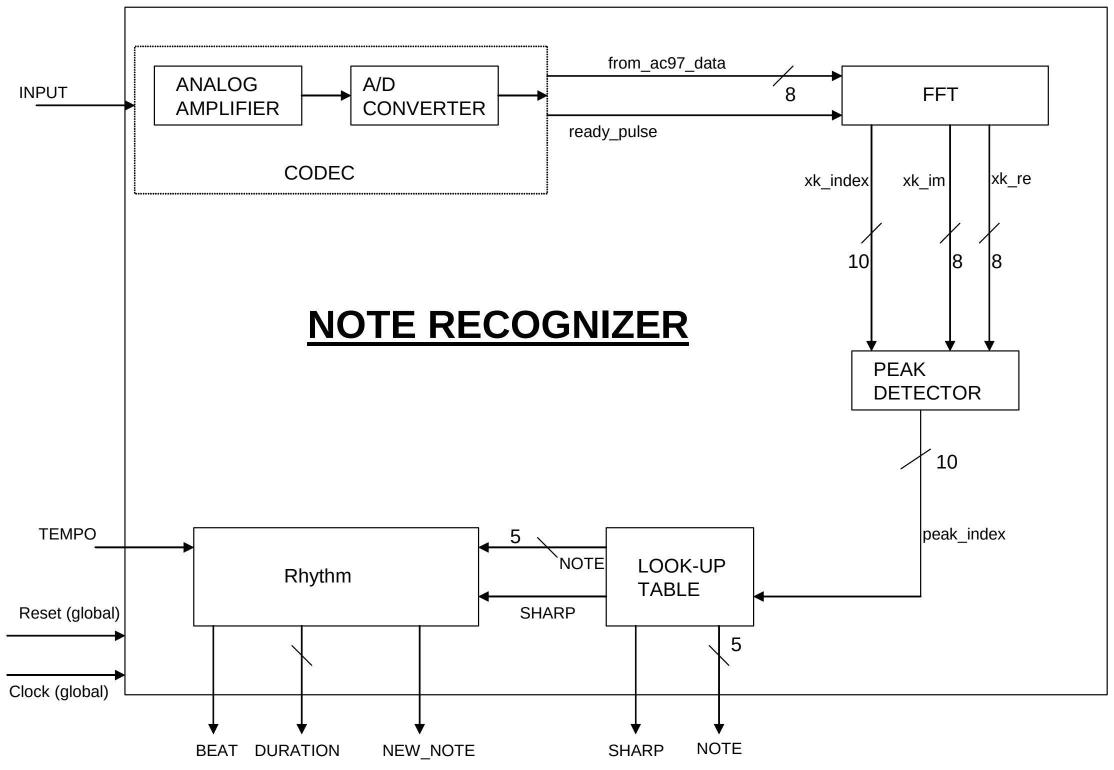

Figure 2: Note Recognizer Block Diagram

2.1.1 Codec

First and foremost, before analyzing can be done, the music audio is passed through a **CODEC**. This will amplify the analog signal and then pass it into an Analog to Digital converter that converts the audio into digital data. The outputs needed from the CODEC are the *ready* and *from_ac97_data.* The ready is converted into a pulse signal which I call *ready_pulse* in the block diagram. The CODEC ADC samples at 48KHz. This means that ready_pulse will pulse at 48KHz and there will be new valid *from_ac97_data* at 48KHz.

2.1.2 FFT

These outputs are passed into the **FFT** module, which is responsible for describing the audio input as a frequency spectrum which can then be analyzed as needed. The FFT core computes an N-point forward DFT where N can be 2^3 – 2^16. The input data is a vector of N complex values represented as two's complement numbers. All memory needed to make the computation is on-chip using either Block RAM or Distributed RAM. The FFT core uses the Cooley-Tukey algorithm for computing the FFT.

The *from_ac97_data* serves as the time domain audio data that will eventually be described in the frequency domain via a Fast Fourier Transform. The *ready_pulse* signal serves as the clock-enable of the FFT. Moreover, the FFT is triggered to input new audio data by the clock-enable input which is appropriately controlled by *ready_pulse.* The FFT gives out many outputs. The outputs useful to this project are *xk_index (*essentially the Discrete Time Frequency) xk_im (the imaginary component associated with that DT frequency) and xk_re (the real component associated with that DT frequency).

One of the design choices made for the **FFT** module involved the point size of the spectrum produced. The tradeoff was subtle. The larger the FFT, the higher the precision and the more precisely the system estimates the frequency being played. On the down side, the larger the FFT, the slower it runs and the longer it takes to compile. Furthermore, a larger FFT will use up more memory. The point size chosen was 4096. It provides us with sufficient precision to distinguish even the lowest notes in our musical range yet not waste run time, compile time or memory. Anything smaller would not be sufficiently precise. Anything larger would be wasteful.

Another design choice was to select the Pipelined, Streaming Input/Output architecture of the FFT core, purely because it allows for continuous data processing which was necessary for our system. This solution pipelines several radix-2 butterfly processing engines to offer continuous data processing. Each engine has its own memory banks to store the input and intermediate data. The core has the ability to simultaneously perform transform calculations on the current frame of data, load input data for the next frame and unload the results of the previous frame.

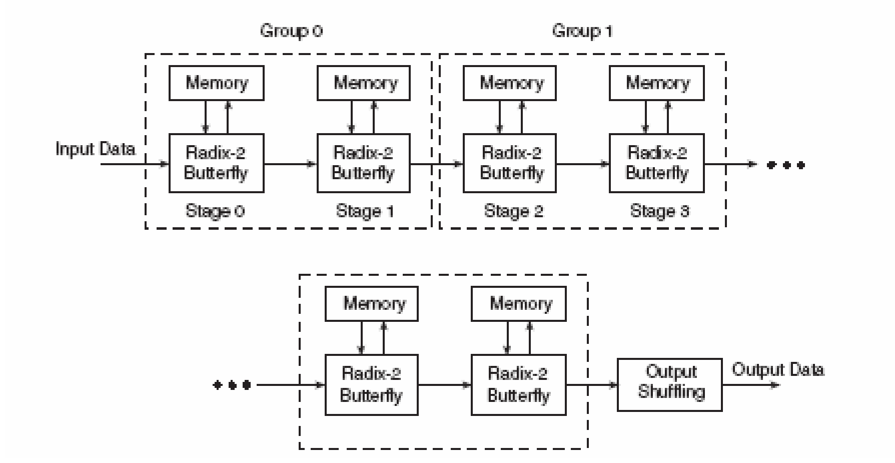

Figure 3: Pipelined, Streaming I/O from [1]

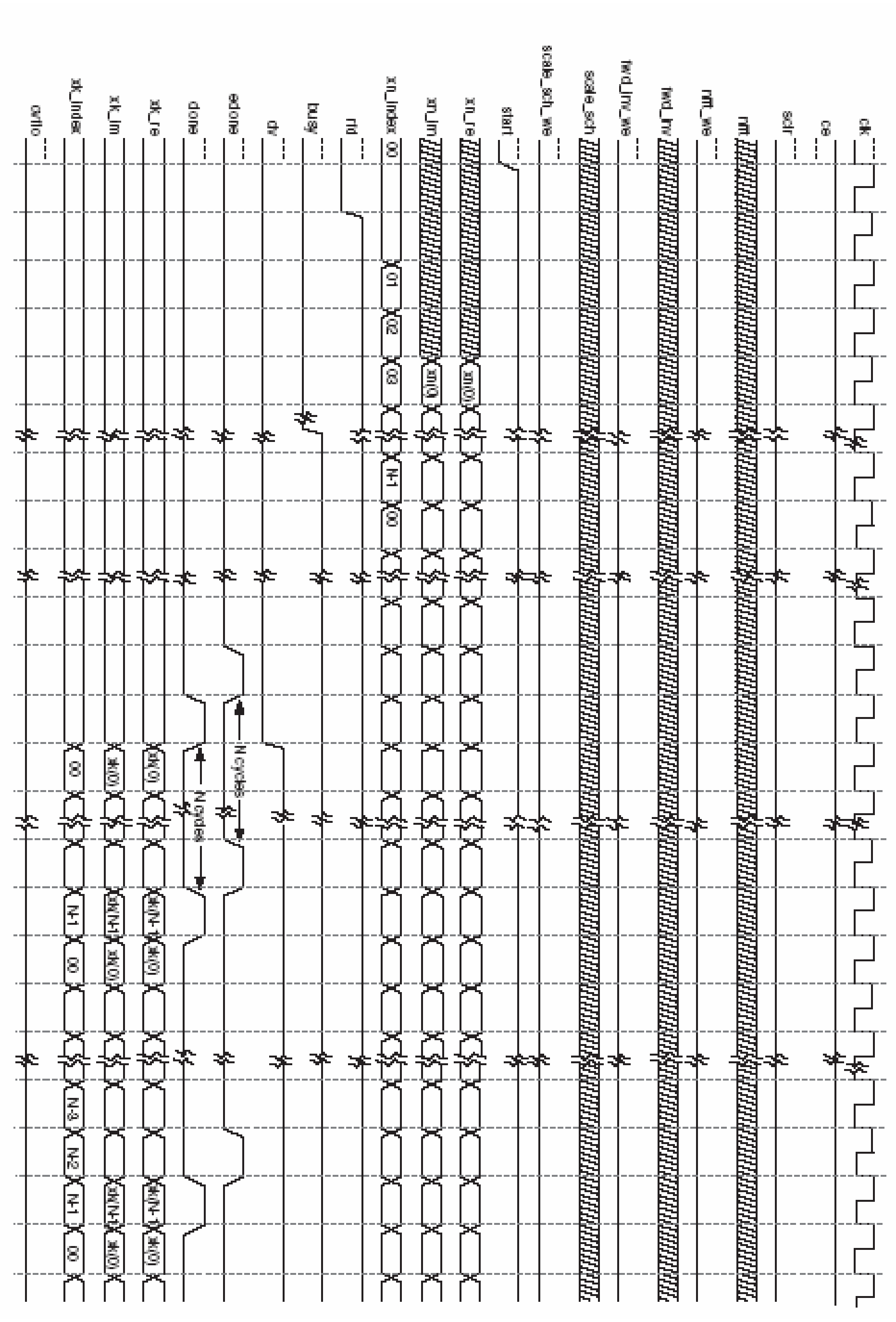

Figure 4: Timing for Continuous Streaming Data from [1]

Another FFT design choice involved the *Scaling Schedule*. Essentially, this is the scaling factor by which the outputs of the FFT are resized. We chose the most conservative scaling schedule in order to avoid overflows and because we are dealing with sinusoidal signals whose frequency spectra consist of sharp, high spikes of large magnitudes.

2.1.3 Peak Detector

In order to recognize the pitch of the note being played, the system needs to detect the pitch frequency of the frequency spectrum produced by the FFT. This is the role of the **Peak Detector** module. It achieves this by scanning through the frequency spectrum essentially measuring the magnitude of each frequency component and comparing it to a running maximum. In more detail, for every DT frequency, the module takes the sum of the square of the real and imaginary parts and compares it to a running maximum (which is initialized to zero). If it is greater than the maximum, that magnitude becomes the new maximum and the associated *xk_index* is stored in a parallel register. Once the peak detector has gone through the audible frequency range, the value of *xk_index* (which holds the DT frequency with the highest peak) is output as *peak_index.*

2.1.4 Look-Up Table

Peak_index is a discrete time frequency that needs to be related to a note pitch (B, D# etc). This is done in two steps by the **Look-Up Table** module. First, it is converted into a frequency in Hertz through some arithmetic. Then, this frequency in Hertz is related to a note pitch through a look-up table based on the data in table 1. As for design decisions, we needed to choose the range over which to "look-up" note values. Our decision was to include all notes in the treble clef in our look-up table since our video output only draws treble clef staves. The system can be easily expanded to include a wider range of notes by simply expanding the look-up table. All frequencies outside the look-up table range are descried as a zero, just like rests.

Note	**Frequency (Hz)**	**Wavelength (cm)**
C_4	261.63	132.
$C^\#_4/D^b_4$	277.18	124.
D_4	293.66	117.
$D^\#_4/E^b_4$	311.13	111.
E_4	329.63	105.
F_4	349.23	98.8
$F^\#_4/G^b_4$	369.99	93.2
G_4	392.00	88.0
$G^\#_4/A^b_4$	415.30	83.1
A_4	440.00	78.4

$A^{\#}_4/B^b_4$	466.16	74.0
B_4	493.88	69.9
C_5	523.25	65.9
$C^{\#}_5/D^b_5$	554.37	62.2
D_5	587.33	58.7
$D^{\#}_5/E^b_5$	622.25	55.4
E_5	659.26	52.3
F_5	698.46	49.4
$F^{\#}_5/G^b_5$	739.99	46.6
G_5	783.99	44.0
$G^{\#}_5/A^b_5$	830.61	41.5
A_5	880.00	39.2
$A^{\#}_5/B^b_5$	932.33	37.0
B_5	987.77	34.9
C_6	1046.50	33.0
$C^{\#}_6/D^b_6$	1108.73	31.1
D_6	1174.66	29.4
$D^{\#}_6/E^b_6$	1244.51	27.7
E_6	1318.51	26.2
F_6	1396.91	24.7
$F^{\#}_6/G^b_6$	1479.98	23.3
G_6	1567.98	22.0
$G^{\#}_6/A^b_6$	1661.22	20.8
A_6	1760.00	19.6
$A^{\#}_6/B^b_6$	1864.66	18.5
B_6	1975.53	17.5

Table 1: Frequencies of Musical Notes from [2]

2.1.5 Rhythm

The **Rhythm** module is responsible for measuring the duration of a note and signaling the arrival of a new note and providing a *beat* signal whenever a beat expires. It takes in *note* and *sharp* from the **look-up table** and *tempo* from the user and outputs *duration, beat* (depends only on *tempo)* and *new_note.*

The *tempo* input determines the length of a beat which in turn determines an internal parameter in the module which corresponds to the number of *clock_27MHz* cycles that occur in one beat. Combining this parameter with a counter incremented every clock cycle, whenever the value of *note* or *sharp* change from **look-up table,** *new_note* goes high for one clock cycle and based on the value of the counter relative to the beat parameter, the module calculates and outputs the *duration* of the note that has just expired. Lastly, every time a beat expires (when the counter reaches the beat parameter value), the signal *beat* goes high for one clock cycle.

2.1.6 Metronome: *(By Roberto Carli & Alessandro Yamhure)*

This module is responsible for producing a flashing LED and a beeping sound in the headphones/speakers based on a *beat* pulse from the **Note Recognizer** module. This was achieved by converting the *beat* pulse into an "up-down" pulse that changes from high to low or vice versa when *beat* pulses high. This "up-down" derivative of *beat* then drives the LED and headphones/speakers to produce a useful metronome.

2.2 Video Display

The video display system transforms information about the value and duration of individual notes into a screen display of music notation with clefs, notes and rests. The inputs to the display system are values for *note*, *duration*, and *sharp*, carrying information about the note, and *newnote*, which determines when to use the previous information again to display a new note on screen. All the inputs are coming from the note recognizer.

The video display system operates in two main steps:

- Tracking: determine the position and shape of the notes, keeping track of the current position of music on the sheet.
- Displaying: use moveable sprites to display notes and clefs at the given positions.

Both the tracking and displaying steps for the system use several modules, which are described in detail below.

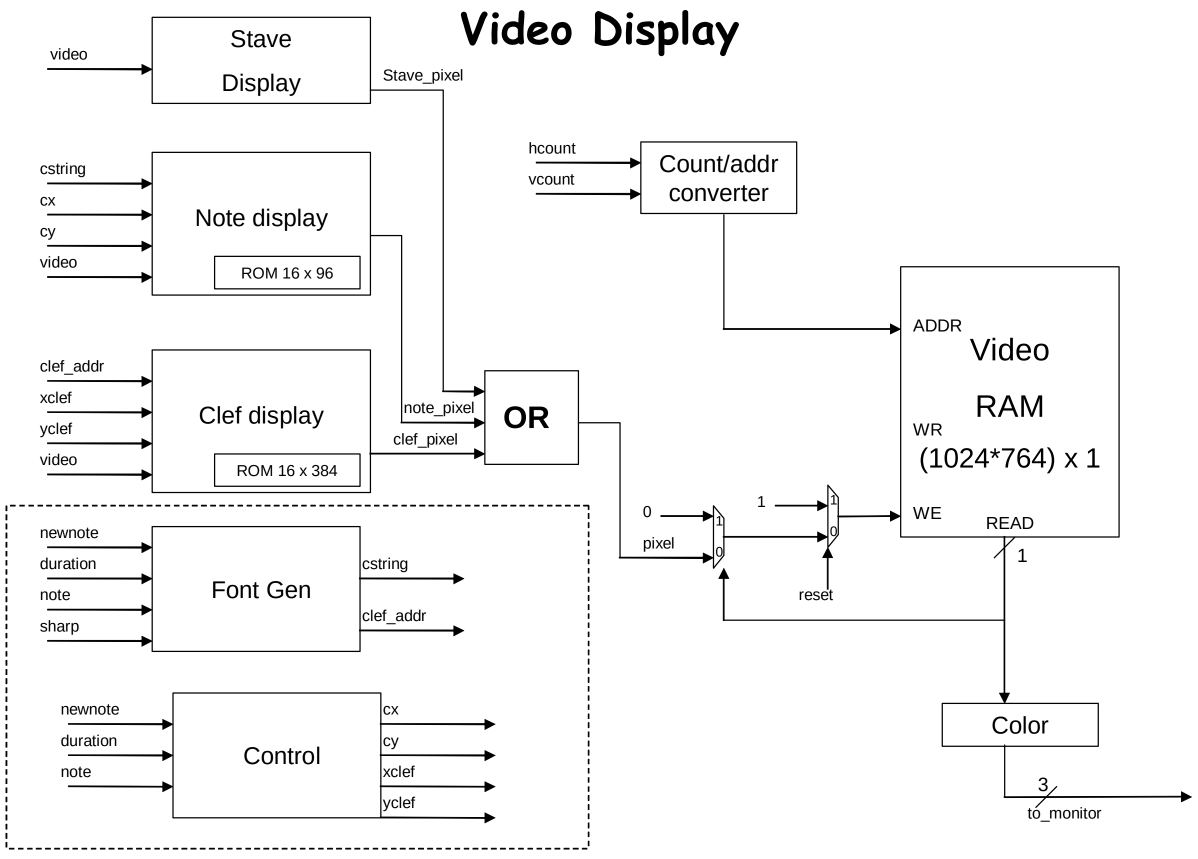

Figure 5: Video Display Block Diagram

Tracking

2.2.1 Control Module

The **control** module is responsible for assigning the correct position to the incoming notes, thus keeping track of the current position in the music sheet, and for stalling the system once a page is full, so that the musician can look at the transcription of his work. In music, the vertical position of a note with respect to the stave is what identifies the note itself, so the control module takes into account the value of the notes in order to output their vertical positions.

The module keeps track of the horizontal position, and of a fixed vertical position, which increases only when a line is complete. The horizontal distance between the notes is a function of their duration, so that their distance gives a good visual understanding of the rhythm (there is a longer space after a full note/rest than after a quarter note/rest). The vertical position is determined by the fixed vertical index minus a variable shift, which depends on the value of the note, or minus a fixed shift for rests, which have to be at given positions on the stave.

2.2.2 Font Generation Module

The **fontgen** module is substantially a lookup table, which outputs the memory addresses for the internal ROMs of the note displayer and clef displayer modules, so that they access the correct font for display. For the note memory address *cstring*, the module outputs two addresses, one for the note and one for an eventual sharp. The module checks whether to display a rest (*note*=0) or a note, then checks for *duration* (full, ¾, half, quarter), and finally for *sharp*. The clef address is hardwired to the treble clef font, as the system only displays notes in treble clef.

Displaying

The displaying step is object-oriented: it is based on moveable sprites for clefs and notes, and s black and white, non-erasable video memory **videoram** which allows minimal hardware requirements.

The system has three sprite modules: a stave display **stave**, a note display **char_string-_display** and a clef display **clef_display**, all producing a 1-bit (black & white) pixel signal. While the stave display is fixed and always present, the note and clef display modules can display different fonts (specified by the font generation module) at given coordinates (specified by the controller). The pixel signals produced are superimposed by an OR gate, and sent to a video memory. The output of the video memory is then transformed into a 3-bit pixel signal that is sent to the monitor.

2.2.3 Note Displayer

The note displayer inputs coordinates *cx* and *cy*, an 8-bit value for two memory addresses *cstring* and the video signals. It outputs a pixel signal for a string of two fonts, specified by the two addresses, at the given coordinates. The module is in many ways similar to an ASCII string displayer, and it outputs a string of always two characters. In the embedded ROM there are a total of 16 fonts for the various note values, rest values and sharp. Each font is an 8x12 image, stored in 12 words of 8 bits each. The total ROM thus measures 8x192, but the module takes in 4-bit addresses and scales them to retrieve the appropriate font, then advances by one line at every hsync, so as to display it on the next line on the monitor. Finally, the module performs some logic with the coordinates and the hcount, vcount video signals, for displaying the note in the correct place.

2.2.4 Clef Displayer

The clef displayer works very similarly to the note displayer, however it only inputs one font address for the clef, and its ROM stores images of size 16x24, as clefs are larger and graphically more complex than notes and rests.

2.2.5 Stave Displayer

The stave displayer is a sprite which outputs only horizontal lines, thus it uses *vcount* as a reference, and draws lines at the correct vertical positions. The displayer inputs parameters for the thickness of the lines, the distance between staves and other graphical details, then it uses combinational logic to determine when to write the *pixel*.

2.2.6 Video-to-Address Converter

This module uses the video signals and converts them to a memory address for the **video memory**, so that there is a 1-to-1 correspondence between the *hcount*, *vcount* coordinate signals and the memory address. Additionally, the size of the memory is only limited to 1024x768, the visible area of the screen, while *hcount* goes beyond this value, thus the module makes sure that only the pixels in the visible area are stored in the memory.

2.2.7 Video Memory

A video memory stores the *pixel* data for the 1024x768 pixels of the screen's visible area. For limiting size, the memory stores only 1-bit *pixel* values, thus producing a monochromatic image. The reason why the system needs a memory is so that the video output is non-erasable unless the reset button is pressed. This choice implies a tradeoff for some expandability, such as any animation, however it allows the system to be very lightweight as it operates with only three sprites, their output staying on screen even when they move. The non-erasable feature is obtained by hardwiring the memory input to a 1, while using reset and the output to change the write-enable control (*see Figure).*

When the memory outputs a 1 for a given *pixel*, the system cannot overwrite that 1, whereas a 0 can be overwritten. At reset, the memory refreshes.

2.2.8 XVGA

The **XVGA** module (written by Nathan Ickes) works on a 65 MHz clock, and generates all the signals required for outputting a pixel video signal to the monitor, at a refresh rate of 60Hz and a resolution of 1024x768. In particular, the signals *hcount*, *vcount*, *hsync* and *vsync* are used by all the video-related modules in the system, as they represent the reference points for the generation of *pixel* signals.

3 Testing and debugging

One of the key advantages of building a modular system is the ability to test each submodule separately and independently, which makes testing independent of other modules and debugging easier and faster.

3.1 Note Recognizer: (by Alessandro Yamhure)

A key aid in the testing of the modules involved in note recognition was the signal generator which produces sinusoidal waves of a desired frequency.

3.1.1 FFT

To test that the **FFT** module was functioning correctly, I would input various frequencies produced by the signal generator and observe the outputs of the FFT on the logic analyzer. There was not a significant amount of debugging for the FFT but understanding how to interface with it and how to interpret and use its outputs took up a decent amount of time. An appropriate output would be a flat spectrum with spikes at the positive and negative discrete time frequencies of the wave being passed in.

3.1.2 Peak Detector

Once the FFT was known to work correctly, I used its outputs (corresponding to a given sinusoidal input) to test the **Peak Detector** module. The output of this module was observed on the LEDs of the labkit. An appropriate output would be the DT frequency of the sinusoidal signal being passed in.

3.1.3 Look-Up Table

In order to test the **Look-Up Table,** I made use of the two previous working modules and observed the outputs on the LEDs of the labkit. As input I used the sound from various musical instruments (electric keyboard, harmonica) as well as the signal generator. An appropriate output involved the correct values of *note* and *sharp* for a given pitch played on an instrument or a selected frequency from the signal generator.

3.1.4 Rhythm

The **Rhythm** module was tested using Xilinx test benches and observing the values of *new_note*, *duration* and *beat* on the LEDs of the labkit. Once *beat* was working correctly, constant values of *note* and *sharp* were passed into the module for a chosen number of beats before being changed. The values of *new_note* and *duration* were observed. Appropriate results included a *new_note* signal that pulses high when the value of EITHER of its inputs (*sharp* or *note)* changes along with a correct *duration* corresponding to the number of beats that expired while those inputs lasted. One very useful testing and debugging technique made use of an **Updown Module** that converts a pulse into a signal that alternates between high and low whenever the original signal pulses. These derived signals were mush easier to observe and made testing and debugging significantly faster and easier for pulse signals like *new_note* and *beat.*

When using Test Benches, I made the *beat* last for one clock cycle in order to make the transitions of *new_note* and *duration* more observable. The expected behavior involves *new_note* pulsing high every 4 clock cycles if the value of the input *note* does not change. If *note* changes, *new_note* pulses and *duration* outputs with the number of clock cycles that expired between the different values of the *note* input.

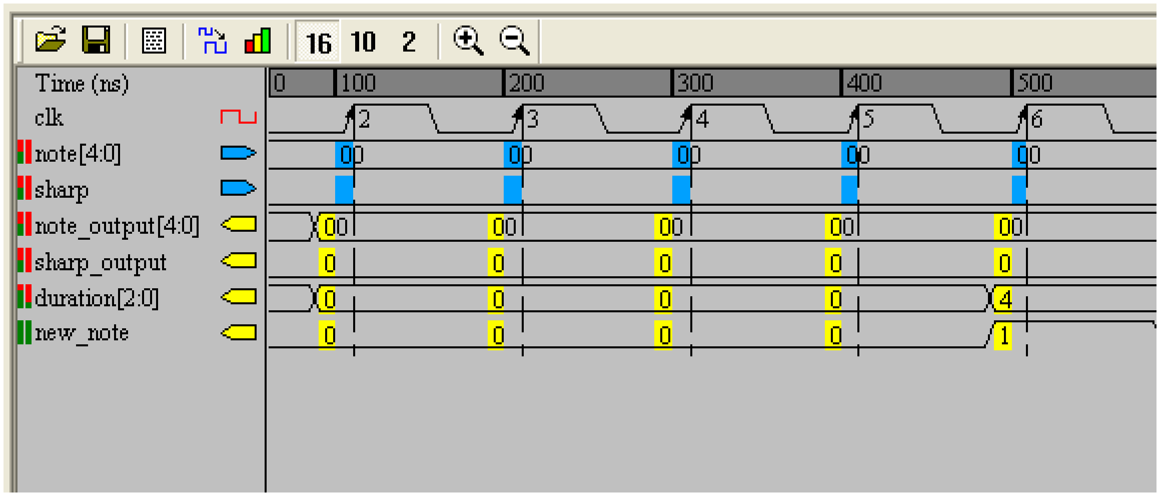

Figure 6: Rhythm module detecting a whole note (*note* does not change)

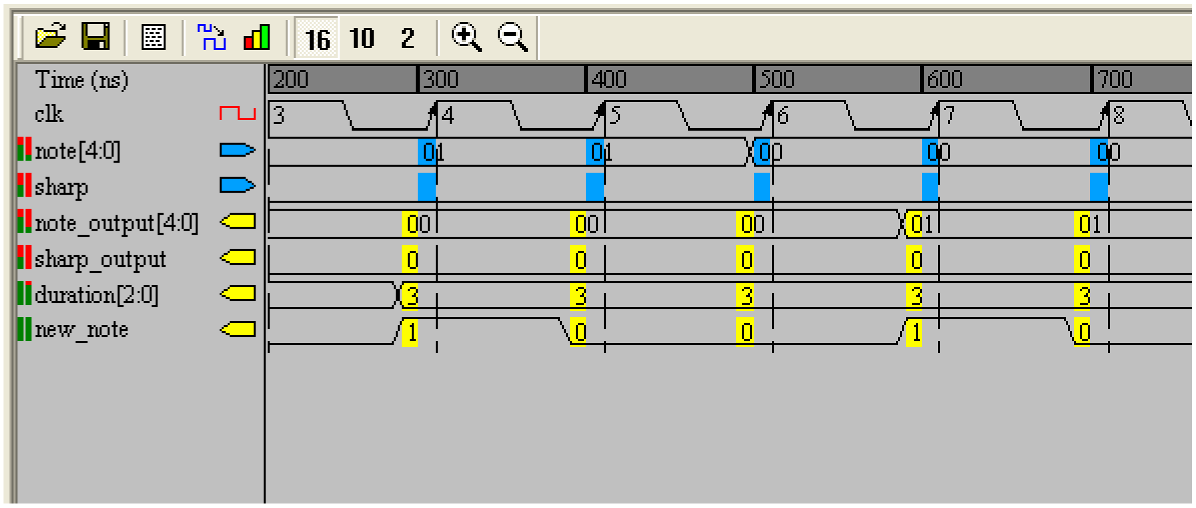

Figure 7: Rhythm module detects a note of duration 3

3.2 Video Display *(Roberto Carli)*

The video display system was implemented following a top-down order. The output to computer monitor was the best way of testing the modules, thus the first step was to implement the **XVGA** and test video output. The second step was to setup the video RAM, produce the correct addressing, and testing various solutions for non-erasable output which could be reset. Moveable test sprites were used for experimenting with the memory. The third step was to create the three main sprites for staves, notes and clefs, and to manually draw and synthesize font memories for the last two modules. The tracking part of the system was the final step. Once the functionality of the various video components was known, the **control** and **fontgen** modules were designed and the general functioning of the system was tested by externally inputting the note/duration values.

3.2.1 Multiple Clocks

Using a top-down order of implementation was very useful for testing, as it was possible to use the monitor output for the debugging of every module. However, the testing phase still presented several obstacles and problems to be addressed. Mainly, for technical reasons, the system works on two clocks, a 65 MHz clock for all the modules dealing with video signals, and a 27 MHz clock for the rest of the system. Even after appropriate synchronization, the use of two clocks caused some problems of module interfacing that had to be addressed.

3.2.2 Movable Sprites

Implementing the movable sprites for the note and clef display was time-consuming and at times problematic. In particular, they exhibited several clock problems, and the memories had to be instantiated with typewritten COE files. Finally, the sprites exhibit long propagation delays which sometimes still cause moderate glitches in the video output.

3.2.3 Control and Font Generation

The testing of the control logic and the font generation was performed in two steps. First, the modules were simulated to ensure the correct functioning of the internal logic. Then, the modules were tested by controlling the movable sprites, thus also testing the addressing and content of the sprites' internal ROMs.

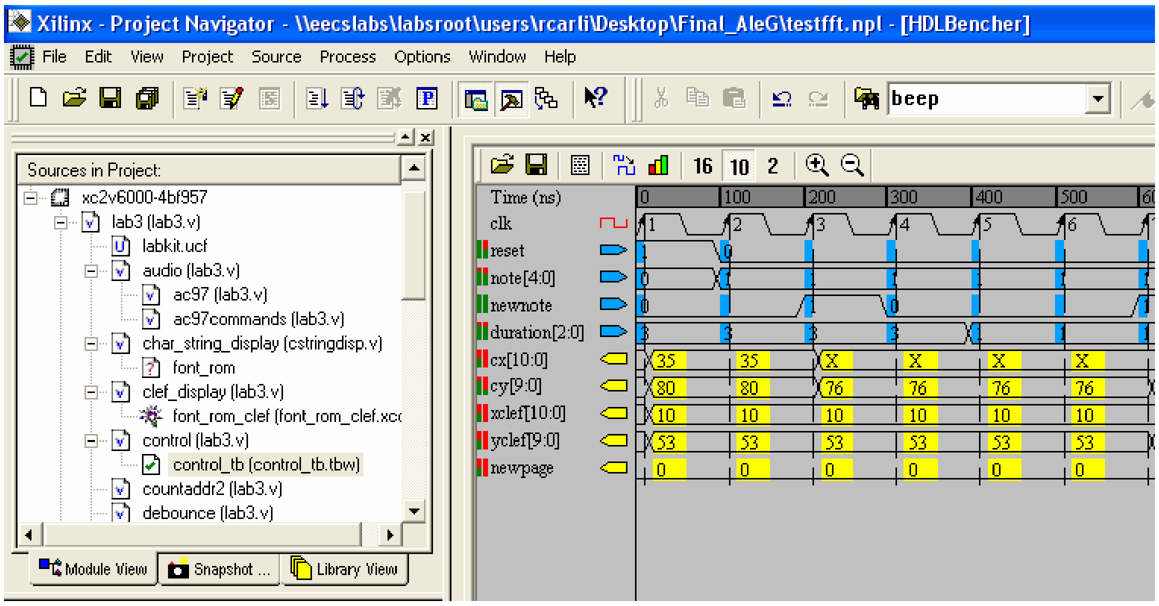

Figure 8: simulation for control logic debugging

3.3 Integration

After independent testing, the two parts of the system were integrated into a single labkit. The note recognizer was coded on a previous lab involving audio recording, whereas the display system started from a video-oriented lab. The video display system was transferred to the note recognizer labkit; the integration caused some problems with the instantiations of VGA signals and the 65 MHz synchronized clock on the new labkit. The integration of the system, however, was done at a relatively early stage, as a working video output was the best and fastest way of debugging and fine-tuning the mechanics of the note-recognizer.

3.4 Final Testing

A very long time was devoted to the testing and fine-tuning of the final system, particularly relating to the characteristics of different instruments. There is a great difference between inputting a synthesized signal and a real instrument voice captured by a microphone. In particular, the characteristics for the attack, sustain and stability of the note change widely between instruments, and the device was tuned to capture efficiently a wide range of instrument sounds. The leniency of the pitch recognition was relaxed further to account for instruments that have a vibrato sound (oscillation of pitch). Duration, instead, presents a tradeoff in its leniency, as a very precise system can miss sustained notes, while a lenient system usually misses rests and fast changes. The position of the microphone and the quality of the audio capture also influences the final results. In conclusion, a very thorough testing was required to find the right balance for the best system performance, while also having to deal with the physical limitations and latencies of the FFT module.

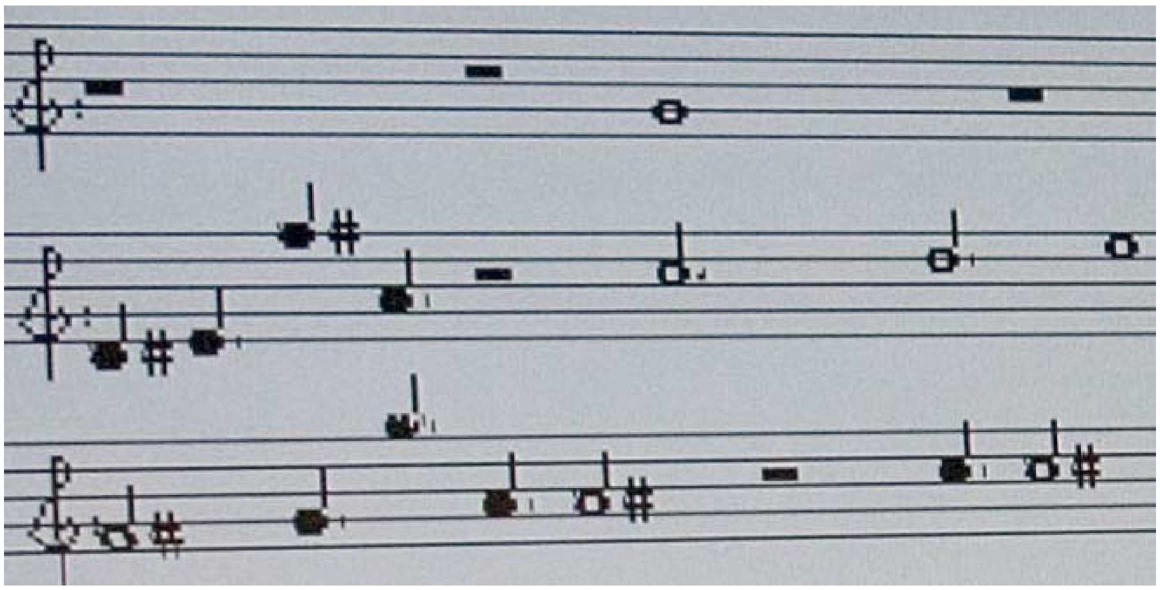

Figure 9: The System's video output

4 Conclusion

Designing the music transcriber was a hard but fascinating experience. It required initiative in dealing with new components, such as the FFT or the video sprites, determination in implementing efficient design solutions, such as the lenient duration or the non-erasable output, and lots of patience in debugging. The final results, however, were very satisfactory and paid back for the hard work put into the project. Our music transcriber is an enjoyable and reliable way of writing down the music of a solo instrument

The system presents some problems and limitations, but it also offers plenty of room for easy upgrades. The video output is structured in such a way that animations, such as a scroll-down page, are not possible, however it allows for an easy introduction of additional features (different clefs, chords, higher note definition) by just adding a new sprite and minor modifications to the control logic. The note recognizer is limited mainly by the propagation delays of the externally designed FFT module, which limits the highest resolution for note duration and at times can miss rests in the input. However, the system can be easily expanded for a higher range of notes and durations, for chords, and it is lenient enough to recognize vibrato notes, whose pitch oscillates.

5 Citations

[1] Xilinx Logicore Fast Fourier Transform v3.0 Product Specification
[2] http://www.phy.mtu.edu

6 Acknowledgments

We would like to thank a few outstanding individuals without whom this project would not have been possible.

First and foremost, the entire 6.111 staff, in particular Ike and Chris, for suggesting incredible solutions and making us believe we thought of them ourselves – a memorable semester indeed.

Also, we thank Charlie “Bird” Parker, Dave Brubeck, Miles Davis and John Coltrane for their great contributions to music. If they could play on it, the system would melt down for the passion.

7 Appendices

```
///////////////////////////////////////////////////////////////////////////////
//
// Switch Debounce Module
//
///////////////////////////////////////////////////////////////////////////////


module debounce (reset, clock, noisy, clean);

   input reset, clock, noisy;
   output clean;

   reg [18:0] count;
   reg new, clean;

   always @(posedge clock)
     if (reset)
       begin
        count <= 0;
        new <= noisy;
        clean <= noisy;
       end
     else if (noisy != new)
       begin
        new <= noisy;
        count <= 0;
       end
     else if (count == 270000)
       clean <= new;
     else
       count <= count+1;

endmodule

///////////////////////////////////////////////////////////////////////////////
//
// bi-directional monaural interface to AC97
//
///////////////////////////////////////////////////////////////////////////////

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);

   input clock_27mhz;
   input reset;
   output [7:0] audio_in_data;
   input [7:0] 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;

   wire [4:0] volume;
   wire source;
   assign volume = 4'd22;  //a reasonable volume value
   assign source = 1;    //mic

   wire [7:0] command_address;
   wire [15:0] command_data;
   wire command_valid;
   wire [19:0] left_in_data, right_in_data;
   wire [19:0] left_out_data, right_out_data;

   reg audio_reset_b;
   reg [9:0] reset_count;

   //wait a little before enabling the AC97 codec
   always @(posedge clock_27mhz) begin
      if (reset) begin
         audio_reset_b = 1'b0;
         reset_count = 0;
      end else if (reset_count == 1023)
        audio_reset_b = 1'b1;
      else
        reset_count = reset_count+1;
   end

   ac97 ac97(ready, command_address, command_data, command_valid,
             left_out_data, 1'b1, right_out_data, 1'b1, left_in_data,
             right_in_data, ac97_sdata_out, ac97_sdata_in, ac97_synch,
             ac97_bit_clock);

   ac97commands cmds(clock_27mhz, ready, command_address, command_data,
                     command_valid, volume, source);


   assign left_out_data = {audio_out_data, 12'b000000000000};
   assign right_out_data = left_out_data;

   //arbitrarily choose left input, get highest-order bits
   assign audio_in_data = left_in_data[19:12];

endmodule

// assemble/disassemble AC97 serial frames
module ac97 (ready,
             command_address, command_data, command_valid,
             left_data, left_valid,
             right_data, right_valid,
             left_in_data, right_in_data,
             ac97_sdata_out, ac97_sdata_in, ac97_synch, ac97_bit_clock);

   output ready;
   input [7:0] command_address;
   input [15:0] command_data;
   input command_valid;
   input [19:0] left_data, right_data;
   input left_valid, right_valid;
   output [19:0] left_in_data, right_in_data;

   input ac97_sdata_in;
   input ac97_bit_clock;
   output ac97_sdata_out;
   output ac97_synch;

   reg ready;

   reg ac97_sdata_out;
   reg ac97_synch;

   reg [7:0] bit_count;

   reg [19:0] l_cmd_addr;
   reg [19:0] l_cmd_data;
   reg [19:0] l_left_data, l_right_data;
   reg l_cmd_v, l_left_v, l_right_v;
   reg [19:0] left_in_data, right_in_data;

   initial begin
      ready <= 1'b0;
      // synthesis attribute init of ready is "0";
      ac97_sdata_out <= 1'b0;
      // synthesis attribute init of ac97_sdata_out is "0";
      ac97_synch <= 1'b0;
      // synthesis attribute init of ac97_synch is "0";

      bit_count <= 8'h00;
      // synthesis attribute init of bit_count is "0000";
      l_cmd_v <= 1'b0;
      // synthesis attribute init of l_cmd_v is "0";
      l_left_v <= 1'b0;
      // synthesis attribute init of l_left_v is "0";
      l_right_v <= 1'b0;
      // synthesis attribute init of l_right_v is "0";

      left_in_data <= 20'h00000;
      // synthesis attribute init of left_in_data is "00000";
      right_in_data <= 20'h00000;
      // synthesis attribute init of right_in_data is "00000";
   end

   always @(posedge ac97_bit_clock) begin
      // Generate the sync signal
      if (bit_count == 255)
        ac97_synch <= 1'b1;
      if (bit_count == 15)
        ac97_synch <= 1'b0;

      // Generate the ready signal
      if (bit_count == 128)
        ready <= 1'b1;
      if (bit_count == 2)
```

```
      ready <= 1'b0;

    // Latch user data at the end of each frame. This ensures that the
    // first frame after reset will be empty.
    if (bit_count == 255)
      begin
         l_cmd_addr <= {command_address, 12'h000};
         l_cmd_data <= {command_data, 4'h0};
         l_cmd_v <= command_valid;
         l_left_data <= left_data;
         l_left_v <= left_valid;
         l_right_data <= right_data;
         l_right_v <= right_valid;
      end

    if ((bit_count >= 0) && (bit_count <= 15))
      // Slot 0: Tags
      case (bit_count[3:0])
        4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
        4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
        4'h2: ac97_sdata_out <= l_cmd_v;   // Command data valid
        4'h3: ac97_sdata_out <= l_left_v;  // Left data valid
     4'h4: ac97_sdata_out <= l_right_v; // Right data valid
        default: ac97_sdata_out <= 1'b0;
      endcase

    else if ((bit_count >= 16) && (bit_count <= 35))
      // Slot 1: Command address (8-bits, left justified)
      ac97_sdata_out <= l_cmd_v ? l_cmd_addr[35-bit_count] : 1'b0;

    else if ((bit_count >= 36) && (bit_count <= 55))
      // Slot 2: Command data (16-bits, left justified)
      ac97_sdata_out <= l_cmd_v ? l_cmd_data[55-bit_count] : 1'b0;

    else if ((bit_count >= 56) && (bit_count <= 75))
      begin
         // Slot 3: Left channel
         ac97_sdata_out <= l_left_v ? l_left_data[19] : 1'b0;
         l_left_data <= { l_left_data[18:0], l_left_data[19] };
      end
    else if ((bit_count >= 76) && (bit_count <= 95))
      // Slot 4: Right channel
         ac97_sdata_out <= l_right_v ? l_right_data[95-bit_count] : 1'b0;
    else
      ac97_sdata_out <= 1'b0;

    bit_count <= bit_count+1;

 end // always @ (posedge ac97_bit_clock)

 always @(negedge ac97_bit_clock) begin
    if ((bit_count >= 57) && (bit_count <= 76))
      // Slot 3: Left channel
      left_in_data <= { left_in_data[18:0], ac97_sdata_in };
    else if ((bit_count >= 77) && (bit_count <= 96))
      // Slot 4: Right channel
      right_in_data <= { right_in_data[18:0], ac97_sdata_in };
 end
```

```
endmodule

// issue initialization commands to AC97
module ac97commands (clock, ready, command_address, command_data,
                     command_valid, volume, source);

   input clock;
   input ready;
   output [7:0] command_address;
   output [15:0] command_data;
   output command_valid;
   input [4:0] volume;
   input source;

   reg [23:0] command;
   reg command_valid;

   reg old_ready;
   reg done;
   reg [3:0] state;

   initial begin
      command <= 4'h0;
      // synthesis attribute init of command is "0";
      command_valid <= 1'b0;
      // synthesis attribute init of command_valid is "0";
      done <= 1'b0;
      // synthesis attribute init of done is "0";
      old_ready <= 1'b0;
      // synthesis attribute init of old_ready is "0";
      state <= 16'h0000;
      // synthesis attribute init of state is "0000";
   end

   assign command_address = command[23:16];
   assign command_data = command[15:0];

   wire [4:0] vol;
   assign vol = 31-volume;

   always @(posedge clock) begin
      if (ready && (!old_ready))
        state <= state+1;

      case (state)
        4'h0: // Read ID
          begin
             command <= 24'h80_0000;
             command_valid <= 1'b1;
          end
        4'h1: // Read ID
          command <= 24'h80_0000;
        4'h3: // headphone volume
          command <= { 8'h04, 3'b000, vol, 3'b000, vol };
        4'h5: // PCM volume
          command <= 24'h18_0808;
        4'h6: // Record source select
```

```
          command <= 24'h1A_0000; // microphone
        4'h7: // Record gain = max
       command <= 24'h1C_0F0F;
        4'h9: // set +20db mic gain
          command <= 24'h0E_8048;
        4'hA: // Set beep volume
          command <= 24'h0A_0000;
        4'hB: // PCM out bypass mix1
          command <= 24'h20_8000;
        default:
          command <= 24'h80_0000;
      endcase // case(state)

      old_ready <= ready;

   end // always @ (posedge clock)

endmodule // ac97commands

///////////////////////////////////////////////////////////////////////////////
//
// generate PCM data for 750hz sine wave (assuming f(ready) = 48khz)
//
///////////////////////////////////////////////////////////////////////////////

module tone750hz (clock, ready, pcm_data);

   input clock;
   input ready;
   output [19:0] pcm_data;

   reg rdy, old_ready;
   reg [8:0] index;
   reg [19:0] pcm_data;

   initial begin
      old_ready <= 1'b0;
      // synthesis attribute init of old_ready is "0";
      index <= 8'h00;
      // synthesis attribute init of index is "00";
      pcm_data <= 20'h00000;
      // synthesis attribute init of pcm_data is "00000";
   end

   always @(posedge clock) begin
      if (rdy && ~old_ready)
     index <= index+1;
      old_ready <= rdy;
      rdy <= ready;
   end

   // one cycle of a sinewave in 64 20-bit samples
   always @(index) begin
      case (index[5:0])
        6'h00: pcm_data <= 20'h00000;
        6'h01: pcm_data <= 20'h0C8BD;
        6'h02: pcm_data <= 20'h18F8B;
        6'h03: pcm_data <= 20'h25280;
        6'h04: pcm_data <= 20'h30FBC;
        6'h05: pcm_data <= 20'h3C56B;
        6'h06: pcm_data <= 20'h471CE;
        6'h07: pcm_data <= 20'h5133C;
        6'h08: pcm_data <= 20'h5A827;
        6'h09: pcm_data <= 20'h62F20;
        6'h0A: pcm_data <= 20'h6A6D9;
        6'h0B: pcm_data <= 20'h70E2C;
        6'h0C: pcm_data <= 20'h7641A;
        6'h0D: pcm_data <= 20'h7A7D0;
        6'h0E: pcm_data <= 20'h7D8A5;
        6'h0F: pcm_data <= 20'h7F623;
        6'h10: pcm_data <= 20'h7FFFF;
        6'h11: pcm_data <= 20'h7F623;
        6'h12: pcm_data <= 20'h7D8A5;
        6'h13: pcm_data <= 20'h7A7D0;
        6'h14: pcm_data <= 20'h7641A;
        6'h15: pcm_data <= 20'h70E2C;
        6'h16: pcm_data <= 20'h6A6D9;
        6'h17: pcm_data <= 20'h62F20;
        6'h18: pcm_data <= 20'h5A827;
        6'h19: pcm_data <= 20'h5133C;
        6'h1A: pcm_data <= 20'h471CE;
        6'h1B: pcm_data <= 20'h3C56B;
        6'h1C: pcm_data <= 20'h30FBC;
        6'h1D: pcm_data <= 20'h25280;
        6'h1E: pcm_data <= 20'h18F8B;
        6'h1F: pcm_data <= 20'h0C8BD;
        6'h20: pcm_data <= 20'h00000;
        6'h21: pcm_data <= 20'hF3743;
        6'h22: pcm_data <= 20'hE7075;
        6'h23: pcm_data <= 20'hDAD80;
        6'h24: pcm_data <= 20'hCF044;
        6'h25: pcm_data <= 20'hC3A95;
        6'h26: pcm_data <= 20'hB8E32;
        6'h27: pcm_data <= 20'hAECC4;
        6'h28: pcm_data <= 20'hA57D9;
        6'h29: pcm_data <= 20'h9D0E0;
        6'h2A: pcm_data <= 20'h95927;
        6'h2B: pcm_data <= 20'h8F1D4;
        6'h2C: pcm_data <= 20'h89BE6;
        6'h2D: pcm_data <= 20'h85830;
        6'h2E: pcm_data <= 20'h8275B;
        6'h2F: pcm_data <= 20'h809DD;
        6'h30: pcm_data <= 20'h80000;
        6'h31: pcm_data <= 20'h809DD;
        6'h32: pcm_data <= 20'h8275B;
        6'h33: pcm_data <= 20'h85830;
        6'h34: pcm_data <= 20'h89BE6;
        6'h35: pcm_data <= 20'h8F1D4;
        6'h36: pcm_data <= 20'h95927;
        6'h37: pcm_data <= 20'h9D0E0;
        6'h38: pcm_data <= 20'hA57D9;
        6'h39: pcm_data <= 20'hAECC4;
        6'h3A: pcm_data <= 20'hB8E32;
        6'h3B: pcm_data <= 20'hC3A95;
        6'h3C: pcm_data <= 20'hCF044;
        6'h3D: pcm_data <= 20'hDAD80;
        6'h3E: pcm_data <= 20'hE7075;
        6'h3F: pcm_data <= 20'hF3743;
      endcase // case(index[5:0])
   end // always @ (index)
endmodule

///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
///////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
```

```
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
///////////////////////////////////////////////////////////////////////////////


module lab3   (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
            ac97_bit_clock,

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

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

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

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

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

            clock_feedback_out, clock_feedback_in,

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

            rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

            mouse_clock, mouse_data, keyboard_clock, keyboard_data,

            clock_27mhz, clock1, clock2,

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

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

            switch,

            led,

            user1, user2, user3, user4,

            daughtercard,

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

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

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

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

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

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

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

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

 input  clock_feedback_in;
 output clock_feedback_out;

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

 output rs232_txd, rs232_rts;
 input  rs232_rxd, rs232_cts;

 input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

 input  clock_27mhz, clock1, clock2;

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

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

```
   output [7:0] led;

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

   inout [43:0] daughtercard;

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

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

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

   ////////////////////////////////////////////////////////////////////////////
   //
   // Reset Generation
   //
   // A shift register primitive is used to generate an active-high reset
   // signal that remains high for 16 clock cycles after configuration finishes
   // and the FPGA's internal clocks begin toggling.
   //
   ////////////////////////////////////////////////////////////////////////////
   wire reset;
   wire power_on_reset;

   SRL16 reset_sr (.D(1'b0), .CLK(clock_27mhz), .Q(power_on_reset),    //////
MODIFIED to 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 db1(power_on_reset, clock_65mhz, ~button_enter, user_reset);
   assign reset = user_reset | power_on_reset;

   assign user_reset = ~button_enter;



   // Audio Input and Output
   wire beat_metro;
   assign beep = beat_metro;
   //lab3 assign audio_reset_b = 1'b0;
   //lab3 assign ac97_synch = 1'b0;
   //lab3 assign ac97_sdata_out = 1'b0;
   // ac97_sdata_in is an input



   // Video Output
```

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

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

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

// Flash ROM
assign flash_data = 16'hZ;
assign flash_address = 24'h0;
assign flash_ce_b = 1'b1;
assign flash_oe_b = 1'b1;
assign flash_we_b = 1'b1;
assign flash_reset_b = 1'b0;
assign flash_byte_b = 1'b1;
// flash_sts is an input

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

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

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

  // Buttons, Switches, and Individual LEDs
  //lab3 assign led = 8'hFF;
  // button0, button1, button2, button3, button_enter, button_right,
  // button_left, button_down, button_up, and switches are inputs

  // User I/Os
  assign user1 = 32'hZ;
  assign user2 = 32'hZ;
  assign user3 = 32'hZ;
  assign user4 = 32'hZ;

  // Daughtercard Connectors
  assign daughtercard = 44'hZ;

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

  // Logic Analyzer
  //lab3 assign analyzer1_data = 16'h0;
  //lab3 assign analyzer1_clock = 1'b1;
  //lab3 assign analyzer2_data = 16'h0;
  //lab3 assign analyzer2_clock = 1'b1;
  //assign analyzer3_data = 16'h0;
  assign analyzer3_clock = 1'b1;
  assign analyzer4_data = 16'h0;
  assign analyzer4_clock = 1'b1;

  wire [7:0] from_ac97_data, to_ac97_data, xk_re, xk_im;
  wire ready, ready_pulse, sharp, sharp_output, beat, new_note, new_note_test;
  wire[10:0] peak_index;
  wire[11:0] xk_index, xn_index;
  wire[4:0] note, note_output;
  wire[2:0] duration;
  wire[24:0] metro_dur;
  wire[7:0] absolute;

  // AC97 driver
  audio a(clock_27mhz, 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
   // light up LEDs when recording





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

/////////////////////////////////////////////////////////////////////////////////
/////////
   //////////   INSTANTIATIONS OUTPUTS  //////////////////////////

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




   wire playback, resetfft;
   debounce benter(reset, clock_27mhz, button_enter, playback);
   debounce debouncefft(reset, clock_27mhz, button0, resetfft);
   assign led = switch[1] ? ~{beat_metro, 4'b0, duration} : switch [0] ?
~{beat_metro, sharp, 1'b0, note} : ~{beat_metro, peak_index[6:0]};


       // Basic video Signals
   wire [10:0] hcount;
   wire [9:0]  vcount;
   wire hsync,vsync,blank;

      // output useful things to the logic analyzer connectors
   assign analyzer1_clock = ready_pulse;
   assign analyzer1_data = {xk_index[11:4], 8'b0};

   assign analyzer2_clock = clock_27mhz;
   assign analyzer2_data = {xk_re, 8'b0};

   assign analyzer3_data = {xk_re, 8'b0};





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

/////////////////////////////////////////////////////////////////////////////////
/////////
   //////////   INSTANTIATIONS NOTE RECOGNIZER
/////////////////////////////////////////

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

```
   //Turn ready into a pulse
   L2P L2Pone(clock_27mhz, ready, ready_pulse);

   // record module
   recorder r(clock_27mhz, reset, playback, ready, from_ac97_data, to_ac97_data);

   //FFT instantiation
   fft myfft(from_ac97_data, 8'b00000000, 1'b1, 5'b01100, 1'b0, 1'b1, 1'b0,
12'b0010_1010_1011 , 1'b0 , ready_pulse, clock_27mhz, xk_re, xk_im, xn_index,
xk_index, rfd, busy, dv, edone, done);

   //Peak Detector instantiation
   peakDetector pd1(clock_27mhz, from_ac97_data, ready_pulse, xk_index[10:0],
xk_re, xk_im, peak_index);

   //Look-Up Table instantiation
   lookuptable lut1(clock_27mhz, ready_pulse, peak_index, note, sharp);

   //Metronome instantiation
   Metronome metronome1(clock_27mhz, metro_dur, beat);

   //Rhythm instantiation
   Rhythm myrhythm(clock_27mhz, metro_dur, note, sharp, note_output,
sharp_output, duration, new_note);


   //Updowns for debugging pulses
   updown updown1(clock_27mhz, beat, beat_metro);

   updown updown2(clock_27mhz, new_note, new_note_test);

   abs myabs(clk, from_ac97_data, absolute);



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

////////////////////////////////////////////////////////////////////////////////
/////////
   //////////   INSTANTIATIONS VIDEO DISPLAY
//////////////////////////////////////////////

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

   //Produce synchronized 65mhz clock for video signal

      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));
   // Metronome beat generation

   metro_gen metro1(clock_27mhz,reset,switch[7:5],metro_dur);



   // UP and DOWN buttons for pong paddle
   wire up,down;
   debounce db2(reset, clock_65mhz, ~button_up, up);
   debounce db3(reset, clock_65mhz, ~button_down, down);

   // generate basic video signals

   xvga xvga1(clock_65mhz,hcount,vcount,hsync,vsync,blank,hreset);

   // Wire instantiations
   wire [6:0] upmargin;
   wire [6:0] line;
   wire [6:0] space;
   wire [6:0] bigspace;
   wire stavepixel;
   wire bwpixel;
   wire ramout;
   wire torampixel;
   wire [2:0] finalpixel;
   wire [19:0] addr;
   wire shsync,svsync,sblank;
   wire newnote;
   wire newclk;
   wire [10:0]  cx;
   wire [9:0]   cy;
   wire [10:0]  xclef;
   wire [9:0]   yclef;
   wire newclock;
   wire hres_clk;
   wire [7:0] font_byte;
   wire [63:0] data_hex;
   wire newpage;
   wire refresh;
   wire newpage_clk;
   wire [7:0] cstring;
   wire [3:0] clef_code;
   wire [2:0]  charpixel;
   wire [2:0] clefpixel;
   wire newpagedebug;
   wire [2:0] durationfoo;
   wire [2:0] durationdisp;
   wire newnotedisp;
   wire [4:0] notedisp;
   wire sharpdisp;





///////////////////////////////////////////////
////   VGA SIGNAL ASSIGNMENTS            /////////
```

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

   // The Buttons selects which video generator to use:
   //  00: transcriber's output
   //  01: 1 pixel outline of active video area (adjust screen controls)
   //  10: color bars
   reg [2:0] rgb;
   reg b,hs,vs;
   always @(posedge clock_65mhz) begin
      // transcriber output signals
       hs <= hsync;
       vs <= vsync;
       b <= blank;
       rgb <= finalpixel;

   end

   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clock_65mhz.
   assign vga_out_red = {8{rgb[2]}};
   assign vga_out_green = {8{rgb[1]}};
   assign vga_out_blue = {8{rgb[0]}};
   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_blank_b = ~b;
   assign vga_out_pixel_clock = ~clock_65mhz;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;


   // Initialize stave sprite

   stave stave1(clock_65mhz,reset,hcount,vcount,hsync,vsync,blank,stavepixel);


   // Assignments for connection between subsystems

      assign durationdisp = duration;

      assign newnotedisp = new_note;

      assign notedisp = note_output;

      assign sharpdisp = sharp_output;

   //initialize note displayer

   char_string_display
char1(clock_65mhz,hcount,vcount,charpixel,cstring,cx,cy,font_byte);

   //initialize clef displayer

   assign clef_code = 4'h1;

   clef_display clef1(clock_65mhz,hcount,vcount,clefpixel,clef_code,xclef,yclef);

   //initialize font generation logic
   fontgen
font(clock_27mhz,reset,newpage,newnotedisp,notedisp,durationdisp,sharpdisp,cstrin
g);

   //initialize hcount/vcount to address converter

   countaddr2 converter(hcount,vcount,addr);


   // Initialize Video Ram (1024*768 locations, 1-bit wide)

   videoram videoram(addr,clock_65mhz,nepixel,ramout,towepixel);

   // Initialize Control Logic

   control
ctrl(clock_27mhz,reset,notedisp,newnotedisp,durationdisp,cx,cy,xclef,yclef,newpag
e);

   // Assign total black&white 1-bit pixel output by OR of output of all sprites

   assign bwpixel = stavepixel | charpixel | clefpixel;

   assign torampixel = bwpixel;

   // Pixel to RAM for non-erasable output, erased only on reset

   assign nepixel = reset ? 1'b0 : 1'b1;

   // Assign the pixel output to control WE together with RAM output for non-
erasable video

   assign towepixel = (reset) ? 1'b1 : (ramout ? 1'b0 : bwpixel);

   // Convert black&white pixel signal to color

   assign finalpixel = ramout ? 3'b000 : 3'b111;



      //Hex display

   display_16hex hex(reset, clock_27mhz, data_hex,
          disp_blank, disp_clock, disp_rs, disp_ce_b,
          disp_reset_b, disp_data_out);


   assign data_hex = {metro_dur[24:22],note_output,cstring};


endmodule

/////////////////////////////////////////////////////////////////////////////////
///////////
//////////////////// END OF LABKIT MODULE
/////////////////////////////////////////////////
```

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




////////////////////////////////////////////////////////////////////////////////
/////////
   //////////   MODULES NOTE RECOGNIZER
///////////////////////////////////////////////////

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


///////////////////////////////////////////////////////////////////////////////
//
// Record/playback
//
///////////////////////////////////////////////////////////////////////////////


//This module simple converts a signal into a pulse. This is needed for signals
like new_note
module L2P(clk, in, out);
input clk, in;
output out;
reg old, out;
always @ (posedge clk) begin
old <= in;
out <= in & ~old;
end

endmodule


//PEAK DETECTOR

//Detects and outputs the peak frequency of a discrete time frequency spectrum

module peakDetector(clk, from_ac97_data, ready_pulse, xk_index, xk_re, xk_im,
peak_index);
input clk, ready_pulse;
input[7:0] from_ac97_data, absolute;
input[10:0] xk_index;
input signed [7:0] xk_re, xk_im;
reg [16:0] magnitude;
reg [16:0] max_value = 0;
reg [10:0] max_index;
reg[10:0] peak;
output[10:0] peak_index;

assign peak_index = peak;

always @ (posedge clk) magnitude <= (xk_re * xk_re) + (xk_im * xk_im);  //sum of
square of the real and imaginary parts.
```

```
always @ (posedge ready_pulse) begin   //FFT works on a clock enable controlled
by ready_pulse. So its outputs arrive at the same rate.


if (xk_index > 77) begin    //when we reach index 77, we have gone through all
the appropriate frequencies.
     peak <= max_index;         //set peak to be the max_index
     max_value <= 0;       //reset running max for next FFT frame.
     end

else if (magnitude > max_value && magnitude > 100 && xk_index < 77 )
     begin

     max_value <= magnitude;          //if the magnitude at that frequency
(index) is larger than running max,
                                    //above a certain threshold and of "legal"
frequency, make it the new max
     max_index <= xk_index;           // and store new index associated with the
new max.

     end

end




endmodule




//LOOK-UP TABLE

module lookuptable(clk, ready_pulse, peak_index, note, sharp);
input clk, ready_pulse;
input[10:0] peak_index;
output[4:0] note;
output sharp;
reg sharp_reg;
reg[4:0] note_reg;
wire[10:0] peak_freq;
assign note = note_reg;
assign sharp = sharp_reg;

assign peak_freq = 48000 * peak_index / 4096;  //Convert to continuous time
frequency in Hertz.

always @ (posedge ready_pulse)              //Convert the frequency in Hertz to a
musical note.

if (peak_freq > 285 && peak_freq < 300)
begin
note_reg <= 1;
sharp_reg <= 0;
end
```

```
else if (peak_freq > 300 && peak_freq < 320)
begin
note_reg <= 1;
sharp_reg <= 1;
end

else if (peak_freq > 320 && peak_freq < 340)
begin
note_reg <= 2;
sharp_reg <= 0;
end

else if (peak_freq > 340 && peak_freq < 360)
begin
note_reg <= 3;
sharp_reg <= 0;
end

else if (peak_freq > 360 && peak_freq < 380)
begin
note_reg <= 3;
sharp_reg <= 1;
end

else if (peak_freq > 380 && peak_freq < 405)
begin
note_reg <= 4;
sharp_reg <= 0;
end

else if (peak_freq > 405 && peak_freq < 425)
begin
note_reg <= 4;
sharp_reg <= 1;
end

else if (peak_freq > 425 && peak_freq < 452)
begin
note_reg <= 5;
sharp_reg <= 0;
end

else if (peak_freq > 452 && peak_freq < 480)
begin
note_reg <= 5;
sharp_reg <= 1;
end


else if (peak_freq > 480 && peak_freq < 510)
begin
note_reg <= 6;
sharp_reg <= 0;
end


else if (peak_freq > 510 && peak_freq < 533)
begin
```

```
note_reg <= 7;
sharp_reg <= 0;
end

else if (peak_freq > 533 && peak_freq < 570)
begin
note_reg <= 7;
sharp_reg <= 1;
end

else if (peak_freq > 570 && peak_freq < 605)
begin
note_reg <= 8;
sharp_reg <= 0;
end

else if (peak_freq > 605 && peak_freq < 640)
begin
note_reg <= 8;
sharp_reg <= 1;
end

else if (peak_freq > 640 && peak_freq < 685)
begin
note_reg <= 9;
sharp_reg <= 0;
end

else if (peak_freq > 685 && peak_freq < 720)
begin
note_reg <= 10;
sharp_reg <= 0;
end

else if (peak_freq > 720 && peak_freq < 765)
begin
note_reg <= 10;
sharp_reg <= 1;
end

else if (peak_freq > 765 && peak_freq < 810)
begin
note_reg <= 11;
sharp_reg <= 0;
end

else if (peak_freq > 810 && peak_freq < 860)
begin
note_reg <= 11;
sharp_reg <= 1;
end


else
begin
note_reg <= 0;
sharp_reg <= 0;
end
```

```
endmodule



//METRONOME

//Works like a clock DIVIDER. It counts up to a user defined parameter
"metro_dur"
//and pulses a beat when the counter reaches that value. The counter is
incremented every clock cycle.



module Metronome(clk, metro_dur, beat);

input clk;
input[24:0] metro_dur;
output beat;
reg beat = 0;
reg[24:0] counter = 0;

 always @(posedge clk)



      if (counter == metro_dur)   //when we reach parameter, pulse the output and
reset counter
         begin
         beat <= 1;
         counter <= 0;
         end

     else
     begin

     counter <= counter + 1;      //else increment
     beat <= 0;

     end


endmodule




//UPDOWN

//Purely used for testing and debugging to make pulse signals observable on LEDs
of labkit.

module updown(clk, sigin, sigout);
input clk, sigin;
output sigout;
reg sigout;
  always @ (posedge clk) begin
        if (sigin) sigout <= !sigout;
        else sigout <= sigout;
  end
endmodule


//RHTYHM

module Rhythm(clk, metro_dur, note, sharp, note_output, sharp_output, duration,
new_note);

input clk, sharp;
input[24:0] metro_dur;
input[4:0] note;
output sharp_output, new_note;
output[2:0] duration;
output[4:0] note_output;
reg[2:0] duration = 0;
reg new_note = 0;
reg[4:0] note_output = 0, old_note = 0;
reg sharp_output = 0, old_sharp = 0;

reg[27:0] counter = 0;


always @ (posedge clk)

if (note != old_note || sharp != old_sharp)  //When the value of EITHER note or
sharp (inputs) changes
     begin                               //new-note goes high and duration is
determined by the value
                                              //up to which the counter has
reached relative to a beat.
     if(counter < (metro_dur / 2))             //Also, the counter is reset and
the note and sharp are stored.
     begin
     counter <= 0;
     new_note <= 0;
     old_note <= note;
     old_sharp <= sharp;
     end

     else if(counter < (3 * metro_dur / 2))
     begin
     counter <= 0;
     new_note <= 1;
     note_output <= old_note;
     old_note <= note;
     duration <= 1;
     sharp_output <= old_sharp;
     old_sharp <= sharp;
     end

     else if(counter < (5 * metro_dur / 2))

     begin
     counter <= 0;
     new_note <= 1;
     note_output <= old_note;
```

```
      old_note <= note;
      duration <= 2;
      sharp_output <= old_sharp;
      old_sharp <= sharp;
      end

      else if(counter < (7 * metro_dur / 2))

      begin
      counter <= 0;
      new_note <= 1;
      note_output <= old_note;
      old_note <= note;
      duration <= 3;
      sharp_output <= old_sharp;
      old_sharp <= sharp;
      end


      else
      begin
      counter <= 0;
      new_note <= 1;
      note_output <= old_note;
      old_note <= note;
      duration <= 4;
      sharp_output <= old_sharp;
      old_sharp <= sharp;
      end


end

else if(counter >= (4 * metro_dur))   //If the counter reaches 4 beats without a
change in note input
begin                                 //we have a whole note. So new_note goes
high, duration is 4 (beats)
counter <= 0;                            //and the counter is reset and the note
and sharp registers are updated.
new_note <= 1;
note_output <= old_note;
old_note <= note;
duration <= 4;
sharp_output <= old_sharp;
old_sharp <= sharp;
end


else                              //else increment the counter and reset new_note
to be LOW.
begin
counter <= counter + 1;
new_note <= 0;
end

endmodule
```

```
//RECORDER (from lab 3)

//Useful for testing and debugging

module recorder(clock_27mhz, reset, playback, ready, from_ac97_data,
to_ac97_data);
   input clock_27mhz;              // 27mhz system clock
   input reset;                    // 1 to reset to initial state
   input playback;                 // 1 for playback, 0 for record
   input ready;                    // 1 when AC97 data is available
   input [7:0] from_ac97_data;     // 8-bit PCM data from mic
   output [7:0] to_ac97_data;      // 8-bit PCM data to headphone

   // detect clock cycle when READY goes 0 -> 1
   // f(READY) = 48khz
   wire new_frame;
   reg old_ready;
   always @ (posedge clock_27mhz) old_ready <= reset ? 0 : ready;
   assign new_frame = ready & ~old_ready;

   // test: playback 750hz tone, or loopback using incoming data
   wire [19:0] tone;
   tone750hz xxx(clock_27mhz, ready, tone);
   reg [7:0] to_ac97_data;
   always @ (posedge clock_27mhz) begin
      if (new_frame) begin
       // just received new data from the AC97
       to_ac97_data <= from_ac97_data;
      end
   end

endmodule




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

////////////////////////////////////////////////////////////////////////////////
/////////
   //////////   MODULES VIDEO DISPLAY
///////////////////////////////////////////////////

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




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

```
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)  {By Nathan Ickes)
//
////////////////////////////////////////////////////////////////////////////////

module xvga(vclock,hcount,vcount,hsync,vsync,blank,hreset);
   input vclock;
   output [10:0] hcount;
   output [9:0] vcount;
   output  vsync;
   output  hsync;
   output  blank;
   output      hreset;

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

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   wire      hsyncon,hsyncoff,hreset,hblankon;
   assign    hblankon = (hcount == 1023);
   assign    hsyncon = (hcount == 1047);
   assign    hsyncoff = (hcount == 1183);
   assign    hreset = (hcount == 1343);

   // vertical: 806 lines total
   // 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
   wire      next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always @(posedge vclock) begin
      hcount <= hreset ? 0 : hcount + 1;
      hblank <= next_hblank;
      hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

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

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



////////////////////////////////////////////////////////////////////////////////
//
// Music stave sprite
//
////////////////////////////////////////////////////////////////////////////////
```

```
module stave (vclock,reset,hcount,vcount,hsync,vsync,blank,pixel);

   input vclock;// 65MHz clock
   input reset;       // 1 to initialize module
   input [10:0] hcount;    // horizontal index of current pixel (0..1023)
   input [9:0] vcount;   // vertical index of current pixel (0..767)
   input hsync;       // XVGA horizontal sync signal (active low)
   input vsync;       // XVGA vertical sync signal (active low)
   input blank;       // XVGA blanking (1 means output black pixel)


   parameter upmargin = 55;
   parameter line = 1;
   parameter space = 7;
   parameter bigspace = 62;


   output pixel;// stave sprite pixel

   reg stpixel;

   always @(hcount or vcount) begin

      if (((vcount >= upmargin && vcount < (upmargin + line)) |
          (vcount >= (upmargin + 1*(line + space))   &&   vcount < (upmargin +
line + 1*(line + space)) ) |
          (vcount >= (upmargin + 2*(line + space))   &&   vcount < (upmargin +
line + 2*(line + space)) ) |
          (vcount >= (upmargin + 3*(line + space))   &&   vcount < (upmargin +
line + 3*(line + space)) ) |
          (vcount >= (upmargin + 4*(line + space))   &&   vcount < (upmargin +
line + 4*(line + space)) ) )  |

         ((vcount >= (upmargin + 1*bigspace) && vcount < (upmargin + 1*bigspace +
line)) |
          (vcount >= (upmargin + 1*bigspace + 1*(line + space))   &&   vcount <
(upmargin + 1*bigspace + line + 1*(line + space)) ) |
          (vcount >= (upmargin + 1*bigspace + 2*(line + space))   &&   vcount <
(upmargin + 1*bigspace + line + 2*(line + space)) ) |
          (vcount >= (upmargin + 1*bigspace + 3*(line + space))   &&   vcount <
(upmargin + 1*bigspace + line + 3*(line + space)) ) |
          (vcount >= (upmargin + 1*bigspace + 4*(line + space))   &&   vcount <
(upmargin + 1*bigspace + line + 4*(line + space)) ) ) |

         ((vcount >= (upmargin + 2*bigspace) && vcount < (upmargin + 2*bigspace +
line)) |
          (vcount >= (upmargin + 2*bigspace + 1*(line + space))   &&   vcount <
(upmargin + 2*bigspace + line + 1*(line + space)) ) |
          (vcount >= (upmargin + 2*bigspace + 2*(line + space))   &&   vcount <
(upmargin + 2*bigspace + line + 2*(line + space)) ) |
          (vcount >= (upmargin + 2*bigspace + 3*(line + space))   &&   vcount <
(upmargin + 2*bigspace + line + 3*(line + space)) ) |
          (vcount >= (upmargin + 2*bigspace + 4*(line + space))   &&   vcount <
(upmargin + 2*bigspace + line + 4*(line + space)) ) ) |

         ((vcount >= (upmargin + 3*bigspace) && vcount < (upmargin + 3*bigspace +
line)) |
          (vcount >= (upmargin + 3*bigspace + 1*(line + space))   &&   vcount <
(upmargin + 3*bigspace + line + 1*(line + space)) ) |
          (vcount >= (upmargin + 3*bigspace + 2*(line + space))   &&   vcount <
(upmargin + 3*bigspace + line + 2*(line + space)) ) |
          (vcount >= (upmargin + 3*bigspace + 3*(line + space))   &&   vcount <
(upmargin + 3*bigspace + line + 3*(line + space)) ) |
          (vcount >= (upmargin + 3*bigspace + 4*(line + space))   &&   vcount <
(upmargin + 3*bigspace + line + 4*(line + space)) ) ) |

         ((vcount >= (upmargin + 4*bigspace) && vcount < (upmargin + 4*bigspace +
line)) |
          (vcount >= (upmargin + 4*bigspace + 1*(line + space))   &&   vcount <
(upmargin + 4*bigspace + line + 1*(line + space)) ) |
          (vcount >= (upmargin + 4*bigspace + 2*(line + space))   &&   vcount <
(upmargin + 4*bigspace + line + 2*(line + space)) ) |
          (vcount >= (upmargin + 4*bigspace + 3*(line + space))   &&   vcount <
(upmargin + 4*bigspace + line + 3*(line + space)) ) |
          (vcount >= (upmargin + 4*bigspace + 4*(line + space))   &&   vcount <
(upmargin + 4*bigspace + line + 4*(line + space)) ) ) |

         ((vcount >= (upmargin + 5*bigspace) && vcount < (upmargin + 5*bigspace +
line)) |
          (vcount >= (upmargin + 5*bigspace + 1*(line + space))   &&   vcount <
(upmargin + 5*bigspace + line + 1*(line + space)) ) |
          (vcount >= (upmargin + 5*bigspace + 2*(line + space))   &&   vcount <
(upmargin + 5*bigspace + line + 2*(line + space)) ) |
          (vcount >= (upmargin + 5*bigspace + 3*(line + space))   &&   vcount <
(upmargin + 5*bigspace + line + 3*(line + space)) ) |
          (vcount >= (upmargin + 5*bigspace + 4*(line + space))   &&   vcount <
(upmargin + 5*bigspace + line + 4*(line + space)) ) ) |

         ((vcount >= (upmargin + 6*bigspace) && vcount < (upmargin + 6*bigspace +
line)) |
          (vcount >= (upmargin + 6*bigspace + 1*(line + space))   &&   vcount <
(upmargin + 6*bigspace + line + 1*(line + space)) ) |
          (vcount >= (upmargin + 6*bigspace + 2*(line + space))   &&   vcount <
(upmargin + 6*bigspace + line + 2*(line + space)) ) |
          (vcount >= (upmargin + 6*bigspace + 3*(line + space))   &&   vcount <
(upmargin + 6*bigspace + line + 3*(line + space)) ) |
          (vcount >= (upmargin + 6*bigspace + 4*(line + space))   &&   vcount <
(upmargin + 6*bigspace + line + 4*(line + space)) ) ) |

         ((vcount >= (upmargin + 7*bigspace) && vcount < (upmargin + 7*bigspace +
line)) |
          (vcount >= (upmargin + 7*bigspace + 1*(line + space))   &&   vcount <
(upmargin + 7*bigspace + line + 1*(line + space)) ) |
          (vcount >= (upmargin + 7*bigspace + 2*(line + space))   &&   vcount <
(upmargin + 7*bigspace + line + 2*(line + space)) ) |
          (vcount >= (upmargin + 7*bigspace + 3*(line + space))   &&   vcount <
(upmargin + 7*bigspace + line + 3*(line + space)) ) |
          (vcount >= (upmargin + 7*bigspace + 4*(line + space))   &&   vcount <
(upmargin + 7*bigspace + line + 4*(line + space)) ) ) |

         ((vcount >= (upmargin + 8*bigspace) && vcount < (upmargin + 8*bigspace +
line)) |
          (vcount >= (upmargin + 8*bigspace + 1*(line + space))   &&   vcount <
(upmargin + 8*bigspace + line + 1*(line + space)) ) |
          (vcount >= (upmargin + 8*bigspace + 2*(line + space))   &&   vcount <
(upmargin + 8*bigspace + line + 2*(line + space)) ) |
          (vcount >= (upmargin + 8*bigspace + 3*(line + space))   &&   vcount <
(upmargin + 8*bigspace + line + 3*(line + space)) ) |
          (vcount >= (upmargin + 8*bigspace + 4*(line + space))   &&   vcount <
(upmargin + 8*bigspace + line + 4*(line + space)) ) ) |

         ((vcount >= (upmargin + 9*bigspace) && vcount < (upmargin + 9*bigspace +
line)) |
          (vcount >= (upmargin + 9*bigspace + 1*(line + space))   &&   vcount <
(upmargin + 9*bigspace + line + 1*(line + space)) ) |
          (vcount >= (upmargin + 9*bigspace + 2*(line + space))   &&   vcount <
(upmargin + 9*bigspace + line + 2*(line + space)) ) |
          (vcount >= (upmargin + 9*bigspace + 3*(line + space))   &&   vcount <
(upmargin + 9*bigspace + line + 3*(line + space)) ) |
          (vcount >= (upmargin + 9*bigspace + 4*(line + space))   &&   vcount <
(upmargin + 9*bigspace + line + 4*(line + space)) ) ) |

         ((vcount >= (upmargin + 10*bigspace) && vcount < (upmargin + 10*bigspace
+ line)) |
          (vcount >= (upmargin + 10*bigspace + 1*(line + space))   &&   vcount <
(upmargin + 10*bigspace + line + 1*(line + space)) ) |
          (vcount >= (upmargin + 10*bigspace + 2*(line + space))   &&   vcount <
(upmargin + 10*bigspace + line + 2*(line + space)) ) |
          (vcount >= (upmargin + 10*bigspace + 3*(line + space))   &&   vcount <
(upmargin + 10*bigspace + line + 3*(line + space)) ) |
          (vcount >= (upmargin + 10*bigspace + 4*(line + space))   &&   vcount <
(upmargin + 10*bigspace + line + 4*(line + space)) ) ) )


       begin
        stpixel <= 1'b1;
       end else begin
        stpixel <= 1'b0;
       end
    end


   assign pixel = stpixel;

   endmodule




//Module for converting hcount/vcount to a valid RAM address, so that by
increasing the address the RAM outputs subsequent pixels
module countaddr2 (hcount,vcount,addr);
 input [10:0] hcount;
 input [9:0] vcount;
 output [19:0] addr;
 reg [19:0] regaddr;

 always @(hcount or vcount) begin
  if (hcount[10] == 1) begin
     regaddr <= 0; end
  else begin
     regaddr <= {vcount, hcount[9:0]};      end
 end

 assign addr = regaddr;

endmodule




// Control Module
module control(clk,reset,note,newnote,duration,cx,cy,xclef,yclef,newpage);

parameter tone = 4;
parameter hdist = 30;
parameter stave_height = 62;
parameter cxinit = 35;
parameter cyinit = 80;
parameter xclefinit = 10;
parameter yclefinit = 53;

input clk;
input reset;
input newnote;
input [4:0] note;
input [2:0] duration;
output [10:0] cx;
output [9:0] cy;
output [10:0] xclef;
output [9:0] yclef;
output newpage;

reg [10:0] cxcount;
reg [10:0] cxfixed;
reg [9:0] cycount;
reg [9:0] cyfixed;
reg [10:0] xclef;
reg [9:0] yclef;
reg [2:0] old_dur;
reg newpage;



always @ (posedge clk)
 begin
   if (reset) begin          //if reset, reset controller
      cxcount <= cxinit;
       cxfixed <= cxinit;
      cycount <= cyinit;
       cyfixed <= cyinit;
       xclef <= xclefinit;
       yclef <= yclefinit;
      end

 else begin
       if (newnote)
        begin
            old_dur <= duration;
            if ((cyfixed >= 700) && (cxcount>= 900)) begin          //if page is
finished, stall until reset
               cxcount <= 1030;
               cycount <= 750;
                cyfixed <= 750;
                newpage <= 1;
               end

            else begin

            yclef <= (cxcount >= 900) ? (cyfixed + stave_height - 27) : (cyfixed
- 27);

            cyfixed <= (cxcount >= 900) ? (cyfixed + stave_height) : (cyfixed);
//cyfixed increase if line is finished

            cxcount <= (cxcount >= 900) ? (cxinit) : (cxcount + (hdist *
old_dur));   //cx reset or go on

            cycount <= (cxcount >= 900) ?
                           (note ? (cyfixed + stave_height - (note*tone)) :
(cyfixed + stave_height - 20)) : //compensate for delay on cyfixed
                            (note ? (cyfixed - (note*tone)) : (cyfixed - 20));
//cy adjusts height according to note value

                end
        end
       newpage <= 0;
       end
  end


assign cy = cycount;

assign cx = cxcount;


endmodule


//Divider, produces 1Hz pulse which simulates newnote for debugging

module mydivider(clk, reset_sync, hzenable);
    input clk, reset_sync;
    output hzenable;
    reg [27:0] count;
    parameter DIVIDER_GO = 30000000;


    wire clk, reset_sync, hzenable;

    always @ (posedge clk)
    begin
      if (reset_sync) begin
        count <= 0;
         end else if (count == DIVIDER_GO) begin
         count <= 0;
```

```
           end else begin
           count <= count+1;
       end

     end
     assign hzenable = (count == DIVIDER_GO);
endmodule


// Updown: convert pulse to clock (debugging)

module updownroby(clk, reset_sync, sigin, sigout);
input clk, reset_sync, sigin;
output sigout;
reg sigout;
  always @ (posedge clk) begin
      if (reset_sync) begin
      sigout <= 0;
      end else if (sigin) begin
      sigout <= !sigout;
      end else begin
      sigout <= sigout;
      end
  end
endmodule


//Module for generating the appropriate fonts


module fontgen(clk,reset,newpage,newnote,note,duration,sharp,cstring);
input clk,sharp,reset,newpage,newnote;
input [4:0] note;
input [2:0] duration;
output [7:0] cstring;
reg [7:0] cstring;

parameter FULL = 4;
parameter DOT = 3;
parameter HALF = 2;
parameter QUARTER = 1;


always @ (posedge clk) begin

if (reset | newpage) begin
   cstring <= 8'h99;    end             //no marking at reset

else begin if (newnote) begin

    case (duration)

     FULL:
      begin
        if (note == 0) begin
           cstring <= 8'h59; end   //rest
        else begin
           if (sharp)
```

```
              cstring <= 8'h10;        // sharp
          else  cstring <= 8'h19; end  // natural
      end

      DOT:
      begin
        if (note == 0) begin
          cstring <= 8'h78; end   //rest
        else begin
          if (sharp)
              cstring <= 8'h20;        // sharp
          else  cstring <= 8'h29; end  // natural
      end

      HALF:
      begin
        if (note == 0) begin
          cstring <= 8'h79; end   //rest
        else begin
          if (sharp)
              cstring <= 8'h30;        // sharp
          else  cstring <= 8'h39; end  // natural
      end

      QUARTER:
      begin
        if (note == 0) begin
          cstring <= 8'h89; end   //rest
        else begin
          if (sharp)
              cstring <= 8'h40;        // sharp
          else  cstring <= 8'h49; end  // natural
      end

      default:
      begin
       cstring <= 8'hAA; // error: duration is not valid
      end

    endcase end end
end

endmodule



//Note displaying module
//Needs ROM for fonts. ROM must be D=192 W=8, representing 16 fonts of dimensions
8x12.
//ROM Initialized with COE file. Creates with Xilinx Core Generator.
// Originally developed by Chris Terman

module char_string_display (vclock,hcount,vcount,pixel,cstring,cx,cy,font_byte);

   parameter NCHAR = 2;    // number of 8-bit characters in cstring
   parameter NCHAR_BITS = 1; // number of bits in NCHAR

   input vclock;// 65MHz clock
   input [10:0] hcount;    // horizontal index of current pixel (0..1023)
   input [9:0]  vcount; // vertical index of current pixel (0..767)
   output [2:0] pixel;     // char display's pixel
   output [7:0] font_byte;
   input [NCHAR*4-1:0] cstring;  // character string to display    !!4-bit address
   input [10:0] cx;
   input [9:0]  cy;

   // 1 line x 8 character display (8 x 12 pixel-sized characters)

   wire [10:0]  hoff = hcount-1-cx;
   wire [9:0]   voff = vcount-cy;
   wire [NCHAR_BITS-1:0] column = NCHAR-1-hoff[NCHAR_BITS-1+4:4];  // < NCHAR, in
2char, col takes value 1 or 0
   wire [2:0]   h = hoff[3:1];            // 0 .. 7
   wire [3:0]   v = voff[4:1];          // 0 .. 11

   // look up character to display (from character string)
   reg [7:0]  char;
   integer  n;
   always @(*)
     for (n=0 ; n<4 ; n = n+1 )       // 8 bits per character (ASCII)  !!4-bit
address
       char[n] <= cstring[column*4+n];             // 8 to 4 bits per character

   // look up raster row from font rom
   wire reverse = char[4];                        //reverse = char 6 --> 3
   wire [7:0] font_addr = char[4:0]*12 + v;    // 12 bytes per character
   wire [7:0]  font_byte;
   font_rom f(font_addr,vclock,font_byte);

   // generate character pixel if we're in the right h,v area
   wire [2:0] cpixel = (font_byte[7 - h]) ? 3'b111 : 3'b000;
   wire dispflag = ((hcount > cx) & (vcount >= cy) & (hcount <= cx+NCHAR*16)
            & (vcount < cy + 24));
   wire [2:0] pixel = dispflag ? cpixel : 0;

endmodule




//Clef displaying module
//Needs ROM for fonts. ROM must be D=384 W=16, representing 16 fonts of
dimensions 16x24.
//ROM Initialized with COE file. Creates with Xilinx Core Generator
// Originally developed by Chris Terman

module clef_display (vclock,hcount,vcount,pixel,cstring,cx,cy);

   parameter NCHAR = 1;    // number of 8-bit characters in cstring
   parameter NCHAR_BITS = 1; // number of bits in NCHAR

   input vclock;// 65MHz clock
   input [10:0] hcount;    // horizontal index of current pixel (0..1023)
   input [9:0]  vcount; // vertical index of current pixel (0..767)
   output [2:0] pixel;     // char display's pixel
   input [NCHAR*4-1:0] cstring;  // character string to display    !!4-bit address
   input [10:0] cx;
   input [9:0]   cy;



   // 1 line x 8 character display (8 x 12 pixel-sized characters)

   wire [10:0]   hoff = hcount-1-cx;
   wire [9:0]    voff = vcount-cy;
   wire [NCHAR_BITS-1:0] column = NCHAR-1-hoff[NCHAR_BITS-1+4:4];  // < NCHAR, in
2char, col takes value 1 or 0
   wire [3:0]    h = hoff[4:1];             // 0 .. 15
   wire [4:0]    v = voff[5:1];           // 0 .. 22

   // look up character to display (from character string)
   reg [7:0]  char;
   integer  n;
   always @(*)
     for (n=0 ; n<4 ; n = n+1 )        // 8 bits per character (ASCII)  !!4-bit
address
       char[n] <= cstring[column*4+n];             // 4 bits per character index

   // look up raster row from font rom
   wire [7:0] font_addr = char[1:0]*24 + v;    // 12 bytes per character
   wire [15:0]  font_byte;                    // 16-bit lines
   font_rom_clef f_clef(font_addr,vclock,font_byte);

   // generate character pixel if we're in the right h,v area
   wire [2:0] cpixel = (font_byte[15 - h]) ? 3'b111 : 3'b000;
   wire dispflag = ((hcount > cx) & (vcount >= cy) & (hcount <= cx+NCHAR*32)
            & (vcount < cy + 48));
   wire [2:0] pixel = dispflag ? cpixel : 0;

endmodule


// Metronome tempo generator. Assigns pre-defined tempos to the system.
Controlled by swtiches.

module metro_gen(clk,reset,switch,metro_dur);
 input clk, reset;
 input [2:0] switch;
 output [24:0] metro_dur;
 reg  [24:0] metro_dur1 = 10000000;


 always @ (posedge clk) begin

   if (reset) begin
      assign metro_dur1 = 10000000; end
      else begin

       case (switch [2:0])
   0: assign metro_dur1 = 9000000;
   1: assign metro_dur1 = 13000000;
   2: assign metro_dur1 = 15000000;
   3: assign metro_dur1 = 17000000;
   4: assign metro_dur1 = 22000000;
   5: assign metro_dur1 = 25000000;
   6: assign metro_dur1 = 27000000;
   7: assign metro_dur1 = 33500000;
     endcase
   end
  end

assign metro_dur = metro_dur1;


endmodule
```