PerfectPitch

Grace Cheung Karl Rieb

Massachusetts Institute of Technology
Department of Electrical Engineering

December 15, 2007

Abstract

Fourier transforms and frequency domain analysis are being used to generate music scores from an audio sample. Frequency domain analysis is performed on the result of applying a fast fourier transform (FFT) to the audio input to find peaks corresponding to the pitch in the audio sample. The frequency peaks are matched to their corresponding music notes, which are used to generate a music score in real time.

Contents

1 Overview **1**

2 Module Specification **2**
2.1 Audio Input (Grace Cheung) 2
2.1.1 Headphone Out to Line In (HOtLI) 3
2.1.2 Amplifier 3
2.1.3 Analog-to-Digital Converter (ADC) 3
2.1.4 AC97 Codec 3
2.2 Signal Processing (Karl Rieb) 4
2.2.1 Fast Fourier Transform (FFT) 4
2.2.2 Magnitude 5
2.2.3 Peak Detector 6
2.2.4 Peak Filter 6
2.2.5 Note Look-Up 7
2.2.6 Metronome 7
2.2.7 Duration 8
2.3 Video Display (Grace Cheung) 8
2.3.1 Note Position 8
2.3.2 Note Sprite 9
2.3.3 Chord Flag Display 10
2.3.4 Rest Display 10
2.3.5 Staves Sprite 10
2.3.6 XVGA 11
2.3.7 Frame Buffer 11

3 Testing **12**
3.1 Signal Processing (Karl Rieb) 12
3.1.1 FFT 12
3.1.2 Magnitude, Peak Detector, Note Look-Up 13
3.1.3 Metronome 13
3.1.4 Duration 13
3.2 Video Display (Grace Cheung) 14
3.2.1 Sprites 14
3.2.2 Note Positions 14
3.2.3 Frame Buffer 14

4 Conclusion **17**

A Appendix: Magnitude **19**

B Appendix: Square Root **21**

C Appendix: Peak Detector **22**

D Appendix: Note Look-Up **25**

E Appendix: Duration **33**

F Appendix: Video Display **39**

List of Figures

1 Top Level View of System Modules 2
2 Audio Input Modules 2
3 Signal Processing Modules 4
4 Video Display Modules 9

List of Tables

1 Note Frequency Correlation [1] 16

1 Overview

The goal of *PerfectPitch* is to allow users the ability to create sheet music for their favorite songs. The system takes in the headphone output of any music player device and converts it into analog line input that is fed into an audio codec. The audio codec amplifies the signal and converts it to digital. Once the audio sample is in digital format, it is used by the fast fourier transform (FFT) module for analysis in the frequency domain. Multiple peaks in the signal are then detected and a filtered set of the peaks are sent to a note look-up table which matches the peak with its corresponding musical note.

The notes are sent as a chord to a timing module that keeps a tempo for the song and calculates the duration of the chord in the audio sample. After the chord and its duration are determined, this information is sent to the video modules.

The video modules display staves and chords as they are processed by the system. The chords are grouped by clef and displayed on the screen appropriately. The shape of the note is determined by a look-up table that contains a set of sprites associated with different note durations. The duration of the chord is indicated by the note heads (full or empty), stems, flags, and appereance of a dot (indicating one and a half duration).

The music score refreshes itself based on user input or when the music reaches the end of a page. Only one page is ever stored in memory while the system is active.

2 Module Specification

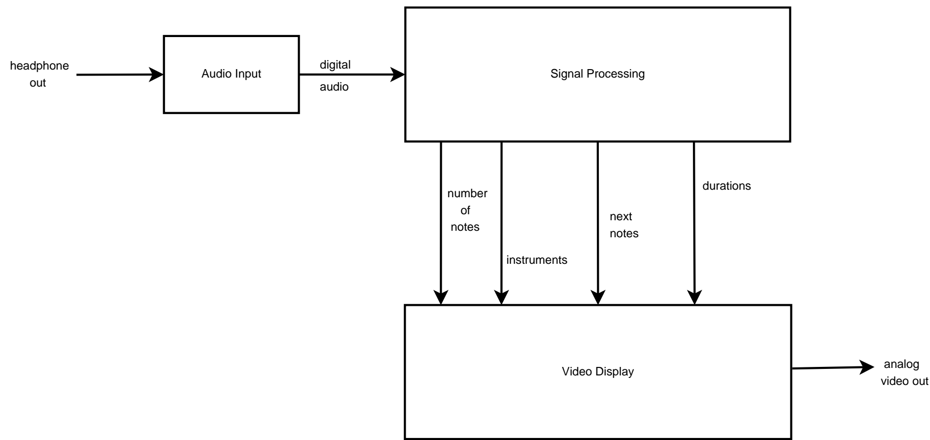

Figure 1: Top Level View of System Modules

The *PerfectPitch* system is comprised of three main components: audio input, signal processing, and video display (see **Figure 1**). Audio input is handled by an external analog component and the AC97 audio codec that converts the analog signal into a useable digital signal. The digital signal is then passed to the signal processing component to be analzed. The pitches of the music notes played in the audio sample are determined, along with their durations. This information is sent to the video display when ready to be drawn on the screen in the form of sheet music.

2.1 Audio Input (Grace Cheung)

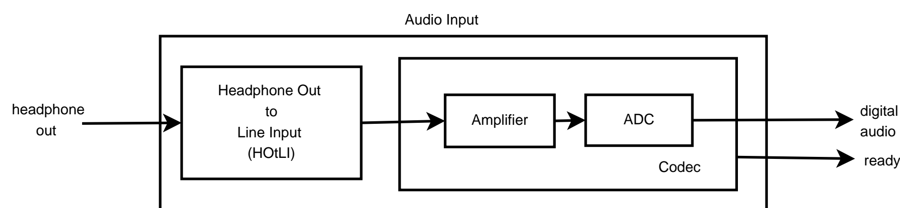

Figure 2: Audio Input Modules

The audio input module takes the analog output from a headphone jack and

converts it to a digital audio signal that's used by the rest of the system. See **Figure 2**.

2.1.1 Headphone Out to Line In (HOtLI)

The analog signal from a headphone output on most music playing devices is not compatible with the signal required for the line in input. The HOtLI module transforms the signal appropriately before it is passed to the audio codec. The transformation is done completely outside of the digital system through the use of a stereo headphone-to-line-in converter manufactured by Apple Inc. The converters are fairly common and can be found from other vendors.

2.1.2 Amplifier

Part of the AC97 code, this module amplifies the analog signal before it is converted to digital.

2.1.3 Analog-to-Digital Converter (ADC)

Part of the AC97 codec, this module takes an amplified analog signal and converts it to digital. This allows the remainder of the system to analyze the signal and generate the sheet music. The module also ensures synchronization with the digital system.

2.1.4 AC97 Codec

The AC97 codec samples the analog audio at $48KHz$, providing a `ready` signal to the signal processing component at that rate. The sampling rate of the codec is important when analyzing the frequency histogram produced by the FFT in the signal processing component. The accuracy of the FFT depends on both the FFT's point size, and the sampling rate of the audio codec. The higher the sampling rate, the lower resolution the output for the FFT will be, however the more accurately the audio sample will be represented.

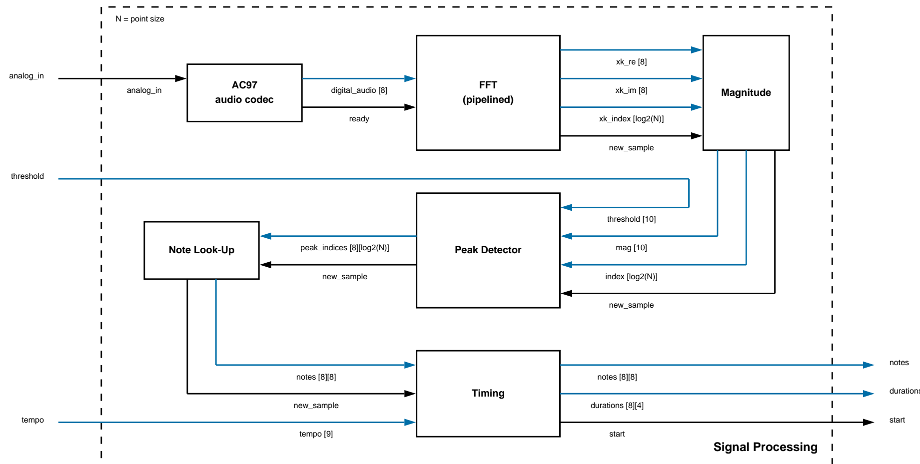

Figure 3: Signal Processing Modules

2.2 Signal Processing (Karl Rieb)

The signal processing module takes the digital audio signal from the audio input module and applies a fourier transform to analyze the signal in the frequency domain. A peak detecting module determines the note frequencies in the audio sample. A look-up table is used to determine the note and the information is sent to the video display module to be displayed. See **Figure 3**.

2.2.1 Fast Fourier Transform (FFT)

The FFT core computes an N-point Discrete Fourier Transform (DFT) where N was chosen to be $2^12 = 4096$. The transform is computed based on the Cooley-Tukey algorithm [2]. Computation of the transform is pipelined, providing streaming results. Using the streaming I/O interface, the system is able to display real-time results to the user and avoids having to keep excessive audio samples in memory. Input to the module is unscaled, rounding is done by using truncation, and index output is in bit-reverse order. A clock enable pin is used to ensure the module only loads in new data when the audio codec is ready. The FFT core module is generated using Xilinx software and is well documented by Xilinx Inc [2].

The FFT module reads audio samples from the audio codec at $48KHz$ and

applies a fast fourier transform to them using a point size of 4096. The chosen point size allows for frequency resolution of approximately $11.72Hz$, enough to accurately differentiate clear pitches with frequencies greater than or equal to the note $D\#3$ ($155.56Hz$). Selecting a larger point size would improve the accuracy of the system at the expense of space and time, whereas a smaller point size would not provide sufficient resolution to determine pitches. The chosen point size is considered the best trade-off between size, speed, and accuracy for the purposes of the system. Increasing the point size would only slightly improve note recognition at the expense of greatly increasing memory usage and computation delays.

The FFT module outputs real and imaginary values with an associated index. The magnitude of the real and imaginary values determines the presence of a frequency in the audio sample. The index determines the frequency by the equation:

$$frequency = \frac{sampling\ rate}{point\ size} \times index$$

2.2.2 Magnitude

The magnitude module takes in two values and computes the magnitude by the equation:

$$magnitude = \sqrt{x^2 + y^2}$$

The module is used to compute the magnitude of the real and imaginary components that are outputs to the FFT module. Since the computation involes finding the square root of possibly large values, the module is pipelined, allowing a constant stream of data in and out. This is vital since the FFT module is also pipelined and produces a continuous stream of values at every clock cycle.

The algorithm used to solve for the square root depends strongly on the size of the input, therefore the magnitude and square root modules were both parameterized and produce pipelines of arbitrary lengths. Using large inputs will consequently increase the delay in the module. The delay to compute the square root, where N is the number of bits of the input is,

$$delay = N + 2\ cycles$$

Since the system does not compute magnitudes of values wider than 31 bits, the longest delay possibly introduced by the module would be approximately $1.30\mu s$.

2.2.3 Peak Detector

The peak detector module maintains an array of index values sorted by their associated magnitudes using the insertion sort algorithm. To improve performance, the insertion sort algorithm is optomized to peform array value shifts and simultaneous comparisons in one clock cycle. Since the module must be capable of handling streaming data from the FFT and magnitude modules, the peak detector module is pipelined with 3 stages.

The first stage compares the new magnitude value to other values in the array or pipeline simultaneous, storing the results into a boolean array indicating whether or not the magnitude was greater than the other values. The second stage takes the information stored in the boolean array and shifts certain array values based on the insertion sort algorithm. The new magnitude, if large enough, is inserted into the array. The final stage of the pipeline takes the current values in the array and sends them to output.

The module is parameterized to allow for an arbitrary size for the sorted array. This permits the system to find any number of peak frequencies from the output of the FFT, creating a system capable of recognizing multiple notes in a given audio sample. Depending on the type of music being played, more or less notes may be required to be detected, and permitting the system to change the value improves accuracy by reducing noise data or including previously ignored frequencies.

2.2.4 Peak Filter

The peak filter module takes the peaks found by the peak detector, and filters them with an user specified threshold. The peak filter attempts to determine which notes were played in the audio sample by only outputing the peaks with the largest magnitudes. This is determined by multiplying the threshold to the magnitude of the peak and comparing the value to the largest peak. Depending on the type of music being sampled, the threshold may need to be adjusted appropriately.

Due to time constraints and unexpected system behavior, this module was never implemented.

2.2.5 Note Look-Up

The note look-up module is a table of music notes and their associated frequencies. The module takes the index values found by the peak detector and converts the index values to a frequency using the equation:

$$frequency = \frac{sampling\ rate}{point\ size} \times index$$

Once the frequency of the index is known, the associated note is determined. A note consists of an octave, pitch, and accidental (in our system only the sharp, #, accidental is utilized). To excessive combinational delays, the note look up table searches hierarchically, first determining the octave, then finding the appropriate pitch and sharp. A set of muxes are used to implement the design.

Since the note look-up module only supports finding the associated note for a single index value, the module is instantiated multiple times according to the number of peaks being detected by the peak detector module.

For determining the associated musical note with a given frequency, the frequency values in table **1** were used.

2.2.6 Metronome

Maintains a predefined tempo by outputting an active-high pulse signal on every 32nd beat. Number of clock cycles per beat is determined by the sampling rate of the AC97 codec. The module uses an active-high sample input signal for notification of the sampling rate. The module is implemented mostly as a simple divider.

By choosing to output 32nd beats, the module allows the duration module to more leniently decide duration of chords.

2.2.7 Duration

The duration module decides the duration of a chord of notes detected by the peak detector module. Using the metronome module, the duration module keeps count of beats and checks for any note changes in each new sample. When note changes are detected, the duration module sends the previous notes and their duration to the video modules to be displayed, and begins timing the new chord.

To determine whether any changes have occured, the module ensures that each note in the previous chord is present in the new chord and that the number of notes in the previous chord are thes same and that of the new chord. If either condition is false, then module considers the new chord as a different chord.

If there are N peaks being detected by the peak detector, then the duration module requires N^2 comparisons to check whether all the notes in the previous chord are present in the new chord. The comparisons are done in parellel to avoid latency.

2.3 Video Display (Grace Cheung)

The video display module takes the notes signals and duration output by the signal processing module and converts the chord into sheet music on the display screen. The sheet music displays two clefs, bass and treble, with chords and/or rests in each clef. The appearance of the chord indicates the chord's duration.

The module is comprised of four submodules: note position, sprite modules, XVGA and frame buffer (see **Figure 4**).

2.3.1 Note Position

The note position module uses parameters defining the size and spacing of the clefs along with the pitch and octave of the note being played to calculate the x- and y-coordinate of each note in the chord to be displayed. Since the module is instantiated multiple times (once for each note in the chord), a base reference frame provided by the video display module is used for its

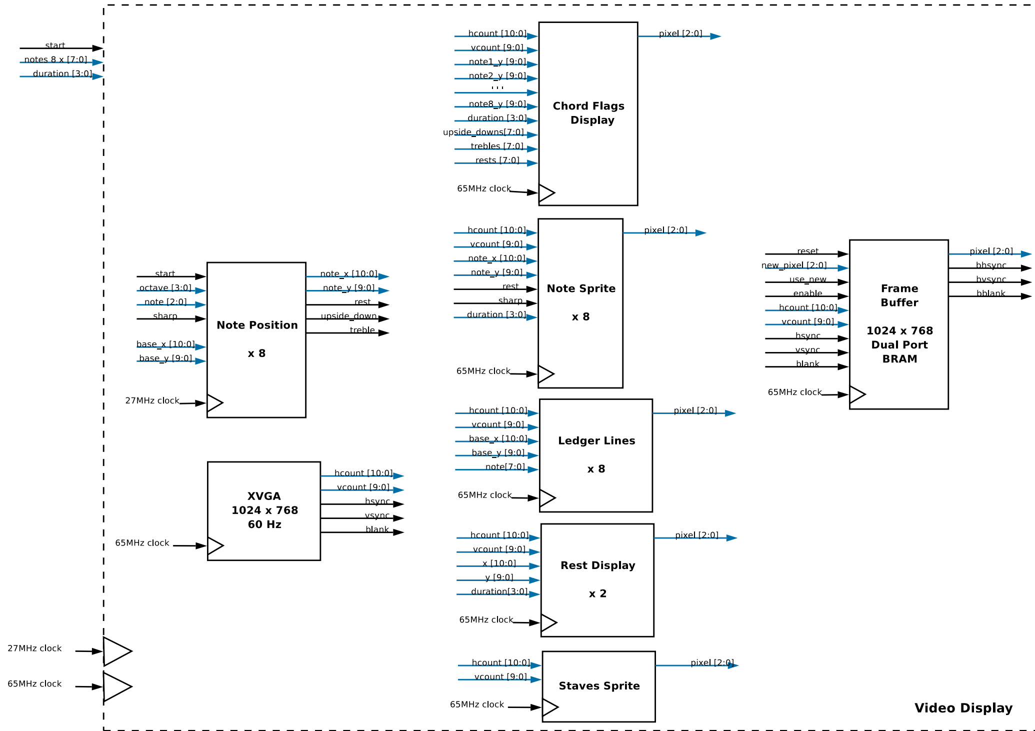

Figure 4: Video Display Modules

calculations, avoiding redunt computation of the same values. The video display module updates the base frame whenever it receives a new chord from the signal processing component.

2.3.2 Note Sprite

The note sprite module uses a ROM containing note related sprites to display the appropriate note head, accidental, and duration dot. The module is entirely combinational with the longest delay being reading from the sprite ROM.

Since the note sprite module is instantiated once for every note in the chord, it does not attempt to draw a stem or flag on any given note. To appropriately draw the stems and flags, the module must have know the position of all the notes played in the chord.

2.3.3 Chord Flag Display

The chord flag display uses information about the position of all the notes in the chord to correctly draw stems connecting the notes and flags indicating the notes' durations. The module uses a ROM containin note related sprites to display the flags.

Chords are displayed per clef, so to determine the length and position of a stem, the chord flag display module must know the highest and lowest treble and bass notes. This is accomplished by comparing all the notes in the chord to eachother to find maximum and minimum values for their y-coordinates. To reduce latency, pairs of notes are simultaneously compared, with the results being recursively compared.

Once top and bottom treble and bass notes are known, the module draws the stems on the left side of the notes if the top-most note is above the third staff line (making the notes essentially upside down), or on the right if the top-most note is on the third staff line or below. The flag sprites are draw on the end of the stems either right side up or upside down.

2.3.4 Rest Display

The rest display module uses a ROM containing note related sprites to display the appropriate rest symbol when no notes in the chord are in a particular clef. Two instances of this module are used by the video display to draw rests for the treble and bass clefs.

2.3.5 Staves Sprite

The staves sprite module displays the background for the screen, which consists of multiple staves with treble and bass clefs. The treble and bass cleff symbols are drawn using a ROM containing clef sprites. The lines for the staves are generated through logic.

Only one instance of this module exists in the video display and is independent of the chords being played. The output of the module should be consistent.

2.3.6 XVGA

The XVGA module takes in a high frequency clock input and outputs the video signals to the sprite modules and the frame buffer. The resolution and refresh rate are set at 1024×768 and $60Hz$, respectively. The video signals sent to the other modules are later collected and sent out of the video display module to a monitor.

2.3.7 Frame Buffer

The frame buffer module contains a copy of the display in a 1024×768" dual-port BRAM. Since stave sprites are independent of the input and always appear on the screen, they are not stored in memory. Due to limitations on the size of usable BRAM, only 1-bit values are used to store information, making the frame buffer a boolean array indicating the presence of a pixel.

When reading from the frame buffer to display an image on the screen, the boolean pixel value is converted to a predefined colored pixel and sent as output to the video display, where it is overlayed on the staves sprite module's pixels. Due to timing constraints and to avoid writing glitchy data to the buffer, an enable input is used to activate and deactivate writes to the buffer. The video display disables writes to the buffer when receiving a new chord, and re-enables writing to memory only after a delay to ensure glitches have passed through the system.

3 Testing

Modules in the signal processing and video display components were tested using different techniques best suited for the given module. The techniques include simulation, test frameworks using user input, and displaying output to a monitor. Integration of the different modules was also tested.

3.1 Signal Processing (Karl Rieb)

Since the signal processing component of the system dealt entirely with manipulating data and performing mathematical computations, testing was mostly done using simulation and testing frameworks. A testing framework is a separate system that uses the modules in *PerfectPitch* independently, allowing a user to select arbitrary inputs and see the resulting output. The selection of input is limited to the input methods provided by the FPGA used. All simulation was done on a Linux system running Gentoo using the open-source *Icarus Verilog v0.8.2* compiler and simulator. All waveform data was viewed by *GTKWave v3.0.22*.

3.1.1 FFT

The FFT generated by the Xilinx software was tested using multiple testing frameworks. Different point sizes, input methods, and output analyzing were used. The following point sizes were generated: 128, 256, 1024, 4096, 16384. Each FFT was pipelined using streaming I/O, unscaled, truncated when rounding, reverse-bit output, and used an optional clock enable (CE) pin. The clock enable pin was wired to either the audio codec's ready signal or constant active-high value (determined by user input). The imaginary input value was set to zero, while the real input value was taken from either the audio codec or a predefined $750Hz$ pure tone (determined by user input). The input was scaleable by the user. Output was analyzed using a logic analyzer and displaying the frequency histogram on a monitor.

Albeit extensive testing, FFT behavior was unable to be consistently reproduced. When reversing the bits of the index output, the FFT worked temporarily, until recompiled. When changes that were made were undone, the previous working FFT no longer functioned correctly. There was also an occasion when the FFT functioned properly when the index bits were not

reversed. This observation is troubling since each FFT was generating with the index bits knowingly reversed.

Common issues with FFT outputs were high magnitudes at beginning and end indices (with relatively zero magnitude elsewhere), and random magnitude output that seemed to have no correlation to its input. On occasion the FFT would have magnitude peaks at index values that were powers of 2.

Example source code was used after multiple failures, but still produced no usable results. Only twice did the FFT function, and both instances were unable to be reproduced after attempts to modify the FFT to work with the other modules.

3.1.2 Magnitude, Peak Detector, Note Look-Up

The magnitude, peak detector, and note look-up modules were tested using simulation on a wide range of data inputs. The modules were later tested using a testing framework. In both cases, the modules produced valid results at the expected delays.

3.1.3 Metronome

The metronome module was tested using a testing framework. Each predefined tempo in the metronome module was verified to produce beats at the correct delays. A beat counter displayed a count of 32nd beats and quarter beats. After a minute, the quarter beats were checked to see if they matched the tempo. In all tests, the metronome module output valid values.

3.1.4 Duration

The duration module was tested using a testing framework. Testing performed on the duration module was limited due to difficult user input methods. Although the module did output correct values for general cases, the module lacked leniency when calculating durations. Subtle, temporary changes in pitch caused the duration module to signal a chord change. Not all music is perfectly played, therefore such strict analysis of chord duration is poorly suited for the system.

Due to time constraints and problems with the FFT module, the duration module was not improved to better handle chord changes.

3.2 Video Display (Grace Cheung)

Due to the graphic nature of the video display modules and limited memory to store snapshots of display values, all testing for the video modules was done using a monitor to display outputs. On occasion when there was noticeable, unexpected behavior, the outputs were viewed through a logic analyzer.

3.2.1 Sprites

Sprites were tested multipled times throughout the development of the video display component. Initial testing involed displaying all sprites on the screen to ensure they were properly read from the sprite ROM. After inclusion of other modules, many time delays were introduced into the system, causing glitches to occur in the sprites. The glitches were resolved by pipelining certain modules, creating clean transitions from previous values to new values. This change solved all glitch problems with the sprites.

3.2.2 Note Positions

Throughout testing, strange behavior occured with note positioning. The note position module outputs valid values, however the video display was making errors when incrementing the base reference frame coordinates. Certain boundary condition testing was not being computed quick enough and caused strange addition errors when incrementing the base coordinates. After removing the condition statement and placing it as combination logic that always runs regardless of the clock, the problem stopped frequently reoccuring, but would not completely disappear. The actual cause of the error is still uncertain.

3.2.3 Frame Buffer

The frame buffer was tested by using a separate color to differentiate between pixels that came from the buffer. For the most part the frame buffer had no issues, except for an occasional 1 or 2 pixels that should not have appeared. The unexpected pixels were not consistently in the same place, and refreshing the buffer removed them. The assumption was made that

the buffer is responding to glitchy input, although many steps were taken to remove glitches from the system by disabling the frame buffer while values are being determined.

Note	Frequency (Hz)
D#3	155.56
E3	164.81
F3	174.61
F#3	185.00
G3	196.00
G#3	207.65
A3	220.00
A#3	233.08
B3	246.94
C4	261.63
C#4	277.18
D4	293.66
D#4	311.13
E4	329.63
F4	349.23
F#4	369.99
G4	392.00
G#4	415.30
A4	440.00
A#4	466.16
B4	493.88
C5	523.25
C#5	554.37
D5	587.33
D#5	622.25
E5	659.26
F5	698.46
F#5	739.99
G5	783.99
G#5	830.61
A5	880.00
A#5	932.33
B5	987.77
C6	1046.50

Table 1: Note Frequency Correlation [1]

4 Conclusion

Music scores are not available for all music, and in many cases certain songs will go forever untranscribed. For those with perfect pitch, transcribing a song is relatively easy. However only a small percentage of the population has the ability. The *PerfectPitch* system provides a hardware solution to the problem by analyzing any audio sample and producing sheet music for it.

The system is capable of detecting an arbitrary number of pitches played in any audio sample, timing their duration, and displaying the pitches as a chord on a screen. Although one of the core components of the system was unable to function correctly (the FFT module), other research has managed to use the component successfully and could at a later point replace the current module in the system.

Limited by the inability to detect instruments or filter out harmonics, the system is unable to produce completely accurate scores. However, the system is incredibly modular, allowing for extense modifications to better suit specific needs. The design of the system can be easily changed to allow for further functionality or improved pitch detection. Further reseach can produce a way to detect instruments, allowing the knowledge of that instruments harmonics help in detecting the pitch played. Additionally, better filtering techniques for eliminating noise and greater leniency in chord duration detection can be easily incorporated into the system.

References

[1] B. H. Suits. *http://www.phy.mtu.edu/ suits/notefreqs.html*. Michigan-Tech University Physics Department, 1998.

[2] Xilinx Inc. *Xilinx LogiCore: Fast Fourier Transform v5.0*, DS260 edition, October 2007.

A Appendix: Magnitude

```
module magnitude_pipelined( clk, vreal, vimaginary,
    index, mag, index_out );
   // SQRT_PIPELINED delay = [(SQR_BITS + 2) / 2] + 1 cycles
   parameter N_BITS = 8;
   parameter INDEX_BITS = 10;
   parameter SQR_BITS = (N_BITS) * 2;
   parameter MAG_BITS = N_BITS + 1;
   parameter DELAY_CYCLES = ((SQR_BITS + 2) / 2) + 1;

   input         clk;
   input signed [N_BITS - 1:0] vreal;
   input signed [N_BITS - 1:0] vimaginary;
   input [INDEX_BITS - 1:0]    index;

   output [INDEX_BITS - 1:0]   index_out;
   output [MAG_BITS - 1:0]     mag;

   reg [SQR_BITS - 1:0]        vreal_sqr;
   reg [SQR_BITS - 1:0]        vimaginary_sqr;
   reg [SQR_BITS:0]         sum;
   reg [INDEX_BITS - 1:0]      index_stage_one;
   reg [INDEX_BITS - 1:0]      index_stage_two;
   reg [INDEX_BITS - 1:0]      index_pipe [DELAY_CYCLES:0];
   reg [31:0]         i;

   wire [MAG_BITS - 1:0]       mag;
   wire [INDEX_BITS - 1:0]     index_out;

   assign         index_out = index_pipe[DELAY_CYCLES];

   sqrt_pipelined sqrtmag( .clk(clk),
   .data(sum),
   .answer(mag) );

   defparam         sqrtmag.N_BITS = SQR_BITS + 1;

   always @ ( posedge clk ) begin
```

```
      // Square
      vreal_sqr <= vreal * vreal;
      vimaginary_sqr <= vimaginary * vimaginary;
      index_stage_one <= index;


      // Add
      sum <= vreal_sqr + vimaginary_sqr;
      index_stage_two <= index_stage_one;

      // Sqrt will arrive
      // [(SQR_BITS + 2) / 2] + 1 cycles
      // later
      index_pipe[0] <= index_stage_two;

      for( i = DELAY_CYCLES; i > 0; i = i - 1 ) begin
 index_pipe[i] <= index_pipe[i - 1];
      end
   end // always @ ( posedge clk )

endmodule // magnitude
```

B Appendix: Square Root

```
module sqrt_pipelined( clk, data, answer );
   parameter N_BITS = 8;
   parameter M_BITS = (N_BITS + 1)/2;

   input      clk;
   input [N_BITS - 1:0]      data;

   output reg [M_BITS - 1:0] answer;

   reg [31:0]       i;
   reg [N_BITS - 1:0]       datap [M_BITS:0];
   reg [M_BITS - 1:0]       ans [M_BITS:0];

   always @ ( posedge clk ) begin
      // Receive New Data
      datap[0] <= data;
      ans[0] <= 0;

      // Compute Rest Data
      for( i = M_BITS; i > 0; i = i - 1 ) begin
          datap[i] <= datap[i - 1];

          //trial[i] <= ans[i - 1] | (1 << M_BITS - i);

          if( (ans[i - 1] | (1 << M_BITS - i)) *
            (ans[i - 1] | (1 << M_BITS - i)) <= datap[i-1] ) begin
            ans[i] <= ans[i - 1] | (1 << M_BITS - i);
         end else begin
            ans[i] <= ans[i - 1];
         end
      end // for ( i = M_BITS; i > 0; i = i - 1 )

      answer <= ans[M_BITS];
   end // always @ ( posedge clk )
endmodule // sqrt_pipelined
```

C Appendix: Peak Detector

```
module peak_detector( clk, reset, index, mag, peak_indices, debug );
   parameter N_PEAKS = 8;
   parameter MAG_BITS = 9;
   parameter INDEX_BITS = 10;
   parameter PK_BITS = N_PEAKS * INDEX_BITS;

   input     clk;
   input     reset;
   input [INDEX_BITS - 1:0] index;
   input [MAG_BITS - 1:0]   mag;


   output [PK_BITS - 1:0]   peak_indices;

   output [63:0]      debug;

   reg [MAG_BITS - 1:0]     mags [N_PEAKS-1:0];
   reg [PK_BITS - 1:0]      peak_indices;
   reg [INDEX_BITS - 1:0]   pk_indices [N_PEAKS-1:0];
   reg [N_PEAKS - 1:0]      greater_than;

   reg [MAG_BITS - 1:0]     mag_pipe;
   reg [INDEX_BITS - 1:0]   index_pipe;
   reg [PK_BITS - 1:0]      zero;
   reg      reseted;


   wire [N_PEAKS - 1:0]     is_insert_index;
   wire [N_PEAKS - 1:0]     shift_down;

   assign      is_insert_index = { greater_than[N_PEAKS-1],
greater_than[N_PEAKS-1:1] ^
greater_than[N_PEAKS-2:0] };

   assign      shift_down = greater_than;

   assign      debug = { pk_indices[6],pk_indices[5],
```

```
     pk_indices[4],pk_indices[3],
     pk_indices[2],pk_indices[1],
     pk_indices[0],index_pipe };

  integer      step;
  integer      i;
  always @ ( posedge clk ) begin
     if( reset ) begin
for( i = 0; i < N_PEAKS; i = i + 1 ) begin
   mags[i] <= 0;
   pk_indices[i] <= 0;
   greater_than[i] <= 0;
end

reseted <= 1;
peak_indices <= 0;
mag_pipe <= 0;
index_pipe <= 0;
zero = 0;
     end else begin
if( ~(index == index_pipe) || reseted ) begin // if ( reset )
   reseted <= 0;
   // Push things down the pipe
   mag_pipe <= mag;
   index_pipe <= index;

   // Find if the magnitude is greater
   // than anything currently in the list
   for( i = 0; i < N_PEAKS; i = i + 1 ) begin
      greater_than[i] <= mag > (is_insert_index[i] ? mag_pipe :
((i < (N_PEAKS - 1)) ?
 (shift_down[i+1] ? mags[i + 1] : mags[i]) :
 mags[i]));
   end
end else begin // if ( ~(index == index_pipe) || reseted )
   for( i = 0; i < N_PEAKS; i = i + 1 ) begin
      greater_than[i] <= 0;
   end
end // else: !if( ~(index == index_pipe) || reseted )
```

```
 for( i = 0; i < N_PEAKS; i = i + 1 ) begin
    // Figure out where to insert
    // if to insert
    if( is_insert_index[i] ) begin
       pk_indices[i] <= index_pipe;
       mags[i] <= mag_pipe;
    end

    if( shift_down[i] && (i > 0) ) begin
       pk_indices[i-1] <= pk_indices[i];
       mags[i-1] <= mags[i];
    end
 end // for ( i = 0; i < N_PEAKS; i++ )

 zero = 0;
 for( i = 0; i < N_PEAKS; i = i + 1 ) begin
    zero = zero + (pk_indices[i] << (i*(INDEX_BITS)));
 end
 peak_indices <= zero;
      end // else: !if( reset )
   end // always @ ( posedge clk )
endmodule // peak_detector
```

D Appendix: Note Look-Up

```
module note_lookup(clk, peak_index, octave, note, sharp);
   parameter SAMPLING_RATE = 48000; // Sampling rate in Hz
   parameter SHIFT_BITS = 12;     // Point size of the FFT (2^12)
   parameter INDEX_BITS = 10;

   parameter C = 0;
   parameter D = 1;
   parameter E = 2;
   parameter F = 3;
   parameter G = 4;
   parameter A = 5;
   parameter B = 6;

   parameter SHARP = 1;

   parameter OCTAVE_0 = 0;
   parameter OCTAVE_1 = 1;
   parameter OCTAVE_2 = 2;
   parameter OCTAVE_3 = 3;
   parameter OCTAVE_4 = 4;
   parameter OCTAVE_5 = 5;
   parameter OCTAVE_6 = 6;
   parameter OCTAVE_7 = 7;
   parameter OCTAVE_8 = 8;


   input     clk;
   input [INDEX_BITS - 1:0] peak_index;

   output [3:0]      octave;
   output [2:0]      note;
   output      sharp;

   wire [12:0]  peak_freq;
   wire [31:0]  mult;
   wire [2:0]  note0;
   wire [2:0]  note1;
```

```
  wire [2:0]  note2;
  wire [2:0]  note3;
  wire [2:0]  note4;
  wire [2:0]  note5;
  wire [2:0]  note6;
  wire [2:0]  note7;
  wire [2:0]  note8;

  wire [2:0]  sharp0;
  wire [2:0]  sharp1;
  wire [2:0]  sharp2;
  wire [2:0]  sharp3;
  wire [2:0]  sharp4;
  wire [2:0]  sharp5;
  wire [2:0]  sharp6;
  wire [2:0]  sharp7;
  wire [2:0]  sharp8;

  // Splitting up the multiply and shift
  // is required since there is some weird bit-size casting
  // involved
  assign  mult = SAMPLING_RATE*peak_index;
  assign  peak_freq = (mult >> SHIFT_BITS);

  assign  octave = (peak_freq < 254) ?

((peak_freq < 64) ?
 ((peak_freq < 32) ? OCTAVE_0 : OCTAVE_1) :
 ((peak_freq < 127) ? OCTAVE_2 : OCTAVE_3)) :

((peak_freq < 1017) ?
 ((peak_freq < 509) ? OCTAVE_4 : OCTAVE_5) :
 ((peak_freq < 4069) ?
  ((peak_freq < 2039) ? OCTAVE_6 : OCTAVE_7) :
  OCTAVE_8));


  assign  note = (octave == OCTAVE_4) ? note4 :
      (octave == OCTAVE_3) ? note3 :
      (octave == OCTAVE_5) ? note5 :
```

```
        (octave == OCTAVE_6) ? note6 :
        (octave == OCTAVE_2) ? note2 :
        (octave == OCTAVE_7) ? note7 :
        (octave == OCTAVE_1) ? note1 :
        (octave == OCTAVE_0) ? note0 : note8;

assign  sharp = (octave == OCTAVE_4) ? sharp4 :
        (octave == OCTAVE_3) ? sharp3 :
        (octave == OCTAVE_5) ? sharp5 :
        (octave == OCTAVE_6) ? sharp6 :
        (octave == OCTAVE_2) ? sharp2 :
        (octave == OCTAVE_7) ? sharp7 :
        (octave == OCTAVE_1) ? sharp1 :
        (octave == OCTAVE_0) ? sharp0 : sharp8;

   /* OCTAVE 0 */
   assign  note0 = (peak_freq < 21) ?             // Split at E-F

((peak_freq < 18) ? C :         // Split at C-D
 ((peak_freq < 20) ? D : E)) :  // Split at D-E

((peak_freq < 27) ?             // Split at G-A
 ((peak_freq < 24) ? F : G) :   // Split at F-G
 ((peak_freq < 30) ? A : B));   // Split at A-B

   /* OCTAVE 1 */
   assign  note1 = (peak_freq < 42) ?             // Split at E-F

((peak_freq < 36) ? C :         // Split at C-D
 ((peak_freq < 40) ? D : E)) :  // Split at D-E

((peak_freq < 54) ?             // Split at G-A
 ((peak_freq < 48) ? F : G) :   // Split at F-G
 ((peak_freq < 60) ? A : B));   // Split at A-B

   /* OCTAVE 2 */
   assign  note2 = (peak_freq < 85) ?             // Split at E-F

((peak_freq < 71) ? C :         // Split at C-D
 ((peak_freq < 80) ? D : E)) :  // Split at D-E
```

```
((peak_freq < 107) ?            // Split at G-A
 ((peak_freq < 95) ? F : G) :   // Split at F-G
 ((peak_freq < 120) ? A : B));  // Split at A-B

   /* OCTAVE 3 */
   assign  note3 = (peak_freq < 170) ?             // Split at E-F

((peak_freq < 143) ? C :        // Split at C-D
 ((peak_freq < 160) ? D : E)) : // Split at D-E

((peak_freq < 213) ?            // Split at G-A
 ((peak_freq < 190) ? F : G) :  // Split at F-G
 ((peak_freq < 240) ? A : B));  // Split at A-B

   /* OCTAVE 4 */
   assign  note4 = (peak_freq < 340) ?             // Split at E-F

((peak_freq < 285) ? C :        // Split at C-D
 ((peak_freq < 320) ? D : E)) : // Split at D-E

((peak_freq < 428) ?            // Split at G-A
 ((peak_freq < 381) ? F : G) :  // Split at F-G
 ((peak_freq < 480) ? A : B));  // Split at A-B

   /* OCTAVE 5 */
   assign  note5 = (peak_freq < 679) ?             // Split at E-F

((peak_freq < 571) ? C :        // Split at C-D
 ((peak_freq < 641) ? D : E)) : // Split at D-E

((peak_freq < 855) ?            // Split at G-A
 ((peak_freq < 762) ? F : G) :  // Split at F-G
 ((peak_freq < 960) ? A : B));  // Split at A-B

   /* OCTAVE 6 */
   assign  note6 = (peak_freq < 1358) ?              // Split at E-F

((peak_freq < 1142) ? C :         // Split at C-D
 ((peak_freq < 1282) ? D : E)) :  // Split at D-E
```

```
((peak_freq < 1711) ?             // Split at G-A
 ((peak_freq < 1524) ? F : G) :   // Split at F-G
 ((peak_freq < 1920) ? A : B));   // Split at A-B

   /* OCTAVE 7 */
   assign  note7 = (peak_freq < 2715) ?              // Split at E-F

((peak_freq < 2283) ? C :         // Split at C-D
 ((peak_freq < 2563) ? D : E)) :  // Split at D-E

((peak_freq < 3421) ?             // Split at G-A
 ((peak_freq < 3048) ? F : G) :   // Split at F-G
 ((peak_freq < 3840) ? A : B));   // Split at A-B

   /* OCTAVE 8 */
   assign  note8 = (peak_freq < 4567) ? C : D;       // Split at C-D
```

```
/* OCTAVE 0 */
   assign  sharp0 = (peak_freq < 21) ?              // Split at E-F

((peak_freq < 18) ? (peak_freq > 17) :         // Split at C-D
 ((peak_freq < 20) ? (peak_freq > 19) : 0)) :  // Split at D-E

((peak_freq < 27) ?             // Split at G-A
 ((peak_freq < 24) ? (peak_freq > 22) : (peak_freq > 25)) :   // Split at F-G
 ((peak_freq < 30) ? (peak_freq > 28) : 0));   // Split at A-B

   /* OCTAVE 1 */
   assign  sharp1 = (peak_freq < 42) ?              // Split at E-F

((peak_freq < 36) ? (peak_freq > 34) :         // Split at C-D
 ((peak_freq < 40) ? (peak_freq > 38) : 0)) :  // Split at D-E

((peak_freq < 54) ?             // Split at G-A
 ((peak_freq < 48) ? (peak_freq > 45) : (peak_freq > 50)) :   // Split at F-G
 ((peak_freq < 60) ? (peak_freq > 57) : 0));   // Split at A-B

   /* OCTAVE 2 */
   assign  sharp2 = (peak_freq < 85) ?              // Split at E-F

((peak_freq < 71) ? (peak_freq > 68) :         // Split at C-D
 ((peak_freq < 80) ? (peak_freq > 76) : 0)) :  // Split at D-E
```

```
((peak_freq < 107) ?            // Split at G-A
 ((peak_freq < 95) ? (peak_freq > 90) : (peak_freq > 101)) :   // Split at F-G
 ((peak_freq < 120) ? (peak_freq > 113) : 0));  // Split at A-B

   /* OCTAVE 3 */
   assign  sharp3 = (peak_freq < 170) ?             // Split at E-F

((peak_freq < 143) ? (peak_freq > 135) :        // Split at C-D
 ((peak_freq < 160) ? (peak_freq > 150) : 0)) : // Split at D-E

((peak_freq < 213) ?            // Split at G-A
 ((peak_freq < 190) ? (peak_freq > 180) : (peak_freq > 201)) :  // Split at F-G
 ((peak_freq < 240) ? (peak_freq > 227) : 0));  // Split at A-B

   /* OCTAVE 4 */
   assign  sharp4 = (peak_freq < 340) ?             // Split at E-F

((peak_freq < 285) ? (peak_freq > 269) :        // Split at C-D
 ((peak_freq < 320) ? (peak_freq > 298) : 0)) : // Split at D-E

((peak_freq < 428) ?            // Split at G-A
 ((peak_freq < 381) ? (peak_freq > 359) : (peak_freq > 403)) :  // Split at F-G
 ((peak_freq < 480) ? (peak_freq > 453) : 0));  // Split at A-B

   /* OCTAVE 5 */
   assign  sharp5 = (peak_freq < 679) ?             // Split at E-F

((peak_freq < 571) ? (peak_freq > 538) :        // Split at C-D
 ((peak_freq < 641) ? (peak_freq > 603) : 0)) : // Split at D-E

((peak_freq < 855) ?            // Split at G-A
 ((peak_freq < 762) ? (peak_freq > 719) : (peak_freq > 807)) :  // Split at F-G
 ((peak_freq < 960) ? (peak_freq > 906) : 0));  // Split at A-B

   /* OCTAVE 6 */
   assign  sharp6 = (peak_freq < 1358) ?              // Split at E-F

((peak_freq < 1142) ? (peak_freq > 1078) :         // Split at C-D
 ((peak_freq < 1282) ? (peak_freq > 1210) : 0)) :  // Split at D-E
```

```
((peak_freq < 1711) ?             // Split at G-A
 ((peak_freq < 1524) ? (peak_freq > 1438) : (peak_freq > 1615)) :   // Split at F-G
 ((peak_freq < 1920) ? (peak_freq > 1812) : 0));   // Split at A-B

   /* OCTAVE 7 */
   assign  sharp7 = (peak_freq < 2715) ?             // Split at E-F

((peak_freq < 2283) ? (peak_freq > 2155) :         // Split at C-D
 ((peak_freq < 2563) ? (peak_freq > 2419) : 0)) :  // Split at D-E

((peak_freq < 3421) ?             // Split at G-A
 ((peak_freq < 3048) ? (peak_freq > 2877) : (peak_freq > 3229)) :   // Split at F-G
 ((peak_freq < 3840) ? (peak_freq > 3625) : 0));   // Split at A-B

   /* OCTAVE 8 */
   assign  sharp8 = (peak_freq < 4567) ? (peak_freq > 4310) : (peak_freq > 4838);
endmodule
```

E Appendix: Duration

```
module chord_duration( clk, beat, reset,
  note1, note2, note3, note4,
  note5, note6, note7, note8,
  duration, notes, new_chord,
  debug );
parameter SIXTEENTH = 0;
parameter SIXTEENTH_DOT = 1;
parameter EIGHTH = 2;
parameter EIGHTH_DOT = 3;
parameter QUARTER = 4;
parameter QUARTER_DOT = 5;
parameter HALF = 6;
parameter HALF_DOT = 7;
parameter WHOLE = 8;


parameter SIXTEENTH_CUTOFF = 2;
parameter SIXTEENTH_DOT_CUTOFF = 3;
parameter EIGHTH_CUTOFF = 5;
parameter EIGHTH_DOT_CUTOFF = 7;
parameter QUARTER_CUTOFF = 10;
parameter QUARTER_DOT_CUTOFF = 14;
parameter HALF_CUTOFF = 20;
parameter HALF_DOT_CUTOFF = 28;

input clk;
input  beat;
input reset;
input [7:0] note1;
input [7:0] note2;
input [7:0] note3;
input [7:0] note4;
input [7:0] note5;
input [7:0] note6;
input [7:0] note7;
input [7:0] note8;
```

```
output reg [3:0] duration;
output reg [63:0] notes;
output reg new_chord;
output [7:0] debug;

reg  [7:0] hold_note1;
reg [7:0] hold_note2;
reg  [7:0] hold_note3;
reg [7:0] hold_note4;
reg  [7:0] hold_note5;
reg [7:0] hold_note6;
reg  [7:0] hold_note7;
reg [7:0] hold_note8;

reg [5:0] beat_count = 0;

wire contains1 = (hold_note1 == note1) |
  (hold_note1 == note2) |
  (hold_note1 == note3) |
  (hold_note1 == note4) |
  (hold_note1 == note5) |
  (hold_note1 == note6) |
  (hold_note1 == note7) |
  (hold_note1 == note8);
wire contains2 = (hold_note2 == note1) |
  (hold_note2 == note2) |
  (hold_note2 == note3) |
  (hold_note2 == note4) |
  (hold_note2 == note5) |
  (hold_note2 == note6) |
  (hold_note2 == note7) |
  (hold_note2 == note8);
wire contains3 = (hold_note3 == note1) |
  (hold_note3 == note2) |
  (hold_note3 == note3) |
  (hold_note3 == note4) |
  (hold_note3 == note5) |
  (hold_note3 == note6) |
  (hold_note3 == note7) |
  (hold_note3 == note8);
```

```
wire contains4 = (hold_note4 == note1) |
  (hold_note4 == note2) |
  (hold_note4 == note3) |
  (hold_note4 == note4) |
  (hold_note4 == note5) |
  (hold_note4 == note6) |
  (hold_note4 == note7) |
  (hold_note4 == note8);
wire contains5 = (hold_note5 == note1) |
  (hold_note5 == note2) |
  (hold_note5 == note3) |
  (hold_note5 == note4) |
  (hold_note5 == note5) |
  (hold_note5 == note6) |
  (hold_note5 == note7) |
  (hold_note5 == note8);
wire contains6 = (hold_note6 == note1) |
  (hold_note6 == note2) |
  (hold_note6 == note3) |
  (hold_note6 == note4) |
  (hold_note6 == note5) |
  (hold_note6 == note6) |
  (hold_note6 == note7) |
  (hold_note6 == note8);
wire contains7 = (hold_note7 == note1) |
  (hold_note7 == note2) |
  (hold_note7 == note3) |
  (hold_note7 == note4) |
  (hold_note7 == note5) |
  (hold_note7 == note6) |
  (hold_note7 == note7) |
  (hold_note7 == note8);
wire contains8 = (hold_note8 == note1) |
  (hold_note8 == note2) |
  (hold_note8 == note3) |
  (hold_note8 == note4) |
  (hold_note8 == note5) |
  (hold_note8 == note6) |
  (hold_note8 == note7) |
  (hold_note8 == note8);
```

```
wire [3:0] note_count = |hold_note1[6:4] +
|hold_note2[6:4] +
|hold_note3[6:4] +
|hold_note4[6:4] +
|hold_note5[6:4] +
|hold_note6[6:4] +
|hold_note7[6:4] +
|hold_note8[6:4];
wire [3:0] new_note_count = |note1[6:4] +
 |note2[6:4] +
 |note3[6:4] +
 |note4[6:4] +
 |note5[6:4] +
 |note6[6:4] +
 |note7[6:4] +
 |note8[6:4];

wire chord_change = ~(contains1 & contains2 & contains3 &
 contains4 & contains5 & contains6 &
 contains7 & contains8) |
 (beat_count >= 32) |
 ~( note_count == new_note_count );

assign debug = { 2'd0, beat_count };

always @ (posedge clk) begin
if( reset ) begin
beat_count <= 0;
hold_note1 <= 0;
hold_note2 <= 0;
hold_note3 <= 0;
hold_note4 <= 0;
hold_note5 <= 0;
hold_note6 <= 0;
hold_note7 <= 0;
hold_note8 <= 0;

duration <= 0;
new_chord <= 0;
```

```
end else if( beat ) begin
if( chord_change ) begin
beat_count  <= 0;

notes[7:0]  <= hold_note1;
notes[15:8] <= hold_note2;
notes[23:16] <= hold_note3;
notes[31:24] <= hold_note4;
notes[39:32] <= hold_note5;
notes[47:40] <= hold_note6;
notes[55:48] <= hold_note7;
notes[63:56] <= hold_note8;

hold_note1  <= note1;
hold_note2  <= note2;
hold_note3  <= note3;
hold_note4  <= note4;
hold_note5  <= note5;
hold_note6  <= note6;
hold_note7  <= note7;
hold_note8  <= note8;

if( beat_count > 1 ) begin
new_chord  <= 1;
end else begin
new_chord  <= 0;
end

if( beat_count > HALF_DOT_CUTOFF ) begin
        duration <= WHOLE;
end else if( beat_count > HALF_CUTOFF ) begin
        duration <= HALF_DOT;
end else if( beat_count > QUARTER_DOT_CUTOFF ) begin
        duration <= HALF;
end else if( beat_count > QUARTER_CUTOFF ) begin
        duration <= QUARTER_DOT;
end else if( beat_count > EIGHTH_DOT_CUTOFF ) begin
        duration <= QUARTER;
end else if( beat_count > EIGHTH_CUTOFF ) begin
        duration <= EIGHTH_DOT;
```

```
end else if( beat_count > SIXTEENTH_DOT_CUTOFF ) begin
        duration <= EIGHTH;
end else if( beat_count > SIXTEENTH_CUTOFF ) begin
        duration <= SIXTEENTH_DOT;
end else begin
        duration <= SIXTEENTH;
end
end else begin
        beat_count  <= beat_count + 1;
        new_chord <= 0;
end
end else begin
        new_chord  <= 0;
end
end
endmodule
```

F Appendix: Video Display

```
module video_display( clk, vclock, reset, start,
     notes, duration, raw,

     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,
     debug, logic_a0, logic_a1, logic_a2, logic_a3 );
   parameter SCREEN_WIDTH = 1024;
   parameter SCREEN_HEIGHT = 768;
   parameter NOTE_COLOR = 3'b101;
   parameter START_X = 11'd60;
   parameter START_Y = 10'd40;
   parameter X_INCREMENT = 11'd30;
   parameter Y_INCREMENT = 10'd140;

   input     clk;         // Clock to synchronize with backend modules
   input     vclock;      // ~65 MHz clock used for video display
   input     reset;       // Active-high: resets the display
   input     start;       // Active-high: indicates new notes
   input [63:0] notes;    // \  /
                          //  \/  8 Notes that were played
                          //  |   note[7]   = sharp
                          //  |   note[6:4] = pitch
                          //  |   ( 0 = rest, 1 = C, ..., 7 = B )
                          //  |   note[3:0] = octave ( A4 = 440 Hz )
                          //  /
                          // /
   input [3:0]  duration; // Duration of a note:
                          //  0 = 16th
                          //  1 = 16th + dot
                          //  2 = 8th
                          //  3 = 8th + dot
                          //  4 = quarter
                          //  5 = quarter + dot
                          //  6 = half
                          //  7 = half + dot
```

```
                       //  8 = whole
input        raw;      // Active-high: disables use of frame buffer

output [7:0] vga_out_red;            //
output [7:0] vga_out_green;          //  VGA Output signals that
output [7:0] vga_out_blue;           //  drive the labkit signals
output       vga_out_sync_b;         //  required to display to
output       vga_out_blank_b;        //  the monitor
output       vga_out_pixel_clock;    //
output       vga_out_hsync;          //
output       vga_out_vsync;          //
output [63:0] debug;                 // 64 bits able to
                                     // display in HEX display
output [7:0]  logic_a0;
output [7:0]  logic_a1;
output [7:0]  logic_a2;
output [7:0]  logic_a3;



assign   logic_a0 = 0;
assign   logic_a1 = 0;
assign   logic_a2 = 0;
assign   logic_a3 = 0;

////////////////////////////////////////////////////
// BASIC XVGA SIGNALS
////////////////////////////////////////////////////
wire [10:0]   hcount;
wire [9:0]   vcount;
wire   hsync,vsync,blank;
xvga xvga1( .vclock(vclock),

    .hcount(hcount),
    .vcount(vcount),
    .hsync(hsync),
    .vsync(vsync),
    .blank(blank) );

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

```
  // UPDATE NOTES AT EVERY START
  ////////////////////////////////////////////////////
  reg [7:0]   note1;
  reg [7:0]   note2;
  reg [7:0]   note3;
  reg [7:0]   note4;
  reg [7:0]   note5;
  reg [7:0]   note6;
  reg [7:0]   note7;
  reg [7:0]   note8;
  reg [3:0]   new_duration;
  reg   buffer_we = 0;
  reg   start_pipe1 = 0;
  reg   start_pipe2 = 0;
  reg   refresh = 1;

  reg [10:0]   base_x;
  reg [9:0]   base_y;

  wire   off_x_edge = ( base_x > (11'd1009 - X_INCREMENT) );
  wire [9:0]   new_y = base_y + Y_INCREMENT;
  wire   last_stave = ( base_y >= 10'd600 );
  always @ (posedge clk) begin
     start_pipe1 <= start;
     start_pipe2 <= start_pipe1;

     if( reset ) begin
refresh <= 1;
base_x <= START_X;
base_y <= START_Y;
buffer_we <= 0;
     end if( start ) begin
note1 <= notes[63:56];
note2 <= notes[55:48];
note3 <= notes[47:40];
note4 <= notes[39:32];
note5 <= notes[31:24];
note6 <= notes[23:16];
note7 <= notes[15:8];
note8 <= notes[7:0];
```

```
new_duration <= duration;
refresh <= 0;
buffer_we <= 0;

if( off_x_edge ) begin
   if( last_stave ) begin
      base_x <= START_X;
      base_y <= START_Y;
      refresh <= 1;
   end else begin
      base_x <= START_X;
      base_y <= new_y;
      refresh <= 0;
   end
end else begin
   base_x <= (base_x + X_INCREMENT);
   refresh <= 0;
end
     end

     if( start_pipe2 & ~start_pipe1 & ~start ) begin
buffer_we <= 1;
     end
  end

  ////////////////////////////////////////////////////
  // NOTE POSITIONS
  ////////////////////////////////////////////////////
  wire [10:0] note_x[7:0];
  wire [9:0]  note_y[7:0];
  wire [7:0]  rest;
  wire [7:0]  upside_down;
  wire [7:0]  treble;
  wire [7:0]  sharp;

  wire [63:0]  debug = {8'd0,treble,
8'd0,rest,
6'h00,base_y,
6'h00,new_y};
```

```
  note_position npos1( .clk(clk),
.start(start),
.reset(reset),
.octave(note1[3:0]),
.note(note1[6:4]),
.sharp(note1[7]),
.base_x(base_x),
.base_y(base_y),

.note_x(note_x[0]),
.note_y(note_y[0]),
.rest(rest[0]),
.upside_down(upside_down[0]),
.treble(treble[0]),
.sharp_out(sharp[0]) );
  defparam  npos1.LINE_HEIGHT = 1;
  defparam  npos1.LINE_SPACING = 5;
  defparam  npos1.STAFF_SPACING = 40;

  note_position npos2( .clk(clk),
.start(start),
.reset(reset),
.octave(note2[3:0]),
.note(note2[6:4]),
.sharp(note2[7]),
.base_x(base_x),
.base_y(base_y),

.note_x(note_x[1]),
.note_y(note_y[1]),
.rest(rest[1]),
.upside_down(upside_down[1]),
.treble(treble[1]),
.sharp_out(sharp[1]) );
  defparam  npos2.LINE_HEIGHT = 1;
  defparam  npos2.LINE_SPACING = 5;
  defparam  npos2.STAFF_SPACING = 40;

  note_position npos3( .clk(clk),
.start(start),
```

```
.reset(reset),
.octave(note3[3:0]),
.note(note3[6:4]),
.sharp(note3[7]),
.base_x(base_x),
.base_y(base_y),

.note_x(note_x[2]),
.note_y(note_y[2]),
.rest(rest[2]),
.upside_down(upside_down[2]),
.treble(treble[2]),
.sharp_out(sharp[2]) );
   defparam  npos3.LINE_HEIGHT = 1;
   defparam  npos3.LINE_SPACING = 5;
   defparam  npos3.STAFF_SPACING = 40;

   note_position npos4( .clk(clk),
.start(start),
.reset(reset),
.octave(note4[3:0]),
.note(note4[6:4]),
.sharp(note4[7]),
.base_x(base_x),
.base_y(base_y),

.note_x(note_x[3]),
.note_y(note_y[3]),
.rest(rest[3]),
.upside_down(upside_down[3]),
.treble(treble[3]),
.sharp_out(sharp[3]) );
   defparam  npos4.LINE_HEIGHT = 1;
   defparam  npos4.LINE_SPACING = 5;
   defparam  npos4.STAFF_SPACING = 40;

   note_position npos5( .clk(clk),
.start(start),
.reset(reset),
.octave(note5[3:0]),
```

```
.note(note5[6:4]),
.sharp(note5[7]),
.base_x(base_x),
.base_y(base_y),

.note_x(note_x[4]),
.note_y(note_y[4]),
.rest(rest[4]),
.upside_down(upside_down[4]),
.treble(treble[4]),
.sharp_out(sharp[4]) );
   defparam  npos5.LINE_HEIGHT = 1;
   defparam  npos5.LINE_SPACING = 5;
   defparam  npos5.STAFF_SPACING = 40;

   note_position npos6( .clk(clk),
.start(start),
.reset(reset),
.octave(note6[3:0]),
.note(note6[6:4]),
.sharp(note6[7]),
.base_x(base_x),
.base_y(base_y),

.note_x(note_x[5]),
.note_y(note_y[5]),
.rest(rest[5]),
.upside_down(upside_down[5]),
.treble(treble[5]),
.sharp_out(sharp[5]) );
   defparam  npos6.LINE_HEIGHT = 1;
   defparam  npos6.LINE_SPACING = 5;
   defparam  npos6.STAFF_SPACING = 40;

   note_position npos7( .clk(clk),
.start(start),
.reset(reset),
.octave(note7[3:0]),
.note(note7[6:4]),
.sharp(note7[7]),
```

```
.base_x(base_x),
.base_y(base_y),

.note_x(note_x[6]),
.note_y(note_y[6]),
.rest(rest[6]),
.upside_down(upside_down[6]),
.treble(treble[6]),
.sharp_out(sharp[6]) );
   defparam  npos7.LINE_HEIGHT = 1;
   defparam  npos7.LINE_SPACING = 5;
   defparam  npos7.STAFF_SPACING = 40;

   note_position npos8( .clk(clk),
.start(start),
.reset(reset),
.octave(note8[3:0]),
.note(note8[6:4]),
.sharp(note8[7]),
.base_x(base_x),
.base_y(base_y),

.note_x(note_x[7]),
.note_y(note_y[7]),
.rest(rest[7]),
.upside_down(upside_down[7]),
.treble(treble[7]),
.sharp_out(sharp[7]) );
   defparam  npos8.LINE_HEIGHT = 1;
   defparam  npos8.LINE_SPACING = 5;
   defparam  npos8.STAFF_SPACING = 40;

   ////////////////////////////////////////////////////
   // STAVES SPRITES
   ////////////////////////////////////////////////////
   wire [2:0]  staves_pixel;
   staves_sprite ssprite(.vclock(vclock),
 .hcount(hcount),
 .vcount(vcount),
```

```
 .pixel(staves_pixel));
   defparam  ssprite.WIDTH = SCREEN_WIDTH;
   defparam  ssprite.HEIGHT = SCREEN_HEIGHT;
   defparam  ssprite.START_X = 0;
   defparam  ssprite.START_Y = START_Y;
   defparam  ssprite.STAVES_SPACING = 140;

   ////////////////////////////////////////////////////
   // NOTE SPRITES
   ////////////////////////////////////////////////////
   wire [7:0]  note_sprite_pixel;
   wire [2:0]  note_flag_pixel;

   chord_flags_display cflags( .vclock(vclock),
       .hcount(hcount),
       .vcount(vcount),
       .x(note_x[0]),
       .note1_y(note_y[0]),
       .note2_y(note_y[1]),
       .note3_y(note_y[2]),
       .note4_y(note_y[3]),
       .note5_y(note_y[4]),
       .note6_y(note_y[5]),
       .note7_y(note_y[6]),
       .note8_y(note_y[7]),
       .duration(new_duration),
       .upside_down(upside_down),
       .treble(treble),
       .rests(rest),

       .flag_pixels(note_flag_pixel));

   note_sprite nsprite1(.vclock(vclock),
.hcount(hcount),
.vcount(vcount),
.note_x(note_x[0]),
.note_y(note_y[0]),
.rest(rest[0]),
.sharp(sharp[0]),
.duration(new_duration),
```

```
.pixel(note_sprite_pixel[0]));

   note_sprite nsprite2(.vclock(vclock),
.hcount(hcount),
.vcount(vcount),
.note_x(note_x[1]),
.note_y(note_y[1]),
.rest(rest[1]),
.sharp(sharp[1]),
.duration(new_duration),

.pixel(note_sprite_pixel[1]));

   note_sprite nsprite3(.vclock(vclock),
.hcount(hcount),
.vcount(vcount),
.note_x(note_x[2]),
.note_y(note_y[2]),
.rest(rest[2]),
.sharp(sharp[2]),
.duration(new_duration),

.pixel(note_sprite_pixel[2]));

   note_sprite nsprite4(.vclock(vclock),
.hcount(hcount),
.vcount(vcount),
.note_x(note_x[3]),
.note_y(note_y[3]),
.rest(rest[3]),
.sharp(sharp[3]),
.duration(new_duration),

.pixel(note_sprite_pixel[3]));

   note_sprite nsprite5(.vclock(vclock),
.hcount(hcount),
.vcount(vcount),
.note_x(note_x[4]),
```

```
.note_y(note_y[4]),
.rest(rest[4]),
.sharp(sharp[4]),
.duration(new_duration),

.pixel(note_sprite_pixel[4]));

   note_sprite nsprite6(.vclock(vclock),
.hcount(hcount),
.vcount(vcount),
.note_x(note_x[5]),
.note_y(note_y[5]),
.rest(rest[5]),
.sharp(sharp[5]),
.duration(new_duration),

.pixel(note_sprite_pixel[5]));

   note_sprite nsprite7(.vclock(vclock),
.hcount(hcount),
.vcount(vcount),
.note_x(note_x[6]),
.note_y(note_y[6]),
.rest(rest[6]),
.sharp(sharp[6]),
.duration(new_duration),

.pixel(note_sprite_pixel[6]));

   note_sprite nsprite8(.vclock(vclock),
.hcount(hcount),
.vcount(vcount),
.note_x(note_x[7]),
.note_y(note_y[7]),
.rest(rest[7]),
.sharp(sharp[7]),
.duration(new_duration),

.pixel(note_sprite_pixel[7]));
```

```
   wire [2:0]  ledger_pixel1, ledger_pixel2,
ledger_pixel3, ledger_pixel4;
   wire [2:0]  ledger_pixel5, ledger_pixel6,
ledger_pixel7, ledger_pixel8;
   wire [2:0]  ledger_pixels = ledger_pixel1 |
ledger_pixel2 | ledger_pixel3 |
ledger_pixel4 | ledger_pixel5 |
ledger_pixel6 | ledger_pixel7 |
ledger_pixel8;
   ledger_line ledg1( vclock, hcount, vcount, note_x[0],
      base_y, note1, ledger_pixel1);
   ledger_line ledg2( vclock, hcount, vcount, note_x[1],
      base_y, note2, ledger_pixel2);
   ledger_line ledg3( vclock, hcount, vcount, note_x[2],
      base_y, note3, ledger_pixel3);
   ledger_line ledg4( vclock, hcount, vcount, note_x[3],
      base_y, note4, ledger_pixel4);
   ledger_line ledg5( vclock, hcount, vcount, note_x[4],
      base_y, note5, ledger_pixel5);
   ledger_line ledg6( vclock, hcount, vcount, note_x[5],
      base_y, note6, ledger_pixel6);
   ledger_line ledg7( vclock, hcount, vcount, note_x[6],
      base_y, note7, ledger_pixel7);
   ledger_line ledg8( vclock, hcount, vcount, note_x[7],
      base_y, note8, ledger_pixel8);

   wire [9:0]  treble_rest_y = base_y + 7;
   wire [9:0]  bass_rest_y = base_y + 7 + 65;
   wire [2:0]  treble_rest_pixel;
   wire [2:0]  bass_rest_pixel;
   rest_display treble_rest(vclock, hcount, vcount,
                         base_x, treble_rest_y,
                         new_duration, treble_rest_pixel);
   rest_display bass_rest(vclock, hcount, vcount,
                       base_x, bass_rest_y,
                       new_duration, bass_rest_pixel);

   reg [2:0]  new_pixel;
   reg [10:0]  shcount;
   reg [9:0]  svcount;
```

```
reg  shsync, svsync, sblank;

wire  treble_all_rests = ((rest & treble) == treble);
wire  bass_all_rests = ((rest & ~treble) == ~treble);

wire [2:0]  ns_pixel = note_flag_pixel |
             ( treble_all_rests ? treble_rest_pixel : 0 ) |
             ( bass_all_rests ? bass_rest_pixel : 0 ) |
             {3{|(note_sprite_pixel)}} | ledger_pixels;

always @ (posedge vclock) begin
   shcount <= hcount;
   svcount <= vcount;
   shsync <= hsync;
   svsync <= vsync;
   sblank <= blank;
   new_pixel <= ns_pixel;
end

////////////////////////////////////////////////////
// FRAME BUFFER
////////////////////////////////////////////////////
wire [2:0] notes_pixel;
wire       bhsync,bvsync,bblank;
frame_buffer buffer(.vclock(vclock),
    .new_pixel(new_pixel),
    .reset(refresh),
    .use_new(raw),
    .enable(buffer_we),
    .hcount(shcount),
    .vcount(svcount),
    .hsync(shsync),
    .vsync(svsync),
    .blank(sblank),

    .bhsync(bhsync),
    .bvsync(bvsync),
    .bblank(bblank),
    .pixel(notes_pixel));
```

```
   /////////////////////////////////////////////////////
   // ASSIGN VGA SIGNALS
   /////////////////////////////////////////////////////
   wire [2:0] pixel = notes_pixel ? NOTE_COLOR : staves_pixel;
   reg [2:0]  rgb;
   reg        b,hs,vs;
   always @(posedge vclock) begin
      hs <= bhsync;
      vs <= bvsync;
      b <= bblank;
      rgb <= pixel;
   end

   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;
   assign vga_out_blank_b = ~b;
   assign vga_out_pixel_clock = ~vclock;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;
endmodule
```