Abstract

We aim to create an interactive music game featuring real instruments. The game contains a number of songs that a user can play along with. The game mechanic is styled after Guitar Hero: while a song is playing its notes will move across the screen and the player must play the notes at the correct time. However, unlike Guitar Hero, which confines itself to five “notes” and a plastic controller, we aspire to let users play along with real instruments instead. Gamers are able to both engage with their music and gain actual musical skill.

By taking a FFT (Fast Fourier Transform) of a live instrument, the game identifies the pitch being played and scores the note on its pitch and timing accuracy. The game lets users play along with almost any monophonic instrument.
Table of Contents

ABSTRACT .................................................................................................................. 1
INTRODUCTION ......................................................................................................... 3
OVERVIEW .................................................................................................................. 3
MODULES ..................................................................................................................... 4
  NOTE IDENTIFICATION – ANDRÉS ROMERO .......................................................... 4
     FFT Audio ............................................................................................................... 4
     FFT Xilinx IP Core ............................................................................................... 5
     Process Audio....................................................................................................... 5
     Addressing Logic ................................................................................................. 6
     Lowest valid note from RAM ............................................................................. 6
GAME LOGIC – ANDRÉS ROMERO ............................................................................ 6
  Menu FSM ................................................................................................................ 6
  Score Updater ......................................................................................................... 7
  ASCII Counter ....................................................................................................... 7
  ASCII Converter ................................................................................................... 7
  High Score Tracking ............................................................................................. 8
MUSICAL SCORE LOADER – PAUL HEMBERGER ...................................................... 8
  Song Encoding ...................................................................................................... 8
  Note Loading ........................................................................................................ 8
  Creating New Songs .............................................................................................. 8
GAME VIDEO DISPLAY – PAUL HEMBERGER ............................................................. 9
  XVGA ....................................................................................................................... 9
  Notes ....................................................................................................................... 10
  Musical Staff .......................................................................................................... 10
  Game Scores & Pitch ............................................................................................. 11
  Visual Feedback & Effects ..................................................................................... 11
MENU VIDEO DISPLAY – PAUL HEMBERGER ........................................................... 11
  Menu Items ........................................................................................................... 12
USB-FIFO & ZBT ACCESS – PAUL HEMBERGER ....................................................... 13

CHALLENGES ............................................................................................................. 14

CONCLUSION ............................................................................................................. 14

APPENDIX A – NOTE CONVERSION TABLES & ENCODING ................................. 15
  Pitch & Frequency Lookup Table ......................................................................... 15
  Pitch Hexadecimal Encoding ............................................................................... 16

APPENDIX B – VERILOG ........................................................................................... 17

APPENDIX C – IMAGES ............................................................................................. 68
Introduction

The player may use a simple instrument, such as a recorder, to play notes in a rhythm based music game. The game loads desired notes from memory and displays them on the screen. Those notes will stream right to left on the screen. Once a note reaches the left edge of the screen it will have a small denoted active region indicating that the player should play that note. The game shows how successful the player was in correctly timing the note by increasing their score proportionally to the time the pitch was correct. The current pitch is also shown on a scale on the top-right corner of the screen to help the player learn their instrument.

Overview

Recorder Hero is composed of four core components: note identification, musical score loading, game logic and video display logic.

![Figure 1: High-level overview of Recorder Hero's components](image-url)
Modules

Note Identification – Andrés Romero

The Note Identification section processes all of the microphone input and sends a hex representation of the current note played to the game logic. It starts by digitizing the input with an AC97 and then runs a Fast Fourier Transform to convert the waves into the frequency domain whose output is stored in RAM. Then a separate set of logic reads the RAM values and determines the lowest frequency bin that went over a predetermined threshold and continuously outputs the representation of that note.

FFT Audio

This module handles timing and setup of the on-board AC97, and was not changed from the one provided by the 6.111 staff. The AC97 is a standard analog to digital converter that can take input from a microphone and output a sixteen-bit per clock cycle digital representation. As the labkit AC97 has a clock of 48kHz, it can digitally recreate sounds from 0Hz to 24kHz. Since the highest notes a standard instrument can output generally don’t go above 8kHz, this range was not problematic. The outputs used by Recorder Hero are the high order eight bits of digitized information coming from the AC97. High order bits were used because low order bits are bound to contain more noise and they were not necessary to correctly identify note pitches.
FFT Xilinx IP Core

Designing the Fast Fourier Transform was a tradeoff between speed and accuracy. The number of points in the FFT determines both the delay between sample refreshes and the amount of Hz per output bin.

\[ P = \text{number of points, or output bins for a given FFT} \]

\[ \text{SampleRefresh} = \left( \frac{1}{48\text{kHz}} \right) \times P \]

\[ \text{BinAccuracy} = \frac{48\text{kHz}}{P} \]

Because of the exponential nature of note frequencies, the FFT must have more points to differentiate notes that are closer in frequency. This can be seen clearly by the difference between C3 & C#3 and C4 & C#4, which are 8 and 16 Hz apart respectively. Yet, more points increase sample refresh for a particular bin and therefore increase the delay between playing a note and the game registering the note as played. Reducing this delay is a very important factor to producing a fun game, while increasing the number of lower octaves allows more instruments to be used to play the game.

<table>
<thead>
<tr>
<th>Note</th>
<th>Frequency (Hz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3</td>
<td>130.81</td>
</tr>
<tr>
<td>C#3</td>
<td>138.59</td>
</tr>
<tr>
<td>C4</td>
<td>261.63</td>
</tr>
<tr>
<td>C#4</td>
<td>277.18</td>
</tr>
</tbody>
</table>

The FFT was originally 16384 points, but the delay was a third of a second which proved to be too long for the game. The FFT was then reduced to 4096 points. The bin accuracy lowered to 11 Hz per bin and the previously lowest octave, which originated at C3, was dropped.

Process Audio

The process audio module originated from an example FFT project. It contains the instantiation of the Fast Fourier Transform hardware, a small square root module and an output filter. The FFT sends a new data point every 48kHz, which contains the bin number corresponding to the data point as well as its imaginary and real magnitudes. As there is a very limited range of frequencies given by the 60 notes that are sensed by
Recorder Hero, not all the data points are used. Therefore the output filter was modified to only output data points that corresponded to the sixty notes the project required, but otherwise the module was left unchanged.

The bins to pass through were determined by the following mapping function:

\[ \text{note frequency} \times \frac{48kHz}{4096} \]

The 12-bit representation of the value in binary is then the bin number that corresponds to that note. If two notes return the same bin number then the FFT does not have enough accuracy to distinguish them.

**Addressing Logic**

New samples come from the FFT at a slow rate, so an immediate set of logic to determine the current note is not possible. A RAM is used as an interface in order to store the latest value and use it within the next iteration of the logic. Consequently the bin numbers and magnitudes have to be translated to a RAM address and a value representation. This is the job of the addressing logic. It works very simply translating each of the 60 bin numbers linearly into the numbers 0-59, and it comparing the magnitude with a threshold value to determine if the specific note was played and stores a single bit in the ram respectively.

**Lowest valid note from RAM**

In a single pitch scenario, such as when whistling into a microphone, the easiest way to determine the pitch of the note is to take the frequency of the bin with maximum magnitude. However due to overtones in various musical instruments, coupled with high frequency bias in some microphones, a max magnitude algorithm is actually not the best way to determine the current note played. Instead, it is most accurate to take the lowest bin that goes over a specific defined threshold value—most often the fundamental frequency of the input. This set of logic goes through the RAM values sequentially until it finds one that has gone over the threshold as defined by a logical one, and then outputs that value until the FFT produces a new data point and it resets.

**Game Logic – Andrés Romero**

Equipped with the ideal notes that should be played as well as the current note from the microphone, the game logic determines the score for the playthrough as well as the high scores for each of the four songs. It also handles menu navigation and song entry.

**Menu FSM**

Once the bitfile is loaded the user is presented with a main menu that has four song choices. This menu can be represented by the following state diagram.
Once enter is pressed and the user enters the InGame state, the chosen song and a reset signal is sent to the musical score loader. The state is also updated so that the display knows it is time to start showing the game screen.

**Score Updater**

As the game starts, the score loader sends the ideal note to this submodule, which compares the current note to the ideal note and updates the score accordingly. The score is updated linearly, and so hitting the correct note for 0.002 seconds gives the player 1 point. This also outputs a “hit” signal to notify the display logic if the correct note was played.

**ASCII Counter**

The **Score Updater** counts in binary, which is necessary for keeping track of high scores throughout playthrough. However, the display logic displays the score as 8-bit ASCII characters. This module counts the score up to 6 digits in ASCII and outputs the current ASCII representation of the score for use in the display and high score modules.

**ASCII Converter**

Similarly, the **Score Updater** outputs the current note played as a hex representation of the note, but the note must be displayed in ASCII. This is a small lookup table that maps the hex representation of the notes to their corresponding 2-byte ASCII characters.
High Score Tracking
When a song ends and the done signal is driven by the Score Loader, this module takes in the current binary and ASCII score representations for use. If the current binary representation of the score is larger than the binary representation of the high score for that song, the ASCII and binary representations of the high score get replaced with the new, larger values. Therefore, as long as the FPGA is not turned off, this module will continuously keep track of the highest score for each individual song and output the ASCII representations for the display module to use.

Musical Score Loader – Paul Hemberger
The musical score module reads song information from Block ROM (BROM) and gives upcoming note information to the game logic and video output.

Song Encoding
Songs are hardcoded into 4x256 BROMs. Each BROM contains a series of hexadecimal values, each value specifying a pitch for an eighth note. The tempos for each song are stored in a case statement as 25-bit values. The pitches are understood by the table in Appendix A.

To create longer duration notes, a particular pitch is repeated several times in the BROM. A zero-value note is considered a rest, and a special 0xF value is used at the end to indicate the end of the song.

Note Loading
Once a song is selected, as controlled by the Menu FSM, the musical score module outputs the next sixteen notes in the song to the game logic and display. It initializes the song with sixteen rests so the player had time to prepare for the first real note of the song.

Then, on each beat the musical_score_loader grabs the next 4 bits (one eighth note) from the song’s BROM and shifts the notes over by one. The beat is created using a counter and the tempo value of the chosen song. The counter continually counts up to the tempo value to produce the beat signal.

Creating New Songs
New songs were created by writing up their notes as a straight sequence of pitches and durations. A Python script would then convert the sequence into a COE file that could be loaded into a BROM. The tempo was hardcoded in the musical_score_loader.v file.
Game Video Display – Paul Hemberger
The video display module displays the game and menus. Because of the quantity of information on the screen the VGA output was set to a resolution of 1024x768. The labkit’s onboard ADV7125 Triple 8-bit Video DAC handled the VGA signal output.

XVGA
The lab3 module supplied by the staff handled direct interactions with the ADV7125.

Figure 4: Diagram showing how a sequence of notes and durations was converted into a song in BROM

Figure 5: In-game screenshot of Recorder Hero
Notes
The **musical_score_loader** provides the video display with a sequence of the next sixteen notes. These notes are rendered on the screen horizontally in time and vertically in pitch. Each note is created with the rectangular **blob** module, provided by the **staff**. Each pitch (ex. C, D, E...) is assigned a y-axis value. Each note is 64px wide. The notes move horizontally at a rate that after one tempo interval they would have moved 64px leftward. As the 1\textsuperscript{st} note begins to move off the screen, the 16\textsuperscript{th} note starts to move in.

This movement was timed by creating two **counter** modules. The first counter, **clock_tempo** would output high each time one beat had passed. On each beat, the notes would assume the pitch value of the next note in the sequence and move **back** 64px to its original position. The second counter, **tempo_move** counted to \(\frac{\text{tempo}}{64}\). Each time **tempo_move** went high the notes would move 1px to the left.

By timing the movement as such the notes and song would appear to continuously move across the screen even though the blobs were simply moving across the same 64px stretch over and over.

![Diagram of the note blobs' movement](image)

**Musical Staff**
The notes, “BAGFEDC” are displayed in descending order on the side of the screen. Staff lines were drawn across the screen to help visually separate each note. The staff lines were also created with the **blob** module.

Unfortunately the **cstringdisp** module provided could only produce characters that were 16x24px. To make the characters larger (and look nicer) we instead created a 72x512px bitmap image containing the characters in a large, crisp font. Using the provided Matlab script the image was processed into a COE file that was fed into an 8x36864 BROM. Because the image was black and white, no bitmap color tables were necessary.
Game Scores & Pitch
The player’s current score, high score and currently detected pitch are displayed on screen using the cstringdisp module provided by the staff. The Score Module provides the scores, and the audio logic provides the current pitch.

Visual Feedback & Effects
Several visual effects were added in to help create a more vibrant and intuitive game.

If the player hits the correct note the note highlights to a shade of yellow. If the player isn’t playing the correct note, it will be highlighted in red. The Score Module tracks whether or not the user is playing the correct note and this output was used to change the colors.

To further aid the player, whichever note they are playing is highlighted along the musical staff text. The whole notes (C, D, E…) were highlighted in yellow, and sharps (C#, D#, F#…) were highlighted in purple. This was done by creating a blob on top of the letters and alphablending it with the musical staff image ROM.

At the bottom of the screen is a legend with blocks of each color and a text explanation of what each color represents.

Menu Video Display – Paul Hemberger
When the Menu FSM indicates that the game was waiting at the menu, the display logic
outputs a background image and the titles of songs that a user could play.

Background Image
The background was a 1024x768px bitmap image. The original image was 8-bit, meaning that it had been decomposed into the optimal 256 colors to represent the image and each pixel of the image was assigned 8-bits to lookup a color. Its three bitmap color tables for Red, Green and Blue (RGB) and the pixel-color values were generated using the provided Matlab script.

The RGB intensities were each stored in 8x256 BROMs as color lookup tables.

Since each pixel had 1-byte of data, the image required a fair bit of memory:

\[
\text{Image Memory Required: } (1024 \times 768 \text{ pixels}) \times \left( \frac{1 \text{ byte}}{\text{pixel}} \right) = 750.422 \text{ Kilobytes}
\]

\[
\text{BRAM Onboard: } \left(18 \frac{\text{Kbit}}{\text{block}}\right) \times (144 \text{ blocks}) = 324 \text{ kilobytes}
\]

\[
\text{ZBT Onboard (1 of 2 blocks): } \left(36 \frac{\text{bits}}{\text{row}}\right) \times (512,000 \text{ rows}) = 2.197 \text{ Megabytes}
\]

Unfortunately the image was much larger than the available BRAM, so Zero Bus Turnaround (ZBT) memory was used instead. Two of the downsides of ZBT versus BRAM included a two-cycle read delay and the inability to load the memory with values on startup. To write the image to ZBT, a USB-Serial FIFO was used as is detailed in USB-FIFO & ZBT Access.

Once the ZBT memory was accessible, we connected a computer to the FIFO to load the image into the game. The vram_display and zbt_6111 modules provided by the staff were both used to render the image. The vram_display module was modified to match how the order of our pixel information stored in ZBT memory.

Menu Items
The song titles are displayed with the cstringdisp module. A rectangular box, made with the blob module was placed next to the currently selected song based on the 3-bit menu state to help the user select a song.

The currently detected note string on the main menu was also made with the cstringdisp module, and is actually exactly the same as the one shown on the game screen.
USB-FIFO & ZBT Access – Paul Hemberger

In order to display the background image, the image contents had to be fed into the ZBT memory each time the board was reset. Transferring large amounts of data was best achieved through an FTDI UM245M USB-to-FIFO component wired to the labkit breadboard and the `usb_transfer` module provided by the staff.

The FIFO was powered by the 5V rail, took input from a USB connection wired to a computer and had its 8 output pins (1-byte of data) wired to the `user1` I/O pins of the labkit.

![Figure 8: The FTDI USB-to-FIFO component wired to the labkit](image)

A Python script running on the computer would send serial byte data to the FIFO. When the FIFO had a new byte to send to the labkit, it would assert pin 13, `RXF#` low and the `usb_transfer` module would read the data off the `user1` pins.

Because the ZBT rows were 36-bits wide, four 8-bit pixels were written into each row. However the FIFO only provided one 8-bit pixel at a time, so they had to be moved into the `zbt_image_writer` module. The `zbt_image_writer` would act as a shift register and store up to 4 8-bit pixels at once. Once it had four pixels, it would assert high and the row would be written into the ZBT memory.
Challenges
Unfortunately there were few bugs that remain unresolved. We implemented some “hack” solutions to cover up their symptoms, but never discovered their causes.

The background image from the ZBT memory showed some residual aliasing and artifacts. When the image was shown, a small sliver on the left side of the image would be the vertical column of a different part of the image. It also appeared that the pixels weren’t displayed in quite the correct order, creating some aliased edges.

On the audio processing side, the Note Identification module continually output the wrong note value—off by exactly one pitch every time regardless of the frequency distance. Despite it being rewritten twice and countless hours searching we were never able to resolve this issue.

Outside of our design, we also ran into many challenges with ISE and its project management. Several times when we tried to integrate our separate modules together, ISE would complain that we were using > 100% of the FPGA, even though all of our modules combine use only about 40%. It would also stall on Synthesis on occasion when it would try to optimize an already compiled-and-optimized version of the FFT.

Conclusion
We believe that this project turned out quite well. It was of reasonable scale for two people and touched on many different components: analog / audio input, Fourier analysis, USB-FIFO access, ZBT & BRAM storage, VGA output and lots of logic design.

We are very happy with our final product and demo. We met all of the initial goals we set out for ourselves and were able to finish a number of optional components; we were able to jam along to a few tunes with several different instruments (recorder, kalimba and piano) successfully and we received a number of compliments about the menu and game visuals.

Of course, there are many more features and polish we wish we had time for: storing the background image in CompactFlash, fixing the FFT and image bugs, multiple notes, more songs, MIDI input… one can dream.

We both had great fun working on Recorder Hero and we got a lot out of the project. We would like to thank Gim Hom, Dylan Sherry, Kevin Zheng, Joe Colosimo and Jessie Stickgold-Sarah for their tireless aid and feedback—12-hour days in lab wouldn’t have been the same without you.
Appendix A – Note Conversion Tables & Encoding

Pitch & Frequency Lookup Table

<table>
<thead>
<tr>
<th>Note &amp; Octave</th>
<th>Frequency (Hz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3</td>
<td>130.81</td>
</tr>
<tr>
<td>C#3/Db3</td>
<td>138.59</td>
</tr>
<tr>
<td>D3</td>
<td>146.83</td>
</tr>
<tr>
<td>D#3/Eb3</td>
<td>155.56</td>
</tr>
<tr>
<td>E3</td>
<td>164.81</td>
</tr>
<tr>
<td>F3</td>
<td>174.61</td>
</tr>
<tr>
<td>F#3/Gb3</td>
<td>185.00</td>
</tr>
<tr>
<td>G3</td>
<td>196.00</td>
</tr>
<tr>
<td>G#3/Ab3</td>
<td>207.65</td>
</tr>
<tr>
<td>A3</td>
<td>220.00</td>
</tr>
<tr>
<td>A#3/Bb3</td>
<td>233.08</td>
</tr>
<tr>
<td>B3</td>
<td>246.94</td>
</tr>
<tr>
<td>C4</td>
<td>261.63</td>
</tr>
<tr>
<td>C#4/Db4</td>
<td>277.18</td>
</tr>
<tr>
<td>D4</td>
<td>293.66</td>
</tr>
<tr>
<td>D#4/Eb4</td>
<td>311.13</td>
</tr>
<tr>
<td>E4</td>
<td>329.63</td>
</tr>
<tr>
<td>F4</td>
<td>349.23</td>
</tr>
<tr>
<td>F#4/Gb4</td>
<td>369.99</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Frequency (Hz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>G4</td>
<td>392.00</td>
</tr>
<tr>
<td>G#4/Ab4</td>
<td>415.30</td>
</tr>
<tr>
<td>A4</td>
<td>440.00</td>
</tr>
<tr>
<td>A#4/Bb4</td>
<td>466.16</td>
</tr>
<tr>
<td>B4</td>
<td>493.88</td>
</tr>
<tr>
<td>C5</td>
<td>523.25</td>
</tr>
<tr>
<td>C#5/Db5</td>
<td>554.37</td>
</tr>
<tr>
<td>D5</td>
<td>587.33</td>
</tr>
<tr>
<td>D#5/Eb5</td>
<td>622.25</td>
</tr>
<tr>
<td>E5</td>
<td>659.26</td>
</tr>
<tr>
<td>F5</td>
<td>698.46</td>
</tr>
<tr>
<td>F#5/Gb5</td>
<td>739.99</td>
</tr>
<tr>
<td>G5</td>
<td>783.99</td>
</tr>
<tr>
<td>G#5/Ab5</td>
<td>830.61</td>
</tr>
<tr>
<td>A5</td>
<td>880.00</td>
</tr>
<tr>
<td>A#5/Bb5</td>
<td>932.33</td>
</tr>
<tr>
<td>B5</td>
<td>987.77</td>
</tr>
<tr>
<td>C6</td>
<td>1046.50</td>
</tr>
<tr>
<td>C#6/Db6</td>
<td>1108.73</td>
</tr>
<tr>
<td>D6</td>
<td>1174.66</td>
</tr>
<tr>
<td>D#6/Eb6</td>
<td>1244.51</td>
</tr>
<tr>
<td>E6</td>
<td>1318.51</td>
</tr>
<tr>
<td>Pitch</td>
<td>Hexadecimal Encoding</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------</td>
</tr>
<tr>
<td>F6</td>
<td>1396.91</td>
</tr>
<tr>
<td>F#6/Gb6</td>
<td>1479.98</td>
</tr>
<tr>
<td>G6</td>
<td>1567.98</td>
</tr>
<tr>
<td>G#6/Ab6</td>
<td>1661.22</td>
</tr>
<tr>
<td>A6</td>
<td>1760.00</td>
</tr>
<tr>
<td>A#6/Bb6</td>
<td>1864.66</td>
</tr>
<tr>
<td>B6</td>
<td>1975.53</td>
</tr>
<tr>
<td>C7</td>
<td>2093.00</td>
</tr>
<tr>
<td>C#7/Db7</td>
<td>2217.46</td>
</tr>
<tr>
<td>D7</td>
<td>2349.32</td>
</tr>
<tr>
<td>D#7/Eb7</td>
<td>2489.02</td>
</tr>
<tr>
<td>E7</td>
<td>2637.02</td>
</tr>
<tr>
<td>F7</td>
<td>2793.83</td>
</tr>
<tr>
<td>F#7/Gb7</td>
<td>2959.96</td>
</tr>
<tr>
<td>G7</td>
<td>3135.96</td>
</tr>
<tr>
<td>G#7/Ab7</td>
<td>3322.44</td>
</tr>
<tr>
<td>A7</td>
<td>3520.00</td>
</tr>
<tr>
<td>A#7/Bb7</td>
<td>3729.31</td>
</tr>
<tr>
<td>B7</td>
<td>3951.07</td>
</tr>
<tr>
<td>C8</td>
<td>4186.01</td>
</tr>
<tr>
<td>C#8/Db8</td>
<td>4434.92</td>
</tr>
<tr>
<td>D8</td>
<td>4698.64</td>
</tr>
<tr>
<td>D#8/Eb8</td>
<td>4978.03</td>
</tr>
</tbody>
</table>

Pitch Hexadecimal Encoding

<table>
<thead>
<tr>
<th>Pitch</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>REST</td>
<td>0000</td>
</tr>
<tr>
<td>C</td>
<td>0001</td>
</tr>
<tr>
<td>C#</td>
<td>0010</td>
</tr>
<tr>
<td>D</td>
<td>0011</td>
</tr>
<tr>
<td>D#</td>
<td>0100</td>
</tr>
<tr>
<td>E</td>
<td>0101</td>
</tr>
<tr>
<td>F</td>
<td>0110</td>
</tr>
<tr>
<td>F#</td>
<td>0111</td>
</tr>
<tr>
<td>G</td>
<td>1000</td>
</tr>
<tr>
<td>G#</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>1010</td>
</tr>
<tr>
<td>A#</td>
<td>1011</td>
</tr>
<tr>
<td>B</td>
<td>1100</td>
</tr>
<tr>
<td>C high</td>
<td>1101</td>
</tr>
<tr>
<td>D high</td>
<td>1110</td>
</tr>
<tr>
<td>End-of-file (EOF)</td>
<td>1111</td>
</tr>
</tbody>
</table>
Appendix B – Verilog

/////////////////////////////////////////////////////////////////////
// Recorder Hero - made by Paul Hemberger and Andres Romero
//
// Fall 2011 6.111 Final Project
/////////////////////////////////////////////////////////////////////
// 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 fft (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_yccrb, 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_yccrb, 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;

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

wire  reset;
SRL16  reset_sr(.D(1'b0), .CLK(clock_27mhz), .Q(reset),
                     .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam  reset_sr.INIT = 16'hFFFF;

// Audio Input and Output
assign  beep= 1'b0;
//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

// VGA Output
//assign  vga_out_red = 10'h0;
//assign  vga_out_green = 10'h0;
//assign  vga_out_blue = 10'h0;
//assign  vga_out_sync_b = 1'b1;
//assign  vga_out_blank_b = 1'b1;
//assign  vga_out_pixel_clock = 1'b0;
//assign  vga_out_hsync = 1'b0;
//assign  vga_out_vsync = 1'b0;

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

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

// SRAMs
/*
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_clk = 1'b0;
assign ram0_we_b = 1'b1;
assign ram0_cen_b = 1'b0; // clock enable
*/

/* enable RAM pins */
assign ram0_ce_b = 1'b0;
assign ram0_oe_b = 1'b0;
assign ram0_adv_ld = 1'b0;
assign ram0_bwe_b = 4'h0;

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

// use FPGA's digital clock manager to produce a
// 65MHz clock (actually 64.8MHz)
wire clock_65mhz_unbuf,clock_65mhz;
DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk1 is 10
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 37
BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));
// use FPGA's digital clock manager to produce a
// 65MHz clock (actually 64.8MHz)
wire clk;

// ZBT Memory
wire locked;
//assign clock_feedback_out = 0; // gph 2011-Nov-10
ramclock rc(.ref_clock(clock_65mhz),
            .fpga_clock(clk),
            .ram0_clock(ram0_clk),
            //.ram1_clock(ram1_clk),  //uncomment if ram1 is
used
            .clock_feedback_in(clock_feedback_in),
            .clock_feedback_out(clock_feedback_out),
            .locked(locked));

wire up;
wire down;
wire enter;
wire resetButton;

//Set up signals from all the buttons used.
debounce bup(reset, clock_27mhz, ~button_up, up);
debounce bdown(reset, clock_27mhz, ~button_down, down);
debounce benter(reset, clock_27mhz, ~button_enter, enter);
debounce breset(reset, clock_27mhz, ~button0, resetButton);

wire boardReset;
wire menuReset;

assign boardReset = (reset | resetButton);

wire songDone;
wire [2:0] menuState;
wire menuResetOut;
wire [1:0] song;

wire[17:0] scoreBinary;
wire[47:0] asciiScore;
wire[47:0] highScore;

//Menu FSM instantiation, deals with menu states and keeping correct high scores
menuFSM menu(up,down,enter,boardReset,songDone,clock_65mhz,scoreBinary,
              asciiScore,menuState,menuResetOut,song,highScore);

assign menuReset = (menuResetOut | boardReset);

wire [15:0] from_ac97_data, to_ac97_data;
wire ready;

// AC97 driver
fft_audio a(clock_27mhz, reset, volume, from_ac97_data, to_ac97_data, ready,
audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch, ac97_bit_clock);

// Loopback incoming audio to headphones
assign to_ac97_data = from_ac97_data;

wire [3:0] currentNote;
wire readVal;
wire [5:0] GuessAddr;

wire [15:0] analyzer1_out;
wire [15:0] analyzer2_out;

// Note identification instantiation, takes in ac97 information and outputs the currentNote played.

noteIdentification a1(reset,clock_27mhz,ready,switch,from_ac97_data,currentNote,GuessAddr,readVal,analyzer1_out,analyzer2_out);

assign analyzer3_data = analyzer2_out;
assign analyzer1_data = analyzer1_out;

wire[3:0] notePlayed;
wire score;
wire [63:0] nn;

// Takes in a hex representation of a note from noteIdentification and a hex representation from the musical score loader and updates the score and if note was played correctly.

scoreUpdater updater(clock_65mhz,currentNote,nn[3:0],menuReset,hit, score,notePlayed,scoreBinary);

//Counts up in ASCII to our maximum score value

binaryCounterASCIIf counter(clock_65mhz,menuReset,score,asciiScore);

wire[15:0] asciiNote;

// Turns the hex representation of a note to a 2 byte ascii representation to be shown

hexToAscii hexy(notePlayed,clock_65mhz,asciiNote);

// Generate basic XVGA video signals

wire [10:0] hcount;
wire [9:0] vcount;
wire hsync,vsync,blank;
xvga xvga1(.vclock(clk),.hcount(hcount),.vcount(vcount),
    .hsync(hsync),.vsync(vsync),.blank(blank));
// feed XVGA signals to user's pong game
wire [23:0] pixel;
wire [63:0] dispdata;
wire phsync,pvsync,pblank;

// Connections to rh_video_display
wire debug_rh_display = 0;
reg right_note = 1;
reg [31:0] count = 0;
reg [3:0] notes[15:0];
wire [25:0] tempo;

/*** Wiring up USB FIFO to talk to user1 pins ***/
wire [7:0] fifo_data_input;
wire [7:0] fifo_data_output;
wire [3:0] fifo_state;
wire rd, rxf, fifo_newout, fifo_hold;
reg [7:0] from_fifo;

assign fifo_data_input = user1[9:2];
assign user1[1] = rd;
assign rxf = user1[0];

usb_input usb_input_module(.clk(clk),
                           .reset(resetButton),
                           .data(fifo_data_input),
                           .rd(rd),
                           .rxf(rxf),
                           .out(fifo_data_output),
                           .newout(fifo_newout),
                           .hold(1'b0),
                           .state(fifo_state));

always @(posedge clk) begin
  if (fifo_newout)
    from_fifo <= fifo_data_output;
end

/*** Wiring up ZBT RAM ***/
wire [35:0] vram_write_data;
wire [35:0] vram_read_data;
wire [18:0] vram_addr;
wire vram_we;

wire ram0_clk_not_used;
zbt_6111 zbt1(clk, 1'b1,
              vram_we,
              vram_addr,
              vram_write_data,
              vram_read_data,
              ram0_clk_not_used, //to get good timing, don't connect ram_clk to
                           
                           ram0_we_b, ram0_address, ram0_data, ram0_cen_b);

/*** Shift-register module to store 4 pixels in each row ***/
wire zbt_iw_newout;
wire [35:0] zbt_iw_output;
reg [35:0] zbt_iw_output_reg; // Required for write to be successful
zbt_image_writer ziw1(.clk(clk),
.reset(resetButton),
.image_data(fifo_data_output),
.new_input(fifo_newout),
.new_output(zbt_iw_newout),
.image_data_zbt(zbt_iw_output));

wire [7:0] image_bits;
reg [18:0] vram_write_addr = 0;
wire [18:0] vram_read_addr;

assign vram_we = zbt_iw_newout;
assign vram_addr = (zbt_iw_newout == 1) ? vram_write_addr : vram_read_addr;
assign vram_write_data = zbt_iw_output_reg;

/***
  * Grab background image pixels from vram_display
  ***/
vrRam_display vd1(.reset(resetButton),
.clk(clk),
.hcount(hcount),
.vcount(vcount),
.vr_pixel(image_bits),
.vram_addr(vram_read_addr),
.vram_read_data(vram_read_data));

reg [9:0] x_pixels = 0;
reg [9:0] y_pixels = 0;

/***
  * Control when to write to ZBT based on zbt_image_writer's ready signal
  ***/
always @ (posedge clk) begin
  if (reset) begin
    vram_write_addr <= 0;
    x_pixels <= 0;
    y_pixels <= 0;
  end
  if (zbt_iw_newout) begin
    vram_write_addr <= {0, y_pixels, x_pixels[9:2]-1'b1};
    zbt_iw_output_reg <= zbt_iw_output;
  end
  if (fifo_newout) begin
    x_pixels <= x_pixels + 1;
  end
  if (x_pixels >= 1023) begin
    x_pixels <= 0;
    y_pixels <= y_pixels + 1;
  end
end

musical_score_loader msl(.clk(clk), .reset(menuReset),
.song_id(song), .next_notes_out(nn),
.tempo_out(tempo),
.song_done(songDone));
// switch[1:0] selects which video generator to use:
// 00: user's pong game
// 01: 1 pixel outline of active video area (adjust screen controls)
// 10: color bars
reg [23:0] rgb;
wire border = (hcount==0 | hcount==1023 | vcount==0 | vcount==767);

reg b,hs,vs;
always @ (posedge clk) begin
    hs <= phsync;
    vs <= pvsync;
    b <= pblank;
    rgb <= pixel;
end

// VGA Output. In order to meet the setup and hold times of the
// AD7125, we send it ~clock_65mhz.
assign vga_out_red = rgb[23:16];
assign vga_out_green = rgb[15:8];
assign vga_out_blue = rgb[7:0];
assign vga_out_sync_b = 1'b1; // not used
assign vga_out_blank_b = ~b;
assign vga_out_pixel_clock = ~clk;
assign vga_out_hsync = hs;
assign vga_out_vsync = vs;

display_16hex hex(.reset(menuReset), .clock_27mhz(clock_27mhz),
    .data({1'b0,menuState,notePlayed,asciiScore,4'd0,currentNote}),
    .disp_blank(disp_blank), .disp_clock(disp_clock),
    .disp_rs(disp_rs), .disp_ce_b(disp_ce_b),
    .disp_reset_b(disp_reset_b), .disp_data_out(disp_data_out));
module debounce (reset, clock, noisy, clean);

  input reset, clock, noisy;
  output clean;
  
  reg [18:0] count;
  reg new, clean;

  always @(posedge clock)
  begin
    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 mono interface to AC97
//
/////////////////////////////////////////////////////////////////////////////
module fft_audio (clock_27mhz, reset, volume,
  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;
  input [4:0] volume;
  output [15:0] audio_in_data;
  input [15:0] 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 [2:0] source;
  assign source = 0;  //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

wire ac97_ready;
ac97 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);

// generate two pulses synchronous with the clock: first capture, then ready
reg [2:0] ready_sync;
always @(posedge clock_27mhz) begin
  ready_sync <= {ready_sync[1:0], ac97_ready};
end
assign ready = ready_sync[1] & ~ready_sync[2];

reg [15:0] out_data;
always @(posedge clock_27mhz)
  if (ready) out_data <= audio_out_data;
assign audio_in_data = left_in_data[19:4];
assign left_out_data = {out_data, 4'b0000};
assign right_out_data = left_out_data;

// generate repeating sequence of read/writes to AC97 registers
ac97commands cmds(clock_27mhz, ready, command_address, command_data,
command_valid, volume, source);
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 "0000"
right_in_data <= 20'h00000;
// synthesis attribute init of right_in_data is "0000"
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 [2:0] source;

reg [23:0] command;
reg command_valid;
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";
  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; // convert to attenuation

always @ (posedge clock) begin
  if (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 <= { 8'h1A, 5'b00000, source, 5'b00000, source};
    4'h7: // Record gain = max
    command <= 24'h1C_0F0F;
    4'h9: // set +20db mic gain
    command <= 24'h0E_8048;
  endcase

endmodule
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)
end // always @ (posedge clock)
endmodule // ac97commands

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Engineer: Andres
// Create Date:  16:37:04 12/03/2012
// Module Name:  process_audio
// Project Name:  Recorder Hero
////////////////////////////////////////////////////////////////////////////////////
module process_audio(clock_27mhz,reset,ready,from_ac97_data,haddr,hdata,hwe);
input clock_27mhz;
input reset;
input ready;
input [15:0] from_ac97_data;
output [11:0] haddr;
output [9:0] hdata;
output hwe;

wire signed [22:0] xk_re,xk_im;
wire [11:0] xk_index;

// IP Core Gen -> Digital Signal Processing -> Transforms -> FFTs
// -> Fast Fourier Transform v3.2
// Transform Length: 4096
// Implementation options: Pipelined, Streaming I/O
// Transform Length options: none
// Input data width: 8
// Phase factor width: 8
// Optional pins: CE
// Scaling options: Unscaled
// Rounding mode: Truncation
// Number of stages using Block Ram: 7
// Output ordering: Bit/Digit Reversed Order

my_fft fft2(.clk(clock_27mhz), .ce(ready | reset),
           .xn_re(from_ac97_data[15:8]), .xn_im(8'b0),
           .start(1'b1), .fwd_inv(1'b1), .fwd_inv_we(reset),
           .xk_re(xk_re), .xk_im(xk_im), .xk_index(xk_index));

// process_fft data
parameter [3:0] sel = 4'b1000;
reg [2:0] state;
reg [11:0] haddr;
reg [19:0] rere,imim;
reg [19:0] mag2;
wire signed [9:0] xk_re_scaled = xk_re >> sel;
wire signed [9:0] xk_im_scaled = xk_im >> sel;
reg hwe;
wire sqrt_done;
always @ (posedge clock_27mhz) begin
  hwe <= 0;
  if (reset) begin
    state <= 0;
  end
  else case (state)
    3'h0: if (ready) state <= 1;
    3'h1: begin
      // Only look at bins that we care about
      state <= 0;
      if (xk_index == 12'b000000101010) begin
        state <= 2;
        haddr <= xk_index;
        rere <= xk_re_scaled * xk_re_scaled;
        imim <= xk_im_scaled * xk_im_scaled;
      end
      if (xk_index == 12'b000000101100) begin
        state <= 2;
        haddr <= xk_index;
        rere <= xk_re_scaled * xk_re_scaled;
        imim <= xk_im_scaled * xk_im_scaled;
      end
      if (xk_index == 12'b000001000010) begin
        state <= 2;
        haddr <= xk_index;
        rere <= xk_re_scaled * xk_re_scaled;
        imim <= xk_im_scaled * xk_im_scaled;
      end
      if (xk_index == 12'b000000010100) begin
        state <= 2;
        haddr <= xk_index;
        rere <= xk_re_scaled * xk_re_scaled;
        imim <= xk_im_scaled * xk_im_scaled;
      end
      if (xk_index == 12'b000001000110) begin
        state <= 2;
        haddr <= xk_index;
        rere <= xk_re_scaled * xk_re_scaled;
        imim <= xk_im_scaled * xk_im_scaled;
      end
      if (xk_index == 12'b000000011011) begin
        state <= 2;
        haddr <= xk_index;
        rere <= xk_re_scaled * xk_re_scaled;
        imim <= xk_im_scaled * xk_im_scaled;
      end
      if (xk_index == 12'b000001000100) begin
        state <= 2;
        haddr <= xk_index;
        rere <= xk_re_scaled * xk_re_scaled;
        imim <= xk_im_scaled * xk_im_scaled;
      end
      if (xk_index == 12'b000000010010) begin
        state <= 2;
        haddr <= xk_index;
        rere <= xk_re_scaled * xk_re_scaled;
        imim <= xk_im_scaled * xk_im_scaled;
      end
      if (xk_index == 12'b000001001011) begin
        state <= 2;
        haddr <= xk_index;
        rere <= xk_re_scaled * xk_re_scaled;
        imim <= xk_im_scaled * xk_im_scaled;
      end
      if (xk_index == 12'b000000010111) begin
        state <= 2;
        haddr <= xk_index;
        rere <= xk_re_scaled * xk_re_scaled;
        imim <= xk_im_scaled * xk_im_scaled;
      end
    end
if (xk_index == 12'b000100101100) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000001011001) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000001001000) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000000100001) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000000100111) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000000100111) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000000010001) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000001001011) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000000001110) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000001001111) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000010011111) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000010010110) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000000001111) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b0000000010001) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b0000000010011) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000000001011) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000000001100) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000000011100) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000001101010) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000011001000) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000000101110) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000010111101) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000100001011) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000010110010) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end
haddr <= xk_index;
rere <= xk_re_scaled * xk_re_scaled;
imim <= xk_im_scaled * xk_im_scaled;

if (xk_index == 12'b000000100101) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000011100001) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000011101110) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000000011101) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000000100011) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000010101001) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000001111110) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000001111111) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000010001101) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000001110111) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b00000000000011111) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b0000011110001) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000000111011) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000001111110) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000001111111) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000010001101) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000001110111) begin
  state <= 2;
  haddr <= xk_index;

rere <= xk_re_scaled * xk_re_scaled;
imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000000011001) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000000011100) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000000101000) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000000111011) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000001000110) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000000011111) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000000111010) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000000011001) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000000111000) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000000011010) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000000111011) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

if (xk_index == 12'b000000011111) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000000010101) begin
state <= 2;
haddr <= xk_index;
rere <= xk_re_scaled * xk_re_scaled;
imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000000100111) begin
state <= 2;
haddr <= xk_index;
rere <= xk_re_scaled * xk_re_scaled;
imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000000010000) begin
state <= 2;
haddr <= xk_index;
rere <= xk_re_scaled * xk_re_scaled;
imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000000010111) begin
state <= 2;
haddr <= xk_index;
rere <= xk_re_scaled * xk_re_scaled;
imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000000011000) begin
state <= 2;
haddr <= xk_index;
rere <= xk_re_scaled * xk_re_scaled;
imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000000011010) begin
state <= 2;
haddr <= xk_index;
rere <= xk_re_scaled * xk_re_scaled;
imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000000001011) begin
state <= 2;
haddr <= xk_index;
rere <= xk_re_scaled * xk_re_scaled;
imim <= xk_im_scaled * xk_im_scaled;
end
if (xk_index == 12'b000000110010) begin
  state <= 2;
  haddr <= xk_index;
  rere <= xk_re_scaled * xk_re_scaled;
  imim <= xk_im_scaled * xk_im_scaled;
end

3'h2: begin
  state <= 3;
  mag2 <= rere + imim;
end

3'h3: if (sqrt_done) begin
  state <= 0;
  hwe <= 1;
end
endcase
end

wire [9:0] mag;
sqrt sqmag(clock_27mhz,mag2,state==2,mag,sqrt_done);
defparam sqmag.NBITS = 20;

assign hdata = mag;

endmodule

module sqrt(clk,data,start,answer,done);
parameter NBITS = 8; // max 32
parameter MBITS = (NBITS+1)/2;
input clk,start;
input [NBITS-1:0] data;
output [MBITS-1:0] answer;
output done;

reg [MBITS-1:0] answer;
reg busy;
reg [4:0] bit;

wire [MBITS-1:0] trial = answer | (1 << bit);

always @ (posedge clk) begin
  if (busy) begin
    if (bit == 0) busy <= 0;
    else bit <= bit - 1;
    if (trial*trial <= data) answer <= trial;
  end
  else if (start) begin
    busy <= 1;
    answer <= 0;
    bit <= MBITS - 1;
  end
end

assign done = ~busy;
endmodule
timescale 1ns / 1ps

module noteIdentification(reset,clk,ready,switch,from_ac97_data,note,GuessAddr,readVal,analyzer1_out,analyzer2_out);
  input reset;
  input clk;
  input ready;
  input [7:0] switch;
  input [15:0] from_ac97_data;
  output [3:0] note;
  output reg[5:0] GuessAddr;
  output readVal;
  output [15:0] analyzer1_out;
  output [15:0] analyzer2_out;
  wire[9:0] threshold;

assign threshold = {2'd0,switch[7:0]};

//Hex representation of each note
parameter[3:0] Z = 4'b0000;
parameter[3:0] C = 4'b0001;
parameter[3:0] Cs = 4'b0010;
parameter[3:0] D = 4'b0011;
parameter[3:0] Ds = 4'b0100;
parameter[3:0] E = 4'b0101;
parameter[3:0] F = 4'b0110;
parameter[3:0] Fs = 4'b0111;
parameter[3:0] G = 4'b1000;
parameter[3:0] Gs = 4'b1001;
parameter[3:0] A = 4'b1010;
parameter[3:0] As = 4'b1011;
parameter[3:0] B = 4'b1100;

//Bin number for each note we sample (from 5 different octaves)
parameter[11:0] Btwo = 12'b000000101010;
parameter[11:0] GSfive = 12'b000100011011;
parameter[11:0] DSfive = 12'b000011010100;
parameter[11:0] Cthree = 12'b000000101100;
parameter[11:0] Aone = 12'b000000100101;
parameter[11:0] Ctwo = 12'b000000010110;
parameter[11:0] Gthree = 12'b000000010001;
parameter[11:0] Astwo = 12'b000000011011;
parameter[11:0] Cfour = 12'b000000010111;
parameter[11:0] Bthree = 12'b000000010100;
parameter[11:0] Afive = 12'b000000011001;
parameter[11:0] Fone = 12'b000000011101;
parameter[11:0] GSone = 12'b000000010001;
parameter[11:0] ATthree = 12'b000001001011;
parameter[11:0] Don = 12'b000000001100;
parameter[11:0] DSfour = 12'b000011011110;
parameter[11:0] Dfive = 12'b000011111110;
parameter[11:0] ASfive = 12'b000011111110;
parameter[11:0] Cfive = 12'b000000001111;
parameter[11:0] CSfive = 12'b000011111110;
parameter[11:0] Eone = 12'b000000001111;
parameter[11:0] FSfive = 12'b000011111110;
parameter[11:0] FSfour = 12'b000000011111;
parameter[11:0] FSthree = 12'b000000111111;
parameter[11:0] FStwo = 12'b000001111111;
parameter[11:0] FSone = 12'b000000001111;
parameter[11:0] DSone = 12'b000000011110;
parameter[11:0] DStwo = 12'b000000011100;
parameter[11:0] DStwo = 12'b000000011001;
parameter[11:0] Dfour = 12'b000000011010;
parameter[11:0] Dfour = 12'b000000001111;
parameter[11:0] Dfour = 12'b000000001011;
parameter[11:0] Dfour = 12'b000000001100;
parameter[11:0] Dfour = 12'b000000001000;
parameter[11:0] DSone = 12'b000000011100;
parameter[11:0] DSone = 12'b000000011011;
parameter[11:0] DSone = 12'b000000011001;
parameter[11:0] DSone = 12'b000000001101;
parameter[11:0] DSone = 12'b000000001011;
//fft module, returns bin number and magnitude given ac97 signal
//process_audio audio(clk,reset,ready,from_ac97_data,haddr,hdata,hwe);

reg done;
reg[3:0] currentGuess = 0;
wire [11:0] haddr;
wire [9:0] hdata;
wire hwe;

reg[5:0] writeaddr = 0;
wire [5:0] writeaddrZ;
reg writeVal = 0;
wire writeValZ;
reg write = 0;
wire writeZ;
reg writeNext;

assign writeZ = write;
assign writeaddrZ = writeaddr;
assign writeValZ = writeVal;

// Use a clock cycle to turn process audio outputs into a proper ram input
always @(posedge clk) begin
    // If process_audio has a new value
    if (hwe) begin
        writeNext <= 1;
    // From bin determine the ram address
    if (haddr == Cone) writeaddr <= 6'd0;
    if (haddr == CSone) writeaddr <= 6'd1;
    if (haddr == Done) writeaddr <= 6'd2;
    if (haddr == DSone) writeaddr <= 6'd3;
    if (haddr == Eone) writeaddr <= 6'd4;
    if (haddr == Fone) writeaddr <= 6'd5;
    if (haddr == FSone) writeaddr <= 6'd6;
    if (haddr == Gone) writeaddr <= 6'd7;
    if (haddr == GSone) writeaddr <= 6'd8;
    if (haddr == Aone) writeaddr <= 6'd9;
    if (haddr == ASone) writeaddr <= 6'd10;
    if (haddr == Bone) writeaddr <= 6'd11;
    if (haddr == Ctwo) writeaddr <= 6'd12;
    if (haddr == CSTwo) writeaddr <= 6'd13;
    if (haddr == Dtwo) writeaddr <= 6'd14;
    if (haddr == DSTwo) writeaddr <= 6'd15;
    if (haddr == Etwo) writeaddr <= 6'd16;
    if (haddr == Ftwo) writeaddr <= 6'd17;
    if (haddr == FSTwo) writeaddr <= 6'd18;
    if (haddr == Gtwo) writeaddr <= 6'd19;
    if (haddr == GSTwo) writeaddr <= 6'd20;
    if (haddr == Atwo) writeaddr <= 6'd21;
    if (haddr == ASTwo) writeaddr <= 6'd22;
    if (haddr == Bthree) writeaddr <= 6'd23;
    if (haddr == Cthree) writeaddr <= 6'd24;
    if (haddr == CSThree) writeaddr <= 6'd25;
    if (haddr == Dthree) writeaddr <= 6'd26;
    if (haddr == DSThree) writeaddr <= 6'd27;
    if (haddr == Ethree) writeaddr <= 6'd28;
    if (haddr == Fthree) writeaddr <= 6'd29;
    if (haddr == FSThree) writeaddr <= 6'd30;
    if (haddr == Gthree) writeaddr <= 6'd31;
    if (haddr == GSThree) writeaddr <= 6'd32;
    if (haddr == Athree) writeaddr <= 6'd33;
    if (haddr == ASThree) writeaddr <= 6'd34;
    if (haddr == Bthree) writeaddr <= 6'd35;
    if (haddr == Cfour) writeaddr <= 6'd36;
    if (haddr == CFour) writeaddr <= 6'd37;
    if (haddr == Dfour) writeaddr <= 6'd38;
    if (haddr == Dfour) writeaddr <= 6'd39;
    if (haddr == Dfour) writeaddr <= 6'd40;
    if (haddr == Dfour) writeaddr <= 6'd41;
    if (haddr == Dfour) writeaddr <= 6'd42;
    if (haddr == Dfour) writeaddr <= 6'd43;
    if (haddr == Dfour) writeaddr <= 6'd44;
    if (haddr == Dfour) writeaddr <= 6'd45;
if (haddr == ASfour) writeaddr <= 6'd46;
if (haddr == Bfour) writeaddr <= 6'd47;
if (haddr == Cfive) writeaddr <= 6'd48;
if (haddr == CSfive) writeaddr <= 6'd49;
if (haddr == Dfive) writeaddr <= 6'd50;
if (haddr == Dfive) writeaddr <= 6'd51;
if (haddr == Efive) writeaddr <= 6'd52;
if (haddr == Ffive) writeaddr <= 6'd53;
if (haddr == FSfive) writeaddr <= 6'd54;
if (haddr == Gfive) writeaddr <= 6'd55;
if (haddr == Gfive) writeaddr <= 6'd56;
if (haddr == Afive) writeaddr <= 6'd57;
if (haddr == ASFive) writeaddr <= 6'd58;
if (haddr == Bfive) writeaddr <= 6'd59;

// Now set the write value (note was detected if over a threshold)
if (hdata > threshold) writeVal <= 1;
else writeVal <= 0;

end
// Delay write one cycle to assure correct values
else begin
  if (writeNext) begin
    write <= 1;
    writeNext <= 0;
  end
  else write <= 0;
end

reg[5:0] currentAddr = 0;
wire[5:0] readaddr;
reg[5:0] cycleAddr;
reg cycleVal;

// Ram that holds the most current values for all 60 notes, and which we read to
determine the lowest note played
cache fftcache(.addra(writeaddrZ),.dina(writeValZ),.wea(write),.clka(clk),
.ADDRb(readaddr),.clkb(clk),.doutb(readVal));

always @ (posedge clk) begin
  // Reset, done on reset or every time a new value is placed in the ram
  if (hwe | reset) begin
    done <= 0;
    currentAddr <= 0;
    currentGuess <= 0;
  end
  // If I haven't found a note played this cycle
  if (!done) begin
    // If the value read is a one (note played)
    if (readVal) begin
      // Determine the note that was played
      case(currentAddr)
        6'd0: currentGuess <= C;
        6'd1: currentGuess <= C5;
        6'd2: currentGuess <= D;
        6'd3: currentGuess <= D5;
        6'd4: currentGuess <= E;
    end
case
  currentGuess <= F;
  currentGuess <= Fs;
  currentGuess <= G;
  currentGuess <= Gs;
  currentGuess <= A;
  currentGuess <= As;
  currentGuess <= B;
  currentGuess <= Bs;
  currentGuess <= C;
  currentGuess <= Cs;
  currentGuess <= D;
  currentGuess <= Ds;
  currentGuess <= E;
  currentGuess <= Es;
  currentGuess <= F;
  currentGuess <= Fs;
  currentGuess <= G;
  currentGuess <= Gs;
  currentGuess <= A;
  currentGuess <= As;
  currentGuess <= B;
  currentGuess <= Bs;
  currentGuess <= C;
  currentGuess <= Cs;
  currentGuess <= D;
  currentGuess <= Ds;
  currentGuess <= E;
  currentGuess <= Es;
  currentGuess <= F;
  currentGuess <= Fs;
  currentGuess <= G;
  currentGuess <= Gs;
  currentGuess <= A;
  currentGuess <= As;
  currentGuess <= B;
endcase

// Debugging purposes
GuessAddr <= currentAddr;
// Stop until next cycle
done <= 1;
end  
  // If found 0, try the next bin  
else currentAddr <- currentAddr + 1;
end

assign readaddr = currentAddr;
assign note = currentGuess;

assign analyzer1_out = {hdata[5:0], writeaddrZ, writeValZ, hwe, clk, ready};
assign analyzer2_out = {hdata[9:6], haddr};

endmodule

`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////////
//  Engineer:  Paul Hemberger
//  Create Date:  15:40:51 11/26/2012  
//  Module Name:  musical_score_loader  
//  Project Name:  Recorder Hero
// /////////////////////////////////////////////////////////////////////////////
module musical_score_loader(
  input clk,
  input reset,
  input [1:0] song_id,
  output [25:0] tempo_out,
  output [63:0] next_notes_out,
  output song_done
);

  reg [7:0] read_addr = 0;
  reg [3:0] next_notes[15:0];
  reg [25:0] tempo;
  reg song_has Ended = 1;

  wire [3:0] next_note_lotr;
  wire [3:0] next_note_ss;
  wire [3:0] next_note_saints;
  wire [3:0] next_note_greensleeves;

  // Load each song in
  song_scales ss(.clka(clk), .addra(read_addr), .douta(next_note_ss));
  lotr_song lotr(.clka(clk), .addra(read_addr), .douta(next_note_lotr));
  when_the_saints wts(.clka(clk), .addra(read_addr), .douta(next_note_saints));
  greensleeves gs(.clka(clk), .addra(read_addr), .douta(next_note_greensleeves));

  // Counter will produce the song's beat
  wire tempo_beat;
  counter c(.clk(clk),
    .reset(reset),
    .count_to(tempo),
    .ready(tempo_beat));
```verilog
// Each song's tempo stored here. The tempos are put into the counter, // the beat will be how long it takes for the clock to count to that value always @(*) begin
  case(song_id)
    2'b00: tempo = 26'b00_1111_0111_1111_0100_1001_0000;
    2'b01: tempo = 26'b00_1111_0111_1111_0100_1001_0000;
    2'b10: tempo = 26'b0_1111_0111_1111_0100_1001_0000_0;
    2'b11: tempo = 26'b0_1111_0111_1111_0100_1001_0000_0;
    default: tempo = 26'b0_1111_0111_1111_0100_1001_0000_0;
  endcase
end

integer i;
always @(posedge clk) begin
  if (reset) begin
    read_addr <= 7'b0;
    song_has Ended <= 0;
  // When a song reloads fill it rests initially
  for (i=0; i < 16; i=i+1) begin
    next_notes[i] <= 4'b0000;
  end
  end else if (tempo_beat) begin
    // Shift each note
    for (i=0; i < 15; i=i+1) begin
      next_notes[i] <= next_notes[i+1];
    end
    // Find the next note to load, this is where different // songs are loaded into the game
    if (song_has_ended == 1) begin
      next_notes[15] <= 4'b0000;
    end else begin
      case(song_id)
        2'b00: next_notes[15] <= next_note_lotr;
        2'b01: next_notes[15] <= next_note_saints;
        default: next_notes[15] <= 4'b0000;
      endcase
      read_addr <= read_addr + 1;
    end
  end

  if (next_notes[15] == 4'b1111) song_has_ended <= 1;
end

assign next_notes_out = { next_notes[15], next_notes[14], next_notes[13],
                         next_notes[12], next_notes[11], next_notes[10],
                         next_notes[9], next_notes[8], next_notes[7],
                         next_notes[6], next_notes[5], next_notes[4],
                         next_notes[3], next_notes[2], next_notes[1],
                         next_notes[0]};
assign song_done = (next_notes[0] == 4'b1111);
assign tempo_out = tempo;
endmodule
```
module counter(
    input clk,
    input reset,
    input [26:0] count_to,
    output ready)
);

    reg [26:0] count;
    reg [26:0] goal;
    reg enable = 0;

    always @(posedge clk) begin
        if (reset) begin
            count <= 0;
            goal <= count_to;
        end else begin
            count <= count + 1;
            if (count == goal) begin
                count <= 0;
            end
        end
    end

    assign ready = (count == goal);
endmodule

module counter_tb;

    // Inputs
    reg clk;
    reg reset;
    reg [26:0] count_to;

    // Outputs
    wire ready;

    // Instantiate the Unit Under Test (UUT)
counter uut (  
  .clk(clk),  
  .reset(reset),  
  .count_to(count_to),  
  .ready(ready)  
);  
always #1 clk = ~clk;  
initial begin  
  // Initialize Inputs  
  clk = 0;  
  reset = 0;  
  count_to = 0;  
  // Wait 100 ns for global reset to finish  
  #100;  
  // Add stimulus here  
  reset = 1;  
  count_to = 15;  
  #2;  
  reset = 0;  
  #50;  
  reset = 1;  
  count_to = 5;  
  #2;  
  reset = 0;  
end  
endmodule  

`timescale 1ns / 1ps  
Thông qua / ///  
// Engineer: Andres Romero  
// Create Date: 14:53:05 11/27/2012  
// Module Name: menuFSM  
// Project Name: Recorder Hero  
// Additional Comments: Is both Menu and High Score tracking in one  
module menuFSM(  
  input up,  
  input down,  
  input enter,  
  input reset,  
  input done,  
  input clk,  
  input[17:0] binaryIn,
input[47:0] asciiIn,
output [2:0] menuState,
output resetComp,
   output [1:0] song,
output [47:0] highScore
);

reg[2:0] state;
reg reset_reg = 0;
reg[1:0] song_reg;

reg previous_button = 0;

parameter[2:0] songOne = 3'b000;
parameter[2:0] songTwo = 3'b001;
parameter[2:0] songThree = 3'b010;
parameter[2:0] songFour = 3'b011;
parameter[3:0] inGame = 3'b111;

reg[17:0] binaryHighScore1 = 0;
reg[47:0] asciiHighScore1 = {6'b00110000};

reg[17:0] binaryHighScore2 = 0;
reg[47:0] asciiHighScore2 = {6'b00110000};

reg[17:0] binaryHighScore3 = 0;
reg[47:0] asciiHighScore3 = {6'b00110000};

reg[17:0] binaryHighScore4 = 0;
reg[47:0] asciiHighScore4 = {6'b00110000};

reg[47:0] highScoreReg = {6'b00110000};

always @(posedge clk) begin
   //Resetting
   if (reset) state <= songOne;
   //If you pressed enter from the menu, load the correct current high score
   else if (enter && (state != inGame)) begin
      case(state[1:0])
         2'b00: highScoreReg <= asciiHighScore1;
         2'b01: highScoreReg <= asciiHighScore2;
         2'b10: highScoreReg <= asciiHighScore3;
         2'b11: highScoreReg <= asciiHighScore4;
         default:;
      endcase
      // Set the correct state as well as current song, and output a reset to the necessary components
      state <= inGame;
      song_reg <= state[1:0];
      reset_reg <= 1;
   end
   else begin
      reset_reg <= 0;
      // Update the current high score for the song if it is higher
   end
end


than the previous one once the song is done

```markdown
if (done && (state == inGame)) begin
  case (song_reg)
    2'b00: begin
      if (binaryIn > binaryHighScore1) begin
        asciiHighScore1 <= asciiIn;
        binaryHighScore1 <= binaryIn;
      end
    end
    2'b01: begin
      if (binaryIn > binaryHighScore2) begin
        asciiHighScore2 <= asciiIn;
        binaryHighScore2 <= binaryIn;
      end
    end
    2'b10: begin
      if (binaryIn > binaryHighScore3) begin
        asciiHighScore3 <= asciiIn;
        binaryHighScore3 <= binaryIn;
      end
    end
    2'b11: begin
      if (binaryIn > binaryHighScore4) begin
        asciiHighScore4 <= asciiIn;
        binaryHighScore4 <= binaryIn;
      end
    end
    default: ;
  endcase
  // Also go back to the menu
  state <= songOne;
end

// Make sure a button wasn't just pressed
if (!previous_button) begin
  // Update the current state depending on the button pressed
  case (state)
    songOne: state <= down ? songTwo: songOne;
    songTwo: state <= up ? songOne: songTwo;
    songThree: state <= up ? songTwo: down ? songFour:
    songFour: state <= up ? songThree: down ? songFour:
    inGame: state <= done ? songOne: inGame;
    default: state <= songOne;
  endcase
  // Stop it from reaching this case until both buttons are depressed
  previous_button <= 1;
end

// Both buttons are depressed, allow it to change states again
if (!down & !up) previous_button <= 0;
end

assign highScore = highScoreReg;
assign menuState = state;
assign resetComp = reset_reg;
assign song = song_reg;
endmodule
```
module binaryCounterASCII( 
    input clk, 
    input reset, 
    input score, 
    output [47:0] asciiScore 
);

reg[4:0] ones = 0; 
reg[4:0] tens = 0; 
reg[4:0] hundreds = 0; 
reg[4:0] thousands = 0; 
reg[4:0] tenthouands = 0; 
reg[4:0] hundredthousands = 0;

//Counters for each digit up to the 6th digit, can count up to whatever ascii value desired.
always @(posedge clk)begin 
    if (reset) begin 
        ones <= 0; 
        tens <= 0; 
        hundreds <= 0; 
        thousands <= 0; 
        tenthouands <= 0; 
        hundredthousands <= 0;
    end
    if (score) begin 
        ones <= ones + 1; 
        if (ones + 1 == 4'd10) begin 
            ones <= 0; 
            tens <= tens + 1; 
        end
        if (tens + 1 == 4'd10) begin 
            tens <= 0; 
            hundreds <= hundreds + 1; 
        end
        if (hundreds + 1 == 4'd10) begin 
            hundreds <= 0; 
            thousands <= thousands + 1; 
        end
        if (thousands + 1 == 4'd10) begin 
            thousands <= 0; 
            tenthouands <= tenthouands + 1; 
        end
        if (tenthouands + 1 == 4'd10) begin 
            tenthouands <= 0; 
            hundredthousands <= hundredthousands + 1; 
        end
    end
end
//Assign output
assign asciiScore = {8'b00110000 + hundredthousands, 8'b00110000 + tenthouands, 8'b00110000 + thousands, 8'b00110000 + hundreds, 8'b00110000 + tens, 8'b00110000 + ones};
endmodule

`timescale 1ns / 1ps

module hexToAscii(
    input [3:0] hex,
    input clk,
    output [15:0] asciiNote
);

parameter[7:0] space = 8'b00100000;
parameter[7:0] pound = 8'b00100011;

parameter[7:0] A = 8'b01000001;
parameter[7:0] B = 8'b01000010;
parameter[7:0] C = 8'b01000011;
parameter[7:0] D = 8'b01000100;
parameter[7:0] E = 8'b01000101;
parameter[7:0] F = 8'b01000110;
parameter[7:0] G = 8'b01000111;

reg [15:0] ascii;

//Takes hex representation of note and translates to ascii.
always @(posedge clk) begin
    case (hex)
        4'b0000: ascii <= {space,space};
        4'b0001: ascii <= {C,space};
        4'b0010: ascii <= {C,pound};
        4'b0011: ascii <= {D,space};
        4'b0100: ascii <= {D,pound};
        4'b0101: ascii <= {E,space};
        4'b0110: ascii <= {F,space};
        4'b0111: ascii <= {F,pound};
        4'b1000: ascii <= {G,space};
        4'b1001: ascii <= {G,pound};
        4'b1010: ascii <= {A,space};
        4'b1011: ascii <= {A,pound};
        4'b1100: ascii <= {B,space};
        default: ascii <= {space,space};
    endcase
end

assign asciiNote = ascii;
endmodule
module vram_display(reset, clk, hcount, vcount, vr_pixel,
                      vram_addr, vram_read_data);

input  reset, clk;
input [10:0] hcount;
input [9:0] vcount;
output [7:0] vr_pixel;
output [18:0] vram_addr;
input [35:0] vram_read_data;

// forecast hcount & vcount 8 clock cycles ahead to get data from ZBT
wire [10:0] hcount_f;
    assign hcount_f = (hcount >= 1024) ? (hcount - 1024) : (hcount + 2);
wire [9:0] vcount_f;
    assign vcount_f = (hcount >= 1024) ? (vcount == 805) ? 0 : vcount + 1) : vcount;

wire [18:0] ram_addr;
    assign vram_addr = {1'b0, vcount_f, hcount_f[9:2]};
//    assign vram_addr = {1'b0, vcount, hcount[9:2]};

wire [1:0] hc4;
    assign hc4 = hcount[1:0];

reg [7:0]  vr_pixel;
reg [35:0] vr_data_latched;
reg [35:0] last_vr_data;

always @(posedge clk)
    last_vr_data <= (hc4==2'd3) ? vr_data_latched : last_vr_data;

always @(posedge clk)
    vr_data_latched <= (hc4==2'd1) ? vram_read_data : vr_data_latched;

always @(*) // each 36-bit word from RAM is decoded to 4 bytes
    case (hc4)
        2'd0: vr_pixel = last_vr_data[7:0];
        2'd1: vr_pixel = last_vr_data[7+8:0+8];
        2'd2: vr_pixel = last_vr_data[7+16:0+16];
        2'd3: vr_pixel = last_vr_data[7+24:0+24];
    endcase
endmodule // vram_display

`timescale 1ns / 1ps

`
// Module Name:  zbt_image_writer
// Project Name:  Recorder Hero

module zbt_image_writer(
    input clk,
    input reset,
    input [7:0] image_data,
    input new_input,
    output new_output,
    output [35:0] image_data_zbt);

    reg n_out = 0;
    reg [2:0] count = 0;
    reg [35:0] image_row = 36'b0;
    integer i;
    always @(posedge clk) begin
        if (reset) begin
            count <= 0;
            image_row <= 36'b0;
            n_out <= 0;
        end else begin
            if (new_input) begin
                if (count == 3) begin
                    count <= 0;
                    image_row[35:32] <= 4'h0;
                    n_out <= 1;
                end else begin
                    count <= count + 1;
                end
                if (count == 0) begin
                    image_row[35:0] <= 36'b0;
                end
                case(count)
                    3'd0: image_row[7:0] <= image_data;
                    3'd1: image_row[15:8] <= image_data;
                    3'd2: image_row[23:16] <= image_data;
                    3'd3: image_row[31:24] <= image_data;
                    default: image_row[35:0] <= 36'b0;
                endcase
                //image_row[(count+1)*8-1 : 8] <= image_data;
                // for (i=(count)*8-1; i < (count+1)*8-1; i=i+1) begin
                //     image_row[i] <= image_data[i];
                // end
            end
            end
            n_out <= 0;
        end
    endmodule

assign new_output = n_out;
assign image_data_zbt = (n_out == 1) ? image_row : 36'b0;
module zbt_image_writer_tb;

    // Inputs
    reg clk;
    reg reset;
    reg [7:0] image_data;
    reg new_input;

    // Outputs
    wire new_output;
    wire [35:0] image_data_zbt;

    // Instantiate the Unit Under Test (UUT)
    zbt_image_writer uut (  
        .clk(clk),
        .reset(reset),
        .image_data(image_data),
        .new_input(new_input),
        .new_output(new_output),
        .image_data_zbt(image_data_zbt)
    );

    always #5 clk = ~clk;

    initial begin
        // Initialize Inputs
        clk = 0;
        reset = 0;
        image_data = 0;
        new_input = 0;

        // Wait 100 ns for global reset to finish
        #100;

        // Add stimulus here
        new_input = 1;
        image_data = 8'hAA;

        #10;
        new_input = 0;

        #10;
    end
new_input = 1;
image_data = 8'hBB;

#10;
new_input = 0;

#10;
new_input = 1;
image_data = 8'hCC;

#10;
new_input = 0;

#10;
new_input = 1;
image_data = 8'hDD;

#10;
new_input = 0;

#50;
end
endmodule
// 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 ? vcount + 1) : vcount;
    vblank <= next_vblank;
    vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low
    blank <= next_vblank | (next_hblank & ~hreset);
end
endmodule

module rh_display (  
    input vclock,  
    input reset,  
    input up,  
    input down,  
    input playing_correct,  
    input [2:0] menu_state,  
    input [15:0] current_note_string,  
    input [63:0] high_score_string,  
    input [63:0] score_string,  
    input [63:0] next_notes,  
    input [25:0] tempo,  
    input [7:0] bono_image_bits,  
    input [10:0] hcount,  
    input [9:0] vcount,  
    input hsync,  
    input vsync,  
    input blank,  
    output phsync,  
    output pvsync,  
    output pblank,  
    output [23:0] pixel,  
    output [63:0] debug  
);

assign phsync = hsync;
assign pvsync = vsync;
assign pblank = blank;

parameter [10:0] SCREEN_WIDTH = 1023;
parameter [9:0] SCREEN_HEIGHT = 767;
parameter NOTE_WIDTH = 64;
parameter NOTE_HEIGHT = 35;
parameter FIRST_LETTER = 128 + 16;
parameter NOTE_STEP = 74;
parameter ACTION_LINE_X = 72;
parameter [23:0] COLOR = 24'hFF_FF_FF;

wire [3:0] notes[15:0];
wire [23:0] note_pixels[15:0];
// All notes positions are based on this one
reg [10:0] lead_note_x = 1023;

assign {notes[15], notes[14], notes[13], notes[12],
notes[11], notes[10], notes[9], notes[8],
notes [7], notes [6], notes [5], notes [4],
notes [3], notes [2], notes [1], notes [0]} = next_notes;

// Track y positions and colors for each note on screen
reg [9:0] note_y_pos[15:0];
reg [23:0] note_color[15:0];
wire [23:0] note_line_pixels[7:0];

/***
 Generators for creating the notes and staff lines ***/
genvar j;
generate
  for(j=0; j<16; j=j+1) begin:generate_note_blobs
    blob #(.
      WIDTH(NOTE_WIDTH),
      HEIGHT(NOTE_HEIGHT))
      note(.x(lead_note_x + NOTE_WIDTH * j),
      .y(note_y_pos[j]),
      .hcount(hcount),
      .vcount(vcount),
      .color(note_color[j]),
      .pixel(note_pixels[j]));
  end
endgenerate

genvar w;
generate
  for(w=0; w<8; w=w+1) begin:generate_note_lines
    blob #(.
      WIDTH(1023-72),
      HEIGHT(1))
      note_line(.x(ACTION_LINE_X),
      .y(128 + w*73),
      .hcount(hcount),
      .vcount(vcount),
      .color(24'hAA_AA_AA),
      .pixel(note_line_pixels[w]));
  end
endgenerate

////////////////////////////////////////////////////////////////////
////// TEMPO CONTROL ////
////////////////////////////////////////////////////////////////////

// Default song tempo is 1/2s per 8th note
reg load_tempo = 0;
reg [25:0] song_tempo = 26'b1111_0111_1111_0100_1001_0000_0;
wire tempo_beat; // Indicates 1/8th note beat
wire tempo_beat_move; // Indicates 1/8th*1/64 to move the notes

counter c(.clk(vclock),
  .reset(load_tempo),
  .count_to(song_tempo),
  .ready(tempo_beat));

// The width of one note is 64, break the tempo up by 64
// to know the interval we need to move the notes by 1px
counter c2(.clk(vclock),
   .reset(load_tempo),
   .count_to(song_tempo/64),
   .ready(tempo_beat_move));

FLICT BOUNDARY LINES *****

wire action_line = hcount == ACTION_LINE_X & vcount >= 128 & vcount <= 639;
wire right_boundary_line = hcount == 1022 & vcount >= 128 & vcount <= 639;

SHARE CONTROLING NOTE POSITIONS & COLORS ///

reg [23:0] onscreen_notes[15:0];
integer n, k;
always @(posedge vclock) begin
    if (reset) begin
        song_tempo <= tempo; // TODO: Revert back to 1023
        lead_note_x <= 1023;
        load_tempo <= 1;
    end else begin
        load_tempo <= 0;
    end
    // Debug single note
    if (hcount == 1 && vcount == 1) begin
        if (up) begin
            lead_note_x <= lead_note_x - 4;
        end else if (down) begin
            lead_note_x <= lead_note_x + 4;
        end
    end
    if (tempo_beat_move) begin
        lead_note_x <= lead_note_x - 1;
    end else if (tempo_beat) begin
        lead_note_x <= ACTION_LINE_X;
    end
    for (k=0; k<16; k=k+1) begin
        case(notes[k])
            4'd0: note_y_pos[k] <= FIRST_LETTER + 768;
            // C, C#
            4'd1: note_y_pos[k] <= FIRST_LETTER +
            6*NOTE_STEP;
            4'd2: note_y_pos[k] <= FIRST_LETTER +
            6*NOTE_STEP;
            // D, D#
            4'd3: note_y_pos[k] <= FIRST_LETTER +
            5*NOTE_STEP;
            4'd4: note_y_pos[k] <= FIRST_LETTER +
            5*NOTE_STEP;
            // E
            4'd5: note_y_pos[k] <= FIRST_LETTER +
            4*NOTE_STEP;
            // F, F#
            4'd6: note_y_pos[k] <= FIRST_LETTER +
            3*NOTE_STEP;
            4'd7: note_y_pos[k] <= FIRST_LETTER +
NOTE_STEP;

// G, G#
4'd8: note_y_pos[k] <= FIRST_LETTER +
2*NOTE_STEP;
4'p9: note_y_pos[k] <= FIRST_LETTER +
2*NOTE_STEP;

// A, A#
4'd10: note_y_pos[k] <= FIRST_LETTER +
1*NOTE_STEP;
4'p11: note_y_pos[k] <= FIRST_LETTER +
1*NOTE_STEP;

// B
4'd12: note_y_pos[k] <= FIRST_LETTER +
0*NOTE_STEP;

// C high
4'd13: note_y_pos[k] <= FIRST_LETTER +
6*NOTE_STEP;

// D high
4'd14: note_y_pos[k] <= FIRST_LETTER +
5*NOTE_STEP;

// EOF
4'd15: note_y_pos[k] <= FIRST_LETTER + 768;
default: note_y_pos[k] <= FIRST_LETTER;
endcase

case(notes[k])
    // Don't display rests
    4'd0: note_color[k] <= 24'h00_00_00;
    // C#
    4'd2: note_color[k] <= 24'h55_55_FF;
    // D#
    4'd4: note_color[k] <= 24'h55_55_FF;
    // F#
    4'd7: note_color[k] <= 24'h55_55_FF;
    // G#
    4'd9: note_color[k] <= 24'h55_55_FF;
    // A#
    4'd11: note_color[k] <= 24'h55_55_FF;
    // C high
    4'd13: note_color[k] <= 24'h00_DD_00;
    // D high
    4'd14: note_color[k] <= 24'h00_DD_00;
default: note_color[k] <= 24'hFF_FF_FF;
endcase

// Change color of to-be-played note based on whether
// the player is playing the right now
if (notes[0] == 4'd0) begin
    note_color[0] <= 24'h00_00_00;
end else if ((playing_correct) && (notes[0] >= 4'd0)) begin
    note_color[0] <= 24'hFF_FF_00;
end else if (!playing_correct) && (notes[0] >= 4'd0)) begin
    note_color[0] <= 24'hFF_FF_45_00;
end

for (n=0; n<16; n=n+1) begin
    onscreen_notes[n] <= (hcount > ACTION_LINE_X) ? note_pixels[n] : 0;
end
wire [2:0] score_pixel;
char_string_display csd_score(.vclock(vclock),
  .hcount(hcount),
  .vcount(vcount),
  .pixel(score_pixel),
  .cstring({"SCORE: ", score_string}),
  .cx(700),
  .cy(10));
defparam csd_score.NCHAR = 15;
defparam csd_score.NCHAR_BITS = 4;

wire [2:0] high_score_pixel;
char_string_display csd_hscore(.vclock(vclock),
  .hcount(hcount),
  .vcount(vcount),
  .pixel(high_score_pixel),
  .cstring({"HIGH SCORE: ", high_score_string}),
  .cx(120),
  .cy(10));
defparam csd_hscore.NCHAR = 20;
defparam csd_hscore.NCHAR_BITS = 5;

wire [2:0] current_note_pixel;
wire [10:0] current_note_x;
wire [9:0] current_note_y;
char_string_display csd_note(.vclock(vclock),
  .hcount(hcount),
  .vcount(vcount),
  .pixel(current_note_pixel),
  .cstring({"NOTE: ", current_note_string}),
  .cx(current_note_x),
  .cy(current_note_y));
defparam csd_note.NCHAR = 8;
defparam csd_note.NCHAR_BITS = 4;

wire [23:0] bmp_pixel;
picture_blob pb(.pixel_clk(vclock),
  .x(4),
  .hcount(hcount),
  .y(128),
  .vcount(vcount),
  .pixel(bmp_pixel));

wire [23:0] bono_pixel;
bono_picture_blob bpb_img(.pixel_clk(vclock),
  .x(0),
  .hcount(hcount+2),
  .y(0),
// Highlight the note currently being played
// on the scale on the Left by alphablending
// the image
reg [23:0] curr_note_color;
reg [9:0] curr_note_y;
reg [23:0] bmp_pixel_alpha;
always @(*) begin
  case(current_note_string)
    "B": curr_note_y = 127;
    "A#": curr_note_y = 127 + 1*NOTE_STEP;
    "A": curr_note_y = 127 + 1*NOTE_STEP;
    "G#": curr_note_y = 127 + 2*NOTE_STEP;
    "G": curr_note_y = 127 + 2*NOTE_STEP;
    "F#": curr_note_y = 127 + 3*NOTE_STEP;
    "F": curr_note_y = 127 + 3*NOTE_STEP;
    "E": curr_note_y = 127 + 4*NOTE_STEP;
    "D#": curr_note_y = 127 + 5*NOTE_STEP;
    "D": curr_note_y = 127 + 5*NOTE_STEP;
    "C#": curr_note_y = 127 + 6*NOTE_STEP;
    "C": curr_note_y = 127 + 6*NOTE_STEP;
  endcase
  case(current_note_string)
    "A#": curr_note_color = 24'h77_77_FF;
    "G#": curr_note_color = 24'h77_77_FF;
    "F#": curr_note_color = 24'h77_77_FF;
    "D#": curr_note_color = 24'h77_77_FF;
    "C#": curr_note_color = 24'h77_77_FF;
    default: curr_note_color = 24'hDD_DD_00;
  endcase

  // Alphablend the currently-played note along the music staff
  if (bmp_pixel
      && vcount >= curr_note_y
      && vcount <= curr_note_y + NOTE_STEP) begin
    bmp_pixel_alpha = bmp_pixel/4 + 3*(curr_note_color / 2);
  end else begin
    bmp_pixel_alpha = bmp_pixel;
  end
end

/////////////////////////////////////////////////////////////////////
// MAIN MENU DISPLAY //
/////////////////////////////////////////////////////////////////////
wire [2:0] song_title_1_pixel;
char_string_display csd_st1(.vclock(vclock),
                           .hcount(hcount),
                           .vcount(vcount),
                           .pixel(song_title_1_pixel),
                           .cstring("Concerning Hobbits"),
                           .cx(23),
                           .cy(285));
defparam csd_st1.NCHAR = 18;
defparam csd_st1.NCHAR_BITS = 5;
wire [2:0] song_title_2_pixel;
char_string_display csd_st2(.vclock(vclock),
  .hcount(hcount),
  .vcount(vcount),
  .pixel(song_title_2_pixel),
  .cstring("When the Saints..."),
  .cx(23),
  .cy(315));
defparam csd_st2.NCHAR = 18;
defparam csd_st2.NCHAR_BITS = 5;

wire [2:0] song_title_3_pixel;
char_string_display csd_st3(.vclock(vclock),
  .hcount(hcount),
  .vcount(vcount),
  .pixel(song_title_3_pixel),
  .cstring("Greensleeves"),
  .cx(23),
  .cy(345));
defparam csd_st3.NCHAR = 12;
defparam csd_st3.NCHAR_BITS = 4;

wire [2:0] song_title_4_pixel;
char_string_display csd_st4(.vclock(vclock),
  .hcount(hcount),
  .vcount(vcount),
  .pixel(song_title_4_pixel),
  .cstring("Practice Scale"),
  .cx(23),
  .cy(375));
defparam csd_st4.NCHAR = 14;
defparam csd_st4.NCHAR_BITS = 4;

// Create a block that indicates selected song
reg [8:0] current_song_y;
always @(*) begin
  case(menu_state[1:0])
    2'b00: current_song_y = 9'd285;
    2'b01: current_song_y = 9'd315;
    2'b10: current_song_y = 9'd345;
    2'b11: current_song_y = 9'd375;
    default: current_song_y = 9'd285;
  endcase
end

wire [23:0] song_select_box_pixel;
blob #(.
  .WIDTH(10), 
  .HEIGHT(10))
song_selector_box(.x(8),
  .y(current_song_y + 6),
  .hcount(hcount),
  .vcount(vcount),
  .color(24'hFF_FF_FF),
  .pixel(song_select_box_pixel));

///////////////////////////////
//////// LEGEND //////////
///////////////////////////////
wire [23:0] legend_sharp_pixel;
wire [23:0] legend_high_pixel;
wire [23:0] legend_hit_pixel;
wire [23:0] legend_miss_pixel;
wire [2:0] legend_sharp_text_pixel;
wire [2:0] legend_high_text_pixel;
wire [2:0] legend_hit_text_pixel;
wire [2:0] legend_miss_text_pixel;

blob #(.
  WIDTH(NOTE_WIDTH),
  .HEIGHT(NOTE_HEIGHT))
  legend_sharp(.x(196),
  .y(640+17),
  .hcount(hcount),
  .vcount(vcount),
  .color(24'h55_55_FF),
  .pixel(legend_sharp_pixel));

cartesian_string csd_lh(.vclock(vclock),
  .hcount(hcount),
  .vcount(vcount),
  .pixel(legend_high_text_pixel),
  .cstring("= Octave higher"),
  .cx(196+64+16),
  .cy(640+17));
defparam csd_lh.NCHAR = 15;
defparam csd_lh.NCHAR_BITS = 4;

blob #(.
  WIDTH(NOTE_WIDTH),
  .HEIGHT(NOTE_HEIGHT))
  legend_hit_text(.x(580),
  .y(640+17),
  .hcount(hcount),
  .vcount(vcount),
  .color(24'hFF_FF_00),
  .pixel(legend_hit_pixel));

cartesian_string csd_lhhittext(.vclock(vclock),
  .hcount(hcount),
  .vcount(vcount),
  .pixel(legend_hit_text_pixel),
  .cstring("= Hit"),
  .cx(580+64+16),
  .cy(640+17));
defparam csd_lhittext.NCHAR = 6;
defparam csd_lhittext.NCHAR_BITS = 3;

blob #(WIDTH(NOTE_WIDTH), HEIGHT(NOTE_HEIGHT))
  legend_miss_text(.x(580),
  .y(640+17+24+24),
  .hcount(hcount),
  .vcount(vcount),
  .color(24'hFF_45_00),
  .pixel(legend_miss_pixel));

char_string_display csd_lmisstext(.vclock(vclock),
  .hcount(hcount),
  .vcount(vcount),
  .pixel(legend_miss_text_pixel),
  .cstring(“Miss!”),
  .cx(580+64+16),
  .cy(640+17+24+24));
defparam csd_lmisstext.NCHAR = 7;
defparam csd_lmisstext.NCHAR_BITS = 3;

///// BUG FIX
// For some reason the image from the zbt shows a small sliver
// on the side. This hack at least covers it up on the main screen
wire [23:0] vga_display_hack_pixel;
blob #(WIDTH(16), HEIGHT(768))
  vga_display_hack(.x(0), .y(0),
  .hcount(hcount),
  .vcount(vcount),
  .color(24'h00_00_00),
  .pixel(vga_display_hack_pixel));

reg [23:0] pixel_reg;
wire [23:0] bono_pixel_fix;
assign bono_pixel_fix = (hcount < 16) ? 24'h00_00_00 : bono_pixel;

////////////////////////
///// FINAL OUTPUT ///
////////////////////////

// Display output for menu / game
always @(posedge vclock) begin
  if (!menu_state[2]) begin
    pixel_reg <= bono_pixel_fix
    | {8{song_title_1_pixel}}
    | {8{song_title_2_pixel}}
    | {8{song_title_3_pixel}}
    | {8{song_title_4_pixel}}
    | {8{current_note_pixel}}
    | song_select_box_pixel;
  end else begin
    pixel_reg <= onsreen_notes[0]
    | onsreen_notes[1]
    | onsreen_notes[2]
    | onsreen_notes[3]
    | onsreen_notes[4]
    | onsreen_notes[5]
    | onsreen_notes[6]
onscreen_notes[7]
onscreen_notes[8]
onscreen_notes[9]
onscreen_notes[10]
onscreen_notes[11]
onscreen_notes[12]
onscreen_notes[13]
onscreen_notes[14]
onscreen_notes[15]
note_line_pixels[0]
note_line_pixels[1]
note_line_pixels[2]
note_line_pixels[3]
note_line_pixels[4]
note_line_pixels[5]
note_line_pixels[6]
note_line_pixels[7]
{24{action_line}}
{24{right_boundary_line}}
{8{score_pixel}}
{8{high_score_pixel}}
{8{current_note_pixel}}
bmp_pixel_alpha
legend_sharp_pixel
legend_high_pixel
legend_hit_pixel
legend_miss_pixel
{8{legend_sharp_text_pixel}}
{8{legend_high_text_pixel}}
{8{legend_hit_text_pixel}}
{8{legend_miss_text_pixel}};

assign pixel = pixel_reg;
assign debug = {bono_pixel};
Appendix C – Images

Figure 9: Main menu image, U2's lead singer, Bono singing into a recorder
Figure 10: Music staff notes

B
A
G
F
E
D
C