Bloomberg Designs

Four-Channel FPGA Ambisonic Audio System

Ben Bloomberg

6.111 Fall 2009

Table of Contents

Introduction 1

Overview 1

Ambisonic Audio Processing 1

System Description 2

Audio Subsystem Modules 2

USB Reader 2

DLP-USB245M 2

Design Iterations 2

Coefficient ROMS 4

Coeff_full Module 4

Generating Coefficients 5

Encoder 5

Summing 5

Decoder 6

Design Considerations 6

Audio Output Driver 6

DCM Manipulation 6

PCM1681 Clocks 6

Serial Audio Output 7

PCM1681 Control 7

Video Subsystem Modules 7

XGA 7

Ambisonic UI 7

Preset Manager 8

Sync Coord Module 8

Conclusion 9

Design & Testing 9

DLP-USB245M FIFO Interface 9

PCM-1681 Interface 10

Audio DSP Logic 11

User Interface 12

System Integration 12

Final Thoughts 12

References 13

Appendix A: Verilog Implementation 14

Appendix B: Verilog Test Fixtures 15

Appendix B: Python Implementation 16

Introduction

Overview

The goal of ambisonics is to create a representation of 3-dimensional audio in a format that can be used on a speaker system of any size and shape. This document describes an efficient FPGA solution for encoding and decoding 3rd order ambisonic audio. To facilitate testing, there is an audio input module based on a DLP-USB254M FIFO chip and a multichannel audio output module based on a Texas Instruments Burr-Brown PCM-1681 8-channel digital to analog converter.

This allows four channels of audio to be placed in real time and output across as many as 8 speakers. The system also contains 4 preset locations which can be edited, saved and recalled. When switching between presets, the system interpolates the location coordinates of each channel so that audio does not stutter or jump, but fades smoothly. As channels move further away from the center of the of the sound field, their gain decreases by a constant factor, so that they seem to disappear into the distance.

The system is controlled via a push buttons and switches on the labkit device, and user feedback is accomplished using XGA output to show the location of the each incoming channel on the screen.

The computer interface to send audio to the system is a simple python script relying on FTDI native D2XX drivers and the PyUSB library for those drivers.

Ambisonic Audio Processing

Ambisonic encoding and decoding is based on spherical harmonics. Virtual audio sources are encoded by taking their location in space and describing it as a linear combination of a basis set of orthogonal spherical harmonics. When separated, the terms of the linear combination can be used as elements of a vector representation of the source. An 'encoded ambisonic stream' consists of one mono channel of audio and 15 copies of that channel, scaled by the relative components of each harmonic for that channel's location in space. This means that the decoding process is essentially a linear algebra projection onto another vector which represents the location of a physical source (a loudspeaker).

Encoded 16 channel streams may be added together to encapsulate multiple virtual sources in a single 16 channel representation. This representation can be decoded for any position in space, creating an accurate reproduction of each virtual source for a specific physical source location. The result is dynamic panning that adapts to any configuration of output devices.

Additionally, because the encoders and decoders operate independently, the process is completely parallel and thus, perfect for FPGA acceleration.

System Description

Audio Subsystem Modules

The audio subsystem is responsible for getting audio samples from the computer, encoding them ambisonically, summing each of the encoded streams together, and then decoding the stream for each physical source. The audio system works using a push-pull method, where a pulse from the audio output driver clears the data in every DSP module. At that point the first module, the coeff_rom, generates coefficients based on the current coordinates from the user interface. When complete, it asserts a coeffs_valid signal to the encoders, which in turn assert encoder_valid to the summing device. The summing device asserts a summing_valid signal to each of the decoders and on the next audio pulse, if the decoders have asserted their decoder_valid signals, the audio is pushed to output driver, the DSP is again cleared and the process starts over.

USB Reader

The USB reader module uses a FIFO buffer to grab individual bytes from a DLP-USB245M chip. It is a state-machine architecture that uses 8 states to pull each byte from the buffer. Unlike the DSP, it pushes a new sample of audio to each of the encoders on every audio pulse.

DLP-USB245M

The DLP devices USB FIFO allows the computer to send data at fairly high rates using a externally buffered transmission bus. On the labkit, the data is made available 1 byte at a time using a signaling scheme that allows bytes to be pulled at any clock speed. When new data is available, the chip sets an output pin RXF low. At that point the FPGA sets the RD pin low to retrieve the data. Once RD goes high again, the DLP chip prepares the next byte in the fifo and sets RXF low when it is ready.

In the final USB reader module, the RD bit is set to the inverse of the MSB of the state, so that RD is low for the first four states and high for the last four states. During 3rd state, data is stored to a FIFO on the labkit. In order to ensure the correct timing, the fifo_wr bit is registered and set 2 cycles before the RD pin goes high. The fifo_wr bit is checked on the next cycle and data is latched to the FIFO on the following cycle.

Design Iterations

The USB Reader state-machine went through several design iterations. Initially, the audio data was not streaming quickly enough to the lab kit. This was diagnosed using the second_notify module to illuminate one of the labkit LEDs when the buffer filled or emptied. A timing diagram for an early version of the USB state-machine is pictured below in figure 1. This diagram was captured when RD was calculated as a separate register, causing 1 cycle of delay on the RD line.

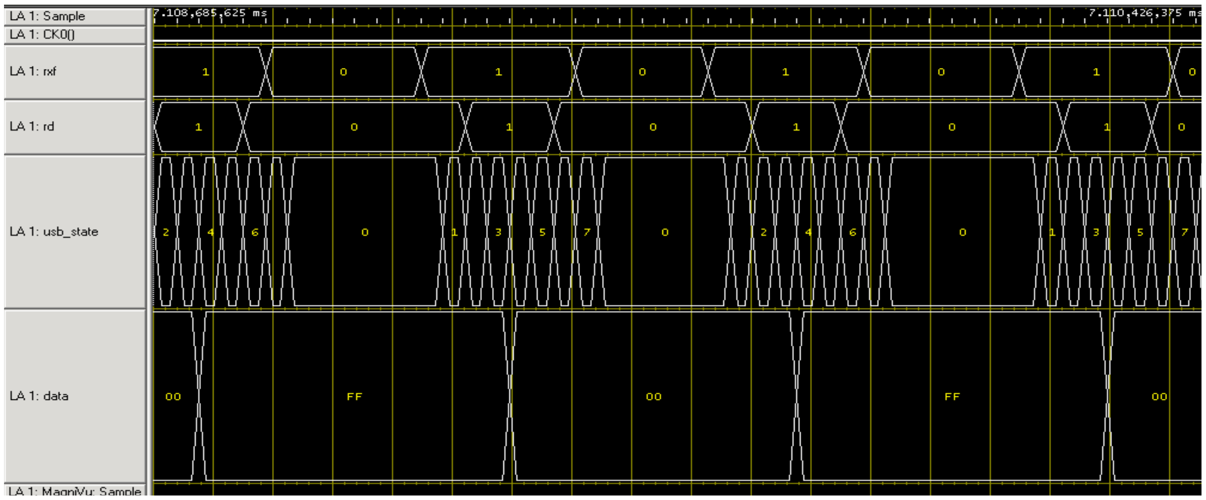

Figure 1: USB state machine using 8 states and a delayed RD signal

After much time spent trying to optimize the state-machine, it was realized that there were 6ms long delays in the USB data and that the chip and the state-machine were not at fault, but it was the computer that was not sending data quickly enough, this is seen in figure 2 below.

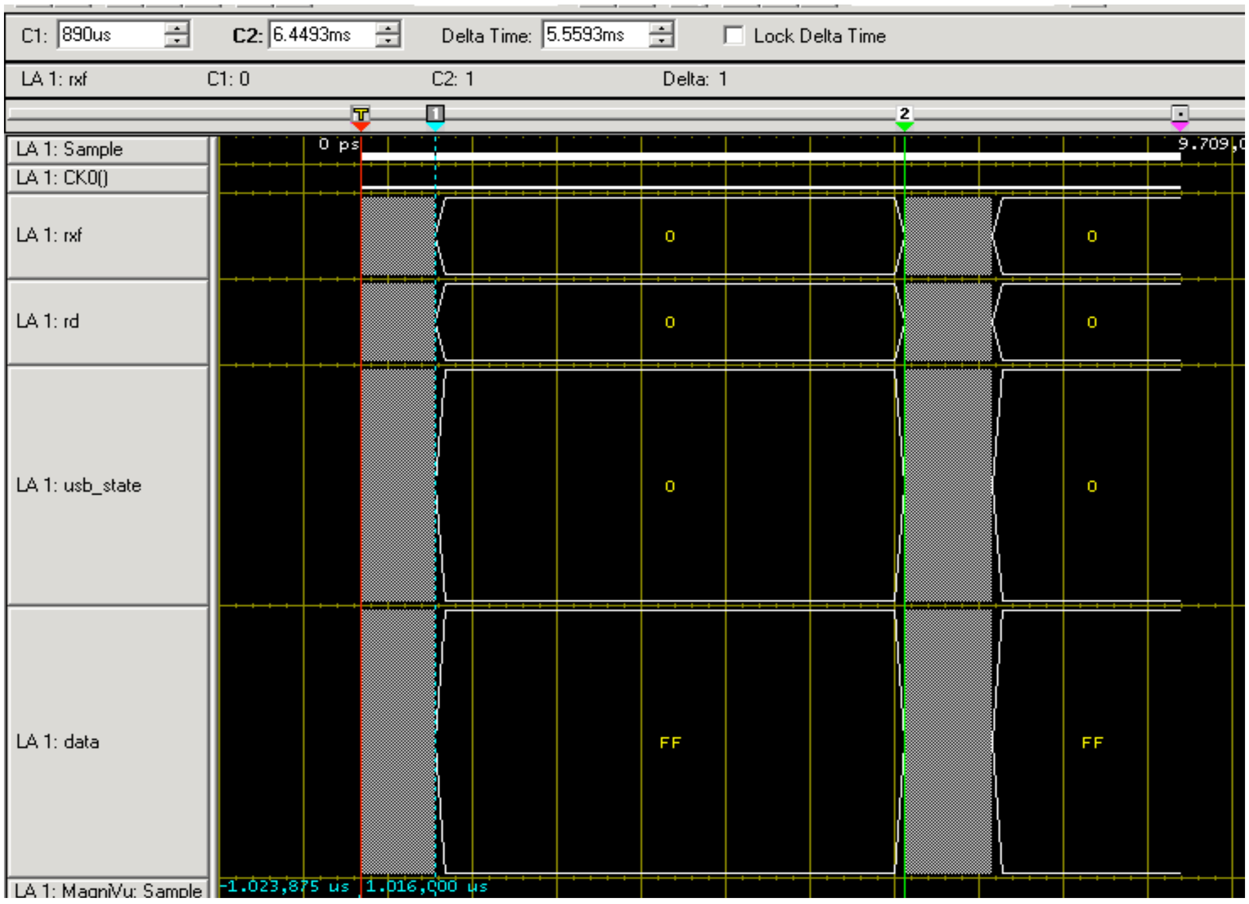

Figure 2: Large delay in USB data transmission using the Mac OS X virtual COM driver.

In order to stream audio, the transport mechanism does not need to be particularly fast, but it does need to be constant and consistent. The large delays in transmission were causing the 4096 byte on-board FIFO to empty before new data was available.

Some changes in the driver and software being used to send the data delivered much more reliable performance. After the software change was made, the optimized state machine was no longer appropriate because it was running too quickly. In figure 3 there is an image of a 5-state machine. Note the sample error in the logic analyzer. By analyzing specially crafted wave forms, it was discovered that the optimized machine was missing data bytes. This was because the machine sometimes did not wait long enough for RXF to go low before reading the next byte.

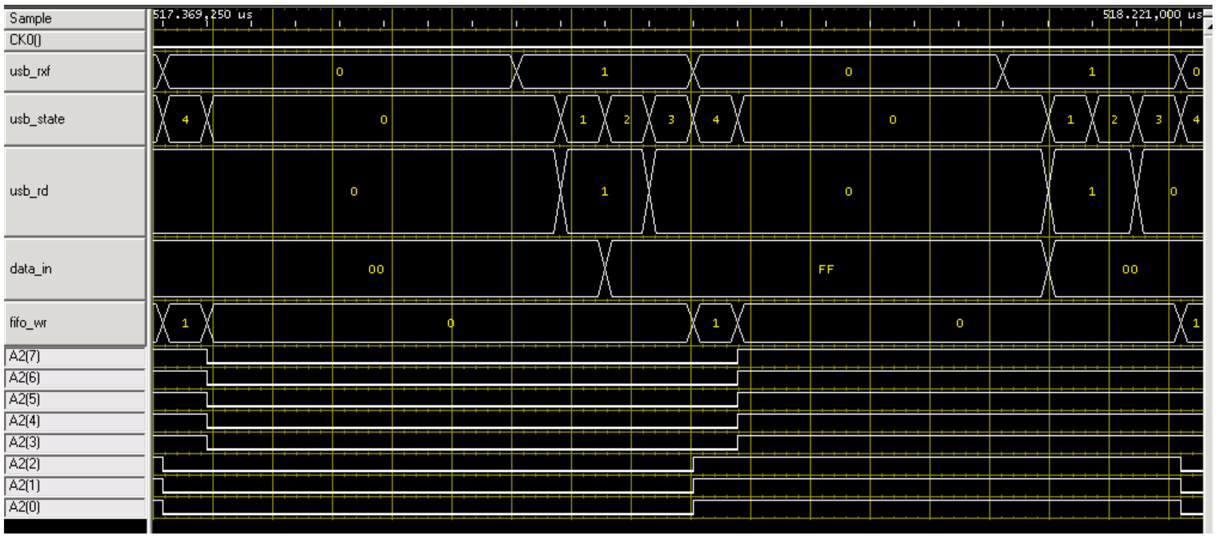

Figure 3: Sample error in optimized 5-state USB reader state-machine.

After reverting to the original 8-state machine, the USB module was able to successfully get 4 channels of 16 bit 48Khz audio in realtime from the computer.

Coefficient ROMS

The coefficient ROMs provide a simple way to side-step most of the computation involved in the Ambisonic encoding and decoding DSP. Spherical harmonic coefficients are constant for a certain location in space, so rather than having to calculate 16 products involving sins, cosines, and exponents, a python script was written to generate .coe files for each harmonic. To further simplify the matter, a decision was made to build a 2-dimensional system for the purposes of this project and not a three dimensional system. This meant that five of the coefficients were always zero and the coefficient ROMs could be kept in BRAM and not ZBT RAM. Keeping that in mind, 11 ROMs were generated using Core Gen. Each rom contains a 16-bit coefficient for every one of 4096 possible location coordinates.

Coeff_full Module

The Coeff Full module is responsible for distributing the coefficients to each encoder. A counter c_counter keeps track of the current encoder. For each value of the counter, all the coefficients are pulled out of the eleven ROMs in parallel and stored to 16-bit register outputs.

Although there are four encoders, the encoder counter has consists of 3 bits. This way, when the counter is finished counting through each encoder, its MSB goes high and a Coeffs_valid flag is set to signal the encoders to start their multiplication.

Generating Coefficients

In some ways generating the coefficients was one of the most difficult parts of this project. The quality of the coefficients affects the ability of the encoders and decoders to position virtual sources precisely. Two figures below show the difference between the gain over an entire sound field for a source placed at (8,8,0) using a weighted set of coordinates and an unweighted set of coordinates. Note that the unweighted coordinates provide a narrower source of direction, but at the cost of uneven phase over the sound field.

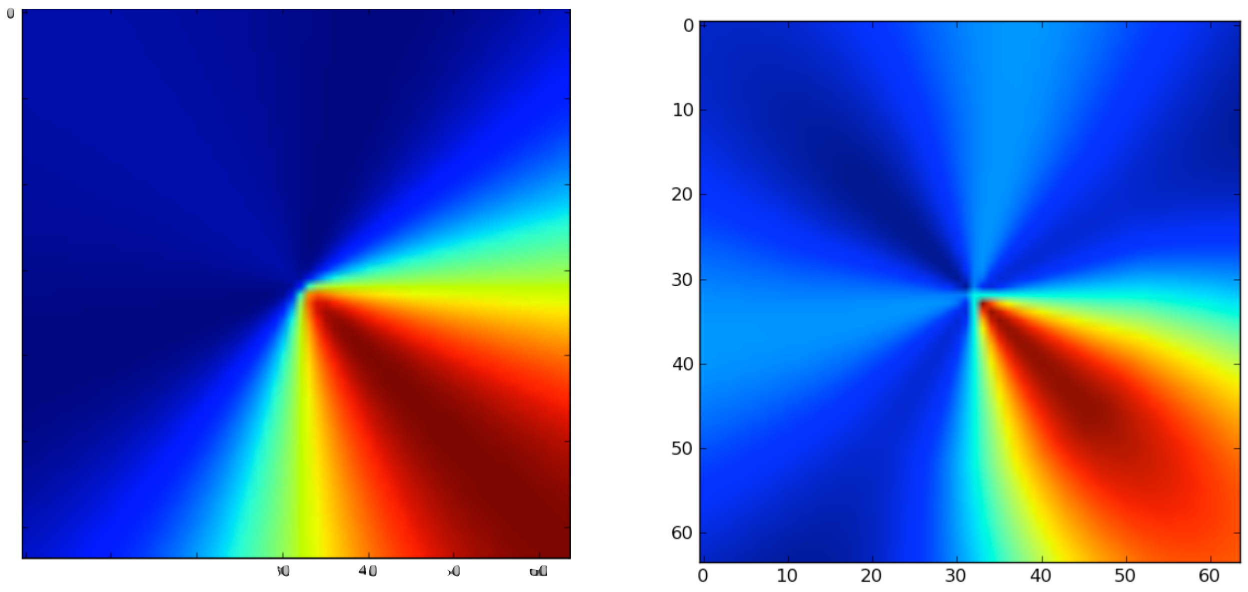

Figure 4: Weighted (left) and unweighted (right) gain distribution graphs of ambisonic samples calculated using Python coefficient generator. Dark blue is negative gain and red is positive gain.

For the final project presentation, unweighted coefficients were used. Because of the small number of speakers, this resulted in sources seeming to disappear around the 0 axes exactly between the physical sources. Using more physical sources would theoretically solve this problem.

For python code to generate coefficients, see coeff_calc.py in Appendix B.

Encoder

The enc_multiplier module is responsible for taking coefficients and multiplying incoming samples by them to create the so-called “encoded” stream. It uses a single 18x18 multiplier with a counter to process each coefficient and generate a 32-bit output representing 16bits of decimal precision. For the purposes of this project, only the 16-MSB, which represent a fraction of the original signal, are stored and passed to the summing module. This works because each coefficient is in the range [-1, 1]. Once the coefficients have been multiplied (the counter has reached 15) the encoder_valid signal is asserted after a delay of one cycle.

Summing

The summing module takes 16 outputs from each encoder device and sums the corresponding channels together. The result of this is an 18-bit signed number. However, the bits are shifted right twice to maintain an overall 16-bit width. This addition takes a single system cycle, so the summing valid signal is asserted one cycle after the encoders are valid.

Decoder

The decoder is very similar to the encoder with two exceptions. It takes 16 samples from the summing device as input which represent the encoded, summed audio stream. It also has as parameters which specify its physical location. These coordinates are used to generate coefficients via a separate static coefficients module. This module has select coefficients in a small ROM which are always available. These coefficients are used to decode the incoming signal from the summing module. Again, each of the coefficients are fed into a multiplier using a counter. However, each corresponding sample is multiplied by its respective coefficient. 16-MSB of the multiplication result is stored in a memory and when all the multiplications are complete, the contents of the memory are summed together. This sample represents a decoded version of the four input sources and it is sent to the audio driver.

Design Considerations

Because the decoder and encoder are so similar, the main logic for the decoder was written first, tested and then simplified and used for the encoder. Originally the two devices were based on a 16x32 Coregen multiplier. To simplify the modules and reduce compilation time, the extra bits of precision were stripped so that the multiplications could happen using simple mul18x18s primitives which have only one cycle of latency.

Audio Output Driver

The audio output driver is responsible for formatting the output audio correctly and passing it to the PCM-1681 chip.

DCM Manipulation

To ensure the PCM1681 runs at 48Khz, the audio subsystem should have been clocked at 36.864Mhz. This would mean that there are 768 system cycles for each clock cycle. Unforunately, it is impossible to get exactly 36.864Mhz with a single Xilinx DCM using the 27Mhz system clock. However, a using the CLKFX with M=15 and D=11 yields 36.818Mhz, which corresponds to a 47.94Khz clock speed. This speed is suitable for the project and although it results in a slightly pitch shifted output, the difference is not noticeable to the untrained ear.

The reason behind using 48Khz audio was to provide a emergency compatibility with the AC97 Codec. Most of the initial testing and development on the USB reader was completed using the internal AC97 Codec and the contingency plan in case the PCM1681 did not work was to complete the project using the labkit's AC97 onboard audio output device.

PCM1681 Clocks

The PCM-1681 was perfect for this project. Audio is passed to the chip using a four serial data lines and 3 synchronized clocks. In order to ensure the 3 clocks are synchronized they are all derived using registered counters from the audio subsystem clock running at 36Mhz.

Table 1: PCM-1681 Chip Clocks

Clock Name	Clock Speed	f_s Relative Speed	Function
audio_sclk	9.21 Mhz	$192f_s$	Chip system clock to operate oversampling
audio_bclk	2.304 Mhz	48fs	Serial Audio data is latched on the rising edge of bclk
audio_lrclk	48 Khz	f_s	When high, serial audio data is latched to left channel, when low serial audio data is latched to right channel.

Originally the audio subsystem clock was passed directly to the chip, however, because the lrclk and bclk were derived from the audio subsystem clock, the rising edges of the three clocks were not in sync. There were also concerns about sending a 36Mhz clock over a wire from the user IO pins on the FPGA because 36Mhz signal was at the very edge of the

PCM1681's timing specification. Adding a third counter fixed the sync issues and ensured that the sclk clock signal arriving at the chip was clean.

Serial Audio Output

Special care was taken to ensure that the PCM driver output serial audio correctly. The timing diagram in figure 5 shows exactly how all three clocks are synchronized. Note that the rising edge of lrclk comes at the falling edge of bclk and that bits are sent to the chip on the falling edge of bclk.

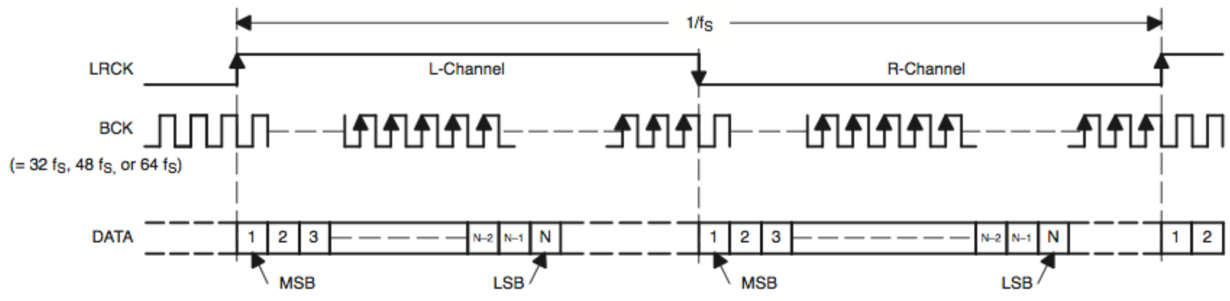

Figure 5: Timing diagram for PCM-1681 Left Justified Audio Input (http://focus.ti.com/lit/ds/symlink/pcm1681.pdf)

The audio driver was designed to send a 16-bit sample using left-justified serial audio. To accomplish this, a system of counters are used. The first counter counts the 48 ticks of audio_bclk for each audio_lrclk cycle. A send_bit signal is asserted only during the first 16 bclk ticks of each half-period of lrclk. While the send_bit is high, a second counter (bit_counter) counts down from 32 on each falling bclk edge. Combinational logic exposes the two-channel audio data one bit at a time, MSB first, according to the value of bit_counter. When send_bit is not high, the serial output values are set to zero.

The PCM1681 has four serial data inputs which accept two audio channels each.

PCM1681 Control

The PCM1681 is capable of 3 methods of control. For this project, the simplest method was used, which allows the specification of left or right justified audio and the control of muting for all outputs and a de-emphasis filter via a 4 channel parallel interface.

Video Subsystem Modules

The video subsystem is clocked at 65Mhz to allow XGA display. On the screen, a 512 by 512 pixel position map shows the locations of the four virtual sources. A blue ring represents an area outside which the gain of the sources starts to fall-off. The gain fall-off was achieved by altering the coefficient ROMs using a formula to add a constant decrease if a virtual source location is more than coordinate 20 units from the center of the map.

XGA

This code was borrowed from lab 5 to produce the necessary display timings for the UI. These values are passed to the Ambisonic UI which paints the various sprites for the position map.

Ambisonic UI

The ambisonic UI code was adapted from the lab 5 Pong UI code. Four Blobs are painted on screen along with a circular outline that represents the gain fall-off border. Their coordinates are calculated by a module which takes the ambisonic coordinate units and converts them to unsigned values from 0 to 512 for plotting on the display. Ambisonics are traditionally implemented using an Azimuth, Elevation, Distance coordinate system. However, the coefficients were generated so that they corresponded to XYZ locations in order to facilitate simple translation for the display code.

Preset Manager

The preset manager is what keeps track of each location for both the encoders and the UI. The ambisonic coordinates are stored as signed 6-bit numbers in two-dimensional memories which contain a value for each virtual source in each preset. Four of the labkit switches engage certain sources to be edited using the up-down and left-right buttons. When a source is moved, its location is updated in the active preset. Pressing one of the four push-buttons sets the active preset so that source coordinates can change abruptly.

There is an abstraction between the coordinates which are passed to the UI and encoders and the memory coordinates. This allows the memory coordinates to be changed abruptly while the UI and encoder track an interpolated value that attempts to approach the new memory value. This causes both the audio source and UI display blobs to move around smoothly.

Sync Coord Module

The coordinates are generated, altered and interpolated according to the video subsystem clock. They are published to the audio system, but to guard against meta-stability where a register is being edited or changed while it is being read, the sync coord module synchronizes the incoming coordinates to the 36Mhz audio system clock using 3 layered flip-flops.

Conclusion

Design & Testing

When taking on this project, I was unsure whether parts of it would work. I do not have significant prior experience designing analog circuits or working with FPGAs. So successfully interfacing with the USB245M module and the PCM1681 was essentially a complete shot in the dark.

To keep expectations for the project realistic, my design and testing was very closely related. The process consisted of coming up with an objective goal and then at least three alternate options and immediately beginning to work on that goal to see if it was possible.

DLP-USB245M FIFO Interface

The first stage of the design relied on getting audio into the FPGA via USB. According to the USB24M Specs[1] this should have been fairly straightforward. I set up a test system using lab 4 code to interface with the onboard AC97 device.

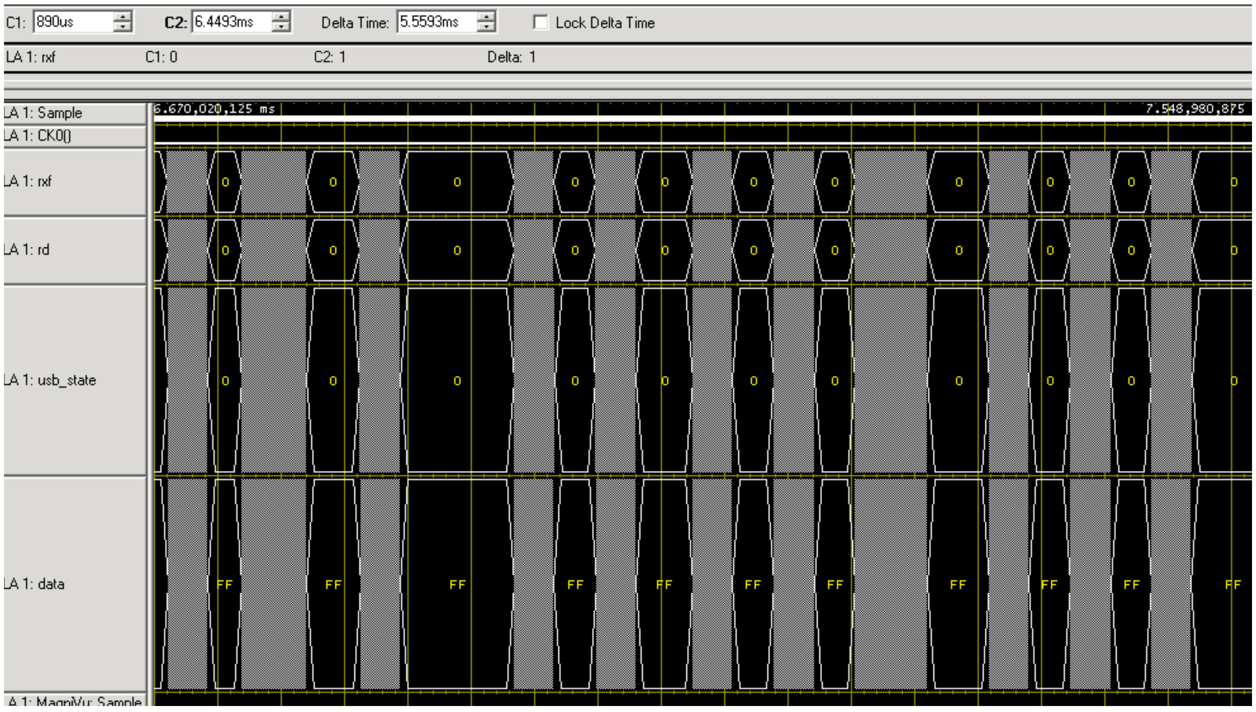

Figure 6: Small gaps in USB transmission of the initial reader state machine

[1] DLP Data Sheet (http://www.ftdichip.com/Documents/DataSheets/DLP/dlp-usb245m13.pdf)

The first hurdle was a very simple one: realizing that many of the signals were active low and that I should disable the WR parts of the chip so that it was only sending audio in one direction. Even then, the audio coming out the chip was recognizable, but garbled and distorted. After looking at the output of the DLP chip on the logic analyzer, it became evident that there were small gaps in the usb transmission. These are different from the large gaps described earlier and can be seen clearly in Figure 6.

To combat these gaps, I added a small fifo buffer to the USB reader machine. I designed the buffer to store individual bytes from the USB module and not audio samples, because it more gracefully dealt with breaks in the USB transmission, even if they were mid-sample. This required a complete redesign of the USB reader verilog module because the packets were first put into the FIFO and the removed from the FIFO and assembled into samples. If the FIFO was empty, the system halts playback until the fifo is full again. This allows the FIFO to completely fill up before playback starts and makes the system more robust to drop-outs.

After adding the FIFO, I heard my first clean audio. The next four days were spent trying to get 4 channels of audio over the USB device. After seeing the analyzer data in Figure 6, I assumed that I could not stream the audio simply because my state machine was not pulling data off the DLP chip quickly enough. I assumed that the more quickly I pulled data, the faster the data would come and I could manage to fill the FIFO before the audio system emptied it. After a day of optimizing the state-machine and learning about which control bits could reliably be combinational and which had to be registered, it became evident that gaps the in the USB transmission were growing as a I was pulling data more quickly. By the end of the optimizations, there were huge 6ms gaps in the USB transmission (seen clearly in Figure 2).

Incidentally, I also realized that there was a bug in my Verilog that was causing an offset, allowing incomplete samples to be pulled from the reader module.

The next day was spent trying to determine if I had actually reached the limit of the DLP chip and debuging my audio state problem. It seemed unlikely that the chip, which was spec'ed at 1MBps, was incapable of transmitting four channels of 16-bit 48Khz audio in real time. Some internet browsing revealed that I was using a driver which incurred a lot of overhead. The native "D2XX" driver was capable of much higher speeds but required more involved software. Luckily, I found a PyUSB python library[2] for interfacing the native D2XX driver exactly the same way I was using the existing driver.

Switching to a virtual machine running the windows edition of the native D2XX driver with the PyUSB library allowed the USB245M chip to send data to the FPGA much more quickly. Immediately, several problems surfaced having to do with optimizations where combinational values were glitching because the state-machine ran so much more quickly. I reverted back to an earlier design with more registers and more states and the sample errors disappeared.

At that point, I had usable 4 channel audio streaming in real-time over USB. The process took a total of four days.

PCM-1681 Interface

Many of the troubles integrating the PCM1681 8-channel DAC into the system came from my own inexperience. It was very exciting to learn how to surface-mount. Unfortunately, the first chip was totally dead because we did not realize that the heat-sink on the bottom of the chip would short several of the pins on the adapter board together. Chris made the second chip, which had its own solder bridge issues that may have cause yet another short.

In trying to get the chips functional, I also found several issues with my audio driver logic. The 1681 timing diagram was unclear and I realized that the numbers labeling each of the bits (in Figure 5) were not the bit numbers but the bit counts.

[2] PyUSB Native D2XX Python Library (http://bleyer.org/pyusb/)

I was sending the data LSB first and not MSB first. I also initially designed the system to be right justified, but then realized that it was only possible to specify 24-bit right justified audio using the parallel control mode. That meant that I was sending 0 to the most-significant 8-bits of the input and then the bits were in the reverse order. The audio would have been extremely quiet and entirely noise.

It took two chips before I realized that my real problem was in how I used the scope to measure the DAC output. I was inadvertently tying the common voltage output to ground thereby shorting and burning out the entire analog part of the chip. Interestingly enough, the logic outputs of the burned out chips still functioned correctly.

I had initially generated the 36Mhz system clock with the intention of sending it directly to the PCM-1681 chip. Chris helped me to realized that the chip sclk should ideally be in perfect sync with the other clocks. Rather than adding an additional layer of buffering on the clock outputs, I generated a new clock which was slower than the system clock but still an even multiple of bclk and lrclk. This had the benefit of providing a slower, cleaner, completely synchronized clock to the chip.

After solving all of those problems and connecting my third chip, the audio DAC was functional!

Audio DSP Logic

The audio DSP logic was complex to design, because I had to determine how to give the encoders and decoders access to 2.1 million coefficients. I also had to have a way to verify the coefficients were correct. In order to facilitate easy testing, I wrote the coeff_calc.py python ambisonic toolkit for generating coefficients, manipulating them and graphing a variety situations.

Initially the thought was to store the coefficient table in flash. Unfortunately, that would have been too slow as it takes several cycles to get random data out of flash and I had to retrieve 64 samples. To make the table small enough, I settled on using 6-bit signed numbers for the X and Y coordinates and a 5 bit signed number for the Z coordinate. This should have allowed 16 coefficient tables to sit comfortably in ZBT ram.

With that in mind, I set out to design a static coefficient module that I could use to test that my software simulation of the ambisonic encoder and decoder. Once that module was created I used Coregen to generate a 32 by 16 multiplier and created the encoding, decoding and summing modules. An initial test of the modules working together proved disastrous, so I set up a test rig consisting of a single encoder and decoder.

Testing that system was extremely complicated. I realized that I had large errors in my arithmetic and I should have simulated the python using a fixed point precision library. I was attempting to conserve all of my decimal accuracy and pass very large 48 and 64 bit samples from module to module. I had also failed to realize that my coefficients were designed for fixed point precision and that although the math did not necessarily look right in python, the coefficients were actually correct and needed no modification. Because I was unaware of this, I spent nearly a day trying to determine the perfect gain factors for each order of the spherical harmonics before I even started writing the verilog for the DSP.

After testing, I realized that something was badly wrong and went to talk with Chris to double check my intuitions about fixed-point precision math. Chris helped me realized that I did not need to keep 32-bits of decimal precision. In fact, I didn't ever need more than 16 bits. Keeping this in mind I rewrote and tested both the encoder and decoder and was able successfully encode and decode single samples. All DSP testing was conducted in Modelsim. I created mockup modules to send samples to the DSP chain.

Initially I attempted to encode and decode in the same location. Ideally this should have produced an identical output. Because of my gains it didn't, however, I was able to tweak them using the test DSP single encoder-decoder system and my coeff_calc script to generate verilog which could be inserted into the coeff_static module.

Once I got a single value to be essentially correct, I tested extreme cases where samples were at high negative and positive values. These tests initially failed abysmally. It was at this point that I realized the coefficients were designed for fixed point systems and I was able to take the gain factors out so that the negative and positive limits were preserved through the encoding and decoding process.

There were also several small bugs in the encoder and decoder, mostly having to do with offsets that had to be set so the multiplier output was stored in the correct location. In some cases, I added logic to delay the valid signals an extra cycle to make sure the outputs were stable before the next module tried use them.

Once I determined the correct coefficients and got both the encoder and decoder working correctly, I added the summing device and began to work with multiple encoders and decoders. The summing device required very little debugging, and I was able to compile the project and load it onto a lab kit to verify that the hardware implementation behaved correctly. Amazingly, it behaved exactly as it was supposed to.

The final step was to build a system for getting the coefficients dynamically to support moving source and the user interface. Again, I decided to make a sacrifice and use only X and Y coordinates, assuming Z was zero at all times. This meant that I could modify my coeff_calc script to generate .coe files and create ROMs for each harmonic setting the 3d harmonics statically to zero. I decided to keep the static coefficient module in the decoders.

I wrote, tested, debugged, and completed the coefficient distribution module, coeff_full and all of the ROMs on the day the project was due. I was able to compile and build a version of the complete DSP chain and run it on the lab kit with the ROMs before integrating the user interface.

User Interface

I created the user interface in parallel with all of the modules. I set up a machine and labkit across the isle and worked on it whenever the audio portions of the project were synthesizing.

In comparison to the audio aspects of the project, the UI was simple to design and debug. I re-wrote my blob_circle module from lab 5 to be more readable and support a THICKNESS parameter that specified whether the middle of the circle was filled in.

The UI was tested thoroughly on a separate labkit before being integrated with the audio subsystem. The audio and video systems connected only by a single module sync_coord, which was never tested.

System Integration

The full system miraculously worked the first time it was built in its entirely. The only testing involved connecting the output speakers in the correct order so that audio moved to the correct speaker when it was moved there on the UI display.

Final Thoughts

I wish that I had worked with a partner on this project. Although I managed to accomplish a lot and most of my goals, it would have been amazing to put a fantastic and complete user interface and spend more time optimizing the coefficients to be exactly correct, adding more decoders and adding some very basic functionality such as volume control. As it is, I will continue this project and try to design better sounding encoders and decoders.

References

- Dave Malham: Space in Music, Music in Space:
  http://www.york.ac.uk/inst/mustech/3d_audio/higher_order_ambisonics.pdf
- PyUSB native python wrapper for D2XX drivers
  http://bleyer.org/pyusb/
- DLP Devices USB245M FTDI USB FIFO Datasheet
  http://www.ftdichip.com/Documents/DataSheets/DLP/dlp-usb245m13.pdf
- Texas Instruments PCM-1681 Chip
  http://focus.ti.com/lit/ds/symlink/pcm1681.pdf

Appendix A: Verilog Implementation

```
`default_nettype none

///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes, 6.111 staff
//
///////////////////////////////////////////////////////////////////////////////

module labkit(
  // Remove comment from any signals you use in your design!

  // AC97
  /*
  output wire beep, audio_reset_b, ac97_synch, ac97_sdata_out,
  input wire ac97_bit_clock, ac97_sdata_in,
  */

  // VGA
  output wire [7:0] vga_out_red, vga_out_green, vga_out_blue,
  output wire vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync, vga_out_vsync,


  // NTSC OUT
  /*
  output wire [9:0] tv_out_ycrcb,
  output wire tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
  output wire tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
  output wire tv_out_subcar_reset;
  */

  // NTSC IN
  /*
  input wire [19:0] tv_in_ycrcb,
  input wire tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
  output wire tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,
  inout wire tv_in_i2c_data,
  */

  // ZBT RAMS
  /*
  inout wire [35:0] ram0_data,
  output wire [18:0] ram0_address,
  output wire ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b,
  output wire [3:0] ram0_bwe_b,
  inout wire [35:0]ram1_data,
  output wire [18:0]ram1_address,
  output wire ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b,
  output wire [3:0] ram1_bwe_b,
  input wire clock_feedback_in,
  output wire clock_feedback_out,
  */

  // FLASH
  /*
  inout wire [15:0] flash_data,
  output wire [23:0] flash_address,
  output wire flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b,
  input  wire flash_sts,
  */

  // RS232
  /*
  output wire rs232_txd, rs232_rts,
  input wire rs232_rxd, rs232_cts,
  */

  // PS2
  //input wire mouse_clock, mouse_data,
  //input wire keyboard_clock, keyboard_data,

  // FLUORESCENT DISPLAY
  /*
  output wire disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b,
  input wire disp_data_in,
  output wire disp_data_out,
  */

  // SYSTEM ACE
  /*
  inout wire [15:0] systemace_data,
  output wire [6:0] systemace_address,
  output wire systemace_ce_b, systemace_we_b, systemace_oe_b,
  input wire systemace_irq, systemace_mpbrdy,
  */

  // BUTTONS, SWITCHES, LEDS
  input wire button0,
  input wire button1,
  input wire button2,
  input wire button3,
  input wire button_enter,
  input wire button_right,
  input wire button_left,
  input wire button_down,
  input wire button_up,
  input wire [7:0] switch,
  output wire [7:0] led,

  // USER CONNECTORS, DAUGHTER CARD, LOGIC ANALYZER
  //inout wire [31:0] user1,
  //inout wire [31:0] user2,
  inout wire [31:0] user3, //pcm1681
  inout wire [31:0] user4, //dlp usb chip
  //inout wire [43:0] daughtercard,
  //output wire [15:0] analyzer1_data, output wire analyzer1_clock,
  //output wire [15:0] analyzer2_data, output wire analyzer2_clock,
  //output wire [15:0] analyzer3_data, output wire analyzer3_clock,
  //output wire [15:0] analyzer4_data, output wire analyzer4_clock,

  // CLOCKS
  //input wire clock1,
  //input wire clock2,
  input wire clock_27mhz
);

  ////////////////////////////////////////////////////////////////////////////
  //
  // 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.
  //
  ////////////////////////////////////////////////////////////////////////////


/*
 ***********************************************
 * SYSTEM STUFF: 36.862 Mhz System Clock and
 *    initial reset signal
 ***********************************************
 */
 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;

 //First generate a 36.864Mhz Clock to use as a system clock
  wire clock_36mhz_unbuf,clock_36mhz;
  DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_36mhz_unbuf));
  // synthesis attribute CLKFX_DIVIDE of vclk1 is 11
  // synthesis attribute CLKFX_MULTIPLY of vclk1 is 15
  // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
  // synthesis attribute CLKIN_PERIOD of vclk1 is 37
  BUFG vclk2(.O(clock_36mhz),.I(clock_36mhz_unbuf));

  wire clock_65mhz_unbuf,clock_65mhz;
  DCM vclk3(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
  // synthesis attribute CLKFX_DIVIDE of vclk3 is 10
  // synthesis attribute CLKFX_MULTIPLY of vclk3 is 24
  // synthesis attribute CLK_FEEDBACK of vclk3 is NONE
  // synthesis attribute CLKIN_PERIOD of vclk3 is 37
  BUFG vclk4(.O(clock_65mhz),.I(clock_65mhz_unbuf));

/*
 ***********************************************
 * VIDEO!: Universal audio signals
 ***********************************************
 */

       wire [7:0] switch_clean = ~switch;

  // UP and DOWN buttons for pong paddle
  wire up,down,left,right,b_enter;
       debounce db1(.reset(reset),.clock(clock_65mhz),.noisy(~button_enter),.clean(b_enter));
  debounce db2(.reset(reset),.clock(clock_65mhz),.noisy(~button_up),.clean(up));
  debounce db3(.reset(reset),.clock(clock_65mhz),.noisy(~button_down),.clean(down));
  debounce db8(.reset(reset),.clock(clock_65mhz),.noisy(~button_left),.clean(left));
  debounce db9(.reset(reset),.clock(clock_65mhz),.noisy(~button_right),.clean(right));


  // generate basic XVGA video signals
  wire [10:0] hcount;
  wire [9:0]  vcount;
  wire hsync,vsync,blank;
  xvga xvga1(.vclock(clock_65mhz),.hcount(hcount),.vcount(vcount),
             .hsync(hsync),.vsync(vsync),.blank(blank));

        // debounce button input
        wire bb0,bb1,bb2,bb3;
        debounce db4(.reset(reset),.clock(clock_65mhz),.noisy(~button0),.clean(bb0));
        debounce db5(.reset(reset),.clock(clock_65mhz),.noisy(~button1),.clean(bb1));
        debounce db6(.reset(reset),.clock(clock_65mhz),.noisy(~button2),.clean(bb2));
        debounce db7(.reset(reset),.clock(clock_65mhz),.noisy(~button3),.clean(bb3));

  // feed XVGA signals to ui
  wire [2:0] pixel;
  wire phsync,pvsync,pblank;
       wire [1:0] active_preset;

  //virtual source coordinates
       wire [16:0] vcoord0, vcoord1, vcoord2, vcoord3; //video sync
       wire [16:0] acoord0, acoord1, acoord2, acoord3; //audio sync
  ambisonic_ui pg(.vclock(clock_65mhz),.reset(reset),
               .up(up),.down(down),.pspeed(switch[7:4]),
               .hcount(hcount),.vcount(vcount),
               .hsync(hsync),.vsync(vsync),.blank(blank),
               .phsync(phsync),.pvsync(pvsync),.pblank(pblank),.pixel(pixel),
               .button0(bb0), .button1(bb1), .button2(bb2), .button3(bb3),
               .left(left), .right(right), .b_enter(b_enter),
               .active_preset(active_preset), .switch_clean(switch_clean),
               .coord0(vcoord0), .coord1(vcoord1), .coord2(vcoord2), .coord3(vcoord3));

       sync_coord sc0(.clock(clock_36mhz), .acoord(acoord0), .vcoord(vcoord0));
       sync_coord sc1(.clock(clock_36mhz), .acoord(acoord1), .vcoord(vcoord1));
       sync_coord sc2(.clock(clock_36mhz), .acoord(acoord2), .vcoord(vcoord2));
       sync_coord sc3(.clock(clock_36mhz), .acoord(acoord3), .vcoord(vcoord3));

  // 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 [2:0] rgb;
  reg b,hs,vs;
  always @(posedge clock_65mhz) begin
     if (switch[1:0] == 2'b01) begin
        // 1 pixel outline of visible area (white)
        hs <= hsync;
        vs <= vsync;
        b <= blank;
        rgb <= (hcount==0 | hcount==1023 | vcount==0 | vcount==767) ? 7 : 0;
     end else if (switch[1:0] == 2'b10) begin
        // color bars
        hs <= hsync;
        vs <= vsync;
        b <= blank;
        rgb <= hcount[8:6];
     end else begin
        // default: pong
        hs <= phsync;
        vs <= pvsync;
        b <= pblank;
        rgb <= pixel;
     end
  end

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

/*
 ***********************************************
 * HOUSE KEEPING: Universal audio signals
 ***********************************************
 */

       //audio pulse every 48Khz
       wire ready;

       //Data sent to PCM1681
       reg [31:0] to_pcm1681_0;
       reg [31:0] to_pcm1681_1;
       reg [31:0] to_pcm1681_2;
       reg [31:0] to_pcm1681_3;
       //4 16-bit channels from USB
       wire [63:0] audio_data;

/*
 ***********************************************
 * COEFFICIENT ROM: Ambisonic 3rd order decoders
 ***********************************************
 */
 wire [15:0] ca0, ca1, ca2, ca3, ca4, ca5, ca6, ca7, ca8, ca9, caa, cab, cac, cad, cae, caf;
 wire [15:0] cb0, cb1, cb2, cb3, cb4, cb5, cb6, cb7, cb8, cb9, cba, cbb, cbc, cbd, cbe, cbf;
 wire [15:0] cc0, cc1, cc2, cc3, cc4, cc5, cc6, cc7, cc8, cc9, cca, ccb, ccc, ccd, cce, ccf;
 wire [15:0] cd0, cd1, cd2, cd3, cd4, cd5, cd6, cd7, cd8, cd9, cda, cdb, cdc, cdd, cde, cdf;
 wire coeffs_valid;
       coeff_full coeff_rom (
   .coord0(acoord0), .coord1(acoord1), .coord2(acoord2), .coord3(acoord3),
   .clock(clock_36mhz), .reset(reset), .ready(ready),
        .a0(ca0), .a1(ca1), .a2(ca2), .a3(ca3), .a4(ca4), .a5(ca5), .a6(ca6), .a7(ca7),
        .a8(ca8), .a9(ca9), .aa(caa), .ab(cab), .ac(cac), .ad(cad), .ae(cae), .af(caf),
   .b0(cb0), .b1(cb1), .b2(cb2), .b3(cb3), .b4(cb4), .b5(cb5), .b6(cb6), .b7(cb7),
   .b8(cb8), .b9(cb9), .ba(cba), .bb(cbb), .bc(cbc), .bd(cbd), .be(cbe), .bf(cbf),
   .c0(cc0), .c1(cc1), .c2(cc2), .c3(cc3), .c4(cc4), .c5(cc5), .c6(cc6), .c7(cc7),
   .c8(cc8), .c9(cc9), .ca(cca), .cb(ccb), .cc(ccc), .cd(ccd), .ce(cce), .cf(ccf),
   .d0(cd0), .d1(cd1), .d2(cd2), .d3(cd3), .d4(cd4), .d5(cd5), .d6(cd6), .d7(cd7),
   .d8(cd8), .d9(cd9), .da(cda), .db(cdb), .dc(cdc), .dd(cdd), .de(cde), .df(cdf),
   .coeffs_valid(coeffs_valid)
   );

/*
 ***********************************************
 * ENCODER: Ambisonic 3rd order decoders
 ***********************************************
 */

 wire [15:0] a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, aa, ab, ac, ad, ae, af;
 wire [15:0] b0, b1, b2, b3, b4, b5, b6, b7, b8, b9, ba, bb, bc, bd, be, bf;
 wire [15:0] c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, ca, cb, cc, cd, ce, cf;
 wire [15:0] d0, d1, d2, d3, d4, d5, d6, d7, d8, d9, da, db, dc, dd, de, df;
 wire enc0r, enc1r, enc2r, enc3r;
 enc_multiplier enc0 (
   .sample(audio_data[15:0]),
   .clock(clock_36mhz),
   .reset(reset),
   .ready(ready),
   .e0(a0), .e1(a1), .e2(a2), .e3(a3), .e4(a4), .e5(a5), .e6(a6), .e7(a7),
   .e8(a8), .e9(a9), .ea(aa), .eb(ab), .ec(ac), .ed(ad), .ee(ae), .ef(af),
        .c0(ca0), .c1(ca1), .c2(ca2), .c3(ca3), .c4(ca4), .c5(ca5), .c6(ca6), .c7(ca7),
   .c8(ca8), .c9(ca9), .ca(caa), .cb(cab), .cc(cac), .cd(cad), .ce(cae), .cf(caf),
   .encoder_valid(enc0r), .c_valid(coeffs_valid)
   );

 enc_multiplier enc1 (
   .sample(audio_data[31:16]),
   .clock(clock_36mhz),
   .reset(reset),
   .ready(ready),
   .e0(b0), .e1(b1), .e2(b2), .e3(b3), .e4(b4), .e5(b5), .e6(b6), .e7(b7),
   .e8(b8), .e9(b9), .ea(ba), .eb(bb), .ec(bc), .ed(bd), .ee(be), .ef(bf),
        .c0(cb0), .c1(cb1), .c2(cb2), .c3(cb3), .c4(cb4), .c5(cb5), .c6(cb6), .c7(cb7),
   .c8(cb8), .c9(cb9), .ca(cba), .cb(cbb), .cc(cbc), .cd(cbd), .ce(cbe), .cf(cbf),
   .encoder_valid(enc1r), .c_valid(coeffs_valid)
   );

 enc_multiplier enc2 (
   .sample(audio_data[47:32]),
   .clock(clock_36mhz),
   .reset(reset),
   .ready(ready),
   .e0(c0), .e1(c1), .e2(c2), .e3(c3), .e4(c4), .e5(c5), .e6(c6), .e7(c7),
   .e8(c8), .e9(c9), .ea(ca), .eb(cb), .ec(cc), .ed(cd), .ee(ce), .ef(cf),
        .c0(cc0), .c1(cc1), .c2(cc2), .c3(cc3), .c4(cc4), .c5(cc5), .c6(cc6), .c7(cc7),
   .c8(cc8), .c9(cc9), .ca(cca), .cb(ccb), .cc(ccc), .cd(ccd), .ce(cce), .cf(ccf),
   .encoder_valid(enc2r), .c_valid(coeffs_valid)
   );
```

```
  enc_multiplier enc3 (
    .sample(audio_data[63:48]),
    .clock(clock_36mhz),
    .reset(reset),
    .ready(ready),
    .e0(d0), .e1(d1), .e2(d2), .e3(d3), .e4(d4), .e5(d5), .e6(d6), .e7(d7),
    .e8(d8), .e9(d9), .ea(da), .eb(db), .ec(dc), .ed(dd), .ee(de), .ef(df),
         .c0(cd0), .c1(cd1), .c2(cd2), .c3(cd3), .c4(cd4), .c5(cd5), .c6(cd6), .c7(cd7),
    .c8(cd8), .c9(cd9), .ca(cda), .cb(cdb), .cc(cdc), .cd(cdd), .ce(cde), .cf(cdf),
    .encoder_valid(enc3r), .c_valid(coeffs_valid)
    );


 /*
  ***********************************************
  * SUMMING: Ambisonic 3rd order decoders
  ***********************************************
  */

  wire [17:0] s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, sa, sb, sc, sd, se, sf;
  wire summing_valid;

summing sum (
    .a0(a0), .b0(b0), .c0(c0), .d0(d0),
         .a1(a1), .b1(b1), .c1(c1), .d1(d1),
    .a2(a2), .b2(b2), .c2(c2), .d2(d2),
    .a3(a3), .b3(b3), .c3(c3), .d3(d3),
    .a4(a4), .b4(b4), .c4(c4), .d4(d4),
    .a5(a5), .b5(b5), .c5(c5), .d5(d5),
    .a6(a6), .b6(b6), .c6(c6), .d6(d6),
    .a7(a7), .b7(b7), .c7(c7), .d7(d7),
    .a8(a8), .b8(b8), .c8(c8), .d8(d8),
    .a9(a9), .b9(b9), .c9(c9), .d9(d9),
    .aa(aa), .ba(ba), .ca(ca), .da(da),
    .ab(ab), .bb(bb), .cb(cb), .db(db),
    .ac(ac), .bc(bc), .cc(cc), .dc(dc),
    .ad(ad), .bd(bd), .cd(cd), .dd(dd),
    .ae(ae), .be(be), .ce(ce), .de(de),
    .af(af), .bf(bf), .cf(cf), .df(df),
    .clock(clock_36mhz),
    .reset(reset),
    .ready(ready),
    .a_valid(enc0r),
    .b_valid(enc1r),
    .c_valid(enc2r),
    .d_valid(enc3r),
    .o0(s0), .o1(s1), .o2(s2), .o3(s3), .o4(s4), .o5(s5), .o6(s6), .o7(s7),
         .o8(s8), .o9(s9), .oa(sa), .ob(sb), .oc(sc), .od(sd), .oe(se), .of(sf),
    .summing_valid(summing_valid)
    );

 /*
  ***********************************************
  * DECODER: Ambisonic 3rd order decoders
  ***********************************************
  */

  //decoder ready signals
  wire dec0r, dec1r, dec2r, dec3r;
  wire signed [23:0] chan1, chan2, chan3, chan4;

        decoder #(.X(6'sd31), .Y(6'sd31), .Z(5'sd0)) dec0 (
    .s0(s0), .s1(s1), .s2(s2), .s3(s3), .s4(s4), .s5(s5), .s6(s6), .s7(s7),
    .s8(s8), .s9(s9), .sa(sa), .sb(sb), .sc(sc), .sd(sd), .se(se), .sf(sf),
    .summing_valid(summing_valid),
    .ready(ready),
    .clock(clock_36mhz),
    .reset(reset),
    .audio_out(chan1),
    .decoder_valid(dec0r)
    );

        decoder #(.X(6'sd31), .Y(-6'sd31), .Z(5'sd0)) dec1 (
    .s0(s0), .s1(s1), .s2(s2), .s3(s3), .s4(s4), .s5(s5), .s6(s6), .s7(s7),
    .s8(s8), .s9(s9), .sa(sa), .sb(sb), .sc(sc), .sd(sd), .se(se), .sf(sf),
    .summing_valid(summing_valid),
    .ready(ready),
    .clock(clock_36mhz),
    .reset(reset),
    .audio_out(chan2),
    .decoder_valid(dec1r)
    );

        decoder #(.X(-6'sd31), .Y(6'sd31), .Z(5'sd0)) dec2 (
    .s0(s0), .s1(s1), .s2(s2), .s3(s3), .s4(s4), .s5(s5), .s6(s6), .s7(s7),
    .s8(s8), .s9(s9), .sa(sa), .sb(sb), .sc(sc), .sd(sd), .se(se), .sf(sf),
    .summing_valid(summing_valid),
    .ready(ready),
    .clock(clock_36mhz),
    .reset(reset),
    .audio_out(chan3),
    .decoder_valid(dec2r)
    );

        decoder #(.X(-6'sd31), .Y(-6'sd31), .Z(5'sd0)) dec3 (
    .s0(s0), .s1(s1), .s2(s2), .s3(s3), .s4(s4), .s5(s5), .s6(s6), .s7(s7),
    .s8(s8), .s9(s9), .sa(sa), .sb(sb), .sc(sc), .sd(sd), .se(se), .sf(sf),
    .summing_valid(summing_valid),
    .ready(ready),
    .clock(clock_36mhz),
    .reset(reset),
    .audio_out(chan4),
    .decoder_valid(dec3r)
    );

        //On the ready pulse, present audio to
        //the pcm1681 driver
        always @(posedge clock_36mhz) begin
                if (reset) begin
                                to_pcm1681_0 <= 0;
                                to_pcm1681_1 <= 0;
                                to_pcm1681_2 <= 0;
                                to_pcm1681_3 <= 0;
                end
                else if (ready) begin
                                to_pcm1681_0 <= {chan1[15:0], chan2[15:0]};
                                to_pcm1681_1 <= {chan3[15:0], chan4[15:0]};
                                to_pcm1681_2 <= {audio_data[15:0], a0[15:0]};
                                to_pcm1681_3 <= {s0[15:0], chan1[15:0]};
                end
        end

        /*
         ***********************************************
         * PCM-1681 DRIVER: generates appropriate clocks
         *    and recieves audio for the 8 channel DAC
         ***********************************************
         */

        wire audio_sclk, audio_clock, audio_bclk, demp, mute;
        wire [1:0] fmt;
        wire [3:0] serial_data;

        //pcm1681 driver
        pcm1681_driver audio_driver     (
    .clock_36mhz(clock_36mhz),
        .audio_12(to_pcm1681_0),
        .audio_34(to_pcm1681_1),
        .audio_56(to_pcm1681_2),
        .audio_78(to_pcm1681_3),
        .audio_sclk(audio_sclk),
        .audio_lrclk(audio_clock),
        .audio_bclk(audio_bclk),
        .serial_data(serial_data),
        .fmt(fmt),
        .demp(demp),
        .mute(mute),
        .reset(reset),
        .ready(ready)
        );

    //Patch the chip
    assign user3[1:0] = fmt;
    assign user3[2] = demp;
    assign user3[3] = mute;
    assign user3[4] = audio_sclk;
    assign user3[5] = serial_data[3];
    assign user3[6] = audio_bclk;
    assign user3[7] = audio_clock;
    assign user3[8] = serial_data[2];
    assign user3[9] = serial_data[1];
    assign user3[10] = serial_data[0];

    assign user3[31:11] = 0;

    /*
     ***********************************************
     * USB READER: gets audio from the usb fifo
     ***********************************************
     */

    //First patch our input devices
    assign user4[9] = 1; //WR high to disable send
    assign user4[31:12] = 0; //These pins are not used
    assign user4[10] = 0; //This is not used either

    // USB reader patch
    wire rd;
    wire rxf = ~user4[11];
    assign user4[8] = ~rd;
    wire [3:0] status;
    wire buffer_full;
    wire buffer_empty;
    wire be_led;
    wire bf_led;

    assign led[7] = ~bf_led; //buffer_full
    assign led[6] = ~be_led; //buffer_empty
    assign led[5:3] = 4'b1111;
    assign led[2:0] = status[2:0];

    wire [7:0] fifo_data;
    wire reader_sample_ready;
    wire [2:0] reader_byte_num;

    wire [7:0] usb_data_in = switch[7] ? 8'b11111111 : user4[7:0];
    wire fifo_wr;

usb_reader usb(.rxf(rxf),
               .ready(ready),
               .clock(clock_36mhz),
               .reset(reset),
               .data(usb_data_in),
               .audio_out_data(audio_data),
                  .rd(rd),
                  .led(status),
                  .fifo_full(buffer_full),
                  .fifo_out(fifo_data),
                  .sample_complete(reader_sample_ready),
                  .byte_num(reader_byte_num),
                  .fifo_wr(fifo_wr));

        //These leds will stay illuminated if the buffer empties or if it is full
        second_notify buffer_full_notify(.clock(clock_36mhz),
                                                                                                 .reset(reset),
                                                                                                 .trigger(buffer_fu
                                                                                                 .second(bf_led));
        second_notify buffer_empty_notify(.clock(clock_36mhz),
                                                                                                 .reset(reset),
                                                                                                 .trigger(status[3]
                                                                                                 .second(be_led));

        /*
         ***********************************************
         * ANALYZER PATCH:: How to see what's going on!
         ***********************************************
         */

                /*
        assign analyzer1_data = switch[0] ? to_pcm1681_0[15:0] : {fifo_data,




        assign analyzer3_data = switch[1] ? to_pcm1681_0[31:16] : {3'b0, enc0r, dec0r, fmt, mute, demp ,


        //assign analyzer3_clock = ready;
        //assign analyzer1_clock = clock_36mhz;



endmodule
```

```
///////////////////////////////////////////////////////////////////////////////
//
// Verilog equivalent to a BRAM, tools will infer the right thing!
// number of locations = 1<<LOGSIZE, width in bits = WIDTH.
// default is a 16K x 1 memory.
//
///////////////////////////////////////////////////////////////////////////////

module mybram #(parameter LOGSIZE=14, WIDTH=1)
              (input wire [LOGSIZE-1:0] addr,
               input wire clk,
               input wire [WIDTH-1:0] din,
               output reg [WIDTH-1:0] dout,
               input wire we);
   // let the tools infer the right number of BRAMs
   (* ram_style = "block" *)
   reg [WIDTH-1:0] mem[(1<<LOGSIZE)-1:0];
   always @(posedge clk) begin
     if (we) mem[addr] <= din;
     dout <= mem[addr];
   end
endmodule

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

module debounce (
  input wire reset, clock, noisy,
  output reg clean
);
  reg [18:0] count;
  reg new;

  always @(posedge clock)
    if (reset) begin
      count <= 0;
      new <= noisy;
      clean <= noisy;
    end
    else if (noisy != new) begin
      // noisy input changed, restart the .01 sec clock
      new <= noisy;
      count <= 0;
    end
    else if (count == 270000)
      // noisy input stable for .01 secs, pass it along!
      clean <= new;
    else
      // waiting for .01 sec to pass
      count <= count+1;

endmodule

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

module lab4audio (
  input wire clock_27mhz,
  input wire reset,
  input wire [4:0] volume,
  output wire [7:0] audio_in_data,
  input wire [39:0] audio_out_data,
  output wire ready,
  output reg audio_reset_b,   // ac97 interface signals
  output wire ac97_sdata_out,
  input wire ac97_sdata_in,
  output wire ac97_synch,
  input wire ac97_bit_clock
);

  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;

  // wait a little before enabling the AC97 codec
  reg [9:0] reset_count;
  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(.ready(ac97_ready),
            .command_address(command_address),
            .command_data(command_data),
            .command_valid(command_valid),
            .left_data(left_out_data), .left_valid(1'b1),
            .right_data(right_out_data), .right_valid(1'b1),
            .left_in_data(left_in_data), .right_in_data(right_in_data),
            .ac97_sdata_out(ac97_sdata_out),
            .ac97_sdata_in(ac97_sdata_in),
            .ac97_synch(ac97_synch),
            .ac97_bit_clock(ac97_bit_clock));

  // ready: one cycle pulse synchronous with clock_27mhz
  reg [2:0] ready_sync;
  always @ (posedge clock_27mhz) ready_sync <= {ready_sync[1:0], ac97_ready};
  assign ready = ready_sync[1] & ~ready_sync[2];

  reg [39:0] out_data;
  always @ (posedge clock_27mhz)
    if (ready) out_data <= audio_out_data;
  assign audio_in_data = left_in_data[19:12];
  assign left_out_data = out_data[19:0];
  assign right_out_data = out_data[39:20];

  // generate repeating sequence of read/writes to AC97 registers
  ac97commands cmds(.clock(clock_27mhz), .ready(ready),
                    .command_address(command_address),
                    .command_data(command_data),
                    .command_valid(command_valid),
                    .volume(volume),
                    .source(3'b000));     // mic
endmodule

// assemble/disassemble AC97 serial frames
module ac97 (
  output reg ready,
  input wire [7:0] command_address,
  input wire [15:0] command_data,
  input wire command_valid,
  input wire [19:0] left_data,
  input wire left_valid,
  input wire [19:0] right_data,
  input wire right_valid,
  output reg [19:0] left_in_data, right_in_data,
  output reg ac97_sdata_out,
  input wire ac97_sdata_in,
  output reg ac97_synch,
  input wire ac97_bit_clock
);
  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;

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

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

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

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

    // Generate the ready signal
    if (bit_count == 128)
      ready <= 1'b1;
    if (bit_count == 2)
      ready <= 1'b0;

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

    if ((bit_count >= 0) && (bit_count <= 15))
      // Slot 0: Tags
      case (bit_count[3:0])
        4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
        4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
        4'h2: ac97_sdata_out <= l_cmd_v;   // Command data valid
        4'h3: ac97_sdata_out <= l_left_v;  // Left data valid
        4'h4: ac97_sdata_out <= l_right_v; // Right data valid
        default: ac97_sdata_out <= 1'b0;
      endcase
    else if ((bit_count >= 16) && (bit_count <= 35))
      // Slot 1: Command address (8-bits, left justified)
      ac97_sdata_out <= l_cmd_v ? l_cmd_addr[35-bit_count] : 1'b0;
    else if ((bit_count >= 36) && (bit_count <= 55))
      // Slot 2: Command data (16-bits, left justified)
      ac97_sdata_out <= l_cmd_v ? l_cmd_data[55-bit_count] : 1'b0;
    else if ((bit_count >= 56) && (bit_count <= 75)) begin
      // Slot 3: Left channel
      ac97_sdata_out <= l_left_v ? l_left_data[19] : 1'b0;
      l_left_data <= { l_left_data[18:0], l_left_data[19] };
    end
    else if ((bit_count >= 76) && (bit_count <= 95))
      // Slot 4: Right channel
      ac97_sdata_out <= l_right_v ? l_right_data[95-bit_count] : 1'b0;
    else
      ac97_sdata_out <= 1'b0;

    bit_count <= bit_count+1;
  end // always @ (posedge ac97_bit_clock)

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

// issue initialization commands to AC97
module ac97commands (
  input wire clock,
  input wire ready,
  output wire [7:0] command_address,
  output wire [15:0] command_data,
  output reg command_valid,
  input wire [4:0] volume,
  input wire [2:0] source
);
  reg [23:0] command;

  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;
      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

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

module tone750hz (
  input wire clock,
  input wire ready,
  output reg [19:0] pcm_data
);
   reg [8:0] index;

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

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

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

////////////////////////////////////////////////////////////////////////////////
//
// generate a pulse for a certian number of clock cycles if a signal goes high
//
////////////////////////////////////////////////////////////////////////////////

module second_notify #(parameter TIME_WIDTH=24) (input clock, reset, trigger, output second);
        reg [TIME_WIDTH-1:0] counter;
        reg [TIME_WIDTH-1:0] new_counter;

        always @* new_counter = trigger ? counter + 1 : reset | counter == 0 ? 0 : counter + 1;
        always @(posedge clock) counter <= new_counter;

        assign second = (counter != 0);
```

```
endmodule //second_notify
```

```
`timescale 1ns / 1ps

module coeff_static
  (input [16:0] coord,
   input clock, reset,
   output signed [15:0] c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, ca, cb, cc, cd, ce, cf,
        output coeffs_valid
   );

        assign coeffs_valid = 1;

   reg [5:0]     index_select;
   reg [15:0]    c [15:0][31:0];

   assign        c0 = c[0][index_select];
   assign        c1 = c[1][index_select];
   assign        c2 = c[2][index_select];
   assign        c3 = c[3][index_select];
   assign        c4 = c[4][index_select];
   assign        c5 = c[5][index_select];
   assign        c6 = c[6][index_select];
   assign        c7 = c[7][index_select];
   assign        c8 = c[8][index_select];
   assign        c9 = c[9][index_select];
   assign        ca = c[10][index_select];
   assign        cb = c[11][index_select];
   assign        cc = c[12][index_select];
   assign        cd = c[13][index_select];
   assign        ce = c[14][index_select];
   assign        cf = c[15][index_select];

   //This module provides staic coefficients
   always @* begin
      case (coord)
        default: index_select = 0;

                {6'sd31, 6'sd31, 5'sd0}: index_select =0;
                {6'sd31, -6'sd31, 5'sd0}: index_select =1;
                {-6'sd31, 6'sd31, 5'sd0}: index_select =2;
                {-6'sd31, -6'sd31, 5'sd0}: index_select =3;

      endcase // case(coord)
   end


   //Here we define the staic constants
   always @(posedge clock) begin
      if (reset) begin

                //Generated Preset 0
                //Location: 31 31 0
                c[0][0] <= 16'sh55fb;
                c[1][0] <= 16'sh55fb;
                c[2][0] <= 16'sh55fb;
                c[3][0] <= 16'sh0;
                c[4][0] <= -16'sh3ccc;
                c[5][0] <= 16'sh0;
                c[6][0] <= 16'sh0;
                c[7][0] <= 16'sh0;
                c[8][0] <= 16'sh7999;
                c[9][0] <= 16'sh0;
                c[10][0] <= -16'sh3e70;
                c[11][0] <= -16'sh3e70;
                c[12][0] <= 16'sh0;
                c[13][0] <= 16'sh0;
                c[14][0] <= -16'sh55fb;
                c[15][0] <= 16'sh55fb;
                //Generated Preset 1
                //Location: 31 -31 0
                c[0][1] <= 16'sh55fb;
                c[1][1] <= 16'sh55fb;
                c[2][1] <= -16'sh55fb;
                c[3][1] <= 16'sh0;
                c[4][1] <= -16'sh3ccc;
                c[5][1] <= 16'sh0;
                c[6][1] <= 16'sh0;
                c[7][1] <= 16'sh0;
                c[8][1] <= -16'sh7999;
                c[9][1] <= 16'sh0;
                c[10][1] <= -16'sh3e70;
                c[11][1] <= 16'sh3e70;
                c[12][1] <= 16'sh0;
                c[13][1] <= 16'sh0;
                c[14][1] <= -16'sh55fb;
                c[15][1] <= -16'sh55fb;

                //Generated Preset 2
                //Location: -31 31 0
                c[0][2] <= 16'sh55fb;
                c[1][2] <= -16'sh55fb;
                c[2][2] <= 16'sh55fb;
                c[3][2] <= 16'sh0;
                c[4][2] <= -16'sh3ccc;
                c[5][2] <= 16'sh0;
                c[6][2] <= 16'sh0;
                c[7][2] <= 16'sh0;
                c[8][2] <= -16'sh7999;
                c[9][2] <= 16'sh0;
                c[10][2] <= 16'sh3e70;
                c[11][2] <= -16'sh3e70;
                c[12][2] <= 16'sh0;
                c[13][2] <= 16'sh0;
                c[14][2] <= 16'sh55fb;
                c[15][2] <= 16'sh55fb;

                //Generated Preset 3
                //Location: -31 -31 0
                c[0][3] <= 16'sh55fb;
                c[1][3] <= -16'sh55fb;
                c[2][3] <= -16'sh55fb;
                c[3][3] <= 16'sh0;
                c[4][3] <= -16'sh3ccc;
                c[5][3] <= 16'sh0;
                c[6][3] <= 16'sh0;
                c[7][3] <= 16'sh0;
                c[8][3] <= 16'sh7999;
                c[9][3] <= 16'sh0;
                c[10][3] <= 16'sh3e70;
                c[11][3] <= 16'sh3e70;
                c[12][3] <= 16'sh0;
                c[13][3] <= 16'sh0;
                c[14][3] <= 16'sh55fb;
                c[15][3] <= -16'sh55fb;


      end
   end



endmodule // coeff_static
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    23:36:31 12/08/2009
// Design Name:
// Module Name:    coeff_full
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module coeff_full
  (input [16:0] coord0, coord1, coord2, coord3,
   input clock, reset, ready,
   output reg signed [15:0] a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, aa, ab, ac, ad, ae, af,
   output reg signed [15:0] b0, b1, b2, b3, b4, b5, b6, b7, b8, b9, ba, bb, bc, bd, be, bf,
   output reg signed [15:0] c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, ca, cb, cc, cd, ce, cf,
   output reg signed [15:0] d0, d1, d2, d3, d4, d5, d6, d7, d8, d9, da, db, dc, dd, de, df,
        output reg coeffs_valid
   );

        reg [11:0] addr_in[3:0];
        reg [16:0] c_out[3:0][15:0];

        //If we use an extra bit, it can signal the "done" state.
        reg [2:0] c_counter;
        wire done = c_counter[2];

        wire [15:0] coeff0, coeff1, coeff2, coeff3, coeff4, coeff7, coeff8, coeff10, coeff11, coeff14, coeff15;
        wire [11:0] addr = addr_in[c_counter[1:0]];
        wire [1:0] offset = c_counter[1:0] - 1;

        always @(posedge clock) begin
                if (ready) begin
                        addr_in[0] <= coord0[16:5];
                        addr_in[1] <= coord1[16:5];
                        addr_in[2] <= coord2[16:5];
                        addr_in[3] <= coord3[16:5];
                end
                c_counter <= reset | ready ? 0 : done ? c_counter : c_counter + 1;
                coeffs_valid <= done;
                //fold ROM outputs
                if (~coeffs_valid) begin
                        c_out[offset][0] <= reset ? 0 : coeff0;
                        c_out[offset][1] <= reset ? 0 : coeff1;
                        c_out[offset][2] <= reset ? 0 : coeff2;
                        c_out[offset][3] <= reset ? 0 : coeff3;
                        c_out[offset][4] <= reset ? 0 : coeff4;
                        c_out[offset][7] <= reset ? 0 : coeff7;
                        c_out[offset][8] <= reset ? 0 : coeff8;
                        c_out[offset][10] <= reset ? 0 : coeff10;
                        c_out[offset][11] <= reset ? 0 : coeff11;
                        c_out[offset][14] <= reset ? 0 : coeff14;
                        c_out[offset][15] <= reset ? 0 : coeff15;
                end

                if (reset) begin
                        a5 <= 0;
                        a6 <= 0;
                ac <= 0;
                ad <= 0;
                a9 <= 0;
                b5 <= 0;
                b6 <= 0;
                bc <= 0;
                bd <= 0;
                b9 <= 0;
                c5 <= 0;
                c6 <= 0;
                cc <= 0;
                cd <= 0;
                c9 <= 0;
                d5 <= 0;
                d6 <= 0;
                dc <= 0;
                dd <= 0;
                d9 <= 0;
        end
end

//patch ROMs
ambi0 rom0(.clka(clock), .addra(addr), .douta(coeff0));
ambi1 rom1(.clka(clock), .addra(addr), .douta(coeff1));
ambi2 rom2(.clka(clock), .addra(addr), .douta(coeff2));
ambi3 rom3(.clka(clock), .addra(addr), .douta(coeff3));
ambi4 rom4(.clka(clock), .addra(addr), .douta(coeff4));
ambi7 rom7(.clka(clock), .addra(addr), .douta(coeff7));
ambi8 rom8(.clka(clock), .addra(addr), .douta(coeff8));
ambi10 rom10(.clka(clock), .addra(addr), .douta(coeff10));
ambi11 rom11(.clka(clock), .addra(addr), .douta(coeff11));
ambi14 rom14(.clka(clock), .addra(addr), .douta(coeff14));
ambi15 rom15(.clka(clock), .addra(addr), .douta(coeff15));

//module outputs from ram
//some outputs here are 0 because they are not used
//in a 2d system. this keeps the amount of ROM smaller
always @(posedge clock) begin
        //output a
        a0 = c_out[0][0];
        a1 = c_out[0][1];
        a2 = c_out[0][2];
        a3 = c_out[0][3];
        a4 = c_out[0][4];
        a7 = c_out[0][7];
        a8 = c_out[0][8];
        aa = c_out[0][10];
        ab = c_out[0][11];
        ae = c_out[0][14];
        af = c_out[0][15];
        //output b
        b0 = c_out[1][0];
        b1 = c_out[1][1];
        b2 = c_out[1][2];
        b3 = c_out[1][3];
        b4 = c_out[1][4];
        b7 = c_out[1][7];
        b8 = c_out[1][8];
        ba = c_out[1][10];
        bb = c_out[1][11];
        be = c_out[1][14];
        bf = c_out[1][15];
        //output c
        c0 = c_out[2][0];
        c1 = c_out[2][1];
        c2 = c_out[2][2];
        c3 = c_out[2][3];
        c4 = c_out[2][4];
```

```
                c7 = c_out[2][7];
                c8 = c_out[2][8];
                ca = c_out[2][10];
                cb = c_out[2][11];
                ce = c_out[2][14];
                cf = c_out[2][15];
                //output d
                d0 = c_out[3][0];
                d1 = c_out[3][1];
                d2 = c_out[3][2];
                d3 = c_out[3][3];
                d4 = c_out[3][4];
                d7 = c_out[3][7];
                d8 = c_out[3][8];
                da = c_out[3][10];
                db = c_out[3][11];
                de = c_out[3][14];
                df = c_out[3][15];
        end


endmodule
```

```
`timescale 1ns / 1ps


//3RD ORDER AMBISONIC DECODER
//Ben Bloomberg, November 2009

module decoder #(parameter X=0, parameter Y=0, parameter Z=0)
  (input signed [17:0] s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, sa, sb, sc, sd, se, sf,
   input summing_valid,
   input ready, clock, reset,
   output reg [23:0] audio_out,
        output reg decoder_valid
   );

        wire [16:0] coord = {X, Y, Z};
        wire signed [15:0] c0,c1,c2,c3,c4,c5,c6,c7,c8,c9,ca,cb,cc,cd,ce,cf;
        wire c_valid;
        coeff_static c_store(
                .coord(coord),
                .clock(clock),
                .reset(reset),
                .c0(c0),.c1(c1),.c2(c2),.c3(c3),
                .c4(c4),.c5(c5),.c6(c6),.c7(c7),
                .c8(c8),.c9(c9),.ca(ca),.cb(cb),
                .cc(cc),.cd(cd),.ce(ce),.cf(cf),
                .coeffs_valid(c_valid));

        //fold the input samples and coefficients into memories for easy access
        reg signed [15:0] s [15:0];
        reg signed [15:0] c [15:0];
        always @* begin
                //Samples
                s[0] = s0[15:0];
                s[1] = s1[15:0];
                s[2] = s2[15:0];
                s[3] = s3[15:0];
                s[4] = s4[15:0];
                s[5] = s5[15:0];
                s[6] = s6[15:0];
                s[7] = s7[15:0];
                s[8] = s8[15:0];
                s[9] = s9[15:0];
                s[10] = sa[15:0];
                s[11] = sb[15:0];
                s[12] = sc[15:0];
                s[13] = sd[15:0];
                s[14] = se[15:0];
                s[15] = sf[15:0];
                //Coeffs
                c[0] = c0;
                c[1] = c1;
                c[2] = c2;
                c[3] = c3;
                c[4] = c4;
                c[5] = c5;
                c[6] = c6;
                c[7] = c7;
                c[8] = c8;
                c[9] = c9;
                c[10] = ca;
                c[11] = cb;
                c[12] = cc;
                c[13] = cd;
                c[14] = ce;
                c[15] = cf;
        end

   // This memory holds multiplier output
```

```
    reg signed [15:0]      mo[15:0];

         wire signed [19:0] decoder_result =     {{4{mo[0][15]}}, mo[0]} + {{4{mo[8][15]}}, mo[8]} +
                                                                                                                       {{
                                                                                                                       {{
                                                                                                                       {{
                                                                                                                       {{
                                                                                                                       {{
                                                                                                                       {{
                                                                                                                       {{

         //new multiplier
         reg signed [15:0] a, b;
         wire signed [31:0] q = a*b;
    reg [3:0] m_counter;

         //detect when we should stop
         reg done;
         wire [3:0] offset = m_counter+1;

    always @(posedge clock) begin
                 done <= ready | reset ? 0 : &{m_counter} ? 1 : done;

                 //on the ready pulse, set up for multiplications
                 m_counter <= reset | ready ? 0 :
                                                  (c_valid & summing_valid & ~done ) ? m_counter + 1:
                                                  m_counter;
                 decoder_valid <= ready | reset ? 0 : done ? 1 : decoder_valid;
                 if (c_valid & summing_valid) begin
                         mo[m_counter] <= q[31:16];
                         a <= s[offset];
                         b <= c[offset];
                 end

                 audio_out <= reset ? 0 :
                                                  (summing_valid & done &{~m_counter}) ? decoder_result[15:0] :
                                                  audio_out;
    end
endmodule // decoder
```

```
`timescale 1ns / 1ps

//3RD ORDER AMBISONIC ENCODER
//Ben Bloomberg, November 2009

module enc_multiplier
  (
        input signed [15:0] c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, ca, cb, cc, cd, ce, cf, //input coefficients
   input signed [15:0] sample, //audio sample to multiply
   input clock, reset, ready, c_valid, //asserted when coefficients can be used
   output signed [15:0] e0, e1, e2, e3, e4, e5, e6, e7, e8, e9, ea, eb, ec, ed, ee, ef, //output encoded stream
   output reg encoder_valid //asserted when we're done
   );

        // This memory holds multiplier output
   reg signed [15:0]       mo[15:0];

        //fold the coefficients into memories for easy access
        reg [15:0] c [15:0];
        always @* begin
                //Coeffs
                c[0] = c0;
                c[1] = c1;
                c[2] = c2;
                c[3] = c3;
                c[4] = c4;
                c[5] = c5;
                c[6] = c6;
                c[7] = c7;
                c[8] = c8;
                c[9] = c9;
                c[10] = ca;
                c[11] = cb;
                c[12] = cc;
                c[13] = cd;
                c[14] = ce;
                c[15] = cf;
        end

        //expand the multiplier output
        assign e0 = mo[0];
        assign e1 = mo[1];
        assign e2 = mo[2];
        assign e3 = mo[3];
        assign e4 = mo[4];
        assign e5 = mo[5];
        assign e6 = mo[6];
        assign e7 = mo[7];
        assign e8 = mo[8];
        assign e9 = mo[9];
        assign ea = mo[10];
        assign eb = mo[11];
        assign ec = mo[12];
        assign ed = mo[13];
        assign ee = mo[14];
        assign ef = mo[15];

        //new multiplier
        reg signed [15:0] a, b;
        wire signed [31:0] q = a*b;
   reg [3:0] m_counter;

        //detect when we should stop
        reg done;
        wire [3:0] offset = m_counter+1;

   always @(posedge clock) begin
                done <= ready | reset ? 0 : &{m_counter} ? 1 : done;

                //on the ready pulse, set up for multiplications
                m_counter <= reset | ready ? 0 :
                                                  (c_valid & ~done ) ? m_counter + 1:
                                                  m_counter;
                encoder_valid <= ready | reset ? 0 : done ? 1 : encoder_valid;
```

```
            if (c_valid) begin
                  mo[m_counter] <= q[31:16];
                  a <= sample;
                  b <= c[offset];
            end
   end

endmodule // enc_multiplier
```

```
// a simple synchronous FIFO (first-in first-out) buffer
// Parameters:
//    LOGSIZE  (parameter) FIFO has 1<<LOGSIZE elements
//    WIDTH    (parameter) each element has WIDTH bits
// Ports:
//    clk      (input) all actions triggered on rising edge
//    reset    (input) synchronously empties fifo
//    din      (input, WIDTH bits) data to be stored
//    wr       (input) when asserted, store new data
//    full     (output) asserted when FIFO is full
//    dout     (output, WIDTH bits) data read from FIFO
//    rd       (input) when asserted, removes first element
//    empty    (output) asserted when fifo is empty
//    overflow (output) asserted when WR but no room, cleared on next RD
module fifo #(parameter  LOGSIZE = 2,   // default size is 4 elements
                         WIDTH = 4)     // default width is 4 bits
            (input clk,reset,rd,wr, input [WIDTH-1:0] din,
             output full,empty, output reg overflow, output [WIDTH-1:0] dout);
  parameter SIZE = 1 << LOGSIZE;  // compute size

  reg [WIDTH-1:0] fifo[SIZE-1:0];   // fifo data stored here
  reg [LOGSIZE-1:0] wptr,rptr;  // fifo write and read pointers

  wire [LOGSIZE-1:0] wptr_inc = wptr + 1;

  assign empty = (wptr == rptr);
  assign full = (wptr_inc == rptr);
  assign dout = fifo[rptr];

  always @(posedge clk) begin
    if (reset) begin
      wptr <= 0;
      rptr <= 0;
      overflow <= 0;
    end
    else if (wr) begin
      // store new data into the fifo
      fifo[wptr] <= din;
      wptr <= wptr_inc;
      overflow <= overflow | (wptr_inc == rptr);
    end

    // bump read pointer if we're done with current value.
    // RD also resets the overflow indicator
    if (rd && (!empty || overflow)) begin
      rptr <= rptr + 1;
      overflow <= 0;
    end
  end
endmodule
```

```verilog
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company: Bloomberg Designs
// Engineer: Ben Bloomberg
//
// Create Date:    19:15:45 11/24/2009
// Design Name:          TI PCM-1681 Driver
// Module Name:    pcm1681_driver
// Project Name:   Ambisonic Encoder
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments: This is an 8 channel audio driver for a PCM-1681. It assumes a 27Mhz clock
//                                                        hardware control of the unit. It is designed to run at 48K
//                                                                clock of 36.864Mhz.
//
////////////////////////////////////////////////////////////////////////////////
module pcm1681_driver(clock_36mhz,
                                                         audio_12,
                                                         audio_34,
                                                         audio_56,
                                                         audio_78,
                                                         audio_sclk,
                                                         audio_lrclk,
                                                         audio_bclk,
                                                         serial_data,
                                                         fmt,
                                                         demp,
                                                         mute,
                                                         reset,
                                                    ready);

         input clock_36mhz;
    input [31:0] audio_12;
    input [31:0] audio_34;
    input [31:0] audio_56;
    input [31:0] audio_78;
    output reg audio_sclk;              //Clock running at system speed (6.144Mhz)
    output reg audio_lrclk;     //Clock running at fs speed (48Khz == fs)
    output reg audio_bclk;      //Clock running at bit speed (2.304Mhz == 48fs)
    output reg [3:0] serial_data;
    output [1:0] fmt;
    output demp;
    output mute;
    input reset;
         output ready;

        // create an audio reset pulse that is held high until the next ready pulse
        // this is used to reset things in other clock domains
        reg audio_reset;

        //Right justified serial audio
        assign fmt = 2'b10;
        //No de-emphasis
        assign demp = 0;
        assign mute = 0;

        //Now we'll use a counter to count 768 samples to get lrclk
        //and 48 samples to get bclk
        reg [8:0] lr_counter;
        reg [2:0] b_counter;

        //Adding a 6.144Mhz counter (diving system by 6)
        //This was originally the 36Mhz clock, but that doesn't go
        //well over wire to the chip. We'll see if this looks better
        reg s_counter;

        //reg for creating ready pulse
        reg lrclk_sync;

        always @(posedge clock_36mhz) begin
                //counters
                lr_counter <= reset | lr_counter == 383 ? 0 : lr_counter + 1;
                s_counter <= reset ? 0 : s_counter + 1;
                b_counter <= reset ? 0 : b_counter + 1;
                //the clocks should be out of phase so that bits are latched appropriately
                audio_lrclk <= reset ? 1 : (lr_counter == 383) ? ~audio_lrclk : audio_lrclk;
                audio_bclk <= reset ? 0 : b_counter == 7 ? ~audio_bclk : audio_bclk;

                //chip sclk
                audio_sclk <= reset ? 1 : s_counter ? ~audio_sclk : audio_sclk;

                //for making ready pulse
                lrclk_sync <= audio_lrclk;

                //audio reset
                audio_reset <= reset ? 1 : ready ? 0 : audio_reset;
        end

        //Generate an AC97-like ready pulse
        assign ready = audio_lrclk & ~lrclk_sync;

        //This bit clock will keep track of which bits are being fed to
        //the DAC. We're going to use right justified serial audio, so we have time to get
        //the bits from the system.
        reg [5:0] bit_clock;
        reg [4:0] bit_num;

        //this is used to tell driver when to send bits
        wire send_bit =  (bit_clock <= 15) ||
                                            ((bit_clock >=24) && (bit_clock <= 39)) ;

        //According to the spec sheet, we should change the bit
        //on the negative edge of bclk
        always @(negedge audio_bclk) begin
                bit_clock <= audio_reset ? 0 : bit_clock == 47 ? 0 : bit_clock+1;
                bit_num <= audio_reset ? 31 : send_bit ? bit_num - 1 : bit_num;
        end

        always @* begin
                serial_data = send_bit ? {audio_12[bit_num], audio_34[bit_num], audio_56[bit_num], audio_78[bit_nu
        end

endmodule
```

```verilog
`timescale 1ns / 1ps

module summing
  (input signed [15:0] a0, b0, c0, d0,
   input signed [15:0] a1, b1, c1, d1,
   input signed [15:0] a2, b2, c2, d2,
   input signed [15:0] a3, b3, c3, d3,
   input signed [15:0] a4, b4, c4, d4,
   input signed [15:0] a5, b5, c5, d5,
   input signed [15:0] a6, b6, c6, d6,
   input signed [15:0] a7, b7, c7, d7,
   input signed [15:0] a8, b8, c8, d8,
   input signed [15:0] a9, b9, c9, d9,
   input signed [15:0] aa, ba, ca, da,
   input signed [15:0] ab, bb, cb, db,
   input signed [15:0] ac, bc, cc, dc,
   input signed [15:0] ad, bd, cd, dd,
   input signed [15:0] ae, be, ce, de,
   input signed [15:0] af, bf, cf, df,
   input clock, reset, ready,
   input a_valid, b_valid, c_valid, d_valid,
   output signed [17:0] o0, o1, o2, o3, o4, o5, o6, o7 ,o8, o9, oa, ob, oc, od, oe, of,
   output reg summing_valid
   );

        assign o0 = (a0 + b0 + c0 + d0) >>> 2;
        assign o1 = (a1 + b1 + c1 + d1) >>> 2;
        assign o2 = (a2 + b2 + c2 + d2) >>> 2;
        assign o3 = (a3 + b3 + c3 + d3) >>> 2;
        assign o4 = (a4 + b4 + c4 + d4) >>> 2;
        assign o5 = (a5 + b5 + c5 + d5) >>> 2;
        assign o6 = (a6 + b6 + c6 + d6) >>> 2;
        assign o7 = (a7 + b7 + c7 + d7) >>> 2;
        assign o8 = (a8 + b8 + c8 + d8) >>> 2;
        assign o9 = (a9 + b9 + c9 + d9) >>> 2;
        assign oa = (aa + ba + ca + da) >>> 2;
        assign ob = (ab + bb + cb + db) >>> 2;
        assign oc = (ac + bc + cc + dc) >>> 2;
        assign od = (ad + bd + cd + dd) >>> 2;
        assign oe = (ae + be + ce + de) >>> 2;
        assign of = (af + bf + cf + df) >>> 2;

   always @(posedge clock) begin
           summing_valid <= reset | ready ? 0 : &{a_valid, b_valid, c_valid, d_valid} ? 1 : summing_valid;
   end // always @ (posedge clock)

endmodule // summing
```

```
//// this is a 2 channel in, 2 channel output reciever.
//// for now the parameters don't do anything
//// this module will run at 48k, 24bits in stereo

module usb_reader #(parameter SAMPLE_BYTES=2, parameter CHANNELS=4)
        (input rxf,
              input ready,
              input clock,
              input reset,
              input [7:0] data,
                        output [7:0] led,
              output reg [(8*CHANNELS*SAMPLE_BYTES)-1:0] audio_out_data,
              output wire rd,
                        output wire fifo_full,
                        output wire sample_complete,
                        output wire [7:0] fifo_out,
                        output reg [5:0] byte_num,
                        output reg fifo_wr
              );
        // State machine and sample storage variables
   reg [7:0] audio_bits [(SAMPLE_BYTES*CHANNELS-1):0];
        reg [2:0] state;
        reg [2:0] next_state;
        reg audio_state, audio_next_state;

        // FIFO buffer!
        wire fifo_rd, fifo_overflow, fifo_empty;
        wire [7:0] fifo_in;

        assign fifo_in = data; //fifo reads directly from the usb

        fifo #( .LOGSIZE(12), .WIDTH(8)) //Should allow us to store >256 samples.
                usb_buffer( .clk(clock),
                                                .reset(reset),
                                                .rd(fifo_rd),
                                                .wr(fifo_wr),
                                                .din(data),
                                                .dout(fifo_out),
                                                .full(fifo_full),
                                                .overflow(fifo_overflow),
                                                .empty(fifo_empty));



   //These are USB states
        localparam S_IDLE                               = 3'd0;
        localparam S_DATA_AVAILABLE     = 3'd1;
        localparam S_WAIT_READ_1                = 3'd2;
        localparam S_WAIT_READ_2                = 3'd3;
        localparam S_READ_BYTE                  = 3'd4;
        localparam S_BYTE_INC                   = 3'd5;
        localparam S_DONE                                       = 3'd6;
        localparam S_WAIT                                       = 3'd7;

        //These are audio states
        localparam AS_IDLE                              = 1'b0;
        localparam AS_READ                              = 1'b1;

        reg go;

        always @(posedge clock) begin
                //take care of state
                state <= reset ? S_WAIT : next_state;
                audio_state <= reset ? AS_IDLE : audio_next_state;

                //now we have two things to accomplish:
                // 1. fill up usb fifo buffer
                // 2. deliver correct number of samples each time ready is asserted
                // Keep in mind that the state machine will halt when there is no data
                // to be retrieved off the USB bus. This means we can't base audio sample
                // retreval off the USB state machine.

                // AUDIO STUFF:
                // When the ready pulse is fired, we should retreive 6 samples from
                // fifo and put them into audio_bits

                //The go register halts initial playback until the buffer is full.
                //For a log size of 12, this incurs 7.111ms of delay on the whole system
                //This is NOT exactly what we want, but we'll play with it later
                go <= reset | fifo_empty ? 0 : fifo_full ? 1 : go;

                //keeps track of which sample is being stored
                //needs to incremet every time we read from fifo
                byte_num <= ready & sample_complete | reset             ? 0 :
                                                (byte_num >= SAMPLE_BYTES*CHANNELS) ? byte_num :
                                                (audio_state == AS_READ)                                ? byte_num
                                                byte_num;

                //advance fifo and store to audio bits if we are reading audio
                if (audio_state) audio_bits[byte_num] <= fifo_out;

                //set the audio out data when we've gotten the right number of samples
                audio_out_data <= reset ? 0 :
                                                                sample_complete & ready ? {
                                                                        audio_bits[7],
                                                                        audio_bits[6],
                                                                        audio_bits[5],
                                                                        audio_bits[4],
                                                                        audio_bits[3],
                                                                        audio_bits[2],
                                                                        audio_bits[1],
                                                                        audio_bits[0]} :
                                                                audio_out_data;

                //There should be a 1 cycle delay between when I check
                //fifo_wr and when the data is written
                fifo_wr <= (state == S_DATA_AVAILABLE);

        end

        assign led[0] = rd;
        assign led[1] = rxf;
        assign led[2] = sample_complete;
        assign led[3] = fifo_empty;
        assign led[7:4] = 4'b1111;

        //Use combinational logic to define fifo_rd so that it happens
        //right when the state changes instead of the next cycle
        assign fifo_rd = audio_state;

        //First deal with FIFO buffer
        //this enables usb_rd for four of eight states
        //it is glitch free because we are only using an inverter
        assign rd       = ~state[2];

        //Sample complete should be glitch free
        //we'll base it on the byte num for now
        //this works only because there are 8 bytes per time slice in 16-bit, 4 channel audio
        assign sample_complete = byte_num[3];

        always @* begin
                case (state)
                        S_IDLE:                                                 next_state = rxf ? state+1 : S_WAI
                        S_DATA_AVAILABLE:                               next_state = rxf ? state+1 : S_WAIT;
                        S_WAIT_READ_1:                                  next_state = rxf ? state+1 : S_WAIT;
```

```
                        S_WAIT_READ_2:                          next_state = rxf ? state+1 : S_WAIT;
                        S_READ_BYTE:                            next_state = rxf ? state+1 : S_WAIT;
                        S_BYTE_INC:                                     next_state = rxf ? state+1 : S_WAIT;
                        S_DONE:                                         next_state = rxf ? state+1 : S_WAIT;
                        S_WAIT:                                         next_state = rxf & ~fifo_full ? state+1 :
                        default:                                        next_state = S_WAIT;
                endcase
                // Audio sample state transitions
                case (audio_state)
                        AS_IDLE:                                                audio_next_state = go & ~sample_co
                        AS_READ:                                                audio_next_state = AS_IDLE;
                endcase
        end


endmodule //
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: 
// Engineer: 
// 
// Create Date:    16:58:15 12/09/2009 
// Design Name: 
// Module Name:    video 
// Project Name: 
// Target Devices: 
// Tool versions: 
// Description: 
//
// Dependencies: 
//
// Revision: 
// Revision 0.01 - File Created
// Additional Comments: 
//
//////////////////////////////////////////////////////////////////////////////////
module coord_translate(coord, x, y);
    input [16:0] coord;
    output [8:0] x;
    output [8:0] y;


         wire signed [9:0] xc,yc;
         assign xc = coord[16:11] + 6'sd32;
         assign yc = coord[10:5] + 6'sd32;

         assign x = xc << 3;
         assign y = yc << 3;

endmodule

////////////////////////////////////////////////////////////////////////////////
//
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)
//
////////////////////////////////////////////////////////////////////////////////

module xvga(input vclock,
            output reg [10:0] hcount,    // pixel number on current line
            output reg [9:0] vcount,     // line number
            output reg vsync,hsync,blank);

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

   // vertical: 806 lines total
   // display 768 lines
   wire vsyncon,vsyncoff,vreset,vblankon;
   assign vblankon = hreset & (vcount == 767);
   assign vsyncon = hreset & (vcount == 776);
   assign vsyncoff = hreset & (vcount == 782);
   assign vreset = hreset & (vcount == 805);

   // sync and blanking
   wire next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always @(posedge vclock) begin
```

```
      hcount <= hreset ? 0 : hcount + 1;
      hblank <= next_hblank;
      hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

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

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

////////////////////////////////////////////////////////////////////////////////
//
// pong_game: the game itself!
//
////////////////////////////////////////////////////////////////////////////////

module ambisonic_ui
        #(parameter XRES = 1024,      // default width: 1024 pixels
               YRES = 768)     // default height: 768 pixels

(       input vclock,   // 65MHz clock
   input reset,         // 1 to initialize module
   input up,            // 1 when paddle should move up
   input down,          // 1 when paddle should move down
   input [3:0] pspeed,  // puck speed in pixels/tick
   input [10:0] hcount, // horizontal index of current pixel (0..1023)
   input [9:0]  vcount, // vertical index of current pixel (0..767)
   input hsync,         // XVGA horizontal sync signal (active low)
   input vsync,         // XVGA vertical sync signal (active low)
   input blank,         // XVGA blanking (1 means output black pixel)
        input button0, button1, button2, button3,left, right, b_enter,
        input [7:0] switch_clean,

   output phsync,       // pong game's horizontal sync
   output pvsync,       // pong game's vertical sync
   output pblank,       // pong game's blanking
   output [2:0] pixel,  // pong game's pixel
        output [1:0] active_preset,
        output [8:0] x0,y0,x1,y1,x2,y2,x3,y3,
        output [16:0] coord0, coord1, coord2, coord3

   );

        reg [9:0] x [3:0];
        reg [9:0] y [3:0];

        wire signed [9:0] xs [3:0];
        wire signed [9:0] ys [3:0];

        wire [2:0] p0, p1, p2, p3, p4;

        //locations
        blob #(.WIDTH(16),.HEIGHT(16),.COLOR(3'b110))   // yellow!
                  l1(.x(x[0]),.y(y[0]),.hcount(hcount),.vcount(vcount),
                                       .pixel(p0));

        blob #(.WIDTH(16),.HEIGHT(16),.COLOR(3'b011))   // cyan!
                  l2(.x(x[1]),.y(y[1]),.hcount(hcount),.vcount(vcount),
                                       .pixel(p1));

        blob #(.WIDTH(16),.HEIGHT(16),.COLOR(3'b101))   // magenta!
                 l3(.x(x[2]),.y(y[2]),.hcount(hcount),.vcount(vcount),
                                       .pixel(p2));

        blob #(.WIDTH(16),.HEIGHT(16),.COLOR(3'b111))   // white!
                  l4(.x(x[3]),.y(y[3]),.hcount(hcount),.vcount(vcount),
                                       .pixel(p3));

        blob_circle #(.RADIUS(160),.THICKNESS(800),.COLOR(3'b001))   // white!
                  l5(.x(512),.y(384),.hcount(hcount),.vcount(vcount),
                                       .pixel(p4), .vclock(vclock));

        // x [16:11]
        // y [10:5]
        // z [4:0]

   //generate vsync for drawing
        reg vsync_delay;
        wire draw_screen;
        always @(posedge vclock) vsync_delay <= vsync;
        assign draw_screen = vsync_delay & ~vsync;


        //add preset manager here

        preset_manager p(.clock(vclock),
                                            .reset(reset),
                                            .update(draw_screen),
                                            .vs_coord_0(coord0),
                                            .vs_coord_1(coord1),
                                            .vs_coord_2(coord2),
                                            .vs_coord_3(coord3),
                                            .active_preset(active_preset),
                                            .button0(button0),
                                            .button1(button1),
                                            .button2(button2),
                                            .button3(button3),
                                            .up(up),
                                            .down(down),
                                            .left(left),
                                            .right(right),
                                            .b_enter(b_enter),
                                            .switch(switch_clean));

        wire hlines = (((hcount == 256)||(hcount == 768)) & (vcount >= 128) & (vcount <= 640));
        wire vlines = (((vcount == 128)||(vcount == 640)) & (hcount >= 256) & (hcount <= 768));

        assign phsync = hsync;
   assign pvsync = vsync;
   assign pblank = blank;
   assign pixel = hlines | vlines ? 3'b111 // paint borders
                                         : |p0? p0 : |p1 ? p1 : |p2 ? p2 : |p3 ? p3 :  p4;

        coord_translate ct0 (.coord(coord0), .x(x0), .y(y0) );
        coord_translate ct1 (.coord(coord1), .x(x1), .y(y1) );
        coord_translate ct2 (.coord(coord2), .x(x2), .y(y2) );
        coord_translate ct3 (.coord(coord3), .x(x3), .y(y3) );


        always @(posedge draw_screen) begin
                //set each coordinate
                x[0] <= x0 + 256;
                y[0] <= y0 + 128;
                x[1] <= x1+ 256;
                y[1] <= y1+ 128;
                x[2] <= x2+ 256;
                y[2] <= y2+ 128;
                x[3] <= x3+ 256;
                y[3] <= y3+ 128;
        end


endmodule
```

```
////////////////////////////////////////////////////////////////////
//
// blob: generate rectangle on screen
//
////////////////////////////////////////////////////////////////////
module blob
   #(parameter WIDTH = 64,      // default width: 64 pixels
               HEIGHT = 64,     // default height: 64 pixels
               COLOR = 3'b111)  // default color: white
   (input [10:0] x,hcount,
    input [9:0] y,vcount,
    output reg [2:0] pixel);

   always @ (x or y or hcount or vcount) begin
      if ((hcount >= x && hcount < (x+WIDTH)) &&
         (vcount >= y && vcount < (y+HEIGHT)))
        pixel = COLOR;
      else pixel = 0;
   end
endmodule

module blob_circle
   #(parameter RADIUS = 32, THICKNESS=4,      // default radius: 64 pixels
               COLOR = 3'b111)  // default color: white
   (input [10:0] x,hcount,
    input [9:0] y,vcount,
         input vclock,
    output reg [2:0] pixel);

         reg [10:0] x_offset;
         reg [9:0] y_offset;

         reg [15:0] xsq, ysq;

         always @(posedge vclock) begin
                if (x < hcount) x_offset <= hcount - x;
                else x_offset <= x - hcount;
                if (y < vcount) y_offset <= vcount - y;
                else y_offset <= y - vcount;

                xsq <= x_offset * x_offset;
                ysq <= y_offset * y_offset;
         end

         wire [16:0] current_radius_sq = xsq+ysq;

         always @ (x or y or hcount or vcount) begin
                 if ((current_radius_sq < (RADIUS*RADIUS)) &&
                    (hcount>=(x-RADIUS) && hcount<=(x+RADIUS)) &&
                         (vcount>=(y-RADIUS) && vcount<=(y+RADIUS)) &&
                         (current_radius_sq > ((RADIUS*RADIUS)-THICKNESS))) pixel=COLOR;
                 else pixel = 0;
         end


endmodule

////////////////////////////////////////////////////////////////////
//
// preset_manager: handles button input and generates
// coordinates for each preset
//
////////////////////////////////////////////////////////////////////
```

```
module preset_manager
        ( input clock, reset,
          input  button0, button1, button2, button3,left,right,up,down, b_enter, update,
          input [7:0] switch,
          output reg [16:0] vs_coord_0, vs_coord_1, vs_coord_2, vs_coord_3,
                // keep track of active preset
          output reg [1:0] active_preset          );


         // coordinate memory
         reg signed [5:0] x [3:0][3:0];
         reg signed [5:0] y [3:0][3:0];

         reg signed [5:0] current_x[3:0];
         reg signed [5:0] current_y[3:0];

         always @(posedge clock) begin
                vs_coord_0 <= {current_x[0], current_y[0], 6'sd0};
                vs_coord_1 <= {current_x[1], current_y[1], 6'sd0};
                vs_coord_2 <= {current_x[2], current_y[2], 6'sd0};
                vs_coord_3 <= {current_x[3], current_y[3], 6'sd0};
                if (reset) active_preset <= 0;
                else if (button0) active_preset <= 0;
                else if (button1) active_preset <= 1;
                else if (button2) active_preset <= 2;
                else if (button3) active_preset <= 3;
         end

//              vs_coord_0 <= {x[0][active_preset], y[0][active_preset], 6'sd0};
//              vs_coord_1 <= {x[1][active_preset], y[1][active_preset], 6'sd0};
//              vs_coord_2 <= {x[2][active_preset], y[2][active_preset], 6'sd0};
//              vs_coord_3 <= {x[3][active_preset], y[3][active_preset], 6'sd0};

         always @(posedge update) begin


                        if (switch[2]) begin
                                if (up) y[0][active_preset] <= (y[0][active_preset]<=-16) ? y[0][active_preset] :
                                if (down) y[0][active_preset] <= (y[0][active_preset]>=15) ? y[0][active_preset] :
                                if (right) x[0][active_preset] <= (x[0][active_preset]>=15) ? x[0][active_preset]
                                if (left) x[0][active_preset] <= (x[0][active_preset]<=-16) ? x[0][active_preset]
                        end
                        if (switch[3]) begin
                                if (up) y[1][active_preset] <= (y[1][active_preset] <=-16) ? y[1][active_preset] :
                                if (down) y[1][active_preset] <= (y[1][active_preset]>=15) ? y[1][active_preset] :
                                if (right) x[1][active_preset] <= (x[1][active_preset]>=15) ? x[1][active_preset]
                                if (left) x[1][active_preset] <= (x[1][active_preset]<=-16) ? x[1][active_preset]
                        end
                        if (switch[4]) begin
                                if (up) y[2][active_preset] <= (y[2][active_preset]<=-16) ? y[2][active_preset] :
                                if (down) y[2][active_preset] <= (y[2][active_preset]>=15) ? y[2][active_preset] :
                                if (right) x[2][active_preset] <= (x[2][active_preset]>=15) ? x[2][active_preset]
                                if (left) x[2][active_preset] <= (x[2][active_preset]<=-16) ? x[2][active_preset]
                        end
                        if (switch[5]) begin
                                if (up) y[3][active_preset] <= (y[3][active_preset]<=-16) ? y[3][active_preset] :
                                if (down) y[3][active_preset] <= (y[3][active_preset]>=15) ? y[3][active_preset] :
                                if (right) x[3][active_preset] <= (x[3][active_preset]>=15) ? x[3][active_preset]
                                if (left) x[3][active_preset] <= (x[3][active_preset]<=-16) ? x[3][active_preset]
                        end

                        ///X Variables
                        current_x[0] <= ((current_x[0]-1) > x[0][active_preset]) ? current_x[0]-1 :
                                                                 ((current_x[0]+1) < x[0][active_preset]) ? curren
                                                                 current_x[0];
                        current_x[1] <= ((current_x[1]-1) > x[1][active_preset]) ? current_x[1]-1 :
                                                                 ((current_x[1]+1) < x[1][active_preset]) ? curren
```

```
                                                                                current_x[1];
                                current_x[2] <= ((current_x[2]-1) > x[2][active_preset]) ? current_x[2]-1 :
                                                                                ((current_x[2]+1) < x[2][active_preset]) ? curren
                                                                                current_x[2];
                                current_x[3] <= ((current_x[3]-1) > x[3][active_preset]) ? current_x[3]-1 :
                                                                                ((current_x[3]+1) < x[3][active_preset]) ? curren
                                                                                current_x[3];

                                ///Y variables
                                current_y[0] <= ((current_y[0]-1) > y[0][active_preset]) ? current_y[0]-1 :
                                                                                ((current_y[0]+1) < y[0][active_preset]) ? curren
                                                                                current_y[0];
                                current_y[1] <= ((current_y[1]-1) > y[1][active_preset]) ? current_y[1]-1 :
                                                                                ((current_y[1]+1) < y[1][active_preset]) ? curren
                                                                                current_y[1];
                                current_y[2] <= ((current_y[2]-1) > y[2][active_preset]) ? current_y[2]-1 :
                                                                                ((current_y[2]+1) < y[2][active_preset]) ? curren
                                                                                current_y[2];
                                current_y[3] <= ((current_y[3]-1) > y[3][active_preset]) ? current_y[3]-1 :
                                                                                ((current_y[3]+1) < y[3][active_preset]) ? curren
                                                                                current_y[3];
          end

endmodule //preset_manager

module sync_coord
        (input clock,
         input [16:0] vcoord,
         output reg [16:0] acoord);

         reg [16:0] a,b,c;
        //this module syncs the coordinates coming from the video system to the
        //audio system

        always @(posedge clock) begin
                a <= vcoord;
                b <= a;
                c <= b;
                acoord <= c;
        end
endmodule
```

Appendix B: Verilog Test Fixtures

```verilog
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:   17:32:56 12/05/2009
// Design Name:   summing
// Module Name:   C:/bb/pcm1681_ambisonic/codec_tester.v
// Project Name:  pcm1681_ambisonic
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: summing
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module codec_tester_v;
        reg reset;
        reg clock_36mhz;

        //audio pulse every 48Khz
        reg ready;

        reg signed [15:0] pcm [3:0];

        reg [15:0] sample;

        //virtual source coordinates
        reg [16:0] coord0, coord1, coord2, coord3;

        initial begin
                coord0 = {6'sd4, 6'sd4, 5'sd0};
                coord1 = {6'sd4, 6'sd4, 5'sd0};
                coord2 = {6'sd4, 6'sd4, 5'sd0};
                coord3 = {6'sd4, 6'sd4, 5'sd0};
                clock_36mhz = 0;
                reset = 1;
                // Wait 100 ns for global reset to finish
                #100;
                reset = 0;
                sample = 16'sd10;
                // Add stimulus here
        end

        always #1 clock_36mhz = ~clock_36mhz;
        always #768 begin
                ready = 1;
                #2 ready =0;
        end

        //always @(posedge ready) sample = sample + 10;
        always begin
                #768 sample = 16'sd32767;
                #768 sample = -16'sd32768;
                #768 sample = 0;
                #768 sample = 5000;
                #768 sample = 16'sd32767;
                #768 sample = 16'sd32467;
                #768 sample = 16'sd32367;
                #768 sample = 16'sd32267;
```

```
		#768 sample = 16'sd32167;
		#768 sample = 16'sd32067;
		#768 sample = 16'sd25067;
		#768 sample = -16'sd32767;
		#768 sample = -16'sd32467;
		#768 sample = -16'sd32367;
		#768 sample = -16'sd32267;
		#768 sample = -16'sd32167;
		#768 sample = -16'sd32067;
		#768 sample = -16'sd25067;

	end


/*
 ***********************************************
 * COEFFICIENT ROM: Ambisonic 3rd order decoders
 ***********************************************
 */
 wire [15:0] ca0, ca1, ca2, ca3, ca4, ca5, ca6, ca7, ca8, ca9, caa, cab, cac, cad, cae, caf;
 wire [15:0] cb0, cb1, cb2, cb3, cb4, cb5, cb6, cb7, cb8, cb9, cba, cbb, cbc, cbd, cbe, cbf;
 wire [15:0] cc0, cc1, cc2, cc3, cc4, cc5, cc6, cc7, cc8, cc9, cca, ccb, ccc, ccd, cce, ccf;
 wire [15:0] cd0, cd1, cd2, cd3, cd4, cd5, cd6, cd7, cd8, cd9, cda, cdb, cdc, cdd, cde, cdf;
 wire coeffs_valid;
	coeff_full coeff_rom (
   .coord0(coord0), .coord1(coord1), .coord2(coord2), .coord3(coord3),
   .clock(clock_36mhz), .reset(reset), .ready(ready),
	.a0(ca0), .a1(ca1), .a2(ca2), .a3(ca3), .a4(ca4), .a5(ca5), .a6(ca6), .a7(ca7),
	.a8(ca8), .a9(ca9), .aa(caa), .ab(cab), .ac(cac), .ad(cad), .ae(cae), .af(caf),
   .b0(cb0), .b1(cb1), .b2(cb2), .b3(cb3), .b4(cb4), .b5(cb5), .b6(cb6), .b7(cb7),
   .b8(cb8), .b9(cb9), .ba(cba), .bb(cbb), .bc(cbc), .bd(cbd), .be(cbe), .bf(cbf),
   .c0(cc0), .c1(cc1), .c2(cc2), .c3(cc3), .c4(cc4), .c5(cc5), .c6(cc6), .c7(cc7),
   .c8(cc8), .c9(cc9), .ca(cca), .cb(ccb), .cc(ccc), .cd(ccd), .ce(cce), .cf(ccf),
   .d0(cd0), .d1(cd1), .d2(cd2), .d3(cd3), .d4(cd4), .d5(cd5), .d6(cd6), .d7(cd7),
   .d8(cd8), .d9(cd9), .da(cda), .db(cdb), .dc(cdc), .dd(cdd), .de(cde), .df(cdf),
   .coeffs_valid(coeffs_valid)
   );

/*
 ***********************************************
 * ENCODER: Ambisonic 3rd order decoders
 ***********************************************
 */

 wire [15:0] a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, aa, ab, ac, ad, ae, af;
 wire [15:0] b0, b1, b2, b3, b4, b5, b6, b7, b8, b9, ba, bb, bc, bd, be, bf;
 wire [15:0] c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, ca, cb, cc, cd, ce, cf;
 wire [15:0] d0, d1, d2, d3, d4, d5, d6, d7, d8, d9, da, db, dc, dd, de, df;
 wire enc0r, enc1r, enc2r, enc3r;
 enc_multiplier enc0 (
   .sample(sample),
   .clock(clock_36mhz),
   .reset(reset),
   .ready(ready),
   .e0(a0), .e1(a1), .e2(a2), .e3(a3), .e4(a4), .e5(a5), .e6(a6), .e7(a7),
   .e8(a8), .e9(a9), .ea(aa), .eb(ab), .ec(ac), .ed(ad), .ee(ae), .ef(af),
	.c0(ca0), .c1(ca1), .c2(ca2), .c3(ca3), .c4(ca4), .c5(ca5), .c6(ca6), .c7(ca7),
   .c8(ca8), .c9(ca9), .ca(caa), .cb(cab), .cc(cac), .cd(cad), .ce(cae), .cf(caf),
   .encoder_valid(enc0r), .c_valid(coeffs_valid)
   );

 enc_multiplier enc1 (
   .sample(sample),
   .clock(clock_36mhz),
   .reset(reset),
   .ready(ready),
   .e0(b0), .e1(b1), .e2(b2), .e3(b3), .e4(b4), .e5(b5), .e6(b6), .e7(b7),
   .e8(b8), .e9(b9), .ea(ba), .eb(bb), .ec(bc), .ed(bd), .ee(be), .ef(bf),
          .c0(cb0), .c1(cb1), .c2(cb2), .c3(cb3), .c4(cb4), .c5(cb5), .c6(cb6), .c7(cb7),
     .c8(cb8), .c9(cb9), .ca(cba), .cb(cbb), .cc(cbc), .cd(cbd), .ce(cbe), .cf(cbf),
     .encoder_valid(enc1r), .c_valid(coeffs_valid)
     );

   enc_multiplier enc2 (
     .sample(sample),
     .clock(clock_36mhz),
     .reset(reset),
     .ready(ready),
     .e0(c0), .e1(c1), .e2(c2), .e3(c3), .e4(c4), .e5(c5), .e6(c6), .e7(c7),
     .e8(c8), .e9(c9), .ea(ca), .eb(cb), .ec(cc), .ed(cd), .ee(ce), .ef(cf),
          .c0(cc0), .c1(cc1), .c2(cc2), .c3(cc3), .c4(cc4), .c5(cc5), .c6(cc6), .c7(cc7),
     .c8(cc8), .c9(cc9), .ca(cca), .cb(ccb), .cc(ccc), .cd(ccd), .ce(cce), .cf(ccf),
     .encoder_valid(enc2r), .c_valid(coeffs_valid)
     );

   enc_multiplier enc3 (
     .sample(sample),
     .clock(clock_36mhz),
     .reset(reset),
     .ready(ready),
     .e0(d0), .e1(d1), .e2(d2), .e3(d3), .e4(d4), .e5(d5), .e6(d6), .e7(d7),
     .e8(d8), .e9(d9), .ea(da), .eb(db), .ec(dc), .ed(dd), .ee(de), .ef(df),
          .c0(cd0), .c1(cd1), .c2(cd2), .c3(cd3), .c4(cd4), .c5(cd5), .c6(cd6), .c7(cd7),
     .c8(cd8), .c9(cd9), .ca(cda), .cb(cdb), .cc(cdc), .cd(cdd), .ce(cde), .cf(cdf),
     .encoder_valid(enc3r), .c_valid(coeffs_valid)
     );


  /*
   ***********************************************
   * SUMMING: Ambisonic 3rd order decoders
   ***********************************************
   */

   wire [17:0] s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, sa, sb, sc, sd, se, sf;
   wire summing_valid;

 summing sum (
     .a0(a0), .b0(b0), .c0(c0), .d0(d0),
          .a1(a1), .b1(b1), .c1(c1), .d1(d1),
     .a2(a2), .b2(b2), .c2(c2), .d2(d2),
     .a3(a3), .b3(b3), .c3(c3), .d3(d3),
     .a4(a4), .b4(b4), .c4(c4), .d4(d4),
     .a5(a5), .b5(b5), .c5(c5), .d5(d5),
     .a6(a6), .b6(b6), .c6(c6), .d6(d6),
     .a7(a7), .b7(b7), .c7(c7), .d7(d7),
     .a8(a8), .b8(b8), .c8(c8), .d8(d8),
     .a9(a9), .b9(b9), .c9(c9), .d9(d9),
     .aa(aa), .ba(ba), .ca(ca), .da(da),
     .ab(ab), .bb(bb), .cb(cb), .db(db),
     .ac(ac), .bc(bc), .cc(cc), .dc(dc),
     .ad(ad), .bd(bd), .cd(cd), .dd(dd),
     .ae(ae), .be(be), .ce(ce), .de(de),
     .af(af), .bf(bf), .cf(cf), .df(df),
     .clock(clock_36mhz),
     .reset(reset),
     .ready(ready),
     .a_valid(enc0r),
     .b_valid(enc1r),
     .c_valid(enc2r),
     .d_valid(enc3r),
     .o0(s0), .o1(s1), .o2(s2), .o3(s3), .o4(s4), .o5(s5), .o6(s6), .o7(s7),
          .o8(s8), .o9(s9), .oa(sa), .ob(sb), .oc(sc), .od(sd), .oe(se), .of(sf),
     .summing_valid(summing_valid)
     );
```

```
/*
 ***********************************************
 * DECODER: Ambisonic 3rd order decoders
 ***********************************************
 */

 //decoder ready signals
 wire dec0r, dec1r, dec2r, dec3r;
 wire signed [23:0] chan1, chan2, chan3, chan4;

       decoder dec0 (
   .s0(s0), .s1(s1), .s2(s2), .s3(s3), .s4(s4), .s5(s5), .s6(s6), .s7(s7),
   .s8(s8), .s9(s9), .sa(sa), .sb(sb), .sc(sc), .sd(sd), .se(se), .sf(sf),
   .summing_valid(summing_valid),
   .ready(ready),
   .clock(clock_36mhz),
   .reset(reset),
   .audio_out(chan1),
   .decoder_valid(dec0r)
   );

       decoder dec1 (
   .s0(s0), .s1(s1), .s2(s2), .s3(s3), .s4(s4), .s5(s5), .s6(s6), .s7(s7),
   .s8(s8), .s9(s9), .sa(sa), .sb(sb), .sc(sc), .sd(sd), .se(se), .sf(sf),
   .summing_valid(summing_valid),
   .ready(ready),
   .clock(clock_36mhz),
   .reset(reset),
   .audio_out(chan2),
   .decoder_valid(dec1r)
   );

       decoder dec2 (
   .s0(s0), .s1(s1), .s2(s2), .s3(s3), .s4(s4), .s5(s5), .s6(s6), .s7(s7),
   .s8(s8), .s9(s9), .sa(sa), .sb(sb), .sc(sc), .sd(sd), .se(se), .sf(sf),
   .summing_valid(summing_valid),
   .ready(ready),
   .clock(clock_36mhz),
   .reset(reset),
   .audio_out(chan3),
   .decoder_valid(dec2r)
   );

       decoder dec3 (
   .s0(s0), .s1(s1), .s2(s2), .s3(s3), .s4(s4), .s5(s5), .s6(s6), .s7(s7),
   .s8(s8), .s9(s9), .sa(sa), .sb(sb), .sc(sc), .sd(sd), .se(se), .sf(sf),
   .summing_valid(summing_valid),
   .ready(ready),
   .clock(clock_36mhz),
   .reset(reset),
   .audio_out(chan4),
   .decoder_valid(dec3r)
   );

       //On the ready pulse, present audio to
       //the pcm1681 driver
       always @(posedge clock_36mhz) begin
               if (reset) begin
                               pcm[0] <= 0;
                               pcm[1] <= 0;
                               pcm[2] <= 0;
                               pcm[3] <= 0;
               end
               else if (&{ready,dec0r,dec1r,dec2r,dec3r}) begin
                               pcm[0] <= chan1[15:0];
                               pcm[1] <= chan2[15:0];
                               pcm[2] <= chan3[15:0];
                               pcm[3] <= chan4[15:0];
```

```
                end
        end

endmodule
```

```
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:   12:41:14 12/09/2009
// Design Name:   coeff_full
// Module Name:   C:/bb/pcm1681_ambisonic/coeff_rom_tester.v
// Project Name:  pcm1681_ambisonic
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: coeff_full
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module coeff_rom_tester_v;

        // Inputs
        reg [16:0] coord0;
        reg [16:0] coord1;
        reg [16:0] coord2;
        reg [16:0] coord3;
        reg clock;
        reg reset;
        reg ready;

        // Outputs
        wire [15:0] a0;
        wire [15:0] a1;
        wire [15:0] a2;
        wire [15:0] a3;
        wire [15:0] a4;
        wire [15:0] a5;
        wire [15:0] a6;
        wire [15:0] a7;
        wire [15:0] a8;
        wire [15:0] a9;
        wire [15:0] aa;
        wire [15:0] ab;
        wire [15:0] ac;
        wire [15:0] ad;
        wire [15:0] ae;
        wire [15:0] af;
        wire [15:0] b0;
        wire [15:0] b1;
        wire [15:0] b2;
        wire [15:0] b3;
        wire [15:0] b4;
        wire [15:0] b5;
        wire [15:0] b6;
        wire [15:0] b7;
        wire [15:0] b8;
        wire [15:0] b9;
        wire [15:0] ba;
        wire [15:0] bb;
        wire [15:0] bc;
        wire [15:0] bd;
        wire [15:0] be;
        wire [15:0] bf;
```

```
	wire [15:0] c0;
	wire [15:0] c1;
	wire [15:0] c2;
	wire [15:0] c3;
	wire [15:0] c4;
	wire [15:0] c5;
	wire [15:0] c6;
	wire [15:0] c7;
	wire [15:0] c8;
	wire [15:0] c9;
	wire [15:0] ca;
	wire [15:0] cb;
	wire [15:0] cc;
	wire [15:0] cd;
	wire [15:0] ce;
	wire [15:0] cf;
	wire [15:0] d0;
	wire [15:0] d1;
	wire [15:0] d2;
	wire [15:0] d3;
	wire [15:0] d4;
	wire [15:0] d5;
	wire [15:0] d6;
	wire [15:0] d7;
	wire [15:0] d8;
	wire [15:0] d9;
	wire [15:0] da;
	wire [15:0] db;
	wire [15:0] dc;
	wire [15:0] dd;
	wire [15:0] de;
	wire [15:0] df;
	wire coeffs_valid;

	// Instantiate the Unit Under Test (UUT)
	coeff_full uut (
		.coord0(coord0),
		.coord1(coord1),
		.coord2(coord2),
		.coord3(coord3),
		.clock(clock),
		.reset(reset),
		.ready(ready),
		.a0(a0),
		.a1(a1),
		.a2(a2),
		.a3(a3),
		.a4(a4),
		.a5(a5),
		.a6(a6),
		.a7(a7),
		.a8(a8),
		.a9(a9),
		.aa(aa),
		.ab(ab),
		.ac(ac),
		.ad(ad),
		.ae(ae),
		.af(af),
		.b0(b0),
		.b1(b1),
		.b2(b2),
		.b3(b3),
		.b4(b4),
		.b5(b5),
		.b6(b6),
		.b7(b7),
		.b8(b8),
		.b9(b9),
		.ba(ba),
		.bb(bb),
		.bc(bc),
		.bd(bd),
		.be(be),
		.bf(bf),
		.c0(c0),
		.c1(c1),
		.c2(c2),
		.c3(c3),
		.c4(c4),
		.c5(c5),
		.c6(c6),
		.c7(c7),
		.c8(c8),
		.c9(c9),
		.ca(ca),
		.cb(cb),
		.cc(cc),
		.cd(cd),
		.ce(ce),
		.cf(cf),
		.d0(d0),
		.d1(d1),
		.d2(d2),
		.d3(d3),
		.d4(d4),
		.d5(d5),
		.d6(d6),
		.d7(d7),
		.d8(d8),
		.d9(d9),
		.da(da),
		.db(db),
		.dc(dc),
		.dd(dd),
		.de(de),
		.df(df),
		.coeffs_valid(coeffs_valid)
	);

	reg signed [5:0] num;
	initial begin
		// Initialize Inputs
		num = 0;
		coord0 ={-num, 6'sd1, 5'sd0};
		coord1 ={6'sd22, num, 5'sd0};
		coord2 ={num, num, 5'sd0};
		coord3 ={num, 6'sd3, 5'sd0};
		clock = 0;
		reset = 1;
		ready = 0;

		// Wait 100 ns for global reset to finish
		#100;
		reset = 0;

		// Add stimulus here

	end
    always #768 begin
		ready = 1;
		#2 ready = 0;
	end
	always #1 clock = ~clock;
	always @(posedge ready) begin
		num = num + 1;
```

```
                coord0 ={-num, 6'sd1, 5'sd0};
                coord1 ={6'sd22, num, 5'sd0};
                coord2 ={num, num, 5'sd0};
                coord3 ={num, 6'sd3, 5'sd0};
        end
endmodule
```

```verilog
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:   12:49:18 12/04/2009
// Design Name:   decoder
// Module Name:   C:/bb/pcm1681_ambisonic/decoder_tester.v
// Project Name:  pcm1681_ambisonic
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: decoder
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module decoder_tester_v;

	// Inputs
	reg [31:0] s0;
	reg [31:0] s1;
	reg [31:0] s2;
	reg [31:0] s3;
	reg [31:0] s4;
	reg [31:0] s5;
	reg [31:0] s6;
	reg [31:0] s7;
	reg [31:0] s8;
	reg [31:0] s9;
	reg [31:0] sa;
	reg [31:0] sb;
	reg [31:0] sc;
	reg [31:0] sd;
	reg [31:0] se;
	reg [31:0] sf;
	reg summing_valid;
	reg ready;
	reg clock;
	reg reset;

	// Outputs
	wire [15:0] audio_out;
	wire decoder_ready;

	// Instantiate the Unit Under Test (UUT)
	decoder uut (
		.s0(s0),
		.s1(s1),
		.s2(s2),
		.s3(s3),
		.s4(s4),
		.s5(s5),
		.s6(s6),
		.s7(s7),
		.s8(s8),
		.s9(s9),
		.sa(sa),
		.sb(sb),
		.sc(sc),
		.sd(sd),
		.se(se),
		.sf(sf),
		.summing_valid(summing_valid),
		.ready(ready),
		.clock(clock),
		.reset(reset),
		.audio_out(audio_out),
		.decoder_valid(decoder_ready)
	);

	initial begin
		// Initialize Inputs
		s0 = 32'sh1;
		s1 = 32'sh1;
		s2 = 32'sh1;
		s3 = 32'sh1;
		s4 = 32'sh1;
		s5 = 32'sh1;
		s6 = 32'sh1;
		s7 = 32'sh1;
		s8 = 32'sh1;
		s9 = 32'sh1;
		sa = 32'sh1;
		sb = 32'sh1;
		sc = 32'sh1;
		sd = 32'sh1;
		se = 32'sh1;
		sf = 32'sh1;
		summing_valid = 0;
		ready = 0;
		clock = 0;
		reset = 1;

		// Wait 100 ns for global reset to finish
		#100;
		reset = 0;
	summing_valid = 1;
		// Add stimulus here

	end

	always #1 clock = ~clock;

	always #501 begin
		ready = 1;
		#2 ready = 0;
	end

	always @(posedge ready) begin
		s0 = s0*10;
		s1 = s1*10;
		s2 = s2*10;
		s3 = s3*10;
		s4 = s4*10;
		s5 = s5*10;
		s6 = s6*10;
		s7 = s7*10;
		s8 = s8*10;
		s9 = s9*10;
		sa = sa*10;
		sb = sb*10;
		sc = sc*10;
		sd = sd*10;
		se = se*10;
		sf = sf*10;
	end

endmodule
```

```verilog
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:   23:58:49 12/07/2009
// Design Name:   enc_multiplier
// Module Name:   C:/bb/pcm1681_ambisonic/encoder_tester.v
// Project Name:  pcm1681_ambisonic
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: enc_multiplier
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module encoder_tester_v;

        // Inputs
        reg [16:0] coord;
        reg [15:0] sample;
        reg clock;
        reg reset;
        reg ready;

        // Outputs
        wire [31:0] e0;
        wire [31:0] e1;
        wire [31:0] e2;
        wire [31:0] e3;
        wire [31:0] e4;
        wire [31:0] e5;
        wire [31:0] e6;
        wire [31:0] e7;
        wire [31:0] e8;
        wire [31:0] e9;
        wire [31:0] ea;
        wire [31:0] eb;
        wire [31:0] ec;
        wire [31:0] ed;
        wire [31:0] ee;
        wire [31:0] ef;
        wire encoder_valid;

        // Instantiate the Unit Under Test (UUT)
        enc_multiplier uut (
                .coord(coord),
                .sample(sample),
                .clock(clock),
                .reset(reset),
                .ready(ready),
                .e0(e0),
                .e1(e1),
                .e2(e2),
                .e3(e3),
                .e4(e4),
                .e5(e5),
                .e6(e6),
                .e7(e7),
                .e8(e8),
                .e9(e9),
                .ea(ea),
                .eb(eb),
                .ec(ec),
                .ed(ed),
                .ee(ee),
                .ef(ef),
                .encoder_valid(encoder_valid)
        );

        initial begin
                // Initialize Inputs
                coord = 0;
                sample = 1;
                clock = 0;
                reset = 1;
                ready = 0;

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

                // Add stimulus here
                reset =0;

        end
   always #1 clock = !clock;

        always #768 begin
                ready = 1;
                #2 ready = 0;
        end

        always @(posedge ready) sample = 1 + sample;


endmodule
```

```verilog
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:   19:21:06 11/30/2009
// Design Name:   pcm1681_driver
// Module Name:   C:/bb/pcm1681/pcm1681_tester.v
// Project Name:  pcm1681
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: pcm1681_driver
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module pcm1681_tester_v;

	// Inputs
	reg clock_36mhz;
	reg [31:0] audio_12;
	reg [31:0] audio_34;
	reg [31:0] audio_56;
	reg [31:0] audio_78;
	reg reset;

	// Outputs
	wire audio_sclk;
	wire audio_lrclk;
	wire audio_bclk;
	wire [3:0] serial_data;
	wire [2:0] fmt;
	wire demp;
	wire mute;
	wire ready;

	// Instantiate the Unit Under Test (UUT)
	pcm1681_driver uut (
		.clock_36mhz(clock_36mhz),
		.audio_12(audio_12),
		.audio_34(audio_34),
		.audio_56(audio_56),
		.audio_78(audio_78),
		.audio_sclk(audio_sclk),
		.audio_lrclk(audio_lrclk),
		.audio_bclk(audio_bclk),
		.serial_data(serial_data),
		.fmt(fmt),
		.demp(demp),
		.mute(mute),
		.reset(reset),
		.ready(ready)
	);

	initial begin
		// Initialize Inputs
		clock_36mhz = 0;
		audio_12 = 32'b10101010101010101010101010101010;
		audio_34 = 0;
		audio_56 = 0;
```

```
                audio_78 = 0;
                reset = 1;

                // Wait 100 ns for global reset to finish
                #100;
                reset = 0;

                // Add stimulus here

        end

        always #1 clock_36mhz = ~clock_36mhz;

endmodule
```

```verilog
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:   16:15:32 12/07/2009
// Design Name:   enc_multiplier
// Module Name:   C:/bb/pcm1681_ambisonic/simple_codec_tester.v
// Project Name:  pcm1681_ambisonic
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: enc_multiplier
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module simple_codec_tester_v;

        // Inputs
        reg [16:0] coord;
        reg [15:0] sample;
        reg clock_36mhz;
        reg reset;
        reg ready;

  wire coeffs_valid;
  wire [15:0] ca0, ca1, ca2, ca3, ca4, ca5, ca6, ca7, ca8, ca9, caa, cab, cac, cad, cae, caf;
  wire [15:0] cb0, cb1, cb2, cb3, cb4, cb5, cb6, cb7, cb8, cb9, cba, cbb, cbc, cbd, cbe, cbf;
  wire [15:0] cc0, cc1, cc2, cc3, cc4, cc5, cc6, cc7, cc8, cc9, cca, ccb, ccc, ccd, cce, ccf;
  wire [15:0] cd0, cd1, cd2, cd3, cd4, cd5, cd6, cd7, cd8, cd9, cda, cdb, cdc, cdd, cde, cdf;
        coeff_full coeff_rom (
    .coord0(coord), .coord1(coord), .coord2(coord), .coord3(coord),
    .clock(clock_36mhz), .reset(reset), .ready(ready),
         .a0(ca0), .a1(ca1), .a2(ca2), .a3(ca3), .a4(ca4), .a5(ca5), .a6(ca6), .a7(ca7),
         .a8(ca8), .a9(ca9), .aa(caa), .ab(cab), .ac(cac), .ad(cad), .ae(cae), .af(caf),
    .b0(cb0), .b1(cb1), .b2(cb2), .b3(cb3), .b4(cb4), .b5(cb5), .b6(cb6), .b7(cb7),
    .b8(cb8), .b9(cb9), .ba(cba), .bb(cbb), .bc(cbc), .bd(cbd), .be(cbe), .bf(cbf),
    .c0(cc0), .c1(cc1), .c2(cc2), .c3(cc3), .c4(cc4), .c5(cc5), .c6(cc6), .c7(cc7),
    .c8(cc8), .c9(cc9), .ca(cca), .cb(ccb), .cc(ccc), .cd(ccd), .ce(cce), .cf(ccf),
    .d0(cd0), .d1(cd1), .d2(cd2), .d3(cd3), .d4(cd4), .d5(cd5), .d6(cd6), .d7(cd7),
    .d8(cd8), .d9(cd9), .da(cda), .db(cdb), .dc(cdc), .dd(cdd), .de(cde), .df(cdf),
    .coeffs_valid(coeffs_valid)
    );

  wire [15:0] a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, aa, ab, ac, ad, ae, af;
  //wire [31:0] b0, b1, b2, b3, b4, b5, b6, b7, b8, b9, ba, bb, bc, bd, be, bf;
  //wire [31:0] c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, ca, cb, cc, cd, ce, cf;
  //wire [31:0] d0, d1, d2, d3, d4, d5, d6, d7, d8, d9, da, db, dc, dd, de, df;
  wire enc0r;
 enc_multiplier enc0 (
    .sample(sample),
    .clock(clock_36mhz),
    .reset(reset),
    .ready(ready),
    .e0(a0), .e1(a1), .e2(a2), .e3(a3), .e4(a4), .e5(a5), .e6(a6), .e7(a7),
    .e8(a8), .e9(a9), .ea(aa), .eb(ab), .ec(ac), .ed(ad), .ee(ae), .ef(af),
         .c0(ca0), .c1(ca1), .c2(ca2), .c3(ca3), .c4(ca4), .c5(ca5), .c6(ca6), .c7(ca7),
    .c8(ca8), .c9(ca9), .ca(caa), .cb(cab), .cc(cac), .cd(cad), .ce(cae), .cf(caf),
    .encoder_valid(enc0r), .c_valid(coeffs_valid)
    );
```

```
        //decoder ready signals
  wire dec0r;
  wire signed [23:0] chan1;

        decoder #(.Y(6'sd31), .X(6'sd31), .Z(5'sd0)) dec0 (
    .s0(a0), .s1(a1), .s2(a2), .s3(a3), .s4(a4), .s5(a5), .s6(a6), .s7(a7),
    .s8(a8), .s9(a9), .sa(aa), .sb(ab), .sc(ac), .sd(ad), .se(ae), .sf(af),
    .summing_valid(enc0r),
    .ready(ready),
    .clock(clock_36mhz),
    .reset(reset),
    .audio_out(chan1),
    .decoder_valid(dec0r)
    );

        wire [15:0] pcm_signal = chan1;


        initial begin
                // Initialize Inputs
                coord = {6'sd4, 6'sd4, 5'sd0};
                sample = 0;
                clock_36mhz = 0;
                reset = 1;
                ready = 0;

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

                // Add stimulus here]
                reset = 0;
        end

        always #1 clock_36mhz = ~clock_36mhz;
        always #768 begin
                ready = 1;
                #2 ready = 0;
        end
   always @(posedge ready) sample = sample + 10;
endmodule
```

```verilog
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:   23:15:57 11/15/2009
// Design Name:   usb_reader
// Module Name:   C:/bb/usb_stereo_audio/usb_fsm_tester.v
// Project Name:  usb_stereo_audio
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: usb_reader
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module usb_fsm_tester_v;

	// Inputs
	reg rxf;
	reg ready;
	reg clock;
	reg reset;
	reg [7:0] data;
	reg [7:0] data_count;

	// Outputs
	wire [7:0] led;
	wire [63:0] audio_out_data;
	wire rd;

	// Instantiate the Unit Under Test (UUT)
	usb_reader uut (
		.rxf(rxf),
		.ready(ready),
		.clock(clock),
		.reset(reset),
		.data(data),
		.led(led),
		.audio_out_data(audio_out_data),
		.rd(rd)
	);

	initial begin
		// Initialize Inputs
		rxf = 0;
		ready = 0;
		clock = 0;
		reset = 1;
		data = 0;
		data_count = 0;

		// Wait 100 ns for global reset to finish
		#100;
		reset = 0;

		// Add stimulus here

	end
```

```
        always #1 clock = ~clock;

        always #501 begin
                ready = 1;
                #1 ready = 0;
        end

        always @(negedge rd) begin
                #1 rxf = 0;
                #1 data_count = data;
                #5 rxf = 1;
        end

        always @(posedge rd) begin
                #1 data = data_count + 1;
        end


endmodule
```

Appendix B: Python Implementation

```python
import pylab as p
import numpy
from matplotlib.colors import LogNorm

x = range(-32, 32)
y = range(-32, 32)
z = range(-16, 16)

#ensure we will not overflow
test_gain = 0.96
exp = 15
data = {}
s = [(31,31,0), (31,-31,0), (-31,31,0), (-31,-31,0)]

def generate_coeffs():
    global data
    for xc in x:
        for yc in y:
            for zc in z:
                if 1==1:
                    azi = p.arctan2(yc,xc)
                    ele = p.arctan2(float(zc),(float(xc**2.0 + yc**2.0)**0.5))
                    dis = (xc**2.0+yc**2.0+zc**2.0)**0.5
                    #first order harmonics
                    wh = 1.0/2.0**0.5
                    xh = p.cos(azi)*p.cos(ele)
                    yh = p.sin(azi)*p.cos(ele)
                    zh = p.sin(ele)
                    #second order harmonics
                    rh = (3*p.sin(ele)**2.0 - 1.0) / 2.0
                    sh = p.cos(azi) * p.sin(2.0*ele)
                    th = p.sin(azi) * p.sin(2.0*ele)
                    uh = p.cos(2.0*azi) * p.cos(ele)**2.0
                    vh = p.sin(2.0*azi) * p.cos(ele)**2.0
                    #third order harmonics
                    kh = p.sin(ele) * ((5.0 * (p.sin(ele)**2.0))-3.0) / 2.0
                    lh = p.sqrt(3.0/8.0) * p.sqrt(45.0/32.0) * p.cos(azi) * p.cos(ele) * (5.0* p.sin(ele)**2.0 -1.
                    mh = p.sqrt(3.0/8.0) * p.sqrt(45.0/32.0) * p.sin(azi) * p.cos(ele) * (5.0* p.sin(ele)**2.0 -1.
                    nh = (p.sqrt(15.0)/2.0) * (3.0/p.sqrt(5.0)) * p.cos(2.0*azi) * p.sin(ele) * p.cos(ele)**2
                    oh = (p.sqrt(15.0)/2.0) * (3.0/p.sqrt(5.0)) * p.sin(2.0*azi) * p.sin(ele) * p.cos(ele)**2
                    ph = p.cos(3.0*azi) * p.cos(ele)**3.0
                    qh = p.sin(3.0*azi) * p.cos(ele)**3.0


##                      #build the data array
##                      data[(xc,yc,zc)] = [
##                                   (azi, ele, dis),
##                                   (wh*0.823242,xh,yh,zh),
##                                   (rh*0.442259,sh*0.442259,th*0.442259,uh*0.442259,vh*0.442259),
##                                   (kh*0.079101,lh*0.079101,mh*0.479101,nh*0.079101,oh*0.079101,ph*0.079101,qh*0.0

                    #build the data array
                    data[(xc,yc,zc)] = [
                                 (azi, ele, dis),
                                 (wh,xh,yh,zh),
                                 (rh,sh,th,uh,vh),
                                 (kh,lh,mh,nh,oh,ph,qh)]
                else:
                    data[(xc,yc,zc)] = [
                                 (0, 0, 0),
                                 (1.0,0,0,0),
                                 (0,0,0,0,0),
                                 (0,0,0,0,0,0,0)]

def get_coeffs(x,y,z):
    [(azi, ele, dis),
     (wh,xh,yh,zh),
     (rh,sh,th,uh,vh),
     (kh,lh,mh,nh,oh,ph,qh)] = data[(x,y,z)]

    print "========= AMBISONIC 3RD ORDER COEFFICIENTS =========="
    print "   Position:", x, y, z
    print "   --------------------------------------------------"
    print "   W: %2f   \t  16'd" % wh, int(wh*2**exp), "\t ", hex(int(wh*2**exp))
    print "   X: %2f   \t  16'd" % xh, int(xh*2**exp), "\t ", hex(int(xh*2**exp))
    print "   Y: %2f   \t  16'd" % yh, int(yh*2**exp), "\t ", hex(int(yh*2**exp))
    print "   Z: %2f   \t  16'd" % zh, int(zh*2**exp), "\t ", hex(int(zh*2**exp))
    print "   R: %2f   \t  16'd" % rh, int(rh*2**exp), "\t ", hex(int(rh*2**exp))
    print "   S: %2f   \t  16'd" % sh, int(sh*2**exp), "\t ", hex(int(sh*2**exp))
    print "   T: %2f   \t  16'd" % th, int(th*2**exp), "\t ", hex(int(th*2**exp))
    print "   U: %2f   \t  16'd" % uh, int(uh*2**exp), "\t ", hex(int(uh*2**exp))
    print "   V: %2f   \t  16'd" % vh, int(vh*2**exp), "\t ", hex(int(vh*2**exp))
    print "   K: %2f   \t  16'd" % kh, int(kh*2**exp), "\t ", hex(int(kh*2**exp))
    print "   L: %2f   \t  16'd" % lh, int(lh*2**exp), "\t ", hex(int(lh*2**exp))
    print "   M: %2f   \t  16'd" % mh, int(mh*2**exp), "\t ", hex(int(mh*2**exp))
    print "   N: %2f   \t  16'd" % nh, int(nh*2**exp), "\t ", hex(int(nh*2**exp))
    print "   O: %2f   \t  16'd" % oh, int(oh*2**exp), "\t ", hex(int(oh*2**exp))
    print "   P: %2f   \t  16'd" % ph, int(ph*2**exp), "\t ", hex(int(ph*2**exp))
    print "   Q: %2f   \t  16'd" % qh, int(qh*2**exp), "\t ", hex(int(qh*2**exp))

def gen_preset(x,y,z,preset_num):
    coeff_data = data[(x,y,z)]
    coeffs = coeff_data[1]+coeff_data[2]+coeff_data[3]
    dis = coeff_data[0][2]
    if dis > 20:
        dis_gain = (44.0 - (dis-20.0))/44.0
    else:
        dis_gain = 1.0
    s = ""
    s += "\t\t//Generated Preset "+ preset_num +"\n"
    #s += str(("\t\t//Location:",x,y,z)) + "\n"
    for i in range(16):
        s+= "\t\tc["+str(i)+"]["+str(preset_num)+"] <= 16'sh"+ str(hex(int(dis_gain*test_gain*coeffs[i]*2**exp))).
    return s

def gen_coe_preset(x,y,z,i):
    coeff_data = data[(x,y,z)]
    coeffs = coeff_data[1]+coeff_data[2]+coeff_data[3]
    dis = coeff_data[0][2]
    if dis > 20:
        dis_gain = (44.0 - (dis-20.0))/44.0
    else:
        dis_gain = 1.0
    fc = test_gain*coeffs[i]*2**exp
    if fc > 32760:
        fc = 32767
    elif fc < -32760:
        fc = -32768
    fc = int(dis_gain*fc)
    return str(fc)+",\n"

def make_preset_list(l):

    case_statement_header = """
   //This module provides staic coefficients
   always @* begin
      case (coord)
        default: index_select = 0;
    """
    case_statement_footer = '''
      endcase // case(coord)
   end
   '''

    init_statement_header = """
   //Here we define the staic constants
   always @(posedge clock) begin
      if (reset) begin
    """
    init_statement_footer = '''
      end
   end
    '''
    count = 0
    print case_statement_header
    for (x,y,z) in l:
        print   "\t\t{6'sd"+str(x)+", 6'sd"+str(y)+", 5'sd"+str(z)+"}: index_select ="+str(count)+";"
        count += 1
    print case_statement_footer

    count = 0
    print init_statement_header
    for (x,y,z) in l:
        print gen_preset(x,y,z,count)
        print
        count += 1
    print init_statement_footer

def calc_gain(vs, ps, sample):
    coeff_data = data[vs]
    dis = coeff_data[0][2]
    if dis > 20:
        v_dis_gain = (44.0 - (dis-20.0))/44.0
    else:
        v_dis_gain = 1.0
    vs_coeffs = [i*v_dis_gain*test_gain for i in coeff_data[1]]+[i*test_gain for i in coeff_data[2]]+[i*test_gain
    coeff_data = data[ps]
    dis = coeff_data[0][2]
    if dis > 20:
        p_dis_gain = (44.0 - (dis-20.0))/44.0
    else:
        p_dis_gain = 1.0
    ps_coeffs = [i*p_dis_gain*test_gain for i in coeff_data[1]]+[i*test_gain for i in coeff_data[2]]+[i*test_gain

    vs_samples_encoded = []
    for i in vs_coeffs:
        vs_samples_encoded.append(i*sample)

    output_sample = sum([vs_samples_encoded[i]*ps_coeffs[i] for i in range(len(ps_coeffs))])
    #print "P distance gain:", p_dis_gain, "\tV dis gain:", v_dis_gain
    return output_sample

def calc_gain_for_plot(l, x, y):
    Z = p.array([calc_gain(l, (int(i),int(j),l[2]), 1.0) for i in x.flat for j in y.flat])
    Z.shape = (64, 64)
    return Z.transpose()

def plot_gain(l):
    N=64
    x = p.linspace(-32.0, 31.0, N)
    y = p.linspace(-32.0, 31.0, N)

    X, Y = p.meshgrid(x,y)
    Z = calc_gain_for_plot(l, x, y)

    ax = p.subplot(111)
    im = p.imshow(Z, cmap=p.cm.jet)
    im.set_interpolation('bilinear')

    p.show()

def plot_azi():
    x = p.arange(-32.0, 31.0, 1.0)
    y = p.arange(-32.0, 31.0, 1.0)
    X,Y = p.meshgrid(x,y)
    Z = calc_azi(X,Y)

    ax = p.subplot(111)
    im = p.imshow(Z, cmap=p.cm.jet)
    im.set_interpolation('bilinear')

    p.show()

def calc_azi(x,y):
   return p.arctan2(y,x)

def calc_speaker_gain(speakers, point):
    for s in speakers:
        print "Speaker:", s, "\tGain:", calc_gain(point, s, 1.0)

def full_preset_list():
    f = open("presets.txt", "w")
    init_statement_header = """
   //Here we define the staic constants
   always @(posedge clock) begin
      if (reset) begin
    """
    init_statement_footer = '''
      end
   end
    '''
    f.write(init_statement_header + "\n")
    for (x,y,z) in [(x,y,z) for x,y,z in data.keys() if z==0]:
        f.write(gen_preset(x,y,z,"{6'sd"+str(x)+", 6'sd"+str(y)+"}"))
    f.write(init_statement_footer)
    f.close()

def coe_file(channel):
    f = open("ambi_block"+str(channel)+".coe","w")

    f.write("memory_initialization_radix=10;\n")
    f.write("memory_initialization_vector=\n")
    vec = range(32) + range(-32,0)
    for i in vec:
        for j in vec:
            f.write("; preset "+str(i)+" "+str(j)+"\n")
            f.write(gen_coe_preset(i,j,0,channel))
    f.close()

def coe_all():
    for i in range(16):
        coe_file(i)
```

```
#! /usr/bin/env python
#import serial
import d2xx
import wave
import aifc

## create a serial connection
#ser = serial.Serial('/dev/tty.usbserial-DPD52KUH', 115200)
#ser = serial.Serial('/dev/tty', 115200)
ser = d2xx.open(0)
ser.setBaudRate(921600)
ser.setTimeouts(1000,300000)


#w = wave.open('01 Headlock.wav','r')
#w = wave.open('morning_side04.wav','r')
#w = wave.open('morning_side04_16bit.wav','r')
#w = wave.open('4chantest_16.wav','r')
#w = wave.open('1chantone3chansilence.wav','r')
#w = wave.open('4chantone.wav','r')
#w = wave.open('3chantone.wav','r')
#w = wave.open('4chansingletone.wav','r')
#w = wave.open('1000hz.wav','r')
#w = wave.open('save.wav','r')
w = wave.open('save_block.wav','r')


#for i in range(480000):
#    if len(w.readframes(1)) != 6:
#        print "AHHH"

#for i in range(4000):
#    s = w.readframes(480)
#    r = ser.write(s)

#wlen = 96000
while (w.tell() < w.getnframes()):
    s = w.readframes(4800000)
    r = ser.write(s)

#a = ( "\x00","\x00","\x00", "\x00","\x00","\x00","\x00","\x00","\x00", "\x00","\x00","\x00","\xff","\xff","\xff",
#a= ("\x00", "\x00", "\x00", "\x00", "\x00", "\x00", "\x00", "\x00",
#   "\xff", "\xff" , "\xff", "\xff", "\xff", "\xff" , "\xff", "\xff")
#b= ("\x55" ,"\x55" ,"\xaa" ,"\xaa")
#c= ("\xaa" ,"\xaa" ,"\x55" ,"\x55")
#d= ("\xff" ,"\xff" ,"\x00" ,"\x00")

#a =  ("\xff", "\xff" , "\xff", "\xff", "\xff", "\xff" , "\xff", "\xff" )
#a =  ("\xaa", "\xff" , "\xff", "\xff", "\xff", "\xff" , "\xff", "\xff" )
#a= ("\xff", "\xff")
#a= ("\x00", "\x00")
#s = ''.join(a*480000)
#s = w.readframes(w.getnframes())
#r = ser.write(s)

ser.close()
```