MIT Course 6.111: Digital Electronics Lab
Virtual Surround Sound

Harrison King Hall
Candidate for B.S. in Computer Science in 2008
Massachusetts Institute of Technology
hkhall@mit.edu

December 13, 2006

Abstract

Completely immersive media not only requires lifelike video but audio as well to convince the user that they are actually within the scene. We develop a system intent upon providing immersive audio for devices capable of playing media that utilize location-aware audio signals and output their audio to stereo headphones. The system can take streams of pulse-code modulation(PCM) digital audio and a continuously variable azimuth to the signal source and compute in real-time the appropriately modulated signal to simulate a virtual speaker playing the stream of audio at that location relative to the user.

The manner after which we simulate surround sound is unique in that it uses a solid spherical model of the users head in order to generate the modulated audio. The signal is the result of three functions on the audio input: an inter-aural frequency shift(ILD) mutes or boosts high frequency signals, an inter-aural time delay(ITD) between the arrival time of a signal at each ear, and a room echo model that simulates the reflections in both time delay and magnitude of an audio signal. The input and outputs from the ILD and ITD are cascaded in order and then summed with the room echo. We provide this new *virtual speaker* sample for output via the AC97 in real time.

We implemented our design on a Xilinx FPGA, using the 6.111 labkit. The design is highly modular: as long as port specifications are met the method by which we calculate the ITD, ILD and room echo models could all be replaced without modifying the other parts of the system. The audio inputs to the system could also be from various sources available on the Labkit or another accessory module.

Contents

List of Tables

List of Figures

1 Introduction

High-Definition(HD) audio and video sources have come to enjoybuzzword status in the home electronic market recently. Systems that are described as such typically produce extremely high fidelity audio that will astound even the most fanatical proponent of pure analog audio, but this level of quality comes at a price. Most notably it requires a dedicated space in which considerable investments in both infrastructure and funds must be made. As a result, there are relatively few properly installed, quality surround sound audio systems in the consumer market.

In an attempt to make the technology more accessible and acceptable to those ill equipped or unwilling to make these investments, virtual surround sound emerged. These systems attempted to fool the user's ear into believing that an audio source was coming from a *phantom speaker*. Companies such as Dolby, Bose, and Denon have implemented home-theater-in-a-box that use stereo speaker setups to achieve believable results at a minimal infrastructure investment. These solutions are still not single-user portable, however. Headphones, on the other hand, provide the same stereo sound capable of being reproduced on a tradition two-speaker stereo system but in a much more compact package. Our focus in the final project is on the confined space of stereo headphones.

Traditionally, the manner in which one talks about how a sound is localized to be at some point in space relative to the user is with reference to an individualized, complex function of frequency and three spatial variables known as the head-related impulse response(HRIR), or its more common frequency domain counterpart the head-related transfer function(HRTF). This method is a very time consuming procedure that requires expensive and specialized equipment. Further this calculation's output is not general enough to be applicable any other users on a large scale. Instead of taking this single user approach we explore whether it is possible to capture a generalized physical model of how the data is transformed in digital hardware.

2 Theory of Virtual Surround Sound

Each user of an audio system differs from all others by some physical characteristics. Of these the most significant are head size, head shape, ear shape, torso shape but there are many other factors such as hairstyle and height that play a minimal role. These variations between users mean that, at least initially, an virtual surround sound module might not seem to work for a user though it is calculating the correct data for the model.

The usual method for simulating surround sound uses a measured HRIR/HRTF to localize the sound. The programs use the Convolvotron method shown in Figure 2 the measured HRIR data with a source PCM signal stream. In an attempt to generalize all of the measured HRIRs, often the results are . As you can see from Figure 1 That as the number of frequencies grows the amount of data stored increases linearly. There is no such problem for the Convolvotron, who functions in the time domain, since for each azimuth

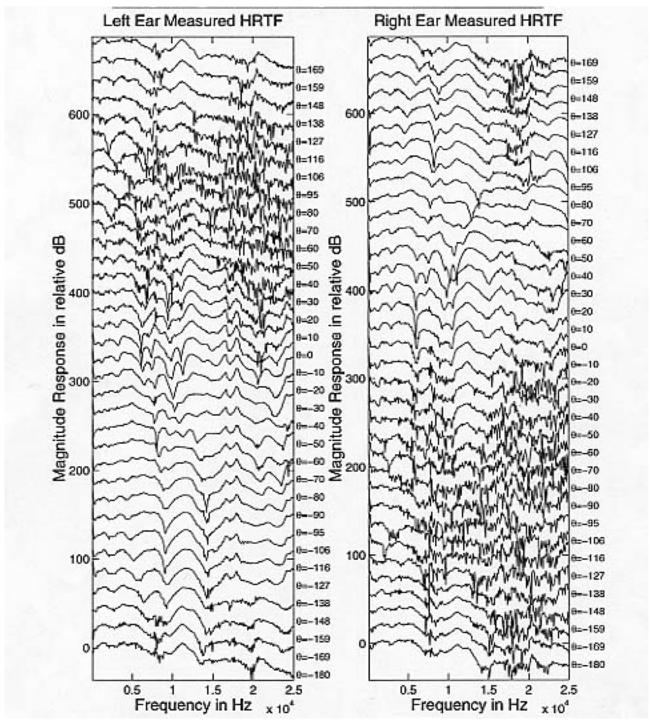

Figure 1: Example of a single measured HRTF dataset. Image courtesy of http://www.ece.rice.edu/ crozell/

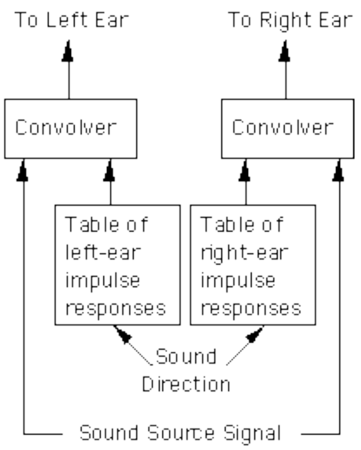

Figure 2: The Convolvotron model for HRIR dataset. Image courtesy of http://interface.cipic.ucdavis.edu/

there is a single value that determines how that signal should be changed. While this seems simple, obtaining quality and matching HRIR data for the user can be challenging as they are often proprietary and provide little to no information about the person it was sampled on. This makes it impossible other to determine the single best fits a user other than the brute-force search or random assignment methods.

2.1 Alternative Approach

To overcome this of a lack of data and ease of matching data to a user, we sought a generalized model based upon easily measurable human feature parameters. The spherical head model is the most obvious generalization of the system that still provides valuable and important data. There are two equations that govern azimuth cues for spatial audio that were discovered and solved approximately 100 years ago by Lord Rayleigh(John Strutt). They are essentially the time difference between when signals arrive at each ear(Interaural Time Difference) and the volume difference due to the head disrupting the wave's path and the added distance the wave must travel(Inter-aual Level Difference).

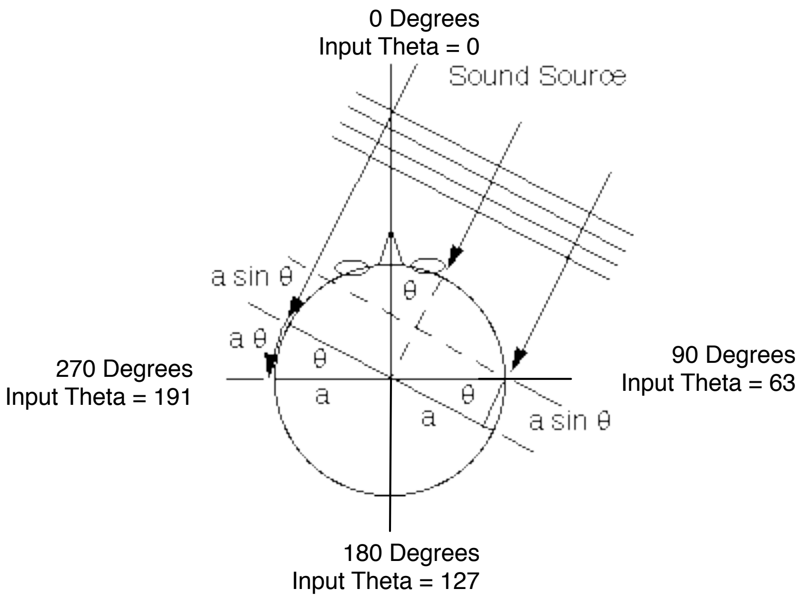

Figure 3: The manner sound wave interacting with a spherical model of the head and how we define Θ. Image courtesy of http://interface.cipic.ucdavis.edu/

It is of interest to note that these two equations are complimentary in that the ITD and ILD impact different portions of the frequency spectrum. The ILD is highly frequency dependent. For signals above $1.5kHz$ and results in up to a $20 - dB$ difference between the ear closer to the sound source and the ear in the shadow of the head relative to the signal source. Lower frequencies are less impacted due to their longer wavelength, thus those below $1.5kHz$ remain unchanged and make it more difficult to locate audio at these frequencies. The ITD, simply shifts the wave a some Θ-dependent amount of cycles. For frequencies below $1.5kHz$ the wavelength is long relative to head size so the fraction of a cycle shift is noticeable and aids in direction determination. For higher frequencies the shift becomes multiple cycles and so there is some ambiguity as to where the sound is originating: at Θ or $\Theta + \pi$.

The complimentary nature of these equations leads us to conclude a combination of them is necessary if the design is to cover all audible frequencies, generally accepted to be $20Hz - 20kHz$, equally well.

2.1.1 Inter-aural Time DIfference

Incident sound waves are diffracted by the head and results in a time delay of the waves that is dependent on Θ. Figure 2.1 shows the necessary geometry to calculate the distance difference to each ear for some sound signal placed in the *far field* of distances beyond 1.5 meters. The time delay can be determined according to the following equation:

$$\begin{aligned} T_{delay}(\Theta) &= \frac{a}{c} \cdot \Big(1 + \cos(\Theta + \phi)\Big) && (1) \\ a &= \text{radius of user's head} && (2) \\ c &= 340\frac{\text{meters}}{\text{second}} && (3) \\ \phi &= \begin{cases} \frac{\pi}{2} & \text{if the audio signal is for the left ear} \\ -\frac{\pi}{2} & \text{if the audio signal is for the right ear} \end{cases} && (4) \end{aligned}$$

Since the frequencies that are most effected by this modification have long wavelengths there is only negligible signal strength loss as the signal travels around the head. Therefore we can ignore that effect in this model.

2.1.2 Inter-aural Level Difference

Since as was discussed earlier the ILD quiets the higher frequencies while leaving the lower frequencies untouched. Thus the equation governing the ILD coefficient multiplied by the signal source exists in frequency domain. This allows us to make certain simplifications that greatly decrease the mathematical complexity.

$$H(s, \theta) = \frac{\alpha(\Theta) \cdot s + \beta}{s + \beta} \qquad (5)$$

$$\begin{aligned}
&= \frac{\alpha(\Theta)\cdot s+\beta}{s+\beta}\cdot\frac{\beta-s}{\beta-s} && (6)\\
&= \frac{\beta\cdot\alpha\cdot s-\alpha\cdot s^2+\beta^2-\beta\cdot s}{\beta^2-s^2} && (7)\\
H_{real}(s,\theta) &= \frac{\alpha\cdot\omega^2+\beta^2}{\beta^2+\omega^2} && (8)\\
&\approx 1 && (9)\\
H_{imaginary}(s,\theta) &= \frac{\omega\cdot\beta\cdot\alpha-\omega\cdot\beta}{\beta^2+\omega^2} && (10)\\
&\approx \frac{\omega\cdot\cos(\Theta+\phi)}{\beta} && (11)\\
\omega &= \text{Sampled frequency between } \{0\ldots 2\cdot\pi\} && (12)\\
\beta &= 2\cdot\frac{c}{a} && (13)\\
\alpha(\Theta) &= 1+\cos(\Theta+\phi) && (14)
\end{aligned}$$

What this tells us is really interesting, namely that we only need modify the imaginary components of the ILD in the frequency domain to fulfill its functionality.

2.1.3 Room Echo

Essentially this is an additional model that allows for reflections off of the surfaces in the space that the audio is being observed in. Since we can model one person viewing audio from the center of a square room we can apply the same modified signal to both channels of output. It is clear that an echo in a sealed room is very difficult to pinpoint the location of because the wave is reflected off of all the surfaces and all of this reflection causes significant signal strength loss. Thus this signal modification is similar to the ITD with a much longer delay and a magnitude loss.

For the average sized room with typical audio characteristics we can characterize the echo as:

$$T_n = \frac{T_{n-10milliseconds}}{4} \quad (15)$$

2.1.4 Algorithm

In order for the ILD and ITD to function on a single audio signal we must cascade them. The ILD and ITD cannot function in parallel on the same input and then be combined to produce the correct output. Their very nature of being complementary for an input seems to imply that they must be cascaded so that the output of the ILD flows to the input of the ITD for each ear's audio path for each individual audio input in the system. Then through some handshaking provided by the sample request signal they can pass data with assurance that it is valid. This description of the handshaking is only necessary if we

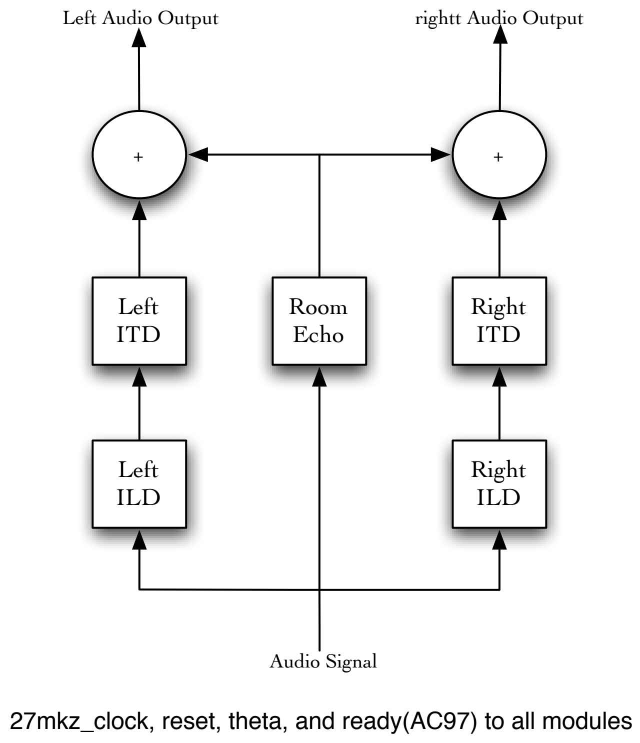

Figure 4: The flow of audio information in a single implementation of the ILD, ITD and Room Echo models. This also is the Block diagram for the Mono Source module because we chose to compute the data serially.

attempt to step though the entire pipeline in one sample request cycle. In this design this is never necessary because all audio after 8 sample request cycles all data in the system will be delayed will be delayed exactly 8 sample request cycles, although we included the ports for clarity and modularity in other applications.

Utilizing this block buildup structure, we have established an algorithmic way for each source to be handled independent of all the others and exploit the mass parallelism of the FPGA. Thus, we can handle multiple streams of audio independently and then sum all of the samples for an ear and output this single sample representing the total audio encountered by that ear this sample time. Thus an implementation of 5 speaker surround sound looks like Figure 2.1.4 which is the project ultimate goal block diagram.

3 Description

This section describes the operation of the Labkit by the user for system functionality and how each module functions.

3.1 Operation

It was designed such that there is very little interaction of the user with the Labkit. This is mainly because it is counter productive when your intent is to have the user suspend their disbelief that they are in a theater environment. Thus, for the highest goal of multiple sources, the only interaction would be volume command and the system reset command. Unfortunately since there were persistent errors in the lower level blocks we were unable to simplify the input output scheme to this level.

The modified scheme controls the azimuth of the audio source as well as the volume and reset commands. Further the user is able to switch between the four completed modules such that they can hear the output of that single module input of all others.

One of the original goals of the project was to get audio on the board using an optical TOSlink connector and decode the AC-3 Frames that come from it. Mid-project this goal was discarded because the complexity of decoding Dolby Digital® is more than can be implemented within the time constraints of 6.111. Therefore I simplified the user's audio inputs to the AC97 microphone input or the $750Hz$ tone sine wave that was provided to the class by the staff. The use of the latter was primarily for debugging as it has known values for every sample.

3.2 Design Criteria

While the few setup clock cycle outputs of the system are largely inconsequential, after some finite period of sample requests by the AC97 module we must be able to supply a real-time(though delayed), continuous stream of PCM samples. With the flow of information and pipelining as described in Figure 2.1.4 each module has at most $\frac{\text{System Clock}}{\text{Audio Request Frequency}}$

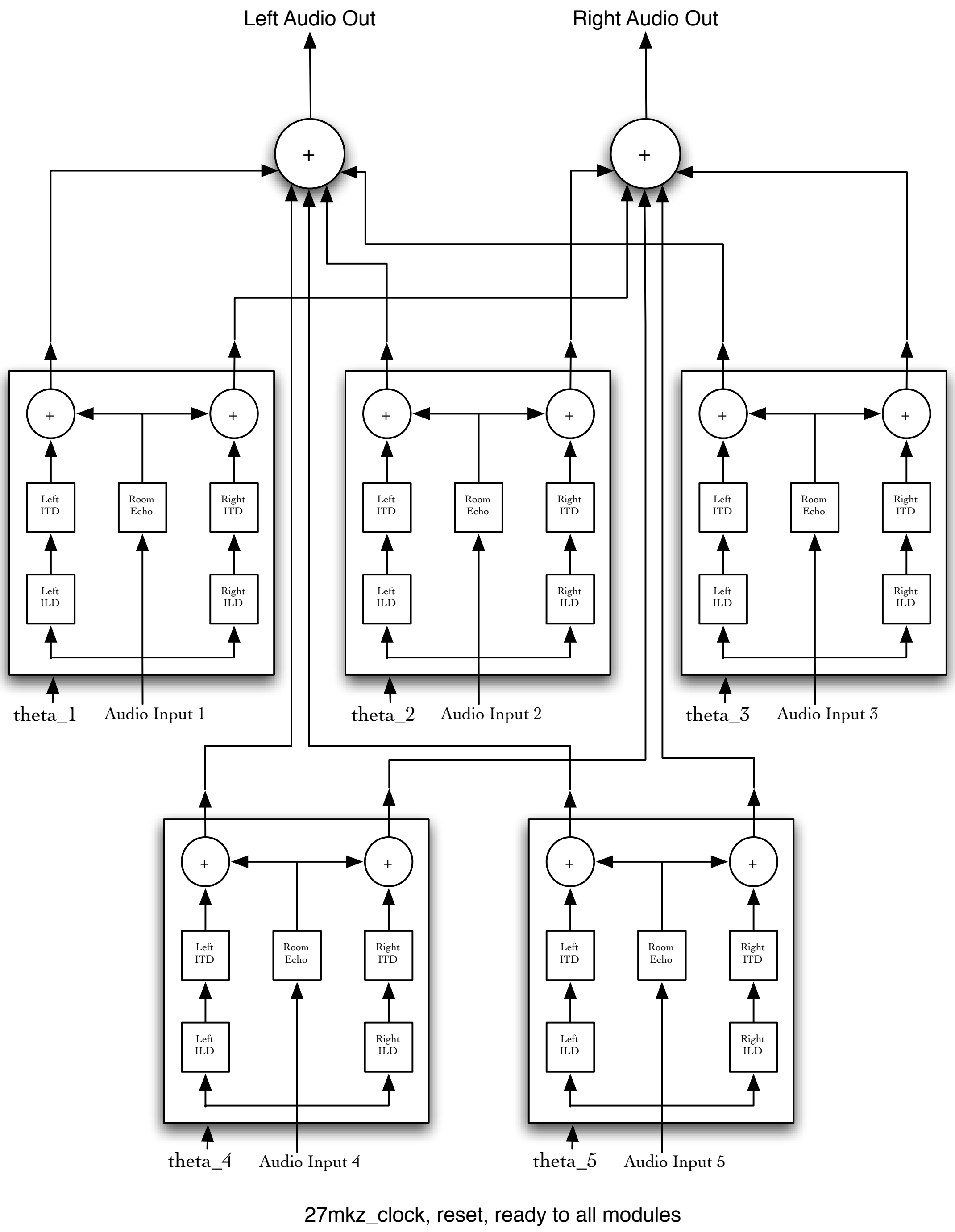

Figure 5: The block diagram for the Surround Sound module.

Table 1: User Controls

Desired Effect	Button or Switch	Action or State
Reset	$Button_{Enter}$	Depress
Increase Volume	$Button_{Up}$	Depress
Decrease Volume	$Button_{Down}$	Depress
Increase Azimuth	$Button_{Right}$	Depress
Decrease Azimuth	$Button_{Left}$	Depress
Hear ITD Only	$Switch[1:0]$	2'b00
Hear ITD Only	$Switch[1:0]$	2'b01
Hear Room Echo Only	$Switch[1:0]$	2'b10
Hear Combined Model for Single Source	$Switch[1:0]$	2'b11

Table 2: User Audio Source Controls

Audio Source	Assignment of $Switch_7$ to Use Input
$750Hz$ Sine Wave	0
Microphone In	1

clock cycles to perform its operations. For our particular implementation we chose to use the $27MHz$ system clock and the AC97's request signal is specified to cycle at $48kHz$. Therefore each module has at most 562 clock cycles to accept data, compute what effect this model has on it, and register it as output. Of course had this constraint come close to being met we could have increased the system clock's frequency in order to gain more clock cycles.

Also, with the large number of sizable Coregens that this project uses it is necessary that they all fit on the 6.111 Labkit that, though large, is not infinitely big.

3.3 Conventions

Based on the input and output capabilities of the AC97 interface, we chose to set the PCM sample format to 8-bit signed data.

3.4 Module Descriptions

3.4.1 AC97 Module

While the majority of this module remains unchanged from the version provided to the class by the 6.111 staff in order to output stereo sound I needed to modify the port assignments by adding an additional one for the other stream of PCM data that I was outputting. The only other modification I made was to differentiate the assignment of these inputs for the AC97 Framemaker.

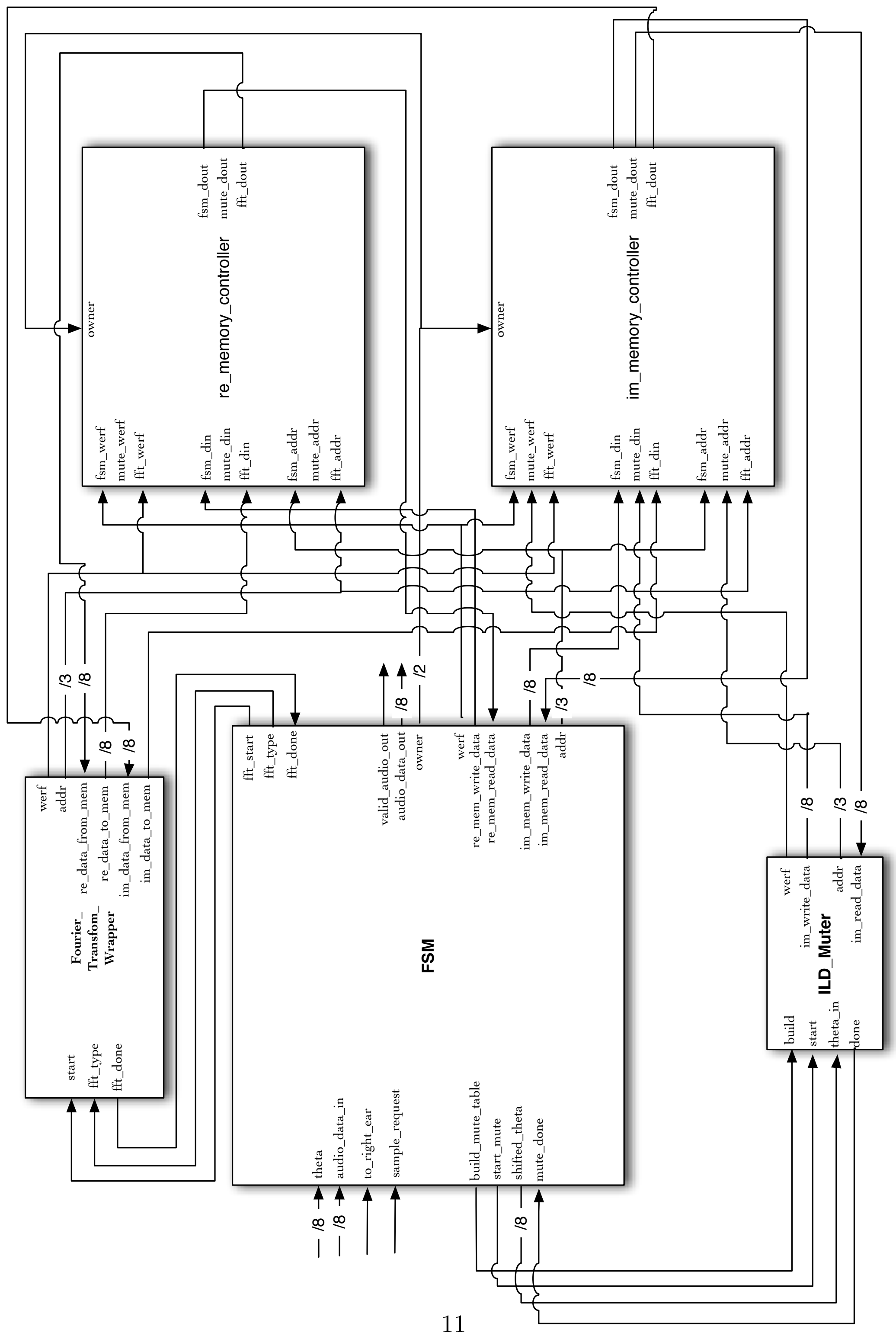

Figure 6: The block diagram for the ILD Module.

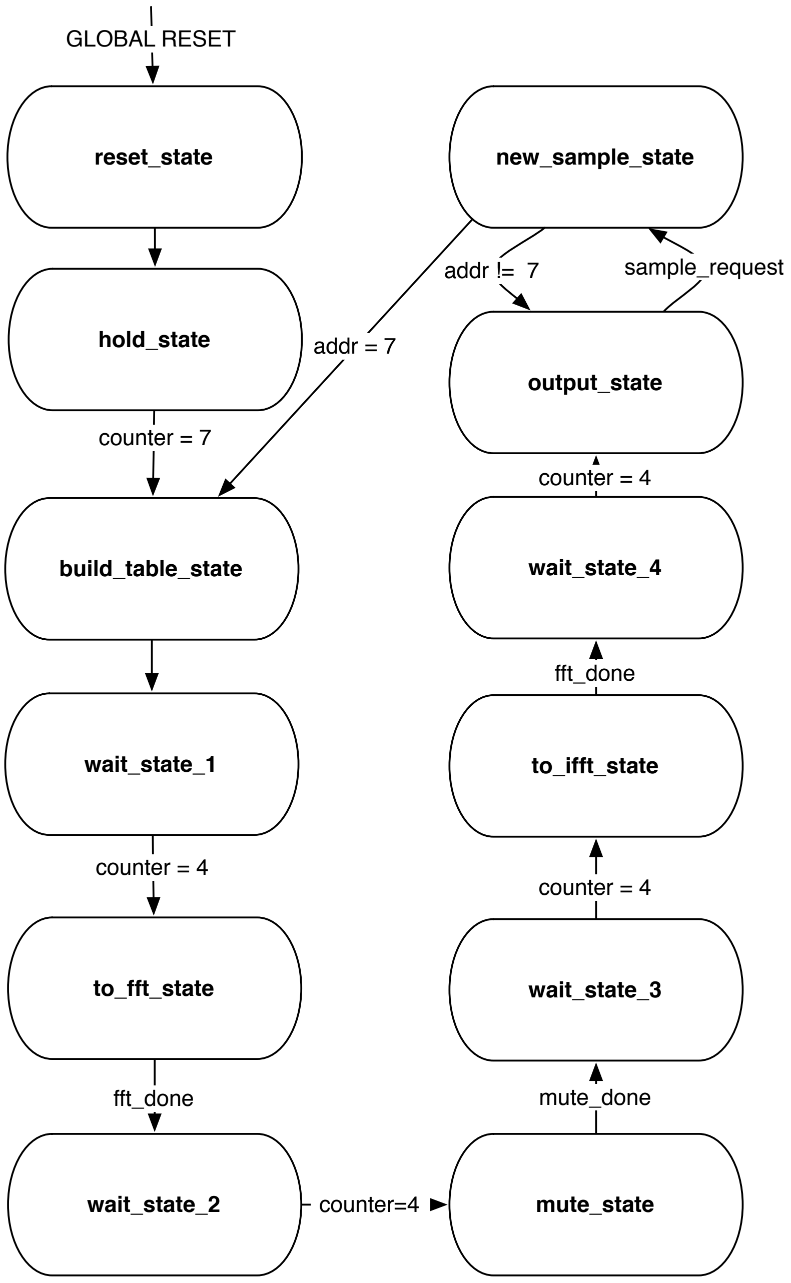

Figure 7: The finite state machine for the ILD module.

3.4.2 ILD Module

The overall idea is to read eight samples into memory. After the last sample is stored we will build a look up table of the Cosine values for the Θ last PCM sample that is being put into the system. Once this table is calculated then to preform the FFT on all the samples getting out the eight real and imaginary coefficients and storing these to a global to the ILD submodules memory. Then for each of the imaginary coefficients we will be multiplying them by the mute coefficient that we calculated and stored in the look up table and writing this back to the global imaginary memory. We then perform the inverse Fourier transform on the samples we have stored in memory. We store both the real an imaginary coefficients in their respective memories. Then on every sample request we output the next real part of the transformed data in a strictly increasing fashion from 0 to 7 and take the new input and store it in the location we just read from being sure to tie the imaginary portion to 8'b0.

The module diagram and FSM for the ILD module can be seen in Figure 3.4.1 and Figure 3.4.1 respectively. As you can see that there are several modules that are controlled by a FSM that passes the memory control holds in a state until it gets what to a done signal from the module. There are also two minor FSMs incorporated into the design of the FFT Wrapper and the ILD Muter submodules module. The functionality of these modules is described below in detail.

Table 3: Specification of Input Port fft_type on Fourier Wrapper

Direction	fft_type's Value
Time to Frequency	1
Frequency to Time	0

Fourier Wrapper Sub-Module You can see the Fourier Wrapper interfaces with the FFT Coregen. The control signals for the coregen FFT are difficult to decipher, that is why I chose to wrap the module with this file. The basic operation is simplified to wiring the Wrapper module to a memory using the obvious port names and asserting the direction that you would like the transform to go from as shown in Table 3 and that you would like for the transform to start on the same cycle. The module itself will read from and write to memory as appropriate.

The size of the transform was the subject of much internal debate within the team. Although the longer the transform size the more accurate the result it takes significantly longer to calculate. For the current system clock we were running on, assuming that you could mute all of the samples in time through parallel processes the maximum length transform is 64 units long. This is pushing the limits of the clock, however so we are not

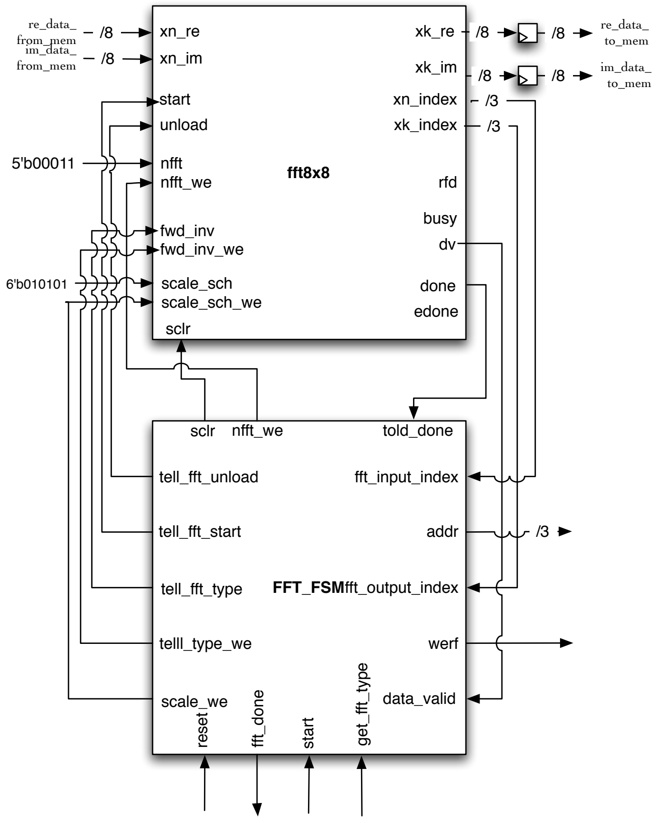

Figure 8: The module diagram for the Fourier Wrapper module.

assured that happy about this length. Also since we have only 8 bits of resolution on the output signal a huge number of coefficients seems unreasonably expensive for the precision of the data output. After considering our options, and realizing that should the full project be implemented and the multiple sources be maximized we would have between 10 and $2 \cdot n$ FFT modules in logic. It is certain that we cannot fit a huge number of such complex modules on the board. Therefore we contented ourselves with an 8 sample transform that maps from 8-bit signed coefficients to 8-bit signed coefficients. We also chose to use the default setups for the Raidix-2 Butterfly version of the FFT coregen because those met our needs sufficiently well that a change would only have complicated matters and not given us any additional data.

ILD Triport Memory Sub-Module Initially this module was not in my design. Rather I was passing all of the real and imaginary audio components back and forth through

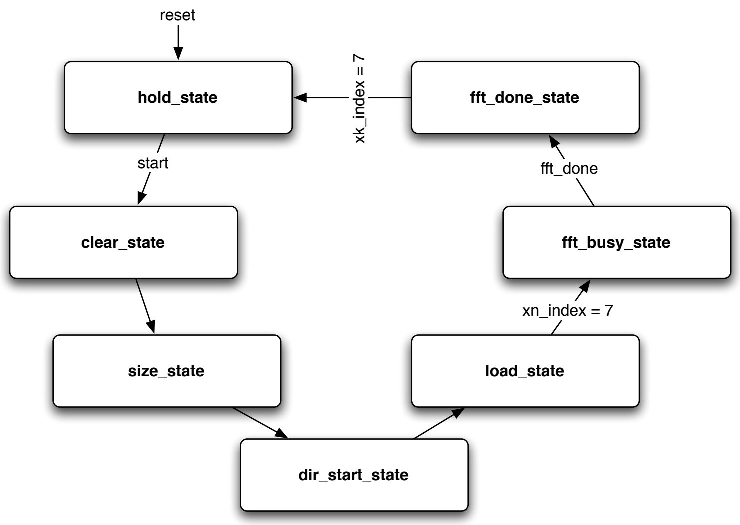

Figure 9: The finite state machine for the Fourier Wrapper module.

Owner Value	Module Accessing Memory
2'b00	ILD FSM
2'b01(invalid assignment)	ILD Muter
2'b10	Fourier Wrapper
2'b11	ILD Muter

accessory states in the ILD's FSM. During implementing this I realized that this was a terrible idea as it ensured that there were several extra clock cycles of delay and much increased complexity as a result of the design. The obvious way to remove the need for message passing is to multiplex the input to the shared memory and also the memory output to each of the submodules that need to access the memory based on some ownership input signal. This led to the creation of the Memory Controller module.

Essentially, it is just a piece of combinational logic that wraps an 8-bit wide by 8 address deep BRAM. The address, write enable, and data in inputs to the BRAM is dependent on the value of the input $owner[2:0]$. Then the outputs of the module, namely: fsm_dout, fft_dout, and mute_dout are assigned the value of the BRAM's output if and only if the input value of owner is is equal to the parameter governing who has write control to the module, otherwise the output port is tied to 8'b0.

Error Averted by Argument While this module does achieve what I hoped for that there is far less data passing through the FSM and that complexity has decreased sharply, due to one off errors because of my registering of data inputs and outputs as they pass to the module I still had to add states to the FSM. These could have been avoidable had I chosen to use continuous assignment, but I am far more comfortable knowing on which clock cycle I can expect the data downstream. This is only a minor hiccup however and doesn't impact the project on a large scale.

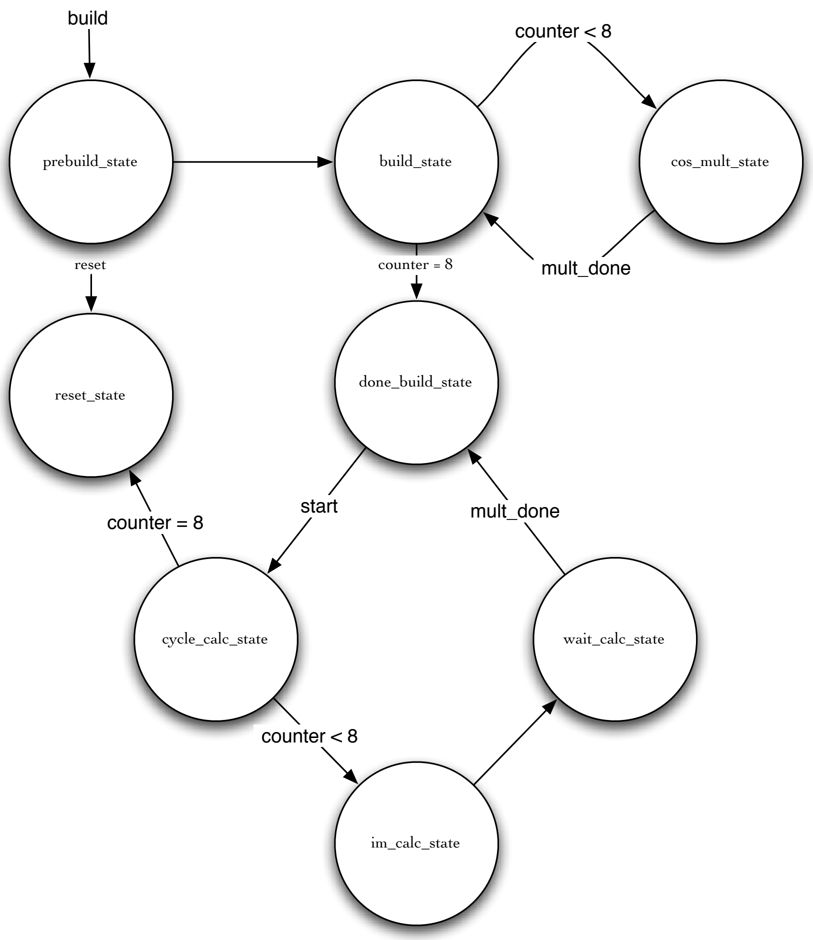

Figure 10: The finite state machine for the ILD Muter module.

ILD Muter Sub-Module This module performs the mute described in Section 2.1.2. It has a few general states of operation and is controlled by an FSM shown in Figure 3.4.2. The states can be characterized by:

1. It is holding, meaning that it has not yet bet told to build a table or mute all of the pieces of data in memory.
2. It is in the process of building a table stored in a module internal BRAM memory, this is based on the build input signal going high.
3. It has finished building the cosine table and is waiting to modify the data pieces in memory.
4. It is modifying each piece of data in memory and then writing it back to the same place in memory based on the start signal going high.
5. It is again holding because it completed all of the steps to squelch or boost the higher frequencies.

Error Detected As was previously stated, there is a bug, or several existing in my project in the ILD module that I had not been able to track down. As I was preparing the FSM diagram, Figure 3.4.2, I realized that I was getting stuck in the cycle_calc_state because my state variable was never assigned a new value. This at least partially explains the reason that the ILD module was not outputting any data other than 0. The ILD FSM ties all of the outputs to 8'b0 until it reaches the output_state which it never will because the ILD FSM will not advance past the mute state until the ILD Muter enables its done output for a clock cycle.[1]

Minimizing Logic You notice that we only are worried about the imaginary coefficients from the Fourier transform in this module. This is due to the fact that in 2.1.2 it was shown that the real coefficients are changed only minimally while the imaginary ones can change by any real factor in the range $0 \leq x \leq 2$. This minimizes the amount of logic required to modify both components concurrently since we can essentially ignore the real ones.

Future Optimizations The downside of this build table in one state during a single sample request state when we have not stored the Θ values for each sample independently is that I am only sampling the Θ input once every 8 sample request cycles. While this is potentially a problem for very quickly moving signals that circumnavigate you in eight clock cycles or less, in reality very few samples we are interested in can travel that fast.

[1]This error has been marked in the code with a comment and been subsequently corrected though there has been no further testing of the module since the project checkoff.

Further, for slower moving signals that only move a few degrees a sample request cycle are also fine because humans are really only able to resolve to a range of 10 to 15 degrees for any given Θ under suboptimal(i.e. not a quiet room) conditions. Although, under optimal conditions and a fine point source humans are able to resolve within 1 degree. Thus, there is some valid range of Θ_{approx} for any $\Theta_{absolute}$ input to the system. Further, if the signal is not moving as in the surround sound implementation we can be assured that the Θ is always exactly correct.

3.4.3 ITD Module

The module diagram for the ITD module can be seen in Figure 3.4.3. As you can see that there are several modules that are controlled by a FSM that holds in a state until it gets what amounts to a done signal from the module.

Description of Components The modules basically break down into a multiplier to calculate the amount of time to delay a signal for a specific θ value, a piece of BRAM memory for storing the audio data that is 12-bits wide and 24 positions deep, the maximum number of sample requests you can delay an audio sample, and a 5-bit wide 24 position deep BRAM storing the number of times each position in the audio memory has been accessed.

The reason the audio memory is 12 positions wide is tied to the reason we need an access counter memory. Essentially, if we delay a sample to a position in memory, time, that already has a sample arriving at it we sum their PCM values together. This can generate data overflow if we restricted it to 8-bits. Therefore, taken to the theoretical adversarial maximum, there could be 23 such additions that occur. This generates a piece of data that is 12-bits wide. We prefer to store these extra bits because it minimizes rounding errors and is essentially a cheap to hold the extra data until the last step. If we had a larger “sweet spot” that the head of the user we were trying to model this would not be the case and rounding on every sum would be necessary. Thus the access memory stores how many times a location in memory has been accessed so that we can multiplex this value to select the most significant bits for the number of additions that have occurred.

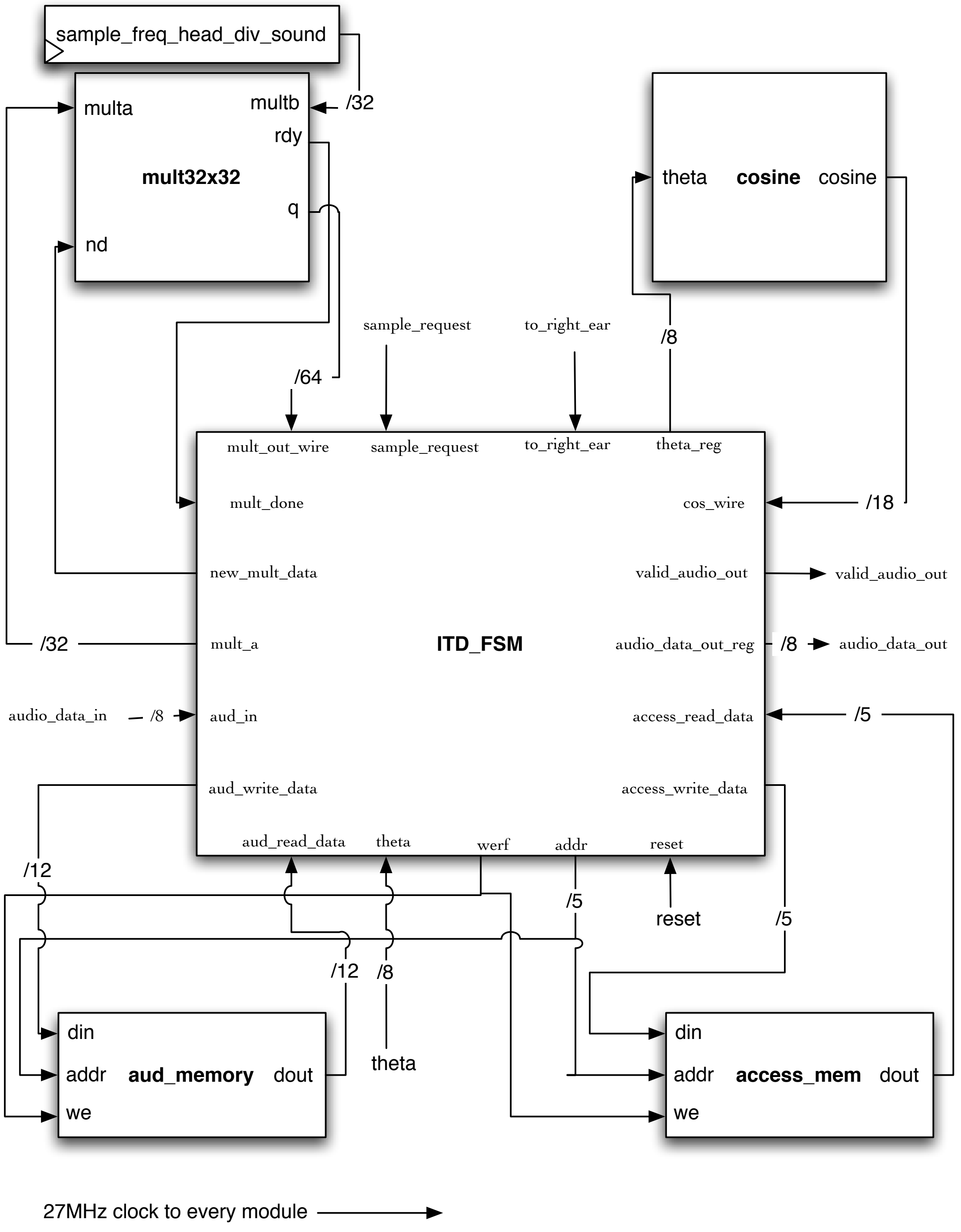

Figure 11: The block diagram for the ITD Module.

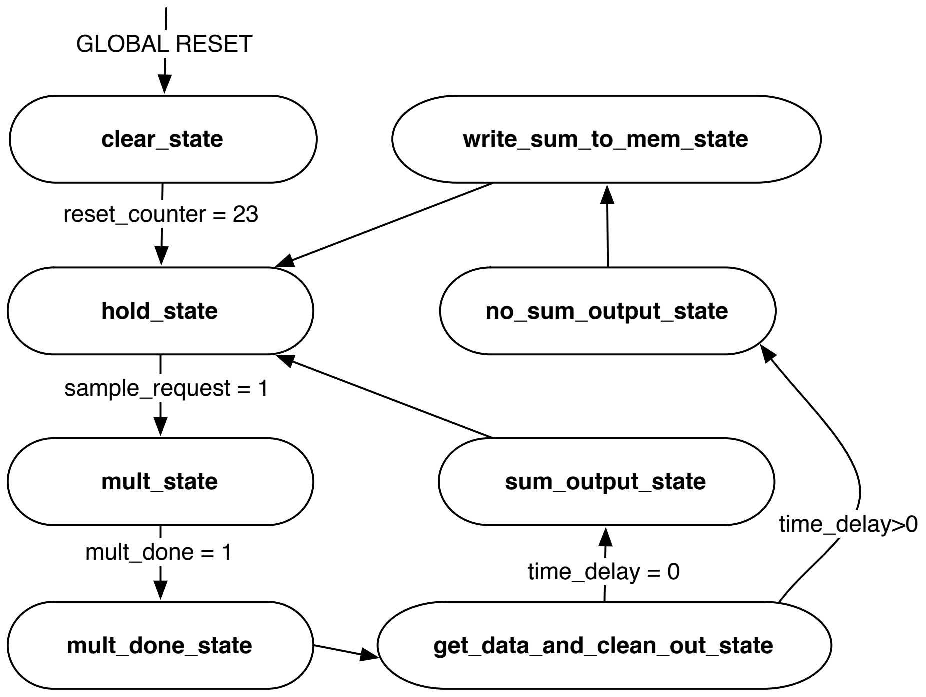

Figure 12: The finite state machine for the ITD module.

FSM Outline A streamlined version of what happens in the system after a reset is the system holds until sample request goes high and the module takes in an audio signal and the cosine presented on its input ports. This allows for the system to vary theta for a signal source, allowing the point source to move around the user. The cosine of the Θ is taken on one clock cycle. The Coregen cosine module was used and it outputs data in signed form where the two most significant bits are the sign value and the integer 1 respectively. The other 16 bits are floating point data by powers of two. This means that we maintain this data format throughout the rest of the calculations.

On the next clock cycle, a multiply is started that takes this calculated $\cos(\Theta + \phi)$ value and calculates $1 - \cos(\Theta + \phi)$ in the format described above and sign extends it to fill up the 32-bit input to the multiplier and a constant value for $\frac{a}{c} \cdot Frequency_{samplerequest}$. For this constant we take a to be 8 centimeters, c to be 34300 centimeters per second and $Frequency_{samplerequest}$ to be $48kHz$ resulting in a known parameter of 11.1953353.

Once the multiplier says that it is finished with the computation we take the integer component of the output. This is the number of audio request signals to wait in order for the audio to be delayed to the correct position in time. There are essentially two cases for what can happen here. If the calculated time delay is 0 then we should read a value for the time position memory and sum and shift it on the fly based on the number of accesses at the spot in memory plus one because we are adding data to it this cycle. The other case is that we are delaying until some time in the future to play the sound. In this case we sum the output of the audio memory at the future delayed position and increment the number of accesses at this time by one.

We also must clear the present time memory location and number of accesses so that we don't sum a signal that has already been played with one that is waiting to be played.

In retrospect, it would have been more efficient in both time and space to use a memory to store all of the cosine values in a table. It would have taken just one pass of the 256 values and written them to a ROM then we could access them by using the input theta as the index to the memory. This would be a very simple modification to the module, but it would not allow for build-time configurable parameters of the head size as is currently able. Although, admittedly, the head parameter can not vary a huge amount because the size of the memory was not configured as build-time configurable so we would get overflow if it became too large.

3.4.4 Room Echo Module

The Room Echo module represents a very simplistic model of a room. In reality is is unlike any room the user has happened to be in. The room is essentially modeled as an insulated right circular cylinder and the user is floating in the center with his or her ears on a single plane that is parallel to the top and bottom of the room and in the same plane as the audio source.

As the name of the module would suggest we are interested in the reflections of the audio source off of the wall of the room and how they are delayed in time and lessened in magnitude. The average sized room has an audio delay between 10 and 30 milliseconds and the magnitude of such waves are typically decreased by some significant amount. We arbitrarily chose to decrease the signal by 75%. Further audio testing needs to be done to validate that this is an reasonable amount to lessen the magnitude by such that it provides adequate output, but the value is set by a parameter that shifts the bits by some amount so it should be fairly simple to do.

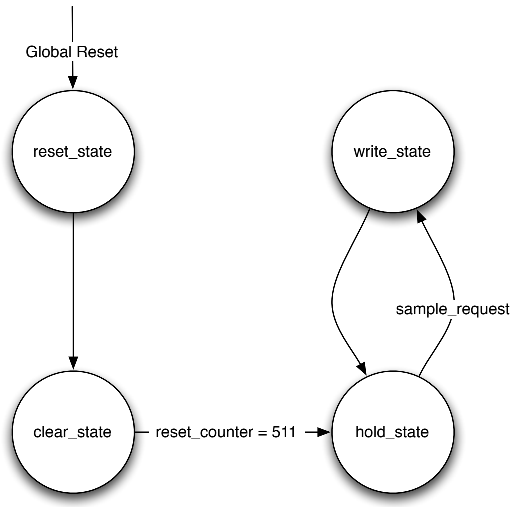

Figure 13: FSM diagram for the Room Echo Module

The delay is handled by an FSM that reads and writes from an 8-bit wide, 512 location deep BRAM memory. Essentially it just iterates through each location in memory and reads the output on every sample request pulse and puts it on its output. It also sets up to write the new incoming data on the sample request cycle to the position in memory that it just read from. For our memory with 512 locations in it this amounts to a delay of just over 10 milliseconds. This is perfect for small room echoes. To see the FSM that administers it this module you can reference Figure 13.

3.4.5 Two Source Sum

This module functions as the plus sign in Figure 2.1.4. It just a piece of combinational logic that sums two 8-bit signed audio signals together and right shifts the sum by 1 to ensure that the output is still 8-bits long and assigns this to its output. These can be cascaded to provide the necessary sums for the Surround Sound module pictured in Figure 2.1.4 where the cascade of these modules would be represented by the large + module that takes in 5 signals and outputs one.

3.4.6 Mono Source Module

The Mono Source Module is just an implementation of the diagram in Figure 2.1.4. We neglect to assign wires to the output ports of valid audio for each of the submodules described. We found them to be unnecessary because each of the submodules takes significantly less than the theoretical number of clock cycles before the next sample request to produce valid output on its audio_data_out port.

Possible Optimizations I considered initially sharing the resources between the modules such as the Cosine look up table from the ILD and ITD modules but in the end I decided that I wanted the system to have as much flexibility as humanly possible. It is reasonable to think that I might want to try different ways to modify cosine for each ear or perform simultaneous lookups of different addresses. Though this is not currently supported if I changed a the ITD, ILD or Room Echo modules it easily could become part of the specification. So I gave up a little bit on logic size to optimize for flexibility which I believe is an acceptable tradeoff in this instance.

Problems Audio is still produced from this module even though the ILD module, at presentation time, was not functional. Though I believe I have identified the source of the problem and fixed it now, at this time there has been no further build session to confirm my suspicion that it is functioning correctly. This phantom audio is due to the Room Echo module shifting and delaying its data then this data being shifted again as it is summed with 8'b0 from the output of the ITDs. Essentially al of my problems from this module on to the more complex ones that utilize this module originate in the ILD module.

3.4.7 Surround Sound Module

Though this module was never implemented because it depended on Mono Source to function correctly, the Module Diagram can be seen in Figure 2.1.4. If Mono Source begins working then this module should work as specified since there are essentially no logic that would not have already been verified as functioning correctly, namely the Mono Source and Two Source Sum modules.

This design is also very very expandable as you wish to add more and more channels. There is a theoretical maximum as you approach 20 channels of audio due to the size requirements of the modules as implemented and speciified by the Coregen documentation. You must take this with a grain of salt, however, as there are no audio systems, that this team could find, in the entire world that store that many streams of audio or spacial localization playback.

The most challenging component of this module that has not been worked out yet is where 5 distinct audio sources are going to come from. Ideally they would be streamed in in real time digitally separately or generated using a Dolby®decoder module that accepts input in the AC-3 format and generates each channel's PCM streams. The most logical solution, though would to load a ROM with PCM data generated in MatLab from some MPEG source and then read out the appropriate 5 samples every sample request cycle. Although this is unimplemented loading from a ROM it is a relatively easy procedure and can be implemented fairly quickly.

4 Testing and Debugging

The final project, though fun and exciting because I picked a project that I was truly interested, quickly became very arduous. I believe that this was largely due to my lack of a quick, visual debugging system that could integrate easily with the Corgens. Thus I was forced to rely on long compile and build times to get output on the logic analyzer. This made even the most simplistic of errors a very time consuming process. Though I am aware that there is a method by which you can enable the corgens to function as they are specified in ModelSim I was unable to get this to function and returned to my traditional method of debugging each module individually using the Logic Analyzer and test-benches for the FSMs.

Debugging would have been made much easier if I had established some waveforms other than the 750Hz sine wave that were known values. Having only one became tedious and painful to listen to for too long. This is why I included that I was going to build the modules in Matlab for validation because audio is so very hard to debug. I wish that I had actually done that but I got too deeply dependent on logic analyzer style of debugging that I was comfortable and familiar with to actually start writing these modules. If I were to do another audio project, or any foreseeably difficult to debug project in the future, establishing the baselines of performance for both a known and arbitrary signal would be my first priority.

4.1 Uncorrected Bugs

As it stood at the time of demonstration there are still significant bugs causing the ILD module not to output any audio other than constant 8-bit zero values. As I described in Section 3.4.2 I believe that I have identified one of the most significant factors in the error, but it might still not be the only one. If there are other errors they are most likely in the Fourier Wrapper module. While I did run several testbench waveforms on the FSM controlling it I never compared in real time if this was synchronous with the FFT. It is possible, though unlikely, that I made a timing error which will result in all of my data to by off to some degree.

5 Conclusions

There are many things that I would have done differently. I should have listened to my roots as a Computer Science student and written the tests before I wrote the code. This would have saved me so much time in the debugging stage when I attempted to remember what a module that I wrote a week earlier should be doing at an arbitrary state. Even if it didn't help me to debug it, it would have provided the rudimental framework for the outline of my project. Also there is a piece of bijection software that maps MatLab code onto Verilog in a $1 - to - 1$ fashion. While the code is unreadable, it gives a piece of working hardware for debugging modules that are dependent on it until your code is actually working.

I also realized that the reason that when the commercial industry as a whole chooses not to pursue a method of doing something it is for one of two reasons:

1. You can get similar or better results easier, faster, cheaper, etc. using another method.

2. The proposed method has not been shown to work in the application that you are attempting.

The problem is that you generally don't know the case until you are done with the project and see the results. Trying things in an unconventional way certainly leads to innovation and creativity, but at the risk of it not functioning as well as current models. In short, I am pleased that I attempted to be unconventional and implement it in a non-traditional way, yet if I had to do it again I feel that convolution would have been a far better match to what I was trying to do with both less logic and greater accuracy. I realized that the setup time to get the best results for a user could essentially be designed as binary search and so the user would only have to try $O(\log n)$ samples before finding the one they like the most and there is more than enough space on the ROM, ZBT, or even BRAM to store the HRIR models for a multiplicity of users. I did eventually find a website that has a large and open source HRTF database[2]. Using MatLab and some provided scripts it would be easy to convert to HRIR and then store in a COE file for use with a memory.

In the end I did enjoy my project, though I wish I had started with a slightly less daunting project than Dolby Digital decoding and virtual surround sound. Starting with such a difficult project made me focus all of my energies on the Dolby decoding and had the virtual surround sound as an afterthought until the project presentation. This made me distinctly change the focus of my project midstream resulting in a significant loss of time and effort. This played a large part in me not completing the project and

[2]I have extensively used the CIPIC Webpage for this project and report, they can be found on the web at the following URL:
http://interface.cipic.ucdavis.edu/

I wish my project had been a little more flashy though. While functionality of what you set out to do is the ultimate goal, if you have a beautiful interface or cool output that is easy to interpret, it keeps you motivated to get it fully working. My project lacked this in the buildup stages but it would have been present had the ILD module functioned correctly. In the future, for prototypes and demonstrations I will try to extract from every project some innovative display or interface that allows the wow factor for all stages of the project.

A Appendix

All of the code that I wrote or modified for the project is included in this appendix.

A.1 Labkit.v

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
///////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2006-Mar-08: Corrected default assignments to "vga_out_red", "vga_out_green"
//              and "vga_out_blue". (Was 10'h0, now 8'h0.)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
```

```
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
///////////////////////////////////////////////////////////////////////////////

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

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

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

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

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

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

       clock_feedback_out, clock_feedback_in,

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

       rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

       mouse_clock, mouse_data, keyboard_clock, keyboard_data,

       clock_27mhz, clock1, clock2,

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

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

       switch,

       led,

       user1, user2, user3, user4,
       daughtercard,

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

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

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

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

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

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

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

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

   input  clock_feedback_in;
   output clock_feedback_out;

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

   output rs232_txd, rs232_rts;
   input  rs232_rxd, rs232_cts;

   input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

   input  clock_27mhz, clock1, clock2;

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

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

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

  inout [43:0] daughtercard;

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

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

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

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

  // VGA Output
  assign vga_out_red = 8'h0;
  assign vga_out_green = 8'h0;
  assign vga_out_blue = 8'h0;
  assign vga_out_sync_b = 1'b1;
  assign vga_out_blank_b = 1'b1;
  assign vga_out_pixel_clock = 1'b0;
  assign vga_out_hsync = 1'b0;
  assign vga_out_vsync = 1'b0;

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

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

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

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

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

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

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

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

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

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

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

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

// Allow user to adjust volume
   wire vup,vdown;
   reg old_vup,old_vdown;
   debounce bup(reset, clock_27mhz, ~button_up, vup);
   debounce bdown(reset, clock_27mhz, ~button_down, vdown);
   reg [4:0] volume;
   always @ (posedge clock_27mhz) begin
     if (reset) volume <= 5'd8;
     else begin
if (vup & ~old_vup & volume != 5'd31) volume <= volume+1;
if (vdown & ~old_vdown & volume != 5'd0) volume <= volume-1;
     end
     old_vup <= vup;
     old_vdown <= vdown;
   end
//END Volume Control

wire ready;
wire signed [7:0] from_ac97_data, to_ac97_data_left, to_ac97_data_right;

   // AC97 driver
   AC97_Interface ac97(clock_27mhz, reset, volume,
from_ac97_data, to_ac97_data_left, to_ac97_data_right, ready,
      audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch, ac97_bit_clock);

//*********TEST*****************
wire signed [19:0] tone;
wire signed [7:0] left, right;
reg [7:0] azm;
```

```
wire bpleft, bpright;
debounce bleft(reset, clock_27mhz, ~button_left, bpleft);
   debounce bright(reset, clock_27mhz, ~button_right, bpright);
reg old_bpleft, old_bpright;
always @ (posedge clock_27mhz) begin
if (reset) azm <= 8'b0;
  else begin
if (bpleft & ~old_bpleft) azm <= azm-1;
if (bpright & ~old_bpright) azm <= azm+1;
      end
old_bpleft <= bpleft;
  old_bpright <= bpright;
   end

wire bpenter;
   debounce breset_debounce(reset, clock_27mhz, ~button_enter, bpenter);
   reg old_bpenter;
reg res;
   always @ (posedge clock_27mhz) begin
    if (reset) res <= 0;
      else begin
if (bpenter & ~old_bpenter) res <= 1;
else res <= 0;
      end
      old_bpenter <= bpenter;
   end

/*
wire [17:0] cos_wire;
cosine test_cos(.THETA(switch[7:0]), .CLK(clock_27mhz), .COSINE(cos_wire));
wire [18:0] sub_wire;
assign sub_wire = (19'b001___0000_0000_0000_0000-{cos_wire[17], cos_wire});
assign led = ~(sub_wire[18:11]);
*/

tone750hz tonemaker(clock_27mhz,  ready, tone);
wire [2:0] debug_state;
wire [4:0] d_delay;
wire [4:0] combo_wire;
wire [7:0] bram_write;
ITD_module left_ear_test(.clk(clock_27mhz), .sample_request(ready),
.audio_data_in(tone[19:12]), .theta(azm),
.to_right_ear(1'b0), .audio_data_out(left), .reset(res));
ITD_module right_ear_test(.clk(clock_27mhz), .sample_request(ready),
.audio_data_in(tone[19:12]), .theta(azm),
 .to_right_ear(1'b1), .audio_data_out(right), .reset(res));
//ILD_module left_ear_test_ild();
//ILD_module right_ear_test_ild();
assign to_ac97_data_left = left;
assign to_ac97_data_right = right;

assign led = ~azm;//{debug_state, volume};
assign analyzer1_data = {tone[19:12], combo_wire, debug_state};
assign analyzer2_data = {left, right};
assign analyzer3_data = {bram_write, res, 2'b0, d_delay};
assign analyzer1_clock = ready;
assign analyzer2_clock = clock_27mhz;
endmodule
```

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

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

   reg [8:0] index;
   reg [19:0] pcm_data;

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

   always @(posedge clock) begin
      if (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
```

A.2 AC97_Commands.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:  Harrison King Hall
// Engineer: Digital Death
//
// Create Date:    15:24:11 11/16/06
// Design Name:
// Module Name:    AC97_Commands
// Project Name:   VSS
// Target Device:
// Tool versions:
// Description: Tihs file was provided by the 6.111 staff for the Fall 2006 term of 6.111.
//
// Dependencies:
//
// Revision:
```

```
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
// issue initialization commands to AC97
module AC97_Commands(clock, ready, command_address, command_data,
                     command_valid, volume, source);

   input clock;
   input ready;
   output [7:0] command_address;
   output [15:0] command_data;
   output command_valid;
   input [4:0] volume;
   input [2:0] source;

   reg [23:0] command;
   reg command_valid;

   reg [3:0] state;

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

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

   wire [4:0] vol;
   assign vol = 31-volume;  // convert to attenuation

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

      case (state)
        4'h0: // Read ID
          begin
             command <= 24'h80_0000;
             command_valid <= 1'b1;
          end
        4'h1: // Read ID
          command <= 24'h80_0000;
        4'h3: // headphone volume
          command <= { 8'h04, 3'b000, vol, 3'b000, vol };
        4'h5: // PCM volume
          command <= 24'h18_0808;
        4'h6: // Record source select
          command <= { 8'h1A, 5'b00000, source, 5'b00000, source};
        4'h7: // Record gain = max
  command <= 24'h1C_0F0F;
        4'h9: // set +20db mic gain
          command <= 24'h0E_8048;
        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
```

A.3 AC97_Framemaker.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:21:01 11/16/06
// Design Name:
// Module Name:    AC97_Framemaker
// Project Name:
// Target Device:
// Tool versions:
// Description: This was provided for Lab 4 in the 2006 Fall term of 6.111.
//It produces the data frames that pass to the AC97 module on the labkit.
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

// assemble/disassemble AC97 serial frames
module AC97_Framemaker(ready,
             command_address, command_data, command_valid,
             left_data, left_valid,
             right_data, right_valid,
             left_in_data, right_in_data,
             ac97_sdata_out, ac97_sdata_in, ac97_synch, ac97_bit_clock);

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

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

   reg ready;

   reg ac97_sdata_out;
   reg ac97_synch;
   reg [7:0] bit_count;

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

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

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

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

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

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

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

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

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

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

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

      bit_count <= bit_count+1;

   end // always @ (posedge ac97_bit_clock)

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

endmodule
```

A.4 AC97_Interface.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:  Digital Death
// Engineer: Harrison King Hall
//
// Create Date:    15:15:13 11/16/06
// Design Name:
// Module Name:    AC97_Interface
// Project Name:   VSS
// Target Device:  6.111 Labkit
// Tool versions:
// Description: THis module handles all of the AC97 processing for the Labkit such that it
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module AC97_Interface(clock_27mhz, reset, volume,
             audio_in_data, audio_out_data_left, audio_out_data_right, ready,
          audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch, ac97_bit_clock);

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

   input clock_27mhz;
   input reset;
   input [4:0] volume;
   output [7:0] audio_in_data;
   input [7:0] audio_out_data_left;
input [7:0] audio_out_data_right;
   output ready;

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

   wire [2:0] source;
   assign source = 0;    //mic

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

   reg audio_reset_b;
   reg [9:0] reset_count;

   //wait a little before enabling the AC97 codec
   always @(posedge clock_27mhz) begin
      if (reset) begin
         audio_reset_b = 1'b0;
         reset_count = 0;
      end else if (reset_count == 1023)
        audio_reset_b = 1'b1;
      else
        reset_count = reset_count+1;
   end
   wire ac97_ready;
   AC97_Framemaker framemaker(ac97_ready, command_address,
    command_data, command_valid,
         left_out_data, 1'b1, right_out_data, 1'b1, left_in_data,
         right_in_data, ac97_sdata_out, ac97_sdata_in, ac97_synch,
         ac97_bit_clock);

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

   reg [19:0] out_data_left;
reg [19:0] out_data_right;

   always @ (posedge clock_27mhz)
     if (ready) begin
   out_data_left <= audio_out_data_left;
out_data_right <= audio_out_data_right;
  end

   assign audio_in_data = left_in_data[19:12];
   assign left_out_data = {out_data_left, 12'b000000000000};
   assign right_out_data = {out_data_right, 12'b000000000000};

   // generate repeating sequence of read/writes to AC97 registers
   AC97_Commands commands(clock_27mhz, ready, command_address,
    command_data, command_valid, volume, source);
endmodule
```

A.5 FFT_FSM.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Harrison King Hall hkhall@mit.edu
//
// 6.111 Final Project: Virtual Surround Sound
//
// Module Name:     FFT_FSM
//  Date Modified: 18:52 12.09.2006
//
// This is the FSM that controls the FFT module.
//
//  Revision:
//  Revision 0.01 - File Created
//  Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module FFT_FSM(clk, start, reset, fft_done, data_valid, fft_input_index,
fft_output_index, told_done, tell_fft_start, tell_fft_unload, get_fft_type,
tell_fft_type, tell_type_we, tell_fft_sclr, addr, werf, nfft_we, scale_we);

input clk;
input start;
input reset;
output fft_done;
```

```
input data_valid;
input [2:0] fft_input_index;
input [2:0] fft_output_index;
input told_done;
output tell_fft_start;
output tell_fft_unload;
output tell_fft_sclr;
output [2:0] addr;
input get_fft_type;
output tell_fft_type;
output tell_type_we;
output werf;
output nfft_we;
output scale_we;

//registering outputs
reg fft_done;
reg nfft_we;
reg scale_we;
reg [2:0] addr;
reg tell_fft_start, tell_fft_unload;
reg tell_fft_sclr;
reg tell_fft_type;
reg tell_type_we;
reg werf;

//state parameters
parameter hold_state = 0;
parameter clear_state = 1;
parameter dir_start_state = 2;
parameter load_state = 3;
parameter fft_busy_state = 4;
parameter fft_done_state = 5;
parameter declare_done_state = 6;
parameter size_state = 7;

parameter transform_size = 5'b00011;

reg [2:0] state;
reg [3:0] counter;
reg [2:0] d1, d2;

always @ (posedge clk) begin
if(reset) state <= hold_state; //global reset signal

case(state)
hold_state: begin
counter <= 0;
fft_done<= 0;
tell_fft_start <= 0;
tell_fft_unload <= 0;
tell_fft_sclr <= 0;
addr <= 0;
d1 <= 0;
d2 <= 0;
tell_fft_type <= 1;
tell_type_we <= 0;
werf <= 0;
```

```
if(start) state <= clear_state; //telling the fft that we are starting
end

clear_state: begin
tell_fft_sclr <= 1; //clear the fft
state <= size_state;
//since we are not using a runtime configurable size and default scaling we can skip these steps
end
size_state: begin
tell_fft_sclr <= 0;
nfft_we <= 1;
state<= dir_start_state;
end

dir_start_state: begin
nfft_we <= 0;
tell_fft_type <= get_fft_type;
tell_type_we <= 1;
tell_fft_start <= 1;
scale_we <= 1;
state <= load_state;
end

load_state: begin
scale_we <= 0;
tell_type_we <= 0;
tell_fft_start <= 0;
d1 <= fft_input_index;
d2 <= d1;
addr <= d2;
if(addr ==7) state <= fft_busy_state;
end

fft_busy_state: begin
if(told_done) begin//if fft is ready ot give us data
tell_fft_unload <= 1;
state <= fft_done_state;
end
end

fft_done_state: begin
tell_fft_unload <= 0;
if(addr == 7) state <= declare_done_state; //if we are on the last state shift to hold_state
else
if(data_valid) begin
werf <= 1;
addr <= fft_output_index; //telling the user where to store their data
end
end

declare_done_state: begin
fft_done <= 1;
state <= hold_state;
end

default : state <= hold_state;
endcase
end //end posedge cllk
```

```
endmodule
```

A.6 Fourier_Transform_Wrapper.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Harrison King Hall hkhall@mit.edu
//
// 6.111 Final Project: Virtual Surround Sound
//
// Module Name:     Fourier_Transform_Wrapper
//  Date Modified: 18:52 12.09.2006
//
// This is the wrapper module that makes it easy to interface with the Coregen
// FFT.  It reads and writes to a memory when told to start and given the type
// of transform to do on the same clock cycle.  The type of transfer is 1 for
// FFT and 0 for IFFT
//
//  Revision:
//  Revision 0.01 - File Created
//  Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module Fourier_Transform_Wrapper(clk, reset, fft_type, start,
re_data_from_mem, im_data_from_mem,
fft_done, werf, addr, re_data_to_mem,
im_data_to_mem);
input clk;
input reset;
input fft_type;
input start;
   input signed [7:0] re_data_from_mem;
   input signed [7:0] im_data_from_mem;
output fft_done;
output werf;
output [2:0] addr;
output signed [7:0] re_data_to_mem;
output signed [7:0] im_data_to_mem;

wire [7:0] re_data_from_mem_wire, im_data_from_mem_wire;

wire  data_valid_wire, fft_says_done_wire, tell_fft_start_wire, tell_fft_start_unload,
dir_wire, dir_we_wire;

wire [2:0] input_index_wire, output_index_wire;
wire sclr_wire, scale_we_wire, fft_edone_wire;

//wire fft_rdy_wire, fft_busy_wire, fft_edone_wire;

reg signed [7:0] re_out_reg, im_out_reg;
wire signed [7:0] re_data_from_fft_wire, im_data_from_fft_wire;
//the registered output from the fft
always @ (posedge clk) begin
re_out_reg <= re_data_from_fft_wire;
im_out_reg <= im_data_from_fft_wire;
end

assign re_data_to_mem = re_out_reg;
assign im_data_to_mem = im_out_reg;

//FFT module
fft8x8 my_fft(
.clk(clk),
.xn_re(re_data_from_mem),
.xn_im(im_data_from_mem),
.start(tell_fft_start_wire),
.unload(tell_fft_start_unload),
.nfft(5'b00011),
.nfft_we(nfft_we_wire),
.fwd_inv(dir_wire),
.fwd_inv_we(dir_we_wire),
.scale_sch(6'b010101),
.scale_sch_we(scale_we_wire),
.xk_re(re_data_from_fft_wire),
.xk_im(im_data_from_fft_wire),
.xk_index(output_index_wire),
.xn_index(input_index_wire),
.rfd(fft_rdy_wire),
.busy(fft_busy_wire),
.dv(data_valid_wire),
.edone(fft_edone_wire),
.done(fft_says_done_wire),
.sclr(sclr_wire));

//FSM
FFT_FSM fsm(
.clk(clk),
.start(start),
.reset(reset),
.fft_done(fft_done),
.data_valid(data_valid_wire),
.fft_input_index(input_index_wire),
.fft_output_index(output_index_wire),
.told_done(fft_says_done_wire),
.tell_fft_start(tell_fft_start_wire),
.tell_fft_unload(tell_fft_start_unload),
.tell_fft_sclr(sclr_wire),
.addr(addr),
.get_fft_type(fft_type),
.tell_fft_type(dir_wire),
.tell_type_we(dir_we_wire),
.nfft_we(nfft_we_wire),
.scale_we(scale_we_wire),
.werf(werf));
endmodule
```

A.7 ILD_FSM.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Harrison King Hall hkhall@mit.edu
//
// 6.111 Final Project: Virtual Surround Sound
//
//  Module Name:     ILD_FSM
//  Date Modified: 18:52 12.09.2006
```

```
//
// This is the FSM that controls the ILD module.
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module ILD_FSM(clk, reset, sample_request, audio_data_in, to_right_ear,
audio_data_out, valid_audio_out, theta, owner,
fft_done, fft_start, fft_type,
re_mem_read_data, im_mem_read_data, werf, addr,
re_mem_write_data,im_mem_write_data,
start_mute, mute_done, build_mute_table, shifted_theta);

//general IO
input clk;
input reset;
input sample_request;
input [7:0] audio_data_in;
input [7:0] theta;
input to_right_ear;
output valid_audio_out;
reg valid_audio_out;
output [7:0] audio_data_out;
reg [7:0] audio_data_out;
output [1:0] owner; //2'b00 and 2'b01 specifies fsm ownership,
//2'b10 specifies fft ownership, 2'b11 specifies mute ownership
reg [1:0] owner;


//FFT IO
input fft_done; //1 when control passes back to fsm
output fft_start; //1 when control passes to the fft
reg fft_start;
output fft_type; //1 for fft, 0 for ifft
reg fft_type;

//memory IO
input [7:0] re_mem_read_data;
input [7:0] im_mem_read_data;
output werf;
reg werf;
output [2:0] addr;
reg [2:0] addr;
output [7:0] re_mem_write_data;
reg [7:0] re_mem_write_data;
output [7:0] im_mem_write_data;
reg [7:0] im_mem_write_data;


//mute IO
output [7:0] shifted_theta;
reg [7:0] shifted_theta;
input mute_done;
output build_mute_table;
reg build_mute_table;
output start_mute;
```

```
reg start_mute;

//state parameters
parameter reset_state  = 0;
parameter hold_state  = 1;
parameter build_table_state = 2;
parameter fft_state =  3;
parameter mute_state  =  4;
parameter ifft_state = 5;
parameter output_state =  6;
parameter new_sample_state  =  7;
parameter wait_state_1 = 8;
parameter wait_state_2 = 9;
parameter wait_state_3 = 10;
parameter wait_state_4 = 11;

//owner parameters
parameter fsm_owner = 2'b00;
parameter fft_owner = 2'b10;
parameter mute_owner = 2'b11;

//transfer types
parameter fft_direction = 1;
parameter ifft_direction = 0;
//the state register
reg [3:0] state;

//registers
reg [3:0] counter;

//Sequential Always block
always @ (posedge clk) begin
if(reset)
state <= reset_state;
else
case(state)
reset_state: begin
build_mute_table <= 0;
start_mute <= 0;
shifted_theta <= 0;
owner <= fsm_owner;
fft_start <= 0;
fft_type <= fft_direction;
valid_audio_out <= 0;
audio_data_out <= 0;
werf <= 0;
re_mem_write_data <= 0;
im_mem_write_data <= 0;
addr <= 0;
counter <= 0;
state <= hold_state;
end

hold_state: begin
//if we have a new sample
if(sample_request)begin
werf <= 1;
addr <= counter[2:0]; //get the last address of the counter
```

```
re_mem_write_data <= audio_data_in; //writing the time domain data to real memory
im_mem_write_data <= 8'b0; //no imaginary data to write
counter <= counter + 1; //inc counter
end
else  begin
werf <= 0;
//next state
if(counter == 8) state <= build_table_state;
end
end

build_table_state: begin
counter <= 0; //resetting counter
if(to_right_ear) //setting it up for an ear
shifted_theta <= theta + 192;
else
shifted_theta <= theta + 64;

build_mute_table <= 1; //building the mute table
state <= wait_state_1;
owner <= fft_owner;
end

wait_state_1: begin //so the ownership can pass successfully
build_mute_table <= 0;
if(counter == 4) begin
fft_start <= 1;
state <= fft_state;
end
else counter <= counter + 1;
end

fft_state: begin
counter <= 0; //resetting counter
fft_start <= 0;
//next state
if(fft_done) begin
state <= wait_state_2;
owner <= mute_owner;
end
end

wait_state_2: begin //so the ownership can pass successfully
if(counter == 4) begin
state <= mute_state;
start_mute <= 1;
end
else counter <= counter + 1;
end

mute_state: begin //mutes all of the audio samples
start_mute <= 0; //not necessary to mute again
counter <= 0;
if(mute_done) begin
owner <= fft_owner;
state <= wait_state_3;
fft_type <= ifft_direction;
end
```

```
end

wait_state_3: begin //so the ownership can pass successfully
if(counter == 4) begin
fft_start <= 1;
state <= ifft_state;
end
else counter <= counter + 1;
end

ifft_state: begin
counter <= 0;
fft_start <= 0;
//next state
if(fft_done) begin
owner <= fsm_owner;
state <= wait_state_4;
end
end

wait_state_4: begin //so the ownership can pass successfully
if(counter == 4) begin
state <= output_state;
addr <= 0;
werf <= 0;
end
else counter <= counter + 1;
end

output_state: begin
//addr was set to one in build_table_state and hasn't changed
audio_data_out <= re_mem_read_data;
valid_audio_out <= 1;
if(sample_request) begin
state <= new_sample_state;
re_mem_write_data <= audio_data_in; //new data
im_mem_write_data <= 8'b0; //new data has no phase component
werf <= 1;
end
end

new_sample_state: begin //writing over the last place in memory
addr <= addr + 1; //resets to zero or increments, either way it does what we want
werf <= 0;
if(addr == 7)
state <= build_table_state;
else
state <= output_state;
end

default: state <= reset_state;
endcase
end//always end
endmodule
```

A.8 ILD_module.v

```
`timescale 1ns / 1ps
```

```
////////////////////////////////////////////////////////////////////////////////
// Harrison King Hall hkhall@mit.edu
//
// 6.111 Final Project: Virtual Surround Sound
//
// Module Name:     ILD_module
//  Date Modified: 18:52 12.09.2006
//
// This is the module that makes in phase high frequencies higher and out of
// phase frequencies muted.  This is the first in the cascade of modules that
// replicates virtual surround sound.
//
//  Revision:
//  Revision 0.01 - File Created
//  Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module ILD_module(clk, reset, sample_request, audio_data_in, theta, to_right_ear,
audio_data_out, valid_audio_out, re_mem_read_fsm);
/*
The ILD_module is a design that provides an interaural level difference for the user by dampening or boosting the
signal strength of frequencies  above 1.5kHz based upon the theta direction of the sound source from the user.  It
follows the equation(in frequency space):
  (1+ cos(theta))* s +  2*(c/a)
H(s{j*omega},theta)  =  --------------------------------------- * freq_data
s +  2*(c/a)

This is then transformed back into the time domain and the appropriate audio sample is selected.
*/
input clk; //system clock 27 Mhz
input reset; //resets the system
input sample_request; //serves as an 48kHz clock provided by the ac97
input [7:0] audio_data_in; //PCM data
input [7:0] theta; //ranges from 0 to 255 are acceptable
 /*   ^ 0(0 degrees)
    |
  |
-o^o-
(270 degrees)191<------( user)------>63(90 degrees)
\___/
  |
  |
  v
127(180 degrees)


 */
input to_right_ear; //1 if signal is going to a right ear/speaker , 0 if it is going to a left ear/speaker
output [7:0] audio_data_out;  //time-delayed audio out
output valid_audio_out; //1 iff output is valid since there is a time delay

//degugging
output [7:0] re_mem_read_fsm;

wire [1:0] owner_wire;
wire fft_done_wire, fft_start_wire, transform_type_wire, fsm_werf_wire, start_mute_wire, mute_done_wire, build_cos_lut_wire
wire [2:0] fsm_addr_wire, fft_addr_wire;
wire fft_werf_wire;
wire signed [7:0] fsm_read_re_mem_wire, fft_write_im_wire, fsm_read_im_mem_wire, fsm_write_re_mem_wire, fsm_write_im_mem_wi
wire signed [7:0] fft_read_re_mem_wire, fft_write_re_wire, fft_read_im_mem_wire, fft_write_re_mem_wire, fft_write_im_mem_wi

//DEBUGGING
assign re_mem_read_fsm = fsm_read_re_mem_wire;

wire [7:0] shifted_theta_wire, fft_read_re_wire, fft_read_im_wire, muter_read_data, muter_write_data;
wire [2:0] mute_addr_wire;
wire mute_werf_wire;

//FFT Wrapper
Fourier_Transform_Wrapper my_fft(    //computes the fft or ifft depending on the step
.clk(clk),
.reset(reset),
.fft_type(transform_type_wire),
.start(fft_start_wire),
.re_data_from_mem(fft_read_re_wire),
.im_data_from_mem(fft_read_im_wire),
.fft_done(fft_done_wire),
.werf(fft_werf_wire),
.addr(fft_addr_wire),
.re_data_to_mem(fft_write_re_wire),
.im_data_to_mem(fft_write_im_wire)
);
//mute block
ILD_Muter muter(
.clk(clk),
.reset(reset),
.build(build_cos_lut_wire),
.start(start_mute_wire),
.theta_in(shifted_theta_wire),
.done(mute_done_wire),
.addr(mute_addr_wire),
.werf(mute_werf_wire),
.im_write_data(muter_write_data),
.im_read_data(muter_read_data)
);

//memory
triport_memory_controller im_mem(
.clk(clk),
.fsm_werf(fsm_werf_wire),
.fsm_din(fsm_write_im_mem_wire),
.fsm_addr(fsm_addr_wire),
.fsm_dout(fsm_read_im_mem_wire),
.mute_werf(mute_werf_wire),
.mute_din(muter_write_data),
.mute_addr(mute_addr_wire),
.mute_dout(muter_read_data),
.fft_werf(fft_werf_wire),
.fft_din(fft_write_im_wire),
.fft_addr(fft_addr_wire),
.fft_dout(fft_read_im_wire),
.owner(owner_wire)
);

triport_memory_controller real_mem(
.clk(clk),
.fsm_werf(fsm_werf_wire),
.fsm_din(fsm_write_re_mem_wire),
.fsm_addr(fsm_addr_wire),
.fsm_dout(fsm_read_re_mem_wire),
//thest lines are not necessary because the muts does not access these ports
// .mute_werf(),
// .mute_din(),
// .mute_addr(),
// .mute_dout(),
.fft_werf(fft_werf_wire),
.fft_din(fft_write_re_wire),
.fft_addr(fft_addr_wire),
.fft_dout(fft_read_re_wire),
.owner(owner_wire)
);

//FSM
ILD_FSM fsm(
.clk(clk),
.reset(reset),
.sample_request(sample_request),
.audio_data_in(audio_data_in),
.to_right_ear(to_right_ear),
.audio_data_out(audio_data_out),
.valid_audio_out(valid_audio_out),
.theta(theta),
.owner(owner_wire),
.fft_done(fft_done_wire),
.fft_start(fft_start_wire),
.fft_type(transform_type_wire),
.re_mem_read_data(fsm_read_re_mem_wire),
.im_mem_read_data(fsm_read_im_mem_wire),
.werf(fsm_werf_wire),
.addr(fsm_addr_wire),
.re_mem_write_data(fsm_write_re_mem_wire),
.im_mem_write_data(fsm_write_im_mem_wire),
.start_mute(start_mute_wire),
.mute_done(mute_done_wire),
.build_mute_table(build_cos_lut_wire),
.shifted_theta(shifted_theta_wire)
);
endmodule
```

A.9 ILD_Muter_FSM.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Harrison King Hall hkhall@mit.edu
//
// 6.111 Final Project: Virtual Surround Sound
//
// Module Name:     ILD_Muter_FSM
// Date Modified: 18:52 12.09.2006
//
// This is the FSM module that controls the ILD_Muter.
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
////////////////////////////////////////////////////////////////////////////////
module ILD_Muter_FSM(clk, reset, data_in, request, rdy, cos_in,
multa, multb, mult_done, mult_val, mult_nd, build,
im_mem_data_write, im_werf, im_addr, im_mem_data_read,
built_data_write, built_werf, built_addr, built_mem_data_read
);

//general IO
input clk, reset, build, request;
input [7:0] data_in;
input [2:0] mute_val;
input signed [17:0] cos_in;
output [15:0] im_out;
output rdy;

//multiplier IO
input mult_done;
input [63:0] mult_val;
output [31:0] multa;
output [31:0] multb;
output mult_nd;

//memory IO
input [31:0] mem_data_read;
output [31:0] mem_data_write;
output [2:0] addr;
output werf;

//registering outputs
reg rdy, mult_nd;
reg [7:0] im_out;
reg [31:0] multa, multb, mem_data_write;
reg werf;
reg [2:0] addr;

reg [2:0] state;
rec [17:0] cos_reg;

//state parameters
parameter reset_state  = 0;
parameter prebuild_state = 1;
parameter build_state = 2;
parameter cos_mult_state  = 3;
parameter done_build_state = 4;
parameter im_calc_state = 5;
parameter cycle_calc_state = 6;
parameter write_calced_state = 7;

//frequency parameter
parameter dc_freq  = 32'b0000_0000_0000_0000___0000_0000_0000_0000;
parameter pi1_freq  = 32'b0000_0000_0000_0000___0000_0000_0000_0110;
parameter pi2_freq  = 32'b0000_0000_0000_0000___0000_0000_0000_1100;
parameter pi3_freq  = 32'b0000_0000_0000_0000___0000_0000_0001_0011;
parameter pi4_freq  = 32'b0000_0000_0000_0000___0000_0000_0001_1001;
parameter pi5_freq  = 32'b0000_0000_0000_0000___0000_0000_0010_0000;
parameter pi6_freq  = 32'b0000_0000_0000_0000___0000_0000_0010_0110;
parameter pi7_freq  = 32'b0000_0000_0000_0000___0000_0000_0010_1100;
```

```
reg  build_done;
reg [7:0] im_in;
reg [3:0] counter; //counts how many data samples we have calculated
reg [2:0] state;
always @ (posedge clk) begin
if (reset) state <= reset_state;
if (build) state <= prebuild_state;

case(state)
reset_state : begin
build_done <= 0; //not done with build
done <= 0; //not done
if(build) state <= prebuild_state; //must build data before we output it
end

prebuild_state: begin
build_done <= 0;
done <= 0; //not done
counter <= 0;
state <= build_state;
cos_reg <= cos_in;
end

build_state: begin
multb <= {{14{cos_reg[17]}}, cos_reg};
state(counter)
0: multa <= dc_freq;
1: multa <= pi1_freq;
2: multa <= pi2_freq;
3: multa <= pi3_freq;
4: multa <= pi4_freq;
5: multa <= pi5_freq;
6: multa <= pi6_freq;
7: multa <= pi7_freq;
endcase
mult_nd <= 1;
built_werf <= 0;
counter <= counter + 1;
if(counter == 8) state <= done_build_state;
else state <= cos_mult_state;
end

cos_mult_state: begin
mult_nd <= 0;
if(mult_done) begin
built_werf <= 1;
built_addr <= counter;
built_mem_data_write <= mult_val[39:8];
state <= build_state;
end
end

done_build_state: begin
built_werf <= 0;
if(start) begin //control has now been passed to the mute module
counter <= 0;
state <= cycle_calc_state;
```

```
end
end

cycle_calc_state: begin //control state that sets up the addresses so they can be read  next state
if(counter == 8) state <= reset_state;
else begin
addr <= counter;
built_addr <= counter;
werf <= 0;
state <= im_calc_state; //this is where my error was, I never switched this state so I hung here forever
end
end

im_calc_state: begin //multiplying the transformed audio data in memory by the stored precalculated constant
multa <= {im_mem_data_read, 24'b0}; //sign extending
multb <= built_mem_data_read; //precalculated constant
mult_nd <= 1;
state <= wait_calc_state;
end

wait_calc_state: begin
mult_nd <= 0;
if(mult_done) begin //when the multiply is done store the value in memory for use with the ifft
im_mem_data_write <= mult_val[57:50]; //the appropriate msbs (somewhere between [63:48])
werf <= 1; //writing the data back to the imaginary memory
state <= write_calced_state;
end
end

write_calced_state: begin //data is written this state, increment counter and disable werf
counter <= counter + 1;
werf <= 0;
state <= cycle_calc_state;
end

default: state <= reset_state;
endcase
end //end always @(posedge clk)

endmodule
```

A.10 ILD_Muter.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Harrison King Hall hkhall@mit.edu
//
// 6.111 Final Project: Virtual Surround Sound
//
// Module Name:     ILD_Muter
// Date Modified: 18:52 12.09.2006
//
// This takes a frequency-space signal and quiets it or boosts it based upon theta.
// The amount of boost or must is goverened by a simplified version of Lord Raleigh's
// solution:
// Mute Ammount = 8cm* omega* cos(theta + phi)
// -----------------------
// 2 * 34300cm/sec
```

```
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
////////////////////////////////////////////////////////////////////////////////
module ILD_Muter(clk, reset, build, start, theta_in, done, addr, werf,
im_write_data, im_read_data);

input clk; //system clock
input reset; //puts it back in start state
input [7:0] theta_in; //compensated theta for specific ear
input build; //control signal to build the cos-LUT
input start; //tells it to take control and mute all of the pieces and put it back in mem
input [7:0] im_read_data;
output done; //all data written to memory and is valid
output werf;
output [2:0] addr;
reg [2:0] addr;
wire [2:0] addr_wire;
output [7:0] im_write_data;
reg [7:0] im_write_data;
wire [7:0] im_write_data_wire;

always @ (posedge clk) begin
addr <= addr_wire;
im_write_data <= im_write_data_wire;
end

wire [31:0] im_multa_wire, im_multb_wire;
wire [63:0] im_mult_out_wire;
wire im_mult_done;
mult32x32 im_mult(
.a(im_multa_wire),
.b(im_multb_wire),
.q(im_mult_out_wire),
.nd(mult_nd_wire),
.rdy(im_mult_done),
.clk(clk));

wire [17:0] cos_wire;
cosine cosine_fn(
.THETA(theta_in),
.CLK(clk),
.COSINE(cos_wire));

wire [7:0] built_read_wire, built_write_wire;
wire [2:0] built_addr_wire;
wire built_werf_wire;
imaginary_memory im_mem(
.clk(clk),
.addr(built_addr_wire),
.we(built_werf_wire),
.din(built_write_wire),
.dout(built_read_wire));

ILD_Muter_FSM fsm(
.clk(clk),
.reset(reset),
.build(build),
.done(done),
.data_in(),
.start(start),
.cos_in(cos_wire),
.multa(im_multa_wire),
.multb(im_multb_wire),
.mult_done(im_mult_done),
.mult_val(im_mult_out_wire),
.mult_nd(mult_nd_wire),
.im_mem_data_write(im_write_data_wire),
.im_werf(werf),
.im_addr(addr_wire),
.im_mem_data_read(im_read_data),
.built_data_write(built_write_wire),
.built_werf(built_werf_wire),
.built_addr(built_addr_wire),
.built_mem_data_read(built_read_wire)
);

endmodule
```

A.11 ITD_FSM.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Harrison King Hall hkhall@mit.edu
//
// 6.111 Final Project: Virtual Surround Sound
//
// Module Name:     ITD_FSM
//  Date Modified: 18:52 12.09.2006
//
// This is the FSM that controls the ITD module.
//
//  Revision:
//  Revision 0.01 - File Created
//  Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module ITD_FSM(clk, reset, sample_request, audio_data_in, to_right_ear, audio_data_out_reg,
valid_audio_out, mult_out_wire, mult_done, new_mult_data, mult_a, theta,
cos_wire, theta_reg, aud_read_data, access_read_data, werf, addr,
aud_write_data, access_write_data);

//general IO
input clk;
input reset;
input sample_request;
input [7:0] audio_data_in;
input to_right_ear;
    output valid_audio_out;
output [7:0] audio_data_out_reg;

//mult IO
input [63:0] mult_out_wire;
input mult_done;
output new_mult_data;
output [31:0] mult_a;

//cos IO
input [7:0] theta;
input signed [17:0] cos_wire;
output [7:0] theta_reg;

//memory IO
input [11:0] aud_read_data;
input [4:0] access_read_data;
output werf;
output [4:0] addr;
output [11:0] aud_write_data;
output [4:0] access_write_data

//registering outputs
reg valid_audio_out;
reg [7:0] audio_data_out_reg;
reg new_mult_data;
reg [31:0] mult_a;
reg [7:0] theta_reg;
reg werf;
reg [4:0] addr;
reg [11:0] aud_write_data;
reg [4:0] access_write_data;

//state parameters
parameter hold_state = 0;
parameter mult_state = 1;
parameter mult_done_state = 2;
parameter get_data_and_clean_out_state = 3;
parameter sum_output_state = 4;
parameter no_sum_output_state = 5;
parameter write_sum_to_mem_state = 6;
parameter clear_state = 7;

//the state register
reg [2:0] state;

//time delay registers
reg [4:0] cur_time; //the location in "time" that we are at with the increment counter
reg [4:0] time_delay; //figures out how much of a delay is necessary for the sample
reg [4:0] delay_loc; //the location in memory the current audio sample should be stored at

//useful things
reg signed [7:0] aud_in; //registered audio input
reg [7:0] theta_reg; //registered input for the theta
reg [2:0] state; //the state we are in
reg [11:0] big_audio_data; //outpur from memory
reg [4:0] reset_counter; //iterates through memory and clears it out

//Sequential Always block
always @ (posedge clk) begin
if(reset)
state <= clear_state;
else
case(state)
hold_state: begin  //state where we wait for sample request
```

```
werf <= 0;
if(sample_request) begin
audio_data_out <= audio_data_out_reg;
if(cur_time == 23)//update the current place in memory
cur_time <= 0;
else
cur_time <= cur_time + 1;

if(to_right_ear)  //setting it up for each ear
theta_reg <= theta + 192;
else
theta_reg <= theta + 64;

aud_in <= audio_data_in; //registering the audio data
valid_audio_out <= 0; //on a new sample the audio out is no longer valid
state <= mult_state; //resetting the state for another go
end //end sample request if statement
end

mult_state: begin //first multiply of (1-cos)*48000, outputs data in form
mult_a <= {13'b0, (19'b001___0000_0000_0000_0000-{cos_wire[17], cos_wire})}; //1-cos(theta+<shift>)
new_mult_data <= 1;
if(mult_done)//if we are done with the multiply
state <= mult_done_state;
end

mult_done_state: begin
time_delay <= mult_out_wire[36:32];
new_mult_data <= 0;
//setting up the addresses so we can read next clock cycle
werf <= 0;
addr <= cur_time;
//next state
state <= get_data_and_clean_out_state;
end

get_data_and_clean_out_state: begin //reading and setting up for cleaning out data
aud_read_data <= aud_read_data_wire;  //getting audio infomation
num_accesses <= access_read_data;
addr <= cur_time; //emptying this location
access_write_data <= 5'b0; //set to empty
aud_write_data <= 12'b0; //set to empty
werf <= 1; //allow empty
if(time_delay == 0)
state <= sum_output_state;   //sum the delayed audio from memory and the current sample
else
state <= no_sum_output_state;   //just output the delayed audio
end

sum_output_state: begin //sum delayed audio with current audio sample
werf <= 0;
big_audio_data <= (aud_read_data + {{4{aud_in[7]}}, aud_in});
case(num_accesses)
0: audio_data_out_reg <= big_audio_data[7:0];
1: audio_data_out_reg <= big_audio_data[8:1];
2: audio_data_out_reg <= big_audio_data[8:1];
3: audio_data_out_reg <= big_audio_data[9:2];
4: audio_data_out_reg <= big_audio_data[9:2];
```

```
5: audio_data_out_reg <= big_audio_data[9:2];
6: audio_data_out_reg <= big_audio_data[9:2];
7: audio_data_out_reg <= big_audio_data[10:3];
8: audio_data_out_reg <= big_audio_data[10:3];
9: audio_data_out_reg <= big_audio_data[10:3];
10: audio_data_out_reg <= big_audio_data[10:3];
11: audio_data_out_reg <= big_audio_data[10:3];
12: audio_data_out_reg <= big_audio_data[10:3];
13: audio_data_out_reg <= big_audio_data[10:3];
14: audio_data_out_reg <= big_audio_data[10:3];
15: audio_data_out_reg <= big_audio_data[11:4];
16: audio_data_out_reg <= big_audio_data[11:4];
17: audio_data_out_reg <= big_audio_data[11:4];
18: audio_data_out_reg <= big_audio_data[11:4];
19: audio_data_out_reg <= big_audio_data[11:4];
20: audio_data_out_reg <= big_audio_data[11:4];
21: audio_data_out_reg <= big_audio_data[11:4];
22: audio_data_out_reg <= big_audio_data[11:4];
23: audio_data_out_reg <= big_audio_data[11:4];
default: audio_data_out_reg <= 8'b0;
endcase
valid_audio_out <= 1;
state <= hold_state; //don't write anything to memory
end

no_sum_output_state: begin //output delayed audio sample and write to memory
werf <= 0;
big_audio_data <= aud_read_data;
case(num_accesses)
0: audio_data_out_reg <= big_audio_data[7:0];
1: audio_data_out_reg <= big_audio_data[7:0];
2: audio_data_out_reg <= big_audio_data[8:1];
3: audio_data_out_reg <= big_audio_data[8:1];
4: audio_data_out_reg <= big_audio_data[9:2];
5: audio_data_out_reg <= big_audio_data[9:2];
6: audio_data_out_reg <= big_audio_data[9:2];
7: audio_data_out_reg <= big_audio_data[9:2];
8: audio_data_out_reg <= big_audio_data[10:3];
9: audio_data_out_reg <= big_audio_data[10:3];
10: audio_data_out_reg <= big_audio_data[10:3];
11: audio_data_out_reg <= big_audio_data[10:3];
12: audio_data_out_reg <= big_audio_data[10:3];
13: audio_data_out_reg <= big_audio_data[10:3];
14: audio_data_out_reg <= big_audio_data[10:3];
15: audio_data_out_reg <= big_audio_data[10:3];
16: audio_data_out_reg <= big_audio_data[11:4];
17: audio_data_out_reg <= big_audio_data[11:4];
18: audio_data_out_reg <= big_audio_data[11:4];
19: audio_data_out_reg <= big_audio_data[11:4];
20: audio_data_out_reg <= big_audio_data[11:4];
21: audio_data_out_reg <= big_audio_data[11:4];
22: audio_data_out_reg <= big_audio_data[11:4];
23: audio_data_out_reg <= big_audio_data[11:4];
default: audio_data_out_reg <= 8'b0;
endcase
valid_audio_out <= 1;
//setting up for the reading from memory
addr <= ((cur_time +time_delay > 23) ? ((cur_time +time_delay) - 24)  : (cur_time +time_delay)); //setting the delay time
```

```
state <= write_sum_to_mem_state;
end

write_sum_to_mem_state: begin
access_write_data <= access_read_data + 1; //num accesses at the delay location + 1
aud_write_data <= (aud_read_data_wire + {{4{aud_in[7]}},aud_in});
//addr doesn't change because we are just updating the data at this location
werf <= 1;
state <= hold_state;
end

clear_state: begin
addr <= reset_counter;
access_write_data <= 5'b0;
aud_write_data <= 12'b0;
werf <= 1;
reset_counter <= reset_counter +1;
if (reset_counter == 24) begin
reset_counter <= 0;
cur_time <= 23;
state <= hold_state;
end
end
endcase
end //end always
endmodule
```

A.12 ITD_module.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Harrison King Hall hkhall@mit.edu
//
// 6.111 Final Project: Virtual Surround Sound
//
// Module Name:     ITD_module
// Date Modified: 18:52 12.09.2006
//
// This is the module that delays frequencies for a specified amount of time
// based upon the theta input to the module and according to the equation
// # cycle delay = 48kHz*8cm*(1-cos(theta + phi))
// -----------------------------
// 34300cm/sec
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
////////////////////////////////////////////////////////////////////////////////
module ITD_module(clk, reset, sample_request, audio_data_in, theta, to_right_ear, audio_data_out, valid_audio_out);
input clk; //system clock 27 Mhz
input reset; //clear all system memory and resets state
input sample_request; //serves as an 48kHz clock provided by the ac97
input signed [7:0] audio_data_in; //PCM data
input [7:0] theta; //is the effective degree resolution of theta
 /*   ^ 0 degrees  (0)
  |
  |
-o^o-
```

```
(191)270 degrees<------( user)------>(63)90 degrees
\___/
  |
  |
  v
180 degrees
 (127)


 */
 input to_right_ear; //1 if signal is going to a right ear/speaker , 0 if it is going to a left ear/speaker
    output [7:0] audio_data_out; //time-delayed audio out
 output valid_audio_out; //since there is a time delay wait until the audio data is valid to output it

//head parameters for the module
//average head radius 8cm, speed of sound 34000cm/s, frequency of sampling 48kHz
parameter sample_freq_head_div_sound =  32'b0000_0000_0000_1011___1011_1000_0000_0000; //48kHz*2^-12

//audio memory regs and wires
wire signed [11:0] aud_write_data_wire; //data to write to the memory
wire signed [11:0] aud_read_data_wire;

//access memory regs and wires
wire [4:0] addr_wire;
wire [4:0] access_data_in_wire; //the data to the access memory
wire [4:0] access_data_out_wire;  //the data from the access memory
wire werf_wire;

//cos wires
wire [7:0] theta_reg_wire;
wire [17:0] cos_wire;

//multiply wires
wire signed [31:0] mult_a;//
wire signed [63:0] mult_out_wire;
wire mult_done; //1 iff the multiply at this step is finished
wire new_mult_data; //does multiply have new data on the inputs

ITD_FSM fsm(
.clk(clk),
.reset(reset),
.sample_request(sample_request),
.audio_data_in(audio_data_in),
.to_right_ear(to_right_ear),
.audio_data_out_reg(audio_data_out),
.valid_audio_out(valid_audio_out),
.mult_out_wire(mult_out_wire),
.mult_done(mult_done),
.new_mult_data(new_mult_data),
.mult_a(mult_a),
.theta(theta),
.cos_wire(cos_wire),
.theta_reg(theta_reg_wire),
.aud_read_data(aud_read_data_wire),
.aud_write_data(aud_write_data_wire),
.werf(werf_wire),
.addr(addr_wire),
.access_read_data(access_data_in_wire),
.access_write_data(access_data_out_wire));


cosine cosine_fn(
.THETA(theta_reg_wire),
.CLK(clk),
.COSINE(cos_wire));

audio_memory audio_delay_memory(
.addr(addr_wire),
.din(aud_write_data_wire),
.we(werf_wire),
.clk(clk),
.dout(aud_read_data_wire));

access_mem num_accesses_mem(
.addr(addr_wire),
.we(werf_wire),
.din(access_data_out_wire),
.dout(access_data_in_wire),
.clk(clk));

mult32x32 cycle_mult(
.a(mult_a),
.b(sample_freq_head_div_sound),
.q(mult_out_wire),
.nd(new_mult_data),
.rdy(mult_done),
.clk(clk));

endmodule
```

A.13 Mono_Source.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Harrison King Hall hkhall@mit.edu
//
// 6.111 Final Project: Virtual Surround Sound
//
// Module Name:     Mono_Source
//  Date Modified: 18:52 12.09.2006
//
// This is the module is the fundamental building block for multiple source
// virtual surroud sound sherical head modeling implementations.  It combines
// the ILD and ITD modules with a simplistic room echo to simulate an average
// sized home theater.
//
//  Revision:
//  Revision 0.01 - File Created
//  Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module Mono_Source(clk, reset, sample_request, audio_data_in, theta,
audio_left_data_out, audio_right_data_out);

input clk; //27 mhz system clock
input reset; //global rest
input sample_request; //sample_request
input [7:0] audio_data_in; //audio from a source that is valid on sample_request
input [7:0] theta; //theta to the sound source
 /*   ^ 0(0 degrees)
    |
  |
-o^o-
  (270 degrees)191<------( user)------>63(90 degrees)
\___/
  |
  |
  v
127(180 degrees)
 */
output [7:0] audio_left_data_out; //audio for the left ear
output [7:0] audio_right_data_out; //audio for the right ear

wire signed [7:0] left_ild_to_itd_wire, right_ild_to_itd_wire, room_echo_out_wire,
left_itd_to_sum, right_itd_to_sum, left_sum_out, right_sum_out;

assign audio_left_data_out = left_sum_out;
assign audio_right_data_out = right_sum_out;

ILD_module ild_left(
.clk(clk),
.reset(reset),
.sample_request(sample_request),
.audio_data_in(audio_data_in),
.theta(theta),
.to_right_ear(1'b0),
.audio_data_out(left_ild_to_itd_wire)
);

ITD_module itd_left(
.clk(clk),
.reset(reset),
.sample_request(sample_request),
.audio_data_in(audio_data_in),
.theta(theta),
.to_right_ear(1'b0),
.audio_data_out(left_itd_to_sum)
);

ILD_module ild_right(
.clk(clk),
.reset(reset),
.sample_request(sample_request),
.audio_data_in(audio_data_in),
.theta(theta),
.to_right_ear(1'b1),
.audio_data_out(right_ild_to_itd_wire)
);

ITD_module itd_right(
.clk(clk),
.reset(reset),
.sample_request(sample_request),
.audio_data_in(audio_data_in),
.theta(theta),
.to_right_ear(1'b1),
.audio_data_out(right_itd_to_sum)
);

RoomEcho_module echo_echo_echo(
.clk(clk),
.reset(reset),
.sample_request(sample_request),
.audio_data_in(audio_data_in),
.audio_data_out(room_echo_out_wire)
);

two_source_sum right_and_echo(
.aud_one(right_itd_to_sum),
.aud_two(room_echo_out_wire),
.aud_out(right_sum_out)
);

two_source_sum left_and_echo(
.aud_one(left_itd_to_sum),
.aud_two(room_echo_out_wire),
.aud_out(left_sum_out)
);
endmodule
```

A.14 RoomEcho_FSM.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Harrison King Hall hkhall@mit.edu
//
// 6.111 Final Project: Virtual Surround Sound
//
// Module Name:     RoomEcho_FSM
//  Date Modified: 18:52 12.09.2006
//
// The RoomEcho_fsm is the controlling FSM for the RoomEcho_module.
//
//  Revision:
//  Revision 0.01 - File Created
//  Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module RoomEcho_FSM(clk, reset, sample_request, write_data, read_data, werf,
addr, new_sample, aud_output);

input clk; //system clock 27 Mhz
input reset; //resets the system
input sample_request; //serves as an 48kHz clock provided by the ac97
output [8:0] addr;
output [7:0] write_data;
input [7:0] read_data;
output werf;
input signed [7:0] new_sample; //PCM data in
output signed [7:0] aud_output; //PCM data out

//registering outputs
```

```
reg [8:0] addr;
reg [7:0] write_data;
reg werf;
reg signed [7:0] aud_output;

reg [8:0] current_pos;
reg [8:0] reset_counter;
reg [1:0] state;

parameter reset_state = 0; //clears memory
parameter clear_state = 3;
parameter hold_state = 1; //waits for sample request while holding data from memory
parameter write_state = 2; //writes data to memory

parameter quiet = 2; //by how much the echo should be silenced

always @ (posedge clk) begin
if (reset) state <= reset_state;
case(state)
reset_state: begin
reset_counter <= 0;
state <= clear_state;
end

clear_state: begin
if (reset_counter == 511) begin
reset_counter <= 0;
current_pos <= 0;
werf <= 0;
addr <= 0;
state <= hold_state;
aud_output <= 0;
end
else begin
addr <= reset_counter;
write_data <= 8'b0;
werf <= 1;
reset_counter <= reset_counter +1;
aud_output <= 0;
end
end

hold_state: begin
if(sample_request) begin
aud_output <= read_data;
current_pos <= current_pos + 1;
werf <= 1;
addr <= current_pos;
write_data <= new_sample>>quiet; //quieting the echo
state <= write_state;
end
end

write_state: begin //the write to mem happens on this clock cycle
//setup for read
werf <= 0;
addr <= current_pos;  //next place in memory
state <= hold_state;
```

```
end

default: state <= reset_state;
endcase
end//end posedge clk
endmodule
```

A.15 RoomEcho_module.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Harrison King Hall hkhall@mit.edu
//
// 6.111 Final Project: Virtual Surround Sound
//
// Module Name:     RoomEcho_module
//  Date Modified: 05:50 12.10.2006
//
// The RoomEcho_module is aimed at generating a simple room simulation for
// the module where the same signal is applied to both ears only delayed by
// approximately 10 ms.  While this is a deceptively simple model for
// simulation its aim is to generate an "out of head" sensation so the source,
// though from correct azimuth, does not appear to come from within the head.
//
//  Revision:
//  Revision 0.01 - File Created
//  Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module RoomEcho_module(clk, reset, sample_request, audio_data_in, audio_data_out);

input clk; //system clock 27 Mhz
input reset; //resets the system
input sample_request; //serves as an 48kHz clock provided by the ac97
input signed [7:0] audio_data_in; //PCM data in
output signed [7:0] audio_data_out;

wire [8:0] addr_wire;
wire [7:0] write_wire, read_wire;
wire werf_wire;

room_delay_memory delay_mem(
.clk(clk),
.addr(addr_wire),
.din(write_wire),
.dout(read_wire),
.we(werf_wire));

RoomEcho_FSM fsm(
.clk(clk),
.sample_request(sample_request),
.reset(reset),
.write_data(write_wire),
.read_data(read_wire),
.werf(werf_wire),
.addr(addr_wire),
.new_sample(audio_data_in),
.aud_output(audio_data_out));
```

```
endmodule
```

A.16 synchronize.v

```
// pulse synchronizer
module synchronize(clk,in,out);
  parameter NSYNC = 2;  // number of sync flops.  must be >= 2
  input clk;
  input in;
  output out;

  reg [NSYNC-2:0] sync;
  reg out;

  always @ (posedge clk)
  begin
    {out,sync} <= {sync[NSYNC-2:0],in};
  end
endmodule
```

A.17 triport_memory_controller.v

```
‘timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Harrison King Hall hkhall@mit.edu
//
// 6.111 Final Project: Virtual Surround Sound
//
// Module Name:     triport_memory_controller
// Date Modified: 18:52 12.09.2006
//
// This is a wrapper for a memory module that multiplexes between 3 inputs and
// outputs based on a ownership signal.
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
////////////////////////////////////////////////////////////////////////////////
module triport_memory_controller(clk, owner, fsm_werf, fsm_din, fsm_addr, fsm_dout, mute_werf,
mute_din, mute_addr, mute_dout, fft_werf, fft_din, fft_addr, fft_dout);

input clk;
input [1:0] owner;
input fsm_werf;
input [7:0] fsm_din;
input [2:0] fsm_addr;
input mute_werf;
input [7:0] mute_din;
input [2:0] mute_addr;
input fft_werf;
input [7:0] fft_din;
input [2:0] fft_addr;
output [7:0] fsm_dout;
output [7:0] fft_dout;
output [7:0] mute_dout;

//owner parameters
parameter fsm_owner = 2’b00;
parameter fft_owner = 2'b10;
parameter mute_owner = 2'b11;

wire [2:0] addr_wire;
wire werf_wire;
wire signed [7:0] write_data_wire, read_data_wire;

//assigning inputs
assign addr_wire = (owner == fsm_owner) ? fsm_addr : (owner == fft_owner) ? fft_addr : mute_addr;
assign write_data_wire = (owner == fsm_owner) ? fsm_din : (owner == fft_owner) ? fft_din : mute_din;
assign werf_wire = (owner == fsm_werf) ? fsm_addr : (owner == fft_owner) ? fft_werf : mute_werf;
//assigning outputs
assign fsm_dout = (owner == fsm_owner) ? read_data_wire : 8'b0;
assign fft_dout = (owner == fft_owner) ? read_data_wire : 8'b0;
assign mute_dout = (owner == mute_owner) ? read_data_wire : 8'b0;

complexnum_mem memory(
.addr(addr_wire),
.we(werf_wire),
.din(write_data_wire),
.dout(read_data_wire),
.clk(clk));

endmodule
```

A.18 two_source_sum.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Harrison King Hall hkhall@mit.edu
//
// 6.111 Final Project: Virtual Surround Sound
//
// Module Name:     two_source_sum
// Date Modified: 18:52 12.09.2006
//
// Simple combines 2 audio sources into 1 and takes the most significant bits.
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
////////////////////////////////////////////////////////////////////////////////
module two_source_sum(aud_one, aud_two, aud_out);
    input signed [7:0] aud_one;
    input signed [7:0] aud_two;
    output signed [7:0] aud_out;

assign aud_out = (aud_one + aud_two) >> 1; //just taking the most significant bits

endmodule
```