Sound Blocks

An Interactive Environment for Real-Time Music Creation

Dennis Miaw & Iris Cheung

6.111

12/13/06

Abstract

The purpose of the Sound Blocks is to provide a unique and interactive interface for creating music in real-time. A user is able to mix, match, and modify sounds by placing and manipulating blocks on an interactive table. There are three types of blocks: source, local-effect, and global-effect. A source block generates sound, such as a drum loop or a bass line. Local-effect blocks, when placed near a source block, modify the waveform of the source block, producing effects such as reverb and delay. A global effect block controls a property that affect the entire system, such as volume.

Table of Contents

List of Figures

INTRODUCTION

The Sound Blocks project consists of a camera which is used to identify colored blocks placed on a table. The location and the colors of the blocks are used to control the playback of music and sounds. Placing blocks on specific regions of the table determines whether or not a sound will be played, or whether or not an effect will be applied to another source. If an effect is applied, the specific location of the block also indicates the magnitude or intensity of the effect. Currently, only red, green, and blue, are being detected, which allows for the capability of triggering three separate sound sources, and three different effects, each of which can be applied to any or all of the sources. The project was implemented with the intention of having four sources and four effects, but time constraints only allowed for the detection of three different colors.

The inspiration for the project Sound Blocks came from a project exhibited in the Emerging Technologies SIGGRAPH 2006 convention called the *Reactable* <**http://mtg.upf.edu/reactable**/>. Like the *Reactable*, the Sound Blocks environment allows multiple users to manipulate an intuitive and free form music generation system by placing and moving blocks on a table. Such a system would be entertaining and easy to use for children and adults of all ages. The Sound Blocks system could be used as a learning tool for studying sound and music, or a performance tool used by musicians.

The objectives of the project are:

- To take a video feed from a camera and design the digital circuitry for determining the location of different blocks on a table based on color-recognition.
- To play a sound file which corresponds to a particular source block color.
- To associate local-effect blocks with source blocks based on relative position to create sound effects such as delay and reverb.
- To implement a global volume block used to change the total volume of the output.
- To use location as a variable for changing the magnitude of effect applied to a source.

OVERVIEW

IMAGE CONTROLLER (Miaw)

The image/visual portion of the project deals with receiving and processing the data coming from the camera input. The camera data is used to determine the location of red, green, and blue colored blocks on the table surface, and the positions and colors are used to determine which sound sources should be played back, and which effects should be applied to the sources. The camera data is also sent through image processing modules provided by the MIT 6.111 course to display the camera image onto the computer monitor. Ultimately, the image portion of the project outputs two values: *connections*, which indicates the sound sources that are present on the table and the effects that should be applied to each source, and *magnitude*, which indicates the magnitude or intensity of each effect. These two values are used by the Audio Controller portion of the project to

actually produce and output the correct sound. See **Figure 1** for a block diagram of the image portion of the project.

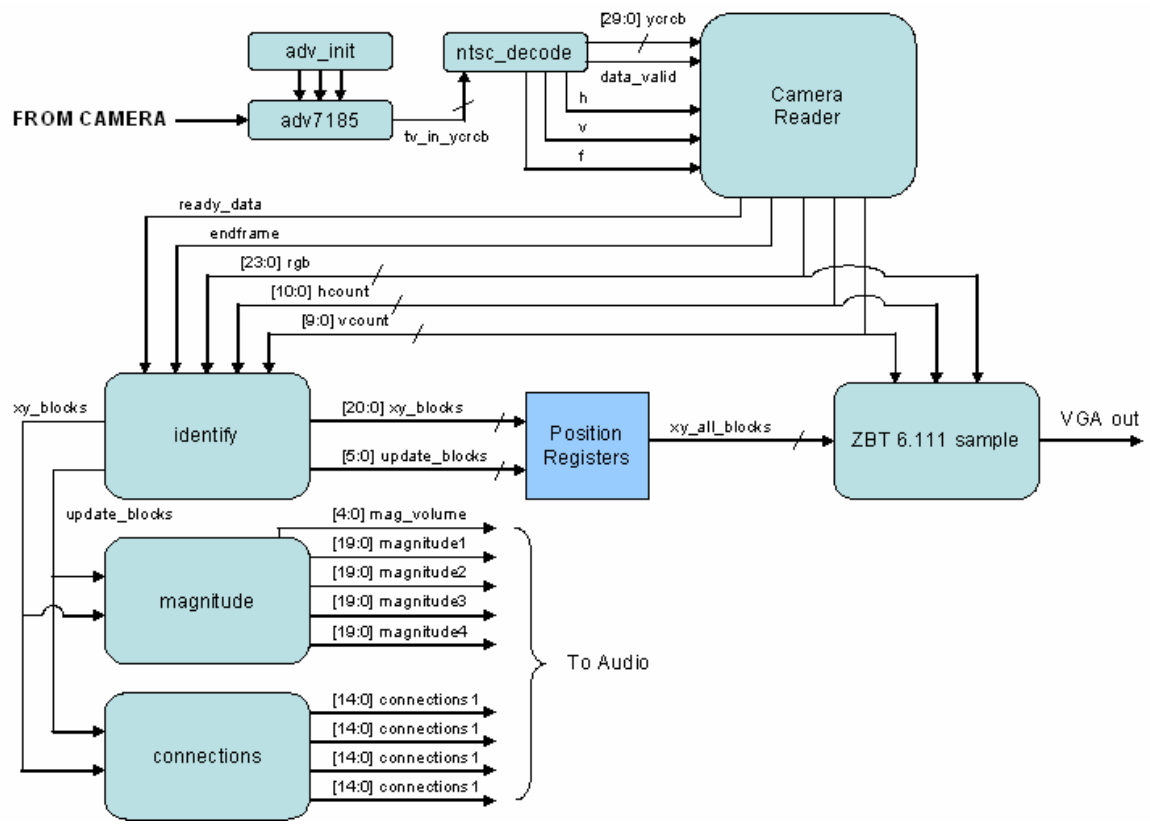

Figure 1: Block Diagram of the Image Controller
Block diagram of the image processing portion of the project, showing the modules and their connections.

MODULE DESCRIPTIONS FOR THE IMAGE CONTROLLER:

Camera Reader Module (Miaw):

The Camera Reader receives the data for *hsync*, *vsync*, *field*, *YCrCb*, and *data_valid* from the NTSC_decode module. The Camera Reader module performs two functions: it converts the incoming *YCrCb* data to *RGB*, and also generates an *hcount* and *vcount* for the incoming camera pixels. The color conversion requires a three-stage pipeline, and follows the standard equations for YCrCrb to RGB conversion. The *hcount* and *vcount* is generated from the *hsync*, *vsync*, and *field* data. When a *data_valid* signal is received, *hcount* increments by one, and when *hsync* goes high, *hcount* gets set to zero and *vcount* increments by two, since the camera image is interlaced. When *vsync* goes high, *hcount* gets set to zero, and *vsync* gets set to either zero or one, depending on the *field* value, again because the image is interlaced.

Because the color conversion is a three-stage pipeline, the *hcount* and *vcount* outputs must also be delayed in order to be in time with the outgoing *RGB* data. This module also

outputs a *ready_data* signal and an *endframe* signal, which correspond to when there is valid data which can be read, and when the end of each frame is reached. These two signals also must be output in time with the rest of the outgoing data signals.

Identify Module (Miaw):

This module receives the *RGB* pixel data along with the associated *hcount* and *vcount* from the Camera Reader module in order to determine the locations of each colored block on the table. The 720x540 pixel camera screen is divided into nine different regions, each of which can contain a red, green, or blue block, with the exception of the region one, which can only contain a red block. Region one controls the overall volume, regions six through nine control the sound sources, and regions two through five control the effects on the sources. A handful of the leftmost and rightmost pixels of the screen are ignored, as these sections tend to be very noisy with respect to color. The useable portion of the *hcount* is actually between 5 and 700. See **Figure 2** for a visual representation of the layout of the regions.

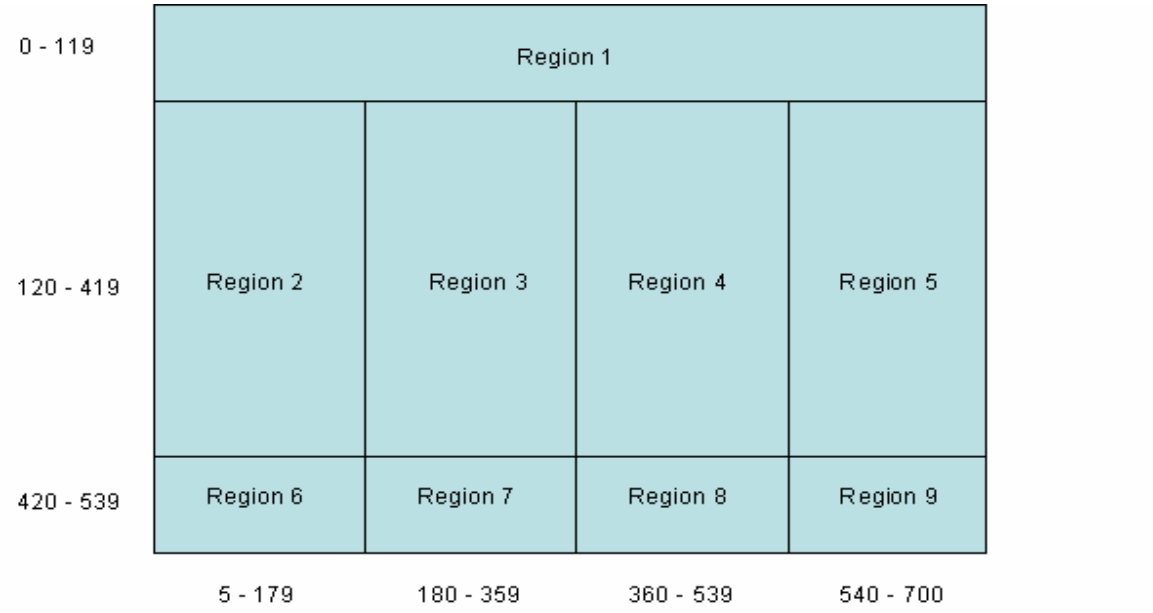

Figure 2: Layout of the Camera view
The layout of the camera image, showing the nine different regions, and their associated hcount and vcount values.

The locations of each block are determined by summing the hcount and vcount values of every red, green, and blue pixel in every region, and then dividing those sums by the total number of pixels of the corresponding color in the region. Therefore, the Identify Module finds the center of mass of red, green, and blue in each region. If the number of pixels counted for any color is less than sixty, however, then the module assumes that the corresponding block is not on the table. This is to help ensure that some random noise will not be accidentally interpreted as a block. The data for each block then sent serially through the signal *xy_blocks*, which contains the hcount and vcount for the block, and the signal *update_blocks*, which indicates which block is currently being sent. If a block is

not on the table, then a specific preset value is sent instead of a valid coordinate. This process occurs once at the end of each frame, when a high *endframe* signal is received from the Camera Reader.

Position Registers Module (Miaw):

The role of the Position Registers is to receive and store the data from the Identify Module so that it can be sent to the VGA display. It was found during testing that the VGA display was much cleaner and had less noise if the data for the blocks was sent all at once rather than serially. Therefore, the Position Registers is simply a group of registers which holds the current value of the location of each block. The registers get updated when it receives new data from the *xy_blocks* and *update_blocks* signals from the Identify Module. It outputs *xy_volume*, which is the coordinate of the red block in region one, *xy_effects*, which contain the coordinates of the effect blocks in regions two through five, and *xy_sources*, which contain the coordinates of the source blocks in regions six through nine.

Connections Module (Miaw):

This module determines which effects are connected to which sources, and in what order. It receives only the x-coordinate and the *update_blocks* data serially from the Identify Module. For the effects in any effect region (regions 2-5), priority goes in order from left to right. Therefore, the leftmost effect block gets connected first, and then the next leftmost gets connected second, and so on. In general, changing the order in which effects are applied to a sound can change the resulting sound at the end, so it is useful to be able to control he order. For source blocks, only one source can be in a source region (regions 6-9) at a time, and only one source of the same color can be on the table at a time. If more than one source is placed in the same region or if the same color block is placed in two or more source regions, priority is given to the source that was placed first. The other sources will produce no sound. Playback of multiple sources at a time is achieved by placing different source blocks into different source regions.

The Connections Module builds the connections by first comparing the x-coordinates, or hcount values, of the effect blocks in each effect region. This way, the order of the effects in each effect region can be determined. The module then associates these effects with a particular source. To do this, the module checks to see which sources are in each source region. Essentially, there is a hold signal for each color (*red_hold*, *green_hold*, *blue*_hold), which gets set high whenever a source of that color is placed on the table, blocking other sources of the same color from having an effect. There is also a similar hold signal for the colors in each column (*red_hold1*, *green_hold1*, *blue_hold1*, *red_hold2*, *green_hold2*, etc), which prevent other sources from becoming activated if there is already a source in that column. Once it determines which source is active in each region, it can create the *connections* output, which consists of four 15-bit-long lists of three-bit numbers. Each of these 15-bit-long lists is associated with one source, and the list contains all the information describing which effects are connected to which sources, as well as the order of the effects for each source.

The Connections Module also outputs a value, *source_columns*, which indicates which column each source is in. This is combined with the output from the Magnitude Module to determine the magnitudes of the effects associated with each source.

Magnitude Module (Miaw):

The Magnitude Module determines the magnitude of each effect on the table. Moving the effect blocks higher up in the cameras view causes the magnitude of those effects to increase. Moving the volume block farther to the right causes the volume to increase. To achieve this functionality, this module receives the *xy_blocks* and *update_blocks* data serially from the Identify Module, and uses the x-coordinate of the red block in region one to determine the volume, and the y coordinates of the remaining effect blocks to determine their respective magnitudes. The bottom five bits of the x data or y data are ignored, in order to provide a tolerance for some slight noise in the position of the blocks. This way, a change of one in the magnitude occurs with every change of thirty-pixels on the camera image. Four sets of magnitudes are output: *mg*, *m1*, *m2*, *m3*, and *m4*, which correspond to the magnitude for the volume, the effects in the first column, the second column, the third column, and the fourth column. The *source_columns* output from the Connections Module is then used to associate the correct column with the correct source.

AUDIO CONTROLLER (Cheung)

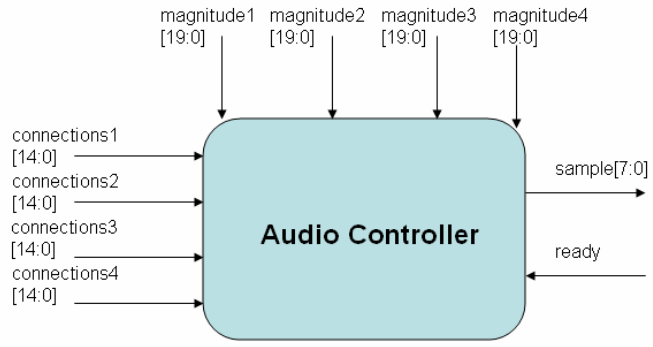

Figure 3. Audio Controller
The audio controller module takes the outputs of the magnitude and connections modules and outputs an 8-bit audio sample to the ac97 audio decoder chip every 48kHz.

The audio controller is responsible for producing an 8-bit audio sample that reflects the configuration of the blocks positioned on the interactive table. A new audio sample is given to the ac97 codec every 10 ready signals to produce a stream of music at a 4.8kHz sampling rate. The audio controller uses the outputs from the connections module to apply any local or global effects to the appropriate source block. The output of the magnitude module is used by the audio controller to apply the appropriate amount of effect to each source block.

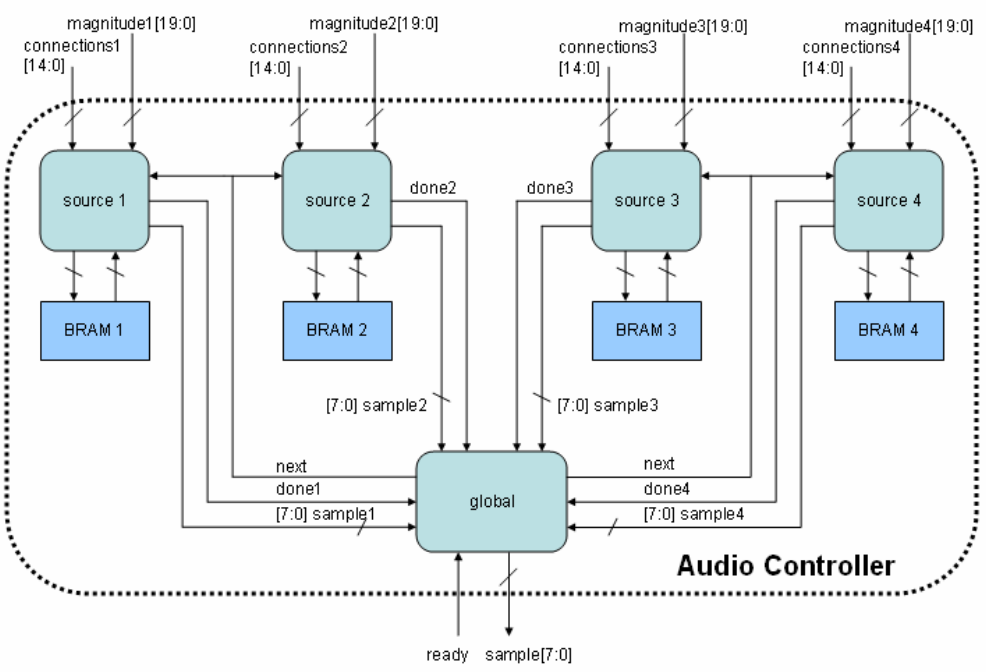

Figure 4. Audio Controller Modules
Each source module parses its corresponding connections and magnitude data and signals a done signal to the global module when it has fully computed an audio sample to be sent to the ac97 audio codec

MODULE DESCRIPTIONS FOR THE AUDIO CONTROLLER

Source Module (Cheung)

Inside the audio controller are two types of modules: source and global (See **Figure 4**). The purpose of a source module is to produce an 8-bit audio sample corresponding to a particular source block based on: the existence of the source block on the table, the effects that have been applied to the source block, and the magnitude of effect(s) applied to the source block. There are four instantiations of the source module in the audio controller, one for each of the four source blocks in our system. Each source module instantiation receives a unique set of connections and magnitude information relevant to that particular source block. Furthermore, each source module connects to its own block RAM (BRAM) which contains the audio sample data of that particular source sound.

Global Module (Cheung)

The global module is responsible for summing the audio samples produced by each source module to create the effect of multiple source blocks being played simultaneously. The global module waits to receive a done signal from each of the source modules before it calculates the sum and stores the resulting value in a register. The global module provides a new sample to the ac97 every 10 ready signals to produce the 4.8kHz outgoing data stream. After providing the ac_97 codec with a new sample, the global module

sends a next signal to each of the source modules to begin the processing of the next sample.

Source Module Finite State Machine (FSM) (Cheung)

A source module produces an audio sample with the appropriate effects applied to it by parsing the output of the connections module, which contains information about the configuration of the blocks on the table with respect to one another. **Figure 5** decodes the output from the connections module for source module1 by labeling each of the wires with the corresponding local effect blocks they describe. The three most significant bits of the output connections1 [14:0] are a binary representation for which local effect is immediately connected to Source Block 1. In **Figure 5**, Source Block 1 is immediately connected to Local-Effect Block 1 (001). By reading the three least significant bits in the connections1 [14:0] wires, which correspond to the local-effect block that is connected to Local-Effect 2, one can see that Local-Effect1 is connected to Local-Effect 2 (010). Finally, by reading the connections1 [5:3] wires, which correspond to the local-effect block that is connected to Local-Effect 2, we see that bits [5:3] contain a value of 111. A value of 111 represents the state where a block does not connect to any additional blocks. In summary, **Figure 5** describes the following configuration of blocks on the table: the Source Block 1 is connected to Local-Effect Block1; Local-Effect Block1 is connected to Local-effect Block 2; and finally Local-Effect2 Block 2 is not connected to any additional blocks.

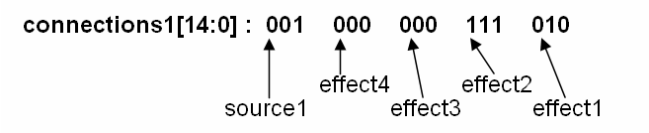

Figure 5. Connections
Each three-bit segment of the connections module output has a pre-determined representation to be used by the source module FSM for applying sound effects to audio samples.

In each source module there is a finite state machine (FSM) which uses the data provided by the output of the connections module to control the application of effects onto each audio sample. A state transition diagram of the source module FSM is shown in **Figure 6**. The first state in the source module FSM is called “Source”. The “Source” state can only be entered if a “next” signal is received, signifying the demand for a new audio sample from the source module. After the “next” signal is received, the current sample is read and stored (or “loaded” as described by **Figure 6**) into a global sample register which will continuously be modified and updated with a newly modified sample as the state machine transitions from one state to the next. Referring back to **Figure 5**, the first three bits in connections1 [14:0] correspond to Source Block 1 being connected to Local-Effect1 (001). Therefore, by following the state transition diagram, in **Figure 6**, one can observe that the FSM transitions to the Local-Effect 1 state. While in the Local-Effect 1 state, the appropriate logic is then executed to apply Local-Effect 1 to the current audio sample and then store it into the global sample register. This process of transitioning

from local-effect state to local-effect state continues until a done signal (“111”) is parsed, in which case the FSM halts transitioning until the “next” signal is once again received.

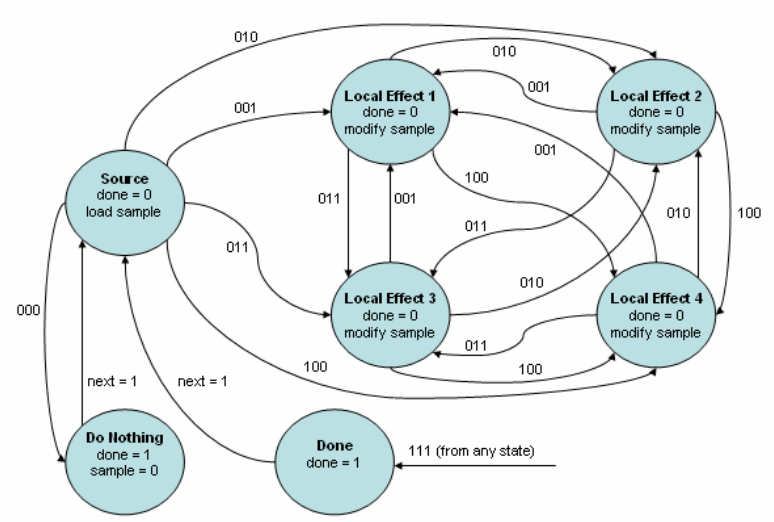

Figure 6. Source Module Finite State Machine (FSM)
The finite state machine within each source module controlling the application of the local-effects to the source block sounds.

IMPLEMENTATION OF LOCAL EFFECTS (Cheung)

Two types of local-effect blocks were implemented: delay, and reverb. Delay was implemented by reading an audio sample from the BRAM at an address earlier than the current address being read from and adding the old audio sample to the current audio sample being sent to the ac97 codec. Reverb was implemented in a similar manner, except that the number of delayed audio samples added to a current sample was on the order of hundreds. Furthermore, to implement reverb, the amount of old sample data being added to the current audio sample decayed proportionally with respect to how long ago an old sample was played. **Figure 7** and **Figure 8** illustrate tests that were conducted using the Logic Analyzer to verify the functionality of the delay and reverb.

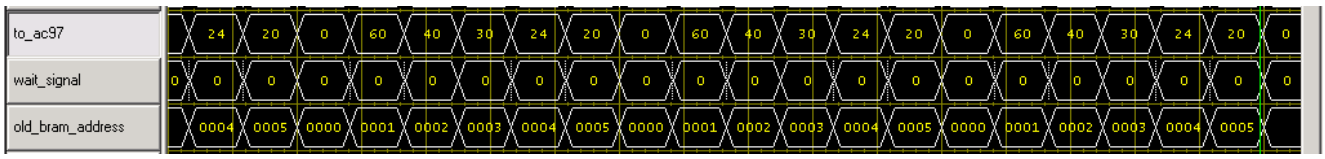

Figure 7. Testing Reverb
Screen shot of the Logic Analyzer measuring the output to the ac97 codec of a pulse signal with a magnitude of 60 with reverb applied to it.

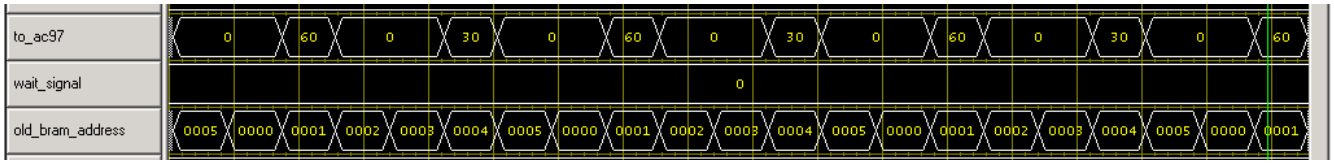

Figure 8. Testing Delay

Screen shot of the Logic Analyzer measuring the output to the ac97 codec of a pulse signal with a magnitude of 60 with a delay scaled by one half applied to it.

MEMORY STORAGE OF SOUND FILES (Cheung)

The ac97 codec expects a data stream at a rate of 48 kHz. To be able to utilize as much BRAM space as possible, the audio samples were stored at 4.8 kHz and then replicated 10 times to produce a 48 kHz data stream. Each audio sample was stored with a width of 8-bits. The size of the BRAM containing a drum-loop in Source Block 1 used 16,536 locations of 8-bits. The other three BRAM's, which contained a bass line, pads, and another synth pad, used 69,044 locations of 8-bits each. The drum loop BRAM was smaller simply because a relatively small number of samples, when played in a continuous loop, could produce an effective drum-loop. A total of 1.79M bits of data were stored using BRAM out of the 2.5M bits available on the labkit.

TESTING AND DEBUGGING AUDIO CONTROLLER (Cheung)

Many tools were utilized in debugging the audio controller, but by far the most useful tool for debugging and testing the Audio Controller was the Logic Analyzer. Often it was not entirely obvious whether or not the delay or reverb effects were being applied properly simply by listening the audio output. To test the working of the delay and reverb effects, a small BRAM-like module was created with a depth of no more than five addresses. Thus, the small BRAM's contents could be manipulated easily. As shown in **Figure 7** and **Figure 8**, by setting the magnitude of only one address in the BRAM to an easy to analyze integer, such as 60, the delay and reverb effects could be easily observed.

CONCLUSION

Overall, we feel that the project was a success. Although the end result deviated quite a bit from the original concept, our fundamental goal of creating an interactive table to control music creation is inherently there. We were pretty much able to achieve all of our goals with regard to having the position of physical blocks control the sound. The idea of having blocks of the same color cause different things to happen based on the region it was in was not part of our original idea. This, however, turned out to be one of the most interesting features of the project, in our opinion. We only wish there was more available memory to hold better quality sound files.

It was also a bit unfortunate, although not really critical, that we were unable to transmit the data serially to the VGA output modules in order to display the locations of the blocks on the screen. Although simpler in concept, transmitting all the blocks simultaneously is not quite as elegant as transmitting each coordinate one after another. Our initial attempt at synchronizing the 27Mhz clock with the 65Mhz clock consisted of a pair of back-to-back registers timed on the 65Mhz clock. This created a huge amount of noise in displaying the location of the blocks on the screen. It was only very late in the project did we learn about using a FIFO module to control the synchronization between the clocks, and at that point we did not have time to implement that design. This would have

definitely been one thing that we would have changed if given the chance to do the project again.

A very important aspect about digital design that we became more and more aware of as the project went on is the issue of timing. Making sure that everything was timed correctly so that modules were reading data at the correct clock cycle, and that they were also outputting data at the correct times was a huge issue, and often caused many seemingly inexplicable errors. Especially when reading from multiple memories, and using pipelined circuits such as dividers, it can be very frustrating to figure out exactly when things need to occur. In general, this caused more setbacks for the audio portion of the project than for the image processing portion. The effect FSMs have to almost explicitly control what happens on every new clock cycle in order to be efficient about applying the effects. This can be confusing to debug when dealing with timing errors.

In the end, we are both very satisfied with the project, and thoroughly enjoyed the process of developing our own project from start to finish.