Keytar Hero

Hubert Hwang
Hui Tang

December 13, 2006

Abstract

The keytar is a musical instrument which, as the name suggests, is a hybrid of a keyboard and a guitar, consisting of a small keyboard worn around the neck like a guitar. Each key plays a synthesized tone of a particular frequency. A functioning keytar, consisting of input handling, tone generation, and audio output modules, was created using modules programmed in Verilog. On top of this base functionality, additional modules were implemented allowing for the selection of different instrument tones, and for adding an echo to the original synthesized tone, allowing the generated sounds to be quite diverse.

Contents

1 Overview **1**

2 Module Description and Implementation **5**
- 2.1 Main Module (Hubert) 5
- 2.2 PS/2 Keyboard Interface (Hui) 7
- 2.3 Sine Wave Generator (Hui) 7
- 2.4 Instrument Bank (Hui) 9
- 2.5 Mixer (Hubert) 9
- 2.6 Echo Module (Hubert) 10

3 Testing and Debugging **11**

4 Conclusions **13**

A Verilog Code **14**
- A.1 Main Module 14
- A.2 PS/2 Keyboard Interface 18
- A.3 Sine Wave Generator 22
- A.4 Instrument Bank 23
- A.5 Mixer 24
- A.6 Echo Module 30

List of Figures

1 Overview of the keytar 2
2 DSP Chain 4
3 Main Module 6
4 Sine Wave Generator 8

1 Overview

The purpose of this project was to build a keytar, which is an electronic instrument that is a hybrid of a keyboard and a guitar. Its playing surface is laid out like a keyboard, but it is worn around the neck like a guitar, and has additional knobs and effects that allow the player to change the sound of the keys. For a keytar to work properly, there are several major pieces that need to be built or programmed. There needs to be a way to press buttons and have those presses register with the instrument, there needs to be a tone generator to map each note to a sine wave, and there needs to be an output module to play the sound so created.

The flow of information in the keytar system (Figure 1) then goes as follows. First, notes are played on some input device. Each note's corresponding sine wave is queried to determine its amplitude at the current point in time, and all of these values are summed up. (The harmonics of the note are also added in, with various constant factors, to simulate different instruments.) The signal is then passed to a chain of digital signal processing (DSP) units, which will alter the signal in various ways. Finally, the signal will be passed to the mixer, which communicates directly with the labkit's onboard AC97 sound chip.

The different tasks are separated into different functional blocks, each performing a fraction of the required behavior. Each block is described briefly below; more detailed descriptions are contained in Section 2.

- **Main Module** – This is the aptly-named main module of this system. This has the responsibility of taking in the note inputs, determining what amplitudes correspond to each note and its overtones at the current time step, and then summing them up to come up with an aggregate signal at the current step. This signal will then get sent to the DSP chain for further processing.
- **PS/2 Keyboard Interface** – Originally, we planned to actually build a keyboard and mount it on a PVC board so that it could be worn around the neck and played like a real keytar. This time required to construct and debug this would have been prohibitive, however, so we instead opted to use a computer keyboard as our input device. This module takes PS/2 keyboard data, transmitted as a serial protocol, and converts it to an array of 38 note flags (`notes`) that will be used in the main module to determine which notes have been hit.
- **Sine Wave Generator** – This module is used to generate sine waves of arbitrary frequencies within a given range. This is done by pre-generating a low-frequency, high-resolution sine wave and then downsampling it at the appropriate ratio. The values for the sine wave are precomputed and stored in a ROM. Then, the main module will issue a series of queries to the sine wave generator, giving a current time step and a frequency. The sine wave generator will then return the amplitude for a sine wave of that frequency at the given time step.
- **Instrument Bank** – This module is used to select what type of sound will be produced by the keytar. A change in overtone content will change what instrument the

OVERVIEW

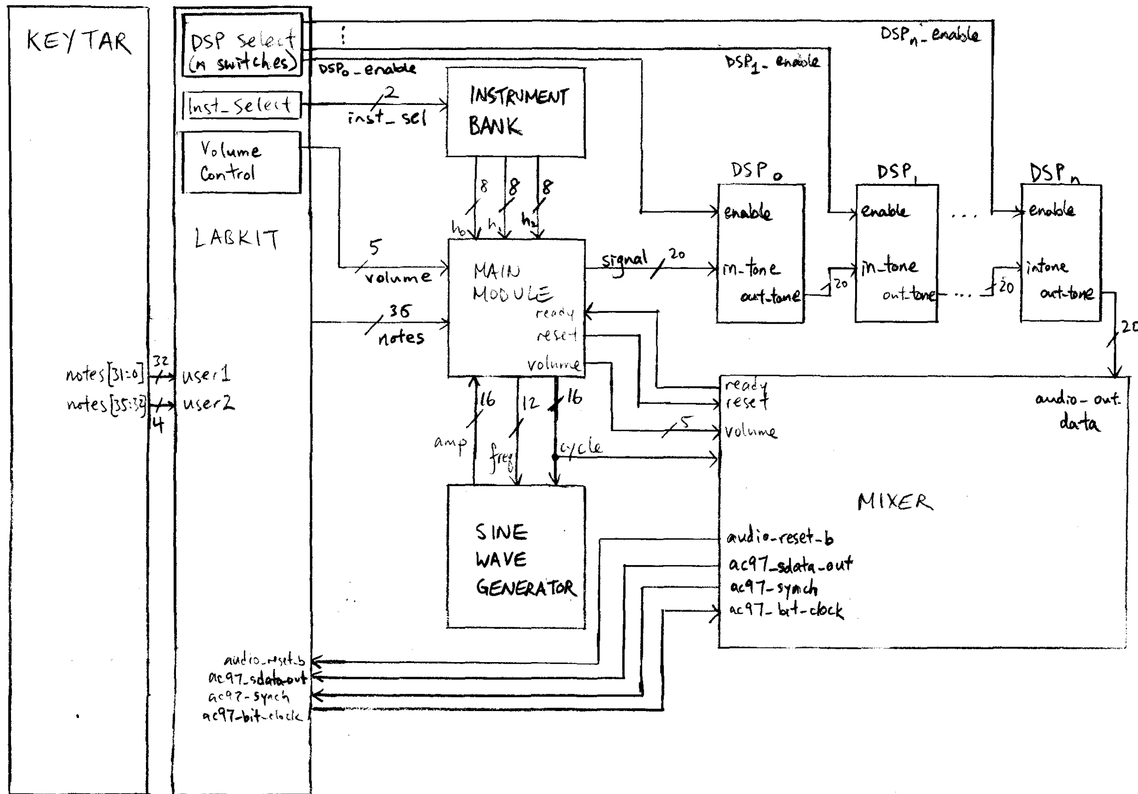


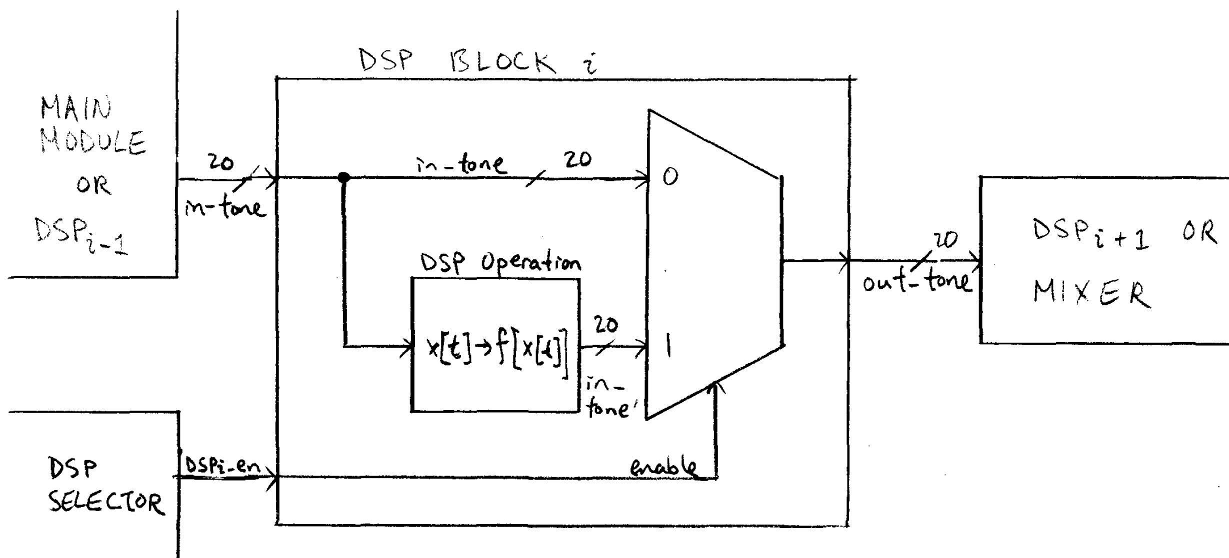
Figure 1: Overview of the keytar

keytar is emulating. This module takes in one input indicating the index of the instrument to emulate, and outputs the relative ratios for the basic harmonic and its next few overtones. These ratios will be used in the main module to determine how to sum up the different amplitudes of the notes played.

- **Mixer** – This module takes a time-variant signal and communicates with the labkit's onboard AC97 chip. This does the work of translating simple monaural data (from the main modules and DSP chain) and converting it into the complex format used by the AC97.

- **DSP Chain** – The DSP chain (Figure 2) comprises a number of modules that are strung together to alter the originally generated signal in some way. Each of the modules takes in a signal from the preceding DSP module in the chain (or the main module) and sends a processed signal to the next (or to the mixer). Each module may be individually enabled or disabled, where a disabled module simply passes through its input. The only module implemented for the DSP chain was an echo module, which saves previous samples and adds them back to the signal, scaled down, at a later time step.

DIGITAL SIGNAL PROCESSING (ONE BLOCK)



ECHO: $f[x[t]] = x[t] + \alpha x[t-\beta]$;
α = amplitude ratio $\ll 1$; β = time offset

Figure 2: DSP Chain

2 Module Description and Implementation

2.1 Main Module (Hubert)

This is the main module of the component. It has the responsibility of taking the note input and generating a signal corresponding to the notes pressed, 48000 times per second. To do this, it also interacts with the sine wave generator and the instrument bank.

Every time a `ready` pulse is received from the mixer, the main module needs to calculate the next sample to send. The sample is determined by several factors: which notes are being pressed, the current instrument (which has different harmonic ratios), and the current time step. The procedure for calculating the signal is as follows.

1. For each note:
   (a) Get the note's associated frequency.
   (b) Pass the frequency and the current time step to the sine wave generator. It will return an amplitude after some amount of latency.
   (c) Scale this amplitude down by the scale factor for the first harmonic (`h_zero`).
   (d) If the note has been hit, add this scaled amplitude to the running total.
2. Repeat step 1 for the four higher harmonics, retrieving the appropriate harmonic frequency and scale factor.
3. Send the top 20 bits of the running total to the DSP chain (`signal`).

Getting the note's associated frequency is handled by the `note_frequencies` module. This basically uses a big switch statement to look up the appropriate frequency given the note's index. It was decided that the number of values was small enough that using a BRAM was not worth the extra effort.

Once the base frequency is retrieved, it is multiplied by the index of the harmonic to get the final frequency. This is then passed to the sine wave generator. After a certain amount of latency, the associated amplitude is returned. Then the amplitude is multiplied by the value retrieved from the instrument bank and added to the running total if the note has been hit. Finally, at the end of the loop, the `signal` output is set to the top 20 bits of the running total.

Most of this is straightforward. The most complicated interaction is that between the main module and the sine wave generator. This is because the sine wave generator must use a divider. The divider provided with the Xilinx tools is fully pipelined and can handle one division per clock cycle. However, it has a latency proportional to the size of the operands. This means that the module could not use the simplest approach, which is to send one frequency at a time and wait for the amplitude response – there are not enough cycles. Instead, the approach taken is to send all of the frequencies one after another, one per cycle. The main module will set `swg_data_out` high to indicate that it is beginning to

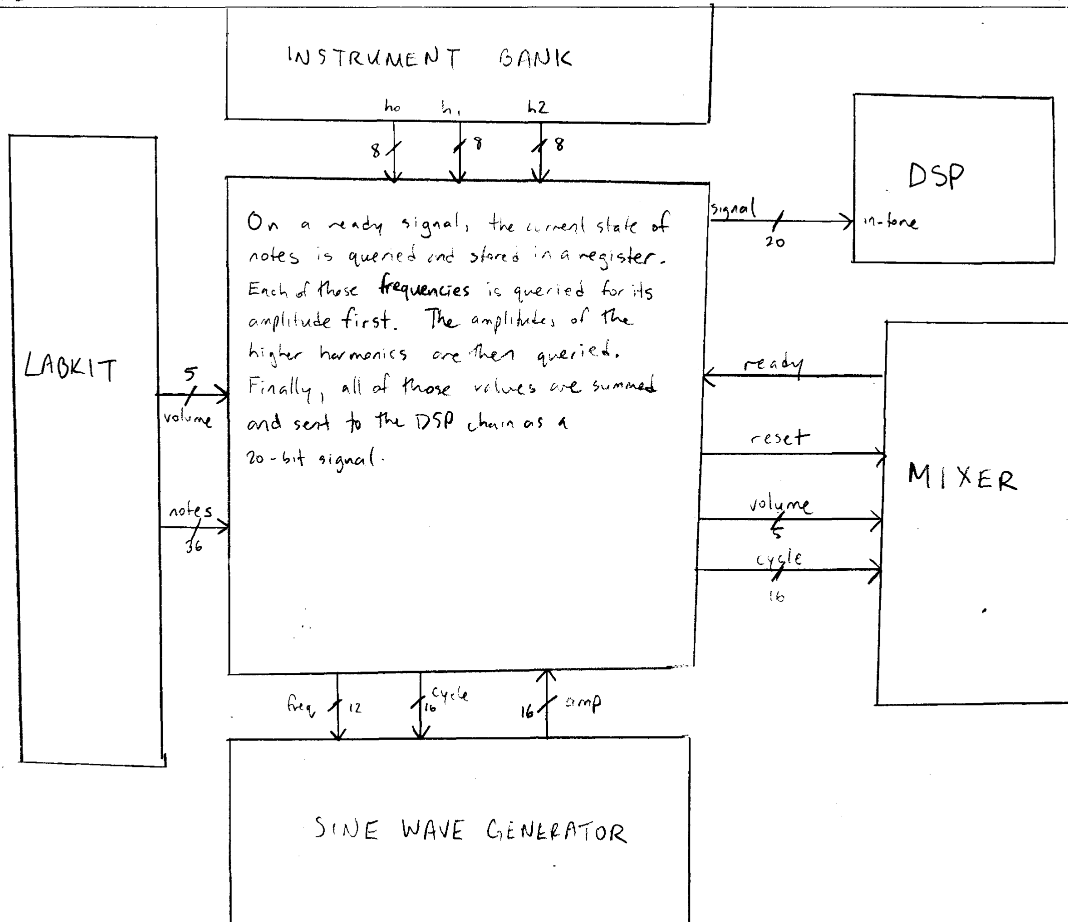


Figure 3: Main Module

send data to the sine wave generator. After the latency time, the sine wave generator will set `swg_data_in` high and send back amplitude responses in the same order, one per cycle. When no more data needs to be sent, `swg_data_out` will be set low again.

2.2 PS/2 Keyboard Interface (Hui)

The PS/2 keyboard sends data serially to the host that is receiving the data. Since the PS/2 uses an active-low setup, all data bits are clocked on the falling edge of the PS/2 clock. The first bit is a start bit. The next 8 bits are the bits for the key code of whichever key is depressed, transmitted least significant bit first. The final two bits are parity and stop bits.

The ps2 module waits for the PS/2 keyboard to send data, and stores the 8-bit key codes into an 8x8-bit FIFO. The data stored in the FIFO is provided to the next module, `ps2_ascii_input`, to read and process as the data comes in. When the FIFO is empty, the ps2 module raises the empty flag, signalling that there is no more data to be read from the keyboard. When there is data in the FIFO, the ps2 module signals to the `ps2_ascii_input` module that there is data waiting to be read.

The `ps2_ascii_input` continuously reads in the key code as long as there is new input to be read. When the FIFO is empty, `ps2_ascii_input` simply holds the last new key code that was read. Upon reading in a new key, the module looks up which note it corresponds to and sets the bit of the appropriate note. If the note is being pressed down, the note bit is set high. If the note is being released, the note bit is set low.

2.3 Sine Wave Generator (Hui)

The sine wave generator takes in a frequency and a time, and produces an amplitude for a sine wave with the input frequency at the input time. The amplitudes are retrieved from a memory block that has stored a sine wave table containing the amplitudes of a base wave of 64Hz at a resolution of 750 evenly spaced samples per period. In order to retrieve the appropriate amplitudes corresponding to the correct frequency, the table of amplitudes needs to be downsampled, which means generating the appropriate memory addresses to look up based on the desired frequency. To do this, the generator takes the product of the frequency and the time, and then divides the top 20 bits of this product by 750. The remainder of this division is the address used to look up the amplitude of the desired wave.

The divider module used by this block is fully pipelined, and its maximum throughput is one division per clock cycle. However, the latency between input and output is proportional to the size (in bits) of the operands. As mentioned in the main module documentation, this means that the most naive approach cannot be used. Instead, this will wait for the `rdy` signal to indicate that frequencies from the main module are arriving. This will then process the requests one by one, getting the remainder and doing the BRAM lookup. When the first result is ready, it will set the `dout` signal high and then start transmitting results back, one per cycle. When no more requests need to be processed, `dout` is then set low.

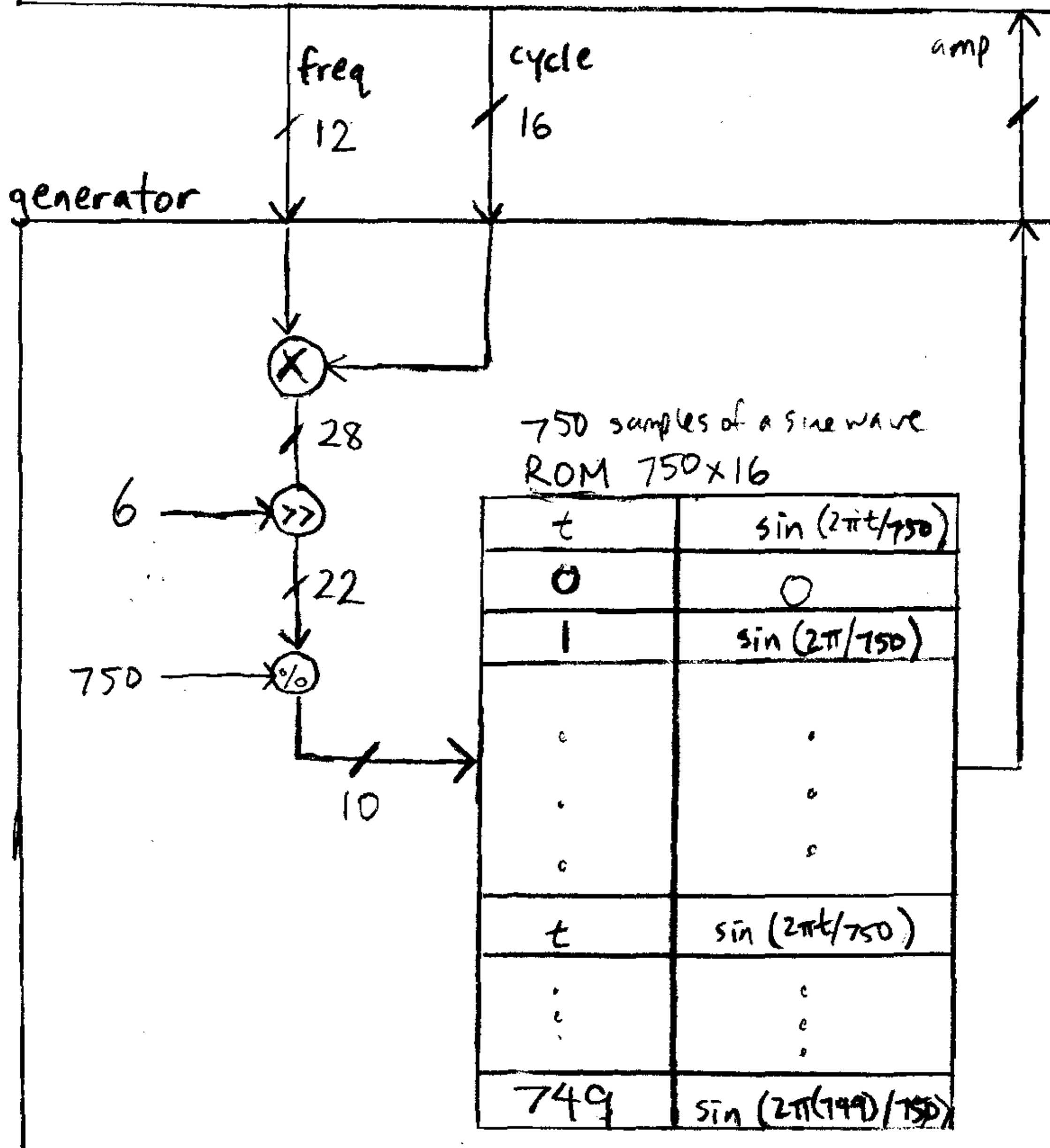


Figure 4: Sine Wave Generator

2.4 Instrument Bank (Hui)

The instrument bank stores the harmonic content of 3 instruments in addition to the pure tone. Different instrument sounds, which can be selected using `switch[1:0]`, are simulated using coefficients to control the relative amplitudes of the harmonics of each note. The first five harmonics in the spectrum are stored.

Since the harmonic contents of the instruments are all hard coded into the module, mapping a particular selection input to a set of harmonic outputs requires simply feeding the selection input into a case statement. Setting both switches to low, that is, selecting 0, gives the pure tone with no higher harmonics. Selection 1 gives a guitar sound. Selection 2 gives a close approximation to a xylophone sound. Selection 3 gives a reedy sound similar to an oboe.

2.5 Mixer (Hubert)

The purpose of the mixer block is to actually allow the generated signal to be made audible. It does this by providing a convenient interface that takes in a single channel of audio input and sends appropriate commands to the onboard AC97 chip to have it output the sound. AC97 is an Intel standard for audio systems; a brief description of the interface follows.

AC97 is a serial interface that passes one bit of data at a time. These bits are divided into 256-bit frames, and 48000 frames are sent per second. Each frame contains a 16-bit tag and twelve 20-bit slots. Slots 1 and 2 are used to set configuration data for the codec, slots 3 and 4 are used to send data to the left and right channels, respectively, and the other slots are unused.

The tag bits have the following meanings:

- Bit 15 (the first bit transmitted) is a valid flag for the entire frame. The entire frame is invalid and ignored if this bit is low.
- Bits 14-3 are valid flags for the individual slots, indicating which slots are valid. Bit 14 corresponds to frame 1, bit 13 to frame 2, and so on. Note that bits 10-3 will always be zero for the purposes of this project.
- Bits 2-0 must be zero.

Slots 1 and 2 may be used to set configuration data. To set configuration data, bit 19 of slot 1 must be low, and then bits 18-12 will specify an eight-bit register address. The rest of the bits in slot 1 will be ignored. The value to store in the register will be in bits 19-4 of slot 2. The rest will again be ignored.

The following registers are the ones relevant to this project:

- Master volume (`0x02`). Bits 19-17 and 11-9 of slot 2 must be zero. Bits 16-12 and 8-4 are five-bit values specifying the attenuation for the left and right channels, respectively. This means that a value of `00000` is the loudest.

- Headphone volume (`0x04`). Controls the volume level sent to the headphone output on the labkit. The format is exactly the same as that of the master volume.

- The PCM volume (`0x18`). Controls the gain applied to signals provided to the chip. Bits 19-16 and 11-8 of slot 2 must be zero. Bits 15-12 and 7-4 are four-bit values specifiying the amount of gain, where `1111` is the most gain.

Slots 3 and 4 are used to send sound data to the left and right channels, respectively. Only the 18 most significant bits of the twenty provided (bits 19-2) are used, and they are transmitted in two's complement MSB order.

To implement this specification, three different modules were used. The `ac97commands` module handles the creation of commands sent in slots 1 and 2. It returns the command in 24 bits: 8 bits for the register index and 16 for the value to store in that register.

The `ac97` module handles serialization of commands and sound data. It keeps an internal counter how far it is into the current frame, in bits, and sends different pieces of data on the serial out line. In particular, it uses the incoming sound data as well as the commands generated by the `ac97commands` module to write frame data correctly.

The `mixer` module is the interface between the previous two modules, the labkit, and the DSP chain. The mixer takes in a monaural signal from the DSP chain, volume controls from the labkit, and sends the serialized frame data to the chip via the `ac97_sdata_out` line. It also has the task of setting the `ready` pulse to let the other parts of the system know when to calculate a new sample.

2.6 Echo Module (Hubert, Hui)

This DSP module provides a way to add reverberation to a signal, by taking sound samples from previous time steps and adding them back to the current signal. This follows the equation:

$$a[x] = a[x] + Ca[x - T]$$

where T is the echo delay and $C < 1$ is the echo constant. The values currently hardcoded into the module are $T = 96000$ (i.e., 2 seconds) and $C = 1/2$.

Because the signals are being generated on the fly, there is no way to get around storing T samples. The module holds a BRAM of size 96000x20 to store the samples, and uses it as a circular buffer. This means that the current index is always the location we want to read from as well as the next location to write to.

Whenever the echo module receives the `ready` signal from the mixer, it will latch the current value of the incoming signal input (`signal_in`) so that it does not vary. Then the processed signal (`signal_out`) is set to `signal_in`, plus the echo if `enable` is high. After this, the latched incoming signal is written to the BRAM by setting `we` high and keeping it that way for five cycles, just to be safe. Finally, the address is incremented for the next `ready` pulse.

3 Testing and Debugging

Due to constraints on when we could be present in lab at the same time, the modules were split between the two of us, and implemented and tested separately. The testing was done both in ModelSim, using Verilog test fixtures, and on the actual labkit by loading the appropriate module and having a human generate all of the inputs to the module that was being tested. Additionally, the logic analyzer was a great help; it allowed us to examine timing diagrams and see which outputs were correct and which were not. It was especially useful to be able to make the timing resolution very coarse to capture macroscopic magnitude effects, and then change it to being very fine to capture every state transition.

This meant that integration was a pain. Theoretically our modules worked according to the specification we had drafted. In practice, bugs abounded. The first major issue was signed versus unsigned numbers. The only way to ensure that a signed multiplication occurs is to have both operands be signed. This caused problems with the original approach to harmonics, which was to store them all in a large array – certain values for harmonic ratios would be incorrectly interpreted as negative numbers, and the generated signal would sound horribly distorted.

This caused other problems. Verilog does not correctly handle sign extension when right shifting. The main module code made heavy use of shift operations without paying any attention to whether the thing being shifted was supposed to be signed or unsigned. When it was supposed to be signed, the resultant signal was always zero.

Once these bugs were fixed, the basic sound generation and playback was working. Pressing certain keys on the keyboard would result in tones being played. Unfortunately, attempting to add the echo module to this was much more work than it should have been.

For the longest time, whether the echo module was enabled or not, no echo resulted – the sound was exactly the same as it was without the echo enabled. After several cycles of testing and recompilation, we discovered that the values read from the echo's BRAM were always zero. This could mean that the address was incorrect, that the reading was incorrect, or that nothing was being written. The first and second possibilities were eliminated early on, but then we were stuck.

A passing TA noticed that the clock variable used to drive the BRAM was unfortunately not the same as the one passed to the echo variable, and so the BRAM was never actually doing anything. Fixing this egregious bug, however, did not fix the problem.

Professor Terman and Gim suggested testing out the write circuitry of the BRAM by simplifying the logic governing write enable and by simplifying the BRAMs being used. The first try involved declaring a single block of BRAM directly and setting it to always write some constant, in this case hex value FF0FF. The logic analyzer showed that the BRAM was failing to write, and outputting zero consistently.

After seeing that the BRAM was failing to write values into memory, Gim suggested moving to a different lab station. Unfortunately, loading the same code onto the new lab kit gave the exact same behavior.

The next test was to generate a new BRAM (at the new lab station) and leave the write enable signal high all the time, to ensure that it would get written eventually. This

finally worked, and started returning values that were supposed to be written in. When the code using this new BRAM was changed back to the required echo logic, it inexplicably started working. Because it seems the old BRAM was just broken, we decided not to alter that part of the code anymore.

Throughout the testing process, we ran into other issues. Most annoying was the fact that when reprogramming, the labkit would not reset itself properly and go into completely random states. Most often nothing would play until the labkit was hard-reset and reprogrammed. Sometimes the output signal came out extremely staticky and distorted. In one case moving from an E to the D# right below it made the tone jump up several octaves. These results could only be fixed by rebooting the labkit, but sometimes we had difficulty realizing that it was a reset issue as opposed to a bug introduced into the code.

Once everything was working, all that was left was to tweak the values provided by the instrument bank. We attempted to search for harmonic spectra for various instruments and found some corresponding to acoustic guitar and xylophone. The actual result does not sound particularly close to those instruments because attack and decay constants were not implemented. Random experimentation with values for the constants led to a third instrument that sounded similar to an oboe, so we kept it.

4 Conclusions

The core objectives of this project were met in their entirety – everything in the checklist that was not marked optional was implemented and integrated into the system. At this point it is possible to play songs, using the computer keyboard as a (clunky and imperfect) interface to the keytar, and so we feel quite satisfied with the result.

In terms of the block structure of the system, the initial design was not changed very much at all. Very minor changes were made to the inputs and outputs of certain blocks in order to provide additional functionality (changing the number of harmonics to five, adding two more keys to the notes array), or in order to fix bugs (changing the size of certain inputs and outputs to allow signed multiplications to work). Despite this, it was never in doubt what the overall system was supposed to look like, which suggests that the original design was reasonably sound.

Knowing what we know now, it would be much easier to do this project all over again. Most of the bugs were caused by inexperience or carelessness, or by circumstances outside of our control. Additionally, we might have attempted to use the ZBT memory instead of generating very large BRAMs. This would cut down on the number of possible causes for write failures.

A Verilog Code

A.1 Main Module

```
module main(clk, notes,
            h_zero, h_one, h_two, h_three, h_four,
            freq, cycle, amp, swg_data_out, swg_data_in,
            ready, reset,
            signal);

   // labkit
   input clk;
   input [37:0]  notes;        // notes from the keyboard

   // instrument bank
   input signed [8:0] h_zero, h_one, h_two, h_three, h_four;

   // sine wave generator
   output [11:0] freq;          // the frequency sent to the swg
   output [15:0] cycle;         // the number of ready pulses received
   input signed [15:0]  amp;    // the amplitude from the swg -- signed 2's complement
   output        swg_data_out; // indicates that we will send data to the swg
   input         swg_data_in;  // indicates that data is to be received

   reg [15:0]    cycle = 0;
   reg           swg_data_out;

   // mixer
   input         ready;         // when the AC97 is ready for another sample
   output        reset;         // a reset signal.

   // DSP chain
   output [19:0] signal;        // the sample to be sent to the DSP

   // internal workings
   reg signed [26:0] sum;     // the register in which to store sums of amps

   reg [8:0] current_out; // the current output index
   reg [8:0] note;        // the current output note
   reg [2:0] harmonic_index_out; // the current harmonic number for out

   reg [8:0] current_in;         // the current input note
   reg [2:0] harmonic_index_in;  // the current harmonic number for in
```

```
wire [11:0] freq_temp;   // temp wire connecting the nf -> hf
note_frequencies nf(note, freq_temp);
harmonic_frequencies hf(freq_temp, harmonic_index_out, freq);

parameter NUM_NOTES = 38;
parameter NUM_HARMONICS = 5;
parameter TOTAL_NOTES = 190;   // NUM_NOTES * NUM_HARMONICS
parameter MAX_CYCLE = 48000;

// reset is later on?  dunno about this one
assign reset = 0;

assign signal = sum[26:7];

// on a ready signal, send the last computed signal,
// then start computation for the next cycle
always @ (posedge clk) begin
   if (ready) begin
      current_out <= 0;
      note <= 0;
      swg_data_out <= 1;
      current_in <= 0;
      harmonic_index_in <= 0;
      harmonic_index_out <= 0;
      sum <= 0;

      if (cycle == MAX_CYCLE)
        cycle <= 0;
      else
        cycle <= cycle + 1;
   end

   if (swg_data_out) begin
      // frequency for note n should already be handled
      // between the note_frequency and harmonic_frequency bits

      if (note + 1 == NUM_NOTES) begin
         note <= 0;
         harmonic_index_out <= harmonic_index_out + 1;
      end
      else
```

```
          note <= note + 1;

         // reached the end of data sending
         if (current_out + 1 == TOTAL_NOTES) swg_data_out <= 0;
         current_out <= current_out + 1;
      end

      if (swg_data_in) begin
         if (current_in + 1 == NUM_NOTES) begin
            // advance to the next harmonic
            harmonic_index_in <= harmonic_index_in + 1;
            current_in <= 0;
         end
         else
           current_in <= current_in + 1;

         // multiply by the correct harmonic scale factor
         case(harmonic_index_in)
           0: sum <= sum + (notes[current_in] ? (amp * h_zero) : 0);
           1: sum <= sum + (notes[current_in] ? (amp * h_one) : 0);
           2: sum <= sum + (notes[current_in] ? (amp * h_two) : 0);
           3: sum <= sum + (notes[current_in] ? (amp * h_three) : 0);
           4: sum <= sum + (notes[current_in] ? (amp * h_four) : 0);
           default: sum <= sum;
         endcase
      end
   end
endmodule // main

module harmonic_frequencies(freq_in, harmonic_num, freq_out);
   input [11:0]  freq_in;
   input [2:0]   harmonic_num;
   output [11:0] freq_out;
   reg [11:0]    freq_out;

   // these are optimized forms of the naive multiplies
   always @ * begin
      case (harmonic_num)
        0: freq_out = freq_in; // (1 * freq_in)
        1: freq_out = freq_in << 1; // (2 * freq_in)
        2: freq_out = freq_in + (freq_in << 1); // (3 * freq_in)
        3: freq_out = freq_in << 2; // (4 * freq_in)
```

```
        4: freq_out = freq_in + (freq_in << 2); // (5 * freq_in)
        default: freq_out = freq_in;
      endcase
   end
endmodule

// 0 = Gb2, everything else is half steps
module note_frequencies(note_index, frequency);
   input  [5:0]  note_index;
   output [11:0] frequency;
   reg [11:0]          frequency;

   // big switch statement
   always @ * begin
      case (note_index[5:0])
        0: frequency <= 93;
        1: frequency <= 98;
        2: frequency <= 104;
        3: frequency <= 110;
        4: frequency <= 117;
        5: frequency <= 123;
        6: frequency <= 131;
        7: frequency <= 139;
        8: frequency <= 147;
        9: frequency <= 156;
        10: frequency <= 165;
        11: frequency <= 175;
        12: frequency <= 185;
        13: frequency <= 196;
        14: frequency <= 208;
        15: frequency <= 220;
        16: frequency <= 233;
        17: frequency <= 247;
        18: frequency <= 262;
        19: frequency <= 277;
        20: frequency <= 294;
        21: frequency <= 311;
        22: frequency <= 330;
        23: frequency <= 349;
        24: frequency <= 370;
        25: frequency <= 392;
        26: frequency <= 415;
```

```
        27: frequency <= 440;
        28: frequency <= 466;
        29: frequency <= 494;
        30: frequency <= 523;
        31: frequency <= 554;
        32: frequency <= 587;
        33: frequency <= 622;
        34: frequency <= 659;
        35: frequency <= 698;
        36: frequency <= 740;
        37: frequency <= 784;
        default: frequency <= 64;
      endcase
   end
endmodule
```

A.2 PS/2 Keyboard Interface

```
//
// File:   ps2_kbd.v
// Date:   24-Oct-05
// Author: C. Terman / I. Chuang
//
// PS2 keyboard input for 6.111 labkit
//
// INPUTS:
//
//   clock_27mhz   - master clock
//   reset         - active high
//   clock         - ps2 interface clock
//   data          - ps2 interface data
//
// OUTPUTS:
//
//   ascii         - 8 bit ascii code for current character
//   ascii_ready   - one clock cycle pulse indicating new char received


///////////////////////////////////////////////////////////////////////////////

module ps2_ascii_input(clock_27mhz, reset, clock, data, notes_ready,
notes);
```

```
// module to generate ascii code for keyboard input
// this is module works synchronously with the system clock

input clock_27mhz;
input reset;          // Active high asynchronous reset
input clock;          // PS/2 clock
input data;           // PS/2 data
output notes_ready;   // notes ready (one clock_27mhz cycle active high)
output [37:0] notes;  // notes of the musical scales

reg [7:0]   lastkey;  // last keycode
reg [7:0]   curkey;   // current keycode
reg notes_ready;      // synchronous one-cycle ready flag

reg [37:0]  notes;    // the 38 notes of the scale
reg  brk = 0;         // logic for whether or not a key is being released

// get keycodes

wire        fifo_rd;         // keyboard read request
wire [7:0]  fifo_data;       // keyboard data
wire        fifo_empty;      // flag: no keyboard data
wire        fifo_overflow;   // keyboard data overflow

ps2 myps2(reset, clock_27mhz, clock, data, fifo_rd, fifo_data,
          fifo_empty,fifo_overflow);

assign fifo_rd = ~fifo_empty;        // continous read
reg    key_ready;

always @(posedge clock_27mhz) begin
   // get key if ready
   curkey <= ~fifo_empty ? fifo_data : curkey;
   lastkey <= ~fifo_empty ? curkey : lastkey;
   key_ready  <= ~fifo_empty;

   // raise notes_ready for last key which was read

   notes_ready <= key_ready & ~(curkey[7]|lastkey[7]);

   case (curkey)
   8'h1C: begin notes[0] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
```

```
8'h1A: begin notes[1] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h1B: begin notes[2] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h22: begin notes[3] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h23: begin notes[4] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h21: begin notes[5] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end

8'h2A: begin notes[6] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h34: begin notes[7] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h32: begin notes[8] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h33: begin notes[9] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h31: begin notes[10] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end

8'h3A: begin notes[11] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h42: begin notes[12] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h41: begin notes[13] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h4B: begin notes[14] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h49: begin notes[15] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h4C: begin notes[16] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h4A: begin notes[17] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end

8'h15: begin notes[18] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h1E: begin notes[19] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h1D: begin notes[20] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h26: begin notes[21] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h24: begin notes[22] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h2D: begin notes[23] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h2E: begin notes[24] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h2C: begin notes[25] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h36: begin notes[26] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h35: begin notes[27] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h3D: begin notes[28] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h3C: begin notes[29] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end

8'h43: begin notes[30] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h46: begin notes[31] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h44: begin notes[32] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h45: begin notes[33] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h4D: begin notes[34] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end

8'h54: begin notes[35] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h55: begin notes[36] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
8'h5B: begin notes[37] <= brk ? 0 : 1; brk <= (lastkey == 8'hF0) ? brk : 0; end
```

```
      8'hF0: brk <= 1;

      default: begin notes <= notes; brk <= 0; end
     endcase // case(curkey)
   end // always @ (posedge clock_27mhz)
endmodule // ps2toascii

////////////////////////////////////////////////////////////////////////////
// new synchronous ps2 keyboard driver, with built-in fifo, from Chris Terman

module ps2(reset, clock_27mhz, ps2c, ps2d, fifo_rd, fifo_data,
           fifo_empty,fifo_overflow);

   input clock_27mhz,reset;
   input ps2c;                        // ps2 clock
   input ps2d;                        // ps2 data
   input fifo_rd;                // fifo read request (active high)
   output [7:0] fifo_data;        // fifo data output
   output        fifo_empty;        // fifo empty (active high)
   output        fifo_overflow;        // fifo overflow - too much kbd input

  reg [3:0] count;      // count incoming data bits
  reg [9:0] shift;      // accumulate incoming data bits

  reg [7:0] fifo[7:0];   // 8 element data fifo
  reg fifo_overflow;
  reg [2:0] wptr,rptr;   // fifo write and read pointers

  wire [2:0] wptr_inc = wptr + 1;

  assign fifo_empty = (wptr == rptr);
  assign fifo_data = fifo[rptr];

  // synchronize PS2 clock to local clock and look for falling edge
  reg [2:0] ps2c_sync;
  always @ (posedge clock_27mhz) ps2c_sync <= {ps2c_sync[1:0],ps2c};
  wire sample = ps2c_sync[2] & ~ps2c_sync[1];

  always @ (posedge clock_27mhz) begin
    if (reset) begin
      count <= 0;
```

```
      wptr <= 0;
      rptr <= 0;
      fifo_overflow <= 0;
    end
    else if (sample) begin
           // order of arrival: 0,8 bits of data (LSB first),odd parity,1
           if (count==10) begin
              // just received what should be the stop bit
              if (shift[0]==0 && ps2d==1 && (^shift[9:1])==1) begin
                 fifo[wptr] <= shift[8:1];
                 wptr <= wptr_inc;
                 fifo_overflow <= fifo_overflow | (wptr_inc == rptr);
              end
              count <= 0;
           end else begin
              shift <= {ps2d,shift[9:1]};
              count <= count + 1;
           end
         end
    // bump read pointer if we're done with current value.
    // Read also resets the overflow indicator
    if (fifo_rd && !fifo_empty) begin
      rptr <= rptr + 1;
      fifo_overflow <= 0;
    end
  end
endmodule
```

A.3 Sine Wave Generator

```
module swg(clock_27mhz, rdy, frequency, cycle, amplitude, dout);
   input clock_27mhz;
   input rdy;
   input [11:0] frequency; // frequency of desired wave: min freq = 64hz
   input [15:0] cycle; // time (ie phase)

   output signed [15:0] amplitude; // amplitude of generated wave
   output          dout;

   reg [25:0]         rdy_prev;

   reg [27:0]          fc_product;
   wire [21:0]          quotient;
```

```
   wire [9:0]          addr; // remainder
   wire          rfd;  // ready for data

   always @ (posedge clock_27mhz) begin
      rdy_prev <= {rdy_prev[24:0], rdy};

      if (rdy) begin
         fc_product <= frequency * cycle;
      end
   end

   swgdiv my_swgdiv(fc_product[27:6], 10'h2EE, quotient, addr,
                 clock_27mhz, rfd, 1'b0, 1'b0, 1'b1);

   wire we = 0;
   wire [15:0]  din;

   // table lookup here using addr_lookup to get amplitude.
   // is the bram synchronous?
   bram_signed_750x16 swg_ram(addr, clock_27mhz, din, amplitude, we);

   assign dout = rdy_prev[25];
endmodule
```

A.4 Instrument Bank

```
// instrument bank takes in 2-bit instrument selector
// and returns 3 coefficients for the relative intensities
// of the base, first, and second harmonics.
module instbank (inst_sel, h_zero, h_one, h_two, h_three, h_four);
   input [1:0] inst_sel; // 2-bit instrument selector

   // the 5 harmonic content coefficients
   output [8:0] h_zero, h_one, h_two, h_three, h_four;

   reg signed [8:0] h_zero, h_one, h_two, h_three, h_four;

   always @ (inst_sel) begin
      case(inst_sel)
        2'b00: begin
           h_zero <= 9'h0FF;
           h_one <= 9'h0;
           h_two <= 9'h0;
```

```
            h_three <= 9'h0;
            h_four <= 9'h0;
         end // pure tone has no higher harmonics
         2'b01: begin
            h_zero <= 9'h064;
            h_one <= 9'h063;
            h_two <= 9'h01F;
            h_three <= 9'h19;
            h_four <= 9'h0;
         end // guitar has 100:60:40:25:30 harmonics ratio, in theory,
             // but it doesn't really sound like it.
         2'b10: begin
            h_zero <= 9'h0CC;
            h_one <= 9'h0;
            h_two <= 9'h033;
            h_three <= 9'h0;
            h_four <= 9'h0;
         end // xylophone has one higher harmonic. wooooo!
         2'b11: begin
            h_zero <= 9'h033;
            h_one <= 9'h033;
            h_two <= 9'h033;
            h_three <= 9'h033;
            h_four <= 9'h033;
         end // This sounds like an oboe.
             // It was supposed to be a string instrument though. o_O;;
      endcase
   end
endmodule
```

A.5 Mixer

```
// The mixer was shamelessly copied from lab 4.
module mixer (clock_27mhz, reset, volume,
              audio_in_data, audio_out_data, ready,
              audio_reset_b, ac97_sdata_out, ac97_sdata_in,
              ac97_synch, ac97_bit_clock);

   input        clock_27mhz;
   input        reset;
   input [4:0]  volume;
   output[19:0] audio_in_data;
   input [19:0] audio_out_data;
```

```
output       ready;

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

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

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

reg          audio_reset_b;
reg [9:0]    reset_count;

//wait a little before enabling the AC97 codec
always @(posedge clock_27mhz) begin
   if (reset) begin
      audio_reset_b = 1'b0;
      reset_count = 0;
   end else if (reset_count == 1023)
     audio_reset_b = 1'b1;
   else
     reset_count = reset_count+1;
end

wire ac97_ready;
ac97 ac97(ac97_ready, command_address, command_data, command_valid,
          left_out_data, 1'b1, right_out_data, 1'b1, left_in_data,
          right_in_data, ac97_sdata_out, ac97_sdata_in, ac97_synch,
          ac97_bit_clock);

// 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;
   always @ (posedge clock_27mhz)
     if (ready) out_data <= audio_out_data;
   assign     audio_in_data = left_in_data;
   assign     left_out_data = out_data;
   assign     right_out_data = left_out_data;

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

// this module serializes the data
module ac97 (ready,
             command_address, command_data, command_valid,
             left_data, left_valid,
             right_data, right_valid,
             left_in_data, right_in_data,
             ac97_sdata_out, ac97_sdata_in, ac97_synch, ac97_bit_clock);

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

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

   reg ready;

   reg ac97_sdata_out;
   reg ac97_synch;

   reg [7:0] bit_count;
```

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

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

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

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

// this module issues initialization commands to AC97
module ac97commands (clock, ready, command_address, command_data,
                     command_valid, volume, source);

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

   reg [23:0] command;
   reg command_valid;

   reg [3:0] state;

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

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

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

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

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

A.6 Echo Module

```
module echo(clock_27mhz, enable, signal_in, signal_out, scale, ready);
   input clock_27mhz;
   input enable;
```

```
input signed [19:0]  signal_in;

output [19:0] signal_out;  // output signal: modified with added
                           // echo if echo module is enabled
reg signed [19:0]  signal_out;

input signed [7:0] scale;
input ready;

reg [16:0] address=0;
wire [19:0] bram_out;

reg        we;
reg[5:0]   we_old;

wire signed [18:0] s_down;

wire [19:0] bram_data;
reg [19:0]  latched_signal_in;

// parameter that determines how much lag the echo has
// if parameter is lowered, echo lag time decreases
// 96000 = 2 seconds
parameter MAX_BUFFER_INDEX = 96000;

assign    bram_out = bram_data;
assign    s_down = bram_out[19:1]; // scale down by half

always @ (posedge clock_27mhz) begin
   we_old <= {we_old[4:0], we};

   if (ready) begin
      latched_signal_in <= signal_in;
       signal_out <= signal_in + (enable ? s_down : 0);
      we <= 1;
   end
   if (we_old[5]) begin
      we <= 0;
   end
   if (we_old[5] & ~we) begin // falling edge of write enable
      // if address is about to hit the maximum buffer size
      // that is, the maximum number of stored elements with which
```

```
      // to generate the echo, loop address back around, start over.
      address <= (((address + 1) == MAX_BUFFER_INDEX) ? 0 : (address + 1));
    end // falling edge of write enable
  end // always @ (posedge clock_27mhz)

  bram_more_echo bme(address, clock_27mhz, latched_signal_in, bram_data, we);
endmodule
```