Embedded Systems Design Project
Final Report

Real-Time Guitar Effects Sampler

Navarun Jagatpal - Fred Rassam - Young Jin Yoon - Elton Chung

May 8th 2007

TABLE OF CONTENTS

Introduction

It is common for electric guitar players to make use of 'pedals' in order to achieve certain effects such as 'distortion/overdrive', 'delay/echo', 'chorus', 'flanger' and 'pitch/phase shifter'. Typically these pedals make use of analog electronics to yield the desired effects.

The objective of our project was to implement such effects ('distortion', 'chorus' and 'vibrato') in hardware digitally via the Altera DE2 FPGA board. Sampling is therefore performed in real-time from a signal provided by a vaccum tube preamp (ART MP Studio – using a 12AX7 tube) to bring the guitar's output to line-level. From there, the user is able to select the desired effect, along with the relevant parameters (volume, frequency, etc.), on the fly and have the FPGA output the processed stream to a set of speakers or headphones.

This project involves the knowledge and use of the onboard audio codec, Digital Signal Processing concepts, simulation and timing analysis along with embedding software for control of the system. Furthermore, the algorithms in question were first designed in MatLab, then coded in 'C' and finally ported to VHDL which facilitates the process of debugging.

Throughout the development of this system, various design challenges were encountered which altered our objectives. At first, we were considering having a feature where the input or output stream could be dumped onto an SD card. This was dropped to accommodate the implemention of the core features correctly in the allotted time. A 'normalizing' feature was also considered as was automatic volume control (gain control) in lieu of it yet these features were left out of the final prototype which is covered in this report.

System Overview

The entire system shown below can be broken down into three basic components: audio CODEC, DSP and software control. The audio component consists of two L/R Sampling Buffers, one for input and one for output. The input buffer converts an ADC input stream into a 32 bit sample buffer and outputting it to the DSP modules as two 16 bit sampled buffers, one for each channel. The output buffer does the opposite and re-streams the sample buffers back to the DAC. There is also an Audio in and an Audio out module which tells the L/R buffers when data is ready to be received and when data should be outputted. The DSP components, Distortion, Vibrato and Chorus (which will be covered in the next section) are connected serially to allow for simultaneous effects. Each component is separated into two to process both left and right channels. Input values for each effect (delay amplitude, frequency, mix value, clip value) are collected through keyboard control using the Avalon bus and software control.

The system also allows different effects to be applied independently to either the left or the right channel. While this does not change anything to our test setup – the preamp outputs only to the left channel – this feature allows two guitars to be connected to the board each having different effects applied to them. Naturally, the same effects could be used for both the left and right channel while different parameters could be selected.

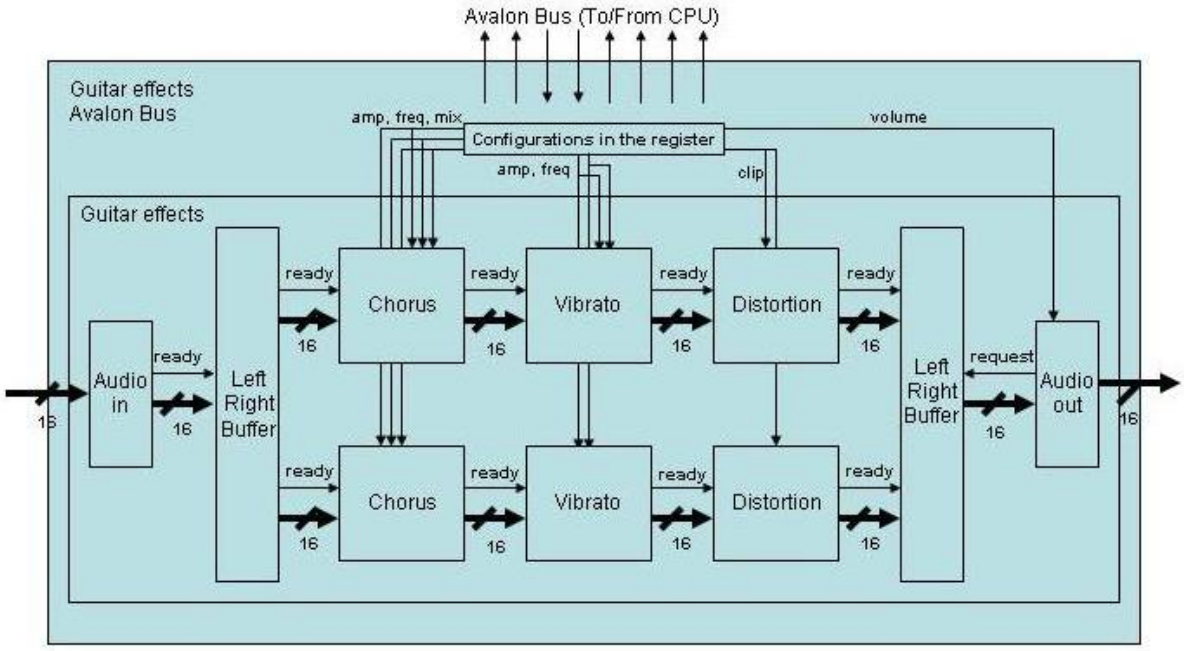

Fig1: Top level diagram

Audio CODEC

The ADC (Analog-to-Digital Converter) and DAC (Digital-to-Analog Converter) elements were handled by the WOLFSON WM8731 which resides on the DE2 board. Fortunately, the Verilog code to implement these functions was provided by the 'lab3' assignment. Thus, our responsibilities for this component involved configuring it appropriately (via what is known as to I2C settings) in addition to writing a buffer so that the DSP portion could effectively process the stream.

The final settings that were used for this component were 16-bit, 48kHz sampling rate with a sync to the CPU clock and input configured for line-level input. Sampling rate is often seen as a tradeoff between quality and 'performance' (where greater sampling rates call for more data throughput and CPU power); based on our source, it was determined that 16-bit / 48kHz was more than enough for our purposes.

As previously mentioned this component was a critical path to the rest

where the input had to be fed to the output to check if it was working before it could be fed anywhere else. It was imperative to have this component working first to debug the others using a real, non-simulated, input. Moreover, the settings chosen for this component affect the rest of the design as a sine table had to be built for the DSP portion and the I2C configuration had to be fed values provided by software via the Avalon bus.

Digital Signal Processing

The following block diagrams shown below for each of the three effects are the models that were first implemented in Matlab, then 'C' and later ported to VHDL. Distortion was the first to be implemented followed by Vibrato and Chorus which were significantly more complicated. As chorus is only a slight modification of vibrato; the alteration was done directly in VHDL without having to write a new model in Matlab or 'C.'

For both Chorus and Vibrato, it was necessary to have sine table as the output signal has a different frequency due to the variable delay. For reasons that are later discussed, our sine table has 1500 samples to represent the sine wave up to pi/2. Thus, interpolation is used to represent the entire periodic function when needed. Frequency changes are achieved by skipping these samples during playback; the more samples skipped, the higher the frequency.

Distortion

The distortion effect takes a normal signal and distorts the waveform by "clipping" the signal. In other words, the effect truncates the upper and lower portion of the signal from the clip value. The algorithm used for this effect is shown in the following block diagram:

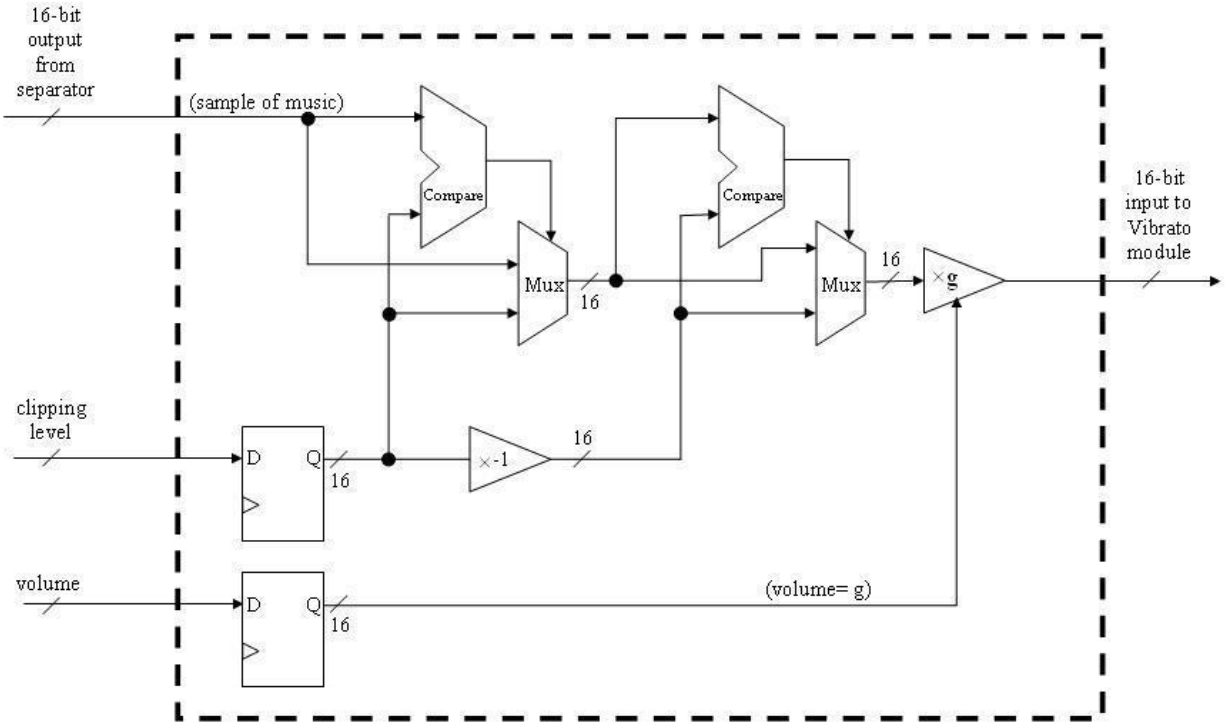

Fig2. Distortion module block diagram

The two comparators shown compare the upper bound and the lower bound of the signal with the clipping level. If the input exceeds the bounds, then it will set the value of the signal equal to the clip value. The volume control on the lower left is used as the gain control for the clipped signal. A clock diagram of the distortion module is shown below:

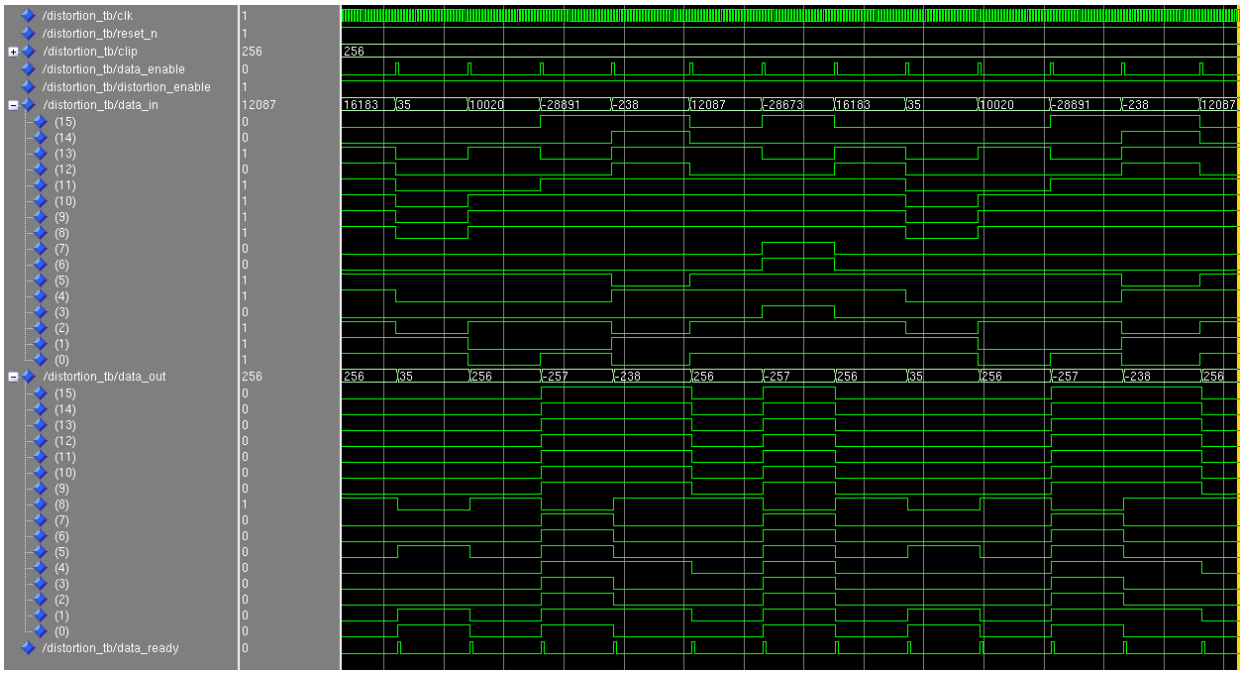

Fig 3. Distortion timing diagram

The clock diagram shows an instance in time of the distortion module. The clip value is an input, and in this instance is set to 256. The data_enable is set every time there is input data, while the data_ready is set everytime there is output data. The input buffer is labeled as data_in and the output buffer is data_out. As you can see, if the input value is over the value of +/- 256 the output value will be clipped at +/- 256.

The VHDL implementation of the distortion module is attached as Appendix A.

Vibrato

Vibrato is an effect created by a regular pulsating change of pitch at around 2-6 Hz and is used to add expression and vocal like quality to instrumental music. This is accomplished by a short variable delay modulated by a sine wave. The algorithm used is represented by the block diagram below:

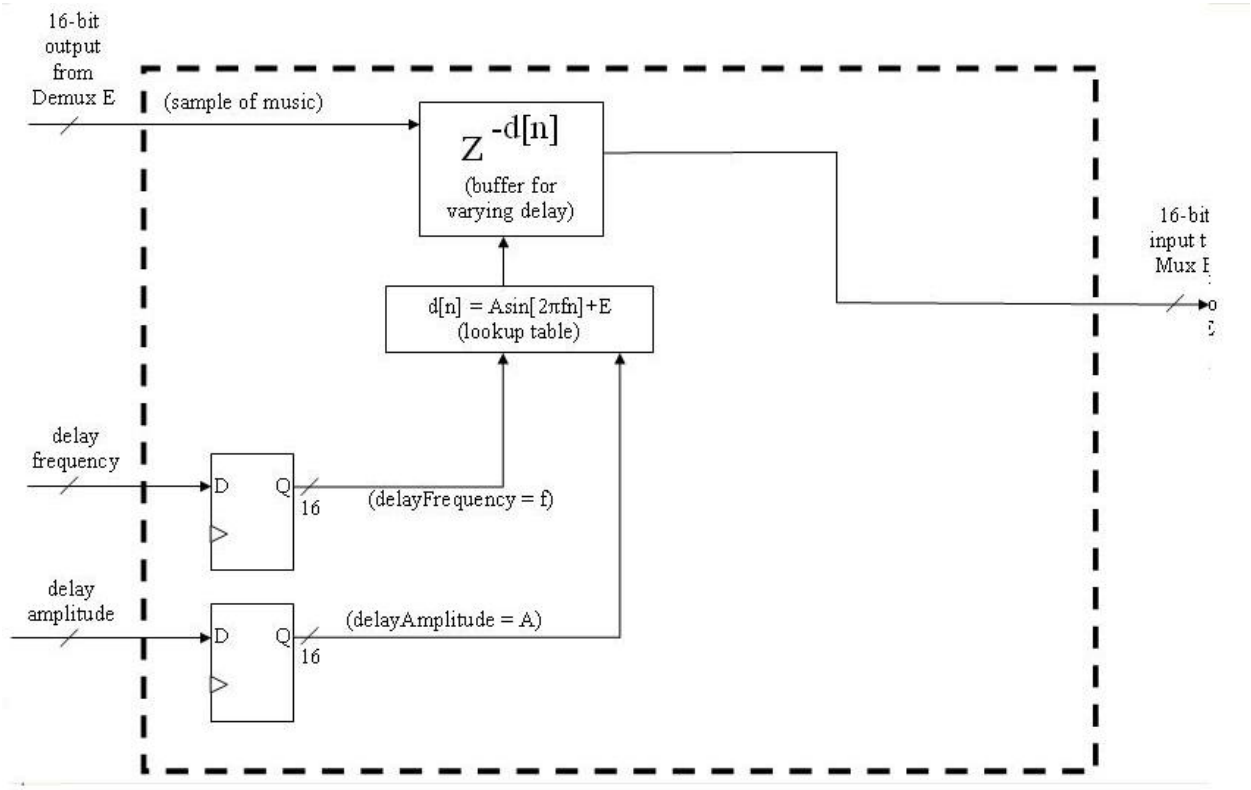

Fig 4: Vibrato Block Diagram

The sine modulation formula *Asin[2πfn]* was calculated via matlab and inserted into VHDL as a lookup table. The delay amplitude is multiplied with the result of the sine lookup table, while the frequency dictates the rate in which the table is referenced for the output buffer.
A clock diagram of the vibrato module is shown below:

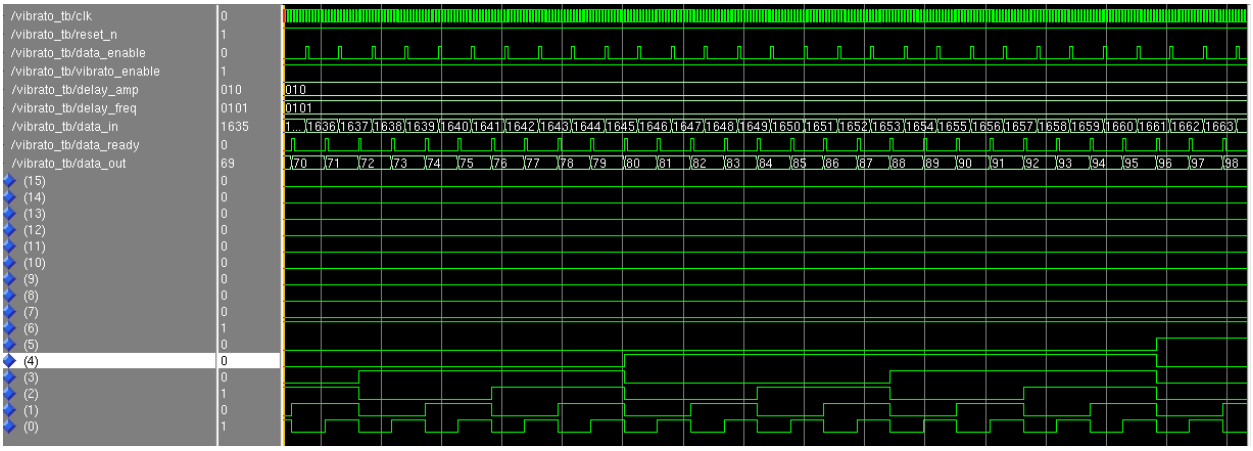

Fig5. Vibrato timing diagram

The clock diagram shows an instance in time of the vibrato module. Vibrato takes amplitude and frequency as an input values. In this instance, they are set static to a given value. Similar to distortion, there is a data_enable and a data_ready which. The data_in buffer goes through a variable delay algorithm before being exported to data_out.

The VHDL implementation of vibrato is attached as Appendix B.

Chorus

Chorus is similar to vibrato in which it is also a delay. However, this delay or multiple delay, are much slower than it is in vibrato. These delays are then blended together to form the effect, resulting in a sound where several guitarists can be heard playing at the same time. The algorithm of chorus is shown in the block diagram below:

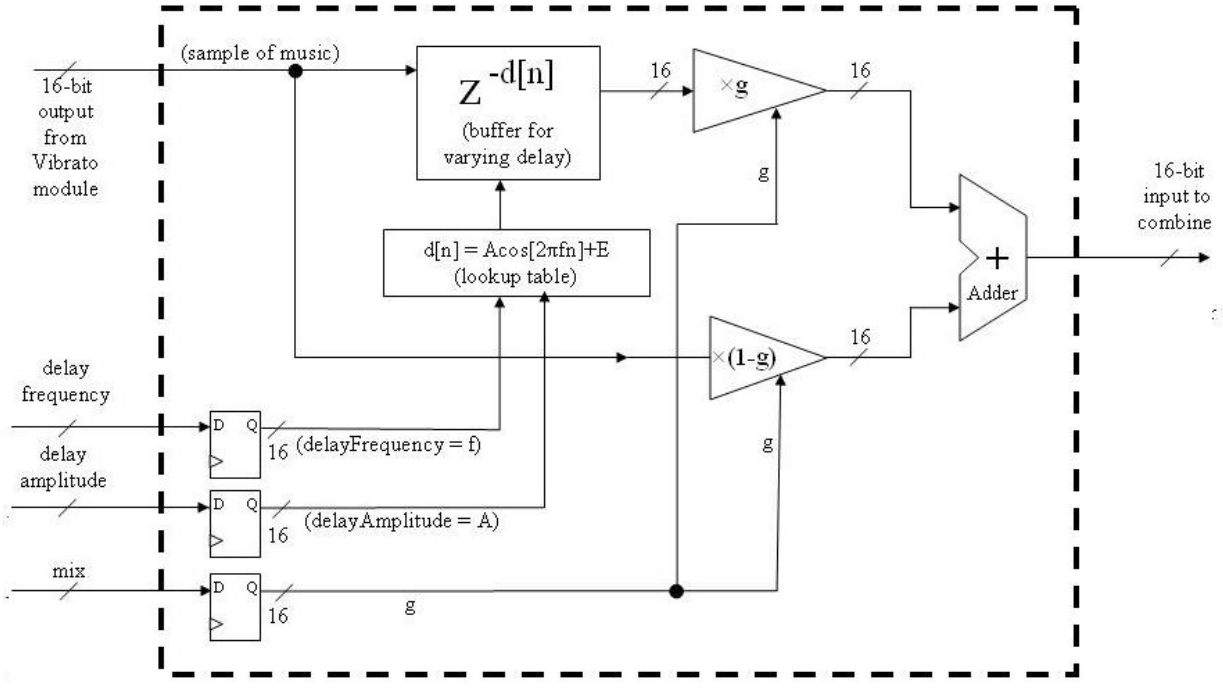

Fig 6. Chorus Block Diagram

The algorithm is very similar to the vibrato algorithm in that it uses the same sine look up table as well as the same methods of using the delay amplitude and frequency. However, chorus is a blend of input signals and as the block diagram shows, we blend the original signal with the delayed signal to produce the desired effect. The chorus module also has a "mix" input where the amount of each blend can be controlled.

A clock diagram of chorus is shown below:

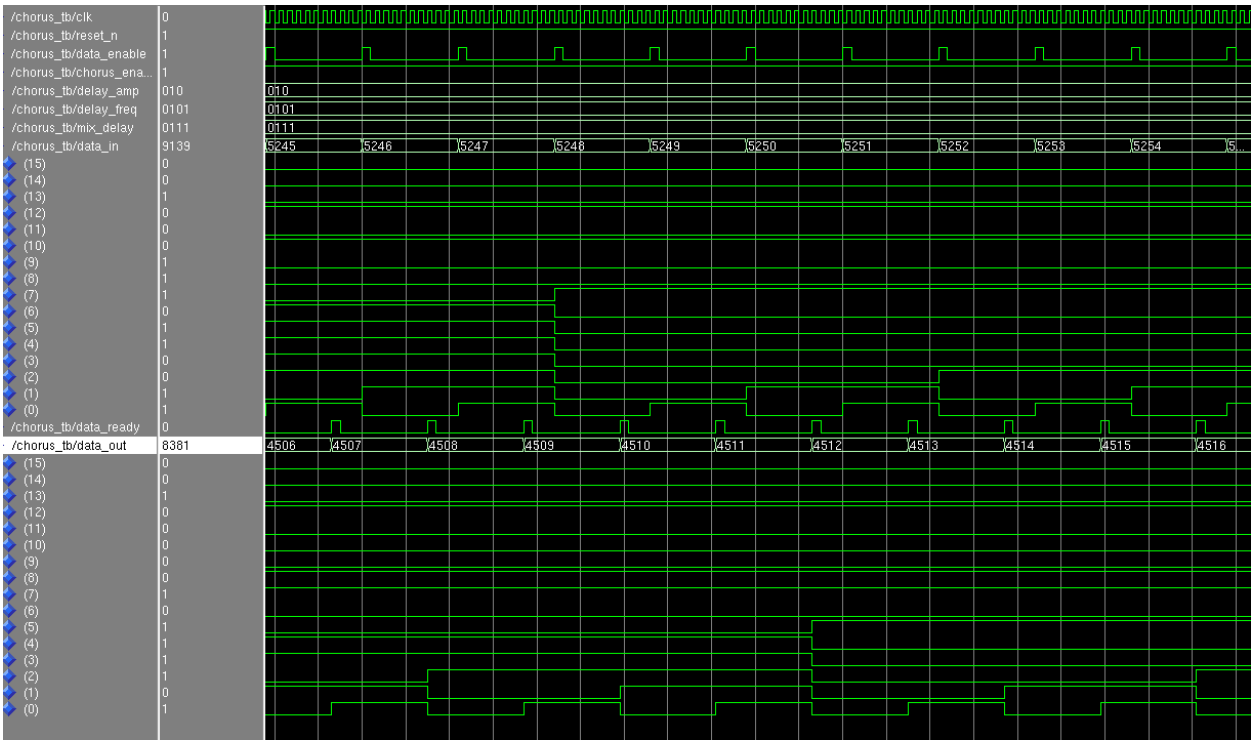

Fig 7. Chorus timing diagram

The clock diagram shows an instance in time of the chorus module. Chorus takes amplitude, frequency and a mix_delay as inputs. This mix_delay is the percentage of variable delay mixed in with the input to produce the chorus effect. Similar to distortion and vibrato, there is a data_enable and data_ready. The data_in buffer goes through a variable delay algorithm and then gets mixed back in with the input signal before being exported to the data_out.

The VHDL implementation of chorus is attached as Appendix C.

Software Control

While this system is fully functional in hardware and doesn't require any software for its intended operation, we decided to implement the switching of the effects in software instead of leaving them in hardware. Therefore, we decided to take in inputs from the PS/2 keyboard instead of using the onboard buttons to switch between effects. The availability of extra buttons gave us the flexibility to alter parameters such as: volume, frequency (chorus & vibrato), delay amplitude (chorus & vibrato) and a mix ratio (between delayed and original signal) for chorus.

Another advantage of using software was that we were able to print to the onboard LCD output with much greater ease. This works well to display the current volume and frequency values, for both the left and right channel, which are altered sequentially from the keyboard.

Procedure: Experiences & Issues

As our first step involved getting the audio CODEC working, there were a few issues we had to overcome in order to enable the use of this component. Firstly, the plan was to use the electric guitar directly as an input which warranted high-gain settings. Nonetheless, the output signal from the guitar must have been far too low for the ADC to pick up anything. Therefore, we had to switch to line-level input and use the preamp which provided an output strong enough – both inputs and outputs on the preamp are adjustable – to be picked up clearly by the ADC. To minimize the level of noise, gain was kept low in the I2C settings while the levels on the amp could be adjusted.

Other changes that were made to the I2C configuration were the sampling rate and the clock settings – we found it easier to sync this component to the CPU's clock rather than to have it the other way around. This is why we are running this component in 'slave' mode. Either way, it was necessary to write a buffer large enough for the effect processes to alter the stream effectively as they operate on many bits at once. Consequently, the size of the buffer had to be changed.

Perhaps the greatest change in our approach to designing this system

was forgoing the, very tempting, option of using the DSP builder which generates the VHDL code from block diagrams we can assemble from within the software package. The reasons for parting with this was that there were licensing issues with the DSP builder which also had to be run from a Windows PC instead of the Linux-based machines in the lab. Moreover, use of the DSP builder is said to not be trivial to the point where it requires no training. Thus, it was determined that it would be better to follow the more 'classical' approach of implementing the algorithms in 'C' and then porting them in VHDL. In the event that the software implementation worked with unresolvable issues remaining with the hardware implementation, we could always try to have the DSP tasks done in software even if that would mean a delayed, non-realtime, operation.

Another fundamental design change was the switch from a mux-based implementation (shown below) to the serial daisy-chained effected introduced earlier. This stems from the fact that we weren't able to get the mux working in the first place. Yet, it was determined that it was also more advantageous in terms of features to use the latter implementation for the components. Not only could effects be applied simultaneously but channels were independent of each other. In our system, we only use two channels but this model could very well scale to include many channels each having different effects and parameters applied. In other words, having to find an alternative for a broken mux was more a blessing than anything.

Some other issues we ran into were, for example, not being able to write a 'normalize' module to bring the outputs to level regardless of how low the input level could be. As we weren't able to overcome the challenges we faced in this attempt, we devised an alternative way to achieve a similar effect – automatic volume control based on the current reading for volume. This feedback mechanism is often referred to as Automatic Gain Control (AGC). As we had implemented volume control via keyboard in software, the idea was to have a function in the 'C' program that would provide this at the user's behest. Although we were confident in being able to deliver this feature, we had reached the point where it was already too late to start developing new features.

Concerning the code itself, one problem was the lack of the ‘mod’ function in VHDL. This was an issue as, in order to make sure the buffer is never exceeded, ‘sample number’ is moded by ‘buffersize’ which results in the delay used by both chorus and vibrato effects. Thus, a workaround using simple subtraction was used instead.

We also had trouble when we wanted to use a larger sine table. Firstly, it wasn’t possible to copy-paste it into Altera’s Quartus software as doing so would cause the machine to hang when the table was substantial for our purposes. Yet, even when editting the VHDL using different software there were issues concerning its size as it had to fit on the board’s SRAM. Fortunately, as sine is a periodic function, it is possible to represent the entire period (2pi) by just storing the information up to pi/2. This is because the remaining data is either a repeat or inverse (negative) of these initial values.

Finally, one issue we did ran into at the very end was concerning the PS/2 keyboard module we had planned to use from ‘lab2’ and ‘lab3’. There was apparently a bug in the packaging that did not allow us to retrieve the .vhd file as needed, so, we had to write this module ourselves after that was discovered. Fortunately, this obstacle was far easier to overcome than the rest.

Conclusion

The overall system design and development was a success as evidenced by its flawless operation. Yet, as with any initiative, successful or not, much can be said about the lessons that were learnt and how the design and development process was tackled by the group. Below are descriptions covering what was learnt throughout the process along with what can be said about how the group worked together as a team.

General Issues

Perhaps the greatest help in debugging the DSP modules was provided by running testbenches with the simulator. Not only was this faster than re-compiling the code and flashing it onto the board, but this is how we were able to obtain the timing diagrams show above for each of the effects. The use of the simulator was non-trivial yet it does prove to be a valuable asset that every group member learnt.

Similarly, as the bulk of our time was spent on implementing the algorithms, the prospect of having used the DSP builder remains tempting despite the challenges associate with it. This is because doing so would have allowed us to concentrate our programming time for other features and end up with a device that demonstrates greater novelty. Although we did not try to find C or VHDL for the effects chosen, it is very likely that we could have used such material either from a book, the web or a previous group who had a similar project.

Getting the audio CODEC working was what we referred to as the 'articulation point.' Being a critical path to the rest of the system, it was very frustrating until that point as we felt that all our efforts would be worthless if that component did not work. Nevertheless, despite the time it took, we were quite confident that this would not be an issue as the code for the codec had been used before successfully; it was just a matter of configuring it correctly.

Group Dynamics

As three out of four of us are graduate students, we each had significant experiences working in group projects before. Typically, the model which is followed is that tasks are assigned to each group member and they report back demonstrating the progress they achieved with the task that was assigned to them. While this method may have its merits, it is often not optimal particularly when some group members are significantly more skilled at certain tasks than others. Also, as all development was done in the lab, it made more sense to work on tasks together. The approach we took is sometimes referred to as the 'Extreme Programming Methodology' where one person (Young) would focus on the code while the others would consistently provide feedback, hence debug it, in real-time.

There were nonetheless, tasks that team members were delegated and focused on. As Navarun was taking a class in 'Music Signal Processing', he was responsible for teaching Young the algorithms for the effects and how to implement them. Meanwhile, Elton and Fred focused on getting the audio CODEC to work correctly. Fred and Navarun had also put together the SOPC (system-on-a-programmable-chip) although Young had to write the bus and the software to control it. This was also the case with the buffer that Elton wrote. Hence, it was determined that helping Young accomplish what he needed was significantly more efficient than writing a portion he would later have to fix.

References

Eric Newman & Steven Bondadio, “Variable Gain Amplifiers Enable Cost Effective IF Sampling Receiver Designs” *Analog Devices* (http://www.mpdigest.com/Articles/2003/Oct2003/Analog/Default.htm)

Wolfson Electronics “Portable Internet Audio CODEC with headphone drivers and Programmable Sample Rates”

Wikipedia “Guitar Effects” (http://en.wikipedia.org/wiki/Guitar_effects

Spin Semiconductors “Audio Effects” (http://www.microtips.com/spin/effect.htm)

Appendix

Appendix A

Distortion VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity distortion is
port (
   clk : in std_logic;
   reset_n : in std_logic;
   data_enable : in std_logic;
   distortion_enable : in std_logic;

   data_in : in std_logic_vector(15 downto 0);
   data_out : out std_logic_vector(15 downto 0);
   clip : in std_logic_vector(15 downto 0);
   data_ready : out std_logic

);
end distortion;

architecture rtl of distortion is
-- signal data_in_buf : std_logic_vector(15 downto 0);
-- signal data_out_buf : std_logic_vector(15 downto 0);
   signal cut_value : std_logic_vector(15 downto 0);
-- constant cut_value := X"3FFF";

begin

   cut_value <= clip;

   process(clk, reset_n)
   begin
   if(reset_n = '0') then
      data_out <= (others => '0');
   elsif ( clk'event and clk = '1') then
      if(data_enable = '1') then
         if(distortion_enable = '1') then
            if(data_in(15) = '1') then
               if((not data_in(14 downto 0)) > cut_value(14 downto 0)) then
                  data_out <= '1' & (not cut_value(14 downto 0));
               else
                  data_out <= data_in;
               end if;
            else
               if(data_in(14 downto 0) > cut_value(14 downto 0)) then
                  data_out <= '0' & cut_value(14 downto 0);
               else
```

```
                  data_out <= data_in;
               end if;
            end if;
         else
            data_out <= data_in;
         end if;
         data_ready <= '1';
      else
         data_ready <= '0';
      end if;
   end if;
   end process;

end architecture;
```

Appendix B

Vibrato VHDL Code

```
------------------------------------------------------------------
-- Vibrato module
--
-- Made by Navarun Jagatpal, April 10, 2007
-- Modifyed by Young Jin Yoon, April 12, 2007
-- Will be debugged by everybody....yeah~!
------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity vibrato is
port (
       --In
       clk: in std_logic;      --Should be 50 MHz
      reset_n: in std_logic;
       data_enable: in std_logic;
       vibrato_enable: in std_logic;
      delay_amp: in std_logic_vector(2 downto 0);
      delay_freq: in std_logic_vector(3 downto 0);
       data_in: in std_logic_vector(15 downto 0);

       --Out
      data_ready: out std_logic;
       data_out: out std_logic_vector(15 downto 0)
       );
end vibrato;

architecture rtl of vibrato is

   signal delta_delay: std_logic_vector(2 downto 0);
   signal freq: std_logic_vector(3 downto 0):= "0000";

   signal freq_cnt: std_logic_vector(3 downto 0):= "0000";

   signal delay: std_logic_vector(11 downto 0);

   signal data_internal : std_logic_vector (15 downto 0);

   signal buffer_in: std_logic_vector(11 downto 0);
   signal buffer_out: std_logic_vector(11 downto 0);

   signal ram_re: std_logic;
   signal ram_we: std_logic;
   signal ram_addr: std_logic_vector(10 downto 0);
   signal ram_data_out: std_logic_vector(15 downto 0);
   signal ram_data_in: std_logic_vector(15 downto 0);
```

```
   signal state: std_logic_vector(3 downto 0) := X"0";

   signal sine_data: std_logic_vector(11 downto 0);
   signal sine_addr: std_logic_vector(12 downto 0):= "0000000000000";
   signal sine_request: std_logic;

   signal sine_amplitude: std_logic_vector(11 downto 0);

    component RAM_1633_16 is
port (
               --In:
               clk : in std_logic;
               we   : in std_logic; -- Write-enable
               re   : in std_logic; -- Read-enable
               addr : in std_logic_vector(10 downto 0); --Address: 11 bits.
This allows 0 to 2047, but the only valid

            --inout:
--addresses range from 0 to 1632.
               data_out : out std_logic_vector(15 downto 0); --Data In: 16
bits
            data_in  : in std_logic_vector(15 downto 0)
       );
   end component;

   component sine_table is
   port (
      clk : in std_logic;
      reset_n : in std_logic;
      data_request : in std_logic;
      amplitude : in std_logic_vector (2 downto 0);
      addr : in std_logic_vector (12 downto 0);

      data_out : out std_logic_vector(11 downto 0)

   );
   end component;


begin

    R1: RAM_1633_16 port map(
      --In:
      clk => clk,
      we  => ram_we, -- Write-enable
      re  => ram_re, -- Read-enable
      addr => ram_addr, --Address: 11 bits. This allows 0 to 2047, but the
only valid

      --inout:
--addresses range from 0 to 1632.
      data_out  => ram_data_out, --Data In: 16 bits
      data_in => ram_data_in
```

```
      );

   S1: sine_table port map (
      clk => clk,
      reset_n => reset_n,
      data_request => sine_request,
      amplitude => delta_delay,
      addr => sine_addr,
      data_out => sine_data
   );

   -- for giving delta value of delay
   process(clk,reset_n)
   begin
      if(reset_n = '0') then
         delta_delay <= "100";
      elsif(clk'event and clk='1') then
         case delay_amp is
         when "000" =>
            sine_amplitude <= "000000011111";
         when "001" =>
            sine_amplitude <= "000000111111";
         when "010" =>
            sine_amplitude <= "000001111111";
         when "011" =>
            sine_amplitude <= "000011111111";
         when "100" =>
            sine_amplitude <= "000111111111";
         when others =>
            sine_amplitude <= (others => 'X');
         end case;
         delta_delay <= delay_amp;
         freq <= delay_freq;
      end if;
   end process;


   process(clk,reset_n)
    begin
      if(reset_n = '0') then
         sine_addr <= "0000000000000";
         buffer_in <= "000000000000";
         data_ready <= '0';
         freq_cnt <= "0000";
      elsif(clk 'event and clk = '1') then
            if(vibrato_enable = '1') then
            case state is
               when X"0" => -- get the next buffer in
                  data_ready <= '0';
                  if(data_enable = '1') then
                        if(buffer_in = "011001100000") then  --if buffer_in =
1632
                           buffer_in <= "000000000000";
```

```
                     else
                        buffer_in <= buffer_in + 1;
                          end if;
                     data_internal <= data_in;
                     state <= state + 1;
                  end if;
               when X"1" => -- save the current value
                  ram_we <= '1';
                  ram_re <= '0';
                  ram_addr <= buffer_in(10 downto 0);
                  ram_data_in <= data_internal;
                  state <= state + 1;
               when X"2" => -- set the sine_table to get the delay
                  ram_we <= '0';
                  if(sine_addr = "0010111011011") then
                     sine_addr <= "0000000000000";
                     freq_cnt <= "0000";
                  elsif(freq_cnt = freq) then
                     sine_addr <= sine_addr + 1;
                     freq_cnt <= "0000";
                  else
                     freq_cnt <= freq_cnt + 1;
                  end if;
                  sine_request <= '1';
                  state <= state + 1;
               when X"3" => -- get the delay value and apply (in + (not
out+1))
                  sine_request <= '0';
                  delay <= sine_data + (1633 - sine_amplitude);
                  --buffer_out <= (buffer_in + ((not (delay(11 downto
0)))+1));
                  buffer_out <= (buffer_in + ((not (delay(11 downto 0)))+1));
                  state <= state + 1;
               when X"4" => -- if in-out is negative, add buffer size
                  if(buffer_out(11) = '1') then
                     buffer_out <= buffer_out + 1633;
                  end if;
                  state <= state + 1;
               when X"5" => -- read the value from RAM
                  ram_we <= '0';
                  ram_re <= '1';
                  ram_addr <= buffer_out(10 downto 0);
                  state <= state + 1;
               when X"6" => -- apply it into out and send rising edge of
data_ready
                  ram_re <= '0';
                  data_ready <= '1';
                  data_out <= ram_data_out;
                  state <= X"0";
               when others =>
               end case;
         elsif(data_enable = '1') then
            state <= X"0";
```

```
            data_ready <= '1';
               data_out <= data_in;
         else
            data_ready <= '0';
            state <= X"0";
         end if;
        end if;
     end process;
end rtl;
```

Appendix C

Chorus VHDL Code

```
----------------------------------------------------------------
-- chorus module
--
-- Made by Navarun Jagatpal, April 10, 2007
-- Modifyed by Young Jin Yoon, April 12, 2007
-- Will be debugged by everybody....yeah~!
----------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity chorus is
port (
       --In
      clk: in std_logic;      --Should be 50 MHz
      reset_n: in std_logic;
      data_enable: in std_logic;
      chorus_enable: in std_logic;
      delay_amp: in std_logic_vector(2 downto 0);
      delay_freq: in std_logic_vector(3 downto 0);
      mix_delay: in std_logic_vector(3 downto 0);
      data_in: in std_logic_vector(15 downto 0);

       --Out
      data_ready: out std_logic;
      data_out: out std_logic_vector(15 downto 0)
      );
end chorus;

architecture rtl of chorus is

   signal delta_delay: std_logic_vector(2 downto 0);
   signal freq: std_logic_vector(3 downto 0):= "0000";

   signal freq_cnt: std_logic_vector(3 downto 0):= "0000";

   signal delay: std_logic_vector(11 downto 0);

   signal data_internal : std_logic_vector (15 downto 0);

   signal buffer_in: std_logic_vector(11 downto 0);
   signal buffer_out: std_logic_vector(11 downto 0);

   signal ram_re: std_logic;
   signal ram_we: std_logic;
   signal ram_addr: std_logic_vector(10 downto 0);
   signal ram_data_out: std_logic_vector(15 downto 0);
```

```
   signal ram_data_in: std_logic_vector(15 downto 0);

   signal state: std_logic_vector(3 downto 0) := X"0";

   signal sine_data: std_logic_vector(11 downto 0);
   signal sine_addr: std_logic_vector(12 downto 0):= "0000000000000";
   signal sine_request: std_logic;

   signal sine_amplitude: std_logic_vector(11 downto 0);

   signal mix_ram_data_out : std_logic_vector(15 downto 0);
   signal mix_data_internal : std_logic_vector(15 downto 0);
   signal mix_delay_internal : std_logic_vector (3 downto 0);
    component RAM_1633_16 is
port (
       --In:
       clk : in std_logic;
       we   : in std_logic; -- Write-enable
       re   : in std_logic; -- Read-enable
       addr : in std_logic_vector(10 downto 0);
      --Address: 11 bits. This allows 0 to 2047, but the only valid

        --addresses range from 0 to 1632.
       data_out : out std_logic_vector(15 downto 0); --Data In: 16 bits
       data_in : in std_logic_vector(15 downto 0)
       );
   end component;

   component sine_table is
   port (
      clk : in std_logic;
      reset_n : in std_logic;
      data_request : in std_logic;
      amplitude : in std_logic_vector (2 downto 0);
      addr : in std_logic_vector (12 downto 0);

      data_out : out std_logic_vector(11 downto 0)

   );
   end component;


begin

    R1: RAM_1633_16 port map(
      --In:
      clk => clk,
      we  => ram_we, -- Write-enable
      re  => ram_re, -- Read-enable
      addr => ram_addr, --Address: 11 bits. This allows 0 to 2047, but the
only valid

      --inout:
```

```
--addresses range from 0 to 1632.
      data_out  => ram_data_out, --Data In: 16 bits
      data_in => ram_data_in
       );

   S1: sine_table port map (
      clk => clk,
      reset_n => reset_n,
      data_request => sine_request,
      amplitude => delta_delay,
      addr => sine_addr,
      data_out => sine_data
   );

   -- for giving delta value of delay
   process(clk,reset_n)
   begin
      if(reset_n = '0') then
         delta_delay <= "100";
      elsif(clk'event and clk='1') then
         case delay_amp is
         when "000" =>
            sine_amplitude <= "000000011111";
         when "001" =>
            sine_amplitude <= "000000111111";
         when "010" =>
            sine_amplitude <= "000001111111";
         when "011" =>
            sine_amplitude <= "000011111111";
         when "100" =>
            sine_amplitude <= "000111111111";
         when others =>
            sine_amplitude <= (others => 'X');
         end case;
         delta_delay <= delay_amp;
         mix_delay_internal <= mix_delay;
         freq <= delay_freq;
      end if;
   end process;


   process(clk,reset_n)
    begin
      if(reset_n = '0') then
         sine_addr <= "0000000000000";
         buffer_in <= "000000000000";
         data_ready <= '0';
         freq_cnt <= "0000";
      elsif(clk 'event and clk = '1') then
            if(chorus_enable = '1') then
            case state is
               when X"0" => -- get the next buffer in
                  data_ready <= '0';
```

```
                  if(data_enable = '1') then
                        if(buffer_in = "011001100000") then
                              --if buffer_in = 1632
                           buffer_in <= "000000000000";
                     else
                        buffer_in <= buffer_in + 1;
                          end if;
                     data_internal <= data_in;
                     state <= state + 1;
                  end if;
               when X"1" => -- save the current value
                  ram_we <= '1';
                  ram_re <= '0';
                  ram_addr <= buffer_in(10 downto 0);
                  ram_data_in <= data_internal;
                  state <= state + 1;
               when X"2" => -- set the sine_table to get the delay
                  ram_we <= '0';
                  if(sine_addr = "0010111011011") then
                     sine_addr <= "0000000000000";
                     freq_cnt <= "0000";
                  elsif(freq_cnt = freq) then
                     sine_addr <= sine_addr + 1;
                     freq_cnt <= "0000";
                  else
                     freq_cnt <= freq_cnt + 1;
                  end if;
                  sine_request <= '1';
                  state <= state + 1;
               when X"3" =>
                  -- get the delay value and apply (in + (not out+1))
                  sine_request <= '0';
                  delay <= sine_data + (1633 - sine_amplitude);
                  buffer_out <= (buffer_in + ((not (delay(11 downto 0)))+1));
                  state <= state + 1;
               when X"4" => -- if in-out is negative, add buffer size
                  if(buffer_out(11) = '1') then
                     buffer_out <= buffer_out + 1633;
                  end if;
                  state <= state + 1;
               when X"5" => -- read the value from RAM
                  ram_we <= '0';
                  ram_re <= '1';
                  ram_addr <= buffer_out(10 downto 0);
                  state <= state + 1;
               when X"6" =>
               -- apply it into out and send rising edge of data_ready
                  ram_re <= '0';
                  case mix_delay_internal is
                  when X"0" =>
                     if(ram_data_out(15) = '1') then
                        mix_ram_data_out <= "11111111"
                                            & ram_data_out(15 downto 8);
```

```
                    else
                       mix_ram_data_out <= "00000000"
                                           & ram_data_out(15 downto 8);
                    end if;
                    if(data_internal(15) = '1') then
                       mix_data_internal <=
                             ("1" & data_internal(15 downto 1)) +
                             ("11" & data_internal(15 downto 2)) +
                             ("111" & data_internal(15 downto 3)) +
                             ("1111" & data_internal(15 downto 4)) +
                             ("11111" & data_internal(15 downto 5)) +
                             ("111111" & data_internal(15 downto 6)) +
                             ("1111111" & data_internal(15 downto 7)) +
                             ("11111111" & data_internal(15 downto 8));
                    else
                       mix_data_internal <=
                             ("0" & data_internal(15 downto 1)) +
                             ("00" & data_internal(15 downto 2)) +
                             ("000" & data_internal(15 downto 3)) +
                             ("0000" & data_internal(15 downto 4)) +
                             ("00000" & data_internal(15 downto 5)) +
                             ("000000" & data_internal(15 downto 6)) +
                             ("0000000" & data_internal(15 downto 7)) +
                             ("00000000" & data_internal(15 downto 8));
                    end if;
                 when X"1" =>
                    if(ram_data_out(15) = '1') then
                       mix_ram_data_out <= "1111111"
                                           & ram_data_out(15 downto 7);
                    else
                       mix_ram_data_out <= "0000000"
                                           & ram_data_out(15 downto 7);
                    end if;
                    if(data_internal(15) = '1') then
                       mix_data_internal <=
                             ("1" & data_internal(15 downto 1)) +
                             ("11" & data_internal(15 downto 2)) +
                             ("111" & data_internal(15 downto 3)) +
                             ("1111" & data_internal(15 downto 4)) +
                             ("11111" & data_internal(15 downto 5)) +
                             ("111111" & data_internal(15 downto 6)) +
                             ("1111111" & data_internal(15 downto 7));
                    else
                       mix_data_internal <=
                             ("0" & data_internal(15 downto 1)) +
                             ("00" & data_internal(15 downto 2)) +
                             ("000" & data_internal(15 downto 3)) +
                             ("0000" & data_internal(15 downto 4)) +
                             ("00000" & data_internal(15 downto 5)) +
                             ("000000" & data_internal(15 downto 6)) +
                             ("0000000" & data_internal(15 downto 7));
                    end if;
                 when X"2" =>
```

```
                  if(ram_data_out(15) = '1') then
                     mix_ram_data_out <= "111111"
                                         & ram_data_out(15 downto 6);
                  else
                     mix_ram_data_out <= "000000"
                                         & ram_data_out(15 downto 6);
                  end if;
                  if(data_internal(15) = '1') then
                     mix_data_internal <=
                           ("1" & data_internal(15 downto 1)) +
                           ("11" & data_internal(15 downto 2)) +
                           ("111" & data_internal(15 downto 3)) +
                           ("1111" & data_internal(15 downto 4)) +
                           ("11111" & data_internal(15 downto 5)) +
                           ("111111" & data_internal(15 downto 6));
                  else
                     mix_data_internal <=
                           ("0" & data_internal(15 downto 1)) +
                           ("00" & data_internal(15 downto 2)) +
                           ("000" & data_internal(15 downto 3)) +
                           ("0000" & data_internal(15 downto 4)) +
                           ("00000" & data_internal(15 downto 5)) +
                           ("000000" & data_internal(15 downto 6));
                  end if;
               when X"3" =>
                  if(ram_data_out(15) = '1') then
                     mix_ram_data_out <= "11111"
                                         & ram_data_out(15 downto 5);
                  else
                     mix_ram_data_out <= "00000"
                                         & ram_data_out(15 downto 5);
                  end if;
                  if(data_internal(15) = '1') then
                     mix_data_internal <=
                           ("1" & data_internal(15 downto 1)) +
                           ("11" & data_internal(15 downto 2)) +
                           ("111" & data_internal(15 downto 3)) +
                           ("1111" & data_internal(15 downto 4)) +
                           ("11111" & data_internal(15 downto 5));
                  else
                     mix_data_internal <=
                           ("0" & data_internal(15 downto 1)) +
                           ("00" & data_internal(15 downto 2)) +
                           ("000" & data_internal(15 downto 3)) +
                           ("0000" & data_internal(15 downto 4)) +
                           ("00000" & data_internal(15 downto 5));
                  end if;
               when X"4" =>
                  if(ram_data_out(15) = '1') then
                     mix_ram_data_out <= "1111"
                                         & ram_data_out(15 downto 4);
                  else
                     mix_ram_data_out <= "0000"
```

```
                                    & ram_data_out(15 downto 4);
            end if;
            if(data_internal(15) = '1') then
               mix_data_internal <=
                     ("1" & data_internal(15 downto 1)) +
                     ("11" & data_internal(15 downto 2)) +
                     ("111" & data_internal(15 downto 3)) +
                     ("1111" & data_internal(15 downto 4));
            else
               mix_data_internal <=
                     ("0" & data_internal(15 downto 1)) +
                     ("00" & data_internal(15 downto 2)) +
                     ("000" & data_internal(15 downto 3)) +
                     ("0000" & data_internal(15 downto 4));
            end if;
         when X"5" =>
            if(ram_data_out(15) = '1') then
               mix_ram_data_out <= "111"
                                    & ram_data_out(15 downto 3);
            else
               mix_ram_data_out <= "000"
                                    & ram_data_out(15 downto 3);
            end if;
            if(data_internal(15) = '1') then
               mix_data_internal <=
                     ("1" & data_internal(15 downto 1)) +
                     ("11" & data_internal(15 downto 2)) +
                     ("111" & data_internal(15 downto 3));
            else
               mix_data_internal <=
                     ("0" & data_internal(15 downto 1)) +
                     ("00" & data_internal(15 downto 2)) +
                     ("000" & data_internal(15 downto 3));
            end if;
         when X"6" =>
            if(ram_data_out(15) = '1') then
               mix_ram_data_out <= "11"
                                    & ram_data_out(15 downto 2);
            else
               mix_ram_data_out <= "00" & ram_data_out(15 downto 2);
            end if;
            if(data_internal(15) = '1') then
               mix_data_internal <=
                     ("1" & data_internal(15 downto 1)) +
                     ("11" & data_internal(15 downto 2));
            else
               mix_data_internal <=
                     ("0" & data_internal(15 downto 1)) +
                     ("00" & data_internal(15 downto 2));
            end if;
         when X"7" =>
            if(ram_data_out(15) = '1') then
               mix_ram_data_out <= "1" & ram_data_out(15 downto 1);
```

```
                    else
                       mix_ram_data_out <= "0" & ram_data_out(15 downto 1);
                    end if;
                    if(data_internal(15) = '1') then
                       mix_data_internal <=
                             ("1" & data_internal(15 downto 1));
                    else
                       mix_data_internal <=
                             ("0" & data_internal(15 downto 1));
                    end if;
                 when X"8" =>
                    if(ram_data_out(15) = '1') then
                       mix_ram_data_out <=
                             ("1" & ram_data_out(15 downto 1)) +
                             ("11" & ram_data_out(15 downto 2));
                    else
                       mix_ram_data_out <=
                             ("0" & ram_data_out(15 downto 1)) +
                             ("00" & ram_data_out(15 downto 2));
                    end if;
                    if(data_internal(15) = '1') then
                       mix_data_internal <=
                             ("11" & data_internal(15 downto 2));
                    else
                       mix_data_internal <=
                             ("00" & data_internal(15 downto 2));
                    end if;
                 when X"9" =>
                    if(ram_data_out(15) = '1') then
                       mix_ram_data_out <=
                             ("1" & ram_data_out(15 downto 1)) +
                             ("11" & ram_data_out(15 downto 2)) +
                             ("111" & ram_data_out(15 downto 3));
                    else
                       mix_ram_data_out <=
                             ("0" & ram_data_out(15 downto 1)) +
                             ("00" & ram_data_out(15 downto 2)) +
                             ("000" & ram_data_out(15 downto 3));
                    end if;
                    if(data_internal(15) = '1') then
                       mix_data_internal <=
                                   ("111" & data_internal(15 downto 3));
                    else
                       mix_data_internal <=
                                   ("000" & data_internal(15 downto 3));
                    end if;
                 when X"A" =>
                    if(ram_data_out(15) = '1') then
                       mix_ram_data_out <=
                             ("1" & ram_data_out(15 downto 1)) +
                             ("11" & ram_data_out(15 downto 2)) +
                             ("111" & ram_data_out(15 downto 3)) +
                             ("1111" & ram_data_out(15 downto 4));
```

```
                    else
                       mix_ram_data_out <=
                             ("0" & ram_data_out(15 downto 1)) +
                             ("00" & ram_data_out(15 downto 2)) +
                             ("000" & ram_data_out(15 downto 3)) +
                             ("0000" & ram_data_out(15 downto 4));
                    end if;
                    if(data_internal(15) = '1') then
                       mix_data_internal <=
                             ("1111" & data_internal(15 downto 4));
                    else
                       mix_data_internal <=
                             ("0000" & data_internal(15 downto 4));
                    end if;
                 when X"B" =>
                    if(ram_data_out(15) = '1') then
                       mix_ram_data_out <=
                             ("1" & ram_data_out(15 downto 1)) +
                             ("11" & ram_data_out(15 downto 2)) +
                             ("111" & ram_data_out(15 downto 3)) +
                             ("1111" & ram_data_out(15 downto 4)) +
                             ("11111" & ram_data_out(15 downto 5));
                    else
                       mix_ram_data_out <=
                             ("0" & ram_data_out(15 downto 1)) +
                             ("00" & ram_data_out(15 downto 2)) +
                             ("000" & ram_data_out(15 downto 3)) +
                             ("0000" & ram_data_out(15 downto 4)) +
                             ("00000" & ram_data_out(15 downto 5));
                    end if;
                    if(data_internal(15) = '1') then
                       mix_data_internal <=
                             ("11111" & data_internal(15 downto 5));
                    else
                       mix_data_internal <=
                             ("00000" & data_internal(15 downto 5));
                    end if;
                 when X"C" =>
                    if(ram_data_out(15) = '1') then
                       mix_ram_data_out <=
                             ("1" & ram_data_out(15 downto 1)) +
                             ("11" & ram_data_out(15 downto 2)) +
                             ("111" & ram_data_out(15 downto 3)) +
                             ("1111" & ram_data_out(15 downto 4)) +
                             ("11111" & ram_data_out(15 downto 5)) +
                             ("111111" & ram_data_out(15 downto 6));
                    else
                       mix_ram_data_out <=
                             ("0" & ram_data_out(15 downto 1)) +
                             ("00" & ram_data_out(15 downto 2)) +
                             ("000" & ram_data_out(15 downto 3)) +
                             ("0000" & ram_data_out(15 downto 4)) +
                             ("00000" & ram_data_out(15 downto 5)) +
```

```
                        ("000000" & ram_data_out(15 downto 6));
                end if;
                if(data_internal(15) = '1') then
                    mix_data_internal <=
                        ("111111" & data_internal(15 downto 6));
                else
                    mix_data_internal <=
                        ("000000" & data_internal(15 downto 6));
                end if;
            when X"D" =>
                if(ram_data_out(15) = '1') then
                    mix_ram_data_out <=
                        ("1" & ram_data_out(15 downto 1)) +
                        ("11" & ram_data_out(15 downto 2)) +
                        ("111" & ram_data_out(15 downto 3)) +
                        ("1111" & ram_data_out(15 downto 4)) +
                        ("11111" & ram_data_out(15 downto 5)) +
                        ("111111" & ram_data_out(15 downto 6)) +
                        ("1111111" & ram_data_out(15 downto 7));
                else
                    mix_ram_data_out <=
                        ("0" & ram_data_out(15 downto 1)) +
                        ("00" & ram_data_out(15 downto 2)) +
                        ("000" & ram_data_out(15 downto 3)) +
                        ("0000" & ram_data_out(15 downto 4)) +
                        ("00000" & ram_data_out(15 downto 5)) +
                        ("000000" & ram_data_out(15 downto 6)) +
                        ("0000000" & ram_data_out(15 downto 7));
                end if;
                if(data_internal(15) = '1') then
                    mix_data_internal <=
                        ("1111111" & data_internal(15 downto 7));
                else
                    mix_data_internal <=
                        ("0000000" & data_internal(15 downto 7));
                end if;
            when X"E" =>
                if(ram_data_out(15) = '1') then
                    mix_ram_data_out <=
                        ("1" & ram_data_out(15 downto 1)) +
                        ("11" & ram_data_out(15 downto 2)) +
                        ("111" & ram_data_out(15 downto 3)) +
                        ("1111" & ram_data_out(15 downto 4)) +
                        ("11111" & ram_data_out(15 downto 5)) +
                        ("111111" & ram_data_out(15 downto 6)) +
                        ("1111111" & ram_data_out(15 downto 7)) +
                        ("11111111" & ram_data_out(15 downto 8));
                else
                    mix_ram_data_out <=
                        ("0" & ram_data_out(15 downto 1)) +
                        ("00" & ram_data_out(15 downto 2)) +
                        ("000" & ram_data_out(15 downto 3)) +
                        ("0000" & ram_data_out(15 downto 4)) +
```

```
                              ("00000" & ram_data_out(15 downto 5)) +
                              ("000000" & ram_data_out(15 downto 6)) +
                              ("0000000" & ram_data_out(15 downto 7)) +
                              ("00000000" & ram_data_out(15 downto 8));
                     end if;
                     if(data_internal(15) = '1') then
                        mix_data_internal <=
                              ("11111111" & data_internal(15 downto 8));
                     else
                        mix_data_internal <=
                              ("00000000" & data_internal(15 downto 8));
                     end if;
                  when X"F" =>
                     if(ram_data_out(15) = '1') then
                        mix_ram_data_out <=
                              ("1" & ram_data_out(15 downto 1)) +
                              ("11" & ram_data_out(15 downto 2)) +
                              ("111" & ram_data_out(15 downto 3)) +
                              ("1111" & ram_data_out(15 downto 4)) +
                              ("11111" & ram_data_out(15 downto 5)) +
                              ("111111" & ram_data_out(15 downto 6)) +
                              ("1111111" & ram_data_out(15 downto 7)) +
                              ("11111111" & ram_data_out(15 downto 8)) +
                              ("111111111" & ram_data_out(15 downto 9));
                     else
                        mix_ram_data_out <=
                              ("0" & ram_data_out(15 downto 1)) +
                              ("00" & ram_data_out(15 downto 2)) +
                              ("000" & ram_data_out(15 downto 3)) +
                              ("0000" & ram_data_out(15 downto 4)) +
                              ("00000" & ram_data_out(15 downto 5)) +
                              ("000000" & ram_data_out(15 downto 6)) +
                              ("0000000" & ram_data_out(15 downto 7)) +
                              ("00000000" & ram_data_out(15 downto 8)) +
                              ("000000000" & ram_data_out(15 downto 9));
                     end if;
                     if(data_internal(15) = '1') then
                        mix_data_internal <=
                              ("111111111" & data_internal(15 downto 9));
                     else
                        mix_data_internal <=
                              ("000000000" & data_internal(15 downto 9));
                     end if;
                  end case;
                  state <= X"7";
               when X"7" =>
                  data_ready <= '1';
                  data_out <= mix_ram_data_out + mix_data_internal;
                  state <= X"0";
               when others =>
               end case;
            elsif(data_enable = '1') then
               state <= X"0";
```

```
            data_ready <= '1';
               data_out <= data_in;
         else
            data_ready <= '0';
            state <= X"0";
         end if;
        end if;
     end process;
end rtl;
```

Appendix D

Miscellaneous VHDL code

LR Request Buffer in

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- LRRequestBuffer_in : separates left and right channel and forward it into
each processing unit

entity LRRequestBuffer_in is
port (
    clk : in std_logic;
   lrck : in std_logic;
    audio_req : in std_logic;
    data_in : in std_logic_vector (15 downto 0);

   data_left : out std_logic;
   data_right : out std_logic;
    dataL_out : out std_logic_vector(15 downto 0);
    dataR_out : out std_logic_vector(15 downto 0)

  );
end  LRRequestBuffer_in;


architecture Behavioral of  LRRequestBuffer_in is

signal state : std_logic := '0';
-- signal prev_lrck : std_logic;

begin
    process(clk)
    begin
      --if audio request is 1 check to see if its rising edge or falling edge
      if(clk 'event and clk = '1') then
--       prev_lrck <= lrck;
         if(audio_req = '1') then
            if(state = '0') then
               if(lrck = '1') then --if clk is up there then shift out as
right channel (1 clk later)
                data_right <= '1';
                data_left <= '0';
                 dataR_out <= data_in;
                state <= '1';
               else
                data_left <= '1';
```

```
                data_right <= '0';
                 dataL_out <= data_in;
                state <= '1';
               end if;
            else
               data_right <= '0';
               data_left <= '0';
            end if;
          else
            data_right <= '0';
            data_left <= '0';
            state <= '0';
          end if;
      end if;
    end process;
end Behavioral;
```

LR Request Buffer out

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- LRRequestBuffer_out : combines left and right channel and
--                       forward it into audio_out


entity LRRequestBuffer_out is
port (
    clk : in std_logic;       --  Audio CODEC Chip Clock AUD_XCK (18.43 MHz)
   lrck : in std_logic;
   data_left : in std_logic;
   data_right : in std_logic;
    dataL_in : in std_logic_vector(15 downto 0);
    dataR_in : in std_logic_vector(15 downto 0);
    audio_req : in std_logic;

    data_out : out std_logic_vector(15 downto 0)

  );
end  LRRequestBuffer_out;


architecture Behavioral of  LRRequestBuffer_out is

signal buffer_left : std_logic_vector (15 downto 0);
signal buffer_right : std_logic_vector (15 downto 0);

begin
```

```
    process(clk)
    begin
      if(clk 'event and clk = '1') then
         if(data_left = '1') then
            buffer_left <= dataL_in;
         end if;
         if(data_right = '1') then
            buffer_right <= dataR_in;
         end if;
         if(audio_req = '1') then
            if(lrck = '1') then
               data_out <= buffer_left;
            else
               data_out <= buffer_right;
            end if;
         end if;
      end if;
    end process;

-- process(clk)
-- begin
--    if(clk 'event and clk = '1') then
--       if(audio_req = '1') then
--          if(lrck = '1') then
--             data_out <= buffer_left;
 --            else
--             data_out <= buffer_right;
--          end if;
--       end if;
--    end if;
-- end process;

end architecture;
```

Sine Table

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity sine_table is
port (
    clk : in std_logic;
    reset_n : in std_logic;
    data_request : in std_logic;
    amplitude : in std_logic_vector (2 downto 0);
    addr : in std_logic_vector (12 downto 0);

    data_out : out std_logic_vector(11 downto 0)

);
end sine_table;

architecture rtl of sine_table is
```

```
signal delta : std_logic_vector (11 downto 0);
signal sine_total : std_logic_vector (11 downto 0);
--signal data_out : std_logic_vector (7 downto 0);


type table is array (1499 downto 0) of std_logic_vector(11 downto 0);
constant sine_tbl : table := (
    X"FFF", X"FFE", X"FFD", X"FFC", X"FFB", X"FFA", X"FF9", X"FF8", X"FF7", X"FF6", -- 1500
    X"FF5", X"FF4", X"FF3", X"FF1", X"FF0", X"FEF", X"FEE", X"FED", X"FEC", X"FEB", -- 1490
    X"FEA", X"FE9", X"FE8", X"FE7", X"FE6", X"FE5", X"FE4", X"FE3", X"FE1", X"FE0", -- 1480
    X"FDF", X"FDE", X"FDD", X"FDC", X"FDB", X"FDA", X"FD9", X"FD8", X"FD7", X"FD6", -- 1470
    X"FD5", X"FD4", X"FD3", X"FD2", X"FD1", X"FCF", X"FCE", X"FCD", X"FCC", X"FCB", -- 1460
    X"FCA", X"FC9", X"FC8", X"FC7", X"FC6", X"FC5", X"FC4", X"FC3", X"FC2", X"FC1", -- 1450
    X"FC0", X"FBF", X"FBE", X"FBD", X"FBC", X"FBB", X"FBA", X"FB9", X"FB8", X"FB7", -- 1440
    X"FB5", X"FB4", X"FB3", X"FB2", X"FB1", X"FB0", X"FAF", X"FAE", X"FAD", X"FAC", -- 1430
    X"FAB", X"FAA", X"FA9", X"FA8", X"FA7", X"FA6", X"FA5", X"FA4", X"FA3", X"FA2", -- 1420
    X"FA1", X"FA0", X"F9F", X"F9E", X"F9D", X"F9C", X"F9B", X"F9A", X"F99", X"F98", -- 1410
    X"F97", X"F96", X"F95", X"F94", X"F94", X"F93", X"F92", X"F91", X"F90", X"F8F", -- 1400
    X"F8E", X"F8D", X"F8C", X"F8B", X"F8A", X"F89", X"F88", X"F87", X"F86", X"F85", -- 1390
    X"F84", X"F83", X"F82", X"F81", X"F80", X"F80", X"F7F", X"F7E", X"F7D", X"F7C", -- 1380
    X"F7B", X"F7A", X"F79", X"F78", X"F77", X"F76", X"F76", X"F75", X"F74", X"F73", -- 1370
    X"F72", X"F71", X"F70", X"F6F", X"F6E", X"F6E", X"F6D", X"F6C", X"F6B", X"F6A", -- 1360
    X"F69", X"F68", X"F67", X"F67", X"F66", X"F65", X"F64", X"F63", X"F62", X"F61", -- 1350
    X"F61", X"F60", X"F5F", X"F5E", X"F5D", X"F5C", X"F5C", X"F5B", X"F5A", X"F59", -- 1340
    X"F58", X"F58", X"F57", X"F56", X"F55", X"F54", X"F54", X"F53", X"F52", X"F51", -- 1330
    X"F50", X"F50", X"F4F", X"F4E", X"F4D", X"F4D", X"F4C", X"F4B", X"F4A", X"F4A", -- 1320
    X"F49", X"F48", X"F47", X"F47", X"F46", X"F45", X"F44", X"F44", X"F43", X"F42", -- 1310
    X"F42", X"F41", X"F40", X"F3F", X"F3F", X"F3E", X"F3D", X"F3D", X"F3C", X"F3B", -- 1300
    X"F3B", X"F3A", X"F39", X"F39", X"F38", X"F37", X"F37", X"F36", X"F35", X"F35", -- 1290
    X"F34", X"F33", X"F33", X"F32", X"F31", X"F31", X"F30", X"F30", X"F2F", X"F2E", -- 1280
    X"F2E", X"F2D", X"F2C", X"F2C", X"F2B", X"F2B", X"F2A", X"F2A", X"F29", X"F28", -- 1270
    X"F28", X"F27", X"F27", X"F26", X"F26", X"F25", X"F24", X"F24", X"F23", X"F23", -- 1260
    X"F22", X"F22", X"F21", X"F21", X"F20", X"F20", X"F1F", X"F1F", X"F1E", X"F1E", -- 1250
    X"F1D", X"F1D", X"F1C", X"F1C", X"F1B", X"F1B", X"F1A", X"F1A", X"F19", X"F19", -- 1240
    X"F18", X"F18", X"F18", X"F17", X"F17", X"F16", X"F16", X"F15", X"F15", X"F15", -- 1230
    X"F14", X"F14", X"F13", X"F13", X"F12", X"F12", X"F12", X"F11", X"F11", X"F11", -- 1220
    X"F10", X"F10", X"F0F", X"F0F", X"F0F", X"F0E", X"F0E", X"F0E", X"F0D", X"F0D", -- 1210
    X"F0D", X"F0C", X"F0C", X"F0C", X"F0B", X"F0B", X"F0B", X"F0B", X"F0A", X"F0A", -- 1200
    X"F0A", X"F09", X"F09", X"F09", X"F09", X"F08", X"F08", X"F08", X"F08", X"F07", -- 1190
    X"F07", X"F07", X"F07", X"F06", X"F06", X"F06", X"F06", X"F05", X"F05", X"F05", -- 1180
    X"F05", X"F05", X"F04", X"F04", X"F04", X"F04", X"F04", X"F04", X"F03", X"F03", -- 1170
    X"F03", X"F03", X"F03", X"F03", X"F03", X"F02", X"F02", X"F02", X"F02", X"F02", -- 1160
    X"F02", X"F02", X"F02", X"F01", X"F01", X"F01", X"F01", X"F01", X"F01", X"F01", -- 1150
    X"F01", X"F01", X"F01", X"F01", X"F01", X"F01", X"F01", X"F01", X"F01", X"F01", -- 1140
    X"F01", X"F01", X"F01", X"F01", X"F01", X"F01", X"F01", X"F01", X"F01", X"F01", -- 1130
    X"F01", X"F01", X"F01", X"F01", X"F01", X"F01", X"F01", X"F01", X"F01", X"F01", -- 1120
    X"F01", X"F01", X"F01", X"F01", X"F01", X"F01", X"F02", X"F02", X"F02", X"F02", -- 1110
    X"F02", X"F02", X"F02", X"F02", X"F03", X"F03", X"F03", X"F03", X"F03", X"F03", -- 1100
    X"F03", X"F04", X"F04", X"F04", X"F04", X"F04", X"F04", X"F05", X"F05", X"F05", -- 1090
    X"F05", X"F05", X"F06", X"F06", X"F06", X"F06", X"F07", X"F07", X"F07", X"F07", -- 1080
    X"F08", X"F08", X"F08", X"F08", X"F09", X"F09", X"F09", X"F09", X"F0A", X"F0A", -- 1070
    X"F0A", X"F0B", X"F0B", X"F0B", X"F0B", X"F0C", X"F0C", X"F0C", X"F0D", X"F0D", -- 1060
    X"F0D", X"F0E", X"F0E", X"F0E", X"F0F", X"F0F", X"F0F", X"F10", X"F10", X"F11", -- 1050
    X"F11", X"F11", X"F12", X"F12", X"F12", X"F13", X"F13", X"F14", X"F14", X"F15", -- 1040
    X"F15", X"F15", X"F16", X"F16", X"F17", X"F17", X"F18", X"F18", X"F18", X"F19", -- 1030
    X"F19", X"F1A", X"F1A", X"F1B", X"F1B", X"F1C", X"F1C", X"F1D", X"F1D", X"F1E", -- 1020
    X"F1E", X"F1F", X"F1F", X"F20", X"F20", X"F21", X"F21", X"F22", X"F22", X"F23", -- 1010
    X"F23", X"F24", X"F24", X"F25", X"F26", X"F26", X"F27", X"F27", X"F28", X"F28", -- 1000
    X"F29", X"F2A", X"F2A", X"F2B", X"F2B", X"F2C", X"F2C", X"F2D", X"F2E", X"F2E", -- 990
    X"F2F", X"F30", X"F30", X"F31", X"F31", X"F32", X"F33", X"F33", X"F34", X"F35", -- 980
    X"F35", X"F36", X"F37", X"F37", X"F38", X"F39", X"F39", X"F3A", X"F3B", X"F3B", -- 970
    X"F3C", X"F3D", X"F3D", X"F3E", X"F3F", X"F3F", X"F40", X"F41", X"F42", X"F42", -- 960
    X"F43", X"F44", X"F44", X"F45", X"F46", X"F47", X"F47", X"F48", X"F49", X"F4A", -- 950
    X"F4A", X"F4B", X"F4C", X"F4D", X"F4D", X"F4E", X"F4F", X"F50", X"F50", X"F51", -- 940
    X"F52", X"F53", X"F54", X"F54", X"F55", X"F56", X"F57", X"F58", X"F58", X"F59", -- 930
    X"F5A", X"F5B", X"F5C", X"F5C", X"F5D", X"F5E", X"F5F", X"F60", X"F61", X"F61", -- 920
```

```
    X"F62", X"F63", X"F64", X"F65", X"F66", X"F67", X"F67", X"F68", X"F69", X"F6A", -- 910
    X"F6B", X"F6C", X"F6D", X"F6E", X"F6E", X"F6F", X"F70", X"F71", X"F72", X"F73", -- 900
    X"F74", X"F75", X"F76", X"F76", X"F77", X"F78", X"F79", X"F7A", X"F7B", X"F7C", -- 890
    X"F7D", X"F7E", X"F7F", X"F80", X"F81", X"F81", X"F82", X"F83", X"F84", X"F85", -- 880
    X"F86", X"F87", X"F88", X"F89", X"F8A", X"F8B", X"F8C", X"F8D", X"F8E", X"F8F", -- 870
    X"F90", X"F91", X"F92", X"F93", X"F94", X"F94", X"F95", X"F96", X"F97", X"F98", -- 860
    X"F99", X"F9A", X"F9B", X"F9C", X"F9D", X"F9E", X"F9F", X"FA0", X"FA1", X"FA2", -- 850
    X"FA3", X"FA4", X"FA5", X"FA6", X"FA7", X"FA8", X"FA9", X"FAA", X"FAB", X"FAC", -- 840
    X"FAD", X"FAE", X"FAF", X"FB0", X"FB1", X"FB2", X"FB3", X"FB4", X"FB5", X"FB7", -- 830
    X"FB8", X"FB9", X"FBA", X"FBB", X"FBC", X"FBD", X"FBE", X"FBF", X"FC0", X"FC1", -- 820
    X"FC2", X"FC3", X"FC4", X"FC5", X"FC6", X"FC7", X"FC8", X"FC9", X"FCA", X"FCB", -- 810
    X"FCC", X"FCD", X"FCE", X"FCF", X"FD1", X"FD2", X"FD3", X"FD4", X"FD5", X"FD6", -- 800
    X"FD7", X"FD8", X"FD9", X"FDA", X"FDB", X"FDC", X"FDD", X"FDE", X"FDF", X"FE0", -- 790
    X"FE1", X"FE3", X"FE4", X"FE5", X"FE6", X"FE7", X"FE8", X"FE9", X"FEA", X"FEB", -- 780
    X"FEC", X"FED", X"FEE", X"FEF", X"FF0", X"FF1", X"FF3", X"FF4", X"FF5", X"FF6", -- 770
    X"FF7", X"FF8", X"FF9", X"FFA", X"FFB", X"FFC", X"FFD", X"FFE", X"FFF", X"000", -- 760
    X"001", X"002", X"003", X"004", X"005", X"006", X"007", X"008", X"009", X"00A", -- 750
    X"00B", X"00C", X"00D", X"00F", X"010", X"011", X"012", X"013", X"014", X"015", -- 740
    X"016", X"017", X"018", X"019", X"01A", X"01B", X"01C", X"01D", X"01F", X"020", -- 730
    X"021", X"022", X"023", X"024", X"025", X"026", X"027", X"028", X"029", X"02A", -- 720
    X"02B", X"02C", X"02D", X"02E", X"02F", X"031", X"032", X"033", X"034", X"035", -- 710
    X"036", X"037", X"038", X"039", X"03A", X"03B", X"03C", X"03D", X"03E", X"03F", -- 700
    X"040", X"041", X"042", X"043", X"044", X"045", X"046", X"047", X"048", X"049", -- 690
    X"04B", X"04C", X"04D", X"04E", X"04F", X"050", X"051", X"052", X"053", X"054", -- 680
    X"055", X"056", X"057", X"058", X"059", X"05A", X"05B", X"05C", X"05D", X"05E", -- 670
    X"05F", X"060", X"061", X"062", X"063", X"064", X"065", X"066", X"067", X"068", -- 660
    X"069", X"06A", X"06B", X"06C", X"06C", X"06D", X"06E", X"06F", X"070", X"071", -- 650
    X"072", X"073", X"074", X"075", X"076", X"077", X"078", X"079", X"07A", X"07B", -- 640
    X"07C", X"07D", X"07E", X"07F", X"080", X"080", X"081", X"082", X"083", X"084", -- 630
    X"085", X"086", X"087", X"088", X"089", X"08A", X"08A", X"08B", X"08C", X"08D", -- 620
    X"08E", X"08F", X"090", X"091", X"092", X"092", X"093", X"094", X"095", X"096", -- 610
    X"097", X"098", X"099", X"099", X"09A", X"09B", X"09C", X"09D", X"09E", X"09F", -- 600
    X"09F", X"0A0", X"0A1", X"0A2", X"0A3", X"0A4", X"0A4", X"0A5", X"0A6", X"0A7", -- 590
    X"0A8", X"0A8", X"0A9", X"0AA", X"0AB", X"0AC", X"0AC", X"0AD", X"0AE", X"0AF", -- 580
    X"0B0", X"0B0", X"0B1", X"0B2", X"0B3", X"0B3", X"0B4", X"0B5", X"0B6", X"0B6", -- 570
    X"0B7", X"0B8", X"0B9", X"0B9", X"0BA", X"0BB", X"0BC", X"0BC", X"0BD", X"0BE", -- 560
    X"0BE", X"0BF", X"0C0", X"0C1", X"0C1", X"0C2", X"0C3", X"0C3", X"0C4", X"0C5", -- 550
    X"0C5", X"0C6", X"0C7", X"0C7", X"0C8", X"0C9", X"0C9", X"0CA", X"0CB", X"0CB", -- 540
    X"0CC", X"0CD", X"0CD", X"0CE", X"0CF", X"0CF", X"0D0", X"0D0", X"0D1", X"0D2", -- 530
    X"0D2", X"0D3", X"0D4", X"0D4", X"0D5", X"0D5", X"0D6", X"0D6", X"0D7", X"0D8", -- 520
    X"0D8", X"0D9", X"0D9", X"0DA", X"0DA", X"0DB", X"0DC", X"0DC", X"0DD", X"0DD", -- 510
    X"0DE", X"0DE", X"0DF", X"0DF", X"0E0", X"0E0", X"0E1", X"0E1", X"0E2", X"0E2", -- 500
    X"0E3", X"0E3", X"0E4", X"0E4", X"0E5", X"0E5", X"0E6", X"0E6", X"0E7", X"0E7", -- 490
    X"0E8", X"0E8", X"0E8", X"0E9", X"0E9", X"0EA", X"0EA", X"0EB", X"0EB", X"0EB", -- 480
    X"0EC", X"0EC", X"0ED", X"0ED", X"0EE", X"0EE", X"0EE", X"0EF", X"0EF", X"0EF", -- 470
    X"0F0", X"0F0", X"0F1", X"0F1", X"0F1", X"0F2", X"0F2", X"0F2", X"0F3", X"0F3", -- 460
    X"0F3", X"0F4", X"0F4", X"0F4", X"0F5", X"0F5", X"0F5", X"0F5", X"0F6", X"0F6", -- 450
    X"0F6", X"0F7", X"0F7", X"0F7", X"0F7", X"0F8", X"0F8", X"0F8", X"0F8", X"0F9", -- 440
    X"0F9", X"0F9", X"0F9", X"0FA", X"0FA", X"0FA", X"0FA", X"0FB", X"0FB", X"0FB", -- 430
    X"0FB", X"0FB", X"0FC", X"0FC", X"0FC", X"0FC", X"0FC", X"0FC", X"0FD", X"0FD", -- 420
    X"0FD", X"0FD", X"0FD", X"0FD", X"0FD", X"0FE", X"0FE", X"0FE", X"0FE", X"0FE", -- 410
    X"0FE", X"0FE", X"0FE", X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", -- 400
    X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", -- 390
    X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", -- 380
    X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", -- 370
    X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", X"0FF", X"0FE", X"0FE", X"0FE", X"0FE", -- 360
    X"0FE", X"0FE", X"0FE", X"0FE", X"0FD", X"0FD", X"0FD", X"0FD", X"0FD", X"0FD", -- 350
    X"0FD", X"0FC", X"0FC", X"0FC", X"0FC", X"0FC", X"0FC", X"0FB", X"0FB", X"0FB", -- 340
    X"0FB", X"0FB", X"0FA", X"0FA", X"0FA", X"0FA", X"0F9", X"0F9", X"0F9", X"0F9", -- 330
    X"0F8", X"0F8", X"0F8", X"0F8", X"0F7", X"0F7", X"0F7", X"0F7", X"0F6", X"0F6", -- 320
    X"0F6", X"0F5", X"0F5", X"0F5", X"0F5", X"0F4", X"0F4", X"0F4", X"0F3", X"0F3", -- 310
    X"0F3", X"0F2", X"0F2", X"0F2", X"0F1", X"0F1", X"0F1", X"0F0", X"0F0", X"0EF", -- 300
    X"0EF", X"0EF", X"0EE", X"0EE", X"0EE", X"0ED", X"0ED", X"0EC", X"0EC", X"0EB", -- 290
    X"0EB", X"0EB", X"0EA", X"0EA", X"0E9", X"0E9", X"0E8", X"0E8", X"0E8", X"0E7", -- 280
    X"0E7", X"0E6", X"0E6", X"0E5", X"0E5", X"0E4", X"0E4", X"0E3", X"0E3", X"0E2", -- 270
    X"0E2", X"0E1", X"0E1", X"0E0", X"0E0", X"0DF", X"0DF", X"0DE", X"0DE", X"0DD", -- 260
    X"0DD", X"0DC", X"0DC", X"0DB", X"0DA", X"0DA", X"0D9", X"0D9", X"0D8", X"0D8", -- 250
```

```
    X"0D7", X"0D6", X"0D6", X"0D5", X"0D5", X"0D4", X"0D4", X"0D3", X"0D2", X"0D2", -- 240
    X"0D1", X"0D0", X"0D0", X"0CF", X"0CF", X"0CE", X"0CD", X"0CD", X"0CC", X"0CB", -- 230
    X"0CB", X"0CA", X"0C9", X"0C9", X"0C8", X"0C7", X"0C7", X"0C6", X"0C5", X"0C5", -- 220
    X"0C4", X"0C3", X"0C3", X"0C2", X"0C1", X"0C1", X"0C0", X"0BF", X"0BE", X"0BE", -- 210
    X"0BD", X"0BC", X"0BC", X"0BB", X"0BA", X"0B9", X"0B9", X"0B8", X"0B7", X"0B6", -- 200
    X"0B6", X"0B5", X"0B4", X"0B3", X"0B3", X"0B2", X"0B1", X"0B0", X"0B0", X"0AF", -- 190
    X"0AE", X"0AD", X"0AC", X"0AC", X"0AB", X"0AA", X"0A9", X"0A8", X"0A8", X"0A7", -- 180
    X"0A6", X"0A5", X"0A4", X"0A4", X"0A3", X"0A2", X"0A1", X"0A0", X"09F", X"09F", -- 170
    X"09E", X"09D", X"09C", X"09B", X"09A", X"099", X"099", X"098", X"097", X"096", -- 160
    X"095", X"094", X"093", X"092", X"092", X"091", X"090", X"08F", X"08E", X"08D", -- 150
    X"08C", X"08B", X"08A", X"08A", X"089", X"088", X"087", X"086", X"085", X"084", -- 140
    X"083", X"082", X"081", X"080", X"07F", X"07F", X"07E", X"07D", X"07C", X"07B", -- 130
    X"07A", X"079", X"078", X"077", X"076", X"075", X"074", X"073", X"072", X"071", -- 120
    X"070", X"06F", X"06E", X"06D", X"06C", X"06C", X"06B", X"06A", X"069", X"068", -- 110
    X"067", X"066", X"065", X"064", X"063", X"062", X"061", X"060", X"05F", X"05E", -- 100
    X"05D", X"05C", X"05B", X"05A", X"059", X"058", X"057", X"056", X"055", X"054", -- 90
    X"053", X"052", X"051", X"050", X"04F", X"04E", X"04D", X"04C", X"04B", X"049", -- 80
    X"048", X"047", X"046", X"045", X"044", X"043", X"042", X"041", X"040", X"03F", -- 70
    X"03E", X"03D", X"03C", X"03B", X"03A", X"039", X"038", X"037", X"036", X"035", -- 60
    X"034", X"033", X"032", X"031", X"02F", X"02E", X"02D", X"02C", X"02B", X"02A", -- 50
    X"029", X"028", X"027", X"026", X"025", X"024", X"023", X"022", X"021", X"020", -- 40
    X"01F", X"01D", X"01C", X"01B", X"01A", X"019", X"018", X"017", X"016", X"015", -- 30
    X"014", X"013", X"012", X"011", X"010", X"00F", X"00D", X"00C", X"00B", X"00A", -- 20
    X"009", X"008", X"007", X"006", X"005", X"004", X"003", X"002", X"001", X"000"  -- 10
);


begin

--  data <= data_out;

    process(clk, reset_n)
    begin
        if(reset_n = '0') then
            delta <= (others => '0');
            sine_total <= (others => '0');
            data_out <= (others => '0');
        elsif(clk 'event and clk = '1') then
            if(data_request = '1') then
                if(addr < "0010111011011") then
                    sine_total <= sine_tbl(conv_integer(addr));
                    case amplitude is
                        when "000" =>
                            if(sine_total(11) = '0') then
                                data_out <= "0000" & sine_total(11 downto 4);
                            else
                                data_out <= "1111" & sine_total(11 downto 4);
                            end if;
                        when "001" =>
                            if(sine_total(11) = '0') then
                                data_out <= "000" & sine_total(11 downto 3);
                            else
                                data_out <= "111" & sine_total(11 downto 3);
                            end if;
                        when "010" =>
                            if(sine_total(11) = '0') then
                                data_out <= "00" & sine_total(11 downto 2);
                            else
                                data_out <= "11" & sine_total(11 downto 2);
                            end if;
                        when "011" =>
                            if(sine_total(11) = '0') then
                                data_out <= '0' & sine_total(11 downto 1);
                            else
                                data_out <= '1' & sine_total(11 downto 1);
                            end if;
                        when "100" =>
```

```
                        data_out <= sine_total;
                    when others =>
                        data_out <= (others => 'X');
                end case;
            else
                data_out <= (others => 'X');
            end if;
        end if;
    end if;
  end process;
end rtl;
```

de2_wm8731_audio_in

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- de2_wm8731_audio_in : generate clock and get the samples from device

entity de2_wm8731_audio_in is
port (
    clk : in std_logic;       --  Audio CODEC Chip Clock AUD_XCK (18.43 MHz)
    reset_n : in std_logic;
    data_out : out std_logic_vector(15 downto 0);
    audio_req : out std_logic;

    -- Audio interface signals
   AUD_ADCLRCK  : out std_logic;   --    Audio CODEC ADC LR Clock
    AUD_ADCDAT   : in  std_logic;   --    Audio CODEC ADC Data
    AUD_BCLK     : inout std_logic  --    Audio CODEC Bit-Stream Clock
  );
end  de2_wm8731_audio_in;


architecture Behavioral of  de2_wm8731_audio_in is

    signal lrck : std_logic;
    signal bclk : std_logic;
    signal xck : std_logic;

    signal lrck_divider : std_logic_vector (7 downto 0);
    signal bclk_divider : std_logic_vector (3 downto 0);

    signal set_bclk : std_logic;
    signal set_lrck : std_logic;
    signal lrck_lat : std_logic;
   signal clr_bclk : std_logic;
    signal datain : std_logic;

    signal shift_in : std_logic_vector ( 15 downto 0);
   signal shift_counter : integer := 15;
```

```
begin
    -- LRCK divider
    -- Audio chip main clock is 18.432MHz / Sample rate 48KHz
    -- Divider is 18.432 MHz / 48KHz = 192 (X"C0")
    -- Left justify mode set by I2C controller

    process(clk, reset_n)
    begin
        if ( reset_n = '0' ) then
            lrck_divider <= (others => '0');
        elsif ( clk'event and clk='1' ) then
            if ( lrck_divider = X"BF")  then        -- "C0" minus 1
                lrck_divider <= X"00";
            else
                lrck_divider <= lrck_divider + '1';
            end if;
        end if;
    end process;

    process(clk, reset_n)
    begin
        if ( reset_n = '0' ) then
            bclk_divider <= (others => '0');
        elsif ( clk'event and clk='1' ) then
            if ( bclk_divider = X"B" or set_lrck = '1')  then
                bclk_divider <= X"0";
            else
                bclk_divider <= bclk_divider + '1';
            end if;
        end if;
    end process;


    process ( lrck_divider )
    begin
        if ( lrck_divider = X"BF") then
            set_lrck <= '1';
        else
            set_lrck <= '0';
        end if;
    end process;

    process ( clk, reset_n)
    begin
        if ( reset_n = '0') then
            lrck <= '0';
        elsif ( clk 'event and clk = '1') then
            if ( set_lrck = '1') then
                lrck <= not lrck;
            end if;
        end if;
    end process;
```

```
    -- BCLK divider
    process ( bclk_divider )
    begin
        if ( bclk_divider (3 downto 0) = "0101") then
            set_bclk <= '1';
        else
            set_bclk <= '0';
        end if;

        if ( bclk_divider (3 downto 0) = "1011") then
            clr_bclk <= '1';
        else
            clr_bclk <= '0';
        end if;
    end process;

    process ( clk, reset_n)
    begin
        if ( reset_n = '0') then
            bclk <= '0';
        elsif ( clk 'event and clk = '1') then
            if ( set_lrck = '1' or clr_bclk = '1') then
                bclk <= '0';
            elsif ( set_bclk = '1') then
                bclk <= '1';
            end if;
        end if;
    end process;

   process (clk)
   begin
   if ( clk 'event and clk = '1') then
--    if ( clr_bclk = '1') then
         -- shift_in <= shift_in (14 downto 0) & '0';
   -- els
      if (set_bclk = '1') then
   -- datain <= AUD_ADCDAT;
         shift_in(shift_counter) <= AUD_ADCDAT;
         if (shift_counter = 0) then
            shift_counter <= 15;
         else
            shift_counter <= shift_counter - 1;
         end if;
      end if;
   end if;
   end process;
   process(clk)
    begin
        if ( clk'event and clk='1' ) then
            lrck_lat <= lrck;
        end if;
    end process;
   process (clk)
```

```
    begin
        if ( clk'event and clk = '1') then
            if (( lrck_lat = '1' and lrck = '0') or ( lrck_lat = '0' and lrck
= '1')) then
                audio_req <= '1';
            else
                audio_req <= '0';
            end if;
        end if;
    end process;
    -- Audio data shift output
    process ( clk, reset_n)
    begin
        if ( clk 'event and clk = '1') then
            if ( set_lrck = '1') then
               data_out <= shift_in;
            end if;
        end if;
    end process;

    -- Audio outputs

    AUD_BCLK     <= bclk;
   AUD_ADCLRCK  <= lrck;

end architecture;
```

de2_wm8731_audio_out

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- de2_wm8731_audio_in : generate clock and set the samples from device

entity de2_wm8731_audio_out is
port (
    clk : in std_logic;       --  Audio CODEC Chip Clock AUD_XCK (18.43 MHz)
    reset_n : in std_logic;
    test_mode : in std_logic;       --    Audio CODEC controller test mode
    data : in std_logic_vector(15 downto 0);
   audio_req : out std_logic;

    -- Audio interface signals
--    AUD_ADCLRCK  : out std_logic;   --    Audio CODEC ADC LR Clock
    AUD_DACLRCK  : out std_logic;   --    Audio CODEC DAC LR Clock
    AUD_DACDAT   : out std_logic   --    Audio CODEC DAC Data
--    AUD_BCLK     : inout std_logic  --    Audio CODEC Bit-Stream Clock
  );
end  de2_wm8731_audio_out;
```

```
architecture Behavioral of  de2_wm8731_audio_out is

    signal lrck : std_logic;
    signal bclk : std_logic;
    signal xck : std_logic;

    signal lrck_divider : std_logic_vector (7 downto 0);
    signal bclk_divider : std_logic_vector (3 downto 0);

    signal set_bclk : std_logic;
    signal set_lrck : std_logic;
    signal clr_bclk : std_logic;
    signal lrck_lat : std_logic;

    signal shift_out : std_logic_vector ( 15 downto 0);

    signal sin_out : std_logic_vector ( 15 downto 0);
    signal sin_counter : std_logic_vector ( 5 downto 0);


begin

    -- LRCK divider
    -- Audio chip main clock is 18.432MHz / Sample rate 48KHz
    -- Divider is 18.432 MHz / 48KHz = 192 (X"C0")
    -- Left justify mode set by I2C controller

    process(clk, reset_n)
    begin
        if ( reset_n = '0' ) then
            lrck_divider <= (others => '0');
        elsif ( clk'event and clk='1' ) then
            if ( lrck_divider = X"BF")  then        -- "C0" minus 1
                lrck_divider <= X"00";
            else
                lrck_divider <= lrck_divider + '1';
            end if;
        end if;
    end process;

    process(clk, reset_n)
    begin
        if ( reset_n = '0' ) then
            bclk_divider <= (others => '0');
        elsif ( clk'event and clk='1' ) then
            if ( bclk_divider = X"B" or set_lrck = '1')  then
                bclk_divider <= X"0";
            else
                bclk_divider <= bclk_divider + '1';
            end if;
        end if;
```

```
    end process;


    process ( lrck_divider )
    begin
        if ( lrck_divider = X"BF") then
            set_lrck <= '1';
        else
            set_lrck <= '0';
        end if;
    end process;

    process ( clk, reset_n)
    begin
        if ( reset_n = '0') then
            lrck <= '0';
        elsif ( clk 'event and clk = '1') then
            if ( set_lrck = '1') then
                lrck <= not lrck;
            end if;
        end if;
    end process;

    -- BCLK divider
    process ( bclk_divider )
    begin
        if ( bclk_divider (3 downto 0) = "0101") then
            set_bclk <= '1';
        else
            set_bclk <= '0';
        end if;

        if ( bclk_divider (3 downto 0) = "1011") then
            clr_bclk <= '1';
        else
            clr_bclk <= '0';
        end if;
    end process;

    process ( clk, reset_n)
    begin
        if ( reset_n = '0') then
            bclk <= '0';
        elsif ( clk 'event and clk = '1') then
            if ( set_lrck = '1' or clr_bclk = '1') then
                bclk <= '0';
            elsif ( set_bclk = '1') then
                bclk <= '1';
            end if;
        end if;
    end process;

    -- Audio data shift output
```

```
    process ( clk, reset_n)
    begin
        if ( reset_n = '0') then
            shift_out <= (others => '0');
        elsif ( clk 'event and clk = '1') then
            if ( set_lrck = '1') then
                if ( test_mode = '1') then
                    shift_out <= sin_out;
                else
                    shift_out <= data;
                end if;
            elsif ( clr_bclk = '1') then
                shift_out <= shift_out (14 downto 0) & '0';
            end if;
        end if;
    end process;

    -- Audio outputs

--    AUD_ADCLRCK  <= lrck;
    AUD_DACLRCK  <= lrck;
    AUD_DACDAT   <= shift_out(15);
--    AUD_BCLK     <= bclk;

    -- Self test with Sin wave

    process(clk, reset_n)
    begin
        if ( reset_n = '0' ) then
            sin_counter <= (others => '0');
        elsif ( clk'event and clk='1' ) then
            if ( lrck_lat = '1' and lrck = '0')  then
                if (sin_counter = "101111") then
                    sin_counter <= "000000";
                else
                    sin_counter <= sin_counter + '1';
                end if;
            end if;
        end if;
    end process;

    process(clk)
    begin
        if ( clk'event and clk='1' ) then
            lrck_lat <= lrck;
        end if;
    end process;
   process (clk)
    begin
        if ( clk'event and clk = '1') then
            if (( lrck_lat = '1' and lrck = '0') or ( lrck_lat = '0' and lrck
= '1')) then
                audio_req <= '1';
```

```
            else
                audio_req <= '0';
            end if;
        end if;
    end process;
    process ( sin_counter )
    begin
        case sin_counter is
            when "000000" => sin_out <= X"0000";
            when "000001" => sin_out <= X"10b4";
            when "000010" => sin_out <= X"2120";
            when "000011" => sin_out <= X"30fb";
            when "000100" => sin_out <= X"3fff";
            when "000101" => sin_out <= X"4deb";
            when "000110" => sin_out <= X"5a81";
            when "000111" => sin_out <= X"658b";
            when "001000" => sin_out <= X"6ed9";
            when "001001" => sin_out <= X"7640";
            when "001010" => sin_out <= X"7ba2";
            when "001011" => sin_out <= X"7ee6";
            when "001100" => sin_out <= X"7fff";
            when "001101" => sin_out <= X"7ee6";
            when "001110" => sin_out <= X"7ba2";
            when "001111" => sin_out <= X"7640";
            when "010000" => sin_out <= X"6ed9";
            when "010001" => sin_out <= X"658b";
            when "010010" => sin_out <= X"5a81";
            when "010011" => sin_out <= X"4deb";
            when "010100" => sin_out <= X"3fff";
            when "010101" => sin_out <= X"30fb";
            when "010110" => sin_out <= X"2120";
            when "010111" => sin_out <= X"10b4";
            when "011000" => sin_out <= X"0000";
            when "011001" => sin_out <= X"ef4b";
            when "011010" => sin_out <= X"dee0";
            when "011011" => sin_out <= X"cf05";
            when "011100" => sin_out <= X"c001";
            when "011101" => sin_out <= X"b215";
            when "011110" => sin_out <= X"a57e";
            when "011111" => sin_out <= X"9a74";
            when "100000" => sin_out <= X"9127";
            when "100001" => sin_out <= X"89bf";
            when "100010" => sin_out <= X"845d";
            when "100011" => sin_out <= X"8119";
            when "100100" => sin_out <= X"8000";
            when "100101" => sin_out <= X"8119";
            when "100110" => sin_out <= X"845d";
            when "100111" => sin_out <= X"89bf";
            when "101000" => sin_out <= X"9127";
            when "101001" => sin_out <= X"9a74";
            when "101010" => sin_out <= X"a57e";
            when "101011" => sin_out <= X"b215";
            when "101100" => sin_out <= X"c000";
```

```
            when "101101" => sin_out <= X"cf05";
            when "101110" => sin_out <= X"dee0";
            when "101111" => sin_out <= X"ef4b";
            when others => sin_out <= X"0000";
        end case;
    end process;


end architecture;
```

Effector

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity effector is
port (
   clk : in std_logic;
   audio_clk : in std_logic;
   reset_n : in std_logic;

   -- for temporal testing... we will include this as bus commande from CPU!
   Ldistortion_enable : in std_logic;
   Lclip : in std_logic_vector(15 downto 0);

   Rdistortion_enable : in std_logic;
   Rclip : in std_logic_vector(15 downto 0);

    Lvibrato_enable: in std_logic;
    Lvibrato_amp: in std_logic_vector(2 downto 0);
   Lvibrato_freq: in std_logic_vector(3 downto 0);

    Rvibrato_enable: in std_logic;
    Rvibrato_amp: in std_logic_vector(2 downto 0);
   Rvibrato_freq: in std_logic_vector(3 downto 0);

   Lchorus_enable: in std_logic;
   Lchorus_amp: in std_logic_vector(2 downto 0);
   Lchorus_freq: in std_logic_vector(3 downto 0);
   Lchorus_mix_delay: in std_logic_vector(3 downto 0);

   Rchorus_enable: in std_logic;
   Rchorus_amp: in std_logic_vector(2 downto 0);
   Rchorus_freq: in std_logic_vector(3 downto 0);
   Rchorus_mix_delay: in std_logic_vector(3 downto 0);

   Lvol_valid: in std_logic;
   Lvol_data: in std_logic_vector(6 downto 0);
```

```
   Rvol_valid: in std_logic;
   Rvol_data: in std_logic_vector(6 downto 0);

    -- Audio interface signals
   AUD_ADCLRCK  : out std_logic;   --    Audio CODEC ADC LR Clock
    AUD_ADCDAT   : in  std_logic;   --    Audio CODEC ADC Data
    AUD_BCLK     : inout std_logic;  --    Audio CODEC Bit-Stream Clock
    AUD_DACLRCK  : out std_logic;   --    Audio CODEC DAC LR Clock
    AUD_DACDAT   : out std_logic;   --    Audio CODEC DAC Data
   I2C_SCLK  : out std_logic;
   I2C_SDAT  : inout std_logic
);
end effector;


architecture rtl of effector is

component de2_wm8731_audio_in is
port (
    clk : in std_logic;       --  Audio CODEC Chip Clock AUD_XCK (18.43 MHz)
    reset_n : in std_logic;
    data_out : out std_logic_vector(15 downto 0);
    audio_req : out std_logic;

    -- Audio interface signals
   AUD_ADCLRCK  : out std_logic;   --    Audio CODEC ADC LR Clock
    AUD_ADCDAT   : in  std_logic;   --    Audio CODEC ADC Data
    AUD_BCLK     : inout std_logic  --    Audio CODEC Bit-Stream Clock
  );
end  component;

signal data_from_ADC_to_LR_in : std_logic_vector(15 downto 0);
signal request_from_ADC_to_LR_in : std_logic;


component LRRequestBuffer_in is
port (
    clk : in std_logic;
   lrck : in std_logic;
    audio_req : in std_logic;
    data_in : in std_logic_vector (15 downto 0);

   data_left : out std_logic;
   data_right : out std_logic;
    dataL_out : out std_logic_vector(15 downto 0);
    dataR_out : out std_logic_vector(15 downto 0)

  );
end  component;

signal dataL_from_LR_in_to_Lchorus : std_logic_vector(15 downto 0);
signal dataR_from_LR_in_to_Rchorus : std_logic_vector (15 downto 0);
```

```
signal data_req_from_LR_in_to_Lchorus : std_logic;
signal data_req_from_LR_in_to_Rchorus : std_logic;

component chorus is
port (
       --In
       clk: in std_logic;      --Should be 50 MHz
      reset_n: in std_logic;
       data_enable: in std_logic;
       chorus_enable: in std_logic;
      delay_amp: in std_logic_vector(2 downto 0);
      delay_freq: in std_logic_vector(3 downto 0);
      mix_delay: in std_logic_vector(3 downto 0);
       data_in: in std_logic_vector(15 downto 0);

       --Out
      data_ready: out std_logic;
       data_out: out std_logic_vector(15 downto 0)
       );
end component;

signal data_ready_from_Lchorus_to_Lvibrato : std_logic;
signal data_ready_from_Rchorus_to_Rvibrato : std_logic;

signal data_from_Lchorus_to_Lvibrato : std_logic_vector(15 downto 0);
signal data_from_Rchorus_to_Rvibrato : std_logic_vector(15 downto 0);

component vibrato is
port (
       --In
       clk: in std_logic;      --Should be 50 MHz
      reset_n: in std_logic;
       data_enable: in std_logic;
       vibrato_enable: in std_logic;
      delay_amp: in std_logic_vector(2 downto 0);
      delay_freq: in std_logic_vector(3 downto 0);
      data_in: in std_logic_vector(15 downto 0);

       --Out
      data_ready: out std_logic;
       data_out: out std_logic_vector(15 downto 0)
       );
end component;

signal data_ready_from_Lvibrato_to_Ldistortion : std_logic;
signal data_ready_from_Rvibrato_to_Rdistortion : std_logic;

signal data_from_Lvibrato_to_Ldistortion : std_logic_vector(15 downto 0);
signal data_from_Rvibrato_to_Rdistortion : std_logic_vector(15 downto 0);

component distortion is
port (
```

```
   clk : in std_logic;
   reset_n : in std_logic;
   data_enable : in std_logic;
   distortion_enable : in std_logic;
   clip : in std_logic_vector(15 downto 0);
   data_in : in std_logic_vector(15 downto 0);

   data_out : out std_logic_vector(15 downto 0);
   data_ready : out std_logic

);
end component;


signal data_ready_from_Ldistortion_to_LR_out : std_logic;
signal data_ready_from_Rdistortion_to_LR_out : std_logic;

signal data_from_Ldistortion_to_LR_out: std_logic_vector(15 downto 0);
signal data_from_Rdistortion_to_LR_out: std_logic_vector(15 downto 0);


component LRRequestBuffer_out is
port (
    clk : in std_logic;       --  Audio CODEC Chip Clock AUD_XCK (18.43 MHz)
   lrck : in std_logic;
   data_left : in std_logic;
   data_right : in std_logic;
    dataL_in : in std_logic_vector(15 downto 0);
    dataR_in : in std_logic_vector(15 downto 0);
    audio_req : in std_logic;

    data_out : out std_logic_vector (15 downto 0)

  );
end  component;

signal request_from_DAC_to_LR_out : std_logic;
signal data_from_LR_out_to_DAC : std_logic_vector(15 downto 0);


component de2_wm8731_audio_out is
port (
    clk : in std_logic;       --  Audio CODEC Chip Clock AUD_XCK (18.43 MHz)
    reset_n : in std_logic;
    test_mode : in std_logic;       --    Audio CODEC controller test mode
    data : in std_logic_vector(15 downto 0);
   audio_req : out std_logic;

    -- Audio interface signals
    AUD_DACLRCK  : out std_logic;   --    Audio CODEC DAC LR Clock
    AUD_DACDAT   : out std_logic   --    Audio CODEC DAC Data
  );
end component;
```

```
signal internal_lrck_in, internal_lrck_out : std_logic;

component de2_i2c_av_config
port (
 iCLK    : in std_logic;
 iRST_N     : in std_logic;
 I2C_SCLK   : out std_logic;
 I2C_SDAT   : inout std_logic;
 left_valid : in std_logic;
 left_data  : in std_logic_vector(6 downto 0);
 right_valid: in std_logic;
 right_data : in std_logic_vector(6 downto 0)
);
end component;


begin

   AUD_ADCLRCK <= internal_lrck_in;
   AUD_DACLRCK <= internal_lrck_out;

   AUDIO_IN: de2_wm8731_audio_in port map(
       clk => audio_clk,
       reset_n => reset_n,
       data_out => data_from_ADC_to_LR_in,
       audio_req => request_from_ADC_to_LR_in,

       -- Audio interface signals
      AUD_ADCLRCK  => internal_lrck_in,
       AUD_ADCDAT   => AUD_ADCDAT,   --    Audio CODEC ADC Data
       AUD_BCLK     => AUD_BCLK  --    Audio CODEC Bit-Stream Clock
   );


   BUFFER_IN: LRRequestBuffer_in port map(
       clk => clk,
      lrck => internal_lrck_in,
       audio_req => request_from_ADC_to_LR_in,
       data_in =>    data_from_ADC_to_LR_in,

      data_left => data_req_from_LR_in_to_Lchorus,
      data_right => data_req_from_LR_in_to_Rchorus,
       dataL_out => dataL_from_LR_in_to_Lchorus,
       dataR_out => dataR_from_LR_in_to_Rchorus
   );



 LCHO:chorus port map(
       --In
       clk => clk,     --Should be 50 MHz
      reset_n => reset_n,
```

```
       data_enable => data_req_from_LR_in_to_Lchorus,
       chorus_enable => Lchorus_enable,
      delay_amp => Lchorus_amp,
      delay_freq => Lchorus_freq,
      mix_delay => Lchorus_mix_delay,

       data_in => dataL_from_LR_in_to_Lchorus,
       --Out
      data_ready => data_ready_from_Lchorus_to_Lvibrato,
       data_out => data_from_Lchorus_to_Lvibrato
  );

RCHO:chorus port map(
       --In
       clk => clk,     --Should be 50 MHz
      reset_n => reset_n,
       data_enable => data_req_from_LR_in_to_Rchorus,
       chorus_enable => Rchorus_enable,
      delay_amp => Rchorus_amp,
      delay_freq => Rchorus_freq,
      mix_delay => Rchorus_mix_delay,
       data_in => dataR_from_LR_in_to_Rchorus,
       --Out
      data_ready => data_ready_from_Rchorus_to_Rvibrato,
       data_out => data_from_Rchorus_to_Rvibrato
  );



  LVIB: vibrato port map(
        --In
        clk => clk,      --Should be 50 MHz
        reset_n => reset_n,
        data_enable => data_ready_from_Lchorus_to_Lvibrato,
        vibrato_enable => Lvibrato_enable,
     delay_amp => Lvibrato_amp,
      delay_freq => Lvibrato_freq,
        data_in => data_from_Lchorus_to_Lvibrato,

        --Out
        data_ready => data_ready_from_Lvibrato_to_Ldistortion,
        data_out => data_from_Lvibrato_to_Ldistortion
   );


  RVIB: vibrato port map(
      --In
      clk => clk,      --Should be 50 MHz
     reset_n => reset_n,
      data_enable => data_ready_from_Rchorus_to_Rvibrato,
      vibrato_enable => Rvibrato_enable,
     delay_amp => Rvibrato_amp,
      delay_freq => Rvibrato_freq,
```

```
      data_in => data_from_Rchorus_to_Rvibrato,

          --Out
      data_ready => data_ready_from_Rvibrato_to_Rdistortion,
      data_out => data_from_Rvibrato_to_Rdistortion
    );

   LDIS: distortion port map (
      clk => clk,
      reset_n => reset_n,
      data_enable => data_ready_from_Lvibrato_to_Ldistortion,
      distortion_enable => Ldistortion_enable,
      clip => Lclip,
      data_in => data_from_Lvibrato_to_Ldistortion,

      data_out => data_from_Ldistortion_to_LR_out,
      data_ready => data_ready_from_Ldistortion_to_LR_out
   );

   RDIS: distortion port map (
      clk => clk,
      reset_n => reset_n,
      data_enable => data_ready_from_Rvibrato_to_Rdistortion,
      distortion_enable => Rdistortion_enable,
      clip => Rclip,
      data_in => data_from_Rvibrato_to_Rdistortion,

      data_out => data_from_Rdistortion_to_LR_out,
      data_ready => data_ready_from_Rdistortion_to_LR_out
   );

   BUFFER_OUT: LRRequestBuffer_out port map(
       clk => clk,
      lrck => internal_lrck_out,
      data_left => data_ready_from_Ldistortion_to_LR_out,
      data_right => data_ready_from_Rdistortion_to_LR_out,
       dataL_in => data_from_Ldistortion_to_LR_out,
       dataR_in => data_from_Rdistortion_to_LR_out,
       audio_req => request_from_DAC_to_LR_out,

       data_out => data_from_LR_out_to_DAC
    );


   AUDIO_OUT: de2_wm8731_audio_out port map(
       clk => audio_clk,
       reset_n => reset_n,
       test_mode => '0',
       data => data_from_LR_out_to_DAC,
      audio_req => request_from_DAC_to_LR_out,

       -- Audio interface signals
       AUD_DACLRCK  => internal_lrck_out,
```

```
       AUD_DACDAT   => AUD_DACDAT
   );


  i2c : de2_i2c_av_config port map (
    iCLK     => clk,
    iRST_n   => reset_n,
    I2C_SCLK => I2C_SCLK,
    I2C_SDAT => I2C_SDAT,
   left_valid => Lvol_valid,
   right_valid => Rvol_valid,
   left_data => Lvol_data,
   right_data => Rvol_data
  );

end rtl;
```

RAM module

```
-----------------------------------------------------------------
-- 1633 by 16 RAM: 1633 slots, 16 bits per slot
--
-- Made by Navarun Jagatpal, April 10, 2007
-----------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity RAM_1633_16 is
port (
               --In:
               clk : in std_logic;
               we   : in std_logic; -- Write-enable
               re   : in std_logic; -- Read-enable
               addr : in std_logic_vector(10 downto 0); --Address: 11 bits.
This allows 0 to 2047, but the only valid

            --inout:
--addresses range from 0 to 1632.
               data_out : out std_logic_vector(15 downto 0); --Data In: 16
bits
            data_in  : in std_logic_vector(15 downto 0)
       );
end RAM_1633_16;

architecture myRAM of RAM_1633_16 is
       type ram_type is array (1632 downto 0) of std_logic_vector(15 downto
0);
       signal RAM : ram_type;
       signal do_internal : std_logic_vector(7 downto 0); --Internal Copy of
Data Out Signal
begin
```

```
	process(clk)
	begin
		if(clk'event and clk='1') then
			if(we = '1') then
				RAM(conv_integer(addr)) <= data_in;
			end if;
			if(re = '1') then
				data_out <= RAM(conv_integer(addr)) ;
			end if;
		end if;
	end process;
end myRAM;
```

Guitar_effects_top

```
--
-- DE2 top-level module that includes the simple audio component
--
-- Stephen A. Edwards, Columbia University, sedwards@cs.columbia.edu
--
-- From an original by Terasic Technology, Inc.
-- (DE2_TOP.v, part of the DE2 system board CD supplied by Altera)
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity guitar_effects_top is

  port (
    -- Clocks

    CLOCK_27,                                   -- 27 MHz
    CLOCK_50,                                   -- 50 MHz
    EXT_CLOCK : in std_logic;                   -- External Clock

    -- Buttons and switches

    KEY : in std_logic_vector(3 downto 0);      -- Push buttons
    SW : in std_logic_vector(17 downto 0);      -- DPDT switches

    -- LED displays

    HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7 -- 7-segment displays
       : out std_logic_vector(6 downto 0);
    LEDG : out std_logic_vector(8 downto 0);    -- Green LEDs
    LEDR : out std_logic_vector(17 downto 0);   -- Red LEDs

    -- RS-232 interface

    UART_TXD : out std_logic;                   -- UART transmitter
    UART_RXD : in std_logic;                    -- UART receiver

    -- IRDA interface

--    IRDA_TXD : out std_logic;                   -- IRDA Transmitter
    IRDA_RXD : in std_logic;                    -- IRDA Receiver
```

```
    -- SDRAM

    DRAM_DQ : inout std_logic_vector(15 downto 0); -- Data Bus
    DRAM_ADDR : out std_logic_vector(11 downto 0); -- Address Bus
    DRAM_LDQM,                                     -- Low-byte Data Mask
    DRAM_UDQM,                                     -- High-byte Data Mask
    DRAM_WE_N,                                     -- Write Enable
    DRAM_CAS_N,                                    -- Column Address Strobe
    DRAM_RAS_N,                                    -- Row Address Strobe
    DRAM_CS_N,                                     -- Chip Select
    DRAM_BA_0,                                     -- Bank Address 0
    DRAM_BA_1,                                     -- Bank Address 0
    DRAM_CLK,                                      -- Clock
    DRAM_CKE : out std_logic;                      -- Clock Enable

    -- FLASH

    FL_DQ : inout std_logic_vector(7 downto 0);      -- Data bus
    FL_ADDR : out std_logic_vector(21 downto 0);  -- Address bus
    FL_WE_N,                                       -- Write Enable
    FL_RST_N,                                      -- Reset
    FL_OE_N,                                       -- Output Enable
    FL_CE_N : out std_logic;                       -- Chip Enable

    -- SRAM

    SRAM_DQ : inout std_logic_vector(15 downto 0); -- Data bus 16 Bits
    SRAM_ADDR : out std_logic_vector(17 downto 0); -- Address bus 18 Bits
    SRAM_UB_N,                                     -- High-byte Data Mask
    SRAM_LB_N,                                     -- Low-byte Data Mask
    SRAM_WE_N,                                     -- Write Enable
    SRAM_CE_N,                                     -- Chip Enable
    SRAM_OE_N : out std_logic;                     -- Output Enable

    -- USB controller

    OTG_DATA : inout std_logic_vector(15 downto 0); -- Data bus
    OTG_ADDR : out std_logic_vector(1 downto 0);    -- Address
    OTG_CS_N,                                       -- Chip Select
    OTG_RD_N,                                       -- Write
    OTG_WR_N,                                       -- Read
    OTG_RST_N,                                      -- Reset
    OTG_FSPEED,                     -- USB Full Speed, 0 = Enable, Z = Disable
    OTG_LSPEED : out std_logic;     -- USB Low Speed, 0 = Enable, Z = Disable
    OTG_INT0,                                       -- Interrupt 0
    OTG_INT1,                                       -- Interrupt 1
    OTG_DREQ0,                                      -- DMA Request 0
    OTG_DREQ1 : in std_logic;                       -- DMA Request 1
    OTG_DACK0_N,                                    -- DMA Acknowledge 0
    OTG_DACK1_N : out std_logic;                    -- DMA Acknowledge 1

    -- 16 X 2 LCD Module

    LCD_ON,                     -- Power ON/OFF
    LCD_BLON,                   -- Back Light ON/OFF
    LCD_RW,                     -- Read/Write Select, 0 = Write, 1 = Read
    LCD_EN,                     -- Enable
    LCD_RS : out std_logic;     -- Command/Data Select, 0 = Command, 1 = Data
    LCD_DATA : inout std_logic_vector(7 downto 0); -- Data bus 8 bits

    -- SD card interface

    SD_DAT,                     -- SD Card Data
    SD_DAT3,                    -- SD Card Data 3
    SD_CMD : inout std_logic;   -- SD Card Command Signal
    SD_CLK : out std_logic;     -- SD Card Clock
```

```
    -- USB JTAG link

    TDI,                        -- CPLD -> FPGA (data in)
    TCK,                        -- CPLD -> FPGA (clk)
    TCS : in std_logic;         -- CPLD -> FPGA (CS)
    TDO : out std_logic;        -- FPGA -> CPLD (data out)

    -- I2C bus

    I2C_SDAT : inout std_logic; -- I2C Data
    I2C_SCLK : out std_logic;   -- I2C Clock

    -- PS/2 port

    PS2_DAT,                    -- Data
    PS2_CLK : in std_logic;     -- Clock

    -- VGA output

    VGA_CLK,                                            -- Clock
    VGA_HS,                                             -- H_SYNC
    VGA_VS,                                             -- V_SYNC
    VGA_BLANK,                                          -- BLANK
    VGA_SYNC : out std_logic;                           -- SYNC
    VGA_R,                                              -- Red[9:0]
    VGA_G,                                              -- Green[9:0]
    VGA_B : out std_logic_vector(9 downto 0);           -- Blue[9:0]

    --  Ethernet Interface

    ENET_DATA : inout std_logic_vector(15 downto 0);    -- DATA bus 16Bits
    ENET_CMD,           -- Command/Data Select, 0 = Command, 1 = Data
    ENET_CS_N,                                          -- Chip Select
    ENET_WR_N,                                          -- Write
    ENET_RD_N,                                          -- Read
    ENET_RST_N,                                         -- Reset
    ENET_CLK : out std_logic;                           -- Clock 25 MHz
    ENET_INT : in std_logic;                            -- Interrupt

    -- Audio CODEC

    AUD_ADCLRCK : inout std_logic;                      -- ADC LR Clock
    AUD_ADCDAT : in std_logic;                          -- ADC Data
    AUD_DACLRCK : inout std_logic;                      -- DAC LR Clock
    AUD_DACDAT : out std_logic;                         -- DAC Data
    AUD_BCLK : inout std_logic;                         -- Bit-Stream Clock
    AUD_XCK : out std_logic;                            -- Chip Clock

    -- Video Decoder

    TD_DATA : in std_logic_vector(7 downto 0);  -- Data bus 8 bits
    TD_HS,                                      -- H_SYNC
    TD_VS : in std_logic;                       -- V_SYNC
    TD_RESET : out std_logic;                   -- Reset

    -- General-purpose I/O

    GPIO_0,                                      -- GPIO Connection 0
    GPIO_1 : inout std_logic_vector(35 downto 0) -- GPIO Connection 1
    );

end guitar_effects_top;

architecture datapath of guitar_effects_top is

component guitar_effects_with_nios is
        port (
```

```
               -- 1) global signals:
                  signal clk : IN STD_LOGIC;
                  signal reset_n : IN STD_LOGIC;

               -- the_de2_ps2_0
                  signal PS2_Clk_to_the_de2_ps2_0 : IN STD_LOGIC;
                  signal PS2_Data_to_the_de2_ps2_0 : IN STD_LOGIC;

               -- the_de2_sram_controller_0
                  signal SRAM_ADDR_from_the_de2_sram_controller_0 : OUT STD_LOGIC_VECTOR (17
DOWNTO 0);
                  signal SRAM_CE_N_from_the_de2_sram_controller_0 : OUT STD_LOGIC;
                  signal SRAM_DQ_to_and_from_the_de2_sram_controller_0 : INOUT STD_LOGIC_VECTOR
(15 DOWNTO 0);
                  signal SRAM_LB_N_from_the_de2_sram_controller_0 : OUT STD_LOGIC;
                  signal SRAM_OE_N_from_the_de2_sram_controller_0 : OUT STD_LOGIC;
                  signal SRAM_UB_N_from_the_de2_sram_controller_0 : OUT STD_LOGIC;
                  signal SRAM_WE_N_from_the_de2_sram_controller_0 : OUT STD_LOGIC;

               -- the_effector_avnet_0
                  signal AUD_ADCDAT_to_the_effector_avnet_0 : IN STD_LOGIC;
                  signal AUD_ADCLRCK_from_the_effector_avnet_0 : OUT STD_LOGIC;
                  signal AUD_BCLK_to_and_from_the_effector_avnet_0 : INOUT STD_LOGIC;
                  signal AUD_DACDAT_from_the_effector_avnet_0 : OUT STD_LOGIC;
                  signal AUD_DACLRCK_from_the_effector_avnet_0 : OUT STD_LOGIC;
                  signal I2C_SCLK_from_the_effector_avnet_0 : OUT STD_LOGIC;
                  signal I2C_SDAT_to_and_from_the_effector_avnet_0 : INOUT STD_LOGIC;
                  signal audio_clk_to_the_effector_avnet_0 : IN STD_LOGIC;

               -- the_lcd_16207_0
                  signal LCD_E_from_the_lcd_16207_0 : OUT STD_LOGIC;
                  signal LCD_RS_from_the_lcd_16207_0 : OUT STD_LOGIC;
                  signal LCD_RW_from_the_lcd_16207_0 : OUT STD_LOGIC;
                  signal LCD_data_to_and_from_the_lcd_16207_0 : INOUT STD_LOGIC_VECTOR (7 DOWNTO
0)
               );
end component;


  signal audio_clock : std_logic_vector(1 downto 0) := "00";

begin

G1: guitar_effects_with_nios port map(
               -- 1) global signals,
                  clk => CLOCK_50,
                  reset_n => SW(17),

               -- the_de2_ps2_0
                  PS2_Clk_to_the_de2_ps2_0 => PS2_CLK,
                  PS2_Data_to_the_de2_ps2_0 => PS2_DAT,

               -- the_de2_sram_controller_0
                  SRAM_ADDR_from_the_de2_sram_controller_0 => SRAM_ADDR,
                  SRAM_CE_N_from_the_de2_sram_controller_0 => SRAM_CE_N,
                  SRAM_DQ_to_and_from_the_de2_sram_controller_0 => SRAM_DQ,
                  SRAM_LB_N_from_the_de2_sram_controller_0 => SRAM_LB_N,
                  SRAM_OE_N_from_the_de2_sram_controller_0 => SRAM_OE_N,
                  SRAM_UB_N_from_the_de2_sram_controller_0 => SRAM_UB_N,
                  SRAM_WE_N_from_the_de2_sram_controller_0 => SRAM_WE_N,

               -- the_effector_avnet_0
                  AUD_ADCDAT_to_the_effector_avnet_0 => AUD_ADCDAT,
                  AUD_ADCLRCK_from_the_effector_avnet_0 => AUD_ADCLRCK,
                  AUD_BCLK_to_and_from_the_effector_avnet_0 => AUD_BCLK,
                  AUD_DACDAT_from_the_effector_avnet_0 => AUD_DACDAT,
                  AUD_DACLRCK_from_the_effector_avnet_0 => AUD_DACLRCK,
```

```
                I2C_SCLK_from_the_effector_avnet_0 => I2C_SCLK,
                I2C_SDAT_to_and_from_the_effector_avnet_0 => I2C_SDAT,
                audio_clk_to_the_effector_avnet_0 => audio_clock(1),

             -- the_lcd_16207_0
                LCD_E_from_the_lcd_16207_0 => LCD_EN,
                LCD_RS_from_the_lcd_16207_0 => LCD_RS,
                LCD_RW_from_the_lcd_16207_0 => LCD_RW,
                LCD_data_to_and_from_the_lcd_16207_0 => LCD_DATA
             );


  process (CLOCK_50)
  begin
    if CLOCK_50'event and CLOCK_50 = '1' then
      audio_clock <= audio_clock + 1;
    end if;
  end process;

  AUD_XCK <= audio_clock(1);


  HEX7     <= "1111111";
  HEX6     <= "1111111";
  HEX5     <= "1111111";
  HEX4     <= "1111111";
  HEX3     <= "1111111";
  HEX2     <= "1111111";
  HEX1     <= "1111111";
  HEX0     <= "1111111";          -- Rightmost
  LEDR(17 downto 0) <= (others => '1');
  LCD_ON   <= '1';
  LCD_BLON <= '1';

  -- Set all bidirectional ports to tri-state
  DRAM_DQ     <= (others => 'Z');
  FL_DQ       <= (others => 'Z');
  SRAM_DQ     <= (others => 'Z');
  OTG_DATA    <= (others => 'Z');
  LCD_DATA    <= (others => 'Z');
  SD_DAT      <= 'Z';
  I2C_SDAT    <= 'Z';
  ENET_DATA   <= (others => 'Z');
  GPIO_0      <= (others => 'Z');
  GPIO_1      <= (others => 'Z');

end datapath;
```

Appendix E

Software Control Code

```
//LCD 80A00 to 80A0F
//effector: 80800 809FF
//
#include <stdio.h>
#include <alt_types.h>
#include "alt_up_ps2_port.h"
#include "LCD.h"
//#include "ps2_keyboard.h"

#define EFFECTOR_BASE 0x80800

#define LVOL 0
#define RVOL 2

#define LDIS_EN 4
#define LDIS_CLIP 6
#define RDIS_EN 8
#define RDIS_CLIP 10

#define LVIB_EN 12
#define LVIB_AMP 14
#define LVIB_FREQ 16

#define RVIB_EN 18
#define RVIB_AMP 20
#define RVIB_FREQ 22

#define LCHO_EN 24
#define LCHO_AMP 26
#define LCHO_FREQ 28
#define LCHO_MIX 30

#define RCHO_EN 32
#define RCHO_AMP 34
#define RCHO_FREQ 36
#define RCHO_MIX 38

#define KEY_NORMAL 0
#define KEY_EXTEND 1
#define KEY_RELEASE 2
#define KEY_EXTEND_RELEASE 3

#define KEY_RELEASED 0
#define KEY_PRESSED 1

// D: distortion enable
// V: vibrato enable
// C: chorus enable
```

```
// ARROW UP : volume up
// ARROW DOWN : volume down

// 1 : select chorus
// 2 : select vibrato
// 3 : select distortion

// Insert : conf 1 up (chorus, vibrato : AMP, distortion : CLIP)
// Delete : conf 1 down (chorus, vibrato : AMP, distortion : CLIP)
// Home : conf 2 up (chorus, vibrato : FREQ, distortion : NONE)
// End : conf 2 down (chorus, vibrato : FREQ, distortion : NONE)
// Page Up : conf 3 up (chorus : MIX, distortion, vibrato : NONE)
// Page Down : conf 3 down (chorus : MIX, distortion, vibrato : NONE)

//#define IO_EFFECTOR (x,y) { volatile short int *temp = 0x80800; *(temp+x) =
y;}

int key_state;
int key_pressed;
// key_board = 0x80A10;
#define KEY_BASE 0x80A10

int main()
{
  short int *EFFECTOR = (short int *)0x80800;
  char display1[16];
  char display2[16];
  int disp_changed = 0;
  printf("Please wait three seconds to initialize keyboard\n");
  int code;
   LCD_Init();
//   LCD_Show_Text("YO WORK!");

   key_state = KEY_NORMAL;
   key_pressed = KEY_RELEASED;

   IOWR_16DIRECT(EFFECTOR+LVOL,0,121);    //left volume = 0  (0 is in the
middle of the volume range)
   IOWR_16DIRECT(EFFECTOR+RVOL,0,121);    //right volume = 0  (0 is in the
middle of the volume range)

   IOWR_16DIRECT(EFFECTOR+LDIS_EN,0,0); //left dist enable = 0 (left
distortion off)
   IOWR_16DIRECT(EFFECTOR+LDIS_CLIP,0,(1<<8)); //left dist clipping value =
(2^8)

   IOWR_16DIRECT(EFFECTOR+RDIS_EN,0,0); //right dist same as left dist
   IOWR_16DIRECT(EFFECTOR+RDIS_CLIP,0,(1<<8));

   IOWR_16DIRECT(EFFECTOR+LVIB_EN,0,0); //left vibrato enable = 0 (left
vibrato off)
   IOWR_16DIRECT(EFFECTOR+LVIB_AMP,0,4);
```

```
   IOWR_16DIRECT(EFFECTOR+LVIB_FREQ,0,0);

   IOWR_16DIRECT(EFFECTOR+RVIB_EN,0,0); //right vibrato same as left vibrato
   IOWR_16DIRECT(EFFECTOR+RVIB_AMP,0,4);
   IOWR_16DIRECT(EFFECTOR+RVIB_FREQ,0,0);

   IOWR_16DIRECT(EFFECTOR+LCHO_EN,0,0); //left chorus enable = 0 (left chorus
off)
   IOWR_16DIRECT(EFFECTOR+LCHO_AMP,0,4);
   IOWR_16DIRECT(EFFECTOR+LCHO_FREQ,0,0);
   IOWR_16DIRECT(EFFECTOR+LCHO_MIX,0,8);

   IOWR_16DIRECT(EFFECTOR+RCHO_EN,0,0); //right chorus same as left chorus
   IOWR_16DIRECT(EFFECTOR+RCHO_AMP,0,4);
   IOWR_16DIRECT(EFFECTOR+RCHO_FREQ,0,0);
   IOWR_16DIRECT(EFFECTOR+RCHO_MIX,0,8);


   char dist_on = 0;
   char vibr_on = 0;
   char chor_on = 0;
   unsigned char volume = 121;
   char selected = 0;
   unsigned short dis_clip, chor_amp, chor_freq, chor_mix, vib_amp, vib_freq;
   dis_clip = 1<<8;
   chor_amp = vib_amp = 4;
   chor_freq = vib_freq = 0;
   chor_mix = 8;
   LCD_Show_Text("Guitar Effect");
   while(1) {
        while(!IORD_8DIRECT(KEY_BASE, 0));
        code = IORD_8DIRECT(KEY_BASE, 4);
//        printf(" %x pressed!",code);

        switch(code) {
        case 0xf0:
            switch(key_state) {
                case KEY_NORMAL:
                    key_state = KEY_RELEASE;
                    break;
                case KEY_EXTEND:
                    key_state = KEY_EXTEND_RELEASE;
                    break;
                case KEY_RELEASE:
                case KEY_EXTEND_RELEASE:
                    break;
            }
            break;
        case 0xe0:
            switch(key_state) {
                case KEY_NORMAL:
                    key_state = KEY_EXTEND;
                    break;
```

```
                case KEY_EXTEND:
                case KEY_RELEASE:
                case KEY_EXTEND_RELEASE:
                    break;
             }
             break;
        case 0x23: //D
            if(key_state == KEY_NORMAL && key_pressed == KEY_RELEASED) {
                // do something with pressing D
                if(dist_on) {
                    sprintf(display1,"Dist off");
                    disp_changed = 1;
                    IOWR_16DIRECT(EFFECTOR+LDIS_EN, 0, 0); //disabling left
distortion
                    IOWR_16DIRECT(EFFECTOR+RDIS_EN, 0, 0); //disabling right
distortion
                    dist_on = 0;
                }
                else {
                    sprintf(display1,"Dist on");
                    disp_changed = 1;
                    IOWR_16DIRECT(EFFECTOR+LDIS_EN, 0, 1); //enabling left
distortion
                    IOWR_16DIRECT(EFFECTOR+RDIS_EN, 0, 1); //enabling right
distortion
                    dist_on = 1;
                }
                key_pressed = KEY_PRESSED;
            }
            if(key_state == KEY_RELEASE && key_pressed == KEY_PRESSED) {
                key_pressed = KEY_RELEASED;
            }
            key_state = KEY_NORMAL;
            break;
        case 0x21: //C
            if(key_state == KEY_NORMAL && key_pressed == KEY_RELEASED) {
                if(chor_on) {
                    sprintf(display1, "Chorus off");
                    disp_changed = 1;
                    IOWR_16DIRECT(EFFECTOR+LCHO_EN, 0, 0); //disabling left
distortion
                    IOWR_16DIRECT(EFFECTOR+RCHO_EN, 0, 0); //disabling right
distortion
                    chor_on = 0;
                }
                else {
                    sprintf(display1,"Chorus on");
                    disp_changed = 1;
                    IOWR_16DIRECT(EFFECTOR+LCHO_EN, 0, 1); //enabling left
distortion
                    IOWR_16DIRECT(EFFECTOR+RCHO_EN, 0, 1); //enabling right
distortion
                    chor_on = 1;
```

```
                }
                key_pressed = KEY_PRESSED;
            }
            if(key_state == KEY_RELEASE && key_pressed == KEY_PRESSED) {
                key_pressed = KEY_RELEASED;
            }
            key_state = KEY_NORMAL;
            break;
        case 0x2A: //V
            if(key_state == KEY_NORMAL && key_pressed == KEY_RELEASED) {
                if(vibr_on) {
                    sprintf(display1,"Vibrato off");
                    disp_changed = 1;
                    IOWR_16DIRECT(EFFECTOR+LVIB_EN, 0, 0); //disabling left
distortion
                    IOWR_16DIRECT(EFFECTOR+RVIB_EN, 0, 0); //disabling right
distortion
                    vibr_on = 0;
                }
                else {
                    sprintf(display1,"Vibrato on");
                    disp_changed = 1;
                    IOWR_16DIRECT(EFFECTOR+LVIB_EN, 0, 1); //enabling left
distortion
                    IOWR_16DIRECT(EFFECTOR+RVIB_EN, 0, 1); //enabling right
distortion
                    vibr_on = 1;
                }
                key_pressed = KEY_PRESSED;
            }
            if(key_state == KEY_RELEASE && key_pressed == KEY_PRESSED) {
                key_pressed = KEY_RELEASED;
            }
            key_state = KEY_NORMAL;
            break;
        case 0x16: // 1
            if(key_state == KEY_NORMAL && key_pressed == KEY_RELEASED) {
                sprintf(display1,"Chrous selected");
                disp_changed = 1;
                selected = 0;
                key_pressed = KEY_PRESSED;
            }
            if(key_state == KEY_RELEASE && key_pressed == KEY_PRESSED) {
                key_pressed = KEY_RELEASED;
            }
            key_state = KEY_NORMAL;
            break;
        case 0x1E: // 2
            if(key_state == KEY_NORMAL && key_pressed == KEY_RELEASED) {
                sprintf(display1,"Vibrato selected");
                disp_changed = 1;
                selected = 1;
                key_pressed = KEY_PRESSED;
```

```
            }
            if(key_state == KEY_RELEASE && key_pressed == KEY_PRESSED) {
                key_pressed = KEY_RELEASED;
            }
            key_state = KEY_NORMAL;
            break;
        case 0x26: // 3
            if(key_state == KEY_NORMAL && key_pressed == KEY_RELEASED) {
               sprintf(display1,"Dist selected");
               disp_changed = 1;
               selected = 2;
               key_pressed = KEY_PRESSED;
            }
            if(key_state == KEY_RELEASE && key_pressed == KEY_PRESSED) {
                key_pressed = KEY_RELEASED;
            }
            key_state = KEY_NORMAL;
            break;
        case 0x75: // with e0 - UP Arrow
            if(key_state == KEY_EXTEND && key_pressed == KEY_RELEASED) {
                if(volume != 127) volume++;
                sprintf(display1,"vol = %d",volume-48);
                disp_changed = 1;
                IOWR_16DIRECT(EFFECTOR+LVOL,0,volume);
                IOWR_16DIRECT(EFFECTOR+RVOL,0,volume);
                key_pressed = KEY_PRESSED;
            }
            if(key_state == KEY_EXTEND_RELEASE && key_pressed == KEY_PRESSED)
{
                key_pressed = KEY_RELEASED;
            }
            key_state = KEY_NORMAL;
            break;
        case 0x72: // with e0 - DOWN Arrow
            if(key_state == KEY_EXTEND && key_pressed == KEY_RELEASED) {
                if(volume != 48) volume--;
                sprintf(display1,"vol = %d",volume-48);
                disp_changed = 1;
                IOWR_16DIRECT(EFFECTOR+LVOL,0,volume);
                IOWR_16DIRECT(EFFECTOR+RVOL,0,volume);
                key_pressed = KEY_PRESSED;
            }
            if(key_state == KEY_EXTEND_RELEASE && key_pressed == KEY_PRESSED)
{
                key_pressed = KEY_RELEASED;
            }
            key_state = KEY_NORMAL;
            break;
        case 0x70: // with e0 - Insert
            if(key_state == KEY_EXTEND && key_pressed == KEY_RELEASED) {
                switch (selected)
                {
                case 0: // chorus
```

```
                    if(chor_amp != 4) chor_amp++;
                    sprintf(display1,"C amp = %d",chor_amp);
                    disp_changed = 1;
                    IOWR_16DIRECT(EFFECTOR+LCHO_AMP,0,chor_amp);
                    IOWR_16DIRECT(EFFECTOR+RCHO_AMP,0,chor_amp);
                    break;
                case 1:
                    if(vib_amp != 4) vib_amp++;// vibrato
                    sprintf(display1,"V amp = %d",vib_amp);
                    disp_changed = 1;
                    IOWR_16DIRECT(EFFECTOR+LVIB_AMP,0,vib_amp);
                    IOWR_16DIRECT(EFFECTOR+RVIB_AMP,0,vib_amp);
                    break;
                case 2:
                    if(dis_clip != 32768) dis_clip <<= 1;// distortion
                    sprintf(display1,"D clip = %d",dis_clip);
                    disp_changed = 1;
                    IOWR_16DIRECT(EFFECTOR+LDIS_CLIP,0,dis_clip);
                    IOWR_16DIRECT(EFFECTOR+RDIS_CLIP,0,dis_clip);
                    break;
                }
                key_pressed = KEY_PRESSED;
            }
            if(key_state == KEY_EXTEND_RELEASE && key_pressed == KEY_PRESSED)
{
                key_pressed = KEY_RELEASED;
            }
            key_state = KEY_NORMAL;
            break;
        case 0x71: //with e0 - Delete
            if(key_state == KEY_EXTEND && key_pressed == KEY_RELEASED) {
                switch (selected)
                {
                case 0: // chorus
                    if(chor_amp != 0) chor_amp--;
                    sprintf(display1,"C amp = %d",chor_amp);
                    disp_changed = 1;
                    IOWR_16DIRECT(EFFECTOR+LCHO_AMP,0,chor_amp);
                    IOWR_16DIRECT(EFFECTOR+RCHO_AMP,0,chor_amp);
                    break;
                case 1:
                    if(vib_amp != 0) vib_amp--;// vibrato
                    sprintf(display1,"V amp = %d",vib_amp);
                    disp_changed = 1;
                    IOWR_16DIRECT(EFFECTOR+LVIB_AMP,0,vib_amp);
                    IOWR_16DIRECT(EFFECTOR+RVIB_AMP,0,vib_amp);
                    break;
                case 2:
                    if(dis_clip != 1) dis_clip >>= 1;// distortion
                    sprintf(display1,"D clip = %d",dis_clip);
                    disp_changed = 1;
                    IOWR_16DIRECT(EFFECTOR+LDIS_CLIP,0,dis_clip);
                    IOWR_16DIRECT(EFFECTOR+RDIS_CLIP,0,dis_clip);
```

```
                    break;
                }
                key_pressed = KEY_PRESSED;
            }
            if(key_state == KEY_EXTEND_RELEASE && key_pressed == KEY_PRESSED)
{
                key_pressed = KEY_RELEASED;
            }
            key_state = KEY_NORMAL;
            break;
        case 0x6C: //with e0 - Home
            if(key_state == KEY_EXTEND && key_pressed == KEY_RELEASED) {
                switch (selected)
                {
                case 0: // chorus
                    if(chor_freq != 15) chor_freq++;
                    sprintf(display1,"C freq = %d",chor_freq);
                    disp_changed = 1;
                    IOWR_16DIRECT(EFFECTOR+LCHO_FREQ,0,chor_freq);
                    IOWR_16DIRECT(EFFECTOR+RCHO_FREQ,0,chor_freq);
                    break;
                case 1:
                    if(vib_freq != 15) vib_freq++;// vibrato
                    sprintf(display1,"V freq = %d",vib_freq);
                    disp_changed = 1;
                    IOWR_16DIRECT(EFFECTOR+LVIB_FREQ,0,vib_freq);
                    IOWR_16DIRECT(EFFECTOR+RVIB_FREQ,0,vib_freq);
                    break;
                case 2:
                    break;
                }
                key_pressed = KEY_PRESSED;
            }
            if(key_state == KEY_EXTEND_RELEASE && key_pressed == KEY_PRESSED)
{
                key_pressed = KEY_RELEASED;
            }
            key_state = KEY_NORMAL;
            break;
        case 0X69: //with e0 - End
            if(key_state == KEY_EXTEND && key_pressed == KEY_RELEASED) {
                switch (selected)
                {
                case 0: // chorus
                    if(chor_freq != 0) chor_freq--;
                    sprintf(display1,"C freq = %d",chor_freq);
                    disp_changed = 1;
                    IOWR_16DIRECT(EFFECTOR+LCHO_FREQ,0,chor_freq);
                    IOWR_16DIRECT(EFFECTOR+RCHO_FREQ,0,chor_freq);
                    break;
                case 1:
                    if(vib_freq != 0) vib_freq--;// vibrato
                    sprintf(display1,"V freq = %d",vib_freq);
```

```
                    disp_changed = 1;
                    IOWR_16DIRECT(EFFECTOR+LVIB_FREQ,0,vib_freq);
                    IOWR_16DIRECT(EFFECTOR+RVIB_FREQ,0,vib_freq);
                    break;
                case 2:
                    break;
                }
                key_pressed = KEY_PRESSED;
            }
            if(key_state == KEY_EXTEND_RELEASE && key_pressed == KEY_PRESSED)
{
                key_pressed = KEY_RELEASED;
            }
            key_state = KEY_NORMAL;
            break;
        case 0x7D: //with e0 - Page Up
            if(key_state == KEY_EXTEND && key_pressed == KEY_RELEASED) {
                switch (selected)
                {
                case 0:
                    if(chor_mix != 15) chor_mix++;
                    sprintf(display1,"C mix = %d",chor_mix);
                    disp_changed = 1;
                    IOWR_16DIRECT(EFFECTOR+LCHO_MIX,0,chor_mix);
                    IOWR_16DIRECT(EFFECTOR+RCHO_MIX,0,chor_mix);
                    break;
                case 1:
                case 2:
                    break;
                }
                key_pressed = KEY_PRESSED;
            }
            if(key_state == KEY_EXTEND_RELEASE && key_pressed == KEY_PRESSED)
{
                key_pressed = KEY_RELEASED;
            }
            key_state = KEY_NORMAL;
            break;
        case 0x7A: //with e0 - Page Down
            if(key_state == KEY_EXTEND && key_pressed == KEY_RELEASED) {
                switch (selected)
                {
                case 0:
                    if(chor_mix != 0) chor_mix--;
                    sprintf(display1,"C mix = %d",chor_mix);
                    disp_changed = 1;
                    IOWR_16DIRECT(EFFECTOR+LCHO_MIX,0,chor_mix);
                    IOWR_16DIRECT(EFFECTOR+RCHO_MIX,0,chor_mix);
                    break;
                case 1:
                case 2:
                    break;
                }
```

```
                key_pressed = KEY_PRESSED;
            }
            if(key_state == KEY_EXTEND_RELEASE && key_pressed == KEY_PRESSED)
{
                key_pressed = KEY_RELEASED;
            }
            key_state = KEY_NORMAL;
            break;
        default:
            key_state = KEY_NORMAL;
            break;
        }

        if(disp_changed == 1) {
            LCD_Init();
            LCD_Show_Text(display1);
            LCD_Line2();
            sprintf(display2,"    %c%c%c %c %d", (dist_on)?'D':'
                     ',(vibr_on)?'V':' ', (chor_on)?'C':' ',
                    (selected == 0)?'C':(selected ==1)?'V':'D',volume-48);
            LCD_Show_Text(display2);
            disp_changed = 0;
        }
   }

  return 0;
}
```

Appendix F

C verification code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sndfile.h>
#include <limits.h>
#include "sin_lookup_table2.h"

#define DEF_DIST_VAL (INT_MAX * 0.001)
#define MAX_Y_BUF 1632
#define MIN_Y_BUF 32
#define MAX_X_BUF 24576
int dist_val;
int inc;

int distortion(int input);
int vibrato(int input, int *buf);
int *vibrato_init(SNDFILE *src);

void error(SNDFILE *reason);
void usage(char *filename);

int main(int argc, char *argv[])
{
   SNDFILE *src, *dst;
   SF_INFO src_info;
   char *src_name, *dst_name;
   char *tmp = NULL;
   int *buf;

   sf_count_t src_cnt, dst_cnt;

   int src_data, dst_data;

   if(argc > 3 || argc < 1) usage(argv[0]);
/* if(argc == 2) {
      dist_val = (int)DEF_DIST_VAL;
   } else {
      dist_val = strtol(argv[2], &tmp, 10);
      if(dist_val == 0 && tmp != NULL) usage(argv[0]);
   }*/
   if(argc != 2) {
      inc = strtol(argv[2], &tmp, 10);
      if(inc == 0 && tmp != NULL) usage(argv[0]);
   } else {
      inc = 1;
   }
   printf("<DEBUG>: MAXIMUM - %d, VAL - %d\n",INT_MAX,dist_val);
   printf("<DEBUG>: -5 % 12 = %d",(-5 % 12));
```

```
   printf("<DEBUG>: before opening the src file\n");
      // open the source file
      src_name = argv[1];
      src = sf_open(src_name, SFM_READ, &src_info);
   printf("<DEBUG>: after opening the src file\n");
      // create the name of dest file
      dst_name = (char *)malloc(strlen(src_name)+4);
      strcpy(dst_name, src_name);
      strcat(dst_name, "_dst\0");
   printf("<DEBUG>: after changing the dst filename\n");

      // open the destination file
      dst = sf_open(dst_name, SFM_WRITE, &src_info);
   printf("<DEBUG>: after opening the dst file\n");

      // set float converting into integer
      sf_command (src, SFC_SET_SCALE_FLOAT_INT_READ, NULL, SF_TRUE);
      sf_command (src, SFC_SET_CLIPPING, NULL, SF_TRUE);
      printf("<DEBUG>: after setting the command\n");

      buf = vibrato_init(src);
      while(1) {
         src_cnt = sf_read_int (src, &src_data, 1);
         if(src_cnt == 0) { sf_perror(src); break;}
         else if(src_cnt < 0) error(src);
         dst_data = vibrato(src_data,buf);
         //dst_data <<= 4;
   //    dst_data = distortion(dst_data);
         dst_cnt = sf_write_int(dst, &dst_data, 1);
         if(dst_cnt <= 0) error(dst);
      }
   printf("<DEBUG>: Complete!\n");
}

int distortion(int input)
{
   if(input > dist_val) return (dist_val * 500);
   else if(input < -dist_val) return (-dist_val * 500);
   else return (input * 500);
}
int vibrato(int input, int *buf)
{
   static int out = 0, in = sin_EFNY2[0]; //empty
   static int sample_num = 0;
   int delay;
   buf[in] = input;
   delay = sin_EFNY2[sample_num];
   delay = delay >> 4;
   out = (in-delay) % (MAX_Y_BUF+1);
   if(out < 0) out += (MAX_Y_BUF+1);
   printf("<DEBUG> delay: %d in: %d, out: %d,
sample_num: %d\n",delay,in,out,sample_num);
   in = (in+1) % (MAX_Y_BUF+1);
```

```
   sample_num = (sample_num+inc) % MAX_X_BUF;
   return buf[out];
}
int *vibrato_init(SNDFILE *src)
{
   static int *buf = (int *)malloc(sizeof(int) * (MAX_Y_BUF+1));
   int src_cnt;
   init_lookup(); // initialize sine table
   src_cnt = sf_read_int (src, buf, sin_EFNY2[0]);
   if(src_cnt == 0) {
      printf("<DEBUG>:");
      sf_perror(src);
      exit(EXIT_FAILURE);
   }
   return buf;


}

void error(SNDFILE *reason)
{
   sf_perror(reason);
   exit(EXIT_FAILURE);
}

void usage(char *filename)
{
   fprintf(stdout,"Usage : %s <file_name> [distortion_factor]\n",filename);
   exit(EXIT_SUCCESS);
}
```