# Octave Keyboard with AutoPlay

Daniel Chen and Vivian Hu

August 26, 2014

The goal of this project was to create a simple one octave keyboard mapped to buttons with the additional capability to autoplay "Kids" by MGMT when a switch is turned on. When the buttons corresponding to notes are pressed, the appropriate notes are played through the speaker, and LEDs which correspond to the keys light up. The LEDs can be disabled using a switch.

This project was created by Vivian Hu and Daniel Chen in Summer 2014 at Dartmouth College for the Digital Electronics (ENGS031/COSC056) course. This report goes over the implementation, design, and usage of the final product, which implements all of these features.

# 1 Introduction: The Problem

The problem that this project solves is the creation of a one-octave keyboard, and the ability to produce certain sounds through circuit logic. An additional issue is representing the song that will be autoplayed.

# 2 Design Solution

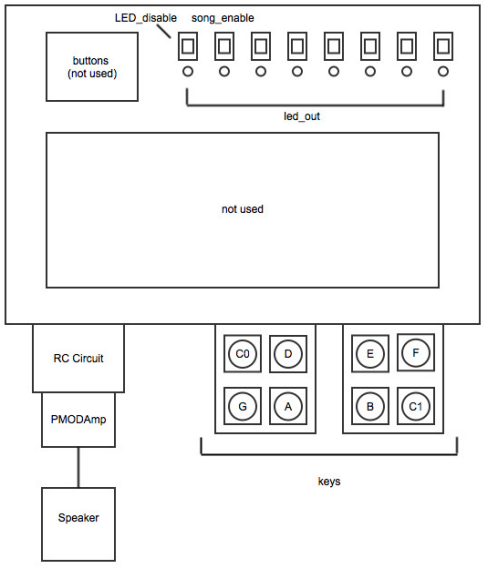
## 2.1 Specifications

The inputs to this circuit are:

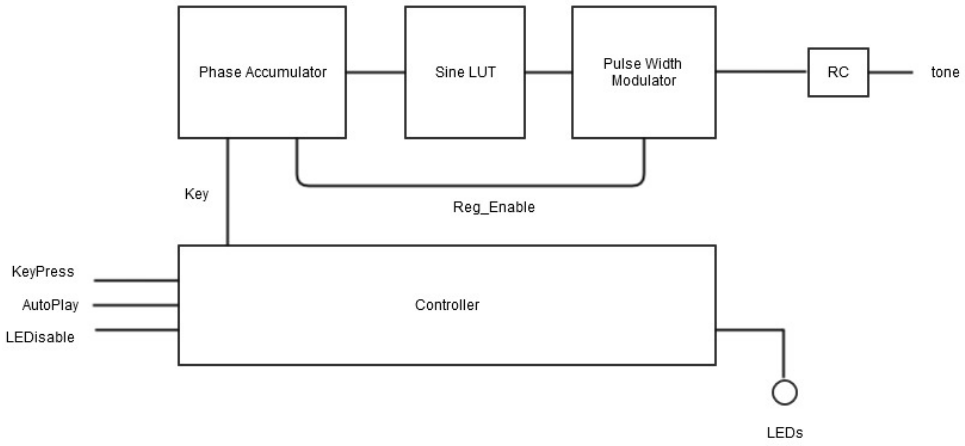
- 8 Buttons that map to the notes to play (bottom right of image to the right).
- An LED disable switch which disables LED output.
- An AutoPlay switch which enables the playing of "Kids" by MGMT.

The outputs to the circuit are:

- 8 LEDs which correspond to the notes being played.
- A speaker which outputs the notes appropriate to the song or specified by the keys.




The picture shown below displays the datapath and control of the circuit. The controller (a finite state machine) takes in all of the inputs and outputs the LEDs. It emits a signal corresponding to the note that needs to be output (a copy of the LED output) which is the Phase Accumulator receives. The data from the Phase Accumulator transfers to the Sine LUT, which goes to the Pulse Width Modulator...



## 2.2 Operating Instructions

### Set up Circuit

To set up the one-octave keyboard circuit, you will need:

- Xilinx ISE Design Suite 14.4
- Digilent Adept
- The source code or the programming file
- 1 x Digilent NEXYS 3 Spartan 6 FPGA
- 2 x 4 button Digilent Button Module
- 1 x Digilent PmodBB
- 1 x Digilent PmodAMP2
- 1 x Speaker

Here are the steps to get the circuit running:

1. (If no bit file) Open the project in Xilinx, generate the programming file.
2. Plug in FPGA (NEXYS 3 Spartan 6) into computer.
3. Insert Digilent Button Modules in the top of JA1 and the top of JB1 on the NEXYS 3.
4. Insert the Digilent PmodBB into JD1.
5. Insert the PmodAMP2 into the top of J4 on the PmodBB.
6. Connect the speaker to J2 on PmodAMP2.
7. Set up the RC circuit on the PmodBB.
8. Open Digilent Adept, select bit file to program the FPGA.

### Playing notes

To play notes, press the buttons on the FPGA corresponding to the notes that you want to play. There are two rows of four buttons. The top four buttons represent low c, d, e and f, while the bottom four buttons are g, a, b and high c.

Only one note can be played at a time (first note pressed takes priority), and user-inputted notes only play when the auto-play switch is off.

**Auto-play "Kids" by MGMT**

To play "Kids" by MGMT, simply flip the AutoPlay switch on. You cannot create additional notes at this time using the buttons.

**Disable LEDs**

To disable the LEDs that light up corresponding to the notes that are pressed, simply turn on the LED disable switch.

## 2.3 Theory of Operation

The key component of our keyboard project is the geneneration of sine waves of varying frequencies using a technique called direct digital synthesis (DDS). To implement this, we use a combination of a phase accumulator and a sine wave lookup table (LUT) to produce wave samples of specific frequencies. In the place of a digital-to-analog converter, we use a pulse width modulator (PWM) to convert these sine wave samples to audio.

The controller of our program is a finite state machine. The controller takes in the user's input and turns it into a single-note output for the Phase Accumulator and the LEDs. It uses the note that was input first. If two notes are both initially pressed at the same exact time, the lower note is used. When the autoplay switch gets turned on, the state machine moves into hard-coded states that play MGMT's "Kids." The song only starts if the user is not currently pressing a key. The song consists of a large number of states that output a time that corresponds to the length of the note or the rest, and waits for either the switch to be turned off (at which point it goes to idle) or for the beat counter to reach it's max (at which point it moves to the next note or rest).

The song itself consists of 3 snippets, which repeat in a particular fashion. Given that the three parts are A, B, and C, the song is a repetition of ACACBCBC. To keep track of this, an additional counter is used to determine whether to go to A or B at the end of C.See the appendix for detailed state diagrams related to the controller.

8 different buttons (button module extensions on the FPGA), or "key" serve as the main "playable" interface of the keyboard. In VHDL, the keys are represented as an 8-bit bus input to the controller, where each bit represents a different note. Because our keyboard is monophonic, the controller checks each bit in succession from low to high C and outputs the first note it detects to the FreqLUT. As a result, only one bit in the 8-bit key_out vector received by ToneFreq-LUT should be high at a time; otherwise, when no button is being pressed, all bits are '0'. This prevents conflicts when multiple buttons are pressed at once. The

controller also takes in an LED_disable input that disables the LEDs and a song_enable that automatically plays back a pre-programmed song (here "Kids" by MGMT).

Key_out is then used as an index of sorts into the ToneFreq-LUT, which contains the pre-calculated phase increment values for each note, which is related to the desired frequency in the following way where N represents the bit size of the accumulator and fclk is the frequency of the clock. We use N=13 here. This increment value is then passed on to the phase accumulator.

The phase accumulator is essentially an incrementer composed of an adder and a register which increases by the increment value every clock cycle. However, although the system is running at a frequency of about 50MHz in order to reduce noise, in actuality the phase accumulator is updating at a frequency of around 10kHz, at the clk10 signal given by the PWMCounter. As a result, clk10 here serves as an enable. The phase produced by the accumulator is then passed as an input address to the SineLUT. However, because our SineLUT only takes an 8-bit phase, only the 8 most significant bits are used.

To obtain wave samples, this value is then used as an index into a SineLUT supplied by the Xilinx Core Generator. Rather than having to generate all the samples of a waveform everytime, the values of a $2^M$ (where M is the bit size of our phase) sample wave are simply stored in block memory (BRAM) for easy access. The SineLUT generates a 10-bit sample for conversion by the PWM.

The PWM maps the amplitude of the signal to a square wave pulse by comparing the sample value (which needs to be first converted from two's complement to unsigned offset binary) to a counter value. When the count is less than the sample, the comparator output is a '1'; otherwise the output is a '0'. In order to send the clk10 enable to the phase accumulator at a 10kHz frequency, the PWM counter also generates a terminal count signal every 5,000 clock cycles.

The signal then passes through an off-board low-pass RC filter that reduces higher frequency signal noise before passing the tone through to the speaker. One importnat thing to note is that the Pmod we used had a 'shutdown' signal that had to be tied high in order to output sound.

Our circuit also features the additional functionality of automatically playing a pre-programed song when given the song_enable signal. Most of this work is done by the controller state-machine (ie which note or sequence to go to next). However, in order to get the song to play at an appropriate speed, we implemented an additional BeatCounter that counts at a rate of about 240 beats per minute. When the signal to autoplay the song is given, the controller sends a count_en signal and a number count_to to the beat counter letting it know how long to "hold" the note before sending the terminal tc_tick back to the controller to move on to the next note/state.

## 2.4 Construction and Debugging

To build the circuit, we began by building and testing the VHDL modules for the PWM. We designed the counter and the comparator and simulated the generation of wave "samples" with a seperate counter in the place of our DDS module. In our testbench, we were able to observe the square wave pulses getting wider as the "sample" increased as expected. From there, we implemented the phase accumulator. Testing was simply a matter of assigning an arbitrary number to the increment signal and confirming the values were adding up correctly. The next step was designing a lookup table with phase increment information for each note frequency and mapping it via the top module to the phase accumulator. We did run into a slight problem here with integer arithmetic. When trying to calculate the increment (as in equation above), dividing the desired frequency by the clock frequency first would always yield 0. To account for this, we simply multiplied the desired frequency by the phase constant ($2^N$) before dividing that product by the clock frequency.

Once the main components were built and mapped together, we implemented the FSM controller, which handled the key and led outputs. From there, using a PmodBB (breadboard), we built a small low-pass RC filter; we ran into a little bit of wiring trouble because we weren't properly wired to ground, but it was pretty easily fixed. We then passed our tone through the RC filter and were able to display our sine waveforms on the oscilloscope. Our signal was definitely a sine wave, but we discovered that our desired frequencies were a power of 10 too large. It turned out this was because we had been testbenching our programs on a 10MHz clock, even though the FPGA was running at 100MHz. Adding a clock divider and clock buffer fixed this issue. After this, we were able to generate tones at the desired frequency, but we were getting a lot of residual high frequency noise. To counter this, we upped our system clock to 50MHz from 10MHz for a higher carrier frequency. This way, even though our PWMCounter was still only sending enable signals to the phase accumulator at 10kHz, the counter itself was running at 50MHz. Adding a larger capacitor helped to significantly reduce the noise, but also the volume of the keybaord itself. Although the noise is barely perceptable now, you can still hear it slightly.

We also ran into a bug where the keyboard would be playing normally and then suddenly everything would stop. We figured some kind of metastability was occuring because we were using switches as "keys". We noticed that the keyboard would usually stop when we held a switch between '0' and '1'. After switching over to buttons, the problem was still occurring, but we remembered that we had forgotten to add debouncers to our buttons. This resolved our issue.

At the very end, once our keyboard was completely playable, we decided to go back in and implement the autoplay functionality. Most of that was done by simply adding more states to the controller. The only

extra component we needed was a "metronome" of sorts in the form of our BeatCounter, which counted at a rate of 4 beats per second, or 240 beats per minute. Testbenching the counter was difficult because it took the simulator so long to generate a couple standard seconds as opposed to the usual nano- or micro- seconds, but we were able to observe terminal count ticks occurring at the right time. The major issue we ran into was that the first note of the song would sometimes be cut short or get dragged out. We resolved this by adding another "idle"-like state that only occured when the $\text{song}_enablewentfromlowtohigh$.

# 3 Evaluation of Design

Our solution is

# 4 Conclusions and Recommendations

The original goal of our project was to simply make a one-octave keyboard that plays all the notes in C major. LEDs would light up when notes were played, unless an "LED disable" switch was turned on.

At the end, we were not only able to accomplish our original goal of the simple keyboard but also to add an additional autoplay feature that played "Kids" by MGMT.

For future groups looking to create this project, we would recommend that they really take the time to understand the operation of the circuit before jumping into the project. Another important consideration is to remember to synchronize the inputs and debounce the buttons throughout the project's creation.

# 5 Acknowledgments

We would like to thank Eric Hansen and Dave Picard for their support and mentorship throughout not only this project but also the course. We would also like to thank the other students of Digital Electronics as well as the TAs. We'd also like to thank MGMT for an awesome song that conveniently contains itself to a single octave.

5.1 and 5.2 list the contributions for each partner. Although both partners had their hands in most aspects of the project, some components generally had one partner who was more involved in its creation.

## 5.1 Vivian's Contributions

- Circuit Design
- DDS, PWM, FreqLUT, PlayCount
- RC Circuit
- Block Diagrams

## 5.2 Daniel's Contributions

- Majority of the controller, including auto-play
- Design and creation of song auto-play, state diagram
- Top level
- Diagrams
- LaTeX for report
- Git creation/management

# 6 References


[1] Cordesses, Lionel. "Direct Digital Synthesis: A Tool for Periodic Wave Generation." *IEEE Signal Processing Magazine.* IEEE, July 2004. Web.

[2] Hansen, Eric. *Lab Assignment 1.* N.p.: ENGS128 - Advanced Digital System Design, Spring 2011. PDF.

[3] *Introduction to Direct Digital Synthesis.* San Jose: Intel Corporation, June 1991. PDF.

[4] Palacheria, Amar. *Using PWM to Generate Analog Output.* N.p.: Microchip Technology Inc., 1997. PDF.

# 7 Appendix

## List of Figures



## List of Tables

## 7.1 System level diagrams

### 7.1.1 Front Panel

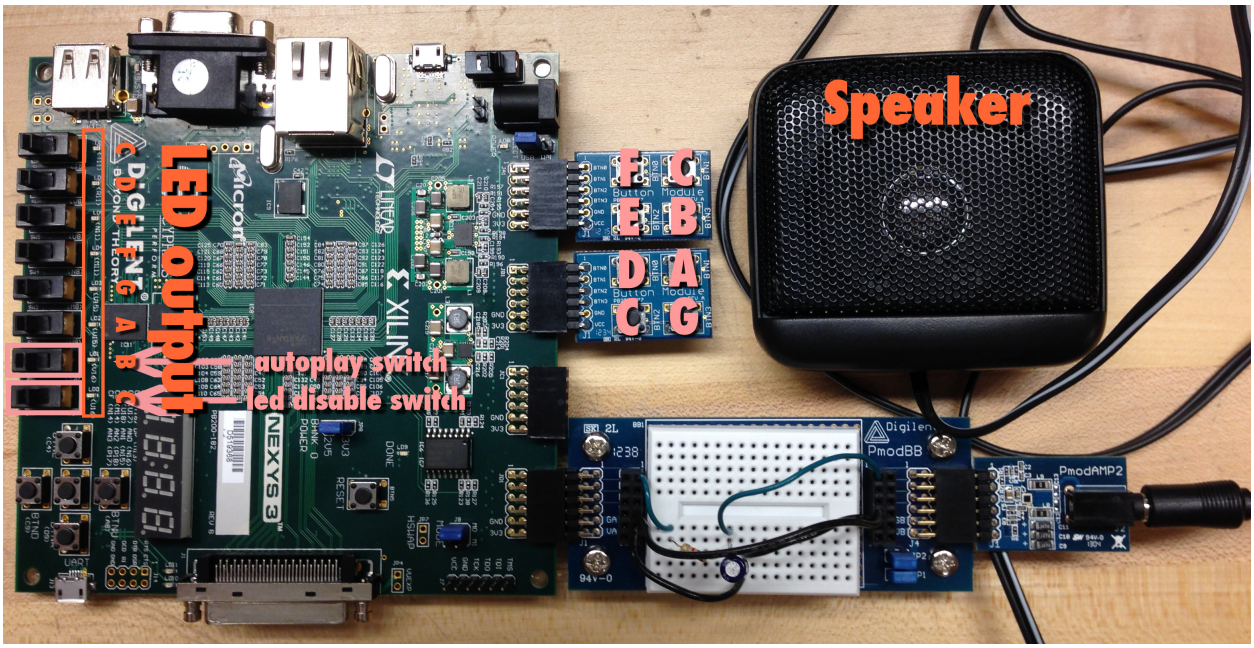



Figure 1: Annotated Digital Photo of Project

### 7.1.2 Block Diagram

### 7.1.3 Schematic Diagram

### 7.1.4 Package Map

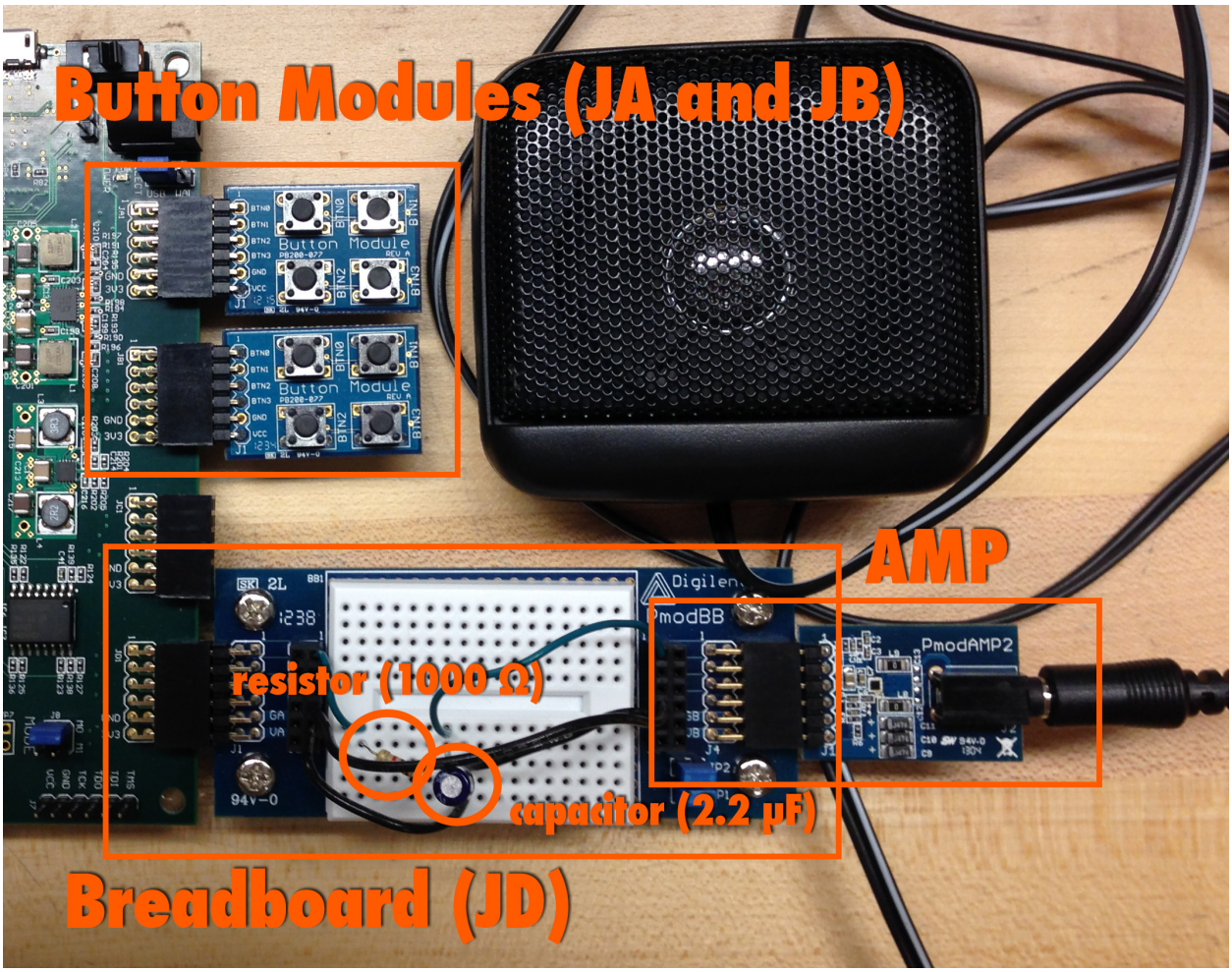



Figure 2: Package Map

### 7.1.5 External Components

Table 1: Parts List

| Reference | Quantity | Description |
|---|---|---|
| Nexys3 | 1 | Digilent Nexys3 board |
| PmodBTN | 2 | Digilent PmodBTN Push Button. 4 buttons |
| PmodAMP2 | 1 | Digilent Amplifier for monophonic output |
| PmodBB | 1 | Digilent Bread board |
| Speaker | 1 | Any generic speaker with standard input |

## 7.2 Programmed Logic

### 7.2.1 State Diagrams

For simplicity, the state machine for this program is represented in two state diagrams. The first, "User Play State Diagram," represents the state machine for when the user is given input through the buttons. The second, "Autoplay State Diagram," represents the state machine when the autoplay mode is on. Both state diagrams share an idle state. If the song enable switch if off, the "User Play State Diagram" should be used. If it is turned on, the "Autoplay State Diagram" should be used.

The program starts at the idle state in the "User Play State Diagram." If the song enable switch is turned on, the state jumps to autoidle in the "Autoplay State Diagram." Otherwise, the state changes depending on the user's input.

**Default Values if unspecified:**
next_state <= curr_state;
out <= (others => '0');
key_out <= output;
count_out <= "0001";
repeat_tick <= '0';
beat_en <= '0';

Figure 3: State Diagram Defaults

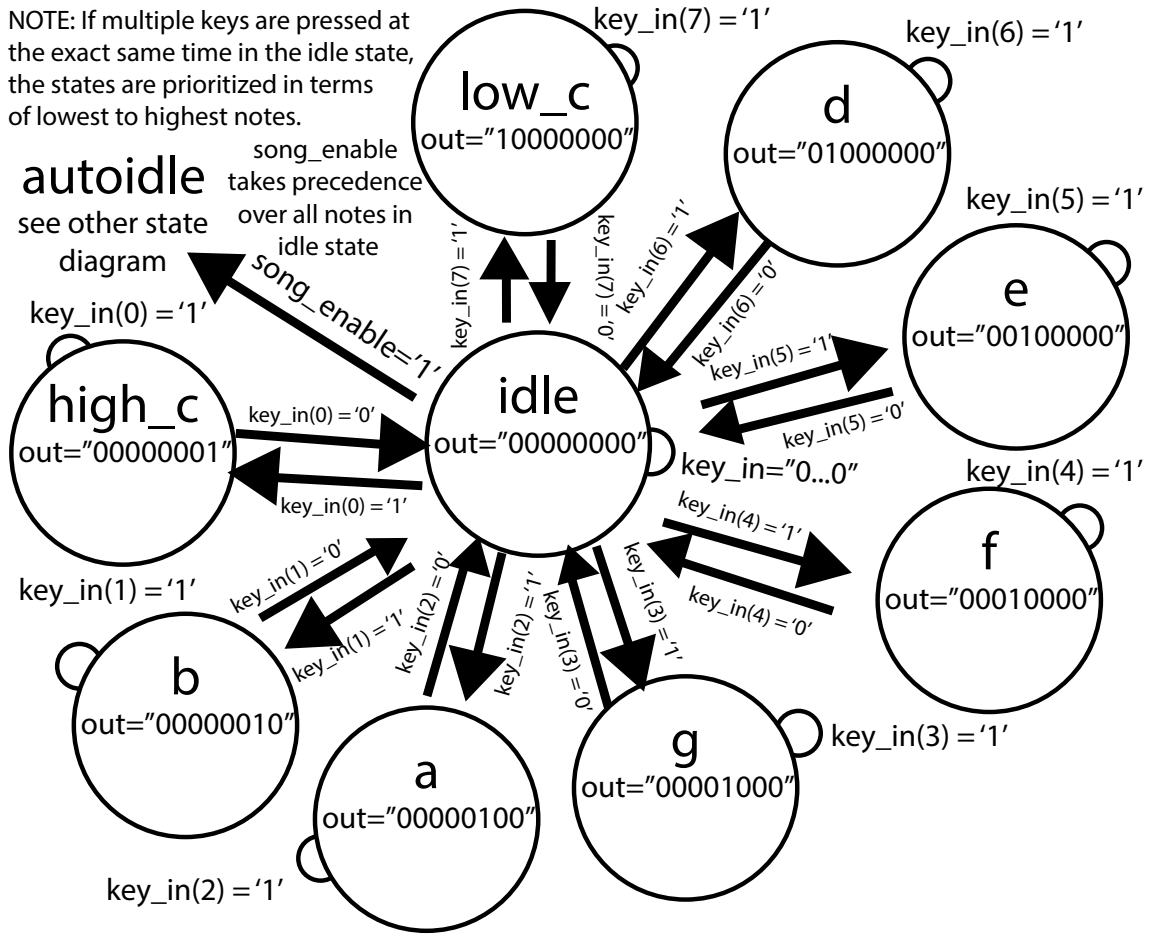



Figure 4: User Play State Diagram

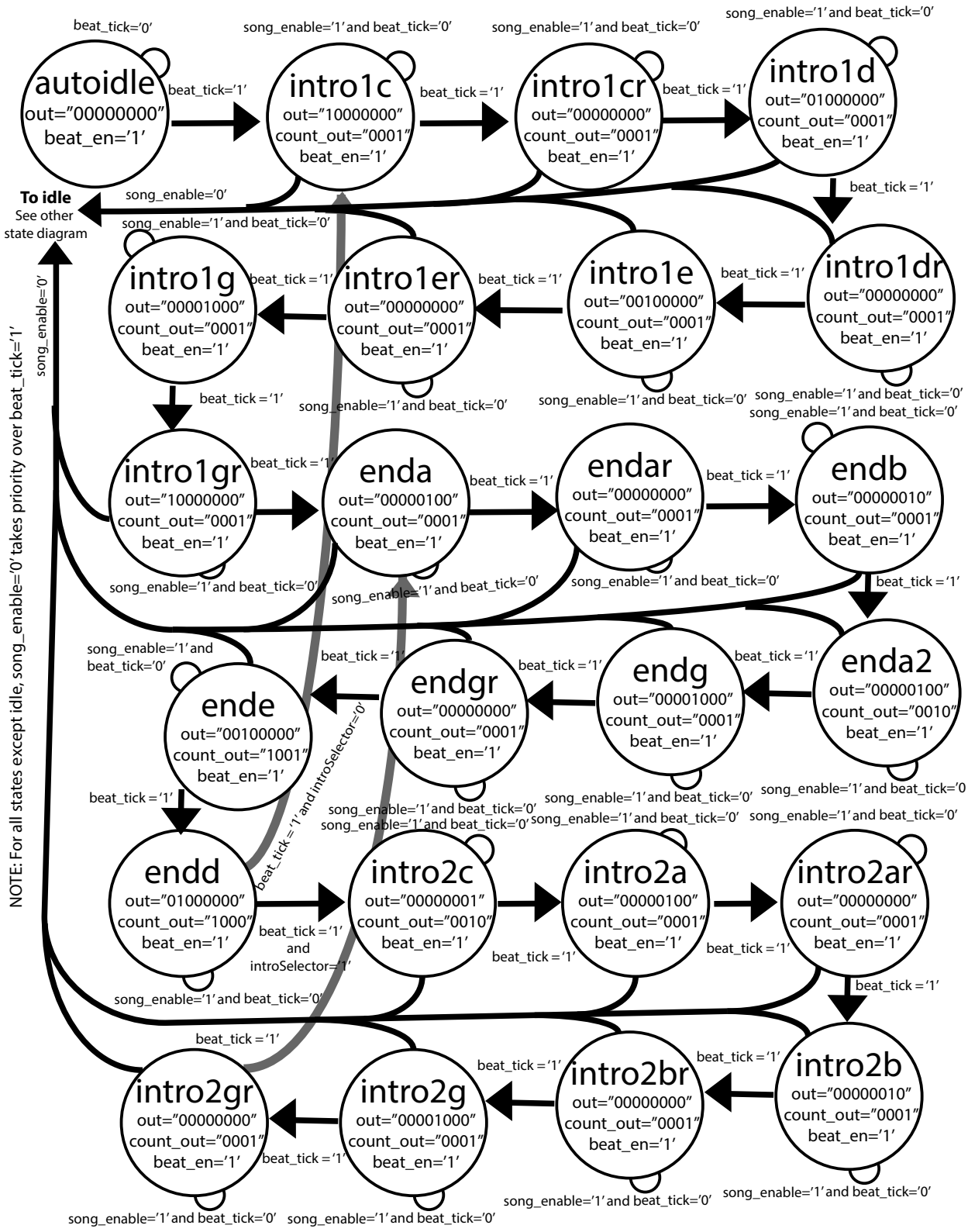



Figure 5: Autoplay State Diagram

### 7.2.2 VHDL Code

Figure 6: OctaveKeyboardTop.vhd

```
1  ----------------------------------------------------------------------------------
2  -- Company: ENGS 31
3  -- Engineer: Vivian Hu, Daniel Chen
4  --
5  -- Create Date: 19:58:26 08/12/2014
6  -- Design Name:
7  -- Module Name: OctaveKeyboardTop - Behavioral
8  -- Project Name: OctaveKeyboard
9  -- Target Devices:
10 -- Tool versions:
11 -- Description: Top level VHDL module for keyboard
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 ----------------------------------------------------------------------------------
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22 use IEEE.NUMERIC_STD.ALL;
23
24 library UNISIM;
25 use UNISIM.VComponents.all;
26
27 entity OctaveKeyboardTop is
28     Generic (   ACCUMSIZE    : integer := 13;     -- phase accumulator bitsize
29                 INDEXSIZE    : integer := 8;        -- SineLUT index bitsize
30                 LUTOUT       : integer := 10);     -- SineLUT out bitsize
31
32     Port (  clk          : in   STD_LOGIC;
33             keys         : in   STD_LOGIC_VECTOR (7 downto 0);
34             led_disable  : in   STD_LOGIC;
35             song_enable  : in   STD_LOGIC;
36             tone         : out  STD_LOGIC;
37             shutdown     : out  STD_LOGIC;
38             led_out      : out  STD_LOGIC_VECTOR (7 downto 0));
39 end OctaveKeyboardTop;
40
41 architecture Behavioral of OctaveKeyboardTop is
42
43     -- signals for 100MHz to 50Mhz clk divider
44     signal clk_en           : std_logic := '0';
45     signal slowclk          : std_logic;
46
```

```
47      signal led_disable_sync : std_logic := '0';
48      signal song_enable_sync : std_logic := '0';
49
50      -- mapping signals
51      signal step : std_logic_vector(ACCUMSIZE-1 downto 0) := (others => '0');
52      signal controllerKeys : std_logic_vector(7 downto 0) := (others => '0');
53      signal phase : std_logic_vector(INDEXSIZE-1 downto 0) := (others => '0');
54      signal lutfreq : std_logic_vector(15 downto 0) := (others => '0');
55      signal reg_en : std_logic := '0';
56      signal tempo_en : std_logic := '0';
57      signal keyDB : std_logic_vector(7 downto 0) := (others => '0');
58      signal countDone : std_logic := '0';
59      signal count : std_logic_vector(3 downto 0) := (others => '0');
60
61      -- BEGIN component declarations
62      COMPONENT Controller
63          PORT (  clk             : in      STD_LOGIC;
64                  key_in          : in      STD_LOGIC_VECTOR (7 downto 0);
65                  led_disable     : in      STD_LOGIC;
66                  song_enable     : in      STD_LOGIC;
67                  beat_tick       : in      STD_LOGIC;
68                  beat_en         : out     STD_LOGIC;
69                  count_out       : out     STD_LOGIC_VECTOR(3 downto 0);
70                  key_out         : out     STD_LOGIC_VECTOR (7 downto 0);
71                  led_out         : out     STD_LOGIC_VECTOR (7 downto 0));
72      END COMPONENT;
73
74      COMPONENT FreqLUT
75          PORT (  clk         : in      STD_LOGIC;
76                  key_in      : in      STD_LOGIC_VECTOR (7 downto 0);
77                  increment   : out     STD_LOGIC_VECTOR (ACCUMSIZE-1 downto 0));
78      END COMPONENT;
79
80      COMPONENT DDS
81          PORT (  clk     : in      STD_LOGIC;
82                  clk10   : in     STD_LOGIC;
83                  step    : in    STD_LOGIC_VECTOR(ACCUMSIZE-1 downto 0);
84                  phase   : out    STD_LOGIC_VECTOR(INDEXSIZE-1 downto 0));
85      END COMPONENT;
86
87      COMPONENT PWM
88          PORT (  clk         : in      STD_LOGIC;
89                  sample      : in      STD_LOGIC_VECTOR(LUTOUT-1 downto 0);
90                  slowclk     : out     STD_LOGIC;
91                  pulse       : out     STD_LOGIC);
92      END COMPONENT;
93
94      COMPONENT SinLUT
95          PORT (  aclk                : in  STD_LOGIC;
96                  s_axis_phase_tvalid : in  STD_LOGIC;
97                  s_axis_phase_tdata  : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
98                  m_axis_data_tvalid  : out STD_LOGIC;
```

```
99                  m_axis_data_tdata   : out STD_LOGIC_VECTOR(15 DOWNTO 0));
100     END COMPONENT;
101
102     COMPONENT debounce
103         PORT (  clk      : in     STD_LOGIC;
104                 switch   : in     STD_LOGIC;
105                 dbswitch : out     STD_LOGIC);
106     END COMPONENT;
107
108     COMPONENT PlayCount is
109         PORT (  clk             : in      STD_LOGIC;
110                 count_en        : in      STD_LOGIC;
111                 count_to        : in      STD_LOGIC_VECTOR (3 downto 0);
112                 tc_tick         : out     STD_LOGIC);
113     END COMPONENT;
114     -- END component declarations
115
116 begin
117
118     -- synchronizer for auto-play-song switch input
119     SynchronizeSwitches: process(clk)
120     begin
121         if rising_edge(clk) then
122             led_disable_sync <= led_disable;
123             song_enable_sync <= song_enable;
124         end if;
125     end process SynchronizeSwitches;
126
127     -- slow main clock down by half (here: 100Mhz to 50Mhz)
128     clkDivider: process(clk)
129     begin
130         if rising_edge(clk) then
131             clk_en <= NOT(clk_en);
132         end if;
133     end process clkDivider;
134
135     -- map signals
136     shutdown <= '1';            -- tie shutdown signal high for speaker pmod
137
138     slowclk_buf: BUFG
139         Port map ( I => clk_en,
140                    O => slowclk );
141
142
143     debouncer0: debounce
144         Port map (  clk => slowclk,
145                     switch => keys(0),
146                     dbswitch => keyDB(0) );
147
148     debouncer1: debounce
149         Port map (  clk => slowclk,
```

```
150                     switch => keys(1),
151                     dbswitch => keyDB(1) );
152
153     debouncer2: debounce
154         Port map (  clk => slowclk,
155                     switch => keys(2),
156                     dbswitch => keyDB(2) );
157
158     debouncer3: debounce
159         Port map (  clk => slowclk,
160                     switch => keys(3),
161                     dbswitch => keyDB(3) );
162
163     debouncer4: debounce
164         Port map (  clk => slowclk,
165                     switch => keys(4),
166                     dbswitch => keyDB(4) );
167
168     debouncer5: debounce
169         Port map (  clk => slowclk,
170                     switch => keys(5),
171                     dbswitch => keyDB(5) );
172
173     debouncer6: debounce
174         Port map (  clk => slowclk,
175                     switch => keys(6),
176                     dbswitch => keyDB(6) );
177
178     debouncer7: debounce
179         Port map (  clk => slowclk,
180                     switch => keys(7),
181                     dbswitch => keyDB(7) );
182
183     KeyControl: Controller
184         PORT MAP (  clk         => slowclk,
185                     key_in         => keys, -- change to keys if simulating
186                     led_disable => led_disable_sync,
187                     song_enable => song_enable_sync,
188                     beat_tick     => countdone,
189                     beat_en        => tempo_en,
190                     count_out    => count,
191                     key_out     => controllerKeys,
192                     led_out        => led_out);
193
194     KeyFrequencies: FreqLUT
195         PORT MAP (  clk         => slowclk,
196                     key_in         => controllerKeys,
197                     increment     => step);
198
199     PhaseAccum: DDS
200         PORT MAP (  clk         => slowclk,
```

```
201                     clk10        => reg_en,
202                     step        => step,
203                     phase        => phase);
204 
205     PulseWM: PWM
206         PORT MAP (  clk            => slowclk,
207                     sample        => lutfreq(9 downto 0),
208                     slowclk        => reg_en,
209                     pulse        => tone);
210 
211     SinFreqs: SinLUT
212         PORT MAP (  aclk                 => slowclk,
213                     s_axis_phase_tvalid => '1',
214                     s_axis_phase_tdata     => phase,
215                     m_axis_data_tvalid     => open,
216                     m_axis_data_tdata     => lutfreq);
217 
218     kidsCounter: PlayCount
219         PORT MAP    ( clk => slowclk,
220                       count_en => tempo_en,
221                       count_to => count,
222                       tc_tick => countDone);
223 
224 end Behavioral;
```

Figure 7: Controller.vhd

```
----------------------------------------------------------------------------------
-- Company: ENGS041 14X
-- Engineer: Vivian Hu and Daniel Chen
--
-- Create Date:    14:49:26 08/11/2014
-- Design Name: Controller FSM
-- Module Name:    Controller - Behavioral
-- Project Name: Octave Keyboard
-- Target Devices: Spartan 6
-- Tool versions:
-- Description: Basic controller which converts to monotone.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Controller is
    Port (      clk         : in  STD_LOGIC;
                key_in      : in  STD_LOGIC_VECTOR(7 downto 0);
                led_disable : in  STD_LOGIC;
                song_enable : in  STD_LOGIC;
                beat_tick   : in  STD_LOGIC;
                beat_en     : out STD_LOGIC;
                count_out   : out STD_LOGIC_VECTOR(3 downto 0);
                key_out     : out STD_LOGIC_VECTOR(7 downto 0);
                led_out     : out STD_LOGIC_VECTOR(7 downto 0));
end Controller;

architecture Behavioral of Controller is
    type statetype is (idle, low_c, d, e, f, g, a, b, high_c, autoidle,
        intro1c, intro1cr, intro1d, intro1dr, intro1e, intro1er, intro1g, intro1gr,
        enda, endar, endb, enda2, endg, endgr, ende, endd,
        intro2c, intro2a, intro2ar, intro2b, intro2br, intro2g, intro2gr
        );
    signal curr_state, next_state : statetype := idle;
    signal output : STD_LOGIC_VECTOR (7 downto 0) := (others => '0');
    signal reps : STD_LOGIC := '1';
    signal introSelector : STD_LOGIC := '0';
    signal repeat_tick : STD_LOGIC := '0';
begin

    StateUpdate: process(clk)
```

```
50      begin
51          if rising_edge(clk) then
52              curr_state <= next_state;
53          end if;
54      end process StateUpdate;
55
56      CombLogic: process(curr_state, next_state, key_in, led_disable, output,
57                         beat_tick, song_enable, introSelector)
58      begin
59          -- defaults
60          next_state <= curr_state;
61          output <= (others => '0');
62          key_out <= output;
63          count_out <= "0001";
64          repeat_tick <= '0';
65          beat_en <= '0';
66
67          if (led_disable = '1') then
68              led_out <= (others => '0');
69          else
70              led_out <= output;
71          end if;
72
73          case curr_state is
74
75              when idle =>
76
77                  if song_enable = '1' then
78                      next_state <= autoidle;
79                  elsif key_in(7) = '1' then
80                      next_state <= low_c;
81                  elsif key_in(6) = '1' then
82                      next_state <= d;
83                  elsif key_in(5) = '1' then
84                      next_state <= e;
85                  elsif key_in(4) = '1' then
86                      next_state <= f;
87                  elsif key_in(3) = '1' then
88                      next_state <= g;
89                  elsif key_in(2) = '1' then
90                      next_state <= a;
91                  elsif key_in(1) = '1' then
92                      next_state <= b;
93                  elsif key_in(0) = '1' then
94                      next_state <= high_c;
95                  else
96                      next_state <= idle;
97                  end if;
98
99              when low_c =>
100                 output <= "10000000";
101                 if key_in(7) = '0' then
```

```
102                     next_state <= idle;
103                 end if;
104 
105             when d =>
106                 output <= "01000000";
107                 if key_in(6) = '0' then
108                     next_state <= idle;
109                 end if;
110 
111             when e =>
112                 output <= "00100000";
113                 if key_in(5) = '0' then
114                     next_state <= idle;
115                 end if;
116 
117             when f =>
118                 output <= "00010000";
119                 if key_in(4) = '0' then
120                     next_state <= idle;
121                 end if;
122 
123             when g =>
124                 output <= "00001000";
125                 if key_in(3) = '0' then
126                     next_state <= idle;
127                 end if;
128 
129             when a =>
130                 output <= "00000100";
131                 if key_in(2) = '0' then
132                     next_state <= idle;
133                 end if;
134 
135             when b =>
136                 output <= "00000010";
137                 if key_in(1) = '0' then
138                     next_state <= idle;
139                 end if;
140 
141             when high_c =>
142                 output <= "00000001";
143                 if key_in(0) = '0' then
144                     next_state <= idle;
145                 end if;
146 
147             when autoidle =>
148                 beat_en <= '1';
149                 output <= (others => '0');
150                 if (beat_tick = '1') then
151                     next_state <= intro1c;
152                 end if;
153 
```

```
154             when intro1c =>
155                 beat_en <= '1';
156                 output <= "10000000";
157                 count_out <= "0001";
158                 if (song_enable = '0') then
159                     next_state <= idle;
160                 elsif(beat_tick = '1') then
161                     next_state <= intro1cr;
162                 end if;
163 
164 
165             when intro1cr =>
166                 beat_en <= '1';
167                 output <= "00000000";
168                 count_out <= "0001";
169                 if (song_enable = '0') then
170                     next_state <= idle;
171                 elsif(beat_tick = '1') then
172                     next_state <= intro1d;
173                 end if;
174 
175 
176             when intro1d =>
177                 beat_en <= '1';
178                 output <= "01000000";
179                 count_out <= "0001";
180                 if (song_enable = '0') then
181                     next_state <= idle;
182                 elsif(beat_tick = '1') then
183                     next_state <= intro1dr;
184                 end if;
185 
186 
187             when intro1dr =>
188                 beat_en <= '1';
189                 output <= "00000000";
190                 count_out <= "0001";
191                 if (song_enable = '0') then
192                     next_state <= idle;
193                 elsif(beat_tick = '1') then
194                     next_state <= intro1e;
195                 end if;
196 
197 
198             when intro1e =>
199                 beat_en <= '1';
200                 output <= "00100000";
201                 count_out <= "0001";
202                 if (song_enable = '0') then
203                     next_state <= idle;
204                 elsif(beat_tick = '1') then
205                     next_state <= intro1er;
```

```
206                 end if;
207
208
209             when intro1er =>
210                 beat_en <= '1';
211                 output <= "00000000";
212                 count_out <= "0001";
213                 if (song_enable = '0') then
214                     next_state <= idle;
215                 elsif(beat_tick = '1') then
216                     next_state <= intro1g;
217                 end if;
218
219
220             when intro1g =>
221                 beat_en <= '1';
222                 output <= "00001000";
223                 count_out <= "0001";
224                 if (song_enable = '0') then
225                     next_state <= idle;
226                 elsif(beat_tick = '1') then
227                     next_state <= intro1gr;
228                 end if;
229
230
231             when intro1gr =>
232                 beat_en <= '1';
233                 output <= "00000000";
234                 count_out <= "0001";
235                 if (song_enable = '0') then
236                     next_state <= idle;
237                 elsif(beat_tick = '1') then
238                     next_state <= enda;
239                 end if;
240
241
242             when enda =>
243                 beat_en <= '1';
244                 output <= "00000100";
245                 count_out <= "0001";
246                 if (song_enable = '0') then
247                     next_state <= idle;
248                 elsif(beat_tick = '1') then
249                     next_state <= endar;
250                 end if;
251
252
253             when endar =>
254                 beat_en <= '1';
255                 output <= "00000000";
256                 count_out <= "0001";
257                 if (song_enable = '0') then
```

```
258                     next_state <= idle;
259                 elsif(beat_tick = '1') then
260                     next_state <= endb;
261                 end if;
262 
263 
264             when endb =>
265                 beat_en <= '1';
266                 output <= "00000010";
267                 count_out <= "0001";
268                 if (song_enable = '0') then
269                     next_state <= idle;
270                 elsif(beat_tick = '1') then
271                     next_state <= enda2;
272                 end if;
273 
274 
275             when enda2 =>
276                 beat_en <= '1';
277                 output <= "00000100";
278                 count_out <= "0010";
279                 if (song_enable = '0') then
280                     next_state <= idle;
281                 elsif(beat_tick = '1') then
282                     next_state <= endg;
283                 end if;
284 
285 
286             when endg =>
287                 beat_en <= '1';
288                 output <= "00001000";
289                 count_out <= "0001";
290                 if (song_enable = '0') then
291                     next_state <= idle;
292                 elsif(beat_tick = '1') then
293                     next_state <= endgr;
294                 end if;
295 
296 
297             when endgr =>
298                 beat_en <= '1';
299                 output <= "00000000";
300                 count_out <= "0001";
301                 if (song_enable = '0') then
302                     next_state <= idle;
303                 elsif(beat_tick = '1') then
304                     next_state <= ende;
305                 end if;
306 
307 
308             when ende =>
309                 beat_en <= '1';
```

```
310                 output <= "00100000";
311                 count_out <= "1001";
312                 if (song_enable = '0') then
313                     next_state <= idle;
314                 elsif(beat_tick = '1') then
315                     next_state <= endd;
316                 end if;
317 
318 
319             when endd =>
320                 beat_en <= '1';
321                 output <= "01000000";
322                 count_out <= "1000";
323 
324                 if (beat_tick = '1') then
325                     repeat_tick <= '1';
326 
327                     if (introSelector = '0') then
328                         next_state <= intro1c;
329                     else
330                         next_state <= intro2c;
331                     end if;
332 
333                 elsif (song_enable = '0') then
334                     next_state <= idle;
335 
336                 end if;
337 
338             when intro2c =>
339                 beat_en <= '1';
340                 output <= "00000001";
341                 count_out <= "0010";
342                 if (song_enable = '0') then
343                     next_state <= idle;
344                 elsif(beat_tick = '1') then
345                     next_state <= intro2a;
346                 end if;
347 
348 
349             when intro2a =>
350                 beat_en <= '1';
351                 output <= "00000100";
352                 count_out <= "0001";
353                 if (song_enable = '0') then
354                     next_state <= idle;
355                 elsif(beat_tick = '1') then
356                     next_state <= intro2ar;
357                 end if;
358 
359 
360             when intro2ar =>
361                 beat_en <= '1';
```

```
362                 output <= "00000000";
363                 count_out <= "0001";
364                 if (song_enable = '0') then
365                     next_state <= idle;
366                 elsif(beat_tick = '1') then
367                     next_state <= intro2b;
368                 end if;
369
370
371             when intro2b =>
372                 beat_en <= '1';
373                 output <= "00000010";
374                 count_out <= "0001";
375                 if (song_enable = '0') then
376                     next_state <= idle;
377                 elsif(beat_tick = '1') then
378                     next_state <= intro2br;
379                 end if;
380
381
382             when intro2br =>
383                 beat_en <= '1';
384                 output <= "00000000";
385                 count_out <= "0001";
386                 if (song_enable = '0') then
387                     next_state <= idle;
388                 elsif(beat_tick = '1') then
389                     next_state <= intro2g;
390                 end if;
391
392
393             when intro2g =>
394                 beat_en <= '1';
395                 output <= "00001000";
396                 count_out <= "0001";
397                 if (song_enable = '0') then
398                     next_state <= idle;
399                 elsif(beat_tick = '1') then
400                     next_state <= intro2gr;
401                 end if;
402
403
404             when intro2gr =>
405                 beat_en <= '1';
406                 output <= "00000000";
407                 count_out <= "0001";
408                 if (song_enable = '0') then
409                     next_state <= idle;
410                 elsif(beat_tick = '1') then
411                     next_state <= enda;
412                 end if;
413
```

```
414             when others =>
415                 next_state <= idle;
416 
417         end case;
418 
419     end process CombLogic;
420 
421     RepeatCounter: process(clk, repeat_tick, reps, introselector)
422     begin
423         if (rising_edge(clk)) then
424             if (repeat_tick = '1') then
425                 if (reps = '1') then
426                     introSelector <= not introSelector;
427                     reps <= '0';
428                 else
429                     reps <= not reps;
430                 end if;
431             end if;
432         end if;
433     end process RepeatCounter;
434 
435 end Behavioral;
```

Figure 8: DDS.vhd

```
----------------------------------------------------------------------------------
-- Company: ENGS031 14X
-- Engineer: Vivian Hu and Daniel Chen
--
-- Create Date:    14:29:30 08/11/2014
-- Design Name: Octave Keyboard
-- Module Name:    DDS - Behavioral
-- Project Name: Octive Keyboard
-- Target Devices: Spartan 6
-- Tool versions:
-- Description: Direct Digital Synthesis
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity DDS is
     Generic (  ACCUMSIZE    : integer := 13;
                INDEXSIZE    : integer := 8;
                CLKFREQ      : integer := 100000000);

    Port (  clk     : in  STD_LOGIC;
            step    : in  STD_LOGIC_VECTOR(ACCUMSIZE-1 downto 0);
            clk10   : in  STD_LOGIC;
            phase   : out STD_LOGIC_VECTOR(INDEXSIZE-1 downto 0));
end DDS;

architecture Behavioral of DDS is
    signal curr_phase : unsigned(ACCUMSIZE-1 downto 0) := (others => '0');
begin
    AccumPhase: process(clk, clk10)
    begin
        if (rising_edge(clk)) then
            if (clk10 = '1') then
                curr_phase <= curr_phase + unsigned(step);
            end if;
        end if;
    end process AccumPhase;

    phase <= std_logic_vector(curr_phase(ACCUMSIZE-1 downto ACCUMSIZE-INDEXSIZE));
end Behavioral;
```

Figure 9: PWM.vhd

```
1  ----------------------------------------------------------------------------------
2  -- Company:
3  -- Engineer:
4  --
5  -- Create Date:    10:13:11 08/13/2014
6  -- Design Name:
7  -- Module Name:    PWM - Behavioral
8  -- Project Name:
9  -- Target Devices:
10 -- Tool versions:
11 -- Description:
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 ----------------------------------------------------------------------------------
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22 use IEEE.NUMERIC_STD.ALL;
23
24 entity PWM is
25      Generic (  LUTOUT    : integer := 10;
26                 CLKFREQ     : integer := 100000000);
27
28     Port (  clk : in  STD_LOGIC;
29             sample : in  STD_LOGIC_VECTOR(LUTOUT-1 downto 0);
30             slowclk : out STD_LOGIC;
31             pulse : out  STD_LOGIC);
32 end PWM;
33
34 architecture Behavioral of PWM is
35     signal count : unsigned(13 downto 0) := (others => '0');
36     signal offset : unsigned(LUTOUT-1 downto 0) := (others => '0');
37     constant max : unsigned(13 downto 0) := "01001110001000";
38 begin
39
40     PWM: process(clk, sample)
41     begin
42         -- SinLUT output from two's complement to unsigned offset binary
43         offset <= unsigned(not sample(LUTOUT-1) & sample(LUTOUT-2 downto 0));
44
45         if (rising_edge(clk)) then
46
47             -- counter running at 50Mhz, but register updating at 10khz
48             count <= count + 1;
49
```

```
50              if (count(9 downto 0) < offset) then
51                  pulse <= '1';
52              else
53                  pulse <= '0';
54              end if;
55
56              if (count = max) then
57                  slowclk <= '1';
58                  count <= (others => '0');
59              else
60                  slowclk <= '0';
61              end if;
62          end if;
63      end process PWM;
64  end Behavioral;
```

Figure 10: PlayCount.vhd

```
1  ----------------------------------------------------------------------------------
2  -- Company:
3  -- Engineer:
4  --
5  -- Create Date:    11:40:24 08/16/2014
6  -- Design Name:
7  -- Module Name:    PlayCount - Behavioral
8  -- Project Name:
9  -- Target Devices:
10 -- Tool versions:
11 -- Description:
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 ----------------------------------------------------------------------------------
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22 use IEEE.NUMERIC_STD.ALL;
23
24 entity PlayCount is
25     Port (  clk : in  STD_LOGIC;
26             count_en : in STD_LOGIC;
27             count_to : in  STD_LOGIC_VECTOR (3 downto 0);
28             tc_tick : out  STD_LOGIC);
29 end PlayCount;
30
31 architecture Behavioral of PlayCount is
32     constant QRTR_CLK_DIV    : integer := 12500000;
33     signal   clkcount    : integer := 0;
34     signal   count        : unsigned(3 downto 0) := "0001";
35 begin
36
37     Qrtr_Sec_Count: process(clk)
38     begin
39         if (rising_edge(clk)) then
40
41             if (count_en = '1') then
42                 tc_tick <= '0';
43
44                 if (clkcount = QRTR_CLK_DIV - 1) then
45                     if (count = unsigned(count_to)) then
46                         tc_tick <= '1';
47                         count <= "0001";
48                     else
49                         count <= count + 1;
```

```
50                      end if;
51
52                      clkcount <= 0;
53                  else
54                      clkcount <= clkcount + 1;
55                  end if;
56              else
57                  tc_tick <= '0';
58              end if;
59          end if;
60      end process Qrtr_Sec_Count;
61  end Behavioral;
```

Figure 11: FreqLUT.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    15:30:42 08/11/2014
-- Design Name:
-- Module Name:    FreqLUT - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity FreqLUT is
     Generic (  ACCUMSIZE : integer := 13;
                CLKFREQ   : integer := 10000);

    Port ( clk : in  STD_LOGIC;
           key_in : in  STD_LOGIC_VECTOR (7 downto 0);
           increment : out  STD_LOGIC_VECTOR (ACCUMSIZE-1 downto 0));
end FreqLUT;

architecture Behavioral of FreqLUT is
    constant PHASECONSTANT : integer := 2**ACCUMSIZE;
    constant LOWC : integer := 262;
    constant D : integer := 294;
    constant E : integer := 330;
    constant F : integer := 349;
    constant G : integer := 392;
    constant A : integer := 440;
    constant B : integer := 494;
    constant HIGHC : integer := 523;
begin

    getIncrement: process(key_in)
    begin

        if (key_in(7) = '1') then
            increment <= std_logic_vector(to_unsigned(LOWC * PHASECONSTANT / CLKFREQ, ACCUMSIZE));
```

```
50          elsif (key_in(6) = '1') then
51              increment <= std_logic_vector(to_unsigned(D * PHASECONSTANT / CLKFREQ, ACCUMSIZE));
52          elsif (key_in(5) = '1') then
53              increment <= std_logic_vector(to_unsigned(E * PHASECONSTANT / CLKFREQ, ACCUMSIZE));
54          elsif (key_in(4) = '1') then
55              increment <= std_logic_vector(to_unsigned(F * PHASECONSTANT / CLKFREQ, ACCUMSIZE));
56          elsif (key_in(3) = '1') then
57              increment <= std_logic_vector(to_unsigned(G * PHASECONSTANT / CLKFREQ, ACCUMSIZE));
58          elsif (key_in(2) = '1') then
59              increment <= std_logic_vector(to_unsigned(A * PHASECONSTANT / CLKFREQ, ACCUMSIZE));
60          elsif (key_in(1) = '1') then
61              increment <= std_logic_vector(to_unsigned(B * PHASECONSTANT / CLKFREQ, ACCUMSIZE));
62          elsif (key_in(0) = '1') then
63              increment <= std_logic_vector(to_unsigned(HIGHC * PHASECONSTANT / CLKFREQ, ACCUMSIZE));
64          else
65              increment <= (others => '0');
66          end if;
67
68      end process getIncrement;
69
70  end Behavioral;
```

Figure 12: keyboardTB.vhd

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:   14:17:41 08/25/2014
-- Design Name:
-- Module Name:   C:/Users/F000JW7/Desktop/octave-keyboard/OctaveKeyboard/keyboardTB.vhd
-- Project Name:  OctaveKeyboard
-- Target Device:
-- Tool versions:
-- Description:
--
-- VHDL Test Bench Created by ISE for module: OctaveKeyboardTop
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-- Notes:
-- This testbench has been automatically generated using types std_logic and
-- std_logic_vector for the ports of the unit under test.  Xilinx recommends
-- that these types always be used for the top-level I/O of a design in order
-- to guarantee that the testbench will bind correctly to the post-implementation
-- simulation model.
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY keyboardTB IS
END keyboardTB;

ARCHITECTURE behavior OF keyboardTB IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT OctaveKeyboardTop
    PORT(
         clk : IN  std_logic;
         keys : IN  std_logic_vector(7 downto 0);
         led_disable : IN  std_logic;
         song_enable : IN  std_logic;
         tone : OUT  std_logic;
         shutdown : OUT  std_logic;
         led_out : OUT  std_logic_vector(7 downto 0)
        );
    END COMPONENT;

```

```
50
51      --Inputs
52      signal clk : std_logic := '0';
53      signal keys : std_logic_vector(7 downto 0) := (others => '0');
54      signal led_disable : std_logic := '0';
55      signal song_enable : std_logic := '0';
56
57            --Outputs
58      signal tone : std_logic;
59      signal shutdown : std_logic;
60      signal led_out : std_logic_vector(7 downto 0);
61
62      -- Clock period definitions
63      constant clk_period : time := 10 ns;
64
65  BEGIN
66
67            -- Instantiate the Unit Under Test (UUT)
68      uut: OctaveKeyboardTop PORT MAP (
69              clk => clk,
70              keys => keys,
71              led_disable => led_disable,
72              song_enable => song_enable,
73              tone => tone,
74              shutdown => shutdown,
75              led_out => led_out
76            );
77
78      -- Clock process definitions
79      clk_process :process
80      begin
81                      clk <= '0';
82                      wait for clk_period/2;
83                      clk <= '1';
84                      wait for clk_period/2;
85      end process;
86
87
88      -- Stimulus process
89      stim_proc: process
90      begin
91         -- hold reset state for 100 ns.
92         wait for 100 ns;
93
94         wait for clk_period*5;
95
96                      led_disable <= '0';
97                      song_enable <= '0';
98
99         keys <= "10000000";
100        wait for 250us;
101                     keys <= "10001000";
```

```
102                 wait for 50us;
103                 keys <= "00010000";
104                 wait for 200us;
105                 keys <= "00100010";
106                 wait for 300us;
107                 led_disable <= '1';
108                 wait for 200us;
109                 keys <= "00000010";
110                 wait for 200us;
111                 keys <= "00000100";
112                 wait for 200us;
113                 keys <= "00001000";
114                 wait for 200us;
115                 keys <= "00011111";
116                 led_disable <= '0';
117                 wait for 50us;
118                 keys <= "00000000";
119                 wait for 200us;
120                 keys <= "10111000";
121                 song_enable <= '1';
122       wait;
123    end process;
124
125 END;
```

### 7.2.3 Resource utilization

Figure 13: Advanced HDL Synthesis Report

```
Macro Statistics
# Counters                                : 3
 14-bit up counter                       : 1
 32-bit up counter                       : 1
 4-bit up counter                        : 1
# Accumulators                            : 1
 13-bit up accumulator                   : 1
# Registers                               : 8
 Flip-Flops                              : 8
# Comparators                             : 2
 10-bit comparator greater               : 1
 4-bit comparator equal                  : 1
# Multiplexers                            : 10
 1-bit 2-to-1 multiplexer                : 2
 13-bit 2-to-1 multiplexer               : 7
 8-bit 2-to-1 multiplexer                : 1
# FSMs                                    : 1
```

Figure 14: Device Utilization Summary

```
Slice Logic Utilization:
 Number of Slice Registers:          189  out of  18224     1%
 Number of Slice LUTs:               335  out of   9112     3%
    Number used as Logic:            335  out of   9112     3%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used: 346
   Number with an unused Flip Flop:  157  out of    346    45%
   Number with an unused LUT:         11  out of    346     3%
   Number of fully used LUT-FF pairs: 178  out of    346    51%
   Number of unique control sets:      20

IO Utilization:
 Number of IOs:                       29
 Number of bonded IOBs:               29  out of    232    12%
    IOB Flip Flops/Latches:            2

Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:            2  out of     16    12%
```

### 7.2.4 UCF

Figure 15: UCF file

```
## Clock signal
NET "clk"          LOC = "V10" | IOSTANDARD = "LVCMOS33";
Net "clk" TNM_NET = sys_clk_pin;
TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 100000 kHz;

## Leds
NET "led_out<0>"  LOC = "U16" | IOSTANDARD = "LVCMOS33";
NET "led_out<1>"  LOC = "V16" | IOSTANDARD = "LVCMOS33";
NET "led_out<2>"  LOC = "U15" | IOSTANDARD = "LVCMOS33";
NET "led_out<3>"  LOC = "V15" | IOSTANDARD = "LVCMOS33";
NET "led_out<4>"  LOC = "M11" | IOSTANDARD = "LVCMOS33";
NET "led_out<5>"  LOC = "N11" | IOSTANDARD = "LVCMOS33";
NET "led_out<6>"  LOC = "R11" | IOSTANDARD = "LVCMOS33";
NET "led_out<7>"  LOC = "T11" | IOSTANDARD = "LVCMOS33";

## Switches
NET "led_disable" LOC = "T10" | IOSTANDARD = "LVCMOS33";
NET "song_enable" LOC = "T9"  | IOSTANDARD = "LVCMOS33";

##JA
NET "keys<4>"      LOC = "T12" | IOSTANDARD = "LVCMOS33";
NET "keys<0>"      LOC = "V12" | IOSTANDARD = "LVCMOS33";
NET "keys<5>"      LOC = "N10" | IOSTANDARD = "LVCMOS33";
NET "keys<1>"      LOC = "P11" | IOSTANDARD = "LVCMOS33";

##JB
NET "keys<6>"      LOC = "K2"  | IOSTANDARD = "LVCMOS33";
NET "keys<2>"      LOC = "K1"  | IOSTANDARD = "LVCMOS33";
NET "keys<7>"      LOC = "L4"  | IOSTANDARD = "LVCMOS33";
NET "keys<3>"      LOC = "L3"  | IOSTANDARD = "LVCMOS33";

##JD, LX16 Die only
NET "tone"         LOC = "G11" | IOSTANDARD = "LVCMOS33";
NET "shutdown"     LOC = "E11" | IOSTANDARD = "LVCMOS33";
```

## 7.3 Memory Map

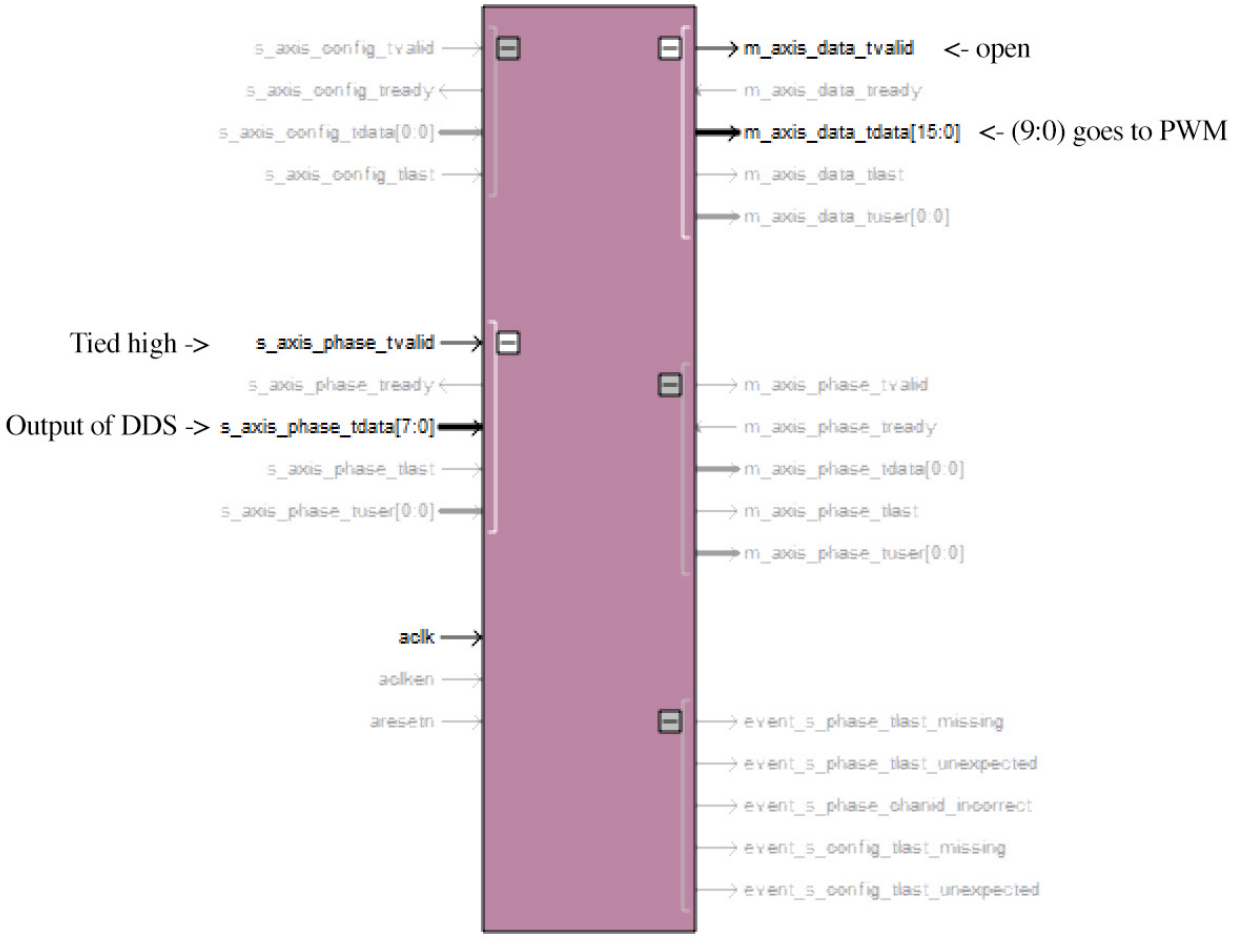



Figure 16: Memory Map for Sine LUT

## 7.4 Timing Diagram

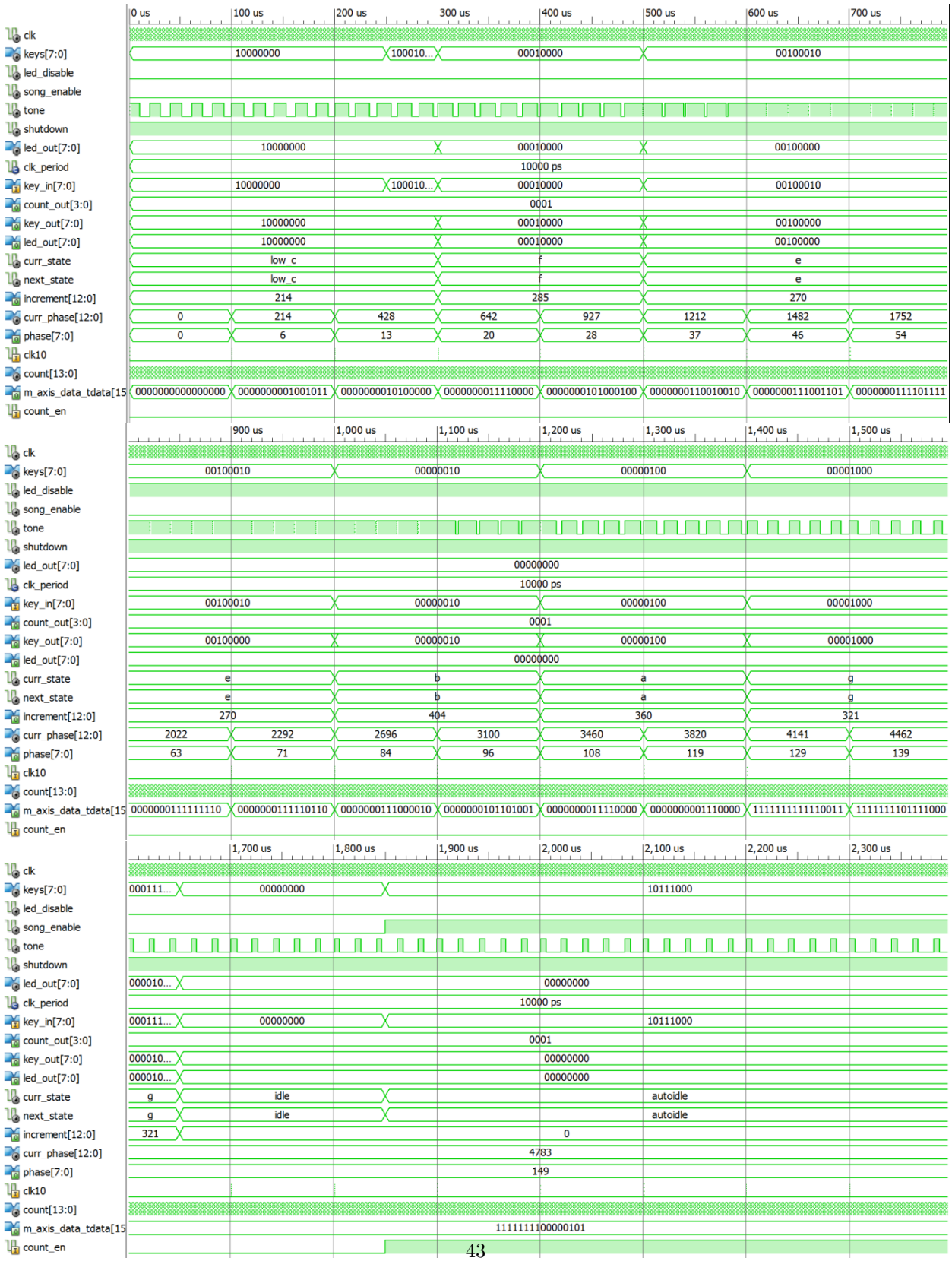



Figure 17: Timing Diagram of Test Bench

The critical path is between clkcount_8 and clkcount_23 in kidsCounter. The delay is 4.866ns. Here is the excerpt from the synthesis report:

```
Timing constraint: Default period analysis for Clock 'clk_en'
  Clock period: 4.866ns (frequency: 205.508MHz)
  Total number of paths / destination ports: 3669 / 262
--------------------------------------------------------------------------------
Delay:               4.866ns (Levels of Logic = 3)
  Source:            kidsCounter/clkcount_8 (FF)
  Destination:       kidsCounter/clkcount_23 (FF)
  Source Clock:      clk_en rising
  Destination Clock: clk_en rising

Data Path: kidsCounter/clkcount_8 to kidsCounter/clkcount_23
                                   Gate     Net
Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
----------------------------------------  ------------
 FD:C->Q               3   0.525   1.221  kidsCounter/clkcount_8 (kidsCounter/clkcount_8)
 LUT6:I0->O            7   0.254   1.186
    kidsCounter/GND_11_o_clkcount[31]_equal_1_o<31>4
    (kidsCounter/GND_11_o_clkcount[31]_equal_1_o<31>3)
 LUT5:I1->O           13   0.254   1.098  kidsCounter/_n00281 (kidsCounter/_n0028)
 LUT4:I3->O            1   0.254   0.000
    kidsCounter/clkcount_23_rstpot
    (kidsCounter/clkcount_23_rstpot)
 FD:D                      0.074          kidsCounter/clkcount_23
----------------------------------------
Total                      4.866ns (1.361ns logic, 3.505ns route)
                                   (28.0% logic, 72.0% route)
```

Based on this information, the maximum clock speed for this circuit is 205.508MHz, meaning the clock period must be at least 4.866ns.

## 7.5 Data Sheets

The proceeding pages contain data sheets for parts used in this project that are not already on the class website. They include data sheets for these parts:

- PmodAMP2
- PmodBB

# PmodAMP2™ Reference Manual

Revision: August 2, 2012
Note: This document applies to REV B of the board.

DIGILENT
www.digilentinc.com
1300 NE Henley Court, Suite 3
Pullman, WA 99163
(509) 334 6306 Voice | (509) 334 6300 Fax

## Overview

The PmodAMP2 amplifies low power audio signals to drive a monophonic output. The module features a digital gain select that allows output at 6 dB and 12 dB with pop-and-click suppression. A Digilent 6-pin connector provides the audio input to the module and a 1/8-inch mono jack supplies the speaker output.

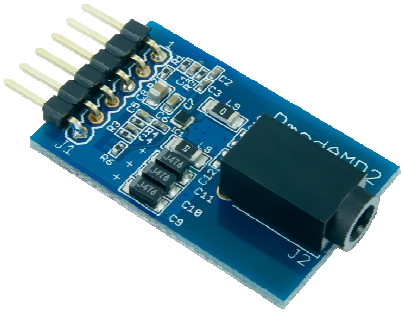
For customer convenience, Digilent has an inexpensive speaker and enclosure available for sale that is suitable for use with the PmodAMP2. Also, unlike most Digilent Pmod modules that accept only digital inputs, the PmodAMP2 accepts analog inputs and pulse width modulated digital inputs.

## Inputs and Outputs (I/O)

The PmodAMP2 accepts either digital or analog inputs at a voltage range of 0-Vcc. Typically a Digilent system board supplies power to the module at 3.3V, though the maximum supply voltage is 5.0V. The connector J1 provides the audio input, gain select, shutdown select, and power. (See figure 1)

There are several suitable inputs for the PmodAMP2. The typical input is a pulse width modulated (PWM) signal produced by a digital output from a Digilent programmable logic system board or microcontroller board. The low pass filter on the input acts as a reconstruction filter to convert the PWM digital signal into an analog voltage on the amplifier input.

The PmodAMP2 also accepts analog inputs with an input voltage range of 0-Vcc. These inputs will often be from an analog to digital converter module, like the Digilent PmodDA1 or PmodDA2. The output of a digital to analog



Features Include:

- *Analog Devices SSM2377: Filterless, High Efficiency, Mono 2.5 Watt Class-D Audio Amplifier*
- *Digital gain select*
- *Pop –and-click suppression*
- *Micropower shutdown mode*
- *1/8-inch mono speaker jack*
- *A 6-pin header for input*
- *2.5V – 5V operating voltage*

Figure 1

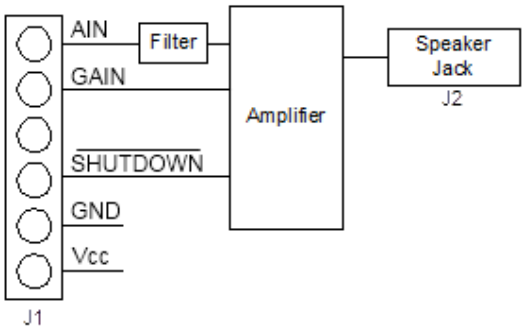





Figure 18: PmodAMP2 Data Sheet Page 1/2

converter module will normally have a voltage range of 0-3.3V and should have a sample rate of at least 16Khz. The low pass filter on the input removes the high frequency artifacts generated during the sampling process.

Additionally, the PmodAMP2 accepts inputs from a variety of line level audio signals. A line level input, like the output of a portable CD player or MP3 player, will typically be a 1V peak-to-peak analog voltage. The input band-pass filter clarifies and amplifies the input voltage from the signal source and then directs the signal to the output jack to drive a speaker. The connector J2 operates as the speaker output. (See figure 1)

## Functional Description

The gain on the PmodAMP2 may be selected by tying the GAIN input to either logic '1' or logic '0'. (See table 1)

**Table 1**

| GAIN Input | Gain |
|---|---|
| 1 | 6 decibels (dB) |
| 0 | 12 decibels (dB) |

The PmodAMP2 features a micropower shutdown mode with a typical shutdown current of 100 nA. Users can enable the shutdown by applying a logic low to the $\overline{\text{SHUTDOWN}}$ pin. A10K-ohm resistor pulls the pin down to ground. To operate the AMP2 users must ensure the $\overline{\text{SHUTDOWN}}$ pin is in the highest position.

Customers will generally use the PmodAMP2 module with a Digilent programmable logic system board or microcontroller board. These boards produce either a pulse width modulated digital signal or an analog signal via a digital to analog converter. Most Digilent system boards have 6-pin connectors that allow the PmodAMP2 to plug directly into the system board or to connect via a Digilent 6-pin cable.

Some older model Digilent boards may need a Digilent Module Interface Board (MIB) and a 6-pin cable to connect to the PmodAMP2. The MIB plugs into the system board and the cable connects the PmodAMP2 to the MIB.

*Note: For more information about the operation and features of the Analog Devices SSM2377 Audio Amplifier integrated circuit please see the datasheet available at www.analog.com.*





Figure 19: PmodAMP2 Data Sheet Page 2/2

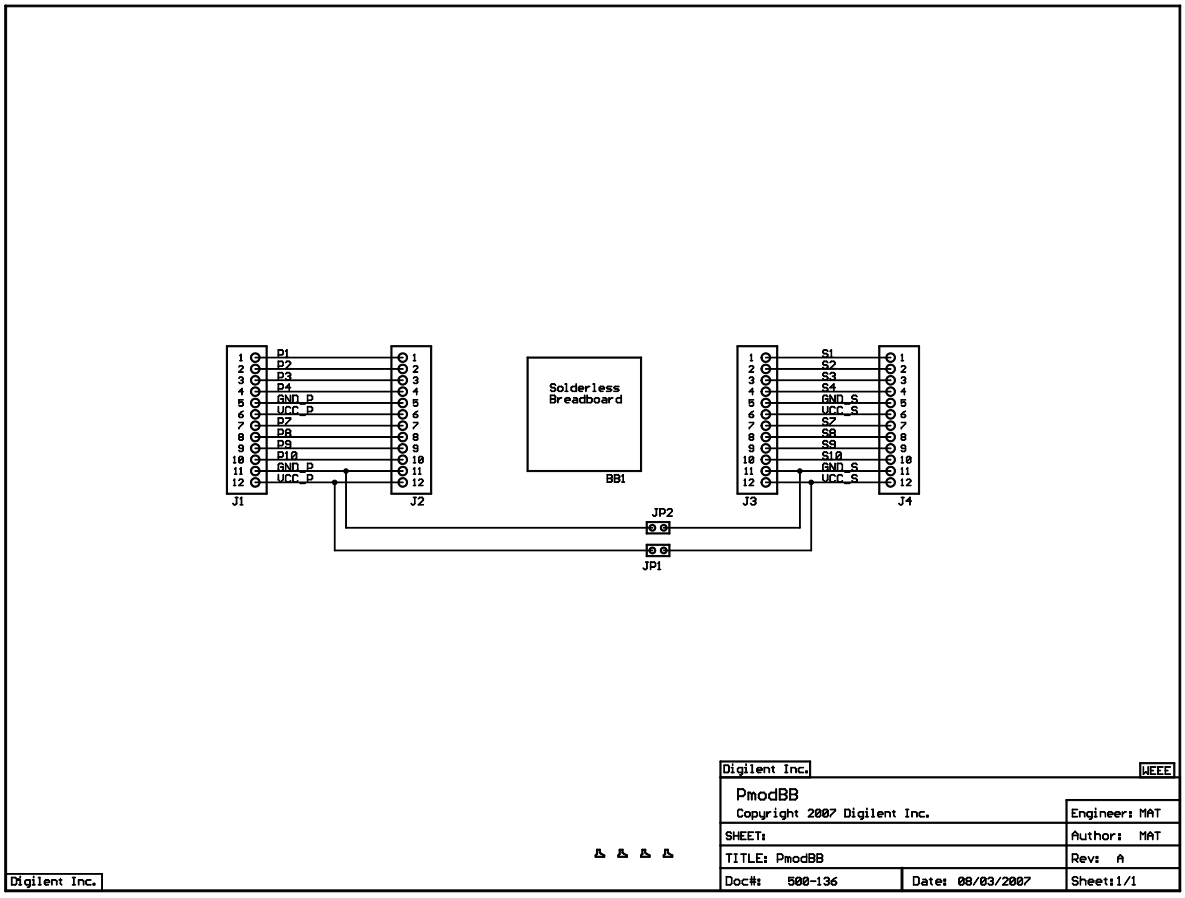



Figure 20: PmodBB Schematic