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

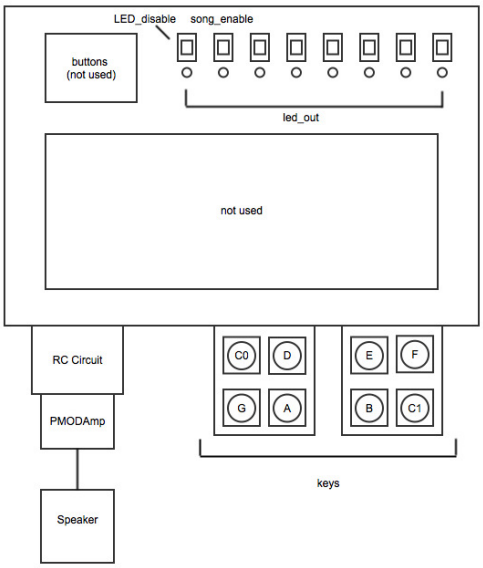
2.1 Specifications

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.

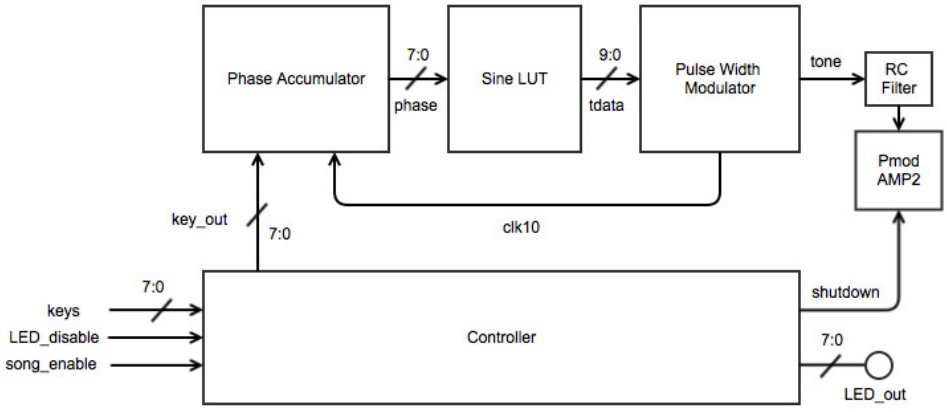
The outputs to the circuit are:

- 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 generated (a copy of the LED output) which is used to obtain the appropriate increment value for each key. The phase accumulator increments with this value, and the register output value is then used as an index into the SineLUT. The wave samples obtained from the SineLUT are passed to the pulse width modulator (PWM) to produce square wave pulses at the desired frequency. The RC filter then smooths out the wave pulses to produce the note we pressed!

For more information on how the circuit works, see section 2.3 "Theory of Operation."



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 song_enable went from low to high.

3 Evaluation of Design

Our solution is a simple standard solution to a keyboard. Although we finished everything we set out to do, there are definitely things that could have been improved upon or added.

We far from maxed out the resources of the Nexys3. This leaves space for many things. We could have used a smaller frequency resolution to increase the note accuracy, added much more notes (potentially by using a switch for octaves), allowed sharps and flats, or allowed recording and playback.

General features that we speculated on but ended up not pursuing included adding a metronome and additional songs.

Additionally, the hardware is not the best for the project. Having the keys split into two rows is non-standard and makes the project unusable for most practical purposes.

If we were to start over on this project, we would do a better job testing each component before implementing them as part of the larger project. We ended up having to do a lot of debugging on issues that could have been picked out earlier by a more "unit tested" approach.

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

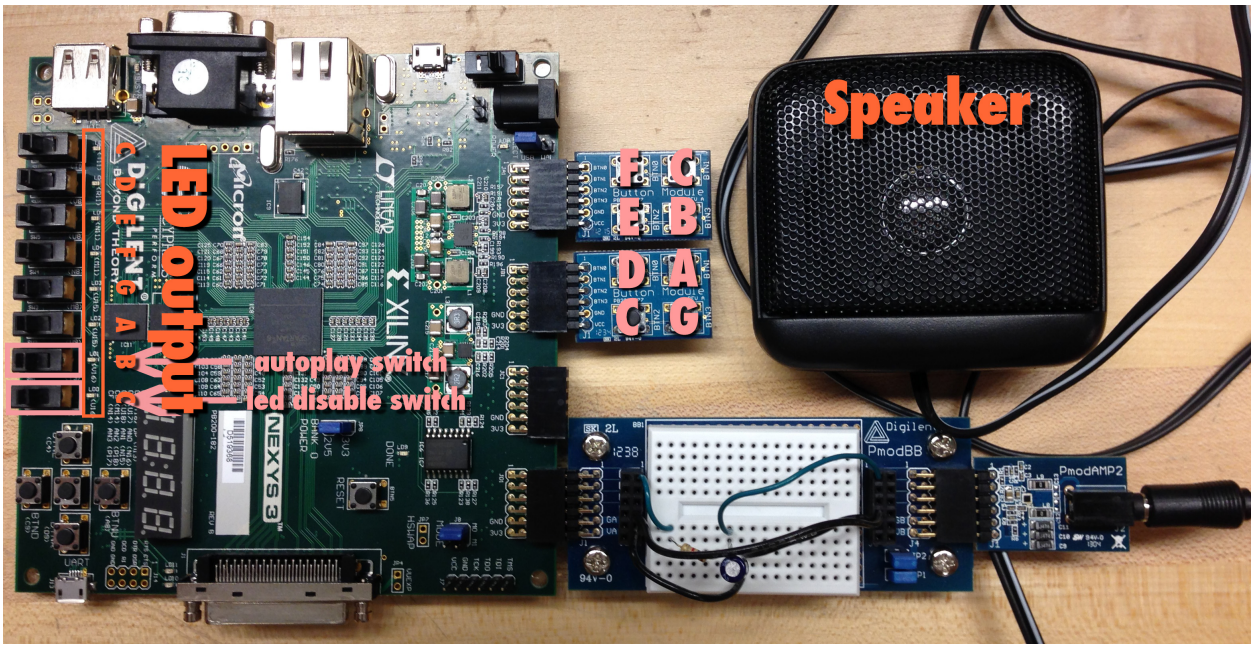


Figure 1: Annotated Digital Photo of Project

7.1.2 Block Diagram

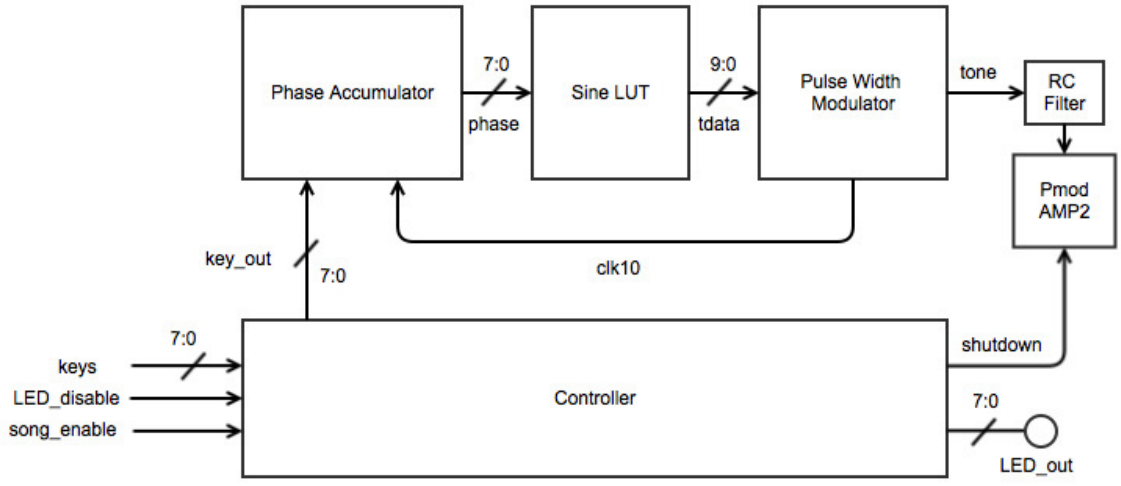


Figure 2: Basic Block Diagram of Keyboard

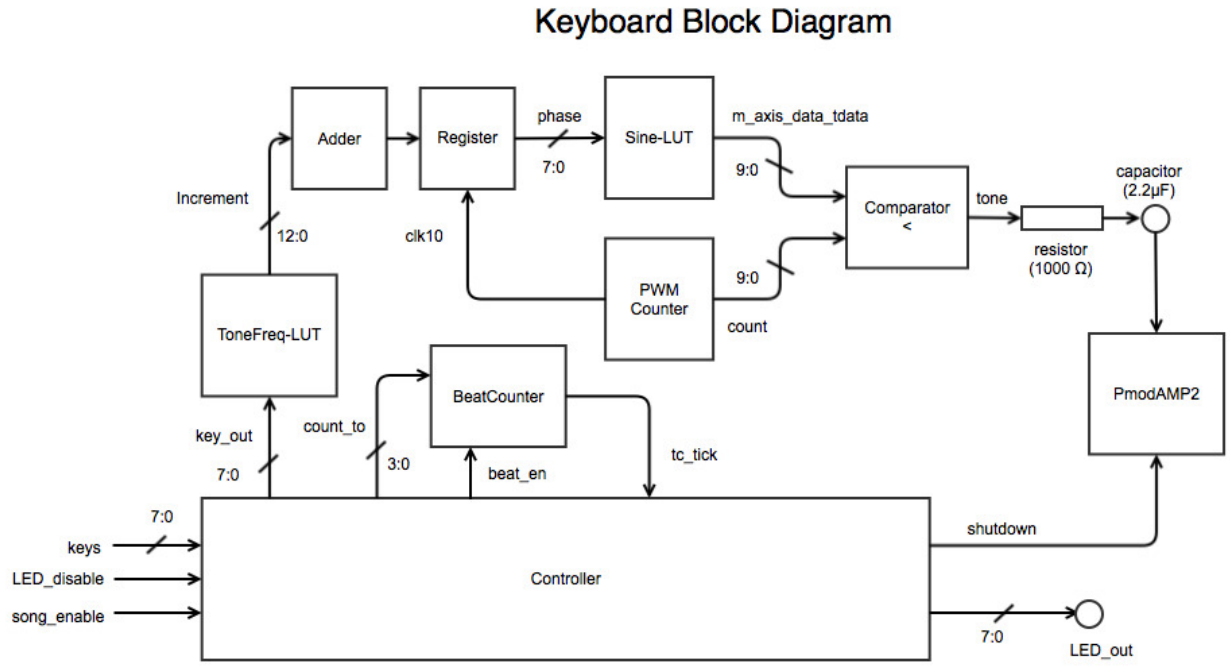


Figure 3: Advanced Block Diagram of Keyboard

7.1.3 Schematic Diagram

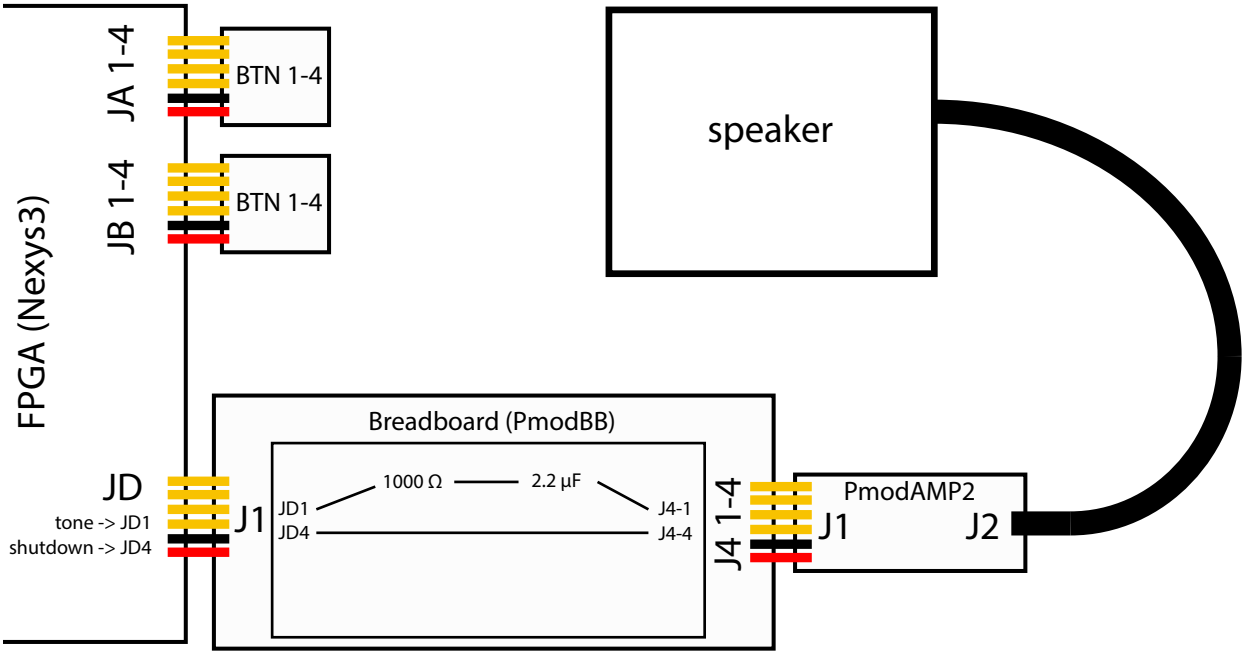


Figure 4: Schematic Diagram for the RC Filter

7.1.4 Package Map

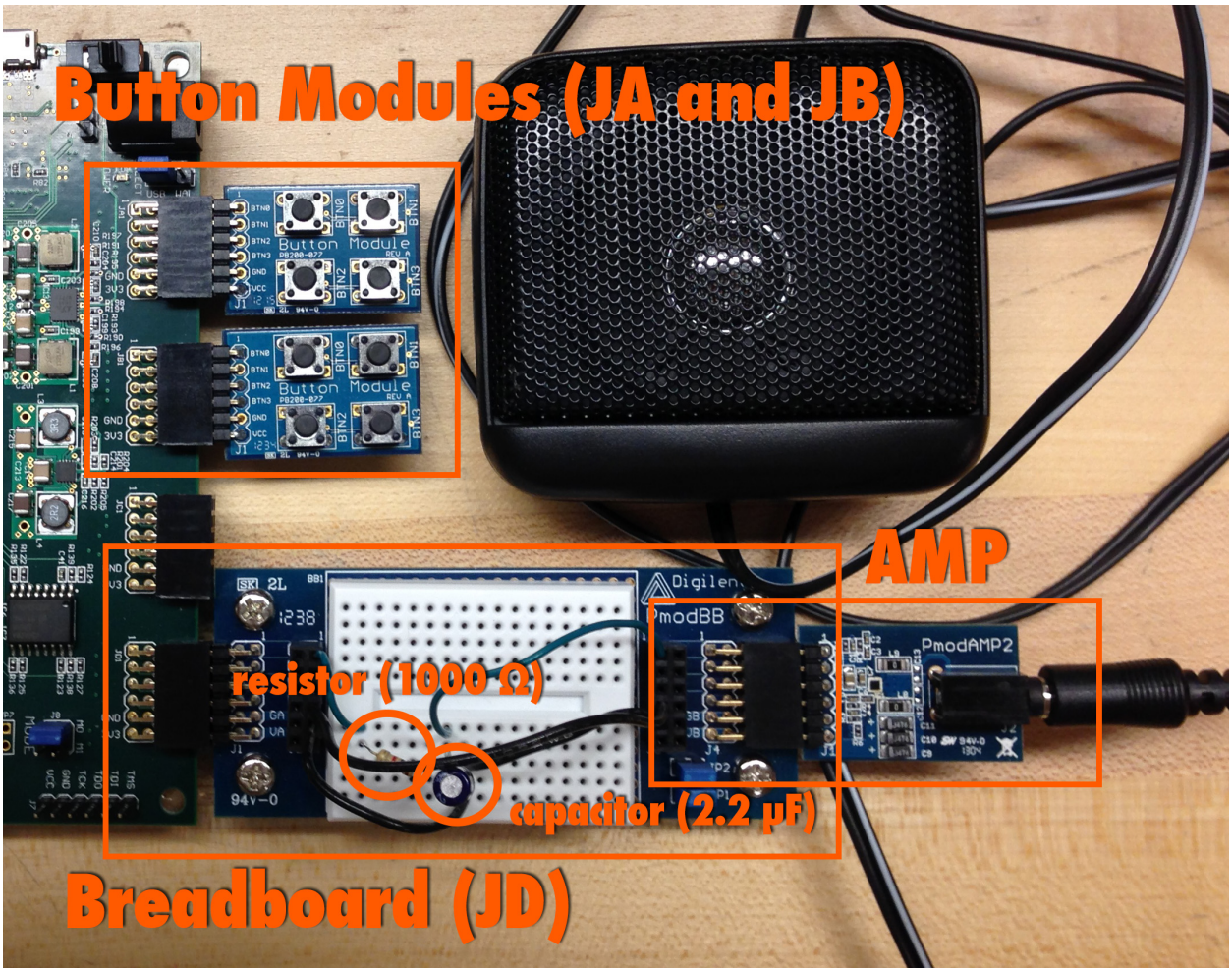


Figure 5: Package Map

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 6: State Diagram Defaults

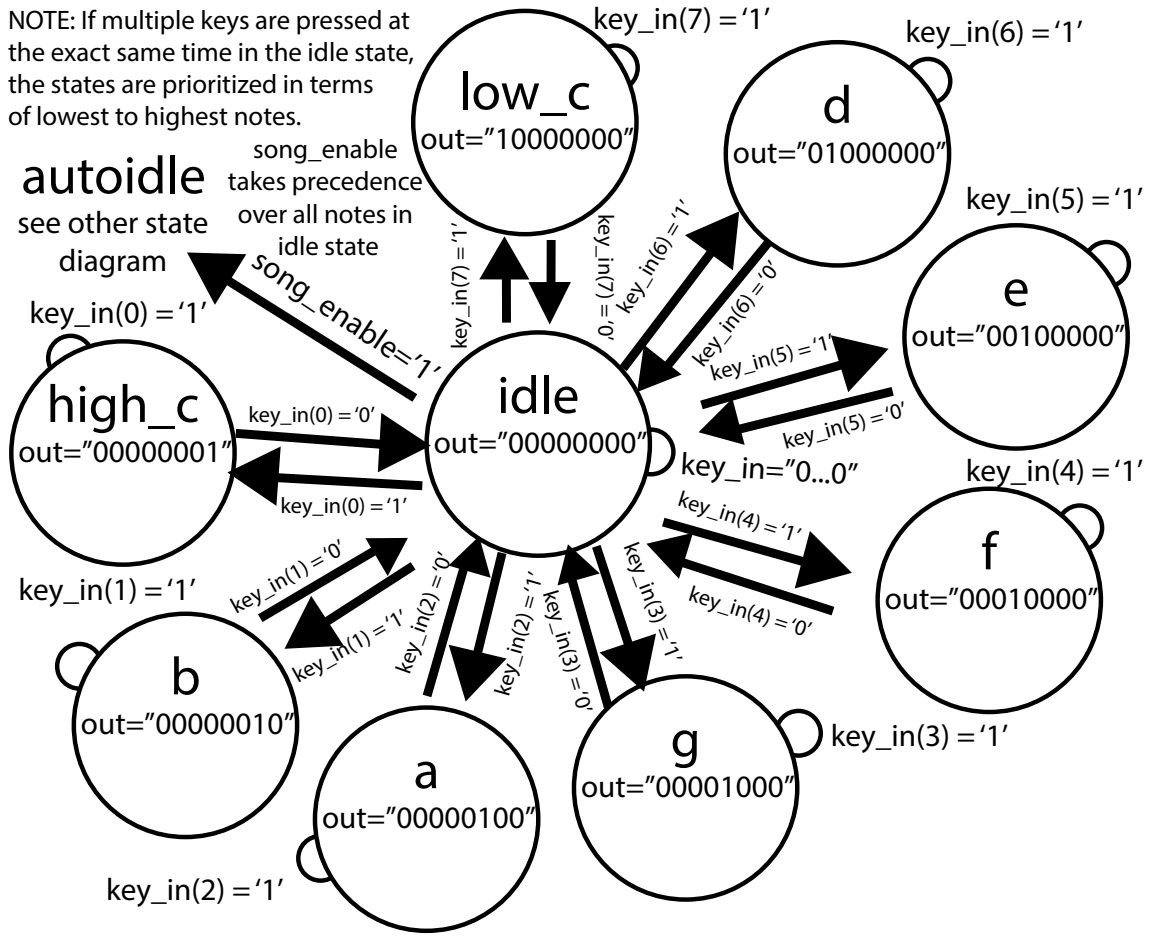


Figure 7: User Play State Diagram

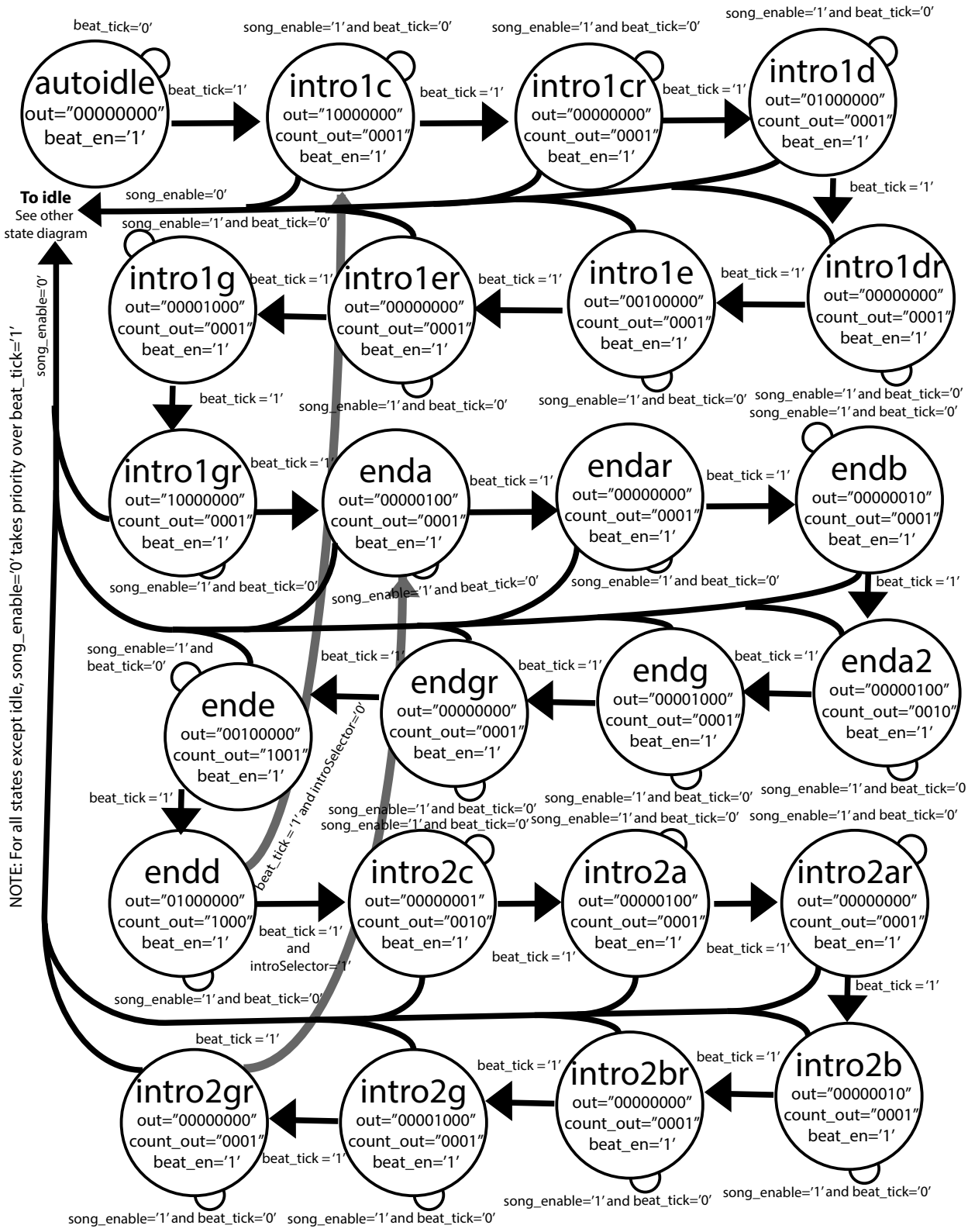


Figure 8: Autoplay State Diagram

7.2.2 VHDL Code

Figure 9: 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: Top level for Octave Keyboard
7  -- Module Name: OctaveKeyboardTop - Behavioral
8  -- Project Name: OctaveKeyboard
9  -- Target Devices: Nexys3
10 -- Tool versions:
11 -- Description: Top level VHDL module for keyboard
12 ----------------------------------------------------------------------------------
13 library IEEE;
14 use IEEE.STD_LOGIC_1164.ALL;
15 use IEEE.NUMERIC_STD.ALL;
16
17 library UNISIM;
18 use UNISIM.VComponents.all;
19
20 entity OctaveKeyboardTop is
21     Generic (   ACCUMSIZE    : integer := 13;   -- phase accumulator bitsize
22                 INDEXSIZE    : integer := 8;    -- SinLUT index bitsize
23                 LUTOUT       : integer := 10);  -- SinLUT out bitsize
24
25     Port (  clk          : in   STD_LOGIC;
26             keys         : in   STD_LOGIC_VECTOR (7 downto 0);
27             led_disable  : in   STD_LOGIC;
28             song_enable  : in   STD_LOGIC;
29             tone         : out  STD_LOGIC;
30             shutdown     : out  STD_LOGIC;
31             led_out      : out  STD_LOGIC_VECTOR (7 downto 0));
32 end OctaveKeyboardTop;
33
34 architecture Behavioral of OctaveKeyboardTop is
35
36     -- signals for 100MHz to 50Mhz clk divider
37     signal clk_en           : std_logic := '0';
38     signal slowclk          : std_logic;
39
40     signal led_disable_sync : std_logic := '0';
41     signal song_enable_sync : std_logic := '0';
42
43     -- mapping signals
44     signal step : std_logic_vector(ACCUMSIZE-1 downto 0) := (others => '0');
45     signal controllerKeys : std_logic_vector(7 downto 0) := (others => '0');
46     signal phase : std_logic_vector(INDEXSIZE-1 downto 0) := (others => '0');
```

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

```
99      END COMPONENT;
100
101     COMPONENT PlayCount is
102         PORT (  clk             : in      STD_LOGIC;
103                 count_en        : in      STD_LOGIC;
104                 count_to        : in      STD_LOGIC_VECTOR (3 downto 0);
105                 tc_tick         : out     STD_LOGIC);
106     END COMPONENT;
107     -- END component declarations
108
109 begin
110     -- synchronizer for auto-play-song switch input
111     SynchronizeSwitches: process(clk)
112     begin
113         if rising_edge(clk) then
114             led_disable_sync <= led_disable;
115             song_enable_sync <= song_enable;
116         end if;
117     end process SynchronizeSwitches;
118
119     -- slow main clock down by half (here: 100Mhz to 50Mhz)
120     clkDivider: process(clk)
121     begin
122         if rising_edge(clk) then
123             clk_en <= NOT(clk_en);
124         end if;
125     end process clkDivider;
126
127     -- map signals
128     shutdown <= '1';      -- tie shutdown signal high for speaker pmod
129
130     slowclk_buf: BUFG
131         Port map ( I => clk_en,
132                    O => slowclk );
133
134     -- debouncers for note buttons
135     debouncer0: debounce
136         Port map (  clk => slowclk,
137                     switch => keys(0),
138                     dbswitch => keyDB(0) );
139
140     debouncer1: debounce
141         Port map (  clk => slowclk,
142                     switch => keys(1),
143                     dbswitch => keyDB(1) );
144
145     debouncer2: debounce
146         Port map (  clk => slowclk,
147                     switch => keys(2),
148                     dbswitch => keyDB(2) );
149
150     debouncer3: debounce
```

```
151         Port map (  clk => slowclk,
152                     switch => keys(3),
153                     dbswitch => keyDB(3) );
154 
155     debouncer4: debounce
156         Port map (  clk => slowclk,
157                     switch => keys(4),
158                     dbswitch => keyDB(4) );
159 
160     debouncer5: debounce
161         Port map (  clk => slowclk,
162                     switch => keys(5),
163                     dbswitch => keyDB(5) );
164 
165     debouncer6: debounce
166         Port map (  clk => slowclk,
167                     switch => keys(6),
168                     dbswitch => keyDB(6) );
169 
170     debouncer7: debounce
171         Port map (  clk => slowclk,
172                     switch => keys(7),
173                     dbswitch => keyDB(7) );
174 
175     KeyControl: Controller
176         PORT MAP (  clk         => slowclk,
177                     key_in         => keyDB, -- change to keys if simulating
178                     led_disable => led_disable_sync,
179                     song_enable => song_enable_sync,
180                     beat_tick     => countdone, -- passed to PlayCount
181                     beat_en        => tempo_en, -- passed to PlayCount
182                     count_out    => count, -- passed to PlayCount
183                     key_out     => controllerKeys, -- passed to FreqLUT
184                     led_out        => led_out); -- straight to LEDs
185 
186     KeyFrequencies: FreqLUT
187         PORT MAP (  clk         => slowclk,
188                     key_in         => controllerKeys, -- from controller
189                     increment     => step); -- passed to DDS
190 
191     PhaseAccum: DDS
192         PORT MAP (  clk         => slowclk,
193                     clk10        => reg_en,
194                     step        => step, -- from freqLUT
195                     phase        => phase); -- to SinLUT
196 
197     PulseWM: PWM
198         PORT MAP (  clk           => slowclk,
199                     sample        => lutfreq(9 downto 0), -- from sinLUT
200                     slowclk       => reg_en,
201                     pulse        => tone); -- to speaker
202 
```

```
203     SinFreqs: SinLUT
204         PORT MAP (  aclk                  => slowclk,
205                     s_axis_phase_tvalid => '1',
206                     s_axis_phase_tdata     => phase, -- from phase accumulator
207                     m_axis_data_tvalid     => open,
208                     m_axis_data_tdata     => lutfreq); -- to PWM
209 
210     kidsCounter: PlayCount
211         PORT MAP    ( clk => slowclk,
212                       count_en => tempo_en, -- from controller
213                       count_to => count,    -- from controller
214                       tc_tick => countDone); -- from controller
215 
216 end Behavioral;
```

Figure 10: 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 key input to monotone. Also
--              contains state machine for "Kids" by MGMT.
----------------------------------------------------------------------------------
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

    -- updates state to next state.
    StateUpdate: process(clk)
    begin
        if rising_edge(clk) then
            curr_state <= next_state;
        end if;
    end process StateUpdate;
```

```
50
51      -- determines next state
52      CombLogic: process(curr_state, next_state, key_in, led_disable, output,
53                         beat_tick, song_enable, introSelector)
54      begin
55
56          -- defaults
57          next_state <= curr_state;
58          output <= (others => '0');
59          key_out <= output;
60          count_out <= "0001";
61          repeat_tick <= '0';
62          beat_en <= '0';
63
64          -- makes led_out all 0 if led_disable is switched on
65          if (led_disable = '1') then
66              led_out <= (others => '0');
67          else
68              led_out <= output;
69          end if;
70
71          case curr_state is
72
73              -- idle state
74              -- goes to autoplay if song enable
75              -- goes to note states if buttons are pressed, prioritized in order
76              -- of lowest to highest notes.
77              when idle =>
78
79                  if song_enable = '1' then
80                      next_state <= autoidle;
81                  elsif key_in(7) = '1' then
82                      next_state <= low_c;
83                  elsif key_in(6) = '1' then
84                      next_state <= d;
85                  elsif key_in(5) = '1' then
86                      next_state <= e;
87                  elsif key_in(4) = '1' then
88                      next_state <= f;
89                  elsif key_in(3) = '1' then
90                      next_state <= g;
91                  elsif key_in(2) = '1' then
92                      next_state <= a;
93                  elsif key_in(1) = '1' then
94                      next_state <= b;
95                  elsif key_in(0) = '1' then
96                      next_state <= high_c;
97                  else
98                      next_state <= idle;
99                  end if;
100
101             -- USER INPUT NOTE STATES
```

```
102             -- output is changed to match the note
103             -- goes back to idle when corresponding key is pressed
104
105             when low_c =>
106                 output <= "10000000";
107                 if key_in(7) = '0' then
108                     next_state <= idle;
109                 end if;
110
111             when d =>
112                 output <= "01000000";
113                 if key_in(6) = '0' then
114                     next_state <= idle;
115                 end if;
116
117             when e =>
118                 output <= "00100000";
119                 if key_in(5) = '0' then
120                     next_state <= idle;
121                 end if;
122
123             when f =>
124                 output <= "00010000";
125                 if key_in(4) = '0' then
126                     next_state <= idle;
127                 end if;
128
129             when g =>
130                 output <= "00001000";
131                 if key_in(3) = '0' then
132                     next_state <= idle;
133                 end if;
134
135             when a =>
136                 output <= "00000100";
137                 if key_in(2) = '0' then
138                     next_state <= idle;
139                 end if;
140
141             when b =>
142                 output <= "00000010";
143                 if key_in(1) = '0' then
144                     next_state <= idle;
145                 end if;
146
147             when high_c =>
148                 output <= "00000001";
149                 if key_in(0) = '0' then
150                     next_state <= idle;
151                 end if;
152
153             -- AUTOPLAY STATES
```

```
154             -- beat_en turns on RepeatCounter
155             -- output corresponds to current not to be pressed
156             -- count_out is the number of quarter seconds a note should be held
157             -- the state moves to the next one once the counter reaches it's
158             -- limit (meaning the note has been held long enough)
159             -- or when the song_enable switch is toggled off (next state becomes
160             -- idle).
161
162             when autoidle =>
163                 beat_en <= '1';
164                 output <= (others => '0');
165                 if (beat_tick = '1') then
166                     next_state <= intro1c;
167                 end if;
168
169             when intro1c =>
170                 beat_en <= '1';
171                 output <= "10000000";
172                 count_out <= "0001";
173                 if (song_enable = '0') then
174                     next_state <= idle;
175                 elsif(beat_tick = '1') then
176                     next_state <= intro1cr;
177                 end if;
178
179
180             when intro1cr =>
181                 beat_en <= '1';
182                 output <= "00000000";
183                 count_out <= "0001";
184                 if (song_enable = '0') then
185                     next_state <= idle;
186                 elsif(beat_tick = '1') then
187                     next_state <= intro1d;
188                 end if;
189
190
191             when intro1d =>
192                 beat_en <= '1';
193                 output <= "01000000";
194                 count_out <= "0001";
195                 if (song_enable = '0') then
196                     next_state <= idle;
197                 elsif(beat_tick = '1') then
198                     next_state <= intro1dr;
199                 end if;
200
201
202             when intro1dr =>
203                 beat_en <= '1';
204                 output <= "00000000";
205                 count_out <= "0001";
```

```
206                 if (song_enable = '0') then
207                     next_state <= idle;
208                 elsif(beat_tick = '1') then
209                     next_state <= intro1e;
210                 end if;
211 
212 
213             when intro1e =>
214                 beat_en <= '1';
215                 output <= "00100000";
216                 count_out <= "0001";
217                 if (song_enable = '0') then
218                     next_state <= idle;
219                 elsif(beat_tick = '1') then
220                     next_state <= intro1er;
221                 end if;
222 
223 
224             when intro1er =>
225                 beat_en <= '1';
226                 output <= "00000000";
227                 count_out <= "0001";
228                 if (song_enable = '0') then
229                     next_state <= idle;
230                 elsif(beat_tick = '1') then
231                     next_state <= intro1g;
232                 end if;
233 
234 
235             when intro1g =>
236                 beat_en <= '1';
237                 output <= "00001000";
238                 count_out <= "0001";
239                 if (song_enable = '0') then
240                     next_state <= idle;
241                 elsif(beat_tick = '1') then
242                     next_state <= intro1gr;
243                 end if;
244 
245 
246             when intro1gr =>
247                 beat_en <= '1';
248                 output <= "00000000";
249                 count_out <= "0001";
250                 if (song_enable = '0') then
251                     next_state <= idle;
252                 elsif(beat_tick = '1') then
253                     next_state <= enda;
254                 end if;
255 
256 
257             when enda =>
```

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

```
310
311
312             when endgr =>
313                 beat_en <= '1';
314                 output <= "00000000";
315                 count_out <= "0001";
316                 if (song_enable = '0') then
317                     next_state <= idle;
318                 elsif(beat_tick = '1') then
319                     next_state <= ende;
320                 end if;
321
322
323             when ende =>
324                 beat_en <= '1';
325                 output <= "00100000";
326                 count_out <= "1001";
327                 if (song_enable = '0') then
328                     next_state <= idle;
329                 elsif(beat_tick = '1') then
330                     next_state <= endd;
331                 end if;
332
333
334             when endd =>
335                 beat_en <= '1';
336                 output <= "01000000";
337                 count_out <= "1000";
338
339                 if (beat_tick = '1') then
340                     repeat_tick <= '1'; -- increase reps
341
342                     -- choose intro depending on what has previously been played
343                     -- The song goes:
344                     --      intro1c to endd
345                     --      intro1c to endd
346                     --      intro2c to endd
347                     --      intro2c to endd
348                     -- and repeat until song_enable is switched off.
349                     if (introSelector = '0') then
350                         next_state <= intro1c;
351                     else
352                         next_state <= intro2c;
353                     end if;
354
355                 elsif (song_enable = '0') then
356                     next_state <= idle;
357
358                 end if;
359
360             when intro2c =>
361                 beat_en <= '1';
```

```
362                 output <= "00000001";
363                 count_out <= "0010";
364                 if (song_enable = '0') then
365                     next_state <= idle;
366                 elsif(beat_tick = '1') then
367                     next_state <= intro2a;
368                 end if;
369
370
371             when intro2a =>
372                 beat_en <= '1';
373                 output <= "00000100";
374                 count_out <= "0001";
375                 if (song_enable = '0') then
376                     next_state <= idle;
377                 elsif(beat_tick = '1') then
378                     next_state <= intro2ar;
379                 end if;
380
381
382             when intro2ar =>
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 <= intro2b;
390                 end if;
391
392
393             when intro2b =>
394                 beat_en <= '1';
395                 output <= "00000010";
396                 count_out <= "0001";
397                 if (song_enable = '0') then
398                     next_state <= idle;
399                 elsif(beat_tick = '1') then
400                     next_state <= intro2br;
401                 end if;
402
403
404             when intro2br =>
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 <= intro2g;
412                 end if;
413
```

```
414
415             when intro2g =>
416                 beat_en <= '1';
417                 output <= "00001000";
418                 count_out <= "0001";
419                 if (song_enable = '0') then
420                     next_state <= idle;
421                 elsif(beat_tick = '1') then
422                     next_state <= intro2gr;
423                 end if;
424
425
426             when intro2gr =>
427                 beat_en <= '1';
428                 output <= "00000000";
429                 count_out <= "0001";
430                 if (song_enable = '0') then
431                     next_state <= idle;
432                 elsif(beat_tick = '1') then
433                     next_state <= enda;
434                 end if;
435
436             when others =>
437                 next_state <= idle;
438
439         end case;
440
441     end process CombLogic;
442
443     -- counter which is keeps track of the autoplay song state
444     -- modifies introSelector based on reps
445     -- introSelector is then used to determine whether to go to intro1 or
446     -- intro2 at the end of the end states.
447     RepeatCounter: process(clk, repeat_tick, reps, introselector)
448     begin
449         if (rising_edge(clk)) then
450             if (repeat_tick = '1') then
451                 if (reps = '1') then
452                     introSelector <= not introSelector;
453                     reps <= '0';
454                 else
455                     reps <= not reps;
456                 end if;
457             end if;
458         end if;
459     end process RepeatCounter;
460
461 end Behavioral;
```

Figure 11: 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: Phase accumulator and register
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity DDS is
    Generic (  ACCUMSIZE    : integer := 13;    -- phase accumulator bitsize
               INDEXSIZE    : integer := 8);    -- SineLUT index bitsize

    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
            -- increment at enable from PWM (10kHz)
            if (clk10 = '1') then
                curr_phase <= curr_phase + unsigned(step);
            end if;
        end if;
    end process AccumPhase;

    -- take only the 8 most significant bits of the phase as index
    phase <= std_logic_vector(curr_phase(ACCUMSIZE-1 downto ACCUMSIZE-INDEXSIZE));

end Behavioral;
```

Figure 12: PWM.vhd

```
----------------------------------------------------------------------------------
-- Company: ENGS031 14X
-- Engineer: Vivian Hu and Daniel Chen
--
-- Create Date: 10:13:11 08/13/2014
-- Design Name: Pulse Width Modulator
-- Module Name: PWM - Behavioral
-- Project Name: Octave Keyboard
-- Target Devices: Nexys3
-- Tool versions:
-- Description: Pulse width modulator (counter and comparator)
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity PWM is
    Generic (  LUTOUT  : integer := 10);    -- SineLUT output bitsize

    Port (  clk     : in  STD_LOGIC;
            sample  : in  STD_LOGIC_VECTOR(LUTOUT-1 downto 0);
            slowclk : out STD_LOGIC;
            pulse   : out  STD_LOGIC);
end PWM;

architecture Behavioral of PWM is
    signal count : unsigned(13 downto 0) := (others => '0');
    signal offset : unsigned(LUTOUT-1 downto 0) := (others => '0');
    constant max : unsigned(13 downto 0) := "01001110001000";   -- max count value
begin

    PWM: process(clk, sample)
    begin
        -- convert SinLUT output from two's complement to unsigned offset binary
        offset <= unsigned(not sample(LUTOUT-1) & sample(LUTOUT-2 downto 0));

        if (rising_edge(clk)) then

            -- increment counter
            count <= count + 1;

            -- compare first 10 bits to SinLUT output
            if (count(9 downto 0) < offset) then
                pulse <= '1';
            else
                pulse <= '0';
            end if;

            -- if max count hit
```

```
50              if (count = max) then
51                  -- send 10kHz enable pulse to phase accumulator
52                  slowclk <= '1';
53                  count <= (others => '0');    -- restart count
54              else
55                  slowclk <= '0';
56              end if;
57          end if;
58      end process PWM;
59
60  end Behavioral;
```

Figure 13: PlayCount.vhd

```
----------------------------------------------------------------------------------
-- Company: ENGS031 14X
-- Engineer: Daniel Chen, Vivian Hu
--
-- Create Date:    11:40:24 08/16/2014
-- Design Name:    Play Counter
-- Module Name:    PlayCount - Behavioral
-- Project Name:   Octave-Keyboard
-- Target Devices: Nexys3
-- Tool versions:
-- Description: Play Counter used to keep track of beats for the controller FSM.
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity PlayCount is
    Port (  clk : in  STD_LOGIC;
            count_en : in STD_LOGIC;
            count_to : in  STD_LOGIC_VECTOR (3 downto 0);
            tc_tick : out  STD_LOGIC);
end PlayCount;

architecture Behavioral of PlayCount is
    constant QRTR_CLK_DIV    : integer := 12500000;
    signal   clkcount    : integer := 0;
    signal   count        : unsigned(3 downto 0) := "0001";
begin

    -- counts for a quarter second
    Qrtr_Sec_Count: process(clk)
    begin
        if (rising_edge(clk)) then
            -- if the clock is rising and counting is enabled
            if (count_en = '1') then
                tc_tick <= '0';

                -- if a quarter second has passed
                if (clkcount = QRTR_CLK_DIV - 1) then

                    -- if enough quarter seconds have passed
                    if (count = unsigned(count_to)) then
                        tc_tick <= '1';  -- tell the controller
                        count <= "0001"; -- reset the count

                    -- otherwise, just wait for enough quarter seconds to pass.
                    else
                        count <= count + 1;
                    end if;
```

```
50
51                      clkcount <= 0;
52
53                  -- wait for a quarter second otherwise
54                  else
55                      clkcount <= clkcount + 1;
56                  end if;
57              else
58                  tc_tick <= '0';
59              end if;
60          end if;
61      end process Qrtr_Sec_Count;
62  end Behavioral;
```

Figure 14: FreqLUT.vhd

```
1  ----------------------------------------------------------------------------------
2  -- Company: ENGS031 14X
3  -- Engineer: Daniel Chen and Vivian Hu
4  --
5  -- Create Date: 15:30:42 08/11/2014
6  -- Design Name: Frequency LUT
7  -- Module Name: FreqLUT - Behavioral
8  -- Project Name: Octave Keyboard
9  -- Target Devices: Nexys3
10 -- Tool versions:
11 -- Description: Frequency LUT that takes the keys and outputs the increment
12 -- corresponding to the notes.
13 ----------------------------------------------------------------------------------
14 library IEEE;
15 use IEEE.STD_LOGIC_1164.ALL;
16 use IEEE.NUMERIC_STD.ALL;
17
18 entity FreqLUT is
19     Generic (  ACCUMSIZE : integer := 13;
20                CLKFREQ   : integer := 10000);
21
22     Port ( clk : in  STD_LOGIC;
23            key_in : in  STD_LOGIC_VECTOR (7 downto 0);
24            increment : out  STD_LOGIC_VECTOR (ACCUMSIZE-1 downto 0));
25 end FreqLUT;
26
27 architecture Behavioral of FreqLUT is
28     constant PHASECONSTANT : integer := 2**ACCUMSIZE;
29
30     -- key frequencies (Hz)
31     constant LOWC : integer := 262;
32     constant D : integer := 294;
33     constant E : integer := 330;
34     constant F : integer := 349;
35     constant G : integer := 392;
36     constant A : integer := 440;
37     constant B : integer := 494;
38     constant HIGHC : integer := 523;
39 begin
40
41 getIncrement: process(clk, key_in)
42 begin
43
44     if rising_edge(clk) then
45         if (key_in(7) = '1') then
46             increment <=std_logic_vector(to_unsigned(LOWC*PHASECONSTANT/CLKFREQ,ACCUMSIZE));
47         elsif (key_in(6) = '1') then
48             increment <=std_logic_vector(to_unsigned(D*PHASECONSTANT/CLKFREQ,ACCUMSIZE));
49         elsif (key_in(5) = '1') then
```

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

Figure 15: keyboardTB.vhd

```
--------------------------------------------------------------------------------
-- Company: ENGS031 14X
-- Engineer: Daniel Chen and Vivian Hu
--
-- Create Date:   14:17:41 08/25/2014
-- Design Name:   Octave-Keyboard Testbench
-- Module Name:   C:/Users/F000JW7/Desktop/octave-keyboard/OctaveKeyboard/keyboardTB.vhd
-- Project Name:  OctaveKeyboard
-- Target Device:  Nexys3
-- Tool versions:
-- Description:   Testbench for Nexys3
--
-- VHDL Test Bench Created by ISE for module: OctaveKeyboardTop
--------------------------------------------------------------------------------
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;

   --Inputs
   signal clk : std_logic := '0';
   signal keys : std_logic_vector(7 downto 0) := (others => '0');
   signal led_disable : std_logic := '0';
   signal song_enable : std_logic := '0';

     --Outputs
   signal tone : std_logic;
   signal shutdown : std_logic;
   signal led_out : std_logic_vector(7 downto 0);

   -- Clock period definitions
   constant clk_period : time := 10 ns;

```

```
BEGIN
    -- Instantiate the Unit Under Test (UUT)
   uut: OctaveKeyboardTop PORT MAP (
          clk => clk,
          keys => keys,
          led_disable => led_disable,
          song_enable => song_enable,
          tone => tone,
          shutdown => shutdown,
          led_out => led_out
        );

   -- Clock process definitions
   clk_process :process
   begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
   end process;


   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;

      wait for clk_period*5;

      -- start with both switches off
      led_disable <= '0';
      song_enable <= '0';

      -- try some keys
      keys <= "10000000";
      wait for 250us;
      keys <= "10001000";
      wait for 50us;
      keys <= "00010000";
      wait for 200us;
      keys <= "00100010";
      wait for 300us;
      led_disable <= '1'; -- check that disabling leds work
      wait for 200us;
      keys <= "00000010";
      wait for 200us;
      keys <= "00000100";
      wait for 200us;
      keys <= "00001000";
      wait for 200us;
      keys <= "00011111";
```

```
102         led_disable <= '0'; -- see that enabling them again works
103         wait for 50us;
104         keys <= "00000000";
105         wait for 200us;
106         keys <= "10111000";
107 
108         -- just see that state changes correctly
109         -- (autoplay takes too long to simulate, as each beat is 1/4 second)
110         song_enable <= '1';
111         wait;
112     end process;
113 END;
```

7.2.3 Resource utilization

Figure 16: 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 17: 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 18: 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

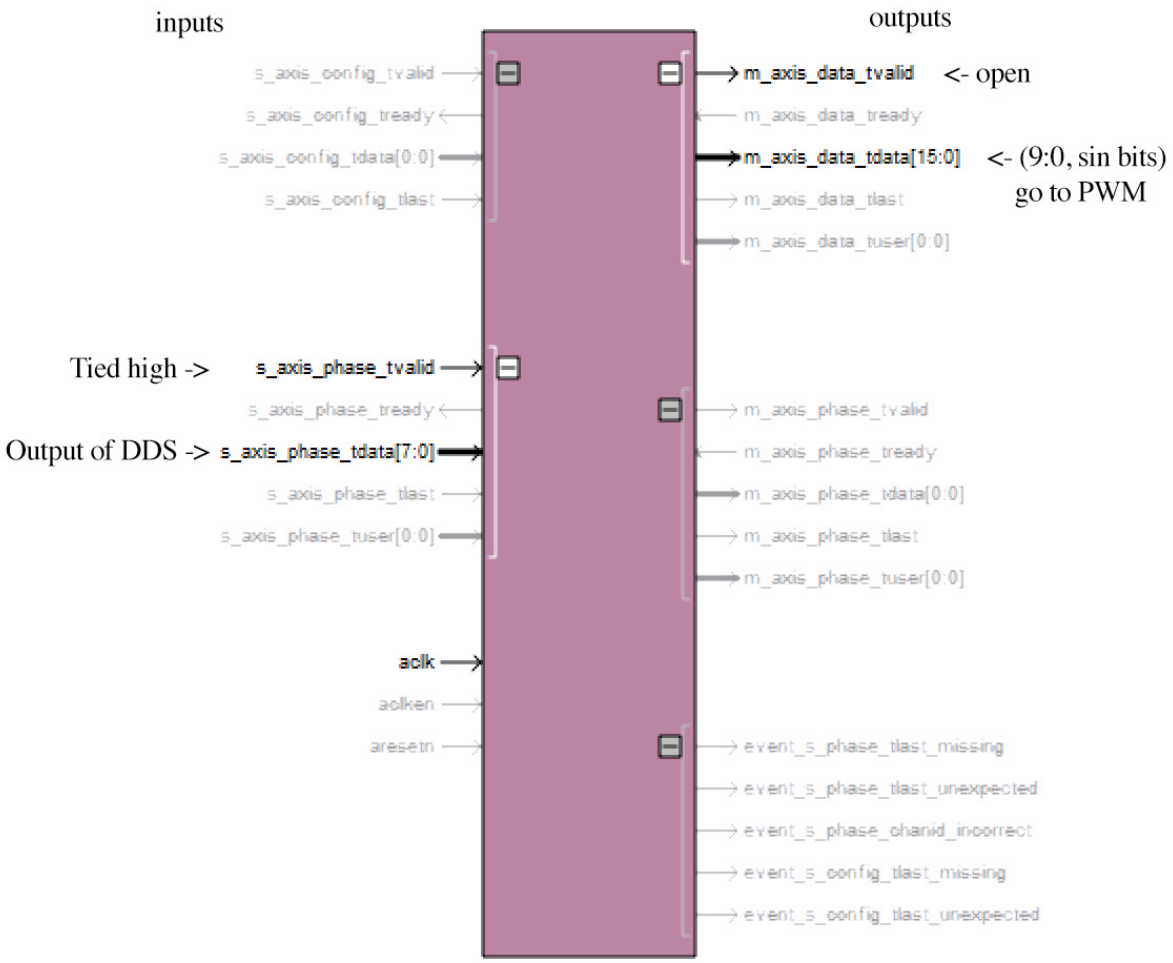


Figure 19: Memory Map for Sine LUT

7.4 Timing Diagram

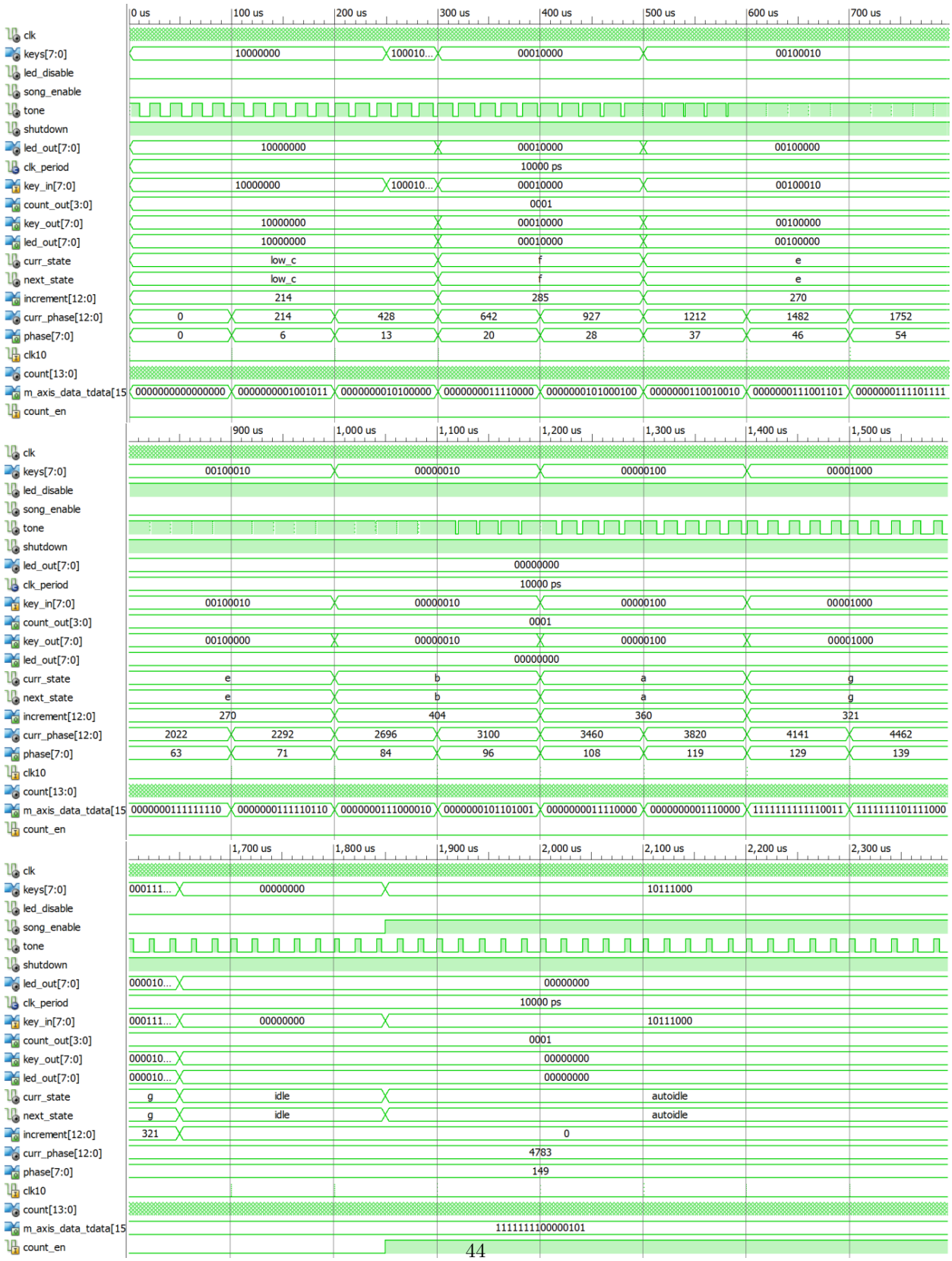


Figure 20: Timing Diagram of Test Bench

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:

Figure 21: Critical Path information (excerpt from 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 Synthesis Warnings

Figure 22: Synthesis Warnings

```
WARNING:HDLCompiler:89 -
        "\\psf\home\Documents\ENGS031\finalproject\OctaveKeyboard\OctaveKeyboardTop.vhd"
        Line 95: <debounce> remains a black-box since it has no binding entity.
WARNING:Xst:1710 -
        FF/Latch <increment_9> (without init value) has a constant value of 0 in block
        <FreqLUT>. This FF/Latch will be trimmed during the optimization process.
WARNING:Xst:1895 -
        Due to other FF/Latch trimming, FF/Latch <increment_10> (without init value)
        has a constant value of 0 in block <FreqLUT>.
        This FF/Latch will be trimmed during the optimization process.
WARNING:Xst:1895 -
        Due to other FF/Latch trimming, FF/Latch <increment_11> (without init value)
        has a constant value of 0 in block <FreqLUT>.
        This FF/Latch will be trimmed during the optimization process.
WARNING:Xst:1895 -
        Due to other FF/Latch trimming, FF/Latch <increment_12> (without init value)
        has a constant value of 0 in block <FreqLUT>.
        This FF/Latch will be trimmed during the optimization process.
WARNING:Xst:1293 -
        FF/Latch <PulseWM/count_13> has a constant value of 0 in block
        <OctaveKeyboardTop>.
        This FF/Latch will be trimmed during the optimization process.
WARNING:Xst:1293 -
        FF/Latch <kidsCounter/clkcount_31> has a constant value of 0 in block
        <OctaveKeyboardTop>.
        This FF/Latch will be trimmed during the optimization process.
WARNING:Xst:1293 -
        FF/Latch <kidsCounter/clkcount_30> has a constant value of 0 in block
        <OctaveKeyboardTop>.
        This FF/Latch will be trimmed during the optimization process.
...
same error for clkcount_29 through clkcount_26
...
WARNING:Xst:1293 -
        FF/Latch <kidsCounter/clkcount_25> has a constant value of 0 in block
        <OctaveKeyboardTop>.
        This FF/Latch will be trimmed during the optimization process.
WARNING:Xst:1293 -
        FF/Latch <kidsCounter/clkcount_24> has a constant value of 0 in block
        <OctaveKeyboardTop>.
        This FF/Latch will be trimmed during the optimization process.
```

None of the warnings from synthesis pose practical issues. The FF/Latch trimming is due to the usage of integers and simply warns us that bits 31 to 24 of clkcount and 13 of count will not be represented in the final program, as they are not used. The first warning, regarding the debouncer core, is resolved in a later step before generating the programming file.

7.6 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.

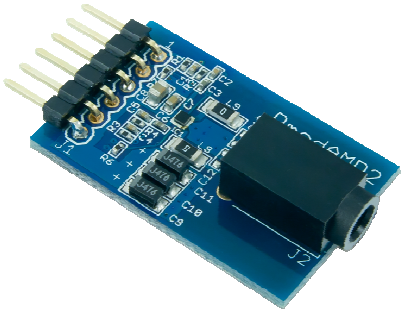
For customer convenience, Digilent has an inexpensive speaker and enclosure available for sale that is suitable for use with the PmodAMP2. Also, unlike most Digilent Pmod modules that accept only digital inputs, the PmodAMP2 accepts analog inputs and pulse width modulated digital inputs.

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

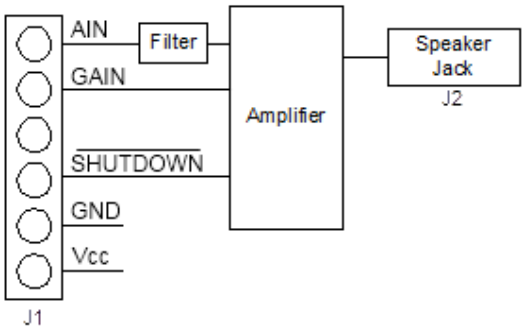


Figure 23: PmodAMP2 Data Sheet Page 1/2

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 24: PmodAMP2 Data Sheet Page 2/2

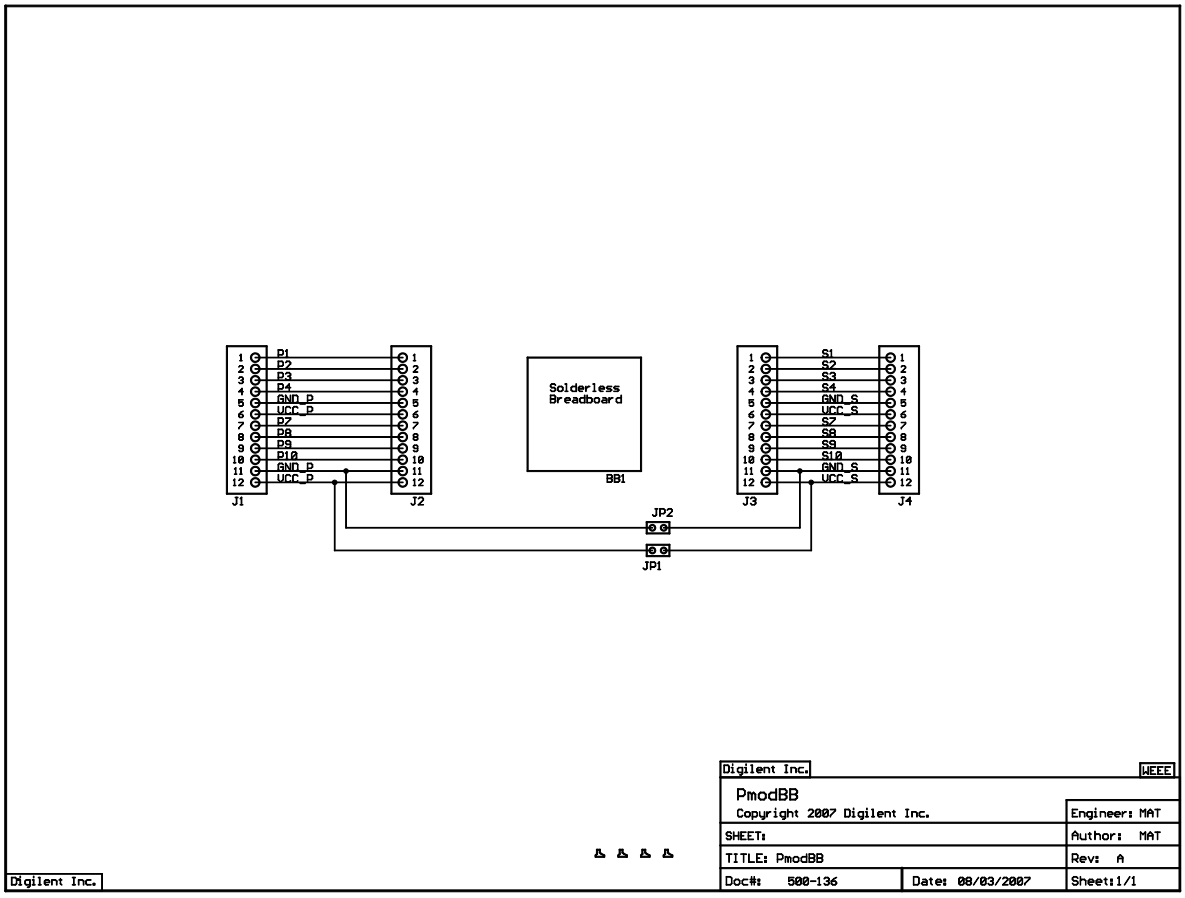


Figure 25: PmodBB Schematic