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

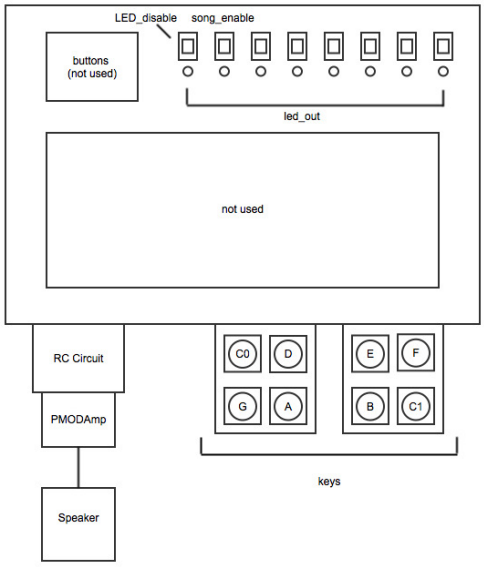
2.1 Specifications

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.

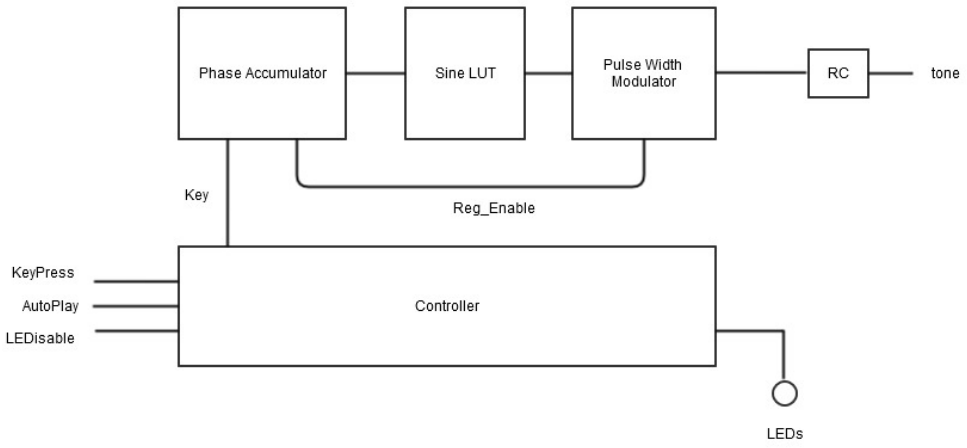
The outputs to the circuit are:

- 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...

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.

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.

2.4 Construction and Debugging

To build the circuit, we began by building and testing the DDS and PWM. The

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

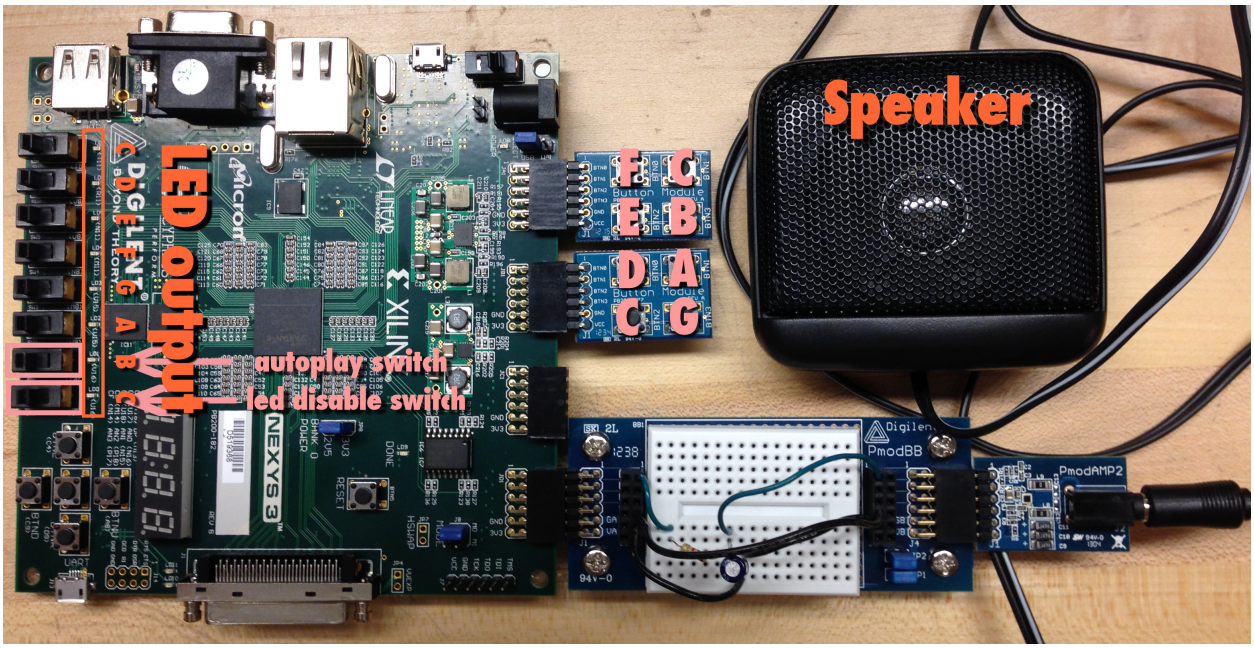


Figure 1: Annotated Digital Photo of Project

7.1.2 Block Diagram

7.1.3 Schematic Diagram

7.1.4 Package Map

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 2: State Diagram Defaults

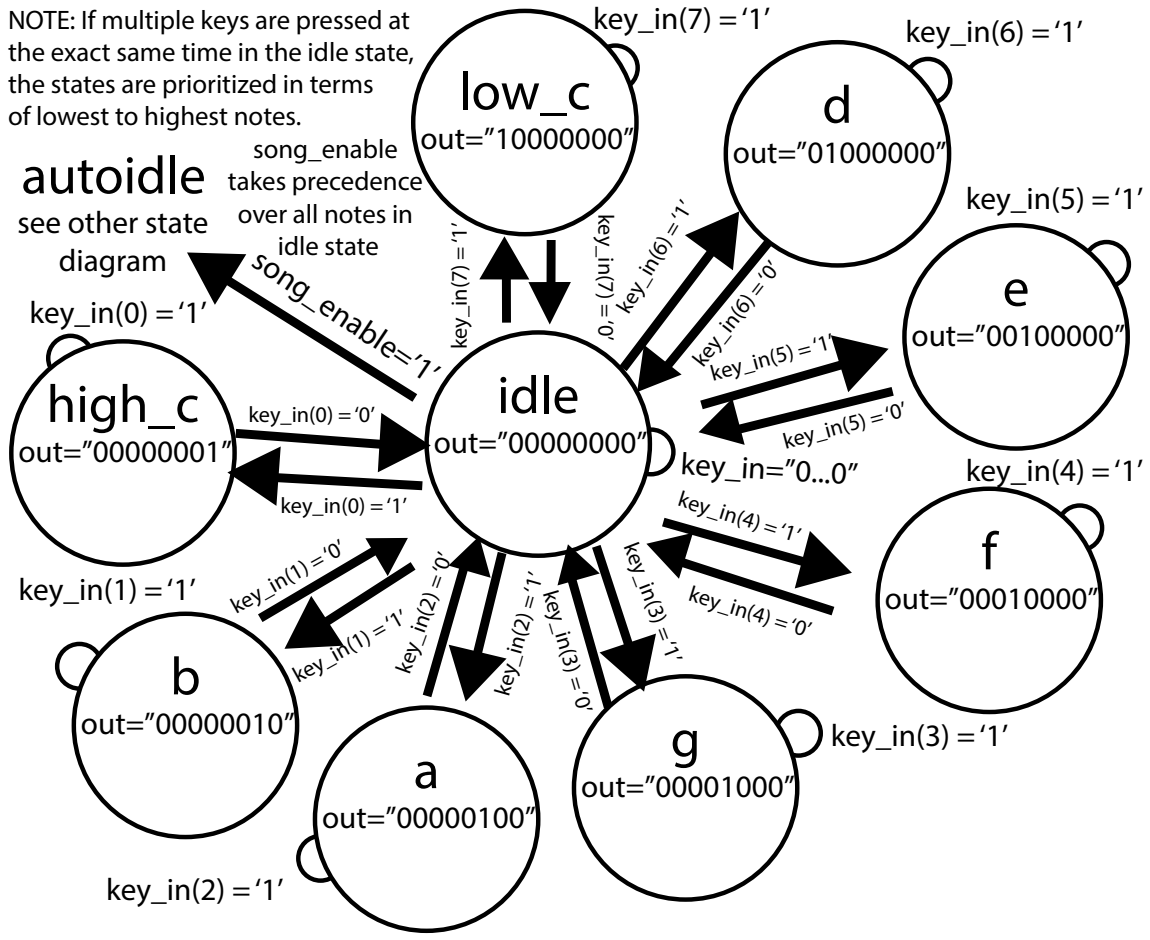


Figure 3: User Play State Diagram

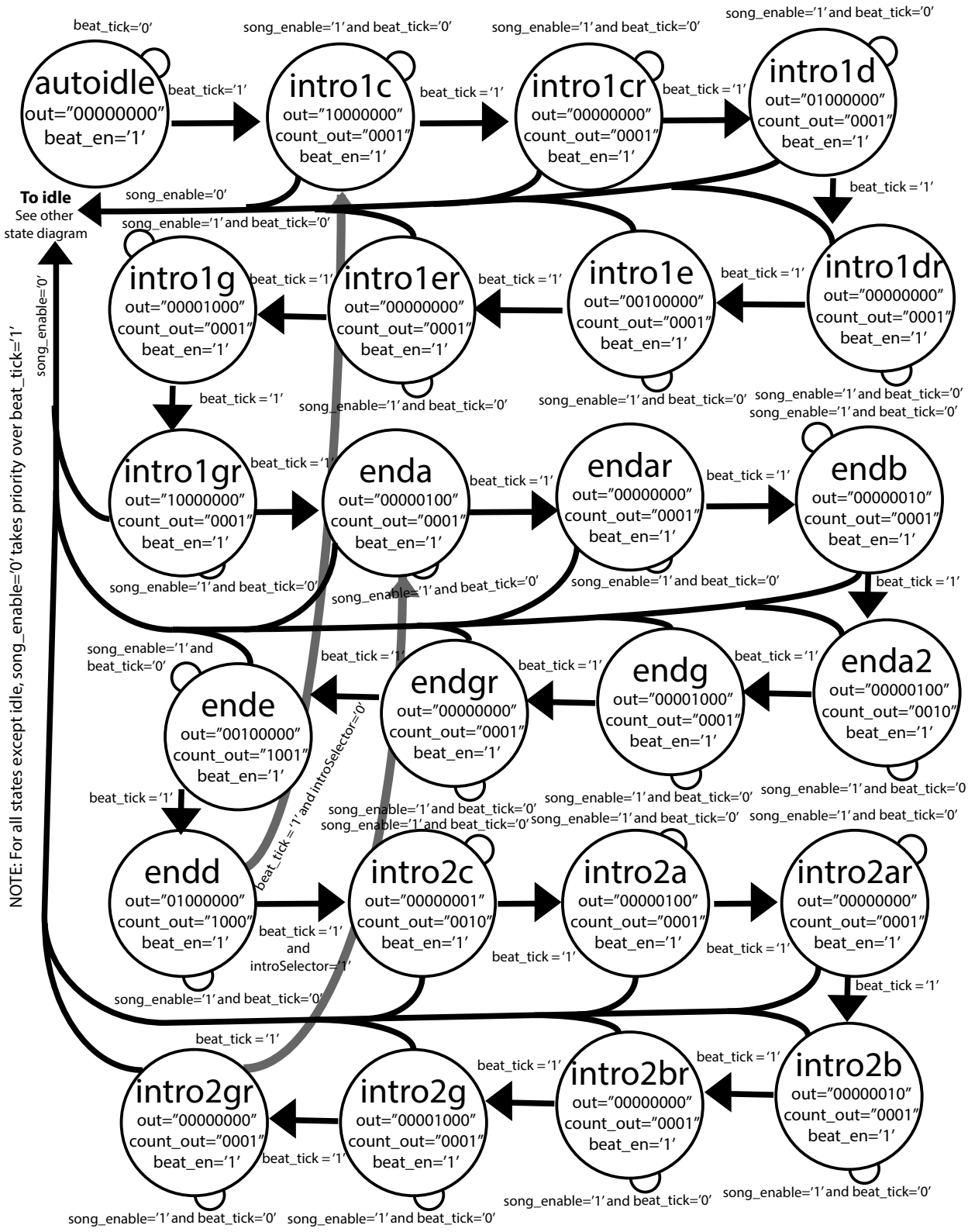


Figure 4: Autoplay State Diagram

7.2.2 VHDL Code

Figure 5: VHDL for Controller

```
----------------------------------------------------------------------------------
-- 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';
```

```
47  begin
48
49      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, beat_tick, song_enable, in
57      begin
58          -- defaults
59          next_state <= curr_state;
60          output <= (others => '0');
61          key_out <= output;
62          count_out <= "0001";
63          repeat_tick <= '0';
64          beat_en <= '0';
65
66          if (led_disable = '1') then
67              led_out <= (others => '0');
68          else
69              led_out <= output;
70          end if;
71
72          case curr_state is
73
74              when idle =>
75
76                  if song_enable = '1' then
77                      next_state <= autoidle;
78                  elsif key_in(7) = '1' then
79                      next_state <= low_c;
80                  elsif key_in(6) = '1' then
81                      next_state <= d;
82                  elsif key_in(5) = '1' then
83                      next_state <= e;
84                  elsif key_in(4) = '1' then
85                      next_state <= f;
86                  elsif key_in(3) = '1' then
87                      next_state <= g;
88                  elsif key_in(2) = '1' then
89                      next_state <= a;
90                  elsif key_in(1) = '1' then
91                      next_state <= b;
92                  elsif key_in(0) = '1' then
93                      next_state <= high_c;
94                  else
95                      next_state <= idle;
96                  end if;
97
98              when low_c =>
```

```
99                  output <= "10000000";
100                 if key_in(7) = '0' then
101                     next_state <= idle;
102                 end if;
103
104             when d =>
105                 output <= "01000000";
106                 if key_in(6) = '0' then
107                     next_state <= idle;
108                 end if;
109
110             when e =>
111                 output <= "00100000";
112                 if key_in(5) = '0' then
113                     next_state <= idle;
114                 end if;
115
116             when f =>
117                 output <= "00010000";
118                 if key_in(4) = '0' then
119                     next_state <= idle;
120                 end if;
121
122             when g =>
123                 output <= "00001000";
124                 if key_in(3) = '0' then
125                     next_state <= idle;
126                 end if;
127
128             when a =>
129                 output <= "00000100";
130                 if key_in(2) = '0' then
131                     next_state <= idle;
132                 end if;
133
134             when b =>
135                 output <= "00000010";
136                 if key_in(1) = '0' then
137                     next_state <= idle;
138                 end if;
139
140             when high_c =>
141                 output <= "00000001";
142                 if key_in(0) = '0' then
143                     next_state <= idle;
144                 end if;
145
146             when autoidle =>
147                 beat_en <= '1';
148                 output <= (others => '0');
149                 if (beat_tick = '1') then
150                     next_state <= intro1c;
```

```
151                 end if;
152
153             when intro1c =>
154                 beat_en <= '1';
155                 output <= "10000000";
156                 count_out <= "0001";
157                 if (song_enable = '0') then
158                     next_state <= idle;
159                 elsif(beat_tick = '1') then
160                     next_state <= intro1cr;
161                 end if;
162
163
164             when intro1cr =>
165                 beat_en <= '1';
166                 output <= "00000000";
167                 count_out <= "0001";
168                 if (song_enable = '0') then
169                     next_state <= idle;
170                 elsif(beat_tick = '1') then
171                     next_state <= intro1d;
172                 end if;
173
174
175             when intro1d =>
176                 beat_en <= '1';
177                 output <= "01000000";
178                 count_out <= "0001";
179                 if (song_enable = '0') then
180                     next_state <= idle;
181                 elsif(beat_tick = '1') then
182                     next_state <= intro1dr;
183                 end if;
184
185
186             when intro1dr =>
187                 beat_en <= '1';
188                 output <= "00000000";
189                 count_out <= "0001";
190                 if (song_enable = '0') then
191                     next_state <= idle;
192                 elsif(beat_tick = '1') then
193                     next_state <= intro1e;
194                 end if;
195
196
197             when intro1e =>
198                 beat_en <= '1';
199                 output <= "00100000";
200                 count_out <= "0001";
201                 if (song_enable = '0') then
202                     next_state <= idle;
```

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

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

```
306
307             when ende =>
308                 beat_en <= '1';
309                 output <= "00100000";
310                 count_out <= "1001";
311                 if (song_enable = '0') then
312                     next_state <= idle;
313                 elsif(beat_tick = '1') then
314                     next_state <= endd;
315                 end if;
316
317
318             when endd =>
319                 beat_en <= '1';
320                 output <= "01000000";
321                 count_out <= "1000";
322
323                 if (beat_tick = '1') then
324                     repeat_tick <= '1';
325
326                     if (introSelector = '0') then
327                         next_state <= intro1c;
328                     else
329                         next_state <= intro2c;
330                     end if;
331
332                 elsif (song_enable = '0') then
333                     next_state <= idle;
334
335                 end if;
336
337             when intro2c =>
338                 beat_en <= '1';
339                 output <= "00000001";
340                 count_out <= "0010";
341                 if (song_enable = '0') then
342                     next_state <= idle;
343                 elsif(beat_tick = '1') then
344                     next_state <= intro2a;
345                 end if;
346
347
348             when intro2a =>
349                 beat_en <= '1';
350                 output <= "00000100";
351                 count_out <= "0001";
352                 if (song_enable = '0') then
353                     next_state <= idle;
354                 elsif(beat_tick = '1') then
355                     next_state <= intro2ar;
356                 end if;
357
```

```
358
359             when intro2ar =>
360                 beat_en <= '1';
361                 output <= "00000000";
362                 count_out <= "0001";
363                 if (song_enable = '0') then
364                     next_state <= idle;
365                 elsif(beat_tick = '1') then
366                     next_state <= intro2b;
367                 end if;
368
369
370             when intro2b =>
371                 beat_en <= '1';
372                 output <= "00000010";
373                 count_out <= "0001";
374                 if (song_enable = '0') then
375                     next_state <= idle;
376                 elsif(beat_tick = '1') then
377                     next_state <= intro2br;
378                 end if;
379
380
381             when intro2br =>
382                 beat_en <= '1';
383                 output <= "00000000";
384                 count_out <= "0001";
385                 if (song_enable = '0') then
386                     next_state <= idle;
387                 elsif(beat_tick = '1') then
388                     next_state <= intro2g;
389                 end if;
390
391
392             when intro2g =>
393                 beat_en <= '1';
394                 output <= "00001000";
395                 count_out <= "0001";
396                 if (song_enable = '0') then
397                     next_state <= idle;
398                 elsif(beat_tick = '1') then
399                     next_state <= intro2gr;
400                 end if;
401
402
403             when intro2gr =>
404                 beat_en <= '1';
405                 output <= "00000000";
406                 count_out <= "0001";
407                 if (song_enable = '0') then
408                     next_state <= idle;
409                 elsif(beat_tick = '1') then
```

```
                    next_state <= enda;
                end if;

            when others =>
                next_state <= idle;

        end case;

    end process CombLogic;

    RepeatCounter: process(clk, repeat_tick, reps, introselector)
    begin
        if (rising_edge(clk)) then
            if (repeat_tick = '1') then
                if (reps = '1') then
                    introSelector <= not introSelector;
                    reps <= '0';
                else
                    reps <= not reps;
                end if;
            end if;
        end if;
    end process RepeatCounter;

end Behavioral;
```

Figure 6: VHDL for DDS

Figure 7: VHDL for PWM

Figure 8: VHDL for PlayCount

Figure 9: VHDL for FreqLUT

7.2.3 Resource utilization

Figure 10: 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 11: 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

7.3 Memory Map

7.4 Timing Diagram

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.

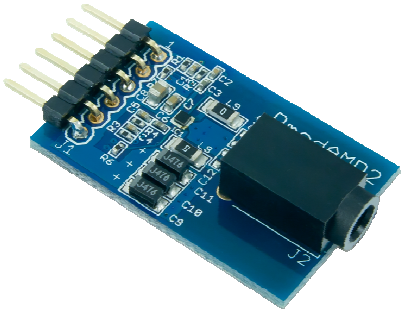
For customer convenience, Digilent has an inexpensive speaker and enclosure available for sale that is suitable for use with the PmodAMP2. Also, unlike most Digilent Pmod modules that accept only digital inputs, the PmodAMP2 accepts analog inputs and pulse width modulated digital inputs.

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

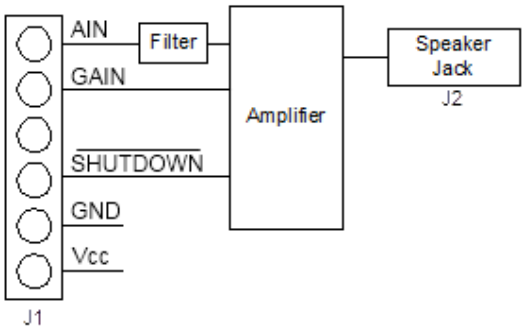


Figure 12: PmodAMP2 Data Sheet Page 1/2

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 13: PmodAMP2 Data Sheet Page 2/2

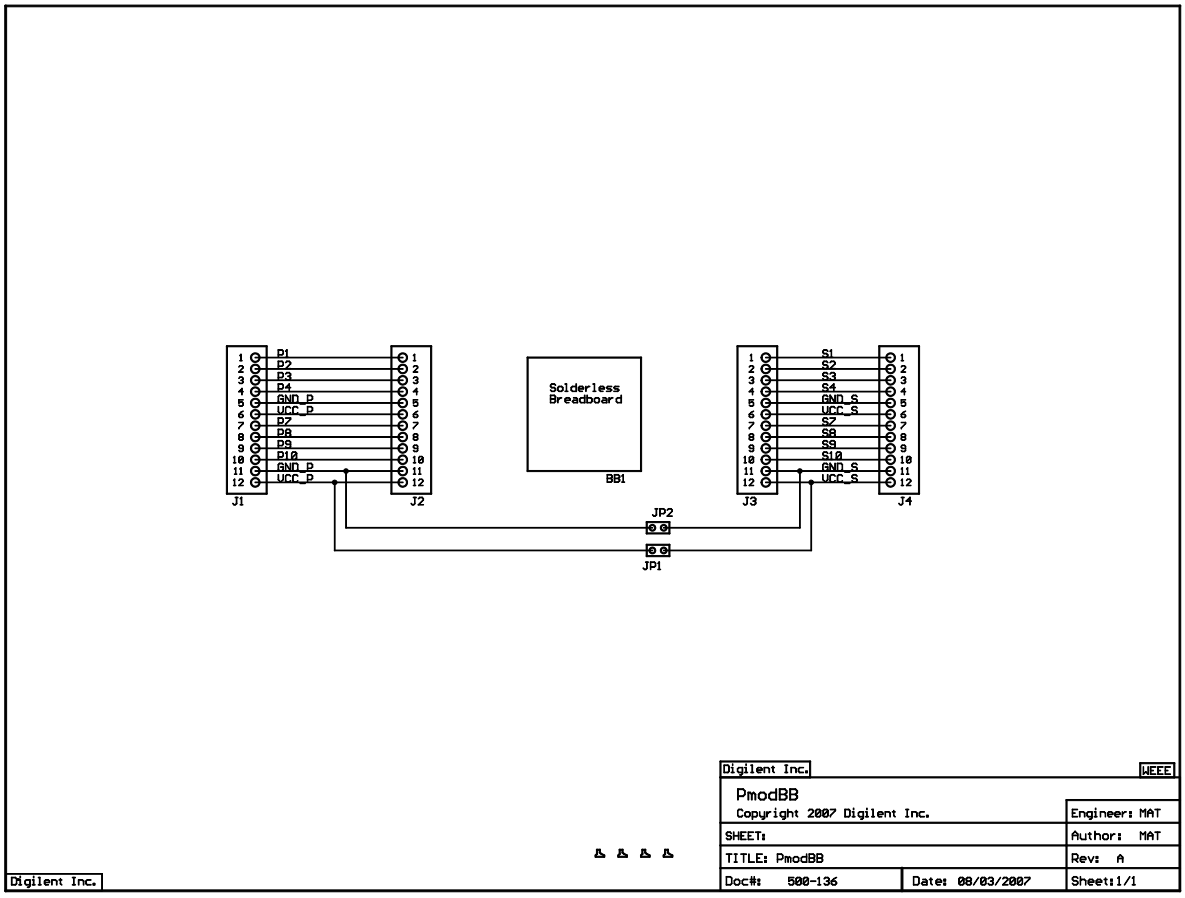


Figure 14: PmodBB Schematic