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.

Contents

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

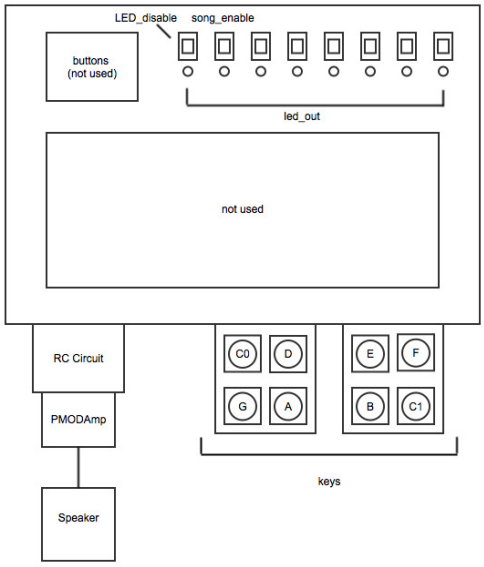
2.1 Specifications

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.

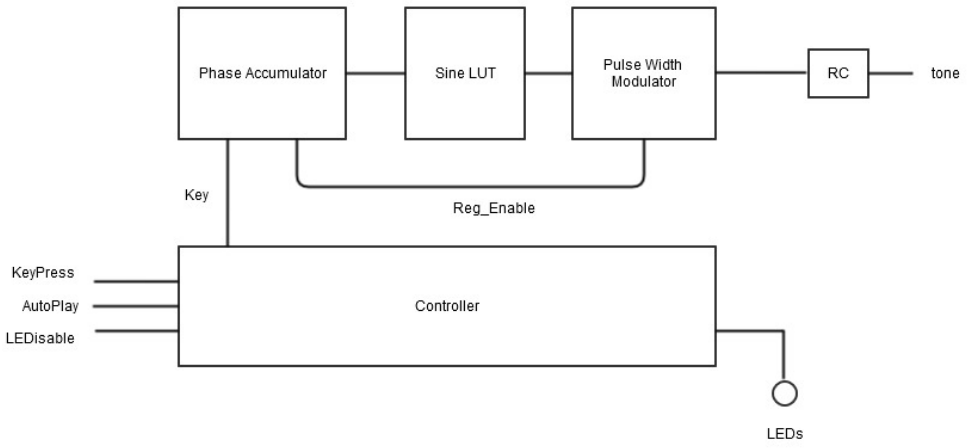
The outputs to the circuit are:

- 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.

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

7.1 System level diagrams

7.1.1 Front Panel

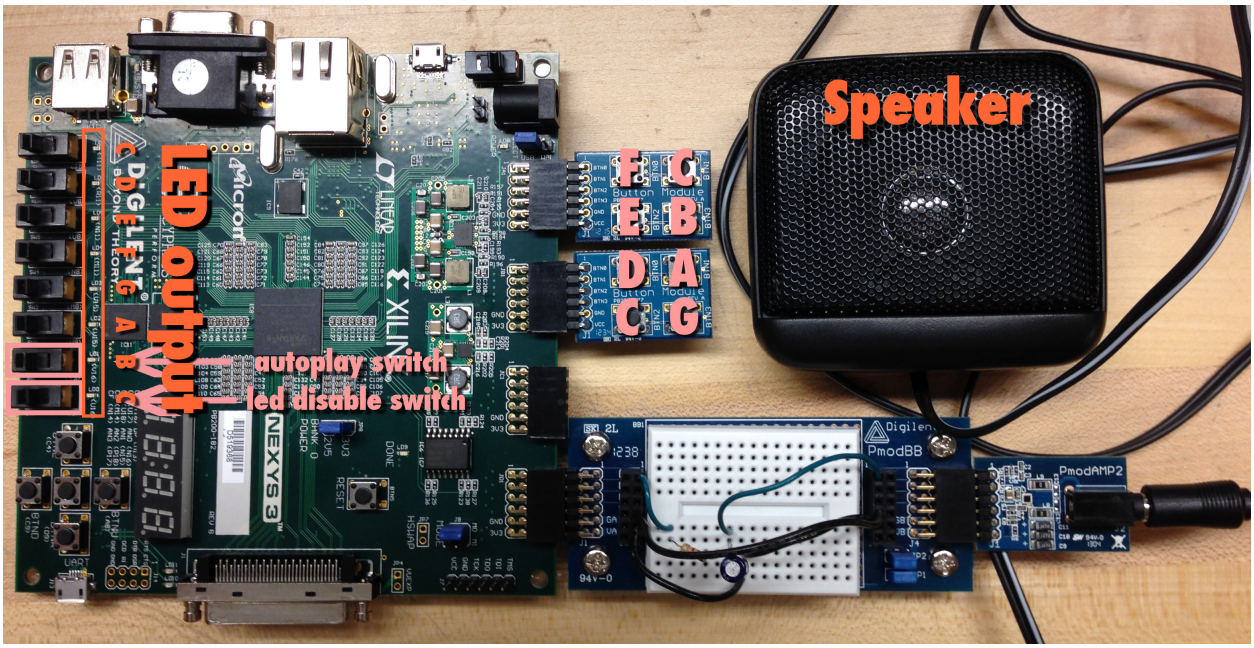


Figure 1: Annotated Digital Photo of Project

7.1.2 Block Diagram

7.1.3 Schematic Diagram

7.1.4 Package Map

7.1.5 External Components

- Design of
- The second item
- The third etc

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

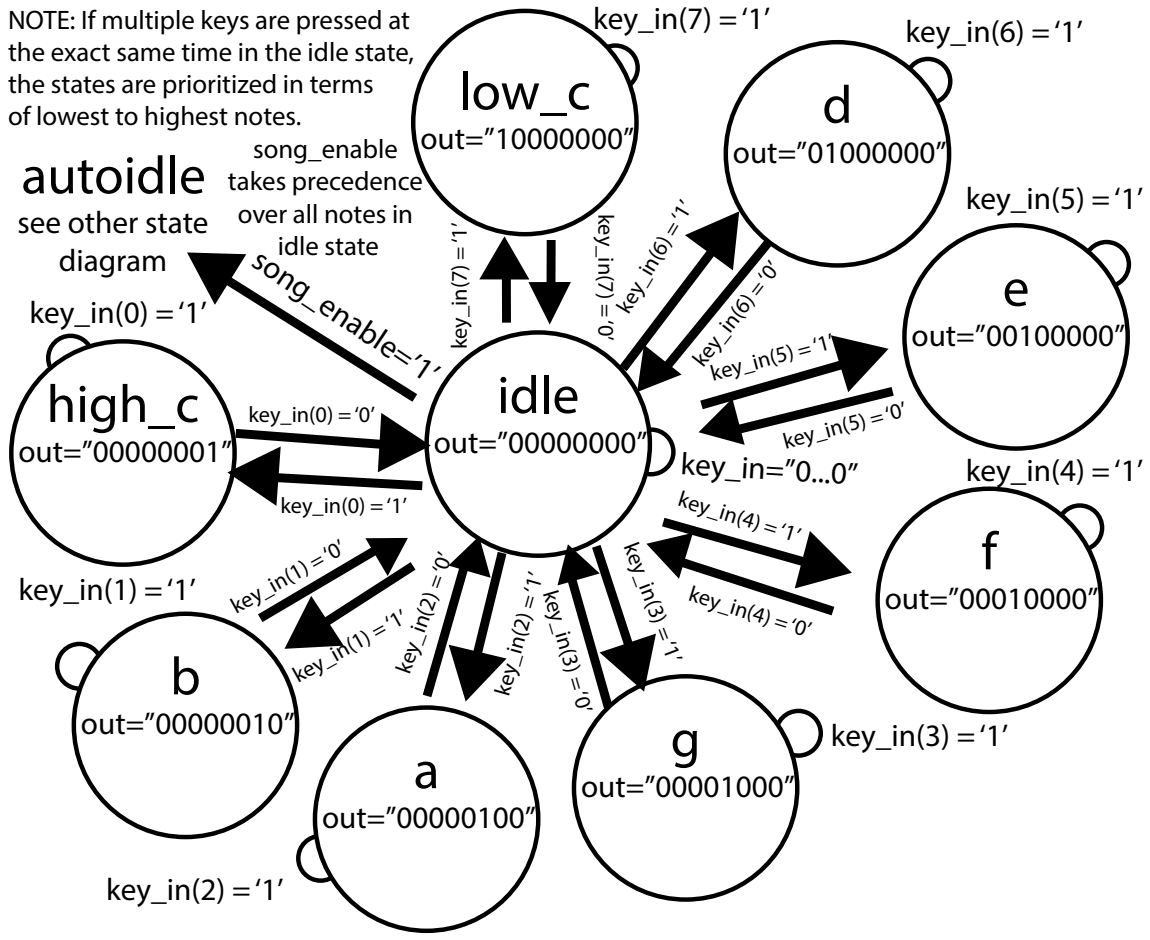


Figure 3: User Play State Diagram

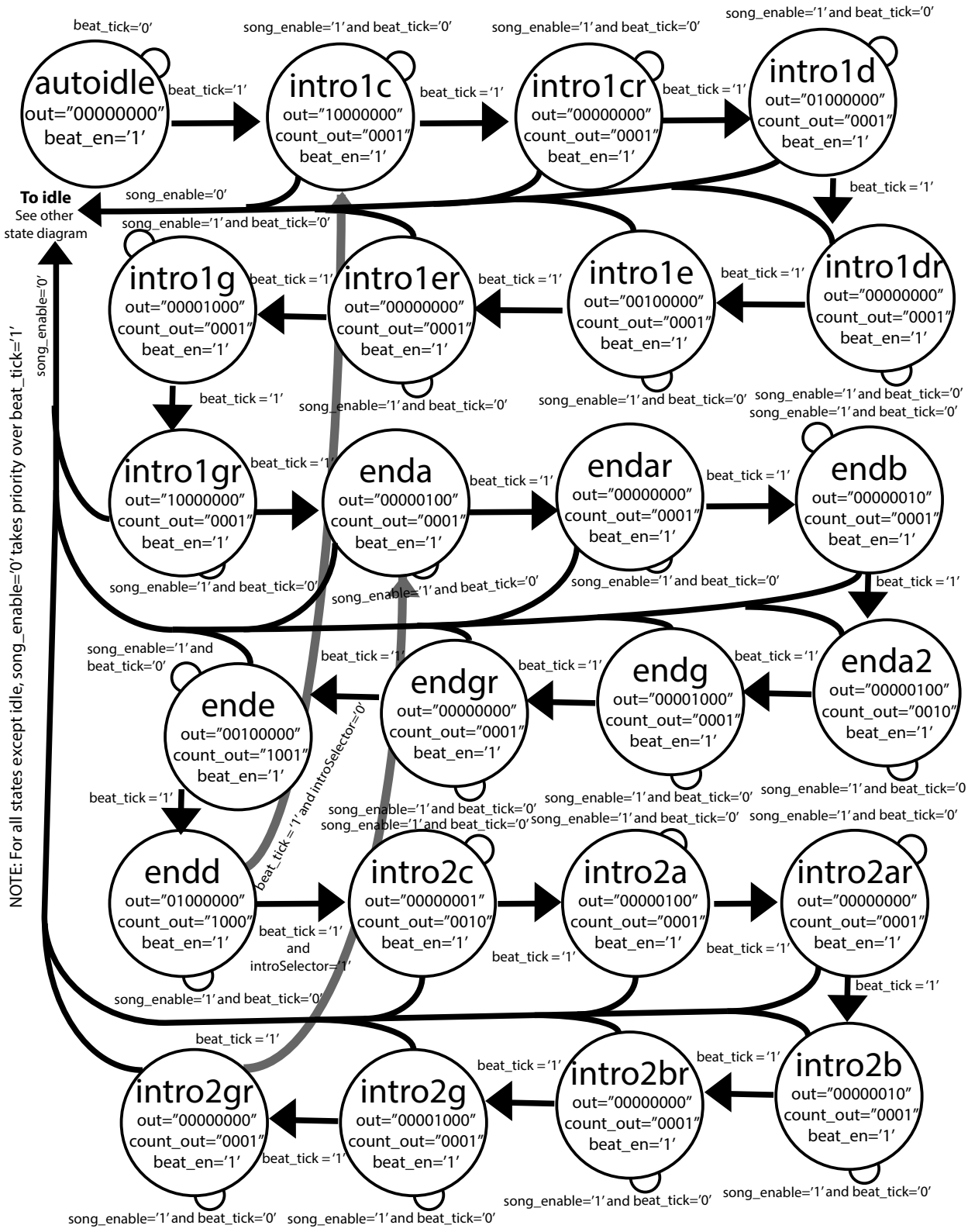


Figure 4: Autoplay State Diagram

7.2.2 VHDL Code

Header comments removed.

Figure 5: VHDL for Controller

```
  process
  begin
    CLK <= '1'; wait for 10 NS;
    CLK <= '0'; wait for 10 NS;
  end process;
```

Figure 6: VHDL for DDS

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

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

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

architecture Behavioral of DDS is
  signal curr_phase : unsigned(ACCUMSIZE-1 downto 0) := (others => '0');
begin

  AccumPhase: process(clk, clk10)
  begin
    if (rising_edge(clk)) then
      if (clk10 = '1') then
        curr_phase <= curr_phase + unsigned(step);
      end if;
    end if;
  end process AccumPhase;

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

Figure 7: VHDL for PWM

```
process
begin
  CLK <= '1'; wait for 10 NS;
  CLK <= '0'; wait for 10 NS;
end process;
```

Figure 8: VHDL for PlayCount

```
process
begin
  CLK <= '1'; wait for 10 NS;
  CLK <= '0'; wait for 10 NS;
end process;
```

Figure 9: VHDL for FreqLUT

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

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

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

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

  getIncrement: process(key_in)
  begin

    if (key_in(7) = '1') then
      increment <= std_logic_vector(to_unsigned(LOWC * PHASECONSTANT / CLKFREQ, ACCUMSIZE));
    elsif (key_in(6) = '1') then
      increment <= std_logic_vector(to_unsigned(D * PHASECONSTANT / CLKFREQ, ACCUMSIZE));
    elsif (key_in(5) = '1') then
      increment <= std_logic_vector(to_unsigned(E * PHASECONSTANT / CLKFREQ, ACCUMSIZE));
    elsif (key_in(4) = '1') then
      increment <= std_logic_vector(to_unsigned(F * PHASECONSTANT / CLKFREQ, ACCUMSIZE));
    elsif (key_in(3) = '1') then
      increment <= std_logic_vector(to_unsigned(G * PHASECONSTANT / CLKFREQ, ACCUMSIZE));
    elsif (key_in(2) = '1') then
      increment <= std_logic_vector(to_unsigned(A * PHASECONSTANT / CLKFREQ, ACCUMSIZE));
    elsif (key_in(1) = '1') then
      increment <= std_logic_vector(to_unsigned(B * PHASECONSTANT / CLKFREQ, ACCUMSIZE));
    elsif (key_in(0) = '1') then
      increment <= std_logic_vector(to_unsigned(HIGHC * PHASECONSTANT / CLKFREQ, ACCUMSIZE));
    else
      increment <= (others => '0');
    end if;

  end process getIncrement;
end Behavioral;
```

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%
```

7.3 Memory Map

7.4 Timing Diagram