Engs 31/CS 56 Final Project

Alex Martinez and Daniel Kim

June 4th, 2018

Abstract

Everyday advancements in technology propel the world into profound innovation, but they also create a veil of ignorance around the inner workings of our most simple appliances. In response, our project exposes the inner workings of a digital appliance we tend to overlook, a keyboard. We plan to explain the functionality of a polyphonic keyboard by analyzing the fundamentals of converting an electronic signal into a keynote, while also providing an option for sustainable sound.

Table of Contents

1 Introduction

Our project aims to answer the question behind the enigma of sound. Specifically, how is sound produced, and how can we control it?

2 Design solution

2.1 Specifications

Our circuit provides switches corresponding to 8 keys on a keyboard (C4 to C5) as possible inputs. Once one switch is flicked, the switch produces a signal that corresponds to a constant number added to a counter. Each input has a respective counter that feeds this number into its own respective look up table (LUT), which matches the value of the constant with positions on a sine wave. The larger the value of the constant the quicker the sine wave completes its cycle resulting in a higher frequency. If multiple keys are pressed, the sine wave outputs of all lookup tables are fed through to the adder that adds up each of the amplitudes to create a chord like wave and then passes this signal to the normalizer to control the volume of the chord. This normalized wave is then passed to a digital to analog converter, which feeds it into an amplifier and finally to a speaker which produces the corresponding frequency.

2.2 Operating Instructions

Materials:

1. Basys 3 FPGA Trainer Board w/ cable
2. Digilent PmodDA2 – Digital to Analog Converter
3. Digilent PmodAMP2 – Amplifier
4. Adapter
5. Speaker

Assembly:

First, connect the microUSB side of the FPGA cable to the FPGA programming port (PROG). Next, attach the amplifier to the adapter by connecting its pins to the AMP_2 INT port on the adapter. Then, connect the adapter to the D/A converter by attaching J2 port side pins of the D/A converter to the AD_INT port on the adapter. Then attach the remaining pins of the D/A converter (on its J1 port) to the top half of the JA port on the left-hand side of the FPGA board. Finally, attach the speaker cable to the J2 port on the amplifier. This completes the assembly of the digital keyboard.

Procedure:

To play the keyboard, flick any of the switches. The octave corresponds as so:

Low C → W13 (on switch panel)
D → W14 (on switch panel)
E → V15 (on switch panel)
F → W15 (on switch panel)
G → W17 (on switch panel)
A → W16 (on switch panel)
B → V16 (on switch panel)
High C → V17 (on switch panel)

2.3 Theory of Operation

Keyboard Multiplexer

This multiplexer chooses which signal to send to the lookup table, based on which switch was flicked. When a user presses a key, it assigns a counter increment variable (m) a value that corresponds to that desired key's appropriate frequency through the equation $F_{m\,=}$ mFs/N.

Frequency Counter

The frequency counter receives the value of m from the keyboard multiplexer, then determines the address that will be sent to the lookup table. This address determines the frequency of the sine wave that is output from the lookup table, which corresponds to different pitches. If multiple keys are pressed then the counter sends the m value of each key pressed to different instantiations of the lookup table. If the sustain enable signal is high and a key was pressed the next state will continue to be key pressed even if the key is released.

Piano Shell

Our Shell file contained the architecture of our entire data path. It mapped each component to its next respective component, by setting the outputs of a previous component to the inputs of the next component. In this file we also contained 8 instantiations of our frequency counter and 8 instantiations of our lookup table (corresponding to our 8 keys).

Lookup Table

The sine wave lookup table was generated through Vivado’s IP Digital Synthesis core. We have chosen the size of the table to be 4096. It takes as input the address from the frequency counter, and outputs a digital sine wave. The input address determines the step size through which the sine wave is generated, thus determining the wave frequency.

Sine Wave Adder

This component handles the possibility that multiple keys are pressed so then multiple waves have been produced through the instantiations of the LUT. The adder combines the amplitude of all wave forms to generate a chord like frequency.

Volume

This component maps the number of waves accumulated in the Adder, to its corresponding multiplier value in a multiplier array. This value controls the volume of the outputting sound. The more waves accumulated, the lower the value is to level its volume.

Digital to Analog Converter

The Digital to Analog Converter programs the pmod-da2 hardware of the board. It takes as input the digital signal created by the lookup table. It converts the digital sine wave to an analog sound output that is audible to the human ear.

2.4 Construction and Debugging

Construction

We built the digital to analog converter first. We roughly followed the template from Lab 5, but applied the reverse to get it working the other way around.

We tested this using a similar test bench that we used to test lab 5, verifying the input and output operations.

We then built the sine lookup table using the Vivado IP cores following the Canvas instructions. This was pretty straightforward, and we used the testbench provided by Vivado for testing.

Then, we built the frequency counter to handle the multiple switch inputs. We handled the different inputs in a case statement and assigned their values to a counter increment variable which was counted then assigned to a lookup table input variable. This variable was fed into the lookup table to match to an appropriate sine wave. We tested this counter using a testbench we created in EDA playground. We had to hard code the take sample signals, which are naturally clocked in our Vivado program.

We then build the Adder and Volume Files. These were constructed after we achieved monophonic sound. These files were the contributions to creating polyphonic sound. We, tested them without a testbench because we already reached the point of creating sound, so we relied on the hardware at this stage in the process.

Debugging

Initially our design consisted of only one lookup table that received a signal from the frequency counter to produce a single sine wave. We then realized that this method would not work for the option that multiple keys are pressed. Our key iterator added up the constant counter incrementers together and then passed this value to the lookup table which would not produce a corresponding chord, instead it would only produce a very high pitch sound because it just adds together the frequencies and not the amplitudes.

After consulting with our learning fellow, Robert Deangelo, he suggested that we create different instantiations of our lookup table to handle the case that multiple keys would be pressed. After creating different sine waves, the amplitudes of these waves would be added together then normalized to control its volume.

After we figured out the polyphonic aspect of the keyboard, we attempted to add a sustain pedal. Holding the sustain high would enable the speaker to continue to output a sound even if the corresponding keys were turned low. We initially had problems because our multiplexer was asynchronous, and hence could not hold the value of *m*. Professor

Luke pointed this out, so we changed our multiplexer to a synchronous design so that we could hold the value of *m*. This fixed the problem and we were able to add an extra functionality to the keyboard that we had not initially planned.

3 Justification and evaluation

Another attempt at our project could have been to not instantiate 8 different counters or lookup tables, but rather place a register after a single lookup table that accumulated all waves as they were outputted by the table. This design was our original intention and could have reduced repetition in our code.

This solution could have been cleaner but was harder to implement than our design. Our instantiations were easy to create and made our project easier to visualize.

4 Conclusions

The goal of our project was to understand the process of how a signal could be generated into sound. Our original proposal intended to create single and multiple key sounds, and our final project achieved this and a sutain enable signal that withheld a sound even after a key was released. However, even though we achieved polyphonic sound we did not reach our objective of generating polyphonic sound that could take in any amount of keys. As stated in our residual warning analysis section, we had to hard code in a maximum amount of three keys for polyphonic sound.

Our advice to future groups trying to recreate this project would be to focus on creating single key sounds first before trying to implement polyphonic sound, but have an idea of how you would create polyphonic sound so you don't have to completely redesign your project.

5 Acknowledgments

We would like to thank David Picard, our professor Geoffrey Luke, and our Learning Fellow Robert Deangelo. David helped us with general debugging, and he also created the D/A to Amplifier adapter (Gain) we used in our keyboard. Professor Luke helped us conceptualize the datapath and improved our initial project proposal. Robert Deangelo helped us solve the problem to the polyphonic aspect to our design regarding multiple lookup tables.

Daniel Kim created the DtoA, Volume, Adder and Piano Shell files. Alex Martinez created the Frequency Counter file and the write up.

6 References

The frequencies we decided to use for our keyboard were found off an online magazine called Nuts & Volts. The article was written by G.Y. Xu.

We also utilized the Digilent PmodDA2 Reference Manual for help with interfacing.

We have provided links to our references below:

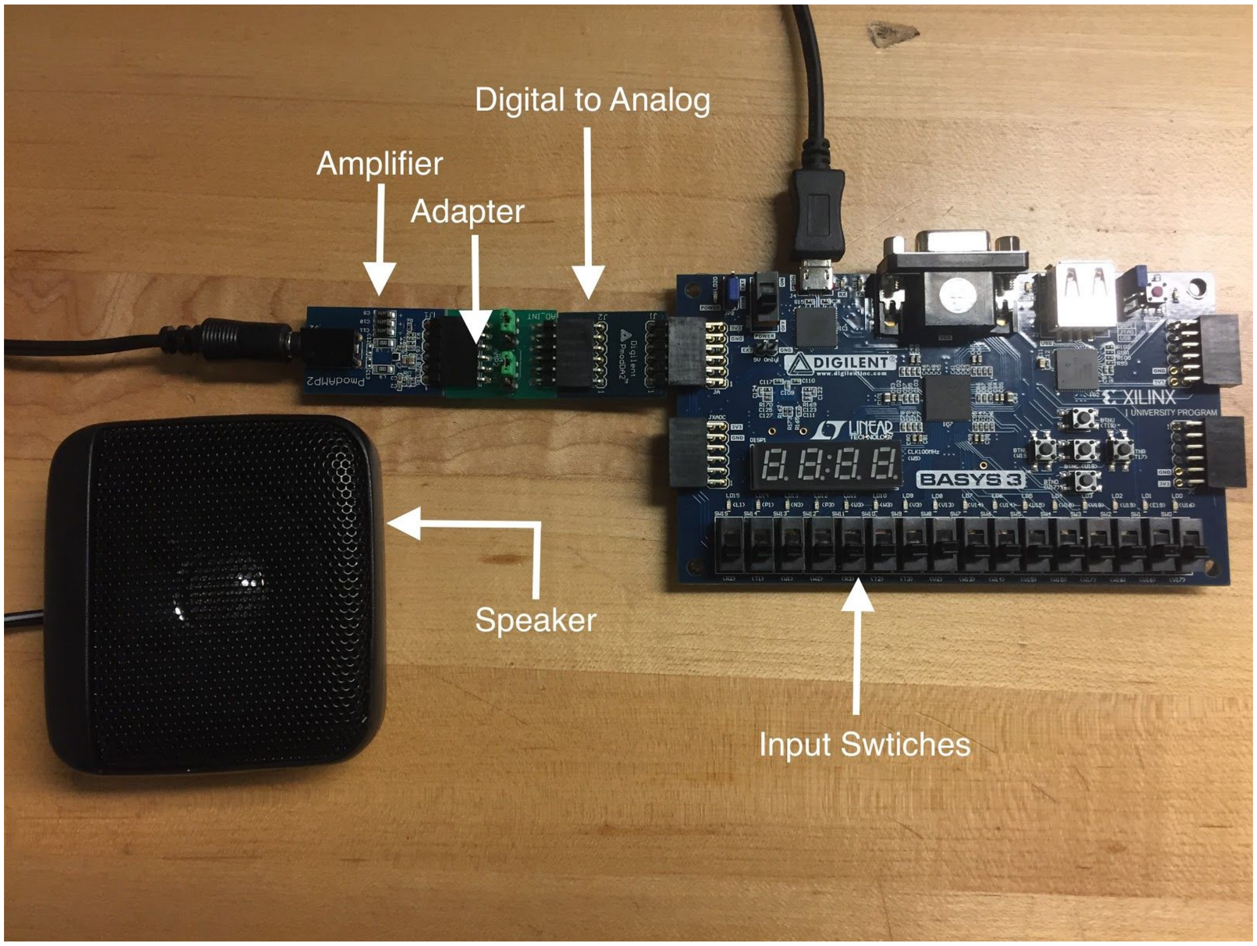
http://www.nutsvolts.com/magazine/article/chip_music_composing_simplified

https://reference.digilentinc.com/_media/reference/pmod/pmodda2/pmodda2_rm.pdf

7 Appendices

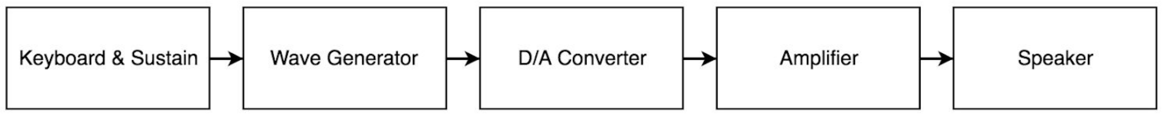
7.1 System Level Diagrams

Front Panel

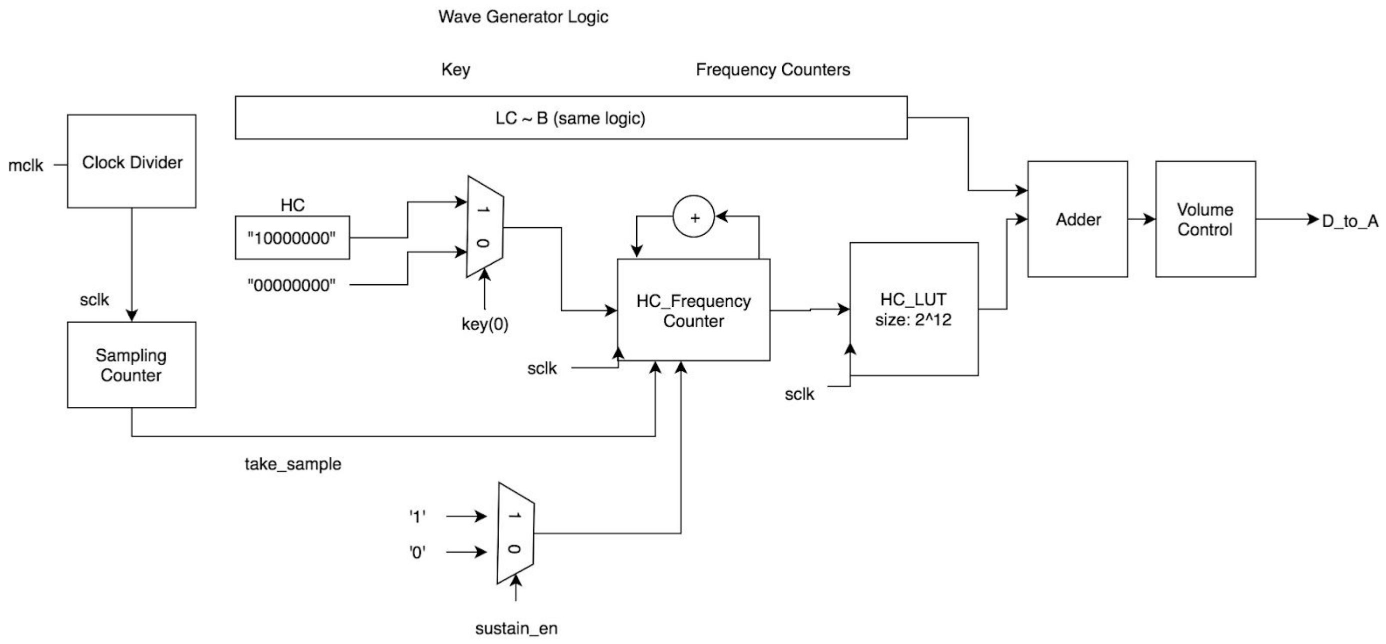


Functional Block Diagram

Top Level



Wave Generator Logic



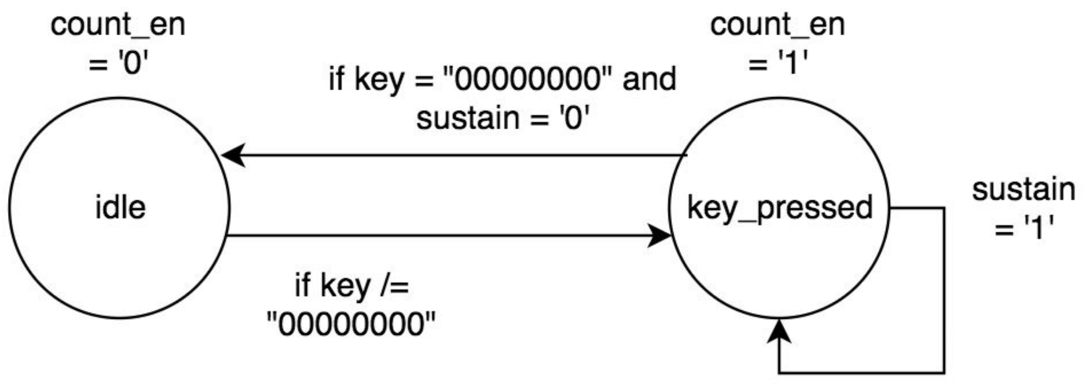
Parts List

Part	Quantity	Part Number	Description
Digital to Analog Converter	1	DA2	Digilent PmodDA2: Two 12-bit A/D Inputs
Basys3	1	Basys3	Digilent Basys3 board
Speaker	1		
Amplifier	1	Amp2	Digilent PmodAmp2: Audio Amplifier
Pmod Adapter (Gain)	1		Thayer School Pmod Adapter

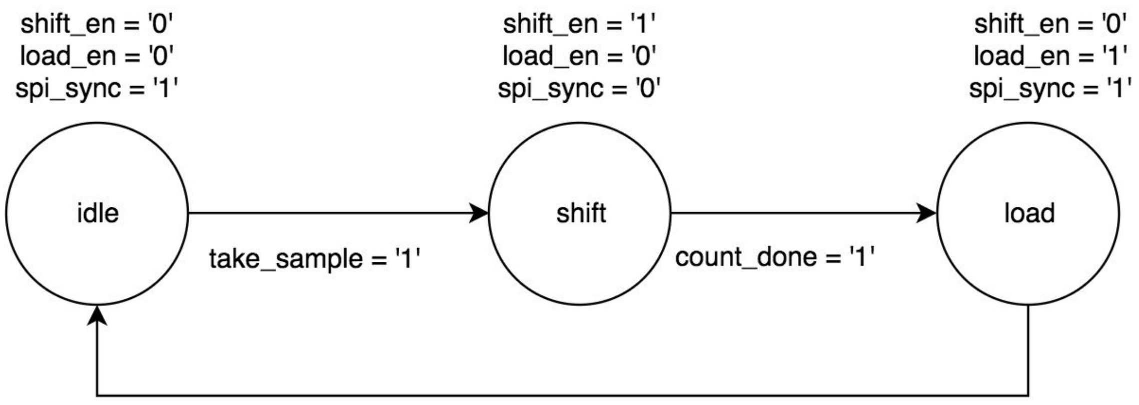
7.2 Programmed Logic

State Diagrams

Frequency Generator



Digital to Analog



VHDL Code

PIANO SHELL FILE

```
----------------------------------------------------------------
------------------
-- Company: ENGS 31
-- Engineer: Daniel Kim, Alex Martinez
--
-- Create Date: 05/30/2018 09:10:47 PM
-- Design Name:
-- Module Name: piano_shell - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------
------------------


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.ALL;               -- needed for arithmetic

library UNISIM;                              -- needed for the
BUFG component
use UNISIM.Vcomponents.ALL;

entity Piano is
port (    mclk :    in std_logic;      -- FPGA board master
clock (100 MHz)
          key       :    in std_logic_vector(7 downto 0);
          sustain_en    :   in std_logic;
        sound  :    out std_logic;
        spi_sclk    :   out std_logic;
        spi_sync    :   out std_logic);
end Piano;

ARCHITECTURE behavior of Piano is

-- Constants and Signals
-- Wires
signal wire_to_freqLC    :   std_logic_vector(7 downto 0);
signal wire_to_freqD    :   std_logic_vector(7 downto 0);
signal wire_to_freqE    :   std_logic_vector(7 downto 0);
signal wire_to_freqF    :   std_logic_vector(7 downto 0);
signal wire_to_freqG    :   std_logic_vector(7 downto 0);
signal wire_to_freqA    :   std_logic_vector(7 downto 0);
signal wire_to_freqB    :   std_logic_vector(7 downto 0);
signal wire_to_freqHC    :   std_logic_vector(7 downto 0);

signal wire_to_lookupLC   :   std_logic_vector(15 downto 0);
signal wire_to_lookupD   :   std_logic_vector(15 downto 0);
signal wire_to_lookupE   :   std_logic_vector(15 downto 0);
signal wire_to_lookupF   :   std_logic_vector(15 downto 0);
signal wire_to_lookupG   :   std_logic_vector(15 downto 0);
signal wire_to_lookupA   :   std_logic_vector(15 downto 0);
signal wire_to_lookupB   :   std_logic_vector(15 downto 0);
signal wire_to_lookupHC   :   std_logic_vector(15 downto 0);

signal wire_to_adderLC   :   std_logic_vector(15 downto 0);
signal wire_to_adderD   :   std_logic_vector(15 downto 0);
signal wire_to_adderE   :   std_logic_vector(15 downto 0);
signal wire_to_adderF   :   std_logic_vector(15 downto 0);
signal wire_to_adderG   :   std_logic_vector(15 downto 0);
signal wire_to_adderA   :   std_logic_vector(15 downto 0);
signal wire_to_adderB   :   std_logic_vector(15 downto 0);
signal wire_to_adderHC   :   std_logic_vector(15 downto 0);

signal wire_to_volume   :   std_logic_vector(15 downto 0);
signal wire_to_pmod     :   std_logic_vector(15 downto 0);
signal num_waves   :   unsigned(3 downto 0) := (others => '0');

signal s_axis_phase_tvalid1  :   std_logic := '1';
signal m_axis_data_tvalid1   : std_logic := '1';
signal s_axis_phase_tvalid2  :   std_logic := '1';
signal m_axis_data_tvalid2   : std_logic := '1';
signal s_axis_phase_tvalid3  :   std_logic := '1';
signal m_axis_data_tvalid3   : std_logic := '1';
signal s_axis_phase_tvalid4  :   std_logic := '1';
signal m_axis_data_tvalid4   : std_logic := '1';
signal s_axis_phase_tvalid5  :   std_logic := '1';
signal m_axis_data_tvalid5   : std_logic := '1';
signal s_axis_phase_tvalid6  :   std_logic := '1';
signal m_axis_data_tvalid6   : std_logic := '1';
signal s_axis_phase_tvalid7  :   std_logic := '1';
signal m_axis_data_tvalid7   : std_logic := '1';
signal s_axis_phase_tvalid8  :   std_logic := '1';
signal m_axis_data_tvalid8   : std_logic := '1';

-- Signals for serial clock divider (100 MHz --> 10 MHz)
constant SCLK_DIVIDER_VALUE : integer := 5;
constant COUNT_LEN : integer := 50;
signal sclkdiv: unsigned(count_LEN-1 downto 0) := (others =>
'0'); -- clock divider counter
signal sclk_unbuf: std_logic := '0';    --unbuffered serial
clock
signal sclk: std_logic := '0';          --internal serial clock

-- Signals for sampling clock
signal take_sample  : std_logic := '0';
signal count    : unsigned(16 downto 0) := (others => '0');

-- Component Declarations
COMPONENT FrequencyCounter
PORT(     clk             :    in std_logic;
          key             :    in   std_logic_vector(7 downto 0);
          sustain_en    :   in std_logic;
          take_sample   :   in std_logic;
          lut_input    :      out  std_logic_vector(15 downto
0));
END COMPONENT;

COMPONENT dds_compiler_0
  PORT (
    aclk : IN STD_LOGIC;
    s_axis_phase_tvalid : IN STD_LOGIC;
    s_axis_phase_tdata : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    m_axis_data_tvalid : OUT STD_LOGIC;
    m_axis_data_tdata : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
  );
END COMPONENT;

COMPONENT Adder
PORT (    clk       :   in std_logic;
          waveLC    :   in std_logic_vector(15 downto 0);
          waveD     :   in std_logic_vector(15 downto 0);
          waveE     :   in std_logic_vector(15 downto 0);
          waveF     :   in std_logic_vector(15 downto 0);
        waveG   :   in std_logic_vector(15 downto 0);
        waveA   :   in std_logic_vector(15 downto 0);
        waveB  :   in std_logic_vector(15 downto 0);
        waveHC   :   in std_logic_vector(15 downto 0);
        wave_sum:   out std_logic_vector(15 downto 0)
          );
END COMPONENT;

COMPONENT Volume
  PORT (
    clk        :   in std_logic;
    wave_in    :   in std_logic_vector(15 downto 0);
    num_waves   :  in std_logic_vector(3 downto 0);
    wave_out    :    out std_logic_vector(15 downto 0)
  );
END COMPONENT;

COMPONENT DtoA
PORT (    sclk      :    in   std_logic;
          take_sample    :    in   std_logic;
        data_in          :    in   std_logic_vector(15 downto
0);
        spi_sclk    :    out std_logic;
        spi_sync    :    out  std_logic;
        spi_DinA    :    out  std_logic);
END COMPONENT;

begin

-- Processes
-- Clock buffer for sclk
Slow_clock_buffer: BUFG
    port map ( I => sclk_unbuf,
               O => sclk);

-- Divide the 100 MHz clock down to 20 Mhz, then toggling flip
flop gives final 10 MHz system clock
Serial_clock_divider: process(mclk)
begin
    if rising_edge(mclk) then
        if sclkdiv = SCLK_DIVIDER_VALUE-1 then
            sclkdiv <= (others => '0');
            sclk_unbuf <= NOT(sclk_unbuf);
        else
            sclkdiv <= sclkdiv + 1;
        end if;
    end if;
end process Serial_clock_divider;

-- Further divide clock down to 44.1 kHz take_sample ticks
Sampling_counter: process(sclk)
begin
    if rising_edge(sclk) then
        take_sample <= '0';
        count <= count + 1;
        if (count = 227) then
            count <= (others => '0');
            take_sample <= '1';
        end if;
    end if;
end process;

--muxr for keypress
key_select: process(key)
begin
    if key(0) = '1' then
        wire_to_freqLC <= "00000001";
    else wire_to_freqLC <= "00000000";
    end if;
    if key(1) = '1' then
        wire_to_freqD <= "00000010";
    else wire_to_freqD <= "00000000";
    end if;
    if key(2) = '1' then
        wire_to_freqE <= "00000100";
    else wire_to_freqE <= "00000000";
    end if;
    if key(3) = '1' then
        wire_to_freqF <= "00001000";
    else wire_to_freqF <= "00000000";
    end if;
    if key(4) = '1' then
        wire_to_freqG <= "00010000";
    else wire_to_freqG <= "00000000";
    end if;
    if key(5) = '1' then
        wire_to_freqA <= "00100000";
    else wire_to_freqA <= "00000000";
    end if;
    if key(6) = '1' then
        wire_to_freqB <= "01000000";
    else wire_to_freqB <= "00000000";
    end if;
    if key(7) = '1' then
        wire_to_freqHC <= "10000000";
    else wire_to_freqHC <= "00000000";
    end if;
end process;

num_keys: process(sclk)
begin
    if rising_edge(sclk) then
        num_waves <= (others => '0');
        for i in 0 to 7 loop
            if key(i) = '1' then
                num_waves <= num_waves + 1;
            end if;
        end loop;
    end if;
end process;

-- Counter Instantiations
freq_counterLC: FrequencyCounter port map(
    clk => sclk,
    key => wire_to_freqLC,
    sustain_en => sustain_en,
    take_sample => take_sample,
    lut_input => wire_to_lookupLC
    );

freq_counterD: FrequencyCounter port map(
    clk => sclk,
    key => wire_to_freqD,
    sustain_en => sustain_en,
    take_sample => take_sample,
    lut_input => wire_to_lookupD
    );

freq_counterE: FrequencyCounter port map(
    clk => sclk,
    key => wire_to_freqE,
    sustain_en => sustain_en,
    take_sample => take_sample,
    lut_input => wire_to_lookupE
    );
```

```
freq_counterF: FrequencyCounter port map(
        clk => sclk,
        key => wire_to_freqF,
        sustain_en => sustain_en,
        take_sample => take_sample,
        lut_input => wire_to_lookupF
        );

freq_counterG: FrequencyCounter port map(
            clk => sclk,
            key => wire_to_freqG,
            sustain_en => sustain_en,
            take_sample => take_sample,
            lut_input => wire_to_lookupG
            );

freq_counterA: FrequencyCounter port map(
    clk => sclk,
    key => wire_to_freqA,
    sustain_en => sustain_en,
    take_sample => take_sample,
    lut_input => wire_to_lookupA
    );

freq_counterB: FrequencyCounter port map(
    clk => sclk,
    key => wire_to_freqB,
    sustain_en => sustain_en,
    take_sample => take_sample,
    lut_input => wire_to_lookupB
    );

freq_counterHC: FrequencyCounter port map(
        clk => sclk,
        key => wire_to_freqHC,
        sustain_en => sustain_en,
        take_sample => take_sample,
        lut_input => wire_to_lookupHC
        );
```

```
--LUT instantiations
lookup_tableLC: dds_compiler_0 port map(
    aclk => sclk,
    s_axis_phase_tvalid => s_axis_phase_tvalid1,
    s_axis_phase_tdata => wire_to_lookupLC,
    m_axis_data_tvalid => m_axis_data_tvalid1,
    m_axis_data_tdata => wire_to_adderLC);

lookup_tableD: dds_compiler_0 port map(
        aclk => sclk,
        s_axis_phase_tvalid => s_axis_phase_tvalid2,
        s_axis_phase_tdata => wire_to_lookupD,
        m_axis_data_tvalid => m_axis_data_tvalid2,
        m_axis_data_tdata => wire_to_adderD);

lookup_tableE: dds_compiler_0 port map(
    aclk => sclk,
    s_axis_phase_tvalid => s_axis_phase_tvalid3,
    s_axis_phase_tdata => wire_to_lookupE,
    m_axis_data_tvalid => m_axis_data_tvalid3,
    m_axis_data_tdata => wire_to_adderE);

lookup_tableF: dds_compiler_0 port map(
        aclk => sclk,
        s_axis_phase_tvalid => s_axis_phase_tvalid4,
        s_axis_phase_tdata => wire_to_lookupF,
        m_axis_data_tvalid => m_axis_data_tvalid4,
        m_axis_data_tdata => wire_to_adderF);

lookup_tableG: dds_compiler_0 port map(
    aclk => sclk,
    s_axis_phase_tvalid => s_axis_phase_tvalid5,
    s_axis_phase_tdata => wire_to_lookupG,
    m_axis_data_tvalid => m_axis_data_tvalid5,
    m_axis_data_tdata => wire_to_adderG);

lookup_tableA: dds_compiler_0 port map(
    aclk => sclk,
    s_axis_phase_tvalid => s_axis_phase_tvalid6,
    s_axis_phase_tdata => wire_to_lookupA,
    m_axis_data_tvalid => m_axis_data_tvalid6,
    m_axis_data_tdata => wire_to_adderA);

lookup_tableB: dds_compiler_0 port map(
    aclk => sclk,
    s_axis_phase_tvalid => s_axis_phase_tvalid7,
    s_axis_phase_tdata => wire_to_lookupB,
    m_axis_data_tvalid => m_axis_data_tvalid7,
    m_axis_data_tdata => wire_to_adderB);

lookup_tableHC: dds_compiler_0 port map(
    aclk => sclk,
    s_axis_phase_tvalid => s_axis_phase_tvalid8,
    s_axis_phase_tdata => wire_to_lookupHC,
    m_axis_data_tvalid => m_axis_data_tvalid8,
    m_axis_data_tdata => wire_to_adderHC);

add: Adder port map(
    clk => sclk,
    waveLC => wire_to_adderLC,
    waveD => wire_to_adderD,
    waveE => wire_to_adderE,
    waveF => wire_to_adderF,
    waveG => wire_to_adderG,
    waveA => wire_to_adderA,
    waveB => wire_to_adderB,
    waveHC => wire_to_adderHC,
    wave_sum => wire_to_volume);

vol: Volume port map(
    clk => sclk,
    num_waves => "0011",
    wave_in => wire_to_volume,
    wave_out => wire_to_pmod);

digital_to_analog: DtoA port map(
    sclk => sclk,
    take_sample => take_sample,
    data_in => wire_to_pmod,
    spi_sclk => spi_sclk,
    spi_sync => spi_sync,
    spi_DinA => sound);

end behavior;
```

FREQUENCY COUNTER FILE

```
----------------------------------------------------------------
------------------
-- Company: ENGS 31
-- Engineer: Daniel Kim, Alex Martinez
--
-- Create Date: 05/30/2018 09:03:00 PM
-- Design Name:
-- Module Name: frequency_counter - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------
------------------


library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY FrequencyCounter is
PORT (    clk          :   in std_logic;
          key           :   in std_logic_vector(7 downto 0);
          sustain_en    :   in std_logic;
          take_sample   :   in std_logic;
          lut_input     :   out std_logic_vector(15 downto 0));
end FrequencyCounter;

ARCHITECTURE behavior of FrequencyCounter is

--Constants
constant N          :    integer := 4096;    --2^12
constant Fs         :    integer := 44100;   --44.1 kHz

--Signals
signal count        :    unsigned(15 downto 0):= (others =>
'0');
signal count_int    :    integer := 0;
signal m            :    integer := 0;
signal count_en     :    std_logic:= '0';
type state_type is (idle, key_pressed);
signal current_state, next_state   : state_type := idle;

BEGIN

-- Multiplexer for switch pressed
MUXR: process(clk)
begin
    if rising_edge(clk) then
        case Key is
        --LowC: 262 Hz
        when "00000001" =>
          m <= 262 * N / Fs;
        --D: 294 Hz
        when "00000010" =>
          m <= 294 * N / Fs;
        --E: 330 Hz
        when "00000100" =>
          m <= 330 * N / Fs;
        --F: 350 Hz
        when "00001000" =>
          m <= 350 * N / Fs;
        --G: 392 Hz
        when "00010000" =>
          m <= 392 * N / Fs;
        --A: 440 Hz
        when "00100000" =>
          m <= 440 * N / Fs;
        --B: 494 Hz
        when "01000000" =>
          m <= 494 * N / Fs;
        --HighC: 523 Hz
        when "10000000" =>
          m <= 523 * N / Fs;
        when others =>
            if sustain_en = '1' then
                m <= m;
            else m <= 0;
            end if;
        end case;
    end if;
end process;

-- State update
current_to_next: process(clk)
begin
     if rising_edge(clk) then
     current_state <= next_state;
    end if;
end process;

-- FSM Controller
state_controller: process(current_state, key, sustain_en)
begin
     count_en <= '0';
     next_state <= current_state;
     case current_state is
    when idle =>
        if (key /= "00000000") then
            next_state <= key_pressed;
        end if;
    when key_pressed =>
     count_en <= '1';
     if sustain_en = '1' then
        next_state <= key_pressed;
     elsif (key = "00000000") then
          next_state <= idle;
        end if;
    end case;
end process;

-- Counter that generates address values
counter: process(clk)
begin
     if rising_edge(clk) then
        if take_sample = '1' then
            if count_en = '1' then
            count_int <= count_int + m;
            if (count_int >= N) then
                count_int <= 0;
            end if;
         else
             count_int <= 0;
              end if;
          end if;
    end if;
end process;

count <= to_unsigned(count_int, 16);
LUT_Input <= std_logic_vector(count);

end behavior;
```

ADDER FILE

```
----------------------------------------------------------------
------------------
-- Company: ENGS 31
-- Engineer: Daniel Kim, Alex Martinez
```

```
--
-- Create Date: 05/31/2018 10:50:23 PM
-- Design Name:
-- Module Name: Adder - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-----------------------------------------------------------------
------------------


library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY Adder is
PORT (    clk       :   in std_logic;
          waveLC    :   in std_logic_vector(15 downto 0);
          waveD     :   in std_logic_vector(15 downto 0);
          waveE     :   in std_logic_vector(15 downto 0);
          waveF     :   in std_logic_vector(15 downto 0);
        waveG   :   in std_logic_vector(15 downto 0);
        waveA   :   in std_logic_vector(15 downto 0);
        waveB  :   in std_logic_vector(15 downto 0);
        waveHC   :   in std_logic_vector(15 downto 0);
        wave_sum:   out std_logic_vector(15 downto 0)
          );
end Adder;

ARCHITECTURE behavior of Adder is
```

```
-- Signal
signal wave_unsignedLC   : unsigned(15 downto 0) := (others =>
'0');
signal wave_unsignedD    : unsigned(15 downto 0) := (others =>
'0');
signal wave_unsignedE    : unsigned(15 downto 0) := (others =>
'0');
signal wave_unsignedF    : unsigned(15 downto 0) := (others =>
'0');
signal wave_unsignedG    : unsigned(15 downto 0) := (others =>
'0');
signal wave_unsignedA    : unsigned(15 downto 0) := (others =>
'0');
signal wave_unsignedB    : unsigned(15 downto 0) := (others =>
'0');
signal wave_unsignedHC   : unsigned(15 downto 0) := (others =>
'0');
signal wave_sum_unsigned : unsigned(15 downto 0) := (others =>
'0');

BEGIN

add: process(clk)
begin
     if rising_edge(clk) then
        wave_sum_unsigned <= wave_unsignedLC + wave_unsignedD +
wave_unsignedE + wave_unsignedF + wave_unsignedG +
wave_unsignedA + wave_unsignedB + wave_unsignedHC;
     end if;
end process;

-- Inputs
wave_unsignedLC <= unsigned(waveLC);
wave_unsignedD <= unsigned(waveD);
wave_unsignedE <= unsigned(waveE);
wave_unsignedF <= unsigned(waveF);
wave_unsignedG <= unsigned(waveG);
wave_unsignedA <= unsigned(waveA);
wave_unsignedB <= unsigned(waveB);
```

```
wave_unsignedHC <= unsigned(waveHC);

-- Outputs
wave_sum <= std_logic_vector(wave_sum_unsigned);

END behavior;
```

VOLUME FILE

```
----------------------------------------------------------------
------------------
-- Company: ENGS 31
-- Engineer: Daniel Kim, Alex Martinez
--
-- Create Date: 05/31/2018 10:50:23 PM
-- Design Name:
-- Module Name: Volume - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------
------------------


library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY Volume is
PORT (    clk       :   in std_logic;
          wave_in   :   in std_logic_vector(15 downto 0);
          num_waves :   in std_logic_vector(3 downto 0);
        wave_out:   out std_logic_vector(15 downto 0)
          );
end Volume;

ARCHITECTURE behavior of Volume is

-- Signal
signal wave_in_unsigned  : unsigned(15 downto 0) := (others =>
'0');
signal wave_out_unsigned : unsigned(27 downto 0) := (others =>
'0');

type multiplier is array (7 downto 0) of unsigned (11 downto 0);
constant mult_array : multiplier := (to_unsigned(4095, 12),
                                    to_unsigned(2896, 12),
                                    to_unsigned(2365, 12),
                                    to_unsigned(2048, 12),
                                    to_unsigned(1832, 12),
                                    to_unsigned(1672, 12),
                                    to_unsigned(1548, 12),
                                    to_unsigned(1448, 12)
                                    );

BEGIN

volume: process(num_waves, wave_in)
begin
     if num_waves = "0000" then
        wave_out_unsigned <= (others => '0');
     else
        wave_out_unsigned <= unsigned(wave_in) *
mult_array(to_integer(unsigned(num_waves)-1));
    end if;
end process;

-- Outputs
wave_out <= std_logic_vector(wave_out_unsigned(27 downto 12));
```

```
END behavior;
```

DIGITAL TO ANALOG FILE

```
----------------------------------------------------------------
------------------
-- Company: ENGS 31
-- Engineer: Daniel Kim, Alex Martinez
--
-- Create Date: 05/30/2018 09:00:19 PM
-- Design Name:
-- Module Name: pmod_da2 - Behavioral
-- Project Name: Piano
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------
------------------


library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY DtoA is
PORT (    sclk      :    in   std_logic;
          take_sample    :    in   std_logic;
        data_in          :    in   std_logic_vector(15 downto
0);
        spi_sclk    :    out std_logic;
        spi_sync    :    out  std_logic;
        spi_DinA    :    out  std_logic);
end DtoA;

ARCHITECTURE behavior of DtoA is

--Constant buffer
--constant buff : std_logic_vector(3 downto 0) := "0000";

--Signals
signal count              : unsigned(3 downto 0) := "0000";
signal count_done         : std_logic := '0';
signal shift_en                : std_logic := '0';
signal load_en            : std_logic := '0';
signal data_register      : std_logic_vector(15 downto 0) :=
(others=>'0');

type statetype is (idle, shift, load);
signal current_state, next_state: statetype := idle;

BEGIN

--State Update
stateUpdate: process(sclk)
begin
     if rising_edge(sclk) then
     current_state<=next_state;
    end if;
end process stateUpdate;

-- Combinational Logic for next state
controllerFSM: process(current_state, count_done, take_sample)
begin
     shift_en <= '0';
    load_en <= '0';
    spi_sync <= '1';
    next_state <= current_state;
    case current_state is
     when idle =>
```

```
          if (take_sample = '1') then
               next_state <= shift;
            end if;
        when shift =>
          shift_en <= '1';
            spi_sync <= '0';
            if (count_done = '1') then
               next_state <= load;
            end if;
        when load =>
          load_en <= '1';
            next_state <= idle;
        when others =>
          next_state <= idle;
    end case;
end process controllerFSM;

-- Counter
counter: process(sclk)
begin
     if rising_edge(sclk) then
      count_done <= '0';
         if (load_en = '1') then
          count <= "0000";
            data_register <= data_in;
         elsif (shift_en = '1') then
          if count = "1110" then
                count_done <= '1';
            end if;
            count <= count + 1;
            data_register <= data_register(14 downto 0) &
data_register(15);
         end if;
     end if;
end process counter;

spi_DinA <= data_register(15);
spi_sclk <= sclk;
```

```
end behavior;
```

Testbench Code

FREQUENCY COUNTER TESTBENCH

```
----------------------------------------------------------------
----------------
-- Engineer:        Daniel Kim, Alex Martinez
-- Course:               Engs 31 16X
--
-- Create Date:     07/22/2016
-- Design Name:
-- Module Name:     pmod_ad1_tb.vhd
-- Project Name:    Lab5
-- Target Device:
-- Tool versions:
-- Description:     VHDL Test Bench for module: pmod_ad1
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:

----------------------------------------------------------------
----------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.all;
use IEEE.MATH_REAL.ALL;

ENTITY FrequencyCounter_tb IS
END FrequencyCounter_tb;

ARCHITECTURE behavior OF FrequencyCounter_tb IS
```

```
component FrequencyCounter
port (-- interface to top level
       clk               :    in std_logic;
       take_sample  :    in std_logic;
          key             :    in   std_logic_vector(7 downto 0);
          lut_input :    out  std_logic_vector(15 downto 0));

end component;

   --Inputs
    signal clk : std_logic := '0';
    signal take_sample : std_logic := '0';
    signal key : std_logic_vector(7 downto 0) := "00000000";
    signal lut_input : std_logic_vector(15 downto 0) :=
"0000000000000000" ;

     constant clk_period : time := 1 ns;

BEGIN
     -- Instantiate the Unit Under Test (UUT)

uut: FrequencyCounter port map(
        clk => clk,
        take_sample => take_sample,
        key => key,
        lut_input => lut_input
        );

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

 process
 begin
     wait for clk_period*5;

    take_sample <= '1';
    Key <= "00000001";
    wait for clk_period*15;

    take_sample <= '0';
    wait for clk_period*15;
    Key <= "00000000";
    wait for clk_period*10;

    take_sample <= '1';
    Key <= "00000010";
    wait for clk_period*15;
    take_sample <= '0';
    wait for clk_period*15;
    Key <= "00000000";
    wait for clk_period*10;

    take_sample <= '1';
    Key <= "00000100";
    wait for clk_period*15;
    take_sample <= '0';
    wait for clk_period*15;
    Key <= "00000000";
    wait;

end process;

END;
```

DIGITAL TO ANALOG TESTBENCH

```
----------------------------------------------------------------
----------------
-- Engineer:        Eric Hansen
-- Course:               Engs 31 16X
--
-- Create Date:     07/22/2016
-- Design Name:
-- Module Name:     pmod_ad1_tb.vhd
-- Project Name:    Lab5
-- Target Device:
-- Tool versions:
-- Description:     VHDL Test Bench for module: pmod_ad1
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:

----------------------------------------------------------------
----------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.all;
use IEEE.MATH_REAL.ALL;

ENTITY DtoA_tb IS
END DtoA_tb;

ARCHITECTURE behavior OF DtoA_tb IS

component DtoA
PORT (    sclk       :    in   std_logic;
          take_sample    :    in   std_logic;
        data_in          :    in   std_logic_vector(15 downto
0);
        spi_sclk    :    out std_logic;
        spi_sync    :    out  std_logic;
        spi_DinA    :    out  std_logic);
end component;

   --Inputs
```

```
    signal sclk : std_logic := '0';
    signal take_sample : std_logic := '0';
    signal data_in : std_logic_vector(15 downto 0) := (others =>
'0');

     --Outputs
    signal spi_sclk : std_logic := '0';
    signal spi_sync : std_logic := '1';
    signal spi_DinA : std_logic := '0';

    -- Clock period definitions
    constant sclk_period : time := 1 us;             -- 1 MHz
serial clock
    constant sampling_count_tc : integer := 25;    -- to achieve
a 40 kHz sampling rate, for testing

     -- Data definitions
     constant TxData : std_logic_vector(15 downto 0) :=
"0111000001101001";
     signal bit_count : integer := 15;

     -- Internal definitions
     signal sampling_count : integer := 0;

BEGIN
     -- Instantiate the Unit Under Test (UUT)

uut: DtoA port map(
        sclk => sclk,
        take_sample => take_sample,
        data_in => data_in,

        -- SPI bus interface to Pmod AD1
        spi_sclk => spi_sclk,
        spi_sync => spi_sync,
        spi_DinA => spi_DinA );

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

   -- Stimulus process:  testbench pretends to the top level
   stim_proc_1: process(sclk)
   begin
    if rising_edge(sclk) then
        if sampling_count < sampling_count_tc-1 then
            sampling_count <= sampling_count + 1;
            take_sample <= '0';
        else
            sampling_count <= 0;
            take_sample <= '1';      -- push take_sample to
interface to initiate a conversion
            data_in <= TxData;
        end if;
    end if;
   end process stim_proc_1;

   -- Stimulus process:  testbench pretends to be the D/A
converter
   stim_proc_2: process(spi_sclk)
   begin
    if falling_edge(spi_sclk) then
        if spi_sync = '0' then
               if bit_count = 0 then bit_count <= 15;
               else bit_count <= bit_count - 1;
               end if;
          end if;
    end if;
   end process stim_proc_2;
END;
```

Resource Utilization

Site Type	Used	Available	%Utilized
Slice LUTs	293	20800	1.41
LUT as Logic	293	20800	1.41
LUT as Memory	0	9600	0.00
Slice Registers	346	41600	0.83
Registers as Flip Flop	346	41600	0.83
Registers as Latch	0	41600	0.00
DSP48E1	1	90	1.11
Bonded IOB	13	106	12.26
BUFGCTRL	2	32	6.25
Block RAMB36/FIFO	0	50	0.00
Total Gate Count for Design	918		
Black Box: dds_compiler_0	8		

Critical timing path

Source: sclkdiv_reg[11]/C (serial_clock_divider in piano_shell.vhd)
Destination: sclkdiv_reg[45]/R (serial_clock_divider in piano_shell.vhd)
Maximum Clock Speed: 206.7 MHz
Minimum Clock Period: 4.838 ns

Analysis of Residual Warnings

Warning:

[Synth 8-3331] design Volume has unconnected port clk

Analysis:

This warning comes from our Volume file having an input clk in the port, which is not being used. There are no synchronous processes in this file so the warning was not critical as it did not interfere with the functionality of the component.

Warning:

[Vivado_Tcl 4-252] No drc_checks matched for command 'get_drc_checks LUTP-1' ["O:/engs31/Piano_Poly/Piano_Poly.srcs/constrs_1/new/piano.xdc":304]

Analysis:

This warning comes from a line in our piano.xdc file "set_property SEVERITY {Warning} [get_drc_checks LUTP-1]". This line is meant to say "LUTLP-1" instead of "LUTP-1". This is what resulted in our lack of Design Rule Checks for LUTP-1. Setting this line for LUTLP-1 was our attempt to disable critical warnings for our combinatorial loop in the piano_shell.vhd file. The loop was intended to iterate through each key and add their generated sine waves for the polyphonic aspect in our project. This warning ended up being benign because we just harded coded in that the maximum waves we would add up would be 3, because we would get a synthesis error for the loop.

Warning:

[Synth 8-6014] Unused sequential element num_waves_reg was removed. ["O:/engs31/Piano_Poly/Piano_Poly.srcs/sources_1/new/piano_shell.vhd":225]

Analysis:

This warning relates to the previous one. The variable for our accumulation of every keys' waves is num_waves. The program added a "_reg" to the end of the variable because of the synchronous process of the loop. Again, because we never used this process so we received this warning message. However, the warning was not critical because it did not interfere with any of our processes as we never used it.

Warning:

[Synth 8-3332] Sequential element (freq_counterLC/count_int_reg[1]) is unused and will be removed from module Piano.

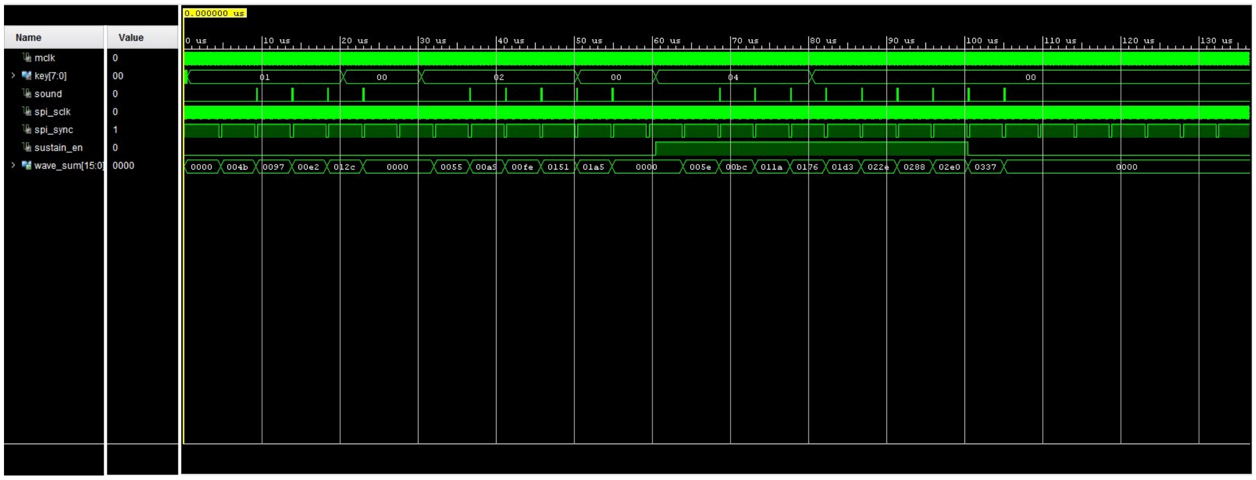
Analysis:

count_int is an integer variable we used to add the constant counter incrementer to when a key was pressed. We encountered this warning and others alike it referencing count_int_reg[0], count_int_reg[2], count_int_reg[3], count_int_reg[4]. This warning comes from our count_int being passed to our LUT input that takes a 16 bit number and indexes to our lookup table. If the count_int has bit values that are not significant when turned into an unsigned, they are unused and thus removed from the module Piano, which holds all the instantiations of our frequency counters that contained this warning.

7.3 Memory Map

We utilized a Xilinx® LogiCORE™ IP Direct Digital Synthesizer (DDS) Compiler to store values of our sine waves, acting as our ROM lookup table (LUT). Once our constant counter increment values, m, were selected for the corresponding key by our key multiplexer in our FrequencyCounter.vhd file, we outputted a LUT_input variable which was passed this value. Each clock cycle, the LUT_input variable was indexed into the LUT to find the corresponding point on a sine wave and then incremented by m so that on the next clock cycle it would find the next point on the sine wave, eventually extending to all the values on the sine wave, 4096 (2^{12} possible points on the sine wave correlating to the fineness of the graph). The output of the LUT was passed to our wire_to_adderX variable (where X is the key being selected) which was then fed to our adder to account for the possibility that multiple keys were pressed. Because there are 8 keys there are 8 instantiations of this LUT, meaning we have 8 blocks of memory with the same values in each one.

7.4 Waveform graphs



Legend (signals top to bottom):

mclk → FPGA board master clock

key[7 :0] → std_logic_vector of keys being pressed

sound → output of DtoA converter (feeds into amplifier)

spi_sclk → spi bus clock in DtoA file
spi_sync → spi bus synchronization signal
sustain_en → sustain enable signal
wave_sum[15:0] → output of Adder file (sum of waves)