Engineering Sciences 31/Computer Science 56

Final Project

Liane Makatura and Stylianos Tegas

August 31, 2015

Abstract

The advent of digital electronics has ushered in an era of rapid innovation and discovery. These electronics have infiltrated nearly every aspect of our daily lives, holding significant influence over activities from education to personal communication. We decided to explore the influence of digital electronics in the entertainment industry, pursuing the creation of a polyphonic keyboard. The notes of this keyboard span one full octave, from C4 (middle C) to C5, including all sharps and flats. It also implements a sustain pedal, which allows the user to suspend notes even after the corresponding key has been released.

Contents

1 Introduction

While digital electronics have a wide range of usefulness in various applications, generating sound is rather difficult without a specific hardware setup in place. Further complicating things, generating more than one note at a time (polyphony) is practically impossible without multiple FPGA systems. Thus, the problem to solve is twofold: how can one generate sound, and how can one generate superpositions of sounds?

2 Design Solution

2.1 Specifications

Inputs:

1. Note Buttons ($C4 - C5$)
2. Sustain Switch

The circuit receives a series of button presses based on the above possible inputs, as well as a possible switch signal. Based on the buttons pressed, the circuit creates addresses for a wave generator lookup table that increment in certain step sizes. These addresses are used to find the corresponding value of the wave for each note. These values are added together on the other side of the lookup table and then normalized. This final value is then sent through a digital to analog converter before being played on a speaker. If the sustain switch is closed, then even after a button is released, the circuit continues to play that note.

2.2 Operating Instructions

Assembly:

Materials:

1. FPGA (Nexys 3) with its microUSB cable
2. Computer (must have Xilinx and Adept software)
3. Required VHDL files
4. Pmod DA2 - Digital to Analog Converter
5. Pmod AMP2 - Sound Amplifier
6. Pmod BTN - Four Push Buttons (x4)
7. Pmod Cable Kit, 6 pin (x5)

Assembly of the keyboard is fairly straightforward. First, gather the above materials. Assemble the cable kits and place one in each of the top halves of the 12 pin connectors on the FPGA (JA, JB, JC, JD). Take the final cable kit and place it in the bottom half of the JD 12 pin connector. Attach the four Pmod BTNs to the cable kits placed in the top halves of the 12 pin connectors. Attach the Pmod DA2 to the cable kit in the bottom half of the JD 12 pin connector, then attach the Pmod AMP2 to the output of the Pmod DA2. The Pmod AMP2 is modified such that its 4th pin (the shutdown pin) is tied to V_{cc}, so that it is always ready to receive data.

Connect the FPGA board to the computer. Assemble the required files in the ISE Design Kit software, then generate a programming file. Finally, open the Adept software, and program the FPGA. Congratulations, you have a functional polyphonic keyboard!

Usage

To use our keyboard, just push any combinations of the buttons on the Pmod BTNs! The top left button on the Pmod connected to the JC 12 pin connector does not connect to an input into the circuit, a feature shared by the top buttons on the Pmod connected to the JA 12 pin connector. To sustain notes, simply flip the switch located at T10 on the FPGA.

2.3 Theory of Operation

Each section in the block diagram of the circuit is explained in more detail below. The operation of the circuit is as follows:

Slow Clock

This is a clock divider that runs the majority of the circuit. It is a clock that runs six times slower than the master clock (100MHz). This slow clock is required for the circuit because of the inherent latency in the wave generator lookup table. Since it takes 6 clock cycles for the lookup table to generate an ouput after it has received an input, the rest of the circuit must run 6 times slower than the lookup table to compensate and not generate unwanted outputs.

The slow clock takes in the master clock as input. It counts to three, and when it reaches three, it flips the value of its internal signal. The internal signal is outputted constantly. Thus, the block acts as a clock running at (100 / 6) = 16.67MHz.

Sampler Counter

This is a counter that creates the note sampling rate of 44.1kHz, used almost ubiquitously in audio digital recording. This counter is required for the circuit because it reduces the number of samples taken, which in turn allows us to create sounds that humans will be able to hear. This is because of the way that direct digital synthesis evaluates and creates tones (i.e. the output frequency is directly proportional to the sampling frequency).

The sampler counter takes the slow clock as input. It counts to 378 before outputting an enable signal. This enable signal is eventually fed into both the note address lookup/update array and the select counter.

Note Address

This is a set of two arrays, used to index the correct sine value to generate its corresponding note. This set of arrays is required for the circuit in order to generate the correct tones from the wave generator lookup table.

The arrays receive input from the select counter. When the enable is asserted, each of the addresses in the first array is updated by its corresponding step value in the second array. These updated addresses are fed constantly into the multiplexer.

Select Counter

This is a counter that chooses which of the thirteen note addresses to send through the multiplexer to the wave generator lookup table. This is required for the circuit in order to actually choose which note to send through. Since the counter will cycle through all thirteen notes, it will allow for any and all of the notes to be played at once.

The counter receives an enable input from the sampler counter. When it does, it begins counting from zero to twelve, allowing the multiplexer to send in each updated note address to the lookup table. The counter then resets and waits for the next enable from the sample counter.

Note Multiplexer

This multiplexer chooses which note to send into the wave generator lookup table. This is required for the circuit in order to be able to generate multiple tones at once.

The multiplexer receives inputs to be multiplexed from the note address arrays. It also receives input from the select counter in the form of select bits. These bits tell the multiplexer which note address to allow past into the wave generator lookup table.

Wave Generator Lookup Table

This block generates a sine value based on the input it receives. This is required for the circuit in order to actually generate a sine wave which will be translated into sound.

This block was constructed by the core generator software in Xilinx, so we are not entirely sure of the mechanisms it uses. In general, though, the generator receives an address. It then indexes into a lookup table, finds the appropriate sine value, and outputs it.

Button Pressed Multiplexer

This multiplexer chooses whether to add the sine value to the accumulator based on whether the button for that note is being pressed. This is required for the circuit in order for the circuit to not always play all thirteen notes at once.

The multiplexer receives a single select bit from the keyboard input. If the button is pressed, it lets the sine value go through. Otherwise, it sends zero to the accumulator.

Accumulator

This block adds the sine values from the wave generator together. This is required for the circuit in order to play multiple notes at once. This idea is based on the theory of superposition (i.e. waves add directly to create superpositions of waves that act differently from either wave).

The accumulator receives input from the button pressed multiplexer. It adds this value to its stored value and saves it in a register when it receives a load input. When the terminal count is reached (all thirteen notes have cycled through), the accumulator is cleared to make room for the updated values.

Normalizer

This block normalizes the output of the accumulator. This is required for the circuit in order to play notes at a consistent volume no matter how many are pressed.

This block receives input from the accumulator and a load signal. Based on the number of buttons pressed (given by another input), the normalizer multiplies the accumulator input by a certain number, then outputs the first 12 bits of that number (equivalent to shifting the bits, or dividing by a power of two). This output is saved to a register and sent to the digital to analog converter.

Digital to Analog Converter

This block converts the digital signal to an analog signal that is fed into the output speaker to create the tones. This is required for the circuit in order to actually produce melodious tones.

This block was created by David Picard, so we are not entirely sure how the block functions. In general, however, it converts the full digital signal into a series of serial bits that are sent to the amplifier to produce the tones.

2.4 Construction and Debugging

We initially constructed note counters for each individual note. They were fed by a slow clock buffer than ran at 44.1kHz, and the note counters stepped by the necessary amount to generate the proper frequencies. We then created the multiplexer that would choose which note to feed into the wave generator lookup table and the select counter that would decide which note to multiplex in. Following this, we used the core generator to create the wave generator lookup table. We lastly implemented the ADDA converter that David Picard supplied to us and constructed a controller for the state machine. This design ultimately failed. We attempted to debug the system, but the system failed to output even a single note.

After some careful thought and discussion with Mitchell Goff, we changed our approach to the problem. We did away with the individual counters and instead used arrays to hold the addresses to the wave generator lookup table. Using a sampler counter, we updated the addresses of each note by the step specified in another array. We also moved the multiplexer system to the top module to simplify the design. This design approach worked, but we had an issue where the FPGA would only sound the high c. After some careful debugging and simulating, we realized that the latency in the wave generator lookup table was causing the circuit to exhibit undefined behavior. To remedy this, we created a slow clock to run all of the parts of the circuit except for the lookup table. By doing this, we effectively removed the latency from the table. We were then finally able to sound the correct notes, although we were not yet achieving polyphony.

The final step was adding in the accumulator and normalizer. The accumulator was constructed and worked without a hitch. However, when the normalizer was implemented, it would only work when the high c note was played. This problem was solved once we realized that we needed an extra state in our controller to allow the normalizer time to properly work. After adding an extra state to the controller, the circuit worked perfectly.

After achieving polyphony, we discussed extra features regarding the keyboard. Adding the suspension was a painless process with no debugging necessary. However, when we attempted to implement chord creation, there was simply not enough time to make it work properly, and so was scrapped.

3 Justification and Evaluation

Our initial solution to the polyphonic keyboard was ultimately scrapped in favor of a more sophisticated and simplistic design. While our initial solution may have worked had we figured out the bug in our code, the solution we ultimately adopted was far easier to understand, far easier to implement, and had the potential for far fewer bugs.

Our final design was straightforward and clear, even to those who did not work on our project. Its data flow is relatively simple given the complexity of the problem, and the combinational logic that it implements is abstracted in a way that generalizes it and allows the components to be used in other (possibly unrelated) projects. Other designs that we initially looked at (including our initial design) are clunky in terms of implementation, and rely far too much on data abstraction. While abstracting logic is good sometimes, it can also be deadly to a program/project if not correctly done. In the case of our final design, we took advantage of the fact that VHDL has the ability to synthesize arrays in hardware. This could be a flaw in our final design, as arrays may not always be present in hardware description languages, and so our project would not be portable. Ultimately, however, we believe that our design is the best possible for the problem we set out to solve.

If we were to have the opportunity to recreate the project, we would have gone over in more detail the goals of the project. Initially, while we understood what general ideas we needed to implement for this project to work, we did not discuss the various options for implementing these ideas. Had we done this, we believe that we would have realized the suitability of arrays for our design. In addition, we would have tried to abstract out our multiplexer. Although we had initially had it in a separate VHDL file, we chose to put it in the top module because we believed it to be a source of our bugs, and we chose not to move it back to its own file once we figured out where the actual bugs were originating. In this sense, the multiplexer would have also been abstracted and easily portable to other projects.

4 Conclusions

The goal of this project was to create a fully functional polyphonic keyboard. In other words, we set out to create an electronic that would be able to output multiple tones at once.

Originally, our proposal entailed multiple counters that would count independently of one another and update the notes. However, after our initial meeting with Professor Eric Hansen and David Picard, we realized that this design would not work with direct digital synthesis. After a fair number of iterations of our design, our final design was able to accomplish exactly what we specified in our proposal. In addition, we were able to add an extra feature that was not initially specified in our proposal.

For groups in the future who consider creating a polyphonic keyboard, our suggestion regarding its construction is to seriously think about the design before beginning to implement it. Ultimately, we scrapped about five of six hours worth of coding because we did not take an hour to sit down with the design and question its implementation. In addition, we would also caution future groups against trying to optimize sound quality at first. Although it would not be too difficult to optimize sound quality at the end of the project, we spent too much time attempting to get the best quality sound for our keyboard when we should have instead focused on simply getting the keyboard to sound at all.

All in all, our experience creating this keyboard was an overwhelmingly positive one. The students, TAs and professors who helped us on our design reaffirmed our faith in collaborative thinking. In addition, we learned much not only about digital electronics, but about our work habits as a team. We are very proud of our work and grateful to all those who made it possible.

5 Acknowledgements

We are grateful to Professor Eric Hansen, David Picard, and our TAs Hunter Black, Matthew West, Ellen Davenport, and Mac Keyser for their assistance during the term and for this project in particular. We would also like to thank our peers for their assistance in various parts of the conceptualization and debugging or our project. In particular, we would like to thank Dana Wieland and Mitchell Goff.

6 References

The note frequencies we generated were found on the Physics of Music - Notes page of Michigan Tech Department of Physics website.
URL: http://www.phy.mtu.edu/ suits/notefreqs.html

7 Appendices

7.1 System Level Diagrams

1. **Front Panel**

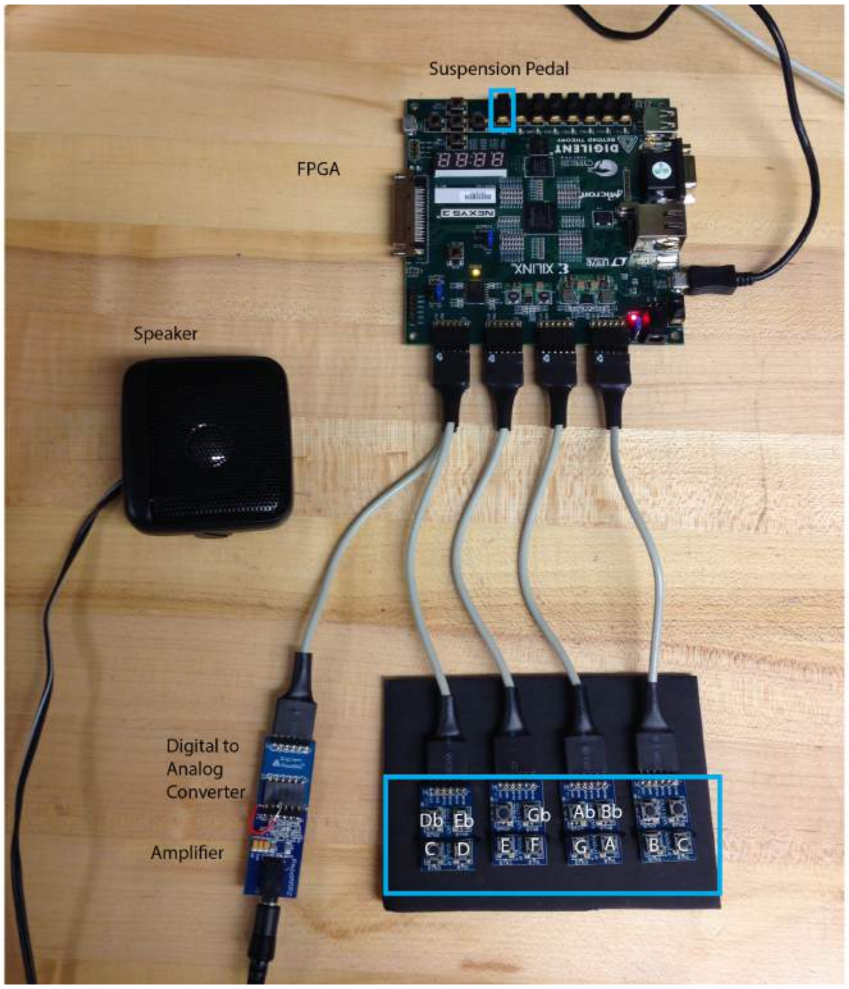

Figure 1: This figure shows all relevant functional components of our polyphonic keyboard.

2. **Functional Block Diagram**

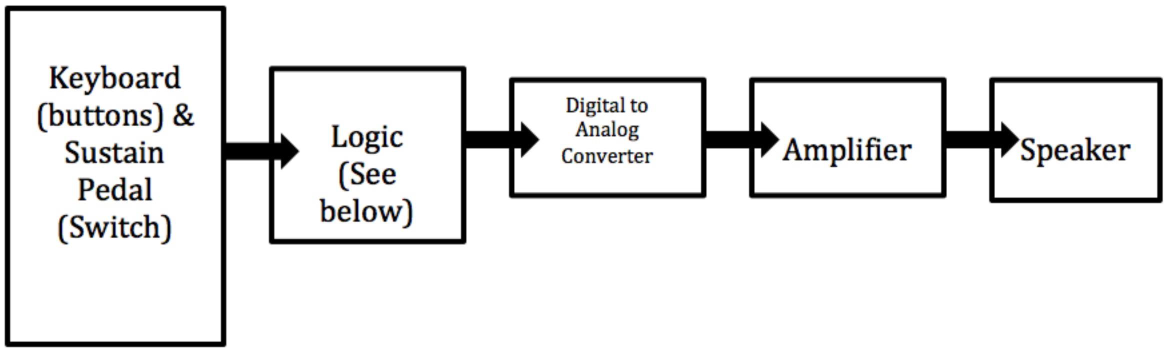
Keyboard (buttons) & Sustain Pedal (Switch)
Logic (See below)
Digital to Analog Converter
Amplifier
Speaker

Figure 2: This figure shows the overall functional hardware block diagram. All elements were provided pmods, with the exception of the "Logic" box that is explored in the next figure.

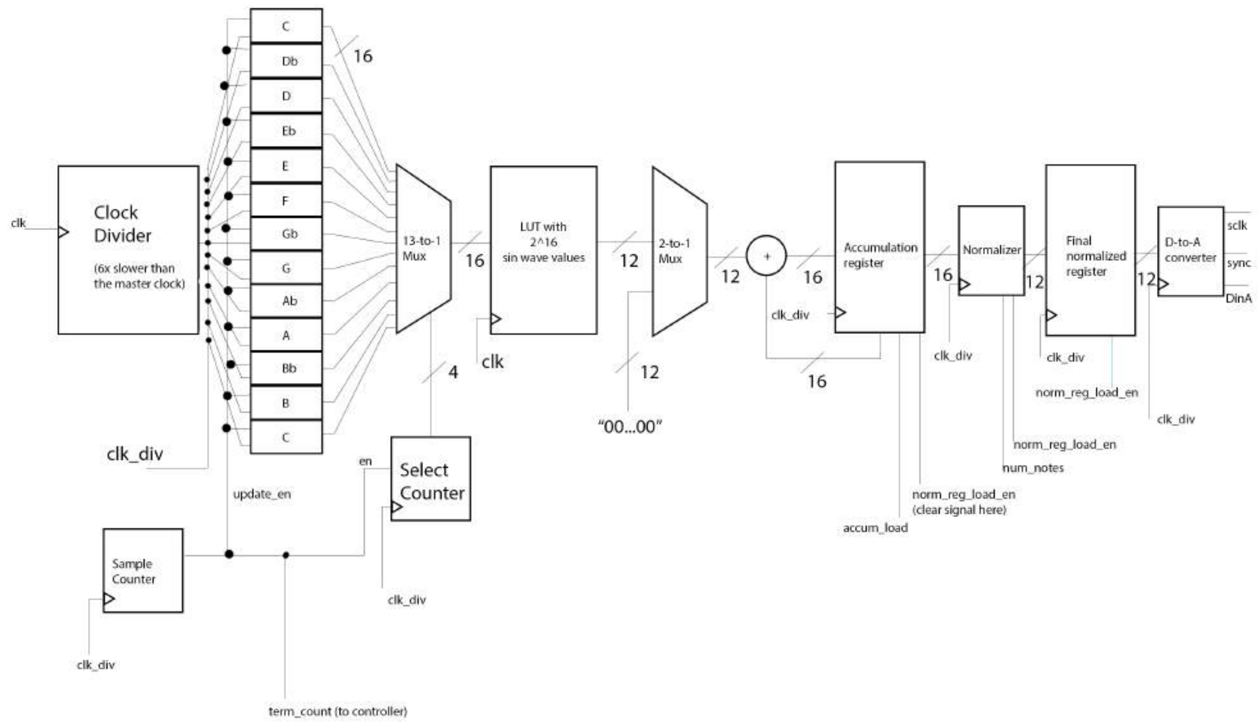

Figure 3: This figure shows the block diagram for the data path of our polyphonic keyboard.

3. **Schematic Diagram** Since we did not wire any off board components, there are no schematic diagrams

to be shown.

4. **Package Map**

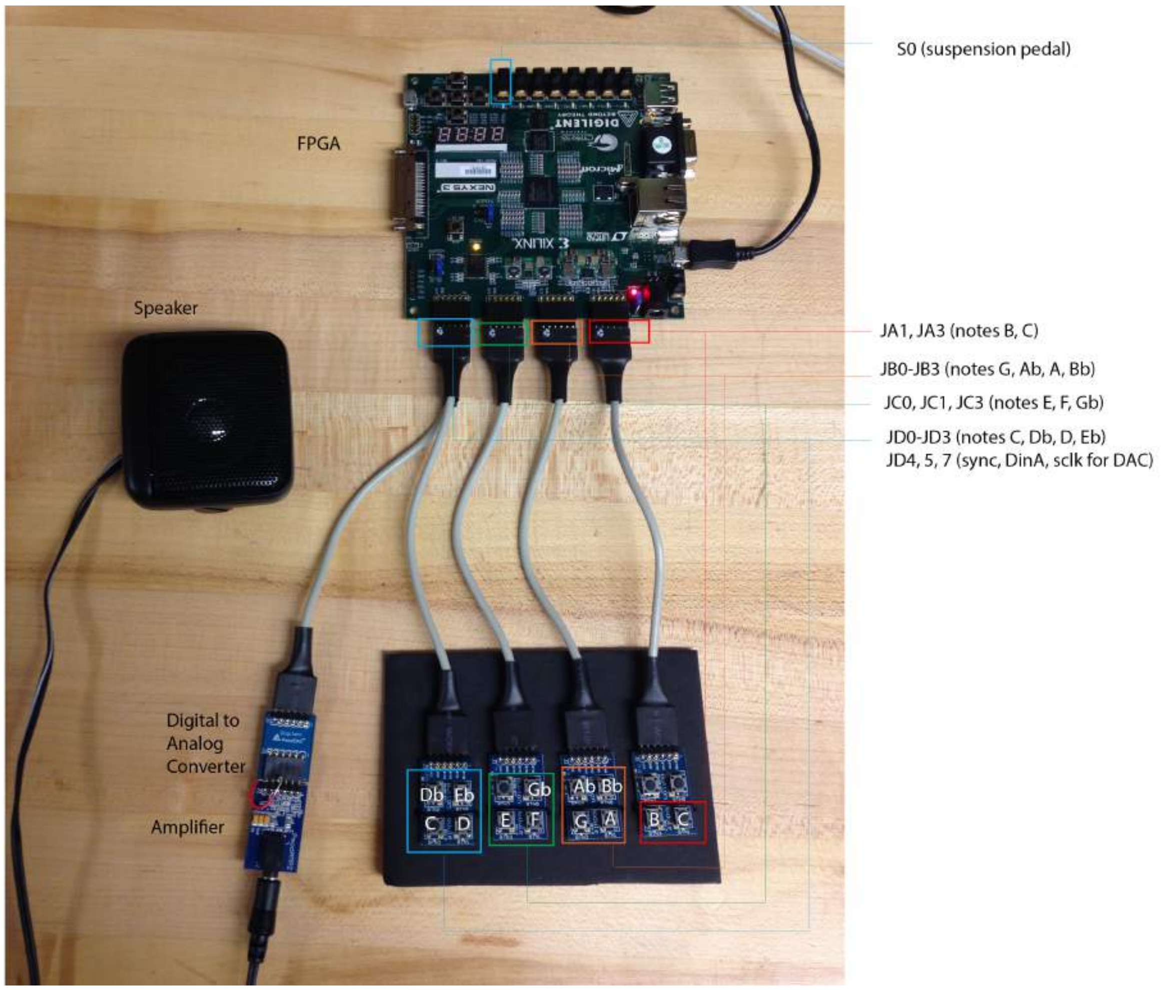
S0 (suspension pedal)
FPGA
Speaker
JA1, JA3 (notes B, C)
JB0-JB3 (notes G, Ab, A, Bb)
JC0, JC1, JC3 (notes E, F, Gb)
JD0-JD3 (notes C, Db, D, Eb)
JD4, 5, 7 (sync, DinA, sclk for DAC)
Digital to Analog Converter
Amplifier
Db Eb
C D
Gb
E F
Ab Bb
G A
B C

Figure 4: This figure shows the port mappings for our polyphonic keyboard.

5. **Parts List**

Reference	Quantity	Part Number	Description
Nexys3	1	Nexys3	Digilent Nexys3 board
ADC1	1	410-113	Digilent Pmod-DA2 D/A converter
AMP	1	410-233P-KIT	Digilent Pmod-AMP2 audio amplifier
BTN1-BTN4	4	410-077P	Digilent Pmod-BTN 4-button module
SPKR	1		Speaker

Table 1: Parts needed to create our keyboard.

7.2 Programmed Logic

1. **State Diagrams**

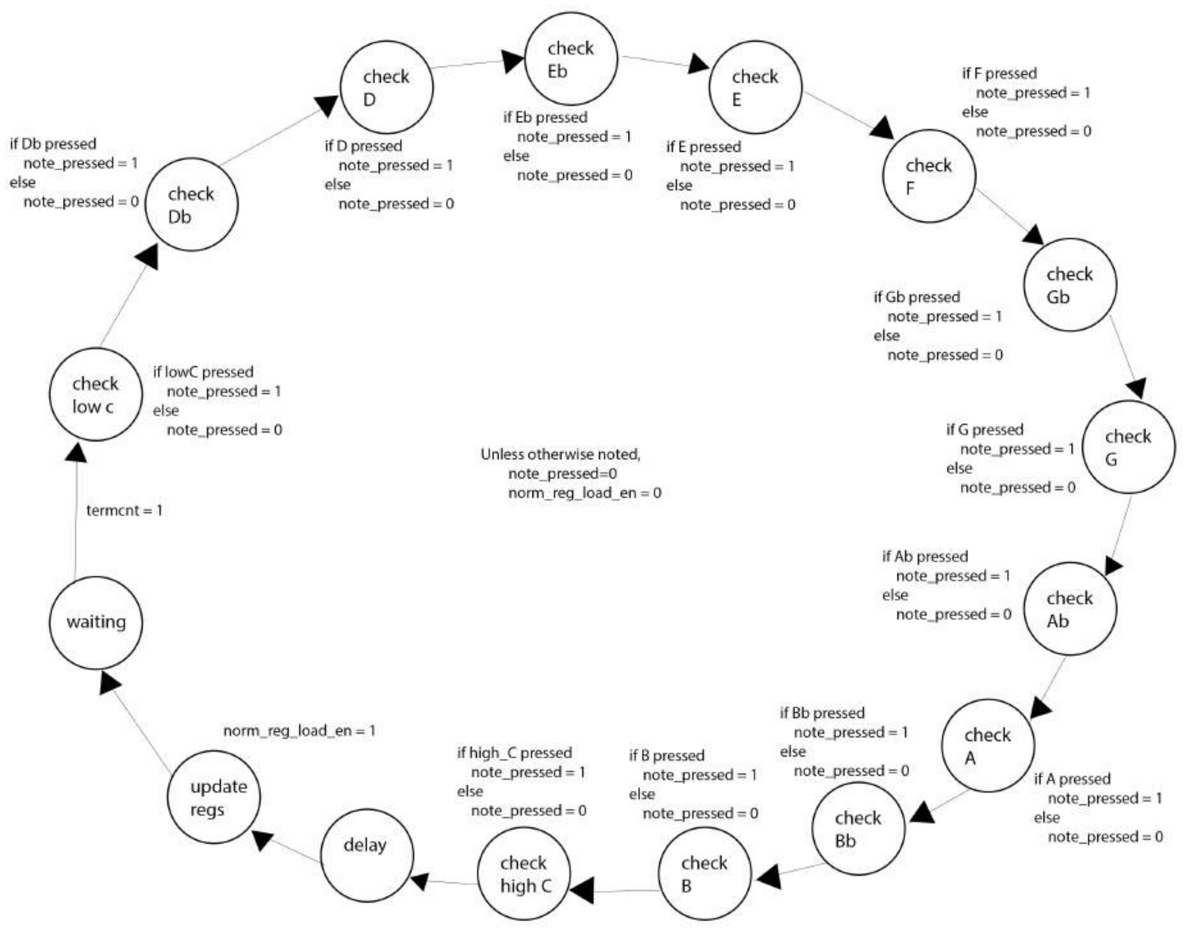
check D
check Eb
check E
if F pressed note_pressed = 1 else note_pressed = 0
if Db pressed note_pressed = 1 else note_pressed = 0
check Db
if D pressed note_pressed = 1 else note_pressed = 0
if Eb pressed note_pressed = 1 else note_pressed = 0
if E pressed note_pressed = 1 else note_pressed = 0
check F
check Gb
if Gb pressed note_pressed = 1 else note_pressed = 0
check low c
if lowC pressed note_pressed = 1 else note_pressed = 0
if G pressed note_pressed = 1 else note_pressed = 0
check G
Unless otherwise noted, note_pressed=0 norm_reg_load_en = 0
termcnt = 1
if Ab pressed note_pressed = 1 else note_pressed = 0
check Ab
waiting
if Bb pressed note_pressed = 1 else note_pressed = 0
check A
norm_reg_load_en = 1
if high_C pressed note_pressed = 1 else note_pressed = 0
if B pressed note_pressed = 1 else note_pressed = 0
if A pressed note_pressed = 1 else note_pressed = 0
update regs
check Bb
delay
check high C
check B

Figure 5: This figure shows the state machine used by our Keyboard Controller module to control the datapath for the polyphonic keyboard.

2. **VHDL Code**
   Keyboard Top Module:

```
                    --
    --------------------------------------------------------------------------------
-- Company: ENGS 31 15X
-- Engineer: Liane Makatura and Stylianos Tegas
--
-- Create Date:    20:33:50 08/19/2015
-- Design Name:
-- Module Name:    KeyboardTop - 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;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity KeyboardTop is
    Port ( clk : in  STD_LOGIC;
        noteb : in STD_LOGIC_VECTOR (12 downto 0);
        sus : in STD_LOGIC;
           sync : out  STD_LOGIC;
           sclk : out  STD_LOGIC;
           DinA : out  STD_LOGIC);
end KeyboardTop;

architecture Behavioral of KeyboardTop is
type wave_counter_dt is array (12 downto 0) of unsigned (15 downto 0);

--constants
constant note_steps : wave_counter_dt := (to_unsigned(389, 16), -- c
                            to_unsigned(412, 16), -- dflat
                            to_unsigned(436, 16), -- d
                            to_unsigned(462, 16), -- eflat
                            to_unsigned(490, 16), -- e
                            to_unsigned(519, 16), -- f
                            to_unsigned(550, 16), -- gflat
                            to_unsigned(583, 16), -- g
                            to_unsigned(617, 16), -- aflat
                            to_unsigned(654, 16), -- a
                            to_unsigned(693, 16), -- bflat
                            to_unsigned(734, 16), -- b
                            to_unsigned(778, 16)  -- c
                            );

--signals

```

```
  signal inotes: std_logic_vector(12 downto 0) := noteb;
  signal clk_div: std_logic := '0'; --clock divider
  signal select_en: std_logic := '0'; --enables checking of buttons
  signal note_pressed: std_logic := '0';      --checks whether button is pressed

  signal selectbits: std_logic_vector (3 downto 0) := (others => '0');  --holds select bits
    for the multiplexer
  signal accum_load: std_logic := '0';  --allows accumulator to load
  signal load: std_logic := '0';  --allows normalizer to load

  --addresses for each of the notes, sent to the mux
  signal note_addresses : wave_counter_dt := (others => (others => '0')); -- initialize all
    addresses to 0

  signal noteadd: std_logic_vector (15 downto 0) := (others => '0');    --address for the
    correct note to send to lut
  signal note_to_add: std_logic_vector (11 downto 0) := (others => '0');  --value of the
    sine wave to add to accumulator
  signal sum_note: std_logic_vector (15 downto 0) := (others => '0');   --the summed values
    of the sine wave for polyphony
  signal num_notes: std_logic_vector (3 downto 0) := (others => '0');   -- how many notes
    are pressed, for normalization
  signal final_note: std_logic_vector (11 downto 0) := (others => '0'); --the normalized
    note to be sent to the adda converter

--components
  -- clock divider for the datapath
  -- brings it down to a clock 6 times slower than the master
  COMPONENT ClockDivider
  PORT(
    clk : IN std_logic;
    clk_div : OUT std_logic
    );
  END COMPONENT;

  -- sampler counter for address updating and cycling
  COMPONENT SamplerCounter
  PORT(
    clk : IN std_logic;
    term_cnt : OUT std_logic
    );
  END COMPONENT;

  -- select counter for the mux, to loop through all notes
  COMPONENT SelectCounter
  PORT(
    clk : IN std_logic;
    en : IN std_logic;
    selectbits : OUT std_logic_vector(3 downto 0)
    );
  END COMPONENT;

  -- digital to analog converter
  COMPONENT adda
  PORT(
    clk : IN std_logic;
    tone : IN std_logic_vector(11 downto 0);
    sync : OUT std_logic;
    DinA : OUT std_logic;
    sclk : OUT std_logic
    );
  END COMPONENT;

  -- LUT for sin values
  COMPONENT WaveGen
  PORT (
    clk : IN STD_LOGIC;
    phase_in : IN STD_LOGIC_VECTOR(15 downto 0);
    sine : OUT STD_LOGIC_VECTOR(11 DOWNTO 0)
  );
  END COMPONENT;

  --controller for data path
  COMPONENT KeyboardController
  PORT(
    clk : IN std_logic;
    noteb : IN std_logic_vector (12 downto 0);
    termcnt : IN std_logic;
    note_pressed : OUT std_logic;
    norm_reg_load_en : OUT std_logic
    );
  END COMPONENT;

  -- accumulates the sin values for polyphony
  COMPONENT Accumulator
  PORT(
    clk : IN std_logic;
    note_to_add : IN std_logic_vector(11 downto 0);
    load_en : IN std_logic;
    clear_en : IN std_logic;
    note_to_norm : OUT std_logic_vector(15 downto 0)
    );
  END COMPONENT;

  -- normalizes the accumulated value by the number of notes played to get the final output
  -- same volume no matter how many notes are being played
  COMPONENT Normalizer
  PORT(
    clk : IN std_logic;
    load_en : IN std_logic;
    note_to_norm : IN std_logic_vector(15 downto 0);
    num_note : IN std_logic_vector(3 downto 0);
    final_note : OUT std_logic_vector(11 downto 0)
    );
  END COMPONENT;


begin

--processes

--creates suspension
suspension: process(clk_div)
begin

  if rising_edge(clk_div) then
    inotes <= inotes;
    if sus = '1' then
      for n in 0 to 12 loop
        if noteb(n) = '1' then
          inotes(n) <= '1';
        else
          inotes(n) <= inotes(n);
        end if;
      end loop;
    else
      inotes <= noteb;
    end if;
  end if;
end process;

-- updates all the note addresses
address_update: process(clk_div)
begin

  if rising_edge(clk_div) then
    if select_en = '1' then
      for n in 0 to 12 loop
        note_addresses(n) <= note_addresses(n) + note_steps(n);
      end loop;
    end if;
  end if;

end process;

-- determines which address to retreive from the sine LUT
address_mux: process(selectbits, note_addresses)
begin

  case selectbits is
    when "0000" =>
      noteadd <= std_logic_vector(note_addresses(12));
    when "0001" =>
      noteadd <= std_logic_vector(note_addresses(11));
    when "0010" =>
      noteadd <= std_logic_vector(note_addresses(10));
    when "0011" =>
      noteadd <= std_logic_vector(note_addresses(9));
    when "0100" =>
      noteadd <= std_logic_vector(note_addresses(8));
    when "0101" =>
      noteadd <= std_logic_vector(note_addresses(7));
    when "0110" =>
      noteadd <= std_logic_vector(note_addresses(6));
    when "0111" =>
      noteadd <= std_logic_vector(note_addresses(5));
    when "1000" =>
      noteadd <= std_logic_vector(note_addresses(4));
    when "1001" =>
      noteadd <= std_logic_vector(note_addresses(3));
    when "1010" =>
      noteadd <= std_logic_vector(note_addresses(2));
    when "1011" =>
      noteadd <= std_logic_vector(note_addresses(1));
    when "1100" =>
      noteadd <= std_logic_vector(note_addresses(0));
    when others =>
      noteadd <= "0000000000000000";
  end case;

end process;

--checks if the note address should go through to accumulator and adds to number of note
    pressed for normalizer
tonemux: process(note_pressed)
begin

  if note_pressed = '1' then
    accum_load <= '1';
  else
    accum_load <= '0';
  end if;

end process;
```

```

-- keeps track of how many notes are being played
num_note_reg: process(clk_div)
begin

  if rising_edge(clk_div) then
    num_notes <= num_notes;
    if note_pressed = '1' then
      num_notes <= std_logic_vector(unsigned(num_notes) + 1);
    elsif select_en <= '1' then
      num_notes <= "0000";
    end if;
  end if;

end process;

--portmaps

  -- clock divider for the wave counters
  -- brings it down to a 1024Hz clock
  Inst_ClockDivider: ClockDivider PORT MAP(
    clk => clk,
    clk_div => clk_div
  );

  -- Counter for sampling rate, outputs tc when it's time to sample
  Inst_SamplerCounter: SamplerCounter PORT MAP(
    clk => clk_div,
    term_cnt => select_en
  );

  -- select counter for the mux, to loop through all notes
  Inst_SelectCounter: SelectCounter PORT MAP(
    clk => clk_div,
    en => select_en,
    selectbits => selectbits
  );

  -- LUT for sin values
  Sine_LUT : WaveGen
  PORT MAP (
    clk => clk,
    phase_in => noteadd,
    sine => note_to_add
  );

  -- digital to analog converter
  Inst_adda: adda PORT MAP(
    clk => clk,
    tone => final_note,
    sync => sync,
    DinA => DinA,
    sclk => sclk
  );

  --Controller for datapath
  Inst_KeyboardController: KeyboardController PORT MAP(
    clk => clk_div,
    noteb => inotes,
    termcnt => select_en,
    note_pressed => note_pressed,
    norm_reg_load_en => load
  );

  -- accumulator for polyphony
  Inst_Accumulator: Accumulator PORT MAP(
    clk => clk_div,
    note_to_add => note_to_add,
    load_en => accum_load,
    clear_en => load,
    note_to_norm => sum_note
  );

  -- normalizes the accumulated value by the number of notes being played
  Inst_Normalizer: Normalizer PORT MAP(
    clk => clk_div,
    load_en => load,
    note_to_norm => sum_note,
    num_note => num_notes,
    final_note => final_note
  );


end Behavioral;



```

Keyboard Controller:

```
            --
--------------------------------------------------------------------------------
        -- Company: ENGS 31 15X
        -- Engineer: Stylianos Tegas and Liane Makatura
        --
        -- Create Date:    22:00:52 08/19/2015
        -- Design Name:
        -- Module Name:    KeyboardController - 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;

        -- Uncomment the following library declaration if using
        -- arithmetic functions with Signed or Unsigned values
        use IEEE.NUMERIC_STD.ALL;

        -- Uncomment the following library declaration if instantiating
        -- any Xilinx primitives in this code.
        --library UNISIM;
        ----use UNISIM.VComponents.all;

        entity KeyboardController is
            Port ( clk : in  STD_LOGIC;
                   noteb : in STD_LOGIC_VECTOR (12 downto 0);
                termcnt : in STD_LOGIC;
                   note_pressed : out  STD_LOGIC;
                   norm_reg_load_en : out  STD_LOGIC);
        end KeyboardController;

        architecture Behavioral of KeyboardController is
        type state_type is (check_lowc, check_dflat, check_d, check_eflat, check_e,
                    check_f, check_gflat, check_g, check_aflat, check_a,
                    check_bflat, check_b, check_highc, update_regs, waiting, delay);
        signal state, next_state : state_type := check_lowc;

        begin

        control: process(state, noteb, termcnt)
        begin

        note_pressed <= '0';  --defaults
        norm_reg_load_en <= '0';
        next_state <= state;

          case state is
            when check_lowc =>
              if noteb(12) = '1' then         -- if low C is pressed, let the mux know
                note_pressed <= '1';
              end if;
              next_state <= check_dflat;
            when check_dflat =>
              if noteb(11) = '1' then         -- if Db is pressed, let the mux know
                note_pressed <= '1';
              end if;
              next_state <= check_d;
            when check_d =>
              if noteb(10) = '1' then         -- if D is pressed, let the mux know
                note_pressed <= '1';
              end if;
              next_state <= check_eflat;
            when check_eflat =>
              if noteb(9) = '1' then          -- if Eb is pressed, let the mux know
                note_pressed <= '1';
              end if;
              next_state <= check_e;
            when check_e =>
              if noteb(8) = '1' then          -- if E is pressed, let the mux know
                note_pressed <= '1';
              end if;
              next_state <= check_f;
            when check_f =>
              if noteb(7) = '1' then          -- if F is pressed, let the mux know
                note_pressed <= '1';
              end if;
              next_state <= check_gflat;
            when check_gflat =>
              if noteb(6) = '1' then          -- if Gb is pressed, let the mux know
                note_pressed <= '1';
              end if;
              next_state <= check_g;
            when check_g =>
              if noteb(5) = '1' then          -- if G is pressed, let the mux know
                note_pressed <= '1';
              end if;
              next_state <= check_aflat;
            when check_aflat =>
              if noteb(4) = '1' then          -- if Ab is pressed, let the mux know
                note_pressed <= '1';
              end if;
              next_state <= check_a;
            when check_a =>
              if noteb(3) = '1' then          -- if A is pressed, let the mux know
                note_pressed <= '1';
              end if;
              next_state <= check_bflat;
            when check_bflat =>
              if noteb(2) = '1' then          -- if Bb is pressed, let the mux know
                note_pressed <= '1';
              end if;
              next_state <= check_b;
            when check_b =>
              if noteb(1) = '1' then          -- if B is pressed, let the mux know
                note_pressed <= '1';
              end if;
              next_state <= check_highc;
            when check_highc =>
              if noteb(0) = '1' then          -- if high C is pressed, let the mux know
                note_pressed <= '1';
              end if;
              next_state <= delay;
            when update_regs =>
              norm_reg_load_en <= '1';        -- load in the normalized value to the
second register
              next_state <= waiting;          -- start checking again
            when waiting =>
              if termcnt = '1' then         -- delay until next sampling time
```

```
            next_state <= check_lowc;
          end if;
        when delay =>                  -- gives time for normalization on high c
          next_state <= update_regs;
      end case;

    end process;


    update_state: process(clk)
    begin
      if rising_edge(clk) then
        state <= next_state;
      end if;
    end process;

    end Behavioral;


```

Clock Divider:

```
            --
--------------------------------------------------------------------------------
        -- Company: ENGS 31 15X
        -- Engineer: Stylianos Tegas and Liane Makatura
        --
        -- Create Date:    20:14:08 08/19/2015
        -- Design Name:
        -- Module Name:    ClockDivider - 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;

        -- Uncomment the following library declaration if using
        -- arithmetic functions with Signed or Unsigned values
        use IEEE.NUMERIC_STD.ALL;

        -- Uncomment the following library declaration if instantiating
        -- any Xilinx primitives in this code.
        --library UNISIM;
        --use UNISIM.VComponents.all;

        entity ClockDivider is
            Port ( clk : in  STD_LOGIC;
                   clk_div : out  STD_LOGIC);
        end ClockDivider;

        architecture Behavioral of ClockDivider is
        constant clk_div_value: integer := 3;
        signal count: unsigned (1 downto 0)  := (others => '0');
        signal tog: std_logic := '0';
        begin

        divider: process(clk)
        begin

          if rising_edge(clk) then
            if count = (clk_div_value - 1) then
              tog <= NOT(tog);          -- toggle the clock value
              count <= (others => '0');
            else
              count <= count + 1;
            end if;
          end if;

        end process;

        clk_div <= tog;       -- assign the output

        end Behavioral;
```

Select Counter:

```
            --
------------------------------------------------------------------------------------
        -- Company: ENGS 31 15X
        -- Engineer: Stylianos Tegas and Liane Makatura
        --
        -- Create Date:    20:09:36 08/19/2015
        -- Design Name:
        -- Module Name:    SelectCounter - 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;

        -- Uncomment the following library declaration if using
        -- arithmetic functions with Signed or Unsigned values
        use IEEE.NUMERIC_STD.ALL;

        -- Uncomment the following library declaration if instantiating
        -- any Xilinx primitives in this code.
        --library UNISIM;
        --use UNISIM.VComponents.all;

        entity SelectCounter is
            Port ( clk : in  STD_LOGIC;
                en : in STD_LOGIC;
                   selectbits : out  STD_LOGIC_VECTOR (3 downto 0));
        end SelectCounter;

        architecture Behavioral of SelectCounter is
        signal iQ: unsigned (3 downto 0) := "1100";
        begin

        count: process(clk)
        begin

          if rising_edge(clk) then

            if en = '1' AND iQ = "1100" then          --if enable is high and iQ has
reached terminal count, reset
              iQ <= "0000";
            elsif iQ >= "0000" AND iQ < "1100" then   --if iQ in process of counting,
continue
              iQ <= iQ + 1;
            else                            --if done counting but enable is not high,
feed 12
              iQ <= "1100";
            end if;
```

```
                  end if;

                end process;

                selectbits <= std_logic_vector(iQ);           --assign select bits

                end Behavioral;

```

Accumulator:

```
--
--------------------------------------------------------------------------------
-- Company: ENGS 31 15X
-- Engineer: Stylianos Tegas and Liane Makatura
--
-- Create Date:    13:57:27 08/22/2015
-- Design Name:
-- Module Name:    Accumulator - 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;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Accumulator is
    Port ( clk : in STD_LOGIC;
        note_to_add : in  STD_LOGIC_VECTOR (11 downto 0);
        load_en : in STD_LOGIC;
        clear_en : in STD_LOGIC;
           note_to_norm : out  STD_LOGIC_VECTOR (15 downto 0));
end Accumulator;

architecture Behavioral of Accumulator is
signal summed_note: std_logic_vector (15 downto 0) := (others => '0');

begin

accumulate: process(clk)
begin

  if rising_edge(clk) then
    summed_note <= summed_note;          -- accumulate the sin waves
    if clear_en = '1' then               -- if clear then clear register
      summed_note <= (others => '0');
    elsif load_en = '1' then             -- if load load new signal
      summed_note <= std_logic_vector(unsigned(summed_note) + unsigned(
note_to_add));
    end if;
  end if;

end process;

note_to_norm <= summed_note;
```

```
                end Behavioral;
```

Normalizer:

```
--
-----------------------------------------------------------------------------------
-- Company: ENGS 31 15X
-- Engineer: Stylianos Tegas and Liane Makatura
--
-- Create Date:    14:36:20 08/22/2015
-- Design Name:
-- Module Name:    Normalizer - 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;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Normalizer is
    Port ( clk : in STD_LOGIC;
        load_en : in STD_LOGIC;
        note_to_norm : in  STD_LOGIC_VECTOR (15 downto 0);
           num_note : in  STD_LOGIC_VECTOR (3 downto 0);
           final_note : out  STD_LOGIC_VECTOR (11 downto 0));
end Normalizer;

architecture Behavioral of Normalizer is
type array_dt is array (12 downto 0) of unsigned (11 downto 0);
constant mult_factors : array_dt :=     (to_unsigned(4095, 12), -- 1 note
                            to_unsigned(2896, 12), -- 2 notes
                            to_unsigned(2365, 12), -- 3 notes
                            to_unsigned(2048, 12), -- 4 notes
                            to_unsigned(1832, 12), -- 5 notes
                            to_unsigned(1672, 12), -- 6 notes
                            to_unsigned(1548, 12), -- 7 notes
                            to_unsigned(1448, 12), -- 8 notes
                            to_unsigned(1365, 12), -- 9 notes
                            to_unsigned(1295, 12), -- 10 notes
                            to_unsigned(1235, 12), -- 11 notes
                            to_unsigned(1182, 12), -- 12 notes
                            to_unsigned(1136, 12)  -- 13 notes
                            );
signal multi_note: std_logic_vector (27 downto 0) := (others => '0');
signal iQ: std_logic_vector (11 downto 0) := (others => '0');

begin

normalize: process(note_to_norm, num_note)
```

```
                begin

                  if num_note = "0000" then     -- prevents indexing out of bounds
                    multi_note <= std_logic_vector(unsigned(note_to_norm) * mult_factors(
    to_integer(unsigned(num_note))));
                  else
                    multi_note <= std_logic_vector(unsigned(note_to_norm) * mult_factors(
    to_integer(unsigned(num_note)-1)));
                  end if;

                end process;

                load: process(clk)
                begin

                  if rising_edge(clk) then
                    if load_en = '1' then
                      iQ <= multi_note(27 downto 16); --take first 12 bits
                    else
                      iQ <= iQ;
                    end if;
                  end if;

                end process;

                final_note <= iQ;

                end Behavioral;

```

Adda Converter:

```
--
----------------------------------------------------------------------------------
-- Company: Dartmouth College
-- Engineer: David Picard
-- Modified by Stylianos Tegas and Liane Makatura
--  Final Project: ENGS 31 15X
--
-- Create Date: 04/29/2015 02:32:19 PM
-- Design Name:
-- Module Name: adda - 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 adda is
    Port (
         clk  : in STD_LOGIC;
         tone : in STD_LOGIC_VECTOR (11 downto 0);
       sync : out std_logic;
       DinA : out std_logic;
       sclk : out std_logic
        );
end adda;

architecture Behavioral of adda is



signal lastsampleclk : STD_LOGIC;
signal sampleclk : STD_LOGIC := '0';

signal den_in : STD_LOGIC;
signal dwe_in : std_logic;
signal di_in : STD_LOGIC_VECTOR(15 downto 0);
signal daddr_in : std_LOGIC_vector(6 downto 0);
signal sampledvalue : STD_LOGIC_VECTOR(15 downto 0) := x"1234";
signal sampledvalue2 : STD_LOGIC_VECTOR(15 downto 0)  := x"5555";
signal DATA1 : STD_LOGIC_VECTOR(15 downto 0)  := x"2345";
signal DATA2 : STD_LOGIC_VECTOR(15 downto 0)  := x"3456";
signal busy, DONE, nSYNC, START : STD_LOGIC;
constant control     : std_logic_vector(3 downto 0)  := "0000";

type states is (Idle, ShiftOut, SyncData);
signal current_state : states;
```

```
            signal next_state    : states;

            signal temp1         : std_logic_vector(15 downto 0);
            signal clk_div       : std_logic;
            signal clk_counter   : unsigned(27 downto 0):= x"0000000";
            signal shiftCounter  : unsigned(4 downto 0);
            signal enShiftCounter: std_logic;
            signal enParalelLoad : std_logic;
            signal clk_counter2  : integer := 0;
            signal clk_en        : std_logic := '0';
            signal data_counter  : unsigned (4 downto 0) := "00000";
            signal data_counter2  : unsigned (11 downto 0) := x"000";

            begin

            sclk <= clk_en;


              Slow_clock_buffer: BUFG
                  port map (I => clk_en,
                            O => sampleclk );
            --
----------------------------------------------------------------------------------

            clock_divide2 : process(clk)
            begin
                if (clk = '1' and clk'event) then
                if clk_counter2 = 5 then clk_en <= not clk_en;
                   clk_counter2 <= 0;
                else clk_counter2 <= clk_counter2 + 1;
                end if;
                end if;
            end process;
            --
----------------------------------------------------------------------------------
            count_bits : process (sampleclk)
            begin
               if (sampleclk'event) and (sampleclk = '1') then
                start <= '0';
                if data_counter = "11001" then data_counter <= (others => '0');
                  start <= '1';
                  else data_counter <= data_counter + 1;
                  end if;

                end if; --clk
              end process;



            --
---------------------------------------------------------------------------------

            counter : process(sampleclk)
                    begin
                        if (sampleclk = '1' and sampleclk'event) then
                       DinA <= temp1(15);
                       if start = '1' then
                        temp1 <= "0000" & not(tone(11)) & tone(10 downto 0);
                        end if;
                           if enParalelLoad = '1' then
                               shiftCounter <= "00000";

                            elsif (enShiftCounter = '1') then
                               temp1 <= temp1(14 downto 0)&temp1(15);
                               shiftCounter <= shiftCounter + 1;
                            end if;
                        end if;
                    end process;

            --
    ------------------------------------------------------------------------------------
            SYNC_PROC: process (sampleclk)
               begin
                  if (sampleclk'event and sampleclk = '1') then
                        current_state <= next_state;
                        sync <= nsync;
                  end if;
               end process;


            --
    --------------------------------------------------------------------------------------
            OUTPUT_DECODE: process (current_state)
               begin
                  if current_state = Idle then
                        enShiftCounter <='0';
                        nSYNC <='1';
                        enParalelLoad <= '1';
                    elsif current_state = ShiftOut then
                        enShiftCounter <='1';
                        nSYNC <='0';
                        enParalelLoad <= '0';
                    else --if current_state = SyncData then
                        enShiftCounter <='0';
                        nSYNC <='1';
                        enParalelLoad <= '0';
                    end if;
               end process;


            --
    ------------------------------------------------------------------------------------
                NEXT_STATE_DECODE: process (current_state, START, shiftCounter)
               begin

                  next_state <= current_state;

                  case (current_state) is
                     when Idle =>
                        if START = '1' then
                           next_state <= ShiftOut;
                        end if;
                     when ShiftOut =>
                        if shiftCounter = "01111" then --"10000" then
                           next_state <= SyncData;
                        end if;
                     when SyncData =>
                        if START = '0' then
                        next_state <= Idle;
                        end if;
                     when others =>
                        next_state <= Idle;
                  end case;
               end process;


            end Behavioral;
```

UCF:

```
## This file is a general .ucf for Nexys3 rev B board
## To use it in a project:
## - remove or comment the lines corresponding to unused pins
## - rename the used signals according to the project

## Clock signal
NET "clk"            LOC = "V10" | IOSTANDARD = "LVCMOS33";   #Bank = 2, pin name =
    IO_L30N_GCLK0_USERCCLK,            Sch name = GCLK
Net "clk" TNM_NET = sys_clk_pin;
TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 100000 kHz;
PIN "Inst_adda/Slow_clock_buffer.O" CLOCK_DEDICATED_ROUTE = FALSE;

## 7 segment display
#NET "seg<0>"         LOC = "T17" | IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name =
    IO_L51P_M1DQ12,                    Sch name = CA
#NET "seg<1>"         LOC = "T18" | IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name =
    IO_L51N_M1DQ13,                    Sch name = CB
#NET "seg<2>"         LOC = "U17" | IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name =
    IO_L52P_M1DQ14,                    Sch name = CC
#NET "seg<3>"         LOC = "U18" | IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name =
    IO_L52N_M1DQ15,                    Sch name = CD
#NET "seg<4>"         LOC = "M14" | IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name = IO_L53P
    ,                           Sch name = CE
#NET "seg<5>"         LOC = "N14" | IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name =
    IO_L53N_VREF,                      Sch name = CF
#NET "seg<6>"         LOC = "L14" | IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name = IO_L61P
    ,                           Sch name = CG
#NET "seg<7>"         LOC = "M13" | IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name = IO_L61N
    ,                           Sch name = DP

#NET "an<0>"          LOC = "N16" | IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name =
    IO_L50N_M1UDQSN,                   Sch name = AN0
#NET "an<1>"          LOC = "N15" | IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name =
    IO_L50P_M1UDQS,                    Sch name = AN1
#NET "an<2>"          LOC = "P18" | IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name =
    IO_L49N_M1DQ11,                    Sch name = AN2
#NET "an<3>"          LOC = "P17" | IOSTANDARD = "LVCMOS33";   #Bank = 1, Pin name =
    IO_L49P_M1DQ10,                    Sch name = AN3


## Leds
#NET "Led<0>"         LOC = "U16" | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name =
    IO_L2P_CMPCLK,                     Sch name = LD0
#NET "Led<1>"         LOC = "V16" | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name =
    IO_L2N_CMPMOSI,                    Sch name = LD1
#NET "Led<2>"         LOC = "U15" | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name = IO_L5P,
                                Sch name = LD2
#NET "Led<3>"         LOC = "V15" | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name = IO_L5N,
                                Sch name = LD3
#NET "Led<4>"         LOC = "M11" | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name = IO_L15P
    ,                           Sch name = LD4
#NET "Led<5>"         LOC = "N11" | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name = IO_L15N
    ,                           Sch name = LD5
#NET "Led<6>"         LOC = "R11" | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name = IO_L16P
    ,                           Sch name = LD6
#NET "Led<7>"         LOC = "T11" | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name =
    IO_L16N_VREF,                      Sch name = LD7


## Switches
NET "sus"           LOC = "T10" | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name =
    IO_L29N_GCLK2,                     Sch name = SW0
#NET "dflatb"          LOC = "T9"  | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name =
    IO_L32P_GCLK29,                    Sch name = SW1
#NET "db"          LOC = "V9"  | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name =
    IO_L32N_GCLK28,                    Sch name = SW2
#NET "eflatb"          LOC = "M8"  | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name =
    IO_L40P,                           Sch name = SW3
#NET "maj"          LOC = "N8"  | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name = IO_L40N,
                              Sch name = SW4
#NET "min"          LOC = "U8"  | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name = IO_L41P,
                              Sch name = SW5
#NET "dim"          LOC = "V8"  | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name =
    IO_L41N_VREF,                      Sch name = SW6
#NET "aug"          LOC = "T5"  | IOSTANDARD = "LVCMOS33";   #Bank = MISC, Pin name =
    IO_L48N_RDWR_B_VREF_2,          Sch name = SW7


## Buttons
#NET "aflatb"          LOC = "B8"  | IOSTANDARD = "LVCMOS33";   #Bank = 0, Pin name = IO_L33P
    ,                           Sch name = BTNS
#NET "maj"          LOC = "A8"  | IOSTANDARD = "LVCMOS33";   #Bank = 0, Pin name = IO_L33N,
                              Sch name = BTNU
#NET "dim"          LOC = "C4"  | IOSTANDARD = "LVCMOS33";   #Bank = 0, Pin name =
    IO_L1N_VREF,                       Sch name = BTNL
#NET "min"          LOC = "C9"  | IOSTANDARD = "LVCMOS33";   #Bank = 0, Pin name =
    IO_L34N_GCLK18,                    Sch name = BTND
#NET "aug"          LOC = "D9"  | IOSTANDARD = "LVCMOS33";   #Bank = 0, Pin name =
    IO_L34P_GCLK19,                    Sch name = BTNR


## 12 pin connectors

##JA
#NET "JA<0>"          LOC = "T12" | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name = IO_L19P
    ,                           Sch name = JA1
NET "noteb<0>"          LOC = "V12" | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name =
    IO_L19N,                           Sch name = JA2
#NET "JA<2>"          LOC = "N10" | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name = IO_L20P
    ,                           Sch name = JA3
NET "noteb<1>"          LOC = "P11" | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name =
    IO_L20N,                           Sch name = JA4
#NET "JA<4>"          LOC = "M10" | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name = IO_L22P
    ,                           Sch name = JA7
#NET "JA<5>"          LOC = "N9"  | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name = IO_L22N
    ,                           Sch name = JA8
#NET "JA<6>"          LOC = "U11" | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name = IO_L23P
    ,                           Sch name = JA9
#NET "JA<7>"          LOC = "V11" | IOSTANDARD = "LVCMOS33";   #Bank = 2, Pin name = IO_L23N
    ,                           Sch name = JA10

##JB
NET "noteb<2>"          LOC = "K2"  | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name =
    IO_L38P_M3DQ2,                     Sch name = JB1
NET "noteb<3>"          LOC = "K1"  | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name =
    IO_L38N_M3DQ3,                     Sch name = JB2
NET "noteb<4>"          LOC = "L4"  | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name =
    IO_L39P_M3LDQS,                    Sch name = JB3
NET "noteb<5>"          LOC = "L3"  | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name =
    IO_L39N_M3LDQSN,                   Sch name = JB4
#NET "RsRx_p"          LOC = "J3"  | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name =
    IO_L40P_M3DQ6,                     Sch name = JB7
#NET "JB<5>"          LOC = "J1"  | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name =
    IO_L40N_M3DQ7,                     Sch name = JB8
#NET "JB<6>"          LOC = "K3"  | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name =
    IO_L42N_GCLK24_M3LDM,              Sch name = JB9
#NET "JB<7>"          LOC = "K5"  | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name =
    IO_L43N_GCLK22_IRDY2_M3CASN,       Sch name = JB10
```

```
##JC
NET "noteb<6>"           LOC = "H3"   | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name =
    IO_L44N_GCLK20_M3A6,                 Sch name = JC1
NET "noteb<7>"           LOC = "L7"   | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name =
    IO_L45P_M3A3,                        Sch name = JC2
#NET "JC<2>"           LOC = "K6"   | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name =
    IO_L45N_M3ODT,                       Sch name = JC3
NET "noteb<8>"           LOC = "G3"   | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name =
    IO_L46P_M3CLK,                       Sch name = JC4
#NET "rx_done_tick_p"           LOC = "G1"  | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name
    = IO_L46N_M3CLKN,                      Sch name = JC7
#NET "JC<5>"           LOC = "J7"   | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name =
    IO_L47P_M3A0,                        Sch name = JC8
#NET "JC<6>"           LOC = "J6"   | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name =
    IO_L47N_M3A1,                        Sch name = JC9
#NET "JC<7>"           LOC = "F2"   | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name =
    IO_L48P_M3BA0,                       Sch name = JC10

##JD, LX16 Die only
NET "noteb<9>"           LOC = "G11" | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name =
    IO_L40P,                             Sch name = JD1
NET "noteb<10>"           LOC = "F10" | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name =
    IO_L40N,                             Sch name = JD2
NET "noteb<11>"           LOC = "F11" | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name =
    IO_L42P,                             Sch name = JD3
NET "noteb<12>"           LOC = "E11" | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name =
    IO_L42N,                             Sch name = JD4
NET "sync"           LOC = "D12" | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name = IO_L47P,
                              Sch name = JD7
NET "DinA"           LOC = "C12" | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name = IO_L47N,
                              Sch name = JD8
#NET "JD<6>"           LOC = "F12" | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name = IO_L51P
    ,                               Sch name = JD9
NET "sclk"           LOC = "E12" | IOSTANDARD = "LVCMOS33";   #Bank = 3, Pin name = IO_L51N,
                              Sch name = JD10
```

Keyboard Testbench:

```vhdl
--------------------------------------------------------------------------------
-- Company: ENGS31 15X
-- Engineer: Stylianos Tegas and Liane Makatura
--
-- Create Date:   19:35:14 08/20/2015
-- Design Name:
-- Module Name:   O:/engs31/Keyboard/KeyboardTest.vhd
-- Project Name:  Keyboard
-- Target Device:
-- Tool versions:
-- Description:
--
-- VHDL Test Bench Created by ISE for module: KeyboardTop
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-- Notes:
-- This testbench has been automatically generated using types std_logic and
-- std_logic_vector for the ports of the unit under test.  Xilinx recommends
-- that these types always be used for the top-level I/O of a design in order
-- to guarantee that the testbench will bind correctly to the post-implementation
-- simulation model.
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY KeyboardTest IS
END KeyboardTest;

ARCHITECTURE behavior OF KeyboardTest IS

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

    COMPONENT KeyboardTop
    PORT(
         clk : IN  std_logic;
      noteb : in STD_LOGIC_VECTOR (12 downto 0);
      sus : in std_logic;
         sync : OUT  std_logic;
         sclk : OUT  std_logic;
         DinA : OUT  std_logic
        );
    END COMPONENT;


   --Inputs
   signal clk : std_logic := '0';
   signal noteb : std_logic_vector (12 downto 0) := (others => '0');
  signal sus : std_logic := '0';

  --Outputs
   signal sync : std_logic;
   signal sclk : std_logic;
   signal DinA : std_logic;
  -- Clock period definitions
  constant clk_period : time := 10 ns;
  constant sclk_period : time := 10 ns;

BEGIN

  -- Instantiate the Unit Under Test (UUT)
  uut: KeyboardTop PORT MAP (
         clk => clk,
         noteb => noteb,
      sus => sus,
         sync => sync,
         sclk => sclk,
         DinA => DinA
       );

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



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

     wait for clk_period*10;

     -- insert stimulus here

   noteb(0) <= '1';
   wait for 2E4*clk_period;
   noteb(0) <= '0';
   wait for 1E4*clk_period;

   noteb(0) <= '1';
   noteb(4) <= '1';
   noteb(7) <= '1';
   noteb(12) <= '1';
   wait for 3E4*clk_period;


   sus <= '1';
   wait for 1E4*clk_period;


   noteb(0) <= '0';
   noteb(4) <= '0';
   noteb(7) <= '0';
   noteb(12) <= '0';
   wait for 2E4*clk_period;

   sus <= '0';

     wait;

  end process;

```

```
END;
```

3. **Resource Utilization**

   According to the Design Summary, our keyboard design had the following resource usage on the FPGA (the following is taken directly from the synthesis report):

   Slice Logic Utilization: Number of Slice Registers: 348 out of 18224 1%
   Number of Slice LUTs: 487 out of 9112 5%
   Number used as Logic: 461 out of 9112 5%
   Number used as Memory: 26 out of 2176 1%
   Number used as SRL: 26

   Slice Logic Distribution:
   Number of LUT Flip Flop pairs used: 540
   Number with an unused Flip Flop: 192 out of 540 35%
   Number with an unused LUT: 53 out of 540 9%
   Number of fully used LUT-FF pairs: 295 out of 540 54%
   Number of unique control sets: 20

   IO Utilization:
   Number of IOs: 18
   Number of bonded IOBs: 18 out of 232 7%
   Specific Feature Utilization:
   Number of Block RAM/FIFO: 11 out of 32 34%
   Number using Block RAM only: 11
   Number of BUFG/BUFGCTRLs: 3 out of 16 18%
   Number of DSP48A1s: 2 out of 32 6%

4. **Critical Timing Path**
   According to the Synthesis Report, the critical path in our keyboard is the delay from the controller to the normalizer with the number of notes signal. The critical timing path is 7.431ns, of which 4.372ns is from the logic and 3.059ns is from the routing of the signal. Below is the full report from the synthesis report:

   ==
   Timing constraint: Default period analysis for Clock Ínst_ClockDivider/tog´
   Clock period: 7.431ns (frequency: 134.576MHz)
   Total number of paths / destination ports: 3484 / 394
   ——————————————————————————-
   Delay: 7.431ns (Levels of Logic = 4)
   Source: num_notes_2 (FF)
   Destination: Inst_Normalizer/iQ_0 (FF)
   Source Clock: Inst_ClockDivider/tog rising
   Destination Clock: Inst_ClockDivider/tog rising
   Data Path: num_notes_2 to Inst_Normalizer/iQ_0
   Gate Net
   Cell:in-¿out fanout Delay Delay Logical Name (Net Name)
   ——————————————- ———— FDR:C-¿Q 26 0.525 1.696 num_notes_2 (num_notes_2)
   begin scope: 'Inst_Normalizer:num_note¡2¿'
   LUT4:I0-¿O 1 0.254 0.681 GND_13_o_X_13_o_wide_mux_3_OUT¡2¿11 (GND_13_o_X_13_o_wide_mux_3_OUT¡2¿)
   DSP48A1:A2-¿M27 1 3.265 0.682
   Mmult_note_to_norm[15]_GND_13_o_MuLt_4_OUT

(note_to_norm[15]_GND_13_o_MuLt_4_OUT¡27¿)
LUT5:I4-¿O 1 0.254 0.000 Mmux_multi_note121 (multi_note¡27¿)
FDE:D 0.074 iQ_11
—————————————-
Total 7.431ns (4.372ns logic, 3.059ns route)
(58.8% logic, 41.2% route)

This implies that the maximum possible frequency at which our design can be clocked is $\frac{1}{7.431} = 0.1346$ gigahertz, or $f_{max} = 134.6MHz$.

5. **Analysis of Residual Warnings**
Excerpt:
WARNING:Xst:1293 - FF/Latch <note_addresses_10_0>has a constant value of 0 in block <KeyboardTop>. This FF/Latch will be trimmed during the optimization process. WARNING:Xst:1293 - FF/Latch <note_addresses_10_1>has a constant value of 0 in block <KeyboardTop>. This FF/Latch will be trimmed during the optimization process.

Analysis: These warnings come about because the step size for D (stored in the 10th element of the note_addresses array) is 436, which is divisible by 4. Thus, neither of the two least significant bits will ever change. Since the value will always be 0, VHDL was able to optimize the design by chopping off these extra flip flops and hard wiring in the appropriate 0 values for the last two bits.

Excerpt:
WARNING:Xst:1293 - FF/Latch <note_addresses_11_0>has a constant value of 0 in block <KeyboardTop>. This FF/Latch will be trimmed during the optimization process. WARNING:Xst:1293 - FF/Latch <note_addresses_11_1>has a constant value of 0 in block <KeyboardTop>. This FF/Latch will be trimmed during the optimization process.

Analysis: These warnings come about because the step size for $C^\sharp/D^\flat$ (stored in the 11th element of the note_addresses array) is 412, which is divisible by 4. Thus, neither of the two least significant bits will ever change. Since the value will always be 0, VHDL was able to optimize the design by chopping off these extra flip flops and hard wiring in the appropriate 0 values for the last two bits.

Excerpt:
WARNING:Xst:1293 - FF/Latch <note_addresses_6_0>has a constant value of 0 in block <KeyboardTop>. This FF/Latch will be trimmed during the optimization process.

Analysis: This warning comes about because the step size for $F^\sharp/G^\flat$ (stored in the 6th element of the note_addresses array) is 550, which is divisible by 2. Thus the least significant bit will never change. Since the value will always be 0, VHDL was able to optimize the design by chopping off this flip flop and hard wiring a 0.

Excerpt:
WARNING:Xst:1293 - FF/Latch <note_addresses_9_0>has a constant value of 0 in block <KeyboardTop>. This FF/Latch will be trimmed during the optimization process.

Analysis: This warning comes about because the step size for $D^\sharp/E^\flat$ (stored in the 9th element of the note_addresses array) is 462, which is divisible by 2. Thus the least significant bit will never change. Since the value will always be 0, VHDL was able to optimize the design by chopping off this flip flop and hard wiring a 0.

Excerpt:
WARNING:Xst:1293 - FF/Latch <note_addresses_8_0>has a constant value of 0 in block <Keyboard-Top>. This FF/Latch will be trimmed during the optimization process.

Analysis: This warning comes about because the step size for E (stored in the 6th element of the note_addresses array) is 490, which is divisible by 2. Thus the least significant bit will never change. Since the value will always be 0, VHDL was able to optimize the design by chopping off this flip flop and hard wiring a 0.

Excerpt:
WARNING:Xst:1293 - FF/Latch <note_addresses_1_0>has a constant value of 0 in block <Keyboard-Top>. This FF/Latch will be trimmed during the optimization process.

Analysis: This warning comes about because the step size for B (stored in the 1st element of the note_addresses array) is 734, which is divisible by 2. Thus the least significant bit will never change. Since the value will always be 0, VHDL was able to optimize the design by chopping off this flip flop and hard wiring a 0.

Excerpt:
WARNING:Xst:1293 - FF/Latch <note_addresses_3_0>has a constant value of 0 in block <Keyboard-Top>. This FF/Latch will be trimmed during the optimization process.

Analysis: This warning comes about because the step size for A (stored in the 0th element of the note_addresses array) is 654, which is divisible by 2. Thus the least significant bit will never change. Since the value will always be 0, VHDL was able to optimize the design by chopping off this flip flop and hard wiring a 0.

Excerpt:
WARNING:Xst:1293 - FF/Latch <note_addresses_0_0>has a constant value of 0 in block <Keyboard-Top>. This FF/Latch will be trimmed during the optimization process.

Analysis: This warning comes about because the step size for $C5$ (stored in the 0th element of the note_addresses array) is 778, which is divisible by 2. Thus the least significant bit will never change. Since the value will always be 0, VHDL was able to optimize the design by chopping off this flip flop and hard wiring a 0.

7.3 Memory Map

Our project used a sine Lookup Table (LUT) that was generated by the DDS Compiler 4.0, found in the IP Core Generator. This LUT contains 2^{16} incremental values of a sin wave, where each value is 12 bits long. This requires a substantial amount of block memory (on the order of 64,000 * 12 bits), but surprisingly we only ended up using 11 block rams to do it. While we are not exactly sure how the Core Generator and Xilinx optimized our memory, we do believe that it was able to recognize and exploit the symmetry of the sin wave values, thus only needing to store $\frac{1}{4}$ of the original memory amount. In addition to the apparent flip flops and registers that are pictured in our block diagram, we also utilized block memory to store our arrays for the note steps and the current note addresses. The note step values are what we used to update the value of the note address each time we wanted to sample, so that we could get the appropriate frequencies for each note. These values remained constant. The note addresses array stored the current address for each of the 13 notes, and this value was updated then passed into the sin LUT to get the appropriate value for the note on each sampling cycle.

7.4 Waveform Graphs

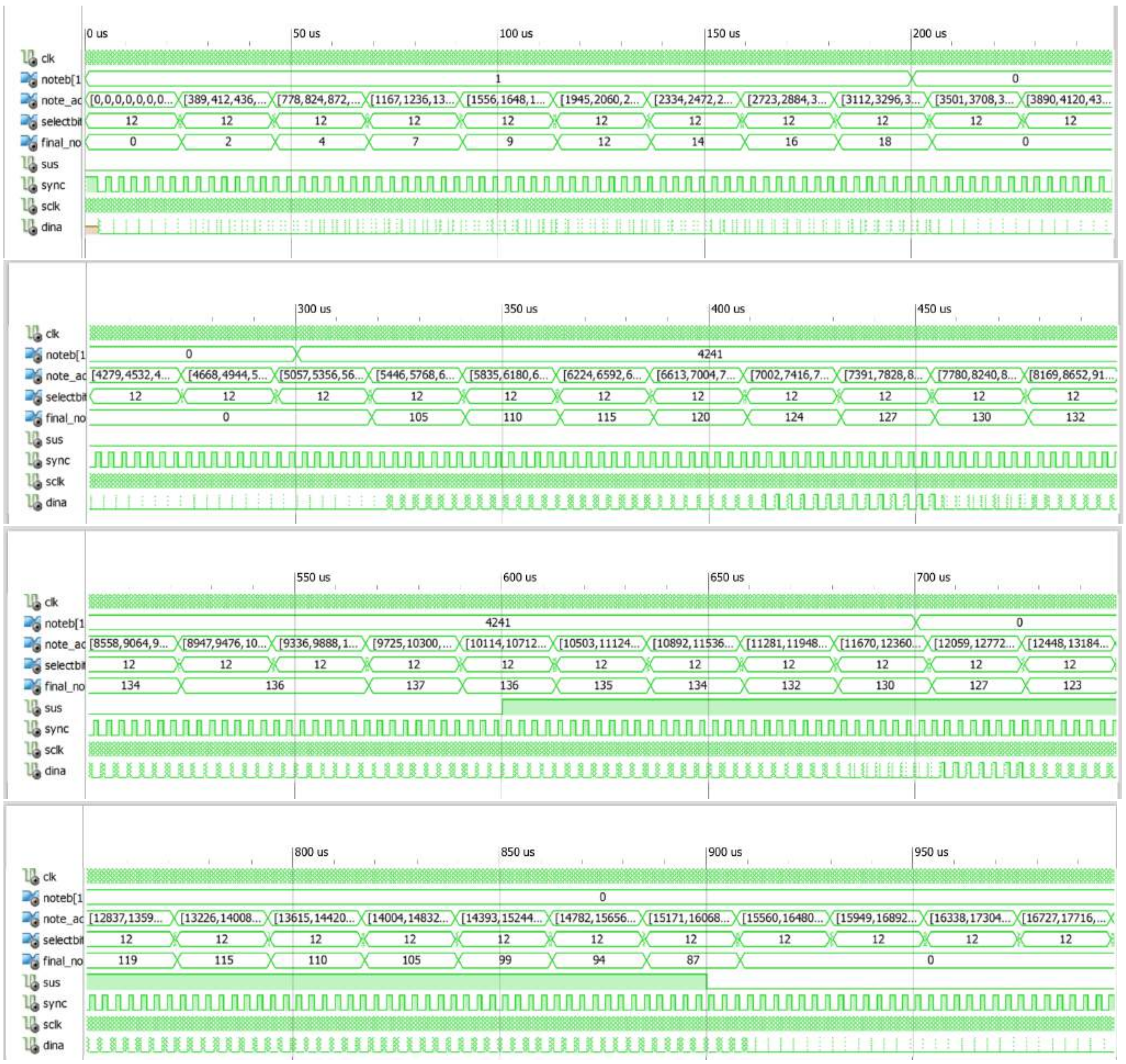

Figure 6: These images (chronologically arranged from top to bottom) show the output of our testbench for the KeyboardTop module, which runs the entire project with all components included. This testbench is included in the VHDL code section, and the waveform shows all values of the inputs and outputs of the system, beginning with the buttons and ending with the outputs of the DAC.

7.5 Data Sheets

All of our external components were manufactured by Digilent, so there were no additional data sheets to include.

7.6 Computer Programs

```
    % Liane Makatura
    % Finding the m values for various frequencies

    freqs = [261.63 277.18 293.66 311.13 329.63 349.23 369.99 392.00 415.30 440.00 466.16
493.88 523.25];

    samp_rate = 44100;
    n_bits = 16;

    mvals = freqs .* (2^n_bits) ./ samp_rate;

    disp(mvals)
```

Note	Resulting Step	Integer M Value
C4	388.8024	389
$C^\sharp/D^\flat$	411.9108	412
D	436.4014	436
$D^\sharp/E^\flat$	462.3632	462
E	489.8556	490
F	518.9827	519
$F^\sharp/G^\flat$	549.8337	550
G	582.5422	583
$G^\sharp/A^\flat$	617.1678	617
A	653.8739	654
$A^\sharp/B^\flat$	692.7497	693
B	733.9438	734
C5	777.5898	778

Table 2: M-values for our 13 notes, obtained from the MATLAB code sample above. These calculations are based on a 44.1kHz sampling rate, and 2^{16} samples of the sine wave.

```
    % Liane Makatura
    % Calculate strength reduction factors

    denom = 4096;

    for num=2:13
        ideal_mult = 1/sqrt(num);
        numer = floor(ideal_mult*4000);
        error = abs((numer / denom) - ideal_mult);
        nextdiff = abs(((numer+1) / denom) - ideal_mult);
        while error > 0.001 || error > nextdiff
            numer = numer + 1;
            error = nextdiff;
            nextdiff = abs(((numer+1) / denom) - ideal_mult);
        end

        fprintf('Notes: %d\n', num)
        fprintf('Ideal Divisor: %6.5f\n', ideal_mult)
        fprintf('Multiplier: %d\n', numer)
        fprintf('Appx Divisor: %6.5f\n', (numer/denom))
        fprintf('Error: %6.5f\n\n', abs((numer/denom)-ideal_mult))

    end
```

From this simple MATLAB program, we were able to generate the following approximations, which we used to normalize the accumulated frequencies for polyphony:

Number of Notes	Ideal Multiplier	Integer Multiplier	Approximated Multiplier	Error
1	$\frac{1}{\sqrt{1}} = 1.00000$	4096	1.00000	0.00000
2	$\frac{1}{\sqrt{2}} \approx 0.70711$	2896	0.70703	0.00008
3	$\frac{1}{\sqrt{3}} \approx 0.57735$	2365	0.57739	0.00004
4	$\frac{1}{\sqrt{4}} = 0.50000$	2048	0.50000	0.00000
5	$\frac{1}{\sqrt{5}} \approx 0.44721$	1832	0.44727	0.00005
6	$\frac{1}{\sqrt{6}} \approx 0.40825$	1672	0.40820	0.00005
7	$\frac{1}{\sqrt{7}} \approx 0.37796$	1548	0.37793	0.00003
8	$\frac{1}{\sqrt{8}} \approx 0.35355$	1448	0.35352	0.00004
9	$\frac{1}{\sqrt{9}} = 0.33333$	1365	0.33325	0.00008
10	$\frac{1}{\sqrt{10}} \approx 0.31623$	1295	0.31616	0.00007
11	$\frac{1}{\sqrt{11}} \approx 0.30151$	1235	0.30151	0.00000
12	$\frac{1}{\sqrt{12}} \approx 0.28868$	1182	0.28857	0.00010
13	$\frac{1}{\sqrt{13}} \approx 0.27735$	1136	0.27734	0.00001

Table 3: Consolidated results from the code snippet above. All approximations assume a 12-bit shift (or division by 4096) after the integer multiplication.