Digital Synthesizer

Robert Third

&

Zawaad Sobhan

Abstract

A synthesizer is an electronic musical instrument that generates electric signals that is converted into sound using amplifiers. The process is normally done in the analog domain using methods such as Subtractive synthesis and Additive synthesis. In additive synthesis, different frequency waves are combined and then moulded through an ADSR (Attack – Decay – Sustain - Release) envelope. This report describes how we modelled this operation in the digital domain using a combination of Direct Digital and Table Look-Up synthesis. The Altera Cyclone IV FPGA was programmed using System Verilog to perform the necessary functions. We decided to embark on this project because of our shared interest in electronic music and the instruments used to create it.

Table of Contents

Table of Tables

Table of Figures

Overview

Analog synthesizers utilize a combination of various voltage controlled oscillators, filters and amplifiers to imitate the sounds of different instruments such as pianos, organs and flutes. These sounds can be obtained using methods such as subtractive and additive synthesis. In additive synthesis sine waves of different frequencies and amplitudes are meshed together to produce a certain timbre. This output is then modulated using an ADSR (Attack – Decay – Sustain - Release) envelope. The envelope influences the way we perceive the generated timbre by controlling how long it takes to reach its maximum amplitude and decay to zero from it. In Subtractive synthesis, the timbre is acted upon by a voltage controlled low pass filter. For our project, we decided to digitally implement the Additive synthesis approach. We felt that sequential logic nature of System Verilog would lend itself well to the generation of the ADSR envelope. The combination of sine waves frequencies was produced using a combination of Direct Digital and Table Look-Up synthesis. MATLAB was used to create a sine wave look-up table (LUT) of the amplitudes for one full cycle (0 - 2π). In Direct Digital synthesis, a Numerically Controlled Oscillator (NCO) synchronously increments a value in a phase accumulator which is used as an index in the look-up table to provide the respective amplitude for that phase. The onboard clock of the Altera Cyclone IV FPGA (Field Programmable Gate Array) was used to time these operations. Pulse Width Modulation was required to convert the digital amplitudes to their analog equivalent. The pulse width modulated signals are usually passed through a reconstruction low pass filter but we opted out of this route as attenuated the sound significantly. ModelSim was used to test and debug the code used to program the FPGA. The onboard LEDs of the FPGA, a seven-segment decoder connected as a peripheral device and an oscilloscope were also used in the debugging process. A 4x4 Keypad connected as a peripheral device to the FPGA had its' keys programmed to generated different packages of fundamental and harmonic frequencies. Quartus Prime 16.1 was used to compile and program the FPGA. Refer to the Table of Contents to locate any information regarding any of the modules used in the construction of this digital synthesizer.

MATLAB and Sinewave Values

For our project to work properly we need to have a table of sinewave value amplitudes that we step through to generate the desired output frequency. We require the values for one complete of a sinewave. To do generate this table of sinewave values we used a short MATLAB script. This script will generate essentially 256 samples of one complete cycle. To ensure that values code be converted into Hex values we had to scale the amplitude values to go from 0-255 which corresponds to 0-FF in Hex. This generated file sintable.tv is read into memory in sinewave.sv and is stepped through by the phase accumulator. The MATLAB script can be found in the appendix along with the rest of system Verilog modules.

Direct Digital Synthesis

In this method, a Frequency Control Register that feeds the phase accumulator in a Numerically Controlled Oscillator (NCO). The Frequency Control Register holds the computed value or step size that the phase accumulator must be incremented at to produce a signal of the desired frequency. The following equation was used to calculate the necessary step sizes:

$$\Delta N = \frac{f_d * 2^k}{f_{clk}}$$

ΔN represents the frequency control value or step size while f_d and f_{clk} are the desired frequency and clock frequency respectively. The 'k' value is the size of the phase accumulator in bits. The synchronously incrementing value stored in the phase accumulator acts as an index to the MATLAB generated sine wave phase to amplitude look-up table. This allows us to extract the corresponding sine amplitude.

Keypad

The keypad is used to determine which frequency needs to be outputted to the speaker. To do this we use three modules: colseg.sv, kpdecode.sv & optionselect.sv. Colseq.sv and kpdecode.sv are used to poll the keypad and retrieve which key if any have been pressed. Once we know which key has been pressed we use optionselect.sv to determine which group of frequencies have been mapped to that key.

To determine the frequencies used we chosen them randomly, we just want to keep them under 3kHz to save our ears. We used the formula mentioned in the direct digital synthesis section to generate the delta N values that is used to step through the sinewave table, this stepping is what generates the desired group of frequencies.

For instance, if we want a 500 Hz output frequency our delta N step size is would be 5.12 based on the 25kHz clock we used. This value means that when reading values from sinewave table we are reading every 6th value.

Below we will include the keypad layout and which frequencies we mapped to each key.

Table 1: Table of Keypad Buttons

S1 • 500 Hz Fundamental • No Harmonics	S2 • 500 Hz Fundamental • 625 Hz & 750 Hz Harmonics	S3 • 500 Hz Fundamental • 1250 Hz & 1500 Hz Harmonics	S3 • 500 Hz Fundamental • 2250 Hz & 2500 Hz Harmonics
S4 • 1000 Hz Fundamental • No Harmonics	S5 • 1000 Hz Fundamental • 1250 Hz & 1500 Hz Harmonics	S6 • 1000 Hz Fundamental • 625 Hz & 750 Hz Harmonics	S7 • 1000 Hz Fundamental • 2250 Hz & 2500 Hz Harmonics
S8 • 2000 Hz Fundamental • No Harmonics	S9 • 2000 Hz Fundamental • 2250 Hz & 2500 Hz Harmonics	S10 • 2000 Hz Fundamental • 625 Hz & 750 Hz Harmonics	S11 • 2000 Hz Fundamental • 1250 Hz & 1500 Hz Harmonics
S12 • 500 Hz Fundamental • 1000 Hz & 2000 Hz Harmonics	S13 • No fundamental • No Harmonic	S14 • 1000 Hz Fundamental • 500 Hz & 2000 Hz Harmonics	S15 • 2000 Hz Fundamental • 500 Hz & 1000 Hz Harmonics

Pulse Width Modulation

The FPGA we used had no built-in DAC so we choose to perform Pulse width modulation (PWM) before we output to our speaker. The PWM was implemented in sinewave.sv and all it does is converts the value of the sine wave amplitude into a pulse of 1s or 0s. We were lucky that we did not need to make a reconstruction filter for the PWM output to recover an analog sinewave. We could directly connect our speaker to the output from the FPGA.

ADSR (Attack- Decay-Sustain-Release) Envelope

The ADSR envelope determines the level of sound over time. It is the combination of a ADSR profile and timbre that decides whether a sound is perceived as a drum noise, piano note, guitar note, etc. The envelope delineates the modulation of the sound's amplitude over the time that a key is pressed.

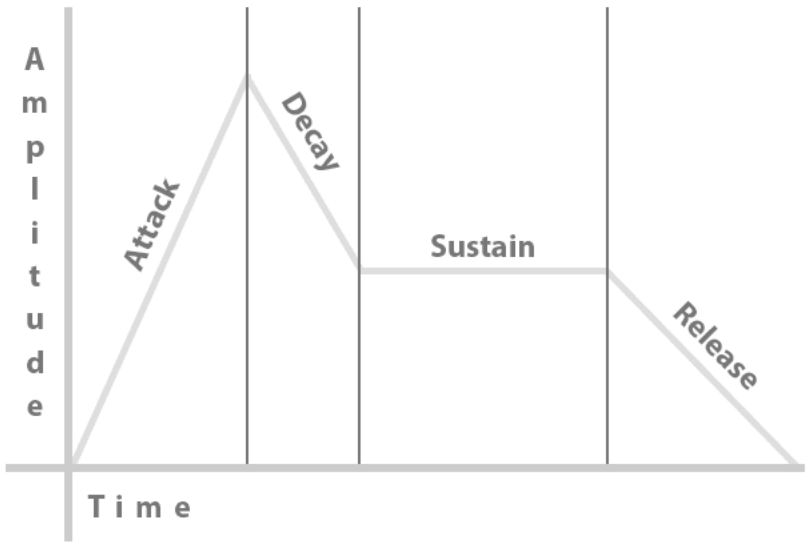

Figure 1: Standard Attack, Decay, Sustain, Release Profile [1]

The envelope can be described by the following four sections:

Attack: The attack controls the time it takes for the signal to reach its maximum amplitude. Percussive sounds have a very quick attack phase whereas string sounds have longer attacks.

Decay: The decay phase takes control as soon as the signal has reached its maximum level. It attenuates the signal to a set sustain level. Percussive sounds have short decays while strings have longer decays.

Sustain: This phase holds the signal at the set sustain level that it was decremented to in the decay phase. This state exists if the key is pressed. Piano notes and percussive sounds have a long sustain. String sounds are characterized by short sustain periods.

Release: The signal is decremented to 0 once the key is released. String sounds have fast release phases.

The ADSR envelope was developed as a state machine. As soon as a key is pressed the state machine moves out of its idle state and into the attack phase. In this phase, the output is reduced by a set maximum value. The set value is decremented for every clock cycle that the machine is in the attack phase. It is decremented until it becomes 0 and the sine amplitude value is being subtracted by 0 allowing it to reach its maximum amplitude. Once the stack scalar use to decrement the amplitude reaches 0 the state machine moves into the decay phase given that the key is still pressed. Here the amplitude is subtracted by a decay scalar that becomes smaller every clock edge. The decay phase continues until the decay scalar reaches a user selected sustain level after which it enters the sustain state. This state continues until the keypad key is released. When the key is released the machine enters the release phase where the output is decremented by a scalar till the output is 0. The state machine described in this section can be seen in Figure 2.

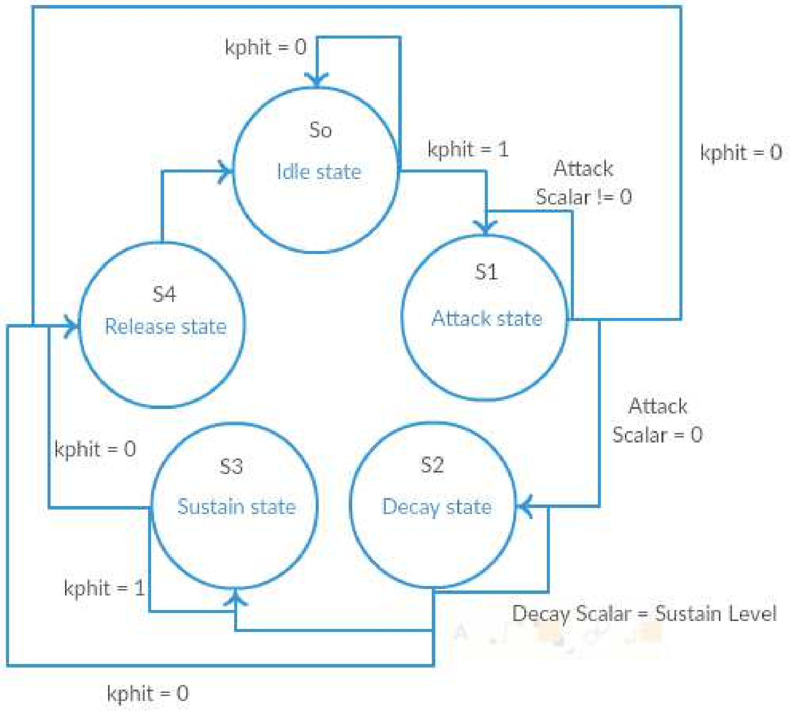

Figure 2: ADSR State Machine

Testing and Debugging

The digital synthesizer required a column sequencer, keypad decoder, option select, clock and sine wave generation module. We tested the keypad and option select modules separately before we instantiated the sine wave generation module.

The column sequencer and keypad decoder was tested using a 7-segment decoder to see whether the keypad buttons were being identified properly. Once the keypad was deemed to be in proper working order we had to ensure that the right package of frequencies was being selected in the option select module. In order to do this, we assigned particular onboard LED pins to the different frequency cases.

The sinewave generation module consisted of the NCO and ADSR modules. The phase accumulator values and its' corresponding sine look-up table amplitude were observed using ModelSim. The states of the ADSR were also checked using in ModelSim using the '$display' function to view the value held by the registers in question.

Conclusion

We were successful in the goals we set for ourselves at the start of this project. From the start, we wanted to use our digital synthesizer to generate different tones. These tones would include a summation of fundamental and harmonic frequencies. As well an ADSR (attack-decay-sustain-release) profile was applied to the generated frequencies before they were outputted to a speaker.

If we had more time we would have liked to improve how the ADSR profile is applied to the generated frequencies. It was hard to discern a difference between different scalar settings in the ADSR profile. So, we would have like to find a way to make the effect of ADSR profile more pronounced. Another thing we would have like to have added would be more ADSR profiles and have them be able to be applied to the generated frequencies on the fly.

Code Modules

Top Level Module – SynthTop.sv

```
/*
File Name: SynthTop.sv
Author: Robert Third & Zawaad Sobhan & Ed Casas
Date: Apr 11th 2017

This is the top level module for the digital synthesiser project.
It includes the pll code generating the 25kHz clock and the instantiations
for the different modules.
*/

// Structure to hold harmonic and fundamental frequencies
typedef struct{
        logic [15:0] fund;
        logic [15:0] har1;
        logic [15:0] har2;} freqs;

// Top Level module for our project
module lab2 (
        input logic CLOCK_50,
        input logic reset_n,
        (* altera_attribute = "-name WEAK_PULL_UP_RESISTOR ON" *)
    input logic  [3:0] kpr,  // rows, active-low w/ pull-ups

        output logic [3:0] kpc,  // column select, active-low
    output logic [3:0] ct,   // " digit enables
        output logic spkr,              // output to speaker
) ;

    logic clk ;               // 25kHz clock for keypad scanning
    logic kphit ;           // a key is pressed
    logic [3:0] num ;         // value of pressed key

        freqs deltaN;                  // structure of frequencies
    assign ct = { {3{1'b0}}, kphit } ;
    pll pll0 ( .inclk0(CLOCK_50), .c0(clk) ) ;

   // Instantiation of the modules
        colseq colseq_0 (.*);
        kpdecode kpdecode_0 (.*);
        optionselect optionselect_0 (.*);
        sinewave sinewave_0 (.*);

endmodule


// Module to generate the 25kHz clock from the 50MHz main clock
module pll ( inclk0, c0);

        input     inclk0;
        output    c0;

        wire [0:0] sub_wire2 = 1'h0;
        wire [4:0] sub_wire3;
        wire  sub_wire0 = inclk0;
        wire [1:0] sub_wire1 = {sub_wire2, sub_wire0};
        wire [0:0] sub_wire4 = sub_wire3[0:0];
        wire  c0 = sub_wire4;

        altpll altpll_component ( .inclk (sub_wire1), .clk
          (sub_wire3), .activeclock (), .areset (1'b0), .clkbad
          (), .clkena ({6{1'b1}}), .clkloss (), .clkswitch
          (1'b0), .configupdate (1'b0), .enable0 (), .enable1 (),
          .extclk (), .extclkena ({4{1'b1}}), .fbin (1'b1),
          .fbmimicbidir (), .fbout (), .fref (), .icdrclk (),
          .locked (), .pfdena (1'b1), .phasecounterselect
          ({4{1'b1}}), .phasedone (), .phasestep (1'b1),
          .phaseupdown (1'b1), .pllena (1'b1), .scanaclr (1'b0),
            .scanclk (1'b0), .scanclkena (1'b1), .scandata (1'b0),
            .scandataout (), .scandone (), .scanread (1'b0),
            .scanwrite (1'b0), .sclkout0 (), .sclkout1 (),
            .vcooverrange (), .vcounderrange ());

        defparam
                altpll_component.bandwidth_type = "AUTO",
                altpll_component.clk0_divide_by = 2000,
                altpll_component.clk0_duty_cycle = 50,
                altpll_component.clk0_multiply_by = 1,
                altpll_component.clk0_phase_shift = "0",
                altpll_component.compensate_clock = "CLK0",
                altpll_component.inclk0_input_frequency = 20000,
                altpll_component.intended_device_family = "Cyclone IV E",
                altpll_component.lpm_hint = "CBX_MODULE_PREFIX=lab1clk",
                altpll_component.lpm_type = "altpll",
                altpll_component.operation_mode = "NORMAL",
                altpll_component.pll_type = "AUTO",
                altpll_component.port_activeclock = "PORT_UNUSED",
                altpll_component.port_areset = "PORT_UNUSED",
                altpll_component.port_clkbad0 = "PORT_UNUSED",
                altpll_component.port_clkbad1 = "PORT_UNUSED",
                altpll_component.port_clkloss = "PORT_UNUSED",
                altpll_component.port_clkswitch = "PORT_UNUSED",
                altpll_component.port_configupdate = "PORT_UNUSED",
                altpll_component.port_fbin = "PORT_UNUSED",
                altpll_component.port_inclk0 = "PORT_USED",
                altpll_component.port_inclk1 = "PORT_UNUSED",
                altpll_component.port_locked = "PORT_UNUSED",
                altpll_component.port_pfdena = "PORT_UNUSED",
                altpll_component.port_phasecounterselect = "PORT_UNUSED",
                altpll_component.port_phasedone = "PORT_UNUSED",
                altpll_component.port_phasestep = "PORT_UNUSED",
                altpll_component.port_phaseupdown = "PORT_UNUSED",
                altpll_component.port_pllena = "PORT_UNUSED",
                altpll_component.port_scanaclr = "PORT_UNUSED",
                altpll_component.port_scanclk = "PORT_UNUSED",
                altpll_component.port_scanclkena = "PORT_UNUSED",
                altpll_component.port_scandata = "PORT_UNUSED",
                altpll_component.port_scandataout = "PORT_UNUSED",
                altpll_component.port_scandone = "PORT_UNUSED",
                altpll_component.port_scanread = "PORT_UNUSED",
                altpll_component.port_scanwrite = "PORT_UNUSED",
                altpll_component.port_clk0 = "PORT_USED",
                altpll_component.port_clk1 = "PORT_UNUSED",
                altpll_component.port_clk2 = "PORT_UNUSED",
                altpll_component.port_clk3 = "PORT_UNUSED",
                altpll_component.port_clk4 = "PORT_UNUSED",
                altpll_component.port_clk5 = "PORT_UNUSED",
                altpll_component.port_clkena0 = "PORT_UNUSED",
                altpll_component.port_clkena1 = "PORT_UNUSED",
                altpll_component.port_clkena2 = "PORT_UNUSED",
                altpll_component.port_clkena3 = "PORT_UNUSED",
                altpll_component.port_clkena4 = "PORT_UNUSED",
                altpll_component.port_clkena5 = "PORT_UNUSED",
                altpll_component.port_extclk0 = "PORT_UNUSED",
                altpll_component.port_extclk1 = "PORT_UNUSED",
                altpll_component.port_extclk2 = "PORT_UNUSED",
                altpll_component.port_extclk3 = "PORT_UNUSED",
                altpll_component.width_clock = 5;


endmodule
```

Module 1 – Sinewave.sv

```
/*
File Name: sinewave.sv
Author: Robert Third & Zawaad Sobhan
Date: Apr 11th 2017

Sine wave generator using MATLAB generated look up table
*/


// Structure to hold harmonic and fundamental frequencies
typedef struct{
        logic [15:0] fund;
        logic [15:0] har1;
        logic [15:0] har2;} freqs;

module sinewave(
        input logic clk,               // 25kHz clock
        input logic kphit,             // if a key has been pressed
        input logic reset_n,   // reset
        input freqs deltaN             // structure of frequencies
        output logic spkr,             // output to the speaker

);

        freqs phaseacc;                        // phase accumaltor struct
        logic [15:0] sintable [0:255];
        logic [15:0] sinout;
        logic [15:0] sinoutreal;
        int i;

        // Variables used in ADSR
        logic [2:0] state;
        logic [2:0] next_state;

        logic [2:0] idle_state = 3'b000;
        logic [2:0] att_state = 3'b001;
        logic [2:0] dec_state =3'b010;
        logic [2:0] rel_state =3'b011;
        logic [2:0] sus_state =3'b100;

        logic [15:0] att_scalar = 16'd125;
        logic [15:0] rel_scalar = 16'd63;
        logic [15:0] dec_scalar = 16'd63;
        logic [15:0] scalar;

        logic [15:0] sus_level = 16'd128;

        initial begin
                $readmemh("sintable.tv", sintable); // read sine amplitude look-up table
                phaseacc.fund = 16'b0;
                phaseacc.har1 = 16'b0;
                phaseacc.har2 = 16'b0;
                state = idle_state;
                next_state = idle_state;
                sus_level = 16'd64;
        end


        // ADSR Section
        always @(posedge clk) begin


                        unique case(state)
                        idle_state: begin
                                if (kphit == 1)         begin
                                        next_state <= att_state;
                                        scalar <= 16'd255;
                                end
                                else
                                        next_state <= idle_state;
                        end
                        att_state: begin
                                if(scalar == 0)
                                        next_state <= dec_state;
                                else if (kphit == 0)
                                        next_state <= rel_state;
                                else
                                        next_state <= att_state;
                        end

                        dec_state: begin
                                if(sinout <= sus_level)
                                        next_state <= sus_state;
                                else if (kphit == 0)
                                        next_state <= rel_state;
                                else
                                        next_state <= dec_state;
                        end

                        sus_state: begin
                                if (kphit == 0)
                                        next_state <= rel_state;
                                else
                                        next_state <= sus_state;
                        end

                        rel_state: begin
                                if(sinout <= 0)
                                        next_state <= idle_state;
                                else if (kphit == 1) begin
                                        next_state <= att_state;
                                        scalar <= 16'd255;
                                end
                                else
                                        next_state <= rel_state;
                        end
                        endcase

                        unique case(state)

                        idle_state: begin
                                scalar <= 16'b0;
                        end

                        att_state: begin
                                scalar <= scalar - att_scalar;
                        end

                        dec_state: begin
                                scalar <= scalar + dec_scalar;

                                if (sinout <= sus_level)
                                        scalar <= 16'b0;
                        end

                        sus_state: begin
                                scalar <= 16'b0;
                        end

                        rel_state: begin
                                if (sinout <= 0)
                                        scalar <= 0;
                                else
                                        scalar <= rel_scalar;

                        end
                        endcase

                        state <= next_state;
        end
```

```
	// Phase accumaltors and pulse width modulation for the speaker
	// output.

	always @(posedge clk) begin

		phaseacc.fund = (phaseacc.fund + deltaN.fund);
		phaseacc.har1 = (phaseacc.har1 + deltaN.har1);
		phaseacc.har2 = (phaseacc.har2 + deltaN.har2);

		// resets the phase accumalators when they overflow
		if(phaseacc.fund >= 256) begin
			phaseacc.fund = 0;
		end

		if(phaseacc.har1 >= 256) begin
			phaseacc.har1 = 0;
		end

		if(phaseacc.har2 >= 256) begin
			phaseacc.har2 = 0;
		end

		// fundamental frequency plus 50% of each harmonic
		sinout = (sintable[phaseacc.fund] + (sintable[phaseacc.har1] >> 1) +
(sintable[phaseacc.har2] >> 1));

		sinoutreal = sinout - scalar;

		// Pulse Width Modulation
		for (i = 0; i <= 255; i++) begin
			if(i <= sinoutreal) begin
				spkr = 1;
			end

			else begin
				spkr = 0;
			end
		end
	end
endmodule
```

Module 2 – Colseq.sv

```
// Filename: colseq.sv
// Author: Robert Third & Zawaad Sobhan
// Date: Apr 11th 2017

// This module is used to drive the 4x4 keypad to be used with the kpdecode to
// figure out which keypad key was pressed.

module colseq (
        input logic [3:0] kpr,
        input logic reset_n, clk,
        output logic [3:0] kpc
        );

        logic [2:0] cnt = 0;

        always_ff @(posedge clk) begin
                if ( ! reset_n) begin
                        kpc = 4'b0111;
                        cnt = 0;
                end

                if (kpr == 4'b1111) begin
                        unique case (cnt)
                                0 : begin
                                kpc = 4'b0111;
                                cnt = 1;
                                end
                                1 : begin
                                kpc = 4'b1011;
                                cnt = 2;
                                end
                                2 : begin
                                kpc = 4'b1101;
                                cnt = 3;
                                end
                                3 : begin
                                kpc = 4'b1110;
                                cnt = 0;
                                end
                        endcase
                end
        end
endmodule
```

Module 3 – kpdecode.sv

```
// Filename: kpdecode.sv
// Author: Robert Third & Zawaad Sobhan
// Date: Apr 11th 2017

// This module is used to show which keypad has been pressed.

module kpdecode (
        input logic [3:0] kpc, kpr,
        output logic kphit,
        output logic [3:0] num
        );


        always_comb begin
                num = 4'h0;
                kphit = 0;
                if (kpr != 4'b1111) begin
                        if        ((kpr == 4'b1110) && (kpc == 4'b1110))      num = 4'hd;
                        else if ((kpr == 4'b1110) && (kpc == 4'b1101))       num = 4'hf;
                        else if ((kpr == 4'b1110) && (kpc == 4'b1011))       num = 4'h0;
                        else if ((kpr == 4'b1110) && (kpc == 4'b0111))       num = 4'he;
                        else if ((kpr == 4'b1101) && (kpc == 4'b1110))       num = 4'hc;
                        else if ((kpr == 4'b1101) && (kpc == 4'b1101))       num = 4'h9;
                        else if ((kpr == 4'b1101) && (kpc == 4'b1011))       num = 4'h8;
                        else if ((kpr == 4'b1101) && (kpc == 4'b0111))       num = 4'h7;
                        else if ((kpr == 4'b1011) && (kpc == 4'b1110))       num = 4'hb;
                        else if ((kpr == 4'b1011) && (kpc == 4'b1101))       num = 4'h6;
                        else if ((kpr == 4'b1011) && (kpc == 4'b1011))       num = 4'h5;
                        else if ((kpr == 4'b1011) && (kpc == 4'b0111))       num = 4'h4;
                        else if ((kpr == 4'b0111) && (kpc == 4'b1110))       num = 4'ha;
                        else if ((kpr == 4'b0111) && (kpc == 4'b1101))       num = 4'h3;
                        else if ((kpr == 4'b0111) && (kpc == 4'b1011))       num = 4'h2;
                        else if ((kpr == 4'b0111) && (kpc == 4'b0111))       num = 4'h1;
                        kphit = 1;
                end
        end

endmodule
```

Module 4 – optionselect.sv

```
/*
File Name: optionselect.sv
Author: Robert Third & Zawaad Sobhan
Date: Apr 11th 2017

This module converts the num value generated by decoed7 into the different
combinations of fundamentals and harmonics for sinewave.
*/



// Structure to hold harmonic and fundamental frequencies

typedef struct{
        logic [15:0] fund;
        logic [15:0] har1;
        logic [15:0] har2;} freqs;

module optionselect (
        input logic  [3:0] num,       // Variable for the option from the keypad
        output freqs deltaN    // structure of the frequencies used
        );

        // The different variables used for the frequencies generated.
        logic [15:0] deltaN500;
        logic [15:0] deltaN625;
        logic [15:0] deltaN750;

        logic [15:0] deltaN1000;
        logic [15:0] deltaN1250;
        logic [15:0] deltaN1500;

        logic [15:0] deltaN2000;
        logic [15:0] deltaN2250;
        logic [15:0] deltaN2500;
        logic [15:0] fclk;

        initial begin

                fclk = 25*10^3; // 25kHz clock
                // Group 1 freq
                deltaN500 = (500*2^8)/fclk; // fund 1
                deltaN625 = (600*2^8)/fclk; // fund 1 har 1
                deltaN750 = (700*2^8)/fclk; // fund 1 har 2
                // Group 2 freq
                deltaN1000 = (1000*2^8)/fclk; // fund 2
                deltaN1250 = (1250*2^8)/fclk; // fund 2 har 1
                deltaN1500 = (1500*2^8)/fclk; // fund 2 har 2
                // Group 3 freq
                deltaN2000 = (2000*2^8)/fclk; // fund 3
                deltaN2250 = (2250*2^8)/fclk; // fund 3 har 1
                deltaN2500 = (2500*2^8)/fclk; // fund 3 har 2

        end

        // The case chooses which frequency combination is mapped to each key.
        always_comb begin

                        unique case(num)

                        4'h0 : begin
                                deltaN.fund = 0;
                                deltaN.har1 = 0;
                                deltaN.har2 = 0;
                        end

                        4'h1: begin
                                deltaN.fund = deltaN500;
                                deltaN.har1 = 0;
                                deltaN.har2 = 0;


                        end
                        4'h2: begin

                                deltaN.fund = deltaN500;
                                deltaN.har1 = deltaN625;
                                deltaN.har2 = deltaN750;

                        end
                        4'h3: begin

                                deltaN.fund = deltaN500;
                                deltaN.har1 = deltaN1250;
                                deltaN.har2 = deltaN1500;

                        end
                        4'h4: begin

                                deltaN.fund = deltaN1000;
                                deltaN.har1 = 0;
                                deltaN.har2 = 0;

                        end
                        4'h5: begin

                                deltaN.fund = deltaN1000;
                                deltaN.har1 = deltaN1250;
                                deltaN.har2 = deltaN1500;

                        end
                        4'h6: begin

                                deltaN.fund = deltaN1000;
                                deltaN.har1 = deltaN625;
                                deltaN.har2 = deltaN750;

                        end
                        4'h7: begin

                                deltaN.fund = deltaN2000;
                                deltaN.har1 = 0;
                                deltaN.har2 = 0;

                        end
                        4'h8: begin

                                deltaN.fund = deltaN2000;
                                deltaN.har1 = deltaN2225;
                                deltaN.har2 = deltaN2500;


                        end
                        4'h9: begin

                                deltaN.fund = deltaN2000;
                                deltaN.har1 = deltaN1250;
                                deltaN.har2 = deltaN1500;

                        end

                        4'ha: begin

                                deltaN.fund = deltaN500;
                                deltaN.har1 = deltaN2250;
                                deltaN.har2 = deltaN2500;

                        end
```

```
                        4'hb: begin

                                deltaN.fund = deltaN1000;
                                deltaN.har1 = deltaN2250;
                                deltaN.har2 = deltaN2500;

                        end

                        4'hc: begin

                                deltaN.fund = deltaN2000;
                                deltaN.har1 = deltaN1250;
                                deltaN.har2 = deltaN1500;

                        end
                        4'hd: begin

                                deltaN.fund = deltaN2000;
                                deltaN.har1 = deltaN500;
                                deltaN.har2 = deltaN1500;

                        end

                        4'he: begin

                                deltaN.fund = deltaN500;
                                deltaN.har1 = deltaN1000;
                                deltaN.har2 = deltaN2000;

                        end
                        4'hf: begin

                                deltaN.fund = deltaN1000;
                                deltaN.har1 = deltaN2000;
                                deltaN.har2 = deltaN500;

                        end
                endcase
        end
endmodule
```

MATLAB Code – Generates sintable.tv

```
// Robert Third & Zawaad Sobhan
// Matlab code for generating sintable.tv
clc
clear all
a = 0:2*pi/((2^8)-1):2*pi;
b = 0:1:255;
c = (128*sin(2*pi*b/256))+ 127.49;
fid = fopen('sintable.tv','w');
fprintf(fid,'%x\n', abs(round(c)));
fclose(fid);
```

References

[1]"Sound Envelopes - Teach Me Audio", Teach Me Audio, 2017. [Online]. Available: https://www.teachmeaudio.com/recording/sound-reproduction/sound-envelopes/. [Accessed: 14-Apr- 2017].