Audio Synthesis Using FPGA

Bachelor of Technology

Submitted by

Akshat Naudiyal (2010110061)
Ananyaa Juyal (2010110793)
Lakshay Mangla (201011000)
Suresh Kaistha (2010110666)

Under supervision

of

Dr. Amitabh Chaterjee
Department of Electrical Engineering

SHIV NADAR
INSTITUTION OF EMINENCE DEEMED TO BE
UNIVERSITY
DELHI NCR

SCHOOL OF ENGINEERING

Introduction:

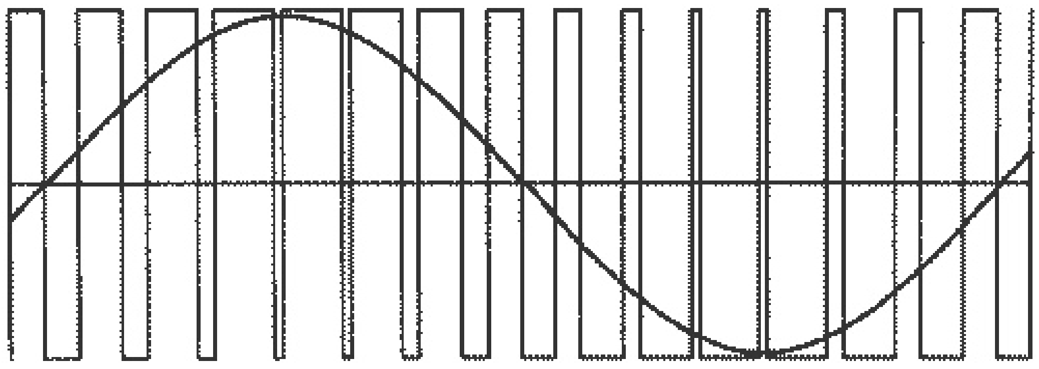
The key goal of this project is to design a digital audio synthesizer on the Nexys A-7 FPGA board. We have used wavetable synthesis to create a 2-octave digital synth covering the notes E3 to E5 in the key of C major (only the white keys). This report walks the reader through the design process, the theory behind a wavetable synthesizer and the RTL code we wrote to implement it on the hardware level.

Wavetable Audio Synthesis Basics:

Sound, the core element of our project, is aptly represented through various waveforms—sine, square, triangle, and sawtooth. These waveforms serve as the building blocks of our audio synthesis, influencing the timbre and character of the generated sound. Wavetable synthesis involves periodic repetition of any such wave to generate sound.

Pulse Width Modulation (PWM):

A common method of digitally representing an analog (audio) signal is through Pulse Width Modulation (PWM). PWM allows us to dynamically control the width of a pulse in a signal, effectively modulating the duty cycle. A larger duty cycle results in a greater amplitude corresponding to the analog signal while a shorter pulse represents a smaller amplitude. Thus, varying pulses at a high frequency, we can effectively represent any analog signal. We use PWM to create a wavetable, and by repeatedly cycling through this wavetable we generate sound. In our case, we cycled through a 32-bit long PWM signal. Therefore, to make a square wave, we wrote a string of sixteen ones followed by sixteen zeros. To represent more complicated waves, we would require a longer bitstream. Figure 1 shows a sinusoidal wave generated using PWM. Since we only had 32 bits available, our approximation of more complex wavetables is not very accurate. Increasing the number of bits and thus, increasing the frequency, we can accurately represent any signal.



To obtain the analog signal from the PWM signal, one needs to pass the bitstream through a low-pass filter. The audio output jack on the Nexys A7 (J8) is driven by a Sallen-Key Butterworth Low-pass 4th Order Filter that provides mono audio output. The signal needs to be driven low for logic ‘0’ and left in high-impedance for logic ‘1’

Clock Divider:

At the heart of precise timing control within our audio synthesizer lies the Clock Divider. This essential component takes an input clock signal and produces an output clock signal at a lower frequency. In our project, the Clock Divider gets an input of 100MHz (generated by the development board) and divides the input clock based on the divisor the user provides. In this project, we used a 32-bit sequence for our PWM. Thus, to generate a 440Hz sound, we needed to divide the clock by:

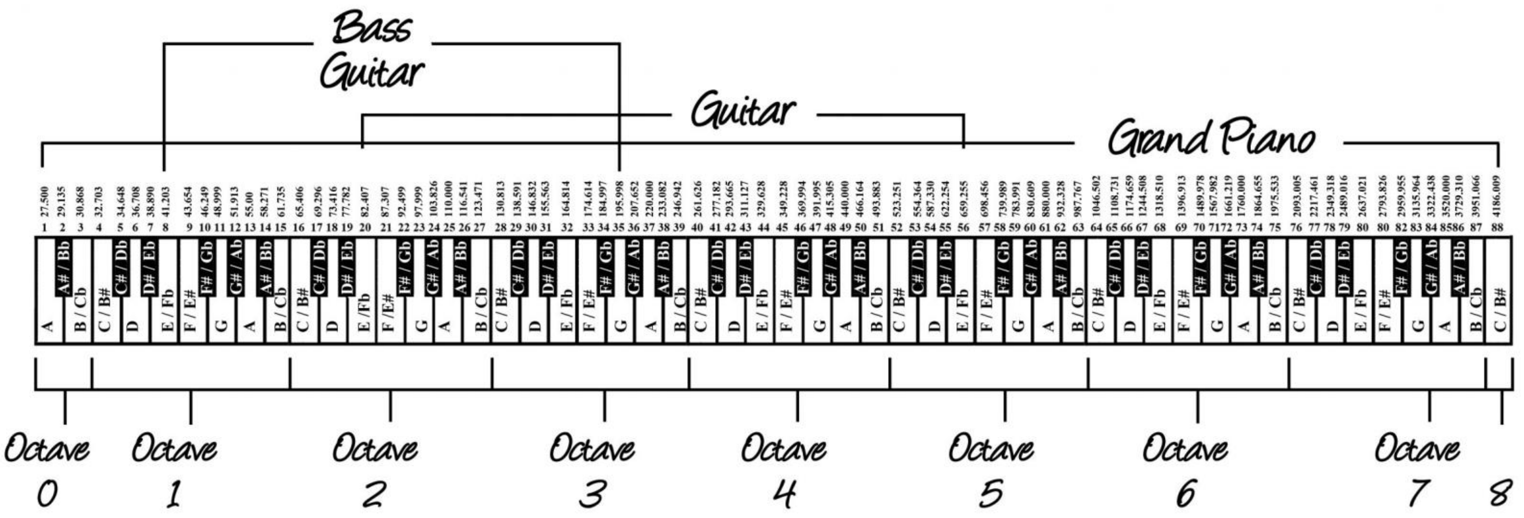
$$Divisor = 100MHz/32*440$$

$$Divisor = 7102$$

To create the entire piano, we referred to frequency corresponding to each note and created a divider based on the required divisor for proper sound synthesis.

Piano Tuning:

Frequencies for equal-tempered scale, A4 = 440 Hz



RTL Code:
The top module incorporates the following modules:

1. **Synthesizer Module:**
    - **Function:** This module contains the code responsible for synthesizing audio, involving switches that correspond to different musical notes.
    - **Purpose:** It enables the generation of diverse tones within the synthesizer.
2. **Button Debounce Modules:**
    - **Function:** These modules serve to debounce the buttons, preventing multiple unintended clicks when a button is pressed. This ensures smooth transitions between different tones of the synthesizer.
    - **Purpose:** Stability in button responses, avoiding erratic behavior.
3. **Clock Divider Module:**
    - **Function:** This module takes the 100MHz clock signal generated by the development board and divides it to a lower frequency within the audible range.
    - **Purpose:** Precise control over timing, ensuring synchronization with the audible output.

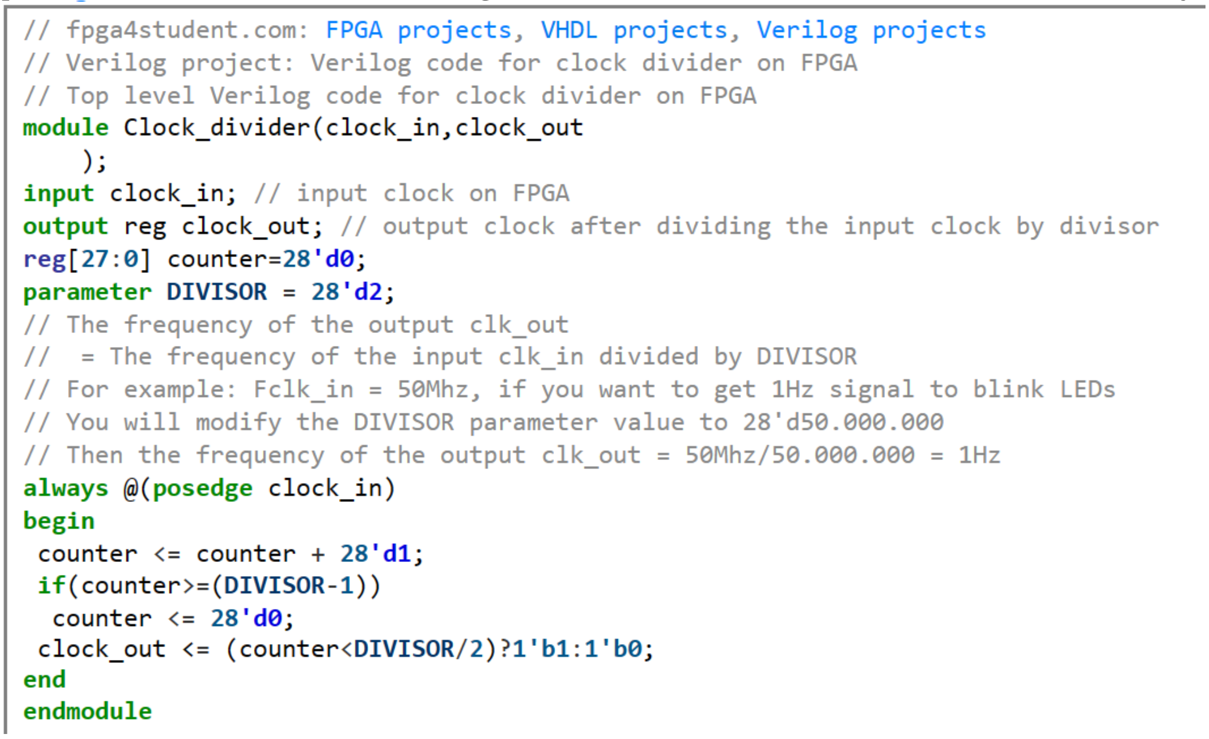
In the top module, we have defined the following inputs and outputs:

- **Inputs:**
    - **CLK (100MHz):** This is the clock signal generated by the development board, which is subsequently divided using the clock divider modules.
    - **Button_Upper:** This button is utilized to change the wavetable used for synthesis, essentially altering the tone of the synthesizer.
- **Outputs:**
    - **3 LED pins:** These pins are employed to visually indicate which wavetable the device is currently playing.
    - **PWM Audio Out bitstream:** This output is directed to output jack 8, where the digital bit stream undergoes low-pass filtering, resulting in the production of sound.

In the following section, we will delve into a detailed description of each module and its corresponding code, shedding light on their individual functionalities and contributions to the overall operation of the FPGA-based audio synthesizer.

Clock Divider Module:

The Clock Divider Module takes a clock signal as an input and allows the user to specify the divisor as a parameter. The output is the divided clock signal. The following code snippet provides an insight into its functionality:



```verilog
// fpga4student.com: FPGA projects, VHDL projects, Verilog projects
// Verilog project: Verilog code for clock divider on FPGA
// Top level Verilog code for clock divider on FPGA
module Clock_divider(clock_in,clock_out
    );
input clock_in; // input clock on FPGA
output reg clock_out; // output clock after dividing the input clock by divisor
reg[27:0] counter=28'd0;
parameter DIVISOR = 28'd2;
// The frequency of the output clk_out
//  = The frequency of the input clk_in divided by DIVISOR
// For example: Fclk_in = 50Mhz, if you want to get 1Hz signal to blink LEDs
// You will modify the DIVISOR parameter value to 28'd50.000.000
// Then the frequency of the output clk_out = 50Mhz/50.000.000 = 1Hz
always @(posedge clock_in)
begin
 counter <= counter + 28'd1;
 if(counter>=(DIVISOR-1))
  counter <= 28'd0;
 clock_out <= (counter<DIVISOR/2)?1'b1:1'b0;
end
endmodule
```

The module employs a 28-bit counter capable of counting up to 2^28. On every positive edge of the input clock, the counter increments until it reaches the value specified by the divisor. At this point, the counter resets to zero. The output clock remains high until the counter is less than divisor/2 and low as long as the counter is greater than or equal to divisor/2. Consequently, the output clock operates at a frequency of F_input_clock / divisor.

Button Debounce Module:

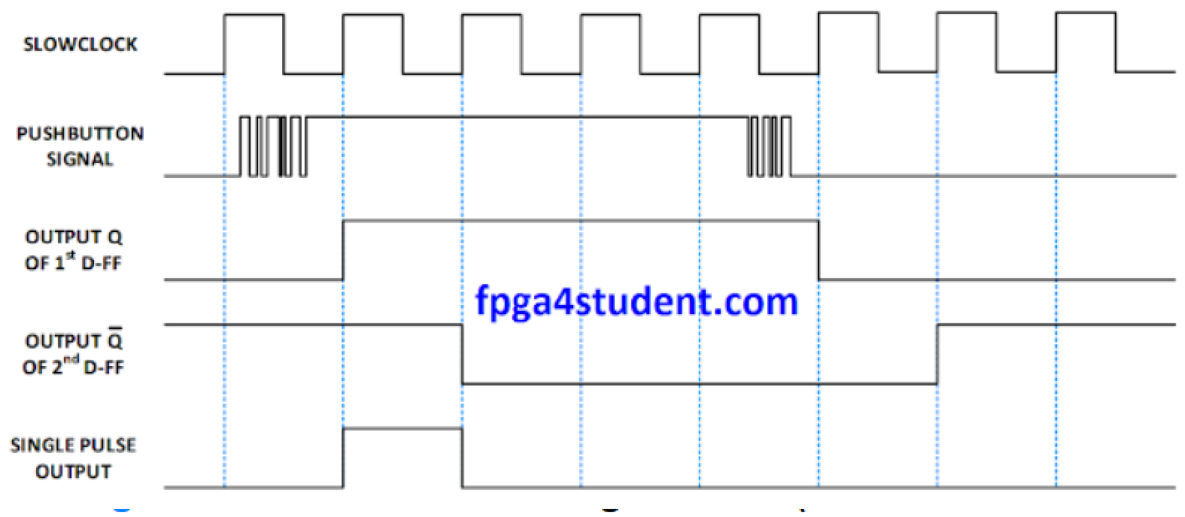
To mitigate the unpredictable bouncing in the signal caused by button presses, we implemented a button debounce code.

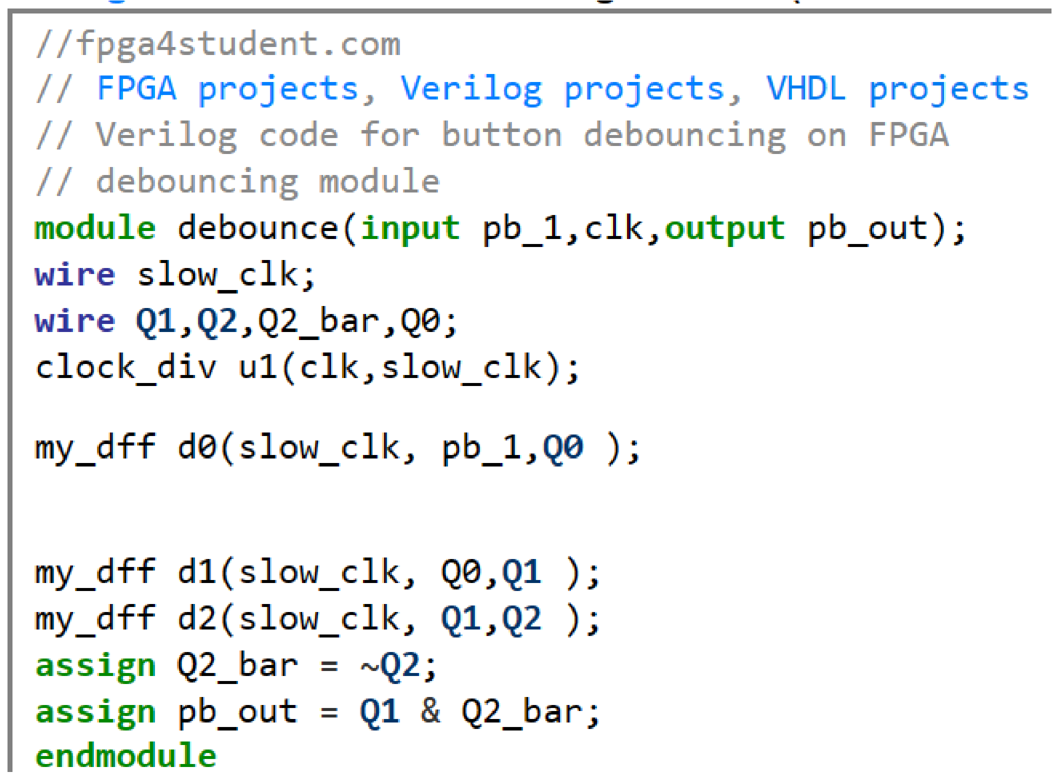
This module instantiates the clock divider module and three D flip-flops. The clock divider divides the input clock by 250,000, yielding a slower clock with a frequency of 100 MHz / 250,000 = 400 Hz.

Here's a breakdown of the debounce process:

1. The first D flip-flop goes high when a positive edge of the slow clock coincides with the button press. At the next positive edge of the slow clock, we obtain Q0.
2. To prevent glitches and metastability, a second D flip-flop is introduced.
3. The third D flip-flop takes the output of the second flip-flop and provides a delayed output.
4. The output of the module is the result of the AND operator between Q1 and the complement of Q2 (denoted as Q2_bar), resulting in a single pulse output.

Below is the code and a timing diagram illustrating the debounce process:





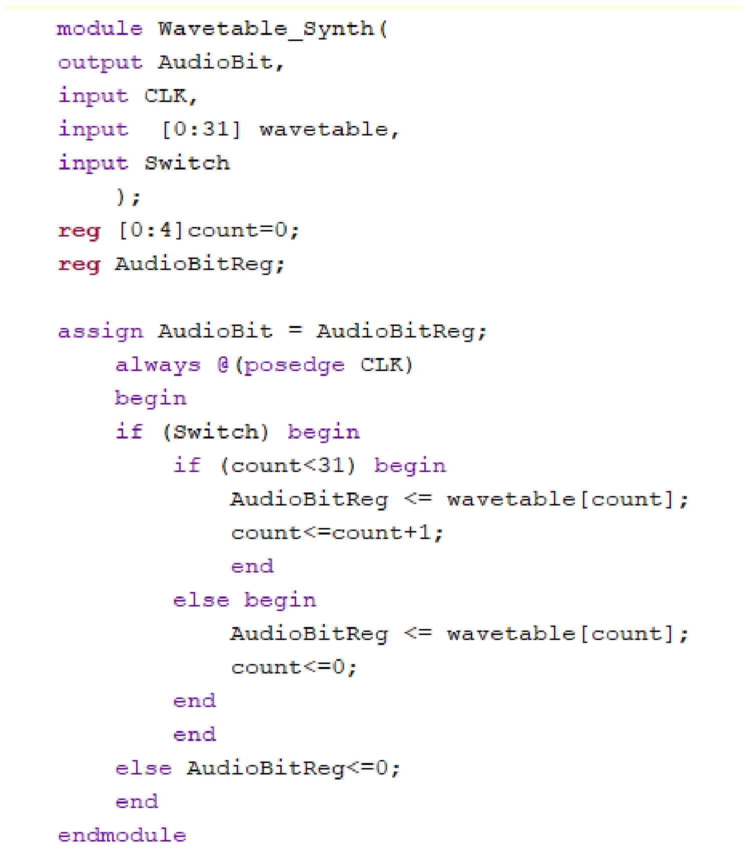
```
//fpga4student.com
// FPGA projects, Verilog projects, VHDL projects
// Verilog code for button debouncing on FPGA
// debouncing module
module debounce(input pb_1,clk,output pb_out);
wire slow_clk;
wire Q1,Q2,Q2_bar,Q0;
clock_div u1(clk,slow_clk);

my_dff d0(slow_clk, pb_1,Q0 );


my_dff d1(slow_clk, Q0,Q1 );
my_dff d2(slow_clk, Q1,Q2 );
assign Q2_bar = ~Q2;
assign pb_out = Q1 & Q2_bar;
endmodule
```

Synthesizer Module:
The synthesizer module is a crucial component of our project, comprising a clock generator block and 15 individual note modules, each corresponding to keys E3 to E5. The clock generator takes the master clock (100MHz) as input and generates a clock array containing 15 distinct clocks, each running at frequencies corresponding to the notes E3 to E5. Each note module invokes the **Wavetable_Synth** module, which forms the basis for our wavetable synthesis.
In the **Wavetable_Synth** module, a wavetable is represented as a register array **reg [31:0] wavetable**, and a clock. On each positive edge of the input clock, this module cyclically outputs bits corresponding to the position of the wavetable register. The cyclic repetition of a specific waveform, achieved through this module, is the essence of wavetable synthesis.

Here is the code snippet for the **Wavetable_Synth** module:



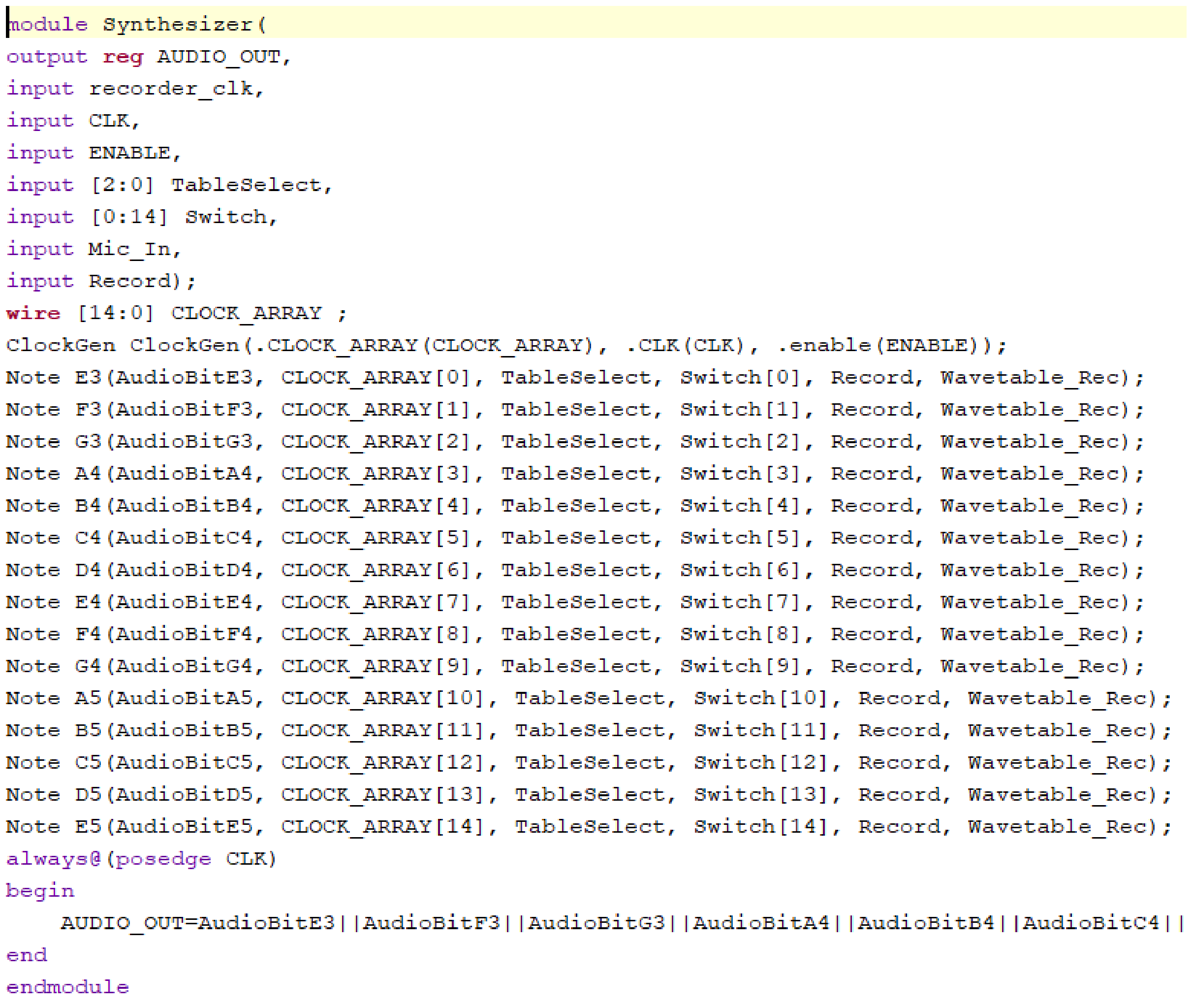
```
module Wavetable_Synth(
output AudioBit,
input CLK,
input  [0:31] wavetable,
input Switch
    );
reg [0:4]count=0;
reg AudioBitReg;

assign AudioBit = AudioBitReg;
    always @(posedge CLK)
    begin
    if (Switch) begin
        if (count<31) begin
            AudioBitReg <= wavetable[count];
            count<=count+1;
            end
        else begin
            AudioBitReg <= wavetable[count];
            count<=0;
        end
        end
    else AudioBitReg<=0;
    end
endmodule
```

Next, the **Note** module selects a particular wavetable based on the clock frequency and the table select register. It takes input from the state of the switch corresponding to that specific note. If the switch is on, audio will play; if it is off, nothing will play. The **Note** module then instantiates the **Wavetable_Synth** module, providing it with the appropriate wavetable.
Here is a snippet of the **Note** module:

```
module Note(
output AudioBit,
input CLK,
input [2:0] TableSelect,
input Press,
input Record,
input [31:0] Rec_WT
    );
    reg [31:0] Wavetable;
always@(posedge CLK)
begin
    if(Record==0) begin
    case (TableSelect)
        'b0 : Wavetable= 'b11111111_00000000_11111111_00000000;
        'b01 : Wavetable='b11110000_00000000_11110000_00000000;
        'b10 : Wavetable='b11110000_00000000_00000000_11110000;
        'b11 : Wavetable='b00010011_01110111_11101110_11001000;
        'b100: Wavetable='b00001010_01010010_10101011_11010101;
        'b101 : Wavetable='b10100000011100001011100001110111;
        'b110 : Wavetable='b01010100000001110001101101001111;
        'b111 : Wavetable='b11010001100100110001010000101100;
        default: Wavetable='bx;
    endcase
    end
    else begin
        Wavetable=Rec_WT;
    end
 end
 Wavetable_Synth synthesizer(AudioBit,CLK,Wavetable,Press);
endmodule
```

Finally, the **Synthesizer** module instantiates various notes based on the clock array and performs a bitwise OR operation on all the note outputs. The final output is directed to the **AUDIO_OUT** bit, which is connected to jack 8.
Here is a snippet of the **Synthesizer** module:



```
module Synthesizer(
output reg AUDIO_OUT,
input recorder_clk,
input CLK,
input ENABLE,
input [2:0] TableSelect,
input [0:14] Switch,
input Mic_In,
input Record);
wire [14:0] CLOCK_ARRAY ;
ClockGen ClockGen(.CLOCK_ARRAY(CLOCK_ARRAY), .CLK(CLK), .enable(ENABLE));
Note E3(AudioBitE3, CLOCK_ARRAY[0], TableSelect, Switch[0], Record, Wavetable_Rec);
Note F3(AudioBitF3, CLOCK_ARRAY[1], TableSelect, Switch[1], Record, Wavetable_Rec);
Note G3(AudioBitG3, CLOCK_ARRAY[2], TableSelect, Switch[2], Record, Wavetable_Rec);
Note A4(AudioBitA4, CLOCK_ARRAY[3], TableSelect, Switch[3], Record, Wavetable_Rec);
Note B4(AudioBitB4, CLOCK_ARRAY[4], TableSelect, Switch[4], Record, Wavetable_Rec);
Note C4(AudioBitC4, CLOCK_ARRAY[5], TableSelect, Switch[5], Record, Wavetable_Rec);
Note D4(AudioBitD4, CLOCK_ARRAY[6], TableSelect, Switch[6], Record, Wavetable_Rec);
Note E4(AudioBitE4, CLOCK_ARRAY[7], TableSelect, Switch[7], Record, Wavetable_Rec);
Note F4(AudioBitF4, CLOCK_ARRAY[8], TableSelect, Switch[8], Record, Wavetable_Rec);
Note G4(AudioBitG4, CLOCK_ARRAY[9], TableSelect, Switch[9], Record, Wavetable_Rec);
Note A5(AudioBitA5, CLOCK_ARRAY[10], TableSelect, Switch[10], Record, Wavetable_Rec);
Note B5(AudioBitB5, CLOCK_ARRAY[11], TableSelect, Switch[11], Record, Wavetable_Rec);
Note C5(AudioBitC5, CLOCK_ARRAY[12], TableSelect, Switch[12], Record, Wavetable_Rec);
Note D5(AudioBitD5, CLOCK_ARRAY[13], TableSelect, Switch[13], Record, Wavetable_Rec);
Note E5(AudioBitE5, CLOCK_ARRAY[14], TableSelect, Switch[14], Record, Wavetable_Rec);
always@(posedge CLK)
begin
    AUDIO_OUT=AudioBitE3||AudioBitF3||AudioBitG3||AudioBitA4||AudioBitB4||AudioBitC4||
end
endmodule
```

Conclusion:

In this project, we have successfully designed and implemented a functional audio synthesizer using FPGA technology. The synthesizer utilizes wavetable synthesis and Pulse Width Modulation (PWM) techniques to generate diverse musical tones, covering a 2-octave range from E3 to E5 in the key of C major. Through our meticulous design process, we have illustrated the theory behind wavetable synthesis and implemented the corresponding RTL code on the Nexys A-7 FPGA board.
Our synthesizer demonstrates the potential for creating a digital musical instrument with a wide range of tones and notes. By leveraging the Clock Divider module, we achieve precise timing control, ensuring accurate and synchronized playback of various musical frequencies. The incorporation of the Button Debounce module enhances the stability of user interactions, preventing unintended multiple clicks and ensuring a smooth transition between different tones.

Future Work:

While we have achieved a functional audio synthesizer, there are several avenues for future improvement and expansion:

1. **Quality of Wavetables:**

- Enhance the quality of wavetables to achieve more accurate and refined musical tones. This could involve exploring advanced wavetable generation algorithms and techniques.

2. **Increasing PWM Frequency:**
    - Elevate the PWM frequency to improve the fidelity of generated sounds. A higher PWM frequency allows for a more detailed representation of complex waveforms, enhancing the overall audio quality.
3. **Polyphony Improvement:**
    - Address the current limitation where playing more than four notes simultaneously results in less melodious output. Investigate and implement advanced signal combination methods to enhance polyphony, providing a more harmonious musical experience.
4. **Custom Wavetable Creation:**
    - Explore the possibility of creating custom wavetables in real-time rather than relying on pre-stored memory. Utilize the recorder available on the development board along with a designed delta-sigma converter to obtain PWM-formatted data. This approach would enable the synthesis of unique and customizable sounds, adding versatility to the synthesizer's capabilities.