Digital System Design Project, 2024, 1–18

Manuscript
Final Report

FINAL REPORT

Fynthesizer: a FPGA Synthesizer

HUANG Tingjun[1,*,†] and CHEN Weiyu[2,*,†]

[1]Information Engineering and [2]Information Engineering

[*]12112619@mail.sustech.edu.cn
[*]12210460@mail.sustech.edu.cn
[†]Contributed equally.

Abstract

This paper presents the design and implementation of Fynthesizer, an FPGA-based synthesizer capable of real-time waveform generation and manipulation. Our core contributions include the development of a versatile synthesizer architecture leveraging oscillators, ADSR envelopes, mixers, and PWM encoders, alongside an intuitive Qt-based GUI for comprehensive control. The system supports up to 12-note polyphony and real-time parameter adjustments, providing a robust platform for audio synthesis. Extensive testing on Ubuntu 20.04 confirms the system's stability and performance. Future work will focus on enhancing sound variety through subtractive synthesis and improving data handling in the MIDI decoder.
Key words: FPGA, Synthesizer, MIDI, Oscillator, ADSR, Polyphony, Mixer, PWM

Contents

Authors' contribution in writing this paper:

HUANG: Sections 1, 2, 3, 4.1, 4.5, 4.6, 6 and 7

CHEN: Sections 4.1.1, 4.2, 4.3, 4.4 and 5

1 Introduction

Digital synthesizers play a crucial role in music production. Digital synthesizers are able to simulate the sound of various instruments by means of digital signal processing without the need for a real instrument. It greatly reduces the cost of producing music for music producers and makes it possible for new sounds to be promoted in musical works.

Implementing digital synthesizers on an FPGA offers significant advantages. Firstly, FPGAs provide a high degree of flexibility and programmability, allowing the customization of synthesizer functions to meet the needs of music production. Secondly, compared to software synthesizers, FPGAs can perform a vast number of parallel processing tasks with lower latency, which is crucial for real-time audio processing.

2 Overview

The physical setup of the experiment is shown in Figure 1. Firstly, you need to connect your MIDI controller to your PC via a MIDI cable. Secondly, connect the power cable (which is also a UART cable) from your PC to the Nexys 4 DDR board (or similar FPGA board). Finally, connect a 3.5mm audio jack from the mono audio out interface on the FPGA board to a pair of speakers.

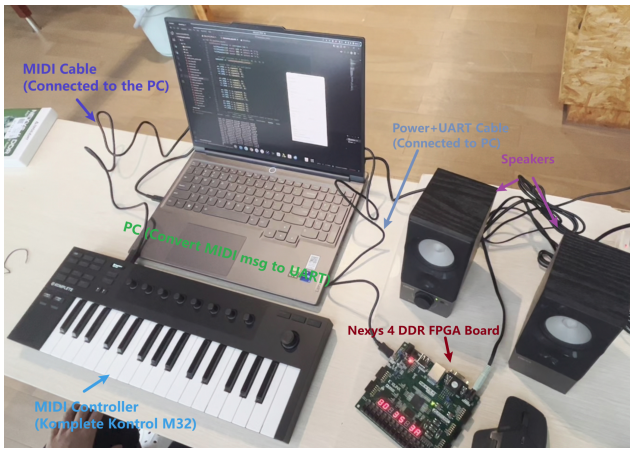
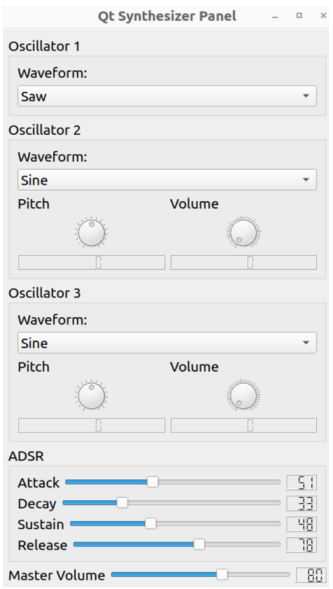


Figure 1. Physical Setup

2.1 System Functions and Specifications

Fynthesizer is an FPGA synthesizer capable of generating new waveforms based on user input and control signals, all formatted according to the MIDI protocol. It utilizes four basic waveforms: sine, triangle, sawtooth, and square. Each new waveform can be synthesized using up to three oscillators, allowing users to adjust the amplitude ratio between the three waveforms. Additionally, the pitch of the two auxiliary waveforms relative to the fundamental frequency can be adjusted. Before output, the new waveform can be further shaped using an ADSR envelope, enabling the emulation of both percussive keyboard instruments and sustained orchestral instruments. Furthermore, our Fynthesizer supports up to 12 notes being played simultaneously, making polyphony possible.

In operation, users can configure the internal synthesizer parameters of the Fynthesizer through a GUI panel written in Qt (`res/QtSynth.py`), as shown in Figure 2a. Once the GUI panel is launched, users can play notes using a MIDI keyboard (as shown in Figure 2b). The master volume control, ADSR, and controls for the two auxiliary oscillators are all bound to the knobs on the keyboard. The GUI panel also displays the control commands sent by the MIDI keyboard in real time, allowing users to see the adjusted parameters and values more intuitively.



(a) GUI Panel of Fynthesizer.

(b) MIDI Instrument

Figure 2. (Left) The GUI panel for configuring the Fynthesizer. The control taken on this panel will be synced to the Fynthesizer via MIDI messages. (Right) MIDI controller we used (Komplete Kontrol M32).

If you do not have a MIDI keyboard, or if the MIDI messages sent by your keyboard are not consistent with ours (Komplete Kontrol M32), you can use the vmpk virtual keyboard (as shown in Figure 3) to test our code. Alternatively, you can use another program we provide, `res/play_midi_file.py`, to simulate instrument playing by directly playing MIDI information from MIDI files.

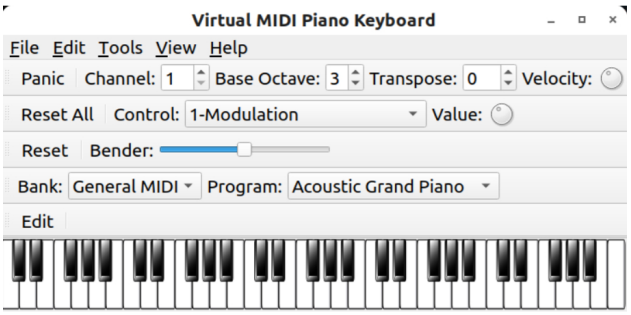


Figure 3. Virtual MIDI Piano Keyboard (vmpk) Interface.

Note that all the above functionalities have been tested on the Ubuntu 20.04 operating system. If your operating system is different from ours, you may need to make some modifications to the Python files in the `res` directory. However, thanks to Python's cross-platform capabilities, the changes required should be minimal.

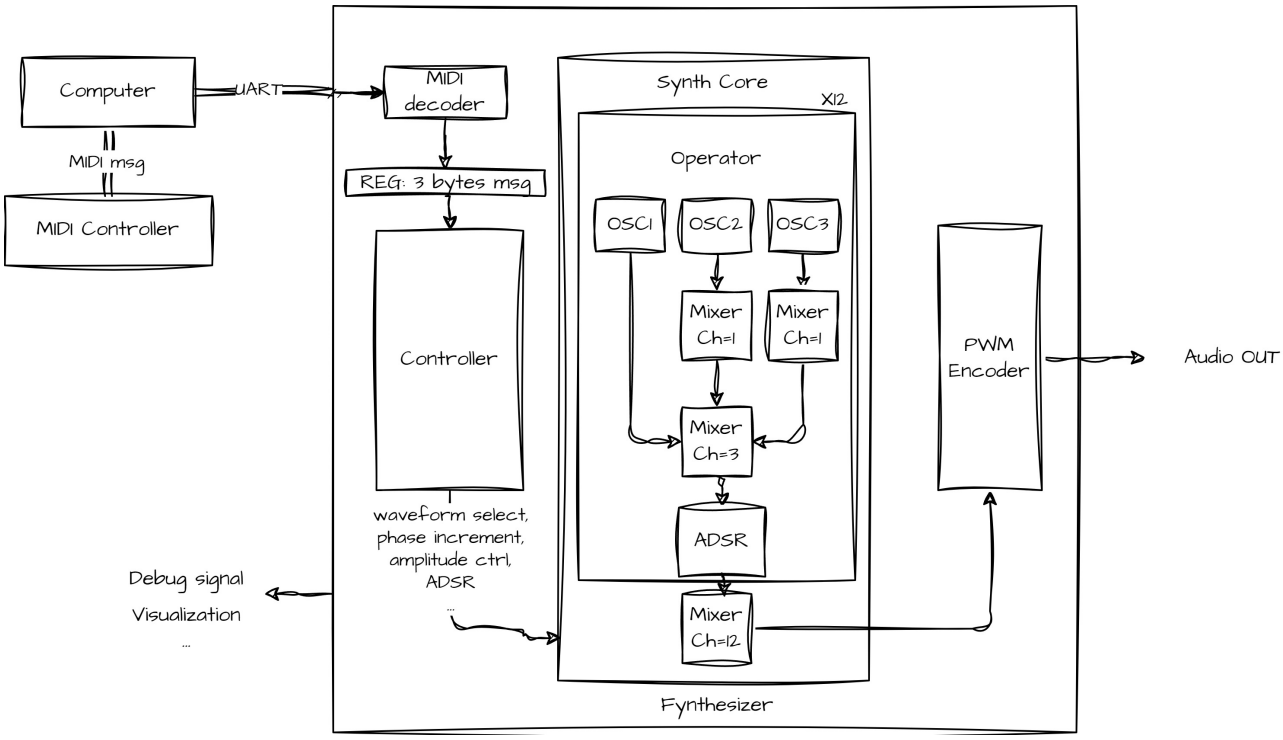


Figure 4. Overall Architecture of Fynthesizer

2.2 Bill of Material

i. Nexys 4 DDR FPGA board
ii. A pair of speakers (Although the FPGA board outputs mono audio, this audio can only be heard from one specific speaker in the pair)
iii. Computer
iv. MIDI keyboard
v. Various cables (MIDI cable, combined power and serial cable for the FPGA board)

2.3 I/O Ports

As shown in the top module entity declaration below,

```
entity Fynthesizer3x is
    port (
        clk : in std_logic;
        rst : in std_logic;
        uart_rxd_in : in std_logic;
        anode : out std_logic_vector(7 downto 0);
        cathode : out std_logic_vector(6 downto
            0);
        volume_visualizer_out : out
            std_logic_vector(15 downto 0);
        pwm_out : out std_logic;
        pwm_sd : out std_logic
    );
end Fynthesizer3x;
```

apart from the clock and reset button, the I/O ports of the Fynthesizer include:

i. `uart_rxd_in`: Used to receive serial signals from the computer.
ii. `anode` and `cathode`: Used to drive the seven-segment display, showing the currently received MIDI message.
iii. `volume_visualizer_out`: Used to drive 16 LEDs, indicating the amplitude of the current output audio signal by the number of lit LEDs.
iv. `pwm_out` and `pwm_sd`: Used to drive the audio output ports on the board.

Among these, driving the UART and audio output ports are the two primary peripherals chosen for this project, meeting the project requirement of driving two peripherals (or pmod).

2.4 Architecture

As shown in Figure 4, the overall architecture of the Fynthesizer is as follows:

First, the MIDI controller connects to the computer via USB MIDI, transmitting MIDI messages. The computer runs our QtSynth Panel, a Python-based program that converts all received and generated MIDI information (from the MIDI controller, panel knobs, vmpk, and MIDI files) into serial data packets. These packets are then sent to the FPGA board via UART, with all MIDI messages defined as three bytes.

On the FPGA board, a MIDI decoder module receives the UART-encoded MIDI messages and stores them in a 3-byte register. The controller reads these decoded MIDI messages and translates them into control signals for the Synth core. The Synth core includes 12 operators and a 12-channel mixer. Each operator features three oscillators, with the amplitude of oscillators 2 and 3 adjustable via a 1-channel mixer. The controller also manages the relative pitch of these oscillators to the fundamental frequency (oscillator 1). The output of the three oscillators is combined through a mixer and shaped using an ADSR envelope.

The outputs from the 12 operators are then mixed into a single

waveform. The value in a special register represents the amplitude of this analog waveform at any given moment. For output, the value in this register is converted into a corresponding duty cycle by a PWM encoder, which then outputs the audio through the board's mono output port.

The architecture for the top module is clearly described in the code. You can find the complete code of the top module in Section 8.1.

3 Schedule and Budget

Our schedule for this project can be concluded in the following timeline:

- May 1 - May 5: Finish the VHDL code for oscillator and ADSR.
- May 6 - May 12: Finish the VHDL code for mixer and PWM encoder.
- May 13 - May 19: Finish the VHDL code for UART on board and communication with PC.
- May 20 - June 2: Finish the GUI on the upper computer and MIDI decoder on board.

Since we have already got a MIDI controller and a pair of speakers, the budget for this project is 0.

4 Design

Since this is a Digital System Design course project, we only involve the detailed design of the VHDL part in this section.

4.1 MIDI Decoder

Please refer to the complete code of MIDI Decoder in Section 8.2.

The MIDI decoder operates on a relatively simple principle, with its I/O port defined as follows:

```
entity midi_decoder is
    Port (
        clk : in STD_LOGIC;
        reset : in STD_LOGIC;
        uart_rxd : in STD_LOGIC;
        midi_data : out STD_LOGIC_VECTOR(23
            downto 0);
        data_ready : out STD_LOGIC
    );
end midi_decoder;
```

The MIDI messages we can decode are listed in Table 1

First Byte	Second Byte	Third Byte
Note on (0x90)	Note name (0x15-0x6C)	Velocity (0x00-0x7F)
Note off (0x80)	Note name (0x15-0x6C)	Velocity (0x00-0x7F)
Control change (0xB0)	Control byte (e.g. 0x01)	Value (0x00-0x7F)

Table 1. Types of MIDI messages

Specifically, the detailed control change messages are shown in Table 2.

To obtain the MIDI message from the UART interface, we need to instantiate a UART receiving module, which requires configuration

Control byte	Description
0x01	Change master volume
0x0E	Attack
0x0F	Decay
0x10	Sustain
0x11	Release
0x12	Note offset for second oscillators
0x13	Amplitude for second oscillators
0x14	Note offset for third oscillators
0x15	Amplitude for third oscillators
0x16	Waveform for base oscillators
0x17	Waveform for second oscillators
0x18	Waveform for third oscillators

Table 2. Control Change Cases

of the clock frequency (onboard clock frequency of 100 MHz) and the desired baud rate (460800). A counter is used to track the position of the current received byte. The received bytes are sequentially placed into the `midi_data` output port. Upon receiving the third byte of a MIDI message, the `data_ready` output port will produce a high signal for one clock cycle, indicating the completion of a MIDI message reception.

4.1.1 UART Receiver

The usage logic of the UART RX module, especially the state machine, is explained in detail below based on the provided VHDL code:

The UART RX module operates by receiving serial data via the `UART_RXD` input and outputting the received data through `DOUT` along with various status signals. It employs a state machine to handle different stages of the data reception process.

Clock Divider and Enable Signal:

A clock divider is instantiated to generate a clock enable signal (`rx_clk_en`) for the UART receiver, which oversamples the incoming data by a factor of 16.

Bit Counter:

The bit counter (`rx_bit_count`) tracks the number of bits received. It increments with each clock enable pulse during the `databits` state and resets after receiving eight bits (one byte).

Data Shift Register:

The received bits are sequentially loaded into a shift register (`rx_data`) during the `databits` state. This register holds the byte of data being received.

Parity Check:

If parity checking is enabled (configured via the `PARITY_BIT` generic), a parity bit is generated and compared with the received parity bit to detect errors. The result is stored in `rx_parity_error`.

Output Register:

Once a complete byte is received, the `rx_done` signal indicates completion. If no errors are detected, `DOUT_VLD` is asserted, indicating valid data. Errors are signaled through `FRAME_ERROR` and `PARITY_ERROR`.

Finite State Machine with Datapath (FSMD):

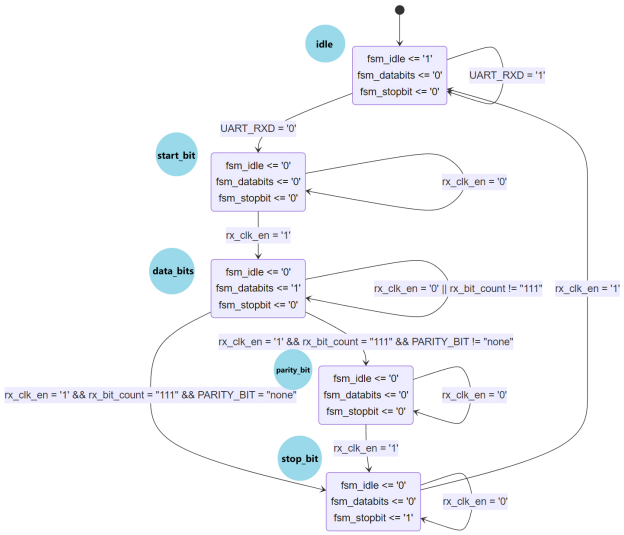


Figure 5. ASM Chart for the UART receiver

The FSMD controls the state transitions and the logic of the UART receiver. It operates in five states:

1. `idle`: The receiver waits for the start bit (a low signal) on `UART_RXD`. Once detected, it transitions to the `startbit` state.

2. `startbit`: The FSM waits for one clock enable pulse to confirm the start bit and then moves to the `databits` state.

3. `databits`: The FSM enables the bit counter and shift register to receive eight bits of data. If parity is enabled, it transitions to the `paritybit` state; otherwise, it moves directly to the `stopbit` state after receiving eight bits.

4. `paritybit`: The FSM checks the parity bit. Once checked, it transitions to the `stopbit` state.

5. `stopbit`: The FSM checks the stop bit (expected to be high). If valid, it signals data reception completion and returns to the `idle` state. If invalid, a frame error is flagged.

FSMD Implementation:

This code snippet demonstrates the FSMD's transition logic, ensuring accurate reception and error detection of UART data. The ASM chart for the FSMD is shown in Figure 5.

4.2 Oscillator

Please refer to the complete code of Oscillator in Section 8.3

Entity Definition

```
entity oscillator is
Port (
    clk: in std_logic;
    reset: in std_logic;
    waveSel: in std_logic_vector(1 downto 0);
    phaseInc: in unsigned(15 downto 0);
    nextSample: in std_logic;
    wavformOut: out signed(15 downto 0)
);
end oscillator;
```

An oscillator is a signal generator that generates triangular, sawtooth, square, sine and other more complex waveforms. The frequency generated is determined by a phase accumulator that varies the step size of the system. Sine or other complex analogue waveforms are stored as 16-bit samples in a 2048 address lookup table. A counter steps through the lookup table based on the phase value. A triangle waveform is simply the output of an address counter for half a cycle and is negated for the other half. A saw wave is simply the output of the address counter. A square wave outputs the minimum value of a signed 16-bit signal when the counter is below 1024 and the maximum value when the counter is above 1024.

The simulation result of four basic signals is below:

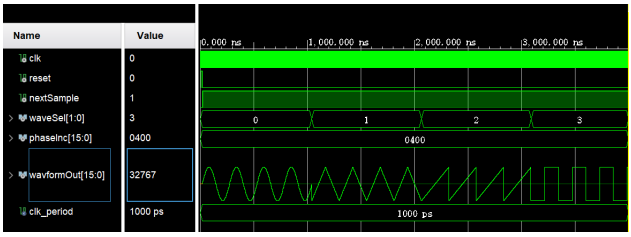


Figure 6. basic waveform

4.3 Amplitude Envelope (ADSR)

Please refer to the complete code of ADSR in Section 8.4

Entity Definition

```
entity adsr is
    generic (
        DATA_WIDTH: integer := 16
    );
    port (
        clk: in std_logic;
        reset: in std_logic;
        en: in std_logic;
        nextSample: in std_logic;
        -- ADRS, all timings in 8bit
        attack: in signed(7 downto 0);  -- time
            to reach max amplitude
        decay: in signed(7 downto 0);   -- time
            to reach sustain amplitude
        sustain: in signed(7 downto 0); --
            sustain amplitude
        rel: in signed(7 downto 0);    -- time to
             reach 0 amplitude
        signalIn: in signed(DATA_WIDTH-1 downto
            0);
        signalOut: out signed(DATA_WIDTH-1 downto
             0)
    );
end adsr;
```

An amplitude envelope is historically 4 variable system that dictates the amplitude of something over time. Within the synth, it was implemented with attack, decay, sustain, and release (ADSR). Attack, decay, and sustain are timing values, while sustain is an amplitude value. Attack corresponds to the amount of time it takes

for the signal to read full amplitude, decay to the amount of time it takes for the signal to go from full amplitude to the level dictated by sustain, and release is the amount of time it takes to bring the amplitude to zero from the level dictated by sustain.

It was implemented as a state machine with four states: **attack, decay, sustain and release**.When being enabled, it would start the attack and continue through to decay, and then sustain. It would stay in sustain until it was no longer enabled, at which it would go to release. If during release, it was enabled again, it would jump to attack. If during attack, decay, or sustain it was disabled, it would move to release.

The ASM chart and conceptual diagram of the state machine is shown below:

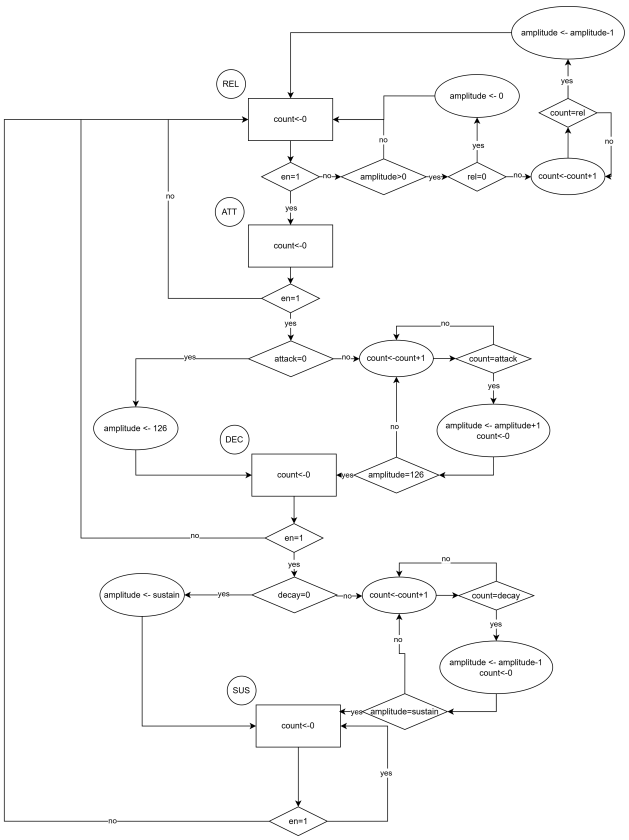


Figure 7. ASM chart of ADSR

The following figure shows a simulation of an adsr process (assuming a constant input signal).

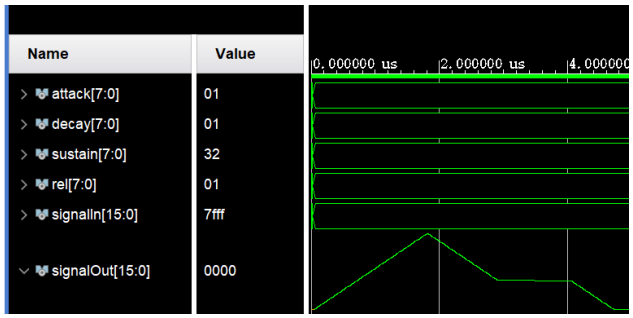


Figure 8. ADSR

4.4 Mixer

Please refer to the complete code of Mixer in Section 8.5.

The mixer within a digital synthesizer serves a critical function in the audio synthesis process. Its primary role is to amalgamate the various note waveforms, which are played simultaneously by different fingers, into a single cohesive waveform. This is essential in creating a unified sound output despite the complexity of individual inputs.

Entity Definition

The mixer entity is defined with generic parameters for the number of active channels and the data width of the signals. It includes ports for the clock, reset, amplitude control, input channels, and the output signal.

```
entity mixer is
    generic (
        ACTIVE_CHANNELS : integer := 10;
        DATA_WIDTH: integer := 16
    );
    port (
        clk : in std_logic;
        reset : in std_logic;
        amplitude: in signed(7 downto 0);
        ch1, ch2, ch3, ch4,
        ch5, ch6, ch7, ch8,
        ch9, ch10, ch11, ch12 : in signed(
            DATA_WIDTH-1 downto 0);
        sigOut : out signed(DATA_WIDTH-1 downto
            0)
    );
end mixer;
```

Log2 Function for Overflow Prevention

A compile-time log2 function `log2ceil` is used to determine the number of extra bits required to represent the combined value of all input channels without overflow. This ensures that the summed signal does not exceed the representable range.

Internal Signal Declarations

Constants `EXTRA_BITS` and `INT_SIG_BITS` are calculated using the log2 function to define the bit width of the internal mixed signal. `mixedSigInt` is a signal with sufficient bit width to hold the summed value of all input channels. `sigOutInt` is used to store the multiplied output before scaling it down to the final output signal width.

Mixing Process

The input channels are resized to the appropriate bit width and summed together to form `mixedSigInt`. This step ensures that no overflow occurs during the summation of the input signals.

Multiplication for Gain Control

A DSP slice is used to multiply the summed signal by the amplitude control signal. This multiplication adjusts the overall gain of the mixed signal. The DSP slice is a specialized hardware component within the FPGA that efficiently handles multiplication operations. It is configured with specific parameters such as the type of FPGA device, the desired latency for the operation, and the widths of the input signals. The result of this multiplication is stored in an intermediate signal before being scaled down to the final output signal width.

Output Signal Scaling

The final step involves scaling down the resultant signal `sigOutInt` to the appropriate data width for the output signal `sigOut`. This scaling ensures that the output signal fits within the specified data width, maintaining the desired volume level and preventing distortion.

By implementing these features, the VHDL code ensures a robust and accurate mixing of audio signals within the FPGA. It carefully handles overflow prevention during the summation of input channels and applies gain control through multiplication, resulting in a high-quality, cohesive audio output.

4.5 Controller

Please refer to the complete code of controller in Section 8.6.

The Controller has two tasks:

1. Dynamically allocate the available operators' resources;

2. Use the decoded MIDI message (note name) to get the corresponding phase increment value.

The types of MIDI messages we can receive are shown in the table 1. The first two types are directly controlled by the controller. Although these control signals are shown coming from the controller in the block diagram (Figure 4), their control is relatively simple, and thus they are directly managed by the top module in the code implementation.

Since each operator executes the note on logic when `en=1` and the note off logic when `en=0`, the controller's first task can be simplified to correctly assigning 0 or 1 to a `std_logic_vector` of length 12 (corresponding to 12 operators). The logic can be described as follows:

1. When `note_on=1`, the highest priority 0 in `en[11 downto 0]` is set to 1, and the corresponding `note_name` is stored in a register (`note_ref`).

2. When `note_off=1`, the controller searches from the highest to the lowest bit in `note_reg` for the specified `note_name` in the current MIDI message. Upon finding a match, the corresponding bit in `en[11 downto 0]` is cleared.

With this logic, we can dynamically allocate operator resources.

Next, we consider the controller's second task: converting the `note_name` stored in `note_reg` to the corresponding phase increment value. The phase increment determines the step size when traversing one cycle in the oscillator. A higher phase increment corresponds to a higher frequency.

In the current experimental setup, the relationship between frequency and phase increment is described by:

$$\begin{aligned} \text{freq} &= \frac{\text{phaseInc}}{2^{18} \times \text{next\_sample\_period}} \\ &= \frac{\text{phaseInc}}{2^{18} \times \text{next\_sample\_counter} \times 10^{-8}(\text{s})} \end{aligned}$$

where `next_sample_counter` is 2^{10}, as the oscillator steps through one sample every 2^{10} system clock cycles (10 ns).

The relationship between frequency and `note_name` is expressed by:

$$\text{freq} = 2^{\frac{\text{note\_name}-69}{12}} \times 440$$

By combining these equations, we can obtain the correct conversion from `note_name` to `phaseInc`.

However, solving these equations directly on FPGA is complex, and a standard MIDI keyboard has only 88 keys. Therefore, we precompute the `phaseInc` values for these 88 keys using a Python script and store them in a lookup table. This approach conserves FPGA resources. Despite being called a lookup table (LUT), the declaration as a constant array results in Vivado synthesizing it as BRAM, not LUT:

```
package phaseInc_package is
    type phaseInc_table is array (0 to 87) of
        unsigned(15 downto 0);
    constant phaseIncs : phaseInc_table := (
        x"0049",
        x"004E",
        x"0052",
        ...
```

4.6 PWM Encoder

Pleaser refer to the complete code of PWM encoder in Section 8.7.

The purpose of the PWM encoder is to convert the amplitude value represented by a register (which can be positive or negative) into a PWM duty cycle and then output the PWM waveform. The input and output ports of the PWM encoder are defined as follows:

```
entity pwm_enc is
    generic (
        pwm_period : UNSIGNED(15 downto 0) := x"
            FFFF" -- Default period of the PWM
            signal in clock cycles
    );
    Port (
        clk : in STD_LOGIC;
        rst : in STD_LOGIC;
        audio_amplitude : in SIGNED(15 downto 0);
             -- Audio amplitude as a signed
            value
```

```
        pwm_out : out STD_LOGIC -- PWM output
            signal
    );
end pwm_enc;
```

For the generic `pwm_period`, we set the value to 4096, which is 2^{12}. This provides a 12-bit resolution, allowing for a more precise representation of the corresponding digital amplitude and reducing quantization errors. However, `pwm_period` should not be too large, as an excessively large period can reduce the audio sampling rate. The sampling rate formula is:

$$\frac{1}{2^N \times \texttt{clk\_period}}$$

where N corresponds to the 12 bits used here.

Additionally, a very large `pwm_period` might cause the PWM frequency to fall within the audible range of the human ear. For example, if the `clk_period` is 10 ns and the resolution is 16 bits, the corresponding PWM frequency would be 1525.87 Hz. In such cases, the speaker would emit a constant noise at 1525.87 Hz, which could overpower other sounds.

To handle potentially negative input amplitudes, we first shift the waveform upwards so that the minimum possible value is 0, then map the amplitude proportionally to a duty cycle between 0 and 100. The formula is:

$$\text{duty cycle} = \frac{\text{amplitude\_lift} \times \text{pwm\_period}}{2^{16}}$$

However, division in VHDL is challenging to implement, so we approximate it by right-shifting the binary number by N bits, effectively dividing by 2^N. The final implemented code is:

```
duty_cycle <= resize((resize(amplitude_unsigned,
    16) * pwm_period) srl 16, 16);
```

Converting the duty cycle to a PWM signal is simpler; it involves comparing the duty cycle value with the current counter value:

```
-- PWM generation logic
if pwm_counter < duty_cycle then
    pwm_out <= '1';
else
    pwm_out <= '0';
end if;
```

5 Extension

5.1 Timbre mimicry

We use python to create a 2048-point lookup table based on the given harmonic coefficients. This lookup table contains sine wave values for the fundamental frequency and harmonics, representing a sample of the waveform over one cycle. This LUT can be set directly to the parameters of the oscillator, thus enabling the imitation of audio. We also read this LUT on the computer, add adsr adjustments and play the sound. This makes it easier to debug the tones so that the data coming into the FPGA can imitate the tones better.

The following figure is the LUT in one period of four kinds of timbre we have imitated.

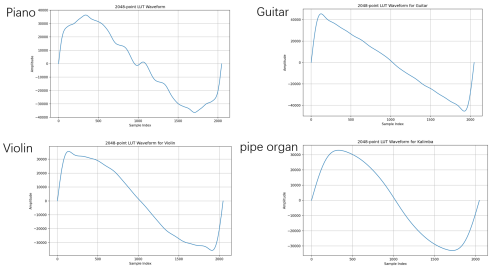


Figure 9. A physical mixer

6 Results

All the functions mentioned above have been completed, as described in the initial system function and specification. Our Fynthesizer can perform all waveform synthesis in real-time, perfectly leveraging the real-time advantages of a hardware synthesizer. There is no overflow during the synthesis process, and the correct waveform timbre is heard during monitoring. Additionally, our attempts to manually simulate instrument sounds were successful. Based on the lookup table (LUT) designed in the extension section, we can generate instrument sounds other than the four basic waveforms, and their listening effects are excellent. The user-friendly GUI design allows us to easily adjust any parameter of the synthesizer and see real-time feedback.

Limitations

Since our synthesizer is purely an "additive synthesizer," the timbres we can create by adjusting parameters are relatively limited. If we add a "subtractive" feature in the future, namely a filter to remove excess harmonics produced by the additive synthesizer (especially those from sawtooth and square waves), we could create softer timbres more akin to real instruments. Additionally, our MIDI decoder module does not implement a buffer mechanism. If a large amount of MIDI information is sent to the FPGA in a short period, it may cause "packet loss," directly leading to "stuck keys." This issue also needs to be addressed in future work.

7 Conclusion and Future Work

The Fynthesizer project successfully demonstrates the potential of FPGA-based real-time audio synthesis. By leveraging hardware components such as oscillators, ADSR envelopes, mixers, and PWM encoders, we have built a robust system capable of producing complex and accurate waveforms. The integration of a user-friendly GUI enhances the overall experience, allowing intuitive control and real-time feedback. Our innovative approach to instrument sound simulation using precomputed lookup tables (LUTs) has also proven effective, showcasing the system's versatility.

Future work will focus on expanding the range of synthesized sounds through the incorporation of subtractive synthesis techniques and enhancing the MIDI decoder with a buffer mechanism to improve data handling and prevent packet loss.

8 Appendix

8.1 Fynthesizer3x.vhd (top module)

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

entity Fynthesizer3x is
    port (
        clk : in std_logic;
        rst : in std_logic;
        uart_rxd_in : in std_logic;
        anode : out std_logic_vector(7 downto 0);
        cathode : out std_logic_vector(6 downto
            0);
        volume_visualizer_out : out
            std_logic_vector(15 downto 0);
        pwm_out : out std_logic;
        pwm_sd : out std_logic
    );
end Fynthesizer3x;

architecture rtl of Fynthesizer3x is
    -- Components declaration
    component midi_decoder is
        Port (
            clk : in STD_LOGIC;
            reset : in STD_LOGIC;
            uart_rxd : in STD_LOGIC;
            midi_data : out STD_LOGIC_VECTOR(23
                downto 0);
            data_ready : out STD_LOGIC
        );
    end component;

    component sevenSegDisplay is
        port (
            clk : in std_logic;
            rst : in std_logic;
            midi_msg : in std_logic_vector(23
                downto 0);
            anode : out std_logic_vector(7 downto
                 0);
            cathode : out std_logic_vector(6
                downto 0)
        );
    end component;

    component note_control3x is
        Port (
            clk : in STD_LOGIC;
            reset : in STD_LOGIC;
            note_on : in STD_LOGIC;
            note_off : in STD_LOGIC;
            note_value : in STD_LOGIC_VECTOR(7
                downto 0);
            note_offset2 : in SIGNED(7 downto 0);
            note_offset3 : in SIGNED(7 downto 0);
            en : out STD_LOGIC_VECTOR(11 downto
                0);
            phaseInc_base : out UNSIGNED(191
                downto 0);
            phaseInc_second : out UNSIGNED(191
                downto 0);
            phaseInc_third : out UNSIGNED(191
                downto 0)
        );
    end component;

    component syn_core3x is
        port (
            -- 100 Mhz System Clock
            clk: in std_logic;
            reset: in std_logic;
            -- 12 16 bit signals LSB = op1
            opPhase1: in unsigned(191 downto 0);
            opPhase2 : in unsigned(191 downto 0);
            opPhase3 : in unsigned(191 downto 0);

            ampl2 : in signed(7 downto 0);
            ampl3 : in signed(7 downto 0);
            -- one hot encoding note on/off
            opEnable: in std_logic_vector(11
                downto 0);
            -- ADSR for the operators
            att, dec, sus, rel: in signed(7
                downto 0);
            -- Master amplitude
            ampl_master: in signed(7 downto 0);
            -- DAC Next Sample
            nextSample: in std_logic;
            -- 16 bit audio data
            audioOut: out signed(15 downto 0);
            waveSel: in std_logic_vector(5 downto
                 0)
        );
    end component;

    component pwm_enc is
        generic(
            pwm_period : UNSIGNED(15 downto 0) :=
                 x"FFFF"
        );
        port(
            clk: in std_logic;
            rst : in std_logic;
            audio_amplitude : in signed(15 downto
                 0);
            pwm_out : out std_logic
        );
    end component;

    component audio_level_meter is
        generic (
            DATA_WIDTH : integer := 16
        );
        port (
            audio_in : in signed(DATA_WIDTH-1
                downto 0);
            level_out : out std_logic_vector(
                DATA_WIDTH-1 downto 0)
        );
    end component;
```

```
    -- Internal signals
    signal midi_data : std_logic_vector(23 downto
         0);
    signal note_on : std_logic;
    signal note_off : std_logic;
    signal note_offset2 : signed(7 downto 0);
    signal note_offset3 : signed(7 downto 0);
    signal note_value : std_logic_vector(7 downto
         0);
    signal en_reg : std_logic_vector(11 downto 0)
        ;
    signal phaseInc_base_reg : unsigned(191
        downto 0);
    signal phaseInc_second_reg : unsigned(191
        downto 0);
    signal phaseInc_third_reg : unsigned(191
        downto 0);
    signal ampl_second : signed(7 downto 0);
    signal ampl_third : signed(7 downto 0);
    signal nextSample : std_logic;
    signal counter: integer range 0 to 2**10-1 :=
         0;
    signal audio : signed(15 downto 0);
    signal midi_received : std_logic;
    signal att, dec, rel : signed(7 downto 0) :=
        (to_signed(0,8));
    signal sus, master_volume : signed(7 downto
        0) := (to_signed(90,8));
    signal wave_sel : std_logic_vector(5 downto
        0) := (others => '0');


begin
    midi_decoder_inst : midi_decoder
        port map (
            clk => clk,
            reset => rst,
            uart_rxd => uart_rxd_in,
            midi_data => midi_data,
            data_ready => midi_received
        );

    sevenSegDisplay_inst : sevenSegDisplay
        port map (
            clk => clk,
            rst => rst,
            midi_msg => midi_data,
            anode => anode,
            cathode => cathode
        );

    note_control_inst : note_control3x
        port map (
            clk => clk,
            reset => rst,
            note_on => note_on,
            note_off => note_off,
            note_value => note_value,
            note_offset2 => note_offset2,
            note_offset3 => note_offset3,
            en => en_reg,
            phaseInc_base => phaseInc_base_reg,
            phaseInc_second =>
                phaseInc_second_reg,
            phaseInc_third => phaseInc_third_reg
        );

    core_inst : syn_core3x
        port map (
            clk => clk,
            reset => rst,
            opPhase1 => phaseInc_base_reg,
            opPhase2 => phaseInc_second_reg,
            opPhase3 => phaseInc_third_reg,

            ampl2 => ampl_second,
            ampl3 => ampl_third,

            opEnable => en_reg,
            att => att,
            dec => dec,
            sus => sus,
            rel => rel,
            ampl_master => master_volume,
            nextSample => nextSample,
            audioOut => audio,
            waveSel => wave_sel
        );

    audio_level_meter_inst : audio_level_meter
        generic map(
            DATA_WIDTH => 16
        )
        port map(
            audio_in => audio,
            level_out => volume_visualizer_out
        );

    pwm_enc_inst: pwm_enc
        generic map(
            pwm_period => to_unsigned(4096,16)
        )
        port map(
            clk => clk,
            rst => rst,
            audio_amplitude => audio,
            pwm_out => pwm_out
        );

    -- Generate nextSample signal
    process (clk)
    begin
        if rising_edge(clk) then
            if counter = 2**10-1 then
                nextSample <= '1';
                counter <= 0;
            else
                nextSample <= '0';
                counter <= counter + 1;
            end if;
        end if;
    end process;
```

```
-- Signal assignment
-- led_out <= audio;
pwm_sd <= '1'; -- amplify the output audio

process(clk, midi_received)
begin
    if rising_edge(clk) then
        if midi_received = '1' then
            if midi_data(23 downto 16) = x"90
                " then
                -- Note On
                note_on <= '1';
                note_off <= '0';
                note_value <= midi_data(15
                    downto 8);
            elsif midi_data(23 downto 16) = x
                "80" then
                -- Note Off
                note_on <= '0';
                note_off <= '1';
                note_value <= midi_data(15
                    downto 8);
            elsif midi_data(23 downto 16) = x
                "B0" then
                -- Control Change
                note_on <= '0';
                note_off <= '0';
                case midi_data(15 downto 8)
                    is
                    when x"01" =>
                    -- change master volume
                        master_volume <=
                            signed(midi_data
                            (7 downto 0));
                    when x"0E" =>
                    -- attack
                        att <= signed(
                            midi_data(7
                            downto 0));
                    when x"0F" =>
                    -- decay
                        dec <= signed(
                            midi_data(7
                            downto 0));
                    when x"10" =>
                    -- sustain
                        sus <= signed(
                            midi_data(7
                            downto 0));
                    when x"11" =>
                    -- release
                        rel <= signed(
                            midi_data(7
                            downto 0));
                    when x"12" =>
                    -- note offset for second
                         oscillators
                        note_offset2 <=
                            signed(midi_data
                            (7 downto 0))+
                            to_signed(-64,8)
                            ;
                    when x"13" =>
                    -- amplitude for second
                        oscillators
                        ampl_second <= signed
                            (midi_data(7
                            downto 0));
                    when x"14" =>
                    -- note offset for third
                        oscillators
                        note_offset3 <=
                            signed(midi_data
                            (7 downto 0))+
                            to_signed(-64,8)
                            ;
                    when x"15" =>
                    -- amplitude for third
                        oscillators
                        ampl_third <= signed(
                            midi_data(7
                            downto 0));
                    when x"16" =>
                    -- waveform for base
                        oscillators
                        wave_sel(5 downto 4)
                            <= midi_data(1
                            downto 0);
                    when x"17" =>
                    -- waveform for second
                        oscillators
                        wave_sel(3 downto 2)
                            <= midi_data(1
                            downto 0);
                    when x"18" =>
                    -- waveform for third
                        oscillators
                        wave_sel(1 downto 0)
                            <= midi_data(1
                            downto 0);
                    when others =>
                        null;
                end case;
            end if;
        else
            note_on <= '0';
            note_off <= '0';
        end if;
    end if;
end process;

end architecture;
```

8.2 midi_decoder.vhd

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

entity midi_decoder is
    Port (
        clk : in STD_LOGIC;
```

```
        reset : in STD_LOGIC;
        uart_rxd : in STD_LOGIC;
        midi_data : out STD_LOGIC_VECTOR(23
            downto 0);
        data_ready : out STD_LOGIC
    );
end midi_decoder;

architecture Behavioral of midi_decoder is
    component UART is
        Generic (
            CLK_FREQ : integer := 50e6;
            BAUD_RATE : integer := 115200;
            PARITY_BIT : string := "none";
            USE_DEBOUNCER : boolean := True
        );
        Port (
            CLK : in std_logic;
            RST : in std_logic;
            UART_TXD : out std_logic;
            UART_RXD : in std_logic;
            DIN : in std_logic_vector(7 downto 0)
                ;
            DIN_VLD : in std_logic;
            DIN_RDY : out std_logic;
            DOUT : out std_logic_vector(7 downto
                0);
            DOUT_VLD : out std_logic;
            FRAME_ERROR : out std_logic;
            PARITY_ERROR : out std_logic
        );
    end component;

    signal uart_dout : std_logic_vector(7 downto
        0);
    signal uart_dout_vld : std_logic;
    signal byte_count : integer range 0 to 2 :=
        0;
    signal midi_data_reg : std_logic_vector(23
        downto 0) := (others => '0');
    signal data_ready_reg : std_logic := '0';

begin
    uart_inst : UART
        generic map(
            CLK_FREQ => 1e8,
            BAUD_RATE => 460800,
            PARITY_BIT => "none",
            USE_DEBOUNCER => True
        )
        port map (
            CLK => clk,
            RST => reset,
            UART_TXD => open,
            UART_RXD => uart_rxd,
            DIN => (others => '0'),
            DIN_VLD => '0',
            DIN_RDY => open,
            DOUT => uart_dout,
            DOUT_VLD => uart_dout_vld,
            FRAME_ERROR => open,
            PARITY_ERROR => open
        );

    process(clk, reset)
    begin
        if reset = '1' then
            byte_count <= 0;
            midi_data_reg <= (others => '0');
            data_ready_reg <= '0';
        elsif rising_edge(clk) then
            if uart_dout_vld = '1' then
                case byte_count is
                    when 0 =>
                        midi_data_reg(23 downto
                            16) <= uart_dout;
                        byte_count <= 1;
                    when 1 =>
                        midi_data_reg(15 downto
                            8) <= uart_dout;
                        byte_count <= 2;
                    when 2 =>
                        midi_data_reg(7 downto 0)
                             <= uart_dout;
                        data_ready_reg <= '1';
                        byte_count <= 0;
                end case;
            end if;
            if data_ready_reg = '1' then
                data_ready_reg <= '0';
            end if;
        end if;
    end process;

    midi_data <= midi_data_reg;
    data_ready <= data_ready_reg;
end Behavioral;
```

8.3 oscillator.vhd

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


use work.syn_parts.all;


entity oscillator is
    Port (
        clk: in std_logic;
        reset: in std_logic;
        waveSel: in std_logic_vector(1 downto 0);
        phaseInc: in unsigned(15 downto 0);
        nextSample: in std_logic;
        wavformOut: out signed(15 downto 0)
    );
end oscillator;

architecture implementation of oscillator is
    signal LUTSine: std_logic_vector(15 downto 0)
        ;
    signal LUTAddr: std_logic_vector(10 downto 0)
```

```
        ;
    signal sine, saw, triangle, square: signed(15
         downto 0) := to_signed(0, 16);
    signal sampleCount, counterPhase: unsigned(17
         downto 0);
    signal counterCtrl: std_logic_vector(1 downto
         0);


begin

    wavformOut <=
        sine when waveSel = "00" else
        triangle when waveSel = "01" else
        saw when waveSel = "10" else
        square when waveSel = "11" else
        (others => '0');

    counterPhase <= "00" & phaseInc;
    counterCtrl <= nextSample & '0';
    sample_counter: counter
    generic map (
        WIDTH => 18
    )
    port map (
        clk => clk,
        reset => reset,
        cw => counterCtrl,
        D => counterPhase,
        count => sampleCount
    );
    LUTAddr <= std_logic_vector(sampleCount(17
        downto 7));
    saw <= signed(sampleCount(17 downto 2) -
        32768);

    square <=
        to_signed(-32767, 16) when (sampleCount <
             2**17) else
        to_signed(32767, 16);




    triangle <=
        (signed(sampleCount(16 downto 1) - 32767)
            ) when sampleCount < 2**17 else
        (signed(32767 - sampleCount(16 downto 1))
            );


   sine_wave_generator: entity work.sine_lut
    generic map (
        DEPTH => 2048,
        WIDTH => 16
    )
    port map (
        clk => clk,
        addr => LUTAddr,
        dout => LUTSine
    );

sine <= signed(LUTSine);

end implementation;
```

8.4 adsr.vhd

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

Library UNISIM;
use UNISIM.vcomponents.all;

Library UNIMACRO;
use UNIMACRO.vcomponents.all;

entity adsr is
    generic (
        DATA_WIDTH: integer := 16
    );
    port (
        clk: in std_logic;
        reset: in std_logic;
        en: in std_logic;
        nextSample: in std_logic;
        attack: in signed(7 downto 0);
        decay: in signed(7 downto 0);
        sustain: in signed(7 downto 0);
        rel: in signed(7 downto 0);
        signalIn: in signed(DATA_WIDTH-1 downto
            0);
        signalOut: out signed(DATA_WIDTH-1 downto
             0)
    );
end adsr;

architecture implementation of adsr is
    type ADSR_STATE is (ATTACK_S, DECAY_S,
        SUSTAIN_S, RELEASE_S);
    constant RESULT_WIDTH : integer := DATA_WIDTH
         + 8;
    signal result: std_logic_vector(RESULT_WIDTH
        -1 downto 0);
    signal amplitude: signed(7 downto 0);
    signal nextIncSig: std_logic;
    signal phase_counter : unsigned(15 downto 0);
begin

    signalOut <= signed(result(RESULT_WIDTH-2
        downto RESULT_WIDTH-DATA_WIDTH-1));

    multipy_signal : MULT_MACRO
    generic map (
        DEVICE => "7SERIES",
        LATENCY => 3,   -- Desired clock cycle
            latency, 0-4
        WIDTH_A => DATA_WIDTH,  -- Multiplier A-
            input bus width, 1-25
        WIDTH_B => 8   -- Multiplier B-input bus
            width, 1-18
    )
```

```
    port map (
        P => result,     -- Multiplier ouput bus,
             width determined by WIDTH_P generic
        A => std_logic_vector(signalIn),     --
            Multiplier input A bus, width
            determined by WIDTH_A generic
        B => std_logic_vector(amplitude),     --
            Multiplier input B bus, width
            determined by WIDTH_B generic
        CE => '1',   -- 1-bit active high input
            clock enable
        CLK => clk, -- 1-bit positive edge clock
            input
        RST => reset  -- 1-bit input active high
            reset
    );

    clk_divider : process(clk, nextSample, reset)
         variable count: integer := 0;
         constant maxCount: integer := 7;
         variable prevState: std_logic := '0';
     begin
         if(rising_edge(clk)) then
             if(reset = '1') then
                 count := 0;
                 nextIncSig <= '0';
             elsif(nextSample = '1' and prevState
                  = '0') then
                 count := count + 1;
                 if (count = maxCount) then
                     nextIncSig <= '1';
                     count := 0;
                 end if;
                 prevState := nextSample;
             else
                 nextIncSig <= '0';
                 prevState := nextSample;
             end if;
         end if;
     end process;

adsr_state_machine : process(clk, nextIncSig,
    reset)
       variable state: ADSR_STATE := RELEASE_S;
       variable count: integer := 0;
   begin
       if (rising_edge(clk)) then
           if (reset = '1') then
               state := RELEASE_S;
               amplitude <= (others => '0');
               count := 0;
           elsif (nextIncSig = '1') then
               case state is
                   when ATTACK_S =>
                       if (en = '0') then
                           state := RELEASE_S;
                       elsif(attack = 0) then
                           amplitude <=
                               to_signed(126,8)
                               ;
                           state := DECAY_S;
                       else
                           count := count + 1;
                           if (count = attack)
                               then
                               amplitude <=
                                   amplitude +
                                   1;
                               count := 0;
                               if(amplitude =
                                   to_signed
                                   (126,8))
                                   then
                                   state :=
                                       DECAY_S;
                                   count := 0;
                               end if;
                           end if;
                       end if;

                   when DECAY_S =>
                       if (en = '0') then
                           state := RELEASE_S;
                       elsif(decay = 0) then
                           amplitude <= sustain;
                           state := SUSTAIN_S;
                       else
                           count := count + 1;
                           if (count = decay)
                               then
                               amplitude <=
                                   amplitude -
                                   1;
                               count := 0;
                               if (amplitude =
                                   sustain)
                                   then
                                   state :=
                                       SUSTAIN_S
                                       ;
                                   count := 0;
                               end if;
                           end if;
                       end if;


                   when SUSTAIN_S =>
                       if(en = '0') then
                           state := RELEASE_S;
                       end if;

                   when RELEASE_S =>
                       if(en = '1') then
                           state := ATTACK_S;
                           count := 0;
                       elsif(amplitude > 0) then
                           if(rel = 0) then
                               amplitude <= (
                                   others =>
                                   '0');
                           else
                               count := count +
                                   1;
                               if (count = rel)
```

```
                                    then
                                    amplitude <=
                                        amplitude
                                         - 1;
                                    count := 0;
                                end if;
                            end if;
                        end if;

                    when others => state :=
                        RELEASE_S;
                end case;
            end if;
        end if;
    end process;


end implementation;
```

8.5 mixer.vhd

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

Library UNISIM;
use UNISIM.vcomponents.all;

Library UNIMACRO;
use UNIMACRO.vcomponents.all;

entity mixer is
    generic (
        ACTIVE_CHANNELS : integer := 10;
        DATA_WIDTH: integer := 16
    );
    port (
        clk : in std_logic;
        reset : in std_logic;
        amplitude: in signed(7 downto 0);
        ch1, ch2, ch3, ch4,
        ch5, ch6, ch7, ch8,
        ch9, ch10, ch11, ch12 : in signed(
            DATA_WIDTH-1 downto 0);
        sigOut : out signed(DATA_WIDTH-1 downto
            0)
    );
end mixer;

architecture implementation of mixer is
    function log2ceil(arg : positive) return
        natural is
        variable tmp : positive     := 1;
        variable log : natural      := 0;
    begin
        if arg = 1 then return 0; end if;
        while arg > tmp loop
            tmp := tmp * 2;
            log := log + 1;
        end loop;
        return log;
    end function;

    constant EXTRA_BITS : integer := log2ceil(
        ACTIVE_CHANNELS);
    constant INT_SIG_BITS: integer := EXTRA_BITS+
        DATA_WIDTH;
    signal mixedSigInt : signed(INT_SIG_BITS-1
        downto 0);
    signal sigOutInt: std_logic_vector(DATA_WIDTH
        +7 downto 0);
begin

    sigOut <= signed(sigOutInt(DATA_WIDTH+6
        downto DATA_WIDTH-9));

    process(clk)
    begin
        if(rising_edge(clk)) then
            if(reset = '1') then
                mixedSigInt <= (others => '0');
            else
                mixedSigInt <=
                    resize(ch1, INT_SIG_BITS) +
                        resize(ch2, INT_SIG_BITS
                        )
                    + resize(ch3, INT_SIG_BITS) +
                         resize(ch4,
                        INT_SIG_BITS)
                    + resize(ch5, INT_SIG_BITS) +
                         resize(ch6,
                        INT_SIG_BITS)
                    + resize(ch7, INT_SIG_BITS) +
                         resize(ch8,
                        INT_SIG_BITS)
                    + resize(ch9, INT_SIG_BITS) +
                         resize(ch10,
                        INT_SIG_BITS)
                    + resize(ch11, INT_SIG_BITS)
                        + resize(ch12,
                        INT_SIG_BITS);
            end if;
        end if;
    end process;


    multipy_signal : MULT_MACRO
    generic map (
        DEVICE => "7SERIES",
        LATENCY => 3,   -- Desired clock cycle
            latency, 0-4
        WIDTH_A => DATA_WIDTH,  -- Multiplier A-
            input bus width, 1-25
        WIDTH_B => 8   -- Multiplier B-input bus
            width, 1-18
    )
    port map (
        P => sigOutInt,     -- Multiplier ouput
            bus, width determined by WIDTH_P
            generic
        A => std_logic_vector(mixedSigInt(
            INT_SIG_BITS-1 downto EXTRA_BITS)),
        B => std_logic_vector(amplitude),
```

```
        CE => '1',   -- 1-bit active high input
            clock enable
        CLK => clk, -- 1-bit positive edge clock
            input
        RST => reset  -- 1-bit input active high
            reset
    );

end implementation;
```

8.6 note_control3x.vhd

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

entity note_control3x is
    Port (
        clk : in STD_LOGIC;
        reset : in STD_LOGIC;
        note_on : in STD_LOGIC;
        note_off : in STD_LOGIC;
        note_value : in STD_LOGIC_VECTOR(7 downto
             0);
        note_offset2 : in SIGNED(7 downto 0);
        note_offset3 : in SIGNED(7 downto 0);
        en : out STD_LOGIC_VECTOR(11 downto 0);
        phaseInc_base : out UNSIGNED(191 downto
            0);
        phaseInc_second : out UNSIGNED(191 downto
             0);
        phaseInc_third : out UNSIGNED(191 downto
            0)
    );
end note_control3x;

architecture Behavioral of note_control3x is
    signal en_reg : STD_LOGIC_VECTOR(11 downto 0)
         := (others => '0');
    signal note_reg : STD_LOGIC_VECTOR(95 downto
        0) := (others => '0');
    signal note2_reg : STD_LOGIC_VECTOR(95 downto
         0) := (others => '0');
    signal note3_reg : STD_LOGIC_VECTOR(95 downto
         0) := (others => '0');
    type phaseInc_array_type is array (0 to 11)
        of UNSIGNED(15 downto 0);  -- Define
        array type
    signal phaseInc_array_base :
        phaseInc_array_type := (others => (
        others => '0'));  -- Intermediate array
        for phaseInc
    signal phaseInc_array_second :
        phaseInc_array_type := (others => (
        others => '0'));
    signal phaseInc_array_third :
        phaseInc_array_type := (others => (
        others => '0'));
begin
    process(clk, reset)
    begin
        if reset = '1' then
            en_reg <= (others => '0');
            note_reg <= (others => '0');
            phaseInc_base <= (others => '0');  --
                 Reset phaseInc_array
            phaseInc_second <= (others => '0');
            phaseInc_third <= (others => '0');
        elsif rising_edge(clk) then
            if note_on = '1' then
                if en_reg(11) = '0' then
                    en_reg(11) <= '1';
                    note_reg(95 downto 88) <=
                        note_value;
                elsif en_reg(10) = '0' then
                    en_reg(10) <= '1';
                    note_reg(87 downto 80) <=
                        note_value;
                elsif en_reg(9) = '0' then
                    en_reg(9) <= '1';
                    note_reg(79 downto 72) <=
                        note_value;
                elsif en_reg(8) = '0' then
                    en_reg(8) <= '1';
                    note_reg(71 downto 64) <=
                        note_value;
                elsif en_reg(7) = '0' then
                    en_reg(7) <= '1';
                    note_reg(63 downto 56) <=
                        note_value;
                elsif en_reg(6) = '0' then
                    en_reg(6) <= '1';
                    note_reg(55 downto 48) <=
                        note_value;
                elsif en_reg(5) = '0' then
                    en_reg(5) <= '1';
                    note_reg(47 downto 40) <=
                        note_value;
                elsif en_reg(4) = '0' then
                    en_reg(4) <= '1';
                    note_reg(39 downto 32) <=
                        note_value;
                elsif en_reg(3) = '0' then
                    en_reg(3) <= '1';
                    note_reg(31 downto 24) <=
                        note_value;
                elsif en_reg(2) = '0' then
                    en_reg(2) <= '1';
                    note_reg(23 downto 16) <=
                        note_value;
                elsif en_reg(1) = '0' then
                    en_reg(1) <= '1';
                    note_reg(15 downto 8) <=
                        note_value;
                elsif en_reg(0) = '0' then
                    en_reg(0) <= '1';
                    note_reg(7 downto 0) <=
                        note_value;
                end if;
            elsif note_off = '1' then
                if note_reg(95 downto 88) =
                    note_value then
                    en_reg(11) <= '0';
```

```
            elsif note_reg(87 downto 80) =
                note_value then
                en_reg(10) <= '0';
            elsif note_reg(79 downto 72) =
                note_value then
                en_reg(9) <= '0';
            elsif note_reg(71 downto 64) =
                note_value then
                en_reg(8) <= '0';
            elsif note_reg(63 downto 56) =
                note_value then
                en_reg(7) <= '0';
            elsif note_reg(55 downto 48) =
                note_value then
                en_reg(6) <= '0';
            elsif note_reg(47 downto 40) =
                note_value then
                en_reg(5) <= '0';
            elsif note_reg(39 downto 32) =
                note_value then
                en_reg(4) <= '0';
            elsif note_reg(31 downto 24) =
                note_value then
                en_reg(3) <= '0';
            elsif note_reg(23 downto 16) =
                note_value then
                en_reg(2) <= '0';
            elsif note_reg(15 downto 8) =
                note_value then
                en_reg(1) <= '0';
            elsif note_reg(7 downto 0) =
                note_value then
                en_reg(0) <= '0';
            end if;
        end if;
        -- Concatenate phaseInc_array to form
             the phaseInc signal
        -- Need to concatenate it in the
            process to avoid multiple
            drivers (another driver is in
            reset = '1' branch)
        phaseInc_base <= phaseInc_array_base
            (11) & phaseInc_array_base(10) &
             phaseInc_array_base(9) &
            phaseInc_array_base(8) &
        phaseInc_array_base(7) &
            phaseInc_array_base(6) &
            phaseInc_array_base(5) &
            phaseInc_array_base(4) &
        phaseInc_array_base(3) &
            phaseInc_array_base(2) &
            phaseInc_array_base(1) &
            phaseInc_array_base(0);

        phaseInc_second <=
            phaseInc_array_second(11) &
            phaseInc_array_second(10) &
            phaseInc_array_second(9) &
            phaseInc_array_second(8) &
        phaseInc_array_second(7) &
            phaseInc_array_second(6) &
            phaseInc_array_second(5) &
            phaseInc_array_second(4) &
        phaseInc_array_second(3) &
            phaseInc_array_second(2) &
            phaseInc_array_second(1) &
            phaseInc_array_second(0);

        phaseInc_third <=
            phaseInc_array_third(11) &
            phaseInc_array_third(10) &
            phaseInc_array_third(9) &
            phaseInc_array_third(8) &
        phaseInc_array_third(7) &
            phaseInc_array_third(6) &
            phaseInc_array_third(5) &
            phaseInc_array_third(4) &
        phaseInc_array_third(3) &
            phaseInc_array_third(2) &
            phaseInc_array_third(1) &
            phaseInc_array_third(0);
    end if;
end process;

-- Instantiate note2phaseInc for each note
gen_phaseInc_base: for i in 0 to 11 generate
    phase_inc_inst: entity work.note2phaseInc
        port map (
            note => note_reg((i+1)*8-1 downto
                 i*8),
            phaseInc => phaseInc_array_base(i
                ) -- Map to the intermediate
                 array
        );
end generate;

-- We only use the 6 Most Significant Bits of
     note_offset (regulate it between -32 to
     31)
note2_reg_gen: for i in 0 to 11 generate
    note2_reg((i+1)*8-1 downto i*8) <=
        std_logic_vector(signed(note_reg((i
        +1)*8-1 downto i*8))+resize(
        note_offset2(7 downto 2),8));
end generate;

gen_phaseInc_second: for i in 0 to 11
    generate
    phase_inc_inst: entity work.note2phaseInc
        port map (
            note => note2_reg((i+1)*8-1
                downto i*8),
            phaseInc => phaseInc_array_second
                (i) -- Map to the
                intermediate array
        );
end generate;

note3_reg_gen: for i in 0 to 11 generate
    note3_reg((i+1)*8-1 downto i*8) <=
        std_logic_vector(signed(note_reg((i
        +1)*8-1 downto i*8))+resize(
        note_offset3(7 downto 2),8));
end generate;
```

```
    gen_phaseInc_third: for i in 0 to 11 generate
        phase_inc_inst: entity work.note2phaseInc
            port map (
                note => note3_reg((i+1)*8-1
                    downto i*8),
                phaseInc => phaseInc_array_third(
                    i) -- Map to the
                    intermediate array
            );
    end generate;

    en <= en_reg;

end Behavioral;
```

8.7 pwm_enc.vhd

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

entity pwm_enc is
    generic (
        pwm_period : UNSIGNED(15 downto 0) := x"
            FFFF" -- Default period of the PWM
            signal in clock cycles
    );
    Port (
        clk : in STD_LOGIC;
        rst : in STD_LOGIC;
        audio_amplitude : in SIGNED(15 downto 0);
             -- Audio amplitude as a signed
            value
        pwm_out : out STD_LOGIC -- PWM output
            signal
    );
end pwm_enc;

architecture Behavioral of pwm_enc is
    constant OFFSET : SIGNED(16 downto 0) :=
        to_signed(32768, 17); -- Offset to move
        the amplitude range to positive
    signal amplitude_shifted : SIGNED(16 downto
        0); -- Shifted amplitude
    signal amplitude_unsigned : UNSIGNED(15
        downto 0); -- Unsigned version of
        shifted amplitude
    signal duty_cycle : UNSIGNED(15 downto 0); --
         Calculated duty cycle
    signal pwm_counter : UNSIGNED(15 downto 0);
        -- Counter for PWM period
begin

    process(clk, rst)
    begin
        if rst = '1' then
            amplitude_shifted <= (others => '0');
            amplitude_unsigned <= (others => '0')
                ;
            duty_cycle <= (others => '0');
            pwm_counter <= (others => '0');
            pwm_out <= '0';
        elsif rising_edge(clk) then
            -- Shift the amplitude by adding the
                OFFSET
            amplitude_shifted <= resize(
                audio_amplitude,17) + OFFSET;

            -- Convert shifted amplitude to
                unsigned
            amplitude_unsigned <= resize(unsigned
                (amplitude_shifted),16);

            -- Calculate duty cycle based on
                shifted and converted amplitude
            duty_cycle <= resize((resize(
                amplitude_unsigned, 16) *
                pwm_period) srl 16, 16);
            -- Equivalent to division by 2**16,
                as long as the least significant
                 bit is not 1, the division is
                correct

            -- PWM generation logic
            if pwm_counter < duty_cycle then
                pwm_out <= '1';
            else
                pwm_out <= '0';
            end if;

            -- Increment PWM counter
            if pwm_counter = pwm_period-1 then
                pwm_counter <= (others => '0');
            else
                pwm_counter <= pwm_counter + 1;
            end if;
        end if;
    end process;
end Behavioral;
```