Andrea Lazzari 2045247

Lorenzo Saccaro 2063593

Paolo Zinesi 2053062

Nicola Zomer 2058479

Sound wave distortion via FPGA using Pmod interface

Management and Analysis of Physics Dataset (mod.A)

February 25, 2022

1 Abstract

In this project we implement on a Field Programable Gate Array (FPGA) a distortion effect in sound waves that is called "Overdrive" or "Clipping". This happens when the amplitude of a sound wave is restricted when it exceeds a given threshold. The resulting sounds are "dirty" and "fuzzy" due to the introduction of high frequency components in the signal.

We used an Artix-7 FPGA from Xilinx [4], together with Xilinx Vivado Design Suite 2018.3 [5]. Moreover, the Digilent Pmod I2S2 [6] is used, in order to allow the FPGA to transmit and receive stereo (left-right) audio signals via the I2S protocol.

2 Introduction

As mentioned above, the overdrive effect is realized in a signal when the amplitude of its wave is restricted when it exceeds some threshold value. This effect is not a particular instance of sound waves, but it can be applied in principle to any kind of waves that we can manipulate. We focused on sound waves because, experimentally, they are the simplest to deal with: a microphone and a Pmod module are enough to send sounds to the FPGA.

In Figure 1, two different approaches to restrict the amplitude of the wave are shown [1, p. 25]. Soft clipping occurs when the wave amplitude is scaled proportionally to fit into the given threshold, causing the peak-to-peak amplitude to decrease. It has to be noted that this effect does not simply scale the amplitude of the whole wave but only the amplitude of a particular region of the signal. Thus, the soft clipping will add to the original signal an additional signal (i.e., the difference between the clipped signal and the original signal) that is null outside the clipping region. The Fourier Transform (FT) of the clipped signal shows additional frequency components if compared to the original Fourier Transform of signal.

Hard clipping occurs instead when hard walls are applied to the wave amplitude. In other words, the wave amplitude is limited by the threshold without affecting the amplitudes inside the threshold. Similarly to the Fourier Transform of a square wave, the Fourier Transform of the (hard) clipped signal shows an infinite number of high frequency components with decreasing weights (e.g., see [2, Figure 5] or right plot in Figure 8).

In our project we implement the hard clipping distortion effect, since thresholding the amplitude of the wave is way easier than scaling it.

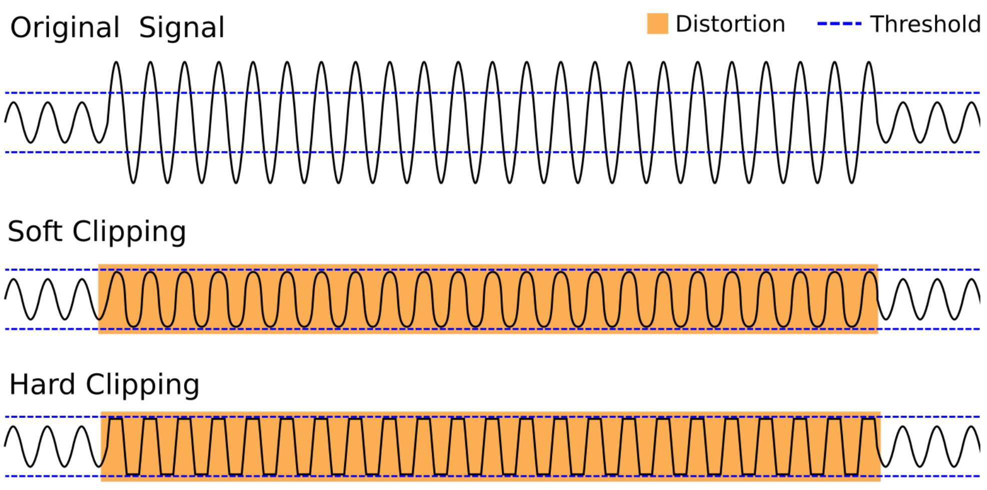

Figure 1: Visual examples of clipping applied to a wave signal.
From Wikipedia [accessed 19 Feb, 2022].

The implementation on the FPGA is carried out by employing a Xilinx Artix 7 device [4], together with Xilinx's Vivado Design Suite [5], and by using the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) and Verilog to model the circuit. After having modelled and implemented the overdrive effect, we represent the waveform and spectrogram of the clipped audio signal by feeding the output signal back into the computer. Furthermore, we compute and plot the Fourier Transform of both the original and clipped signals and compare the outcome with the mathematical results obtained in [2]. The generation and recording of the audio signal are done in Python, using the `PyAudio` module [7]. The resulting FPGA-computer loop is shown in the figure below.

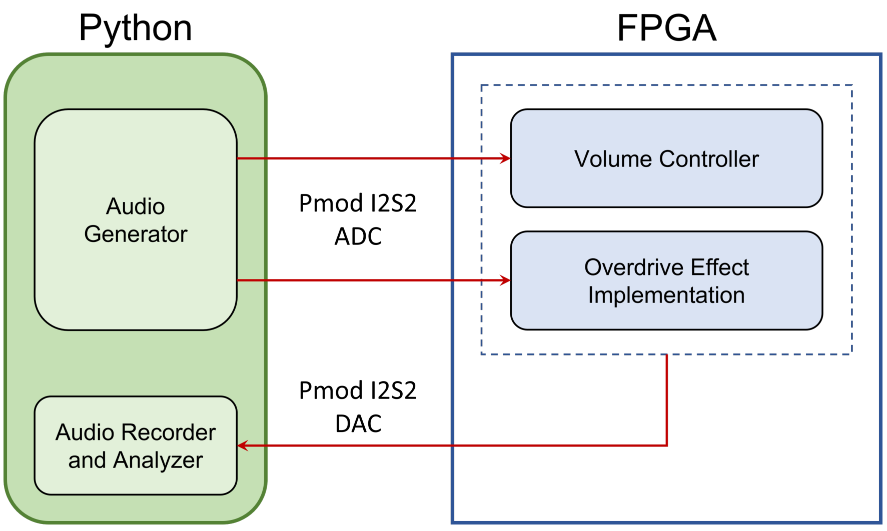

Figure 2: Overdrive effect, overall architecture

This report is organized as follows. Section 3 addresses the implementation of the hard clipping distortion effect on FPGA. Section 4 presents the input audio signals used and how we generated a sinusoidal sound wave. Section 5 explains how we recorded and saved the output audio data and presents the Python code used for the analysis. Section 6 discusses the results obtained. Finally, the report ends with Conclusions (Section 7).

3 Implementation on FPGA

3.1 Pmod I2S2

The Pmod I2S2 implements 16-24 bit A/D and D/A converters that enable stereo input and output via the I2S (Inter-IC Sound) protocol, a serial bus interface standard used for connecting digital audio devices together.
The bus consists of four signals (see [3, Figure 17] for Pmod I2S component detailed view):

- Serial clock (SCLK), i.e., the bit clock line.
- Left-right clock (LRCLK), i.e., the word clock line.
- Serial data stream (SD), i.e., one multiplexed data line.
- Master clock (MCLK), that has the role of synchronizing the internal operations of the analog and digital converters.

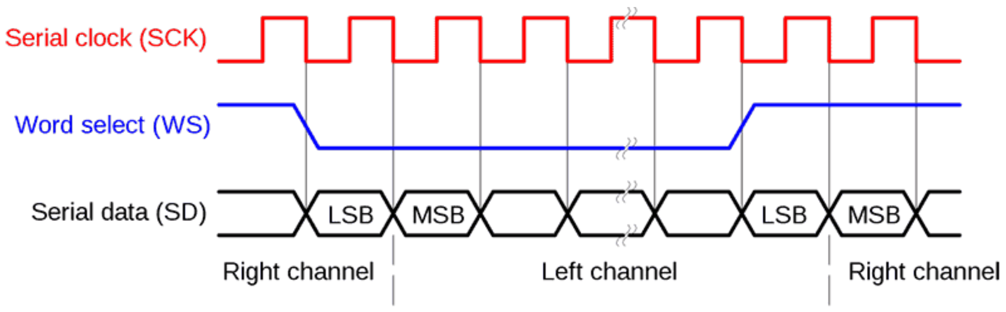

Figure 3: Timing diagram of I2S protocol.
From Wikipedia [accessed 20 Feb, 2022].

The MCLK can be derived from the LRCLK and typically the MCLK/LRCLK frequency ratio is 256, which corresponds to enabling a 16 bit per channel stereo output. In this case, both the left and right channels consist of 16-bit data stream and during one sampling period (equal to a LRCLK period) 32 bits are sent through the serial bus. The bit clock frequency is the product of the sample rate, the number of bits per channel and the number of channels. So, a sample frequency of 44.1 kHz, with 2 channels and 16 bits per channel, corresponds to a SCLK frequency of 1.4112 MHz and a MCLK frequency of 11.2896 MHz.

The clocking wizard LogiCORE™ IP (intellectual property) deals with the implementation of the appropriate clocks, whose values can be easily set in Vivado. The axis_i2s2 module, available in the reference design, manages the input and output data flow between the board and the Pmod I2S2.

3.2 Volume Control

We started by implementing a module that creates a pass-through from the input to the output jack, following the Diligent reference design [6, Pmod - I2S2 Github]. The audio signal that passes through the board is scaled depending on which and how many switches are closed. If all switches are open the input data will be muted, while all switches closed corresponds to full volume.

The intermediate cases between these two extremes are more complicated. Indeed, we can relate each switch to a binary digit and the 4 switches together form a 4-digit binary number, such that each switch is associated to a digit in a specific position. In detail, the switch number "0" on the FPGA board corresponds to the least significant digit, while the switch number "3" corresponds to the most significant one. So, depending on which switches are closed, we can have all the possible 2^4 combinations between 0000 (all switches open) and 1111 (all switches closed). This number is proportional to how much the audio data is scaled.

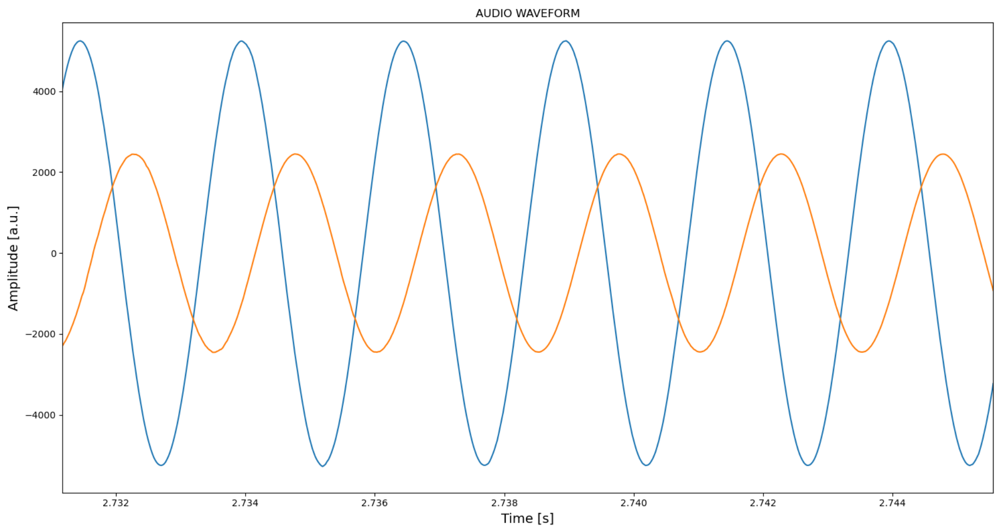

Figure 4: Volume Control effect.
Original input (blue) and scaled output (orange).

3.3 Overdrive effect

The default threshold levels are coded as 24 bits values (lines 24-25, axis_volume_clipping.v). The choice of these values is just based on the fact that for simulated waves the full range is used (all 23 value bits equal to 1 when there is the peak), so, to have a clear visualization (see later), the threshold is set to about 1/4 of the max possible value. It is possible to set different values for the positive and negative threshold, but in this work we kept them symmetrical. When dealing with real or recorded sounds, the full range is usually not used so one should set the thresholds to lower levels to be able to use the overdrive effectively.

In the first `always` block (from line 28, axis_volume_clipping.v) the switches and the button values (the latter used for the activation of the clipping effect) are stored in corresponding registers. The multiplier value is then updated with the following trick: the first 4 bits are set accordingly to the switches' positions while the other 24 are set to 0. The number obtained is then divided by 1111: the multiplier is now a 25 bit number with every group of 4 bits equal to the switches' state except the first that is 0. For example, if the switches from 3 to 0 are set to 1011 the multiplier is then composed as 1011 0000 0000 0000 0000 0000 0000/1111 = 0 1011 1011 1011 1011 1011 1011. The 25th bit is needed for the 1111 case when the multiplier is equal to 1 0000 0000 0000 0000 0000 0000.

In the second `always` block (from line 44, axis_volume_clipping.v) the actual volume scaling and clipping effect are applied. Before the multiplication takes place, the data needs to be extended from 24 to 48 bits: the leading 24 bits are just a copy of the first bit (the sign) of the original data. Using the signed function to preserve the sign, the data is multiplied with the multiplier defined before. The same approach is used for the threshold levels: in this way the clipping effect will work regardless of the volume (even at low volumes it is possible to apply the effect). Finally, only if the button (btn 0, in our case) is pressed the overdrive effect is enabled: according to the sign bit of data, that now is scaled, the value is set to the positive or to the (symmetrical) negative threshold if the value is, respectively, above or below that threshold. The process is repeated for both channels.

top.vhd

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

entity top_vhdl is
    generic(NUMBER_OF_SWITCHES : integer := 4;
            RESET_POLARITY : std_logic := '0');
    port    (clk : in std_logic;
             sw : in std_logic_vector(NUMBER_OF_SWITCHES-1 downto 0);
             btn: in std_logic;
```

```vhdl
             reset : in std_logic;
             tx_mclk : out std_logic;
             tx_lrck : out std_logic;
             tx_sclk : out std_logic;
             tx_data : out std_logic;
             rx_mclk : out std_logic;
             rx_lrck : out std_logic;
             rx_sclk : out std_logic;
             rx_data : in std_logic);
end top_vhdl;

architecture Behavioral of top_vhdl is

component clk_wiz_0
port
 (-- Clock in ports
  -- Clock out ports
  axis_clk          : out    std_logic;
  clk_in1           : in     std_logic
 );
end component;

component axis_i2s2
port(
    axis_clk : in std_logic;
    axis_resetn : in std_logic;
    tx_axis_s_data : in std_logic_vector(31 downto 0);
    tx_axis_s_valid : in std_logic;
    tx_axis_s_ready : out std_logic;
    tx_axis_s_last : in std_logic;
    rx_axis_m_data : out std_logic_vector(31 downto 0);
    rx_axis_m_valid : out std_logic;
    rx_axis_m_ready : in std_logic;
    rx_axis_m_last : out std_logic;
    tx_mclk : out std_logic;
    tx_lrck : out std_logic;
    tx_sclk : out std_logic;
    tx_sdout : out std_logic;
    rx_mclk : out std_logic;
    rx_lrck : out std_logic;
    rx_sclk : out std_logic;
    rx_sdin : in std_logic);
end component;

component axis_volume_clipping
generic(DATA_WIDTH : integer := 24;
        SWITCH_WIDTH : integer := NUMBER_OF_SWITCHES);
port(
    clk : in std_logic;
    btn : in std_logic;
    sw : in std_logic_vector(3 downto 0);
    s_axis_data : in std_logic_vector(23 downto 0);
    s_axis_valid : in std_logic;
    s_axis_ready : out std_logic;
    s_axis_last : in std_logic;
    m_axis_data : out std_logic_vector(23 downto 0);
    m_axis_valid : out std_logic;
    m_axis_ready : in std_logic;
    m_axis_last : out std_logic);
end component;


signal axis_clk : std_logic;
signal axis_tx_data : std_logic_vector(23 downto 0);
signal axis_tx_data32 : std_logic_vector(31 downto 0);
signal axis_tx_valid : std_logic;
signal axis_tx_ready : std_logic;
signal axis_tx_last : std_logic;
signal axis_rx_data : std_logic_vector(23 downto 0);
signal axis_rx_data32 : std_logic_vector(31 downto 0);
signal axis_rx_valid : std_logic;
signal axis_rx_ready : std_logic;
signal axis_rx_last : std_logic;


signal resetn : std_logic;



begin

resetn <= '0' when reset = RESET_POLARITY else
          '1';

m_clk : clk_wiz_0
   port map (
   axis_clk => axis_clk,
   clk_in1 => clk
 );

axis_tx_data32(23 downto 0) <= axis_tx_data;
axis_rx_data <= axis_rx_data32(23 downto 0);

m_i2s2 : axis_i2s2
    port map (
        axis_clk => axis_clk,
        axis_resetn => resetn,
        tx_axis_s_data => axis_tx_data32,
        tx_axis_s_valid => axis_tx_valid,
        tx_axis_s_ready => axis_tx_ready,
        tx_axis_s_last => axis_tx_last,
        rx_axis_m_data => axis_rx_data32,
        rx_axis_m_valid => axis_rx_valid,
        rx_axis_m_ready => axis_rx_ready,
        rx_axis_m_last => axis_rx_last,
        tx_mclk => tx_mclk,
        tx_lrck => tx_lrck,
        tx_sclk => tx_sclk,
        tx_sdout => tx_data,
        rx_mclk => rx_mclk,
        rx_lrck => rx_lrck,
        rx_sclk => rx_sclk,
        rx_sdin => rx_data
    );


m_vol_clip : axis_volume_clipping
    generic map(SWITCH_WIDTH => 4,
            DATA_WIDTH => 24)
    port map (
        clk => axis_clk,
        btn => btn,
        sw => sw,
        s_axis_data => axis_rx_data,
        s_axis_valid => axis_rx_valid,
        s_axis_ready => axis_rx_ready, --same
        s_axis_last => axis_rx_last,
        m_axis_data => axis_tx_data, --same
        m_axis_valid => axis_tx_valid, --same
        m_axis_ready => axis_tx_ready,
        m_axis_last => axis_tx_last --same
    );
```

```


end Behavioral;
```

axis_volume_clipping.v

```
module axis_volume_clipping #(
    parameter SWITCH_WIDTH = 4, // WARNING: this module has not been tested with other values of SWITCH_WIDTH, it will likely need some changes
    parameter DATA_WIDTH = 24
) (
    input wire clk,
    input wire btn,
    input wire [SWITCH_WIDTH-1:0] sw,

    //AXIS SLAVE INTERFACE
    input  wire [DATA_WIDTH-1:0] s_axis_data,
    input  wire s_axis_valid,
    output reg  s_axis_ready = 1'b1,
    input  wire s_axis_last,

    // AXIS MASTER INTERFACE
    output reg [DATA_WIDTH-1:0] m_axis_data = 1'b0,
    output reg m_axis_valid = 1'b0,
    input  wire m_axis_ready,
    output reg m_axis_last = 1'b0
);
    localparam MULTIPLIER_WIDTH = 24;

    // store data to be manipulated
    reg [MULTIPLIER_WIDTH+DATA_WIDTH-1:0] data [1:0];

    // reg to save btn and sw values
    reg btn_r = 1'b0;
    reg [SWITCH_WIDTH-1:0] sw_sync_r [2:0];
    wire [SWITCH_WIDTH-1:0] sw_sync = sw_sync_r[2];

    // volume multiplier
    reg [MULTIPLIER_WIDTH:0] multiplier = 'b0; // range of 0x00:0x10 for width=4

    // threshold levels for clipping
    reg [MULTIPLIER_WIDTH:0] threshold_h = {1'b0, 7'b001111, 16'h0000};
    reg [MULTIPLIER_WIDTH:0] threshold_l = {1'b1, 7'b000001, 16'h0000};

    wire m_select = m_axis_last;
    wire m_new_word = (m_axis_valid == 1'b1 && m_axis_ready == 1'b1) ? 1'b1 : 1'b0;
    wire m_new_packet = (m_new_word == 1'b1 && m_axis_last == 1'b1) ? 1'b1 : 1'b0;

    wire s_select = s_axis_last;
    wire s_new_word = (s_axis_valid == 1'b1 && s_axis_ready == 1'b1) ? 1'b1 : 1'b0;
    wire s_new_packet = (s_new_word == 1'b1 && s_axis_last == 1'b1) ? 1'b1 : 1'b0;
    reg s_new_packet_r = 1'b0;

    // threshold to be updated with volume level
    reg [MULTIPLIER_WIDTH+DATA_WIDTH-1:0] thresh_h = 'b0;
    reg [MULTIPLIER_WIDTH+DATA_WIDTH-1:0] thresh_l = 'b0;

    always@(posedge clk) begin

        // save switches and button values
        sw_sync_r[2] <= sw_sync_r[1];
        sw_sync_r[1] <= sw_sync_r[0];
        sw_sync_r[0] <= sw;

        btn_r <= btn;

```

```verilog
     // update multiplier variable: 24 bit (6 groups of 4 bit with values equal to sw_sync)
     multiplier <= {sw_sync,{MULTIPLIER_WIDTH{1'b0}}} / {SWITCH_WIDTH{1'b1}};

     s_new_packet_r <= s_new_packet;

 end

 always@(posedge clk)
     if (s_new_word == 1'b1) begin// sign extend and register AXIS slave data
         data[s_select] = {{MULTIPLIER_WIDTH{s_axis_data[DATA_WIDTH-1]}}, s_axis_data};

         // prepare threshold values to be multiplied
         thresh_h = {{MULTIPLIER_WIDTH{threshold_h[DATA_WIDTH-1]}}, threshold_h};
         thresh_l = {{MULTIPLIER_WIDTH{threshold_l[DATA_WIDTH-1]}}, threshold_l};

     end else if (s_new_packet_r == 1'b1) begin

             // core volume control algorithm, infers a DSP48 slice
             data[0] = $signed(data[0]) * multiplier;
             data[1] = $signed(data[1]) * multiplier;

             // threshold volume scaling
             thresh_h = $signed(thresh_h) * multiplier;
             // assuming same value for negative clipping, can be changed for different positive/negative threshold values
             thresh_l = ~thresh_h + 1;

         // clipping code: if values are higher/lower than thresh_h/thresh_l assign these values
         // clipping effect is active only when pressing btn 0 on the board
         if (btn_r == 1'b1) begin
            if (data[0][MULTIPLIER_WIDTH+DATA_WIDTH-1] == 1'b0) begin
                 if (data[0] > thresh_h) begin
                     data[0] = thresh_h ;
                 end
            end
            if (data[0][MULTIPLIER_WIDTH+DATA_WIDTH-1] == 1'b1) begin
                 if (data[0] < thresh_l) begin
                     data[0] = thresh_l;
                 end
            end
            if (data[1][MULTIPLIER_WIDTH+DATA_WIDTH-1] == 1'b0) begin
                 if (data[1] > thresh_h) begin
                     data[1] = thresh_h ;
                 end
            end
            if (data[1][MULTIPLIER_WIDTH+DATA_WIDTH-1] == 1'b1) begin
                 if (data[1] < thresh_l) begin
                     data[1] = thresh_l;
                 end
            end
          end
      end


// updating variables for input and output data control
 always@(posedge clk)
     if (s_new_packet_r == 1'b1)
         m_axis_valid <= 1'b1;
     else if (m_new_packet == 1'b1)
         m_axis_valid <= 1'b0;

```

```
    always@(posedge clk)
        if (m_new_packet == 1'b1)
            m_axis_last <= 1'b0;
        else if (m_new_word == 1'b1)
            m_axis_last <= 1'b1;

    always@(m_axis_valid, data[0], data[1], m_select)
        if (m_axis_valid == 1'b1) begin
            m_axis_data = data[m_select][MULTIPLIER_WIDTH+DATA_WIDTH-1:
   MULTIPLIER_WIDTH];
        end else
            m_axis_data = 'b0;


    always@(posedge clk)
        if (s_new_packet == 1'b1)
            s_axis_ready <= 1'b0;
        else if (m_new_packet == 1'b1)
            s_axis_ready <= 1'b1;

endmodule
```

4 Pure Tone Generation using PyAudio

As input signals we used pure tones, i.e. sounds with a sinusoidal waveform, and short instrumental solo pieces, saved as .wav files. To generate the former, we wrote a simple Python script using the PyAudio module [7].

audio_generator.py

```
p = pyaudio.PyAudio()

# parameters
volume = 1                # range [0.0, 1.0]
fs = 44100                # sampling rate, Hz
duration = 10.0           # seconds
f = 10000.0               # sine frequency, Hz

# generate samples, note conversion to float32 array
samples = (np.sin(2*np.pi*np.arange(fs*duration)*f/fs)).astype(np.float32)

# pyaudio class instance
stream = p.open(
            format=pyaudio.paFloat32,
            channels=1,
            rate=fs,
            output=True)

# play
stream.write(volume*samples)
```

5 Audio Recorder and Analyzer

We used a Jack to Jack cable to send the output audio data back to the computer, in order to analyze it and study the overdrive effect. In particular, we plotted the waveform, the spectrogram and the Fast Fourier Transform (FFT) of the original and the clipped signal. To record the output signal and perform the analysis, we realized a Jupyter Notebook, which is briefly explained in this section.

audio_recorder.ipynb (Parameters)

```python
# input devices
import sounddevice as sd
print(sd.query_devices())

# from the output list select the number of the input device and give it to "INDEX"

# constants
INDEX = 1                       # audio device index
CHUNK = 1024 * 2                # how many audio samples per frame we display
FORMAT = pyaudio.paInt16        # 16bit format per sample
CHANNELS = 1                    # single channel for microphone
RATE = 44100                    # samples per second [Hz]
rec_time = 20                   # recording time [seconds]
delay_time = 2                  # time after which it starts recording [seconds]
```

Firstly, the outgoing audio is recorded and saved as a numpy array and a .wav file. We start recording after a certain period of time (`delay_time`) and stop after a finite time (`rec_time`).

audio_recorder.ipynb (Recorder)

```python
# pyaudio class instance
p = pyaudio.PyAudio()

# stream object to get data from microphone
stream = p.open(
    format=FORMAT,
    channels=CHANNELS,
    rate=RATE,
    input=True,
    frames_per_buffer=CHUNK,
    input_device_index=INDEX
)

# start the process
start_time = time.time()

# delay
while True:
    current_time = time.time()
    elapsed_delaytime = current_time - start_time
    if elapsed_delaytime > delay_time:
        break

# recording
frames = []
while True:
    current_time = time.time()
    elapsed_time = current_time - (start_time + delay_time)

    if elapsed_time > rec_time:
        break

    # binary data
    data = stream.read(CHUNK)   # read 1 chunk at a time
    frames.append(data)

stream.stop_stream()
stream.close()
p.terminate()

# save the recorded data as a WAV file
wf = wave.open(filename, 'wb')
wf.setnchannels(CHANNELS)
wf.setsampwidth(p.get_sample_size(FORMAT))
wf.setframerate(RATE)
wf.writeframes(b''.join(frames))
wf.close()

# join the frames and convert to integers
amplitude = np.frombuffer(b''.join(frames), dtype=np.int16)
```

Then we represent the waveform of the recorded audio signal, both as a function of the samples and of time, and we plot its spectrogram.

Finally, we perform the Fourier trasform, using the module `fftpack` from `scipy` [8]. This is done separately for the original and clipped signal, which are recorded together in the same numpy array (`amplitude`) and .wav file, by changing the extremes of the considered interval (`signal_start`, `signal_end`).

6 Results

6.1 Implementation results

The resulting register transfer level (RTL) schematic is shown in Figure 5. The resources utilization, presented in Figure 6 and realized with the Xilinx Vivado Design Suite tool, allows further development of the project, leaving room to implement real time control of the amplitude threshold or other audio effects.

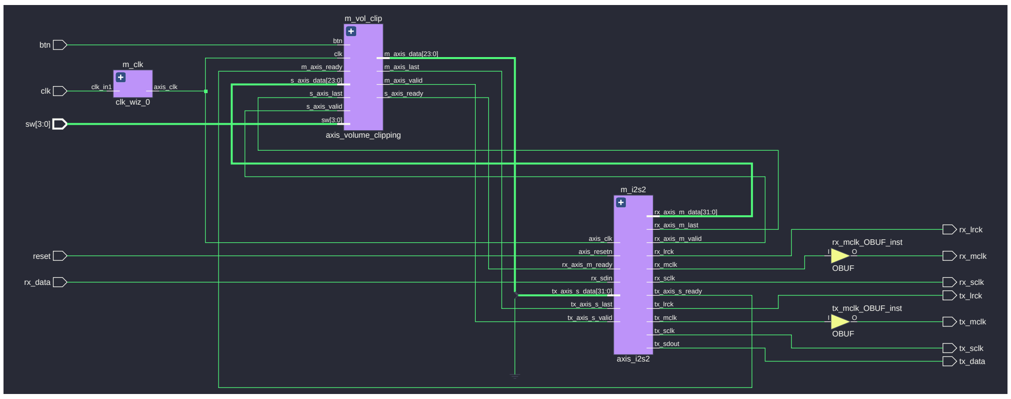

Figure 5: Overdrive effect RTL design

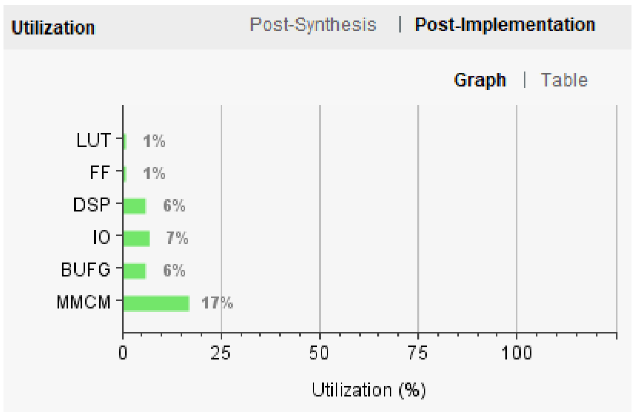

Figure 6: Resources utilization

6.2 Analysis of the output signal

6.2.1 Single-tone sine wave signal

The first test was performed on a single-tone sine wave signal that was generated as described in Section 4. Different waves were generated by varying the frequency but only one of them is reported here for clarity. The generated sound was then sent to the FPGA to be processed and the output sound was recorded by the Jupyter Notebook described in Section 5.

In the plots below the effect of the FPGA processing is clearly visible: the original sound wave is cut when its amplitude exceeds the threshold. As expected, the FFT presents only a single peak for the original wave while the output wave FFT presents many peaks with decreasing power at increasing frequencies. The additional frequencies added to the sound create the distortion effect we are looking for.

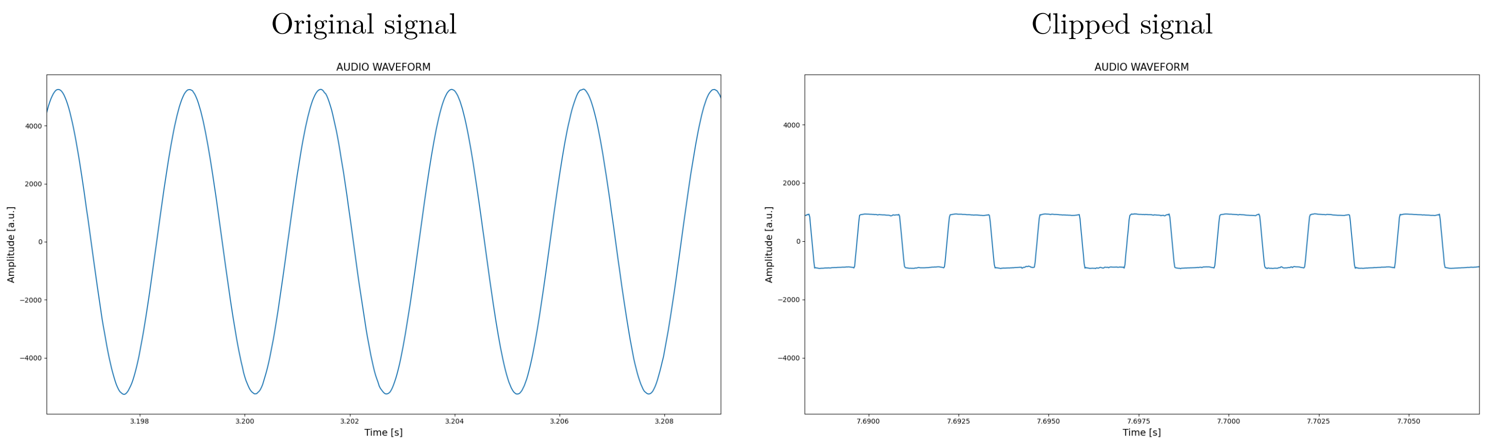

Figure 7: Waveform of 400 Hz sine wave

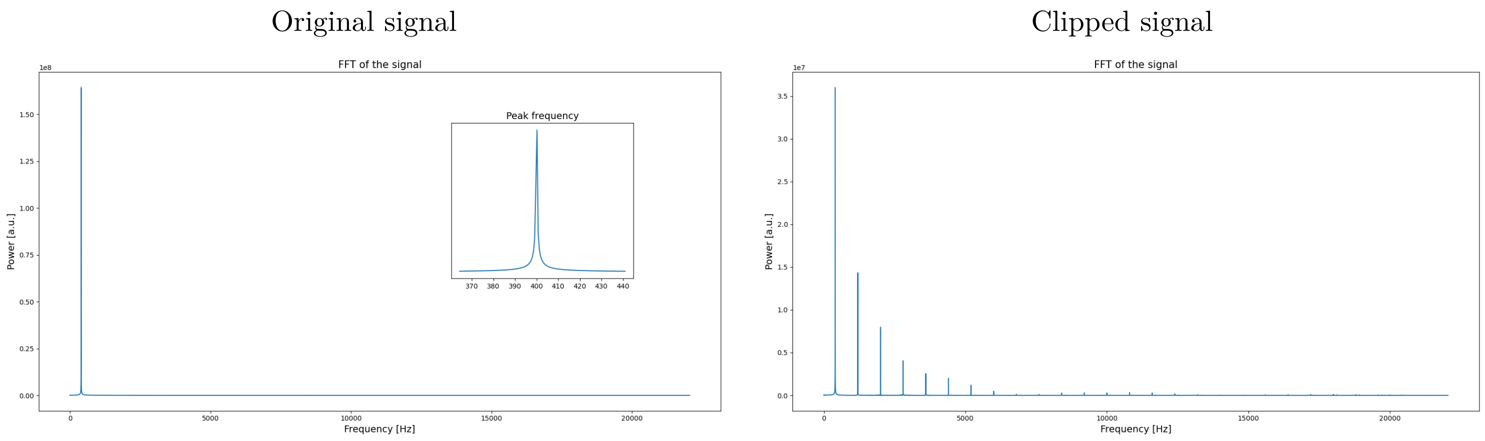

Figure 8: FFT of 400 Hz sine wave

6.2.2 Instrumental solo piece

The distortion effect was also applied to an audio sample produced by a celesta instrument. This sample is profoundly different from the single-tone sound because of the complexity of the produced sound, a mixture of different frequencies that gives as a result the characteristic tone of the instrument. However, by using Fourier analysis this complex sound can be decomposed into single-tone waves with their own frequencies and the observations about single-tone waves still hold.

In the plots below the original sound and the clipped sound are shown on the same axis. We provided as input to the FPGA the sound sample twice, but the button on the FPGA that enables the clipping effect was pressed only at the second play. We can see clearly that the sound amplitude is truncated outside the threshold only when the button was pressed, while no effect was applied when the button was not pressed. Thus, we also tested the ability to control the clipping effect in real-time. The spectrograms of the two sounds differ greatly because of the high frequency components introduced by the clipping effect. While the former spectrogram ranged over a small frequency range, the latter spread over a wider range of frequencies.

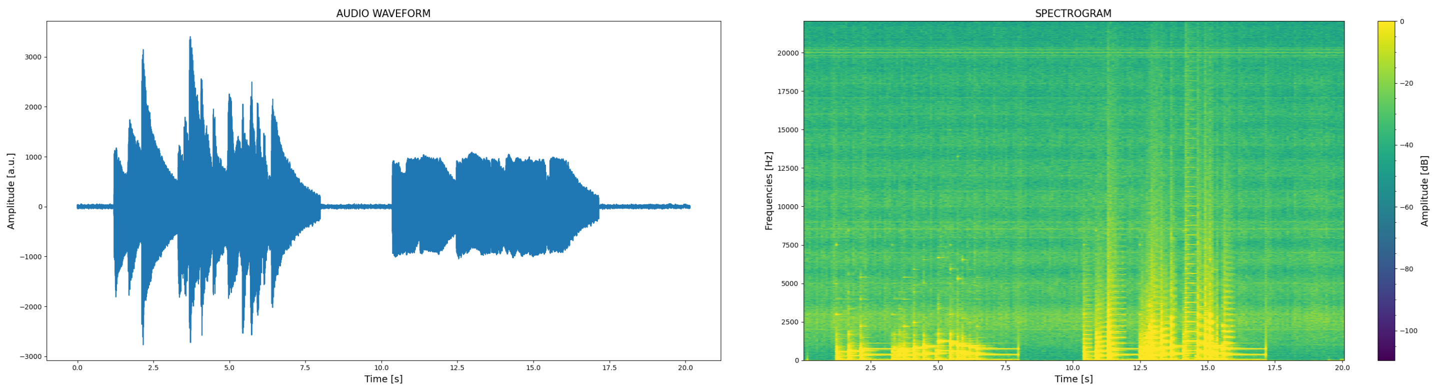

Figure 9: Waveform and spectrogram of celesta audio sample

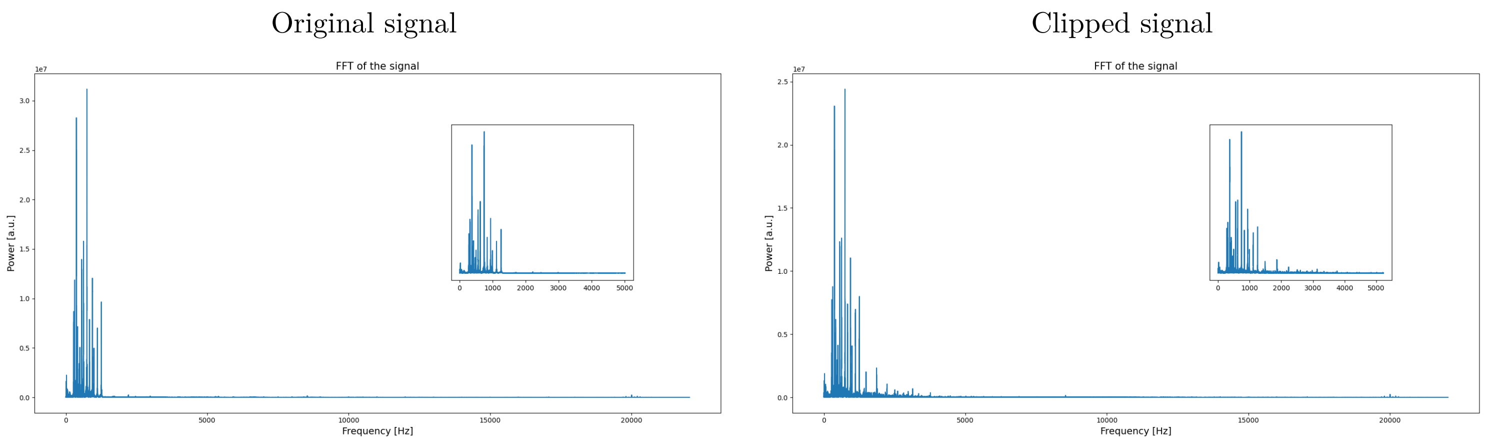

Figure 10: FFTs of the celesta audio sample

7 Conclusions

This report presented an implementation of the overdrive effect on FPGA. It was realized after programming the simple volume control, which helped us understanding how the Pmod I2S2 works and how to implement on FPGA an audio pass-through from Line-In to the Line-Out jack, using Xilinx's Vivado Design Suite. After implementing the effect, we programmed a button on the FPGA to enable and disable it in real-time.

Both effects (volume control and clipping) were designed and tested on sinusoidal waves (generated with a Python script) and on a generic audio sample (instrumental solo piece).

As expected, the original audio waves are cut when the amplitude exceeds the selected threshold. Furthermore, in the Fourier domain, when the input signal is clipped each frequency component is spread into many additional frequencies, with decreasing power.

A future improvement of this work could be the implementation of a real-time variable threshold, to be dynamically selected with a knob or with the switches.

References

[1] Connor William Blasie. "Design of DSP Guitar Effects with FPGA Implementation". MA thesis. Rochester Institute of Technology, 2020. URL: https://scholarworks.rit.edu/theses/10361.

[2] Sujit Rokka Chhetri et al. "Implementation of Audio Effect Generator in FPGA". In: *Nepal Journal of Science and Technology* 15 (2015), 89–98. DOI: 10.3126/njst.v15i1.12022. URL: https://www.nepjol.info/index.php/NJST/article/view/12022.

[3] Ciprian Dragoi et al. "Efficient FPGA Implementation of Classic Audio Effects". In: July 2021, pp. 1–6. DOI: 10.1109/ECAI52376.2021.9515041.

[4] *Xilinx Artix 7*. URL: https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html.

[5] *Vivado Design Studio*. URL: https://www.xilinx.com/products/design-tools/vivado.html.

[6] *Diligent Reference - Pmod I2S2*. URL: https://digilent.com/reference/pmod/pmodi2s2/start.

[7] *PyAudio 0.2.11 documentation*. URL: https://people.csail.mit.edu/hubert/pyaudio/docs/.

[8] *SciPy fftpack documentation*. URL: https://docs.scipy.org/doc/scipy/reference/fftpack.html.