FPGA-Based Audio Equalizer

Team Details:

1. K. Eswar Adithya (12311440) – B41
2. S. Sarath Kumar (12312393) – B42

Subject: Workshop on Reconfigurable Computing (ECE365)

Submitted To: Dr. Sobhit Saxena.

Abstract

This report details the design and implementation of a digital audio playback system on a Xilinx Artix-7 FPGA, specifically using the Digilent Nexys 4 development board. The project demonstrates a complete software-to-hardware workflow for rendering audio without the need for an external Digital-to-Analog Converter (DAC). The core methodology involves pre-processing a digital audio file in a software environment (MATLAB/Python), where digital filtering techniques (FIR/IIR) can be applied. The processed audio samples are then converted into a Xilinx Coefficient File (.coe) and used to initialize an on-chip Block RAM (BRAM).

A custom hardware module, designed in Verilog, is implemented on the FPGA to read the audio samples sequentially from the BRAM at a standard audio sampling rate of 44.1 kHz. The digital samples are then converted into an analog-equivalent signal using Pulse-Width Modulation (PWM). The Verilog design includes a clock divider to generate the precise audio sampling clock from the board's 100 MHz system clock, an address counter for BRAM access, and a PWM generator. The resulting PWM signal is output through the FPGA's pins to an audio jack, enabling playback on standard speakers or headphones. The project successfully validates the use of FPGAs for direct digital audio synthesis and demonstrates the efficacy of PWM as a simple, resource-efficient DAC.

Table of Contents

1. Introduction

1.1. Motivation

Digital audio processing is a cornerstone of modern electronics, from consumer devices to professional studio equipment. The ability to synthesize, manipulate, and play back sound using digital hardware is a fundamental skill in electronics engineering. This project is motivated by the desire to create a standalone, custom hardware system for audio playback, moving beyond software-based solutions to explore the intricacies of hardware-level signal generation.

1.2. Importance of FPGAs in Audio Processing

Field-Programmable Gate Arrays (FPGAs) offer a unique platform for digital signal processing (DSP) applications due to their parallel architecture. Unlike microprocessors that execute instructions sequentially, FPGAs can perform many operations simultaneously, making them ideal for high-throughput tasks like audio processing. This project leverages the FPGA to create a dedicated hardware circuit for audio playback, which is more efficient and deterministic than a software-based player.

1.3. Project Scope

The scope of this project encompasses the entire lifecycle of an embedded audio system, from software signal conditioning to hardware implementation and verification. Key aspects include:

- Pre-processing a .wav audio file in MATLAB or Python.
- Generating a hardware-compatible memory initialization file (.coe).
- Designing a Verilog module to read from on-chip memory.
- Implementing a digital-to-analog conversion scheme using Pulse-Width Modulation (PWM).
- Testing the final hardware implementation on a Digilent Nexys 4 FPGA board.

1.4. Report Structure

This report provides a comprehensive overview of the project. It begins with the project objectives and required components. The core of the report details the workflow, design methodology, software preparation, and Verilog hardware design. Subsequent sections present the simulation and hardware results, followed by a conclusion that summarizes the project's achievements and suggests future enhancements. The appendix contains the complete source code for reproducibility.

2. Project Objectives

The primary objectives of this project are:

- To implement a complete digital audio playback system on a Xilinx Artix-7 FPGA.
- To use MATLAB and Python for pre-processing audio signals, including the application of FIR and IIR digital filters.
- To successfully convert a standard .wav audio file into a Xilinx .coe memory initialization file.
- To design and verify a Verilog module that reads audio samples from an on-chip Block RAM (BRAM).
- To implement a Pulse-Width Modulation (PWM) circuit to function as a simple Digital-to-Analog

Converter (DAC).
- To program the FPGA and validate the hardware by playing the stored audio through an external speaker or headphones.

3. Components Required

3.1. Hardware

- **FPGA Development Board:** Digilent Nexys 4 (featuring a Xilinx Artix-7 FPGA).
- **Audio Output:** 3.5mm audio jack, headphones, or an amplified speaker.
- **Connection Wires:** For connecting the FPGA pins to the audio jack.
- **Programming Cable:** USB-JTAG cable for programming the FPGA.

3.2. Software

- **FPGA Design Suite:** Xilinx Vivado Design Suite.
- **Simulation Tool:** Vivado Simulator or ModelSim.
- **Signal Processing:** MATLAB R2021a or Python 3.8 with SciPy/NumPy libraries.

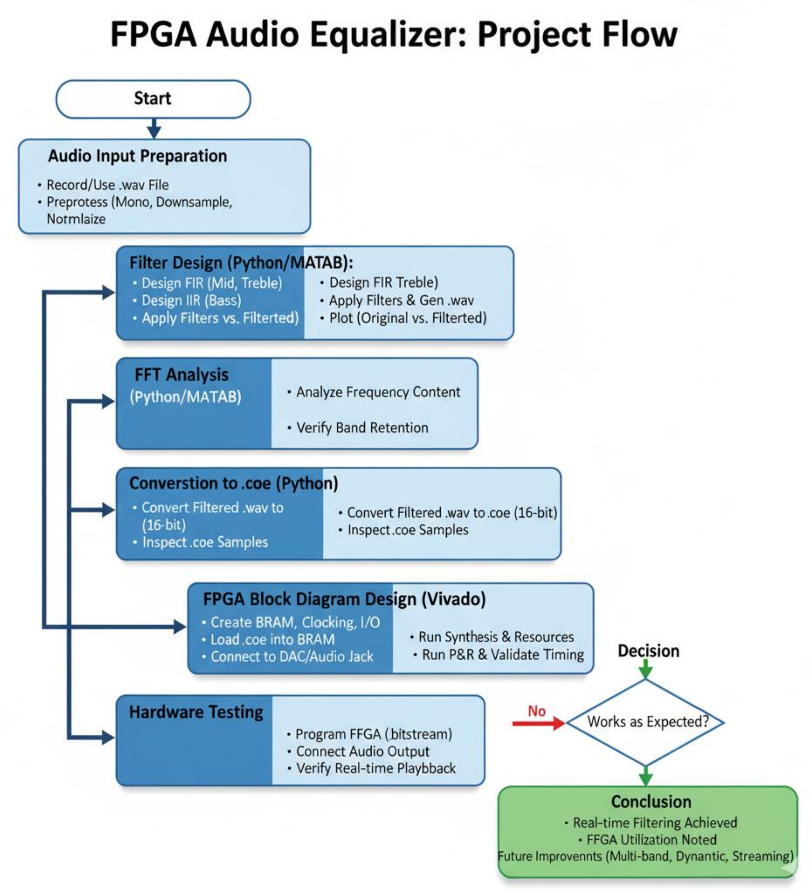
4. Project Workflow

4.1. Step-by-Step Process

The project follows a structured workflow that bridges the gap between software design and hardware implementation:

1. **Audio Filtering:** An input .wav file is first processed in MATLAB or Python. Digital FIR or IIR filters are applied to modify its frequency content (e.g., applying a low-pass filter).
2. **COE File Generation:** The filtered, 16-bit audio data is converted into a hexadecimal format and saved as a .coe file using a custom MATLAB script (wav_to_coe.m).
3. **Block RAM Initialization:** In Vivado, a Block Memory Generator IP core is created. This BRAM is configured to be initialized using the .coe file generated in the previous step.
4. **Verilog Design:** A top-level Verilog module is designed to orchestrate the playback. This module includes logic to generate a 44.1 kHz clock, an address counter to read samples from the BRAM, and a PWM generator.
5. **Synthesis and Implementation:** The Verilog design is synthesized and implemented in Vivado, which translates the hardware description into a bitstream file that can configure the FPGA.
6. **Hardware Testing:** The bitstream is downloaded to the Nexys 4 board. The PWM output pin is connected to an audio jack, and the resulting audio is played through speakers to verify the system's functionality.

4.2. Project Flowchart

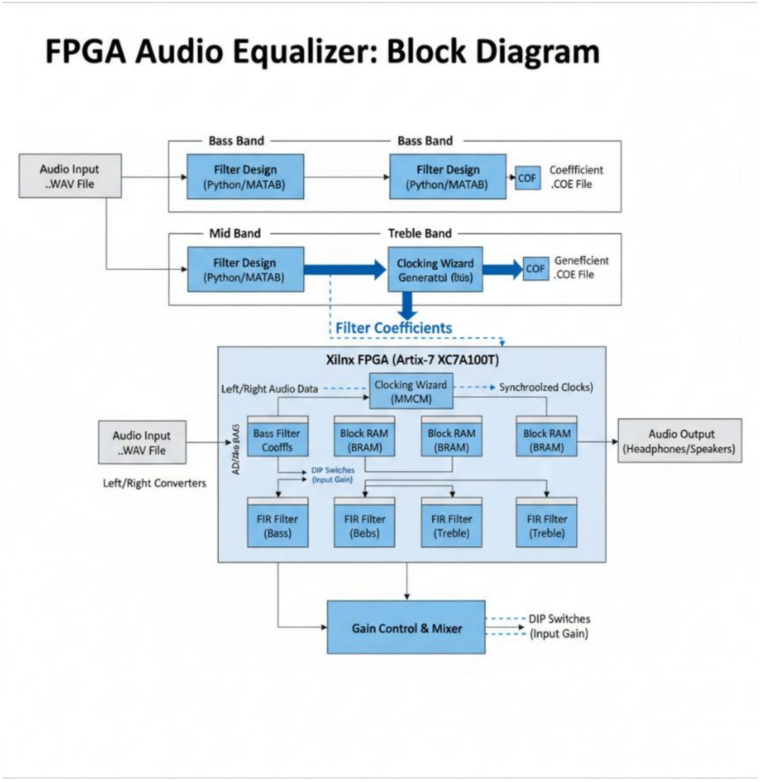

FPGA Audio Equalizer: Project Flow
Start
Audio Input Preparation
• Record/Use .wav File
• Preprotess (Mono, Downsample, Notmlaize
Filter Design (Python/MATAB):
• Design FIR (Mid, Treble)
• Design IIR (Bass)
• Apply Filters vs. Filtered)
• Design FIR Treble)
• Apply Filters & Gen .wav
• Plot (Original vs. Filterted)
FFT Analysis (Python/MATAB)
• Analyze Frequency Content
• Verify Band Retention
Converstion to .coe (Python)
• Convert Filtered .wav to (16-bit)
• Inspect .coe Samples
• Convert Filtered .wav to .coe (16-bit)
• Inspect .coe Samples
FPGA Block Diagram Design (Vivado)
• Create BRAM, Clocking, I/O
• Load .coe into BRAM
• Connect to DAC/Audio Jack
• Run Synthesis & Resources
• Run P&R & Validate Timing
Decision
No
Works as Expected?
Hardware Testing
• Program FFGA (bitstream)
• Connect Audio Output
• Verify Real-time Playblack
Conclusion
• Real-time Filtering Achieved
• FFGA Utilization Noted
Future Improvennts (Multi-band, Dynantic, Streaming)

5. Design Methodology

5.1. System Block Diagram

The hardware architecture is composed of several interconnected blocks within the FPGA, as shown in the diagram below.

- **System Clock (100 MHz):** The main clock input from the Nexys 4 board.
- **Clock Divider:** A custom logic block that scales the 100 MHz system clock down to the 44.1 kHz audio sampling rate.
- **Address Counter:** Generates a sequential address to read samples from the BRAM at the audio clock rate.
- **Block RAM (BRAM):** An on-chip memory block initialized with the .coe file containing the audio samples.
- **PWM Generator:** Takes the 8-bit audio sample from the BRAM and generates a corresponding PWM signal.
- **PWM Output:** The final digital signal sent to the audio jack.



5.2. Pulse-Width Modulation (PWM) as a DAC

A Digital-to-Analog Converter (DAC) is required to convert the digital audio samples into an analog voltage that speakers can use. This project implements a simple and resource-efficient DAC using Pulse-Width Modulation.

The principle is to generate a high-frequency square wave whose duty cycle (the percentage of time the signal is 'high') is proportional to the amplitude of the digital audio sample. For an 8-bit sample (values from 0 to 255), the duty cycle would vary accordingly:

- A sample value of **0** results in a **0%** duty cycle (signal is always low).
- A sample value of **128** results in a **~50%** duty cycle.
- A sample value of **255** results in a **100%** duty cycle (signal is always high).

When this fast-switching PWM signal is passed through a low-pass filter (which is inherent in the input stage of an amplifier or speaker), the signal is averaged out. The resulting average voltage is directly proportional to the duty cycle, effectively creating an analog representation of the original digital sample.

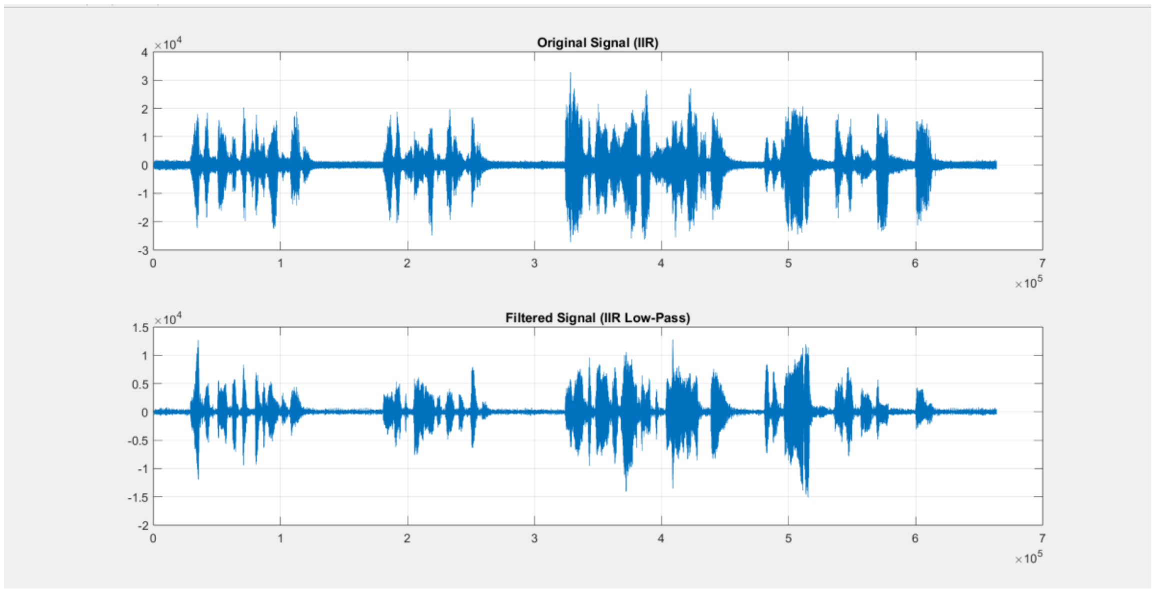
5.3. Vivado Block Design

In Vivado, the design consists of the custom Verilog module (top_audio.v) and one IP core: the **Block Memory Generator**. The IP core is configured to have a single port with an 8-bit data width and a 16-bit address width, allowing it to store up to 65,536 samples. The path to the audio_4s.coe file is specified in the IP core's configuration to initialize its contents.

6. Software Processing and Data Preparation

6.1. Audio Filtering (MATLAB/Python)

Before preparing the audio for hardware, it can be processed using powerful software tools. The provided Python scripts (fir.py and iir.py) demonstrate how digital filters can be designed and applied to a .wav file. This step allows for effects like equalization (boosting bass, cutting treble) or noise reduction to be performed. The output of this stage is a modified .wav file ready for conversion.



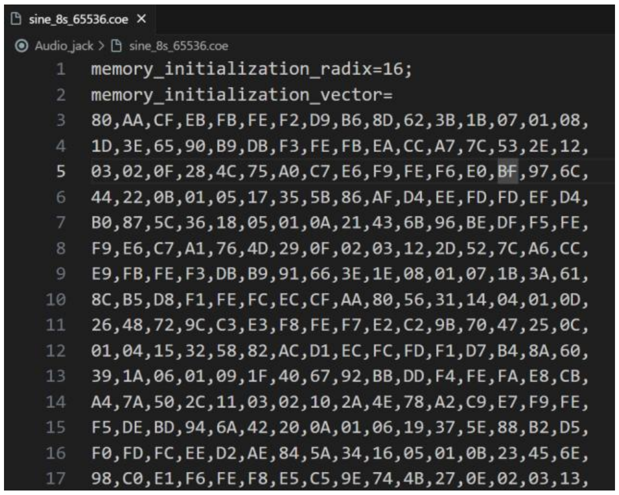
6.2. Conversion to Coefficient File (.coe)

The Xilinx BRAM IP core requires a specific text file format for initialization, known as a Coefficient File (.coe). A MATLAB script, wav_to_coe.m, was used to perform this conversion.

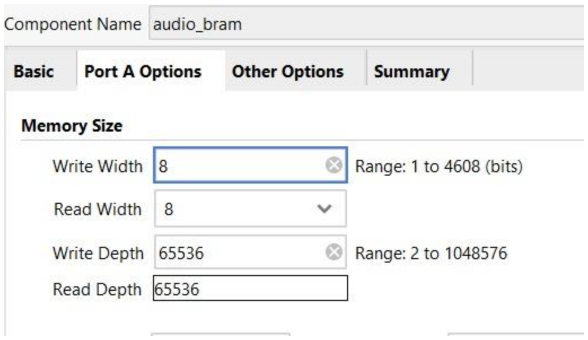
The script performs the following actions:

1. **Reads the .wav file** using the audioread function.
2. **Converts stereo to mono** by taking only the first channel if necessary.
3. **Scales the data:** audioread returns floating-point values between -1.0 and 1.0. These are scaled to the 16-bit signed integer range [-32768, 32767].
4. **Writes the COE header:** It prints the required header lines memory_initialization_radix=16; and memory_initialization_vector=.
5. **Writes the data:** It iterates through each sample, converts it to its 16-bit two's complement hexadecimal representation, and writes it to the file, followed by a comma. The last sample is followed by a semicolon.

Note: The Verilog design was later simplified to use 8-bit audio, so the .coe file would be generated with 8-bit (2-digit hex) values instead.



```
memory_initialization_radix=16;
memory_initialization_vector=
80,AA,CF,EB,FB,FE,F2,D9,B6,8D,62,3B,1B,07,01,08,
1D,3E,65,90,B9,DB,F3,FE,FB,EA,CC,A7,7C,53,2E,12,
03,02,0F,28,4C,75,A0,C7,E6,F9,FE,F6,E0,BF,97,6C,
44,22,0B,01,05,17,35,5B,86,AF,D4,EE,FD,FD,EF,D4,
B0,87,5C,36,18,05,01,0A,21,43,6B,96,BE,DF,F5,FE,
F9,E6,C7,A1,76,4D,29,0F,02,03,12,2D,52,7C,A6,CC,
E9,FB,FE,F3,DB,B9,91,66,3E,1E,08,01,07,1B,3A,61,
8C,B5,D8,F1,FE,FC,EC,CF,AA,80,56,31,14,04,01,0D,
26,48,72,9C,C3,E3,F8,FE,F7,E2,C2,9B,70,47,25,0C,
01,04,15,32,58,82,AC,D1,EC,FC,FD,F1,D7,B4,8A,60,
39,1A,06,01,09,1F,40,67,92,BB,DD,F4,FE,FA,E8,CB,
A4,7A,50,2C,11,03,02,10,2A,4E,78,A2,C9,E7,F9,FE,
F5,DE,BD,94,6A,42,20,0A,01,06,19,37,5E,88,B2,D5,
F0,FD,FC,EE,D2,AE,84,5A,34,16,05,01,0B,23,45,6E,
98,C0,E1,F6,FE,F8,E5,C5,9E,74,4B,27,0E,02,03,13,
```



Component Name: audio_bram

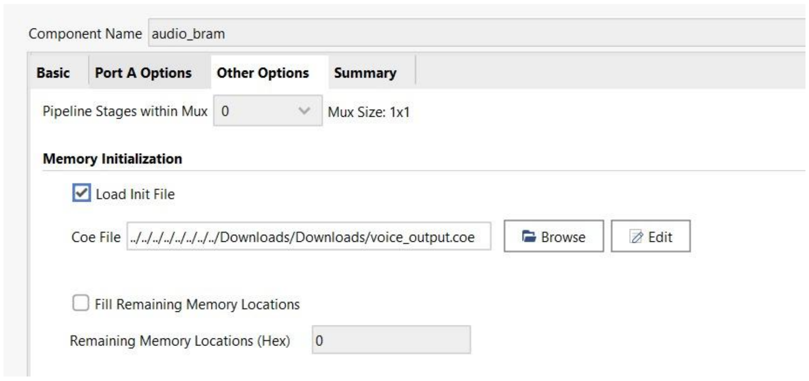
Memory Size

Write Width: 8 — Range: 1 to 4608 (bits)

Read Width: 8

Write Depth: 65536 — Range: 2 to 1048576

Read Depth: 65536



Component Name: audio_bram

Pipeline Stages within Mux: 0 — Mux Size: 1x1

Memory Initialization

[✓] Load Init File

Coe File: ../../../../../../../Downloads/Downloads/voice_output.coe

[ ] Fill Remaining Memory Locations

Remaining Memory Locations (Hex): 0

7. Verilog Hardware Design

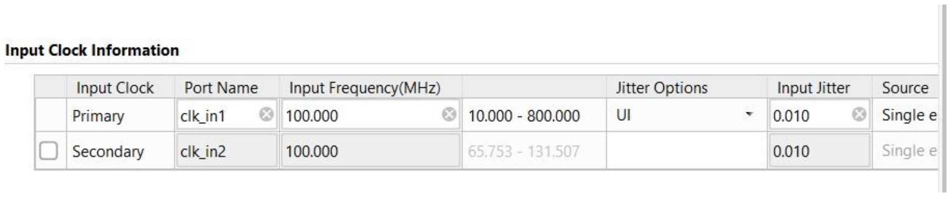
7.1. Overview of top_audio.v

The top_audio.v module is the heart of the hardware design. It integrates all the necessary logic to read from the BRAM and generate the PWM output. It is a fully synchronous design driven by the 100 MHz system clock.

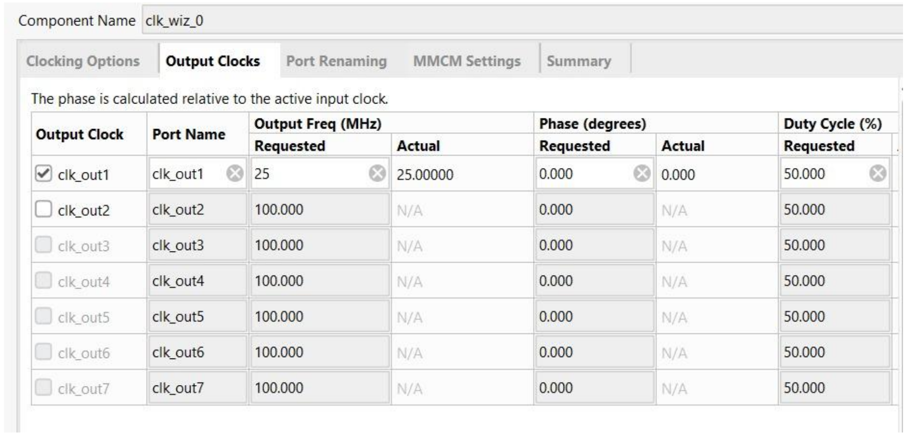
7.2. Clock Divider

To play audio at the correct speed, the system needs to read one sample from the BRAM every 1 / 44100 seconds. A clock divider is implemented to generate a 44.1 kHz clock (audio_clk_reg) from the 100 MHz master clock (clk).

The counter increments on every 100 MHz clock cycle. When it reaches DIVIDER_VALUE, the audio clock toggles, and the counter resets.


Input Clock Information

	Input Clock	Port Name	Input Frequency(MHz)		Jitter Options	Input Jitter	Source
	Primary	clk_in1	100.000	10.000 - 800.000	UI	0.010	Single e
☐	Secondary	clk_in2	100.000	65.753 - 131.507		0.010	Single e


Component Name clk_wiz_0

The phase is calculated relative to the active input clock.

Output Clock	Port Name	Output Freq (MHz)		Phase (degrees)		Duty Cycle (%)
		Requested	Actual	Requested	Actual	Requested
✓ clk_out1	clk_out1	25	25.00000	0.000	0.000	50.000
☐ clk_out2	clk_out2	100.000	N/A	0.000	N/A	50.000
☐ clk_out3	clk_out3	100.000	N/A	0.000	N/A	50.000
☐ clk_out4	clk_out4	100.000	N/A	0.000	N/A	50.000
☐ clk_out5	clk_out5	100.000	N/A	0.000	N/A	50.000
☐ clk_out6	clk_out6	100.000	N/A	0.000	N/A	50.000
☐ clk_out7	clk_out7	100.000	N/A	0.000	N/A	50.000

7.3. Block RAM Instantiation and Address Generation

The Block Memory Generator IP, named blk_mem_gen_0, is instantiated in the Verilog code. A 16-bit register, audio_addr, serves as the address bus for the BRAM. This address is incremented on every rising edge of the generated audio_clk_reg, ensuring that a new sample is fetched at the correct rate.

7.4. PWM Generator Logic

The PWM generation is achieved with a simple yet elegant piece of combinational logic. An 8-bit counter (pwm_counter) runs continuously at the full 100 MHz system clock speed. The PWM output signal (aud_pwm) is generated by comparing the value of this high-speed counter to the current audio sample value.

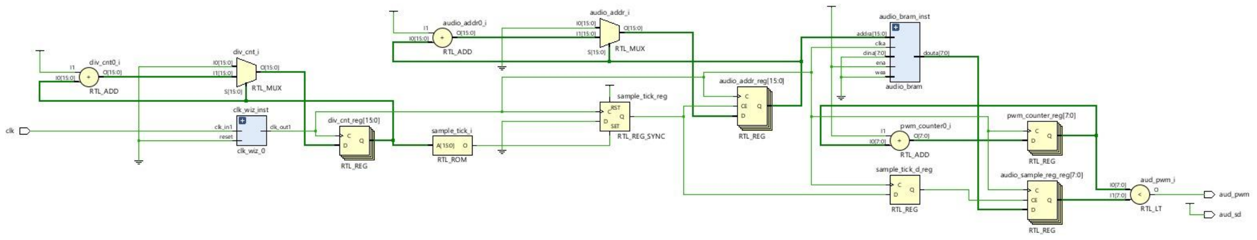
The output aud_pwm is high for the duration that pwm_counter is less than audio_sample. Since pwm_counter cycles from 0 to 255, this directly maps the 8-bit audio sample's amplitude to the PWM signal's duty cycle.

8. Results

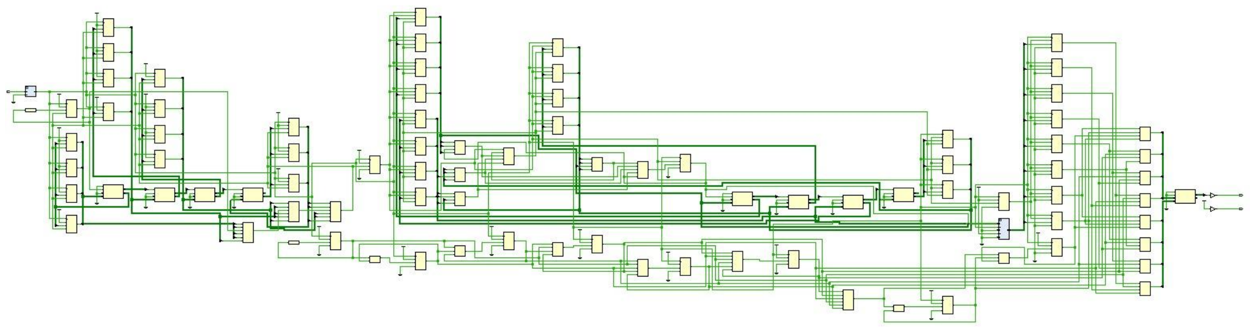
8.1. Synthesis and Implementation Results

The design was successfully synthesized and implemented for the Artix-7 device. The resource utilization was minimal, demonstrating the efficiency of the design.

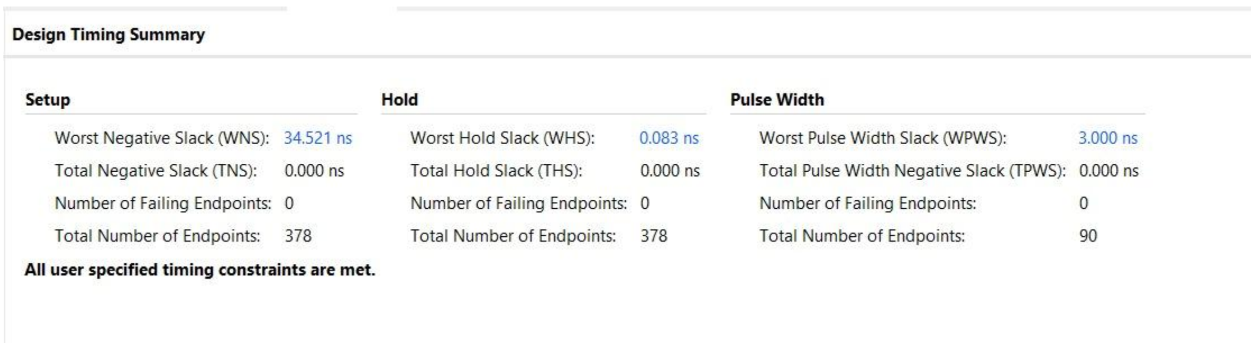
Schematic:



Synthesis:



Timing Summary:

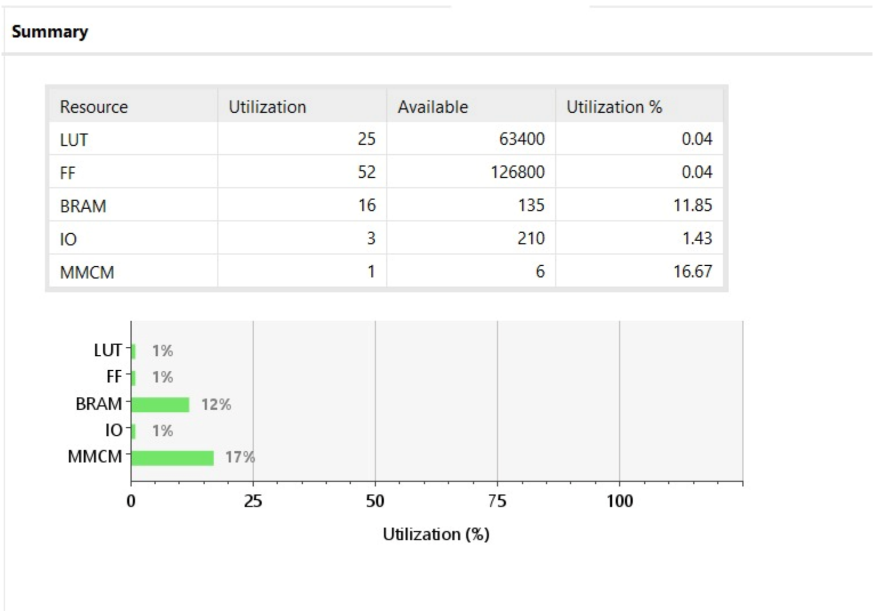


Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	34.521 ns	Worst Hold Slack (WHS):	0.083 ns	Worst Pulse Width Slack (WPWS):	3.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	378	Total Number of Endpoints:	378	Total Number of Endpoints:	90

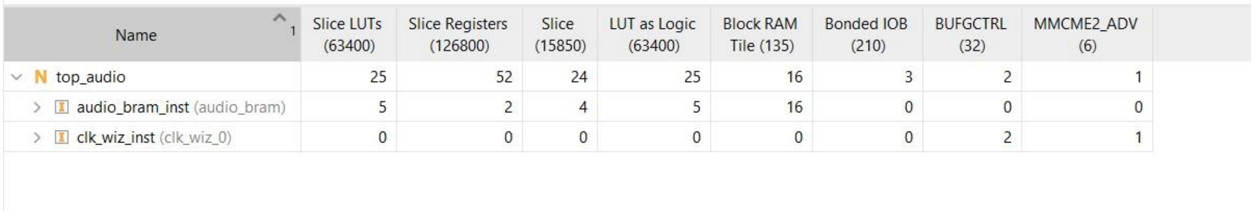
All user specified timing constraints are met.

Resource Utilization:



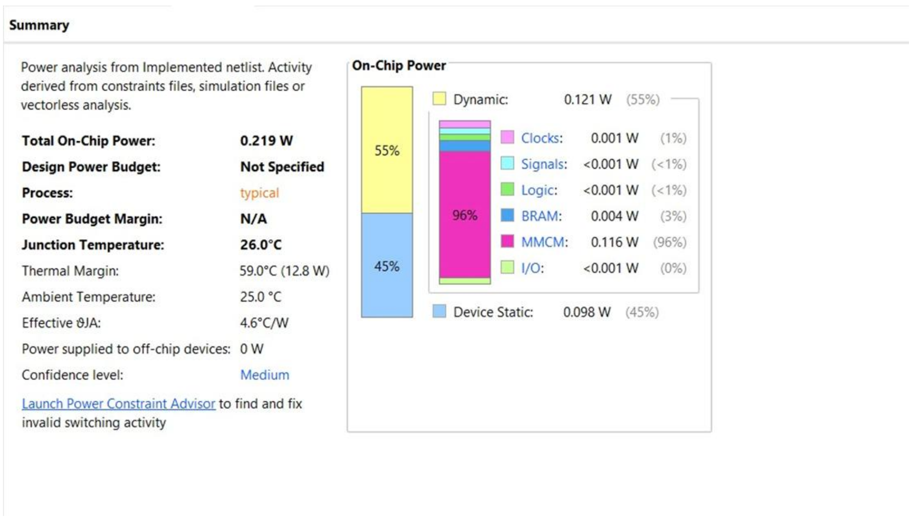
Summary

Resource	Utilization	Available	Utilization %
LUT	25	63400	0.04
FF	52	126800	0.04
BRAM	16	135	11.85
IO	3	210	1.43
MMCM	1	6	16.67



Name	Slice LUTs (63400)	Slice Registers (126800)	Slice (15850)	LUT as Logic (63400)	Block RAM Tile (135)	Bonded IOB (210)	BUFGCTRL (32)	MMCME2_ADV (6)
top_audio	25	52	24	25	16	3	2	1
audio_bram_inst (audio_bram)	5	2	4	5	16	0	0	0
clk_wiz_inst (clk_wiz_0)	0	0	0	0	0	0	2	1

Power Usage:



Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.219 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**26.0°C**
Thermal Margin:	59.0°C (12.8 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	0.121 W	(55%)
Clocks:	0.001 W	(1%)
Signals:	<0.001 W	(<1%)
Logic:	<0.001 W	(<1%)
BRAM:	0.004 W	(3%)
MMCM:	0.116 W	(96%)
I/O:	<0.001 W	(0%)
Device Static:	0.098 W	(45%)

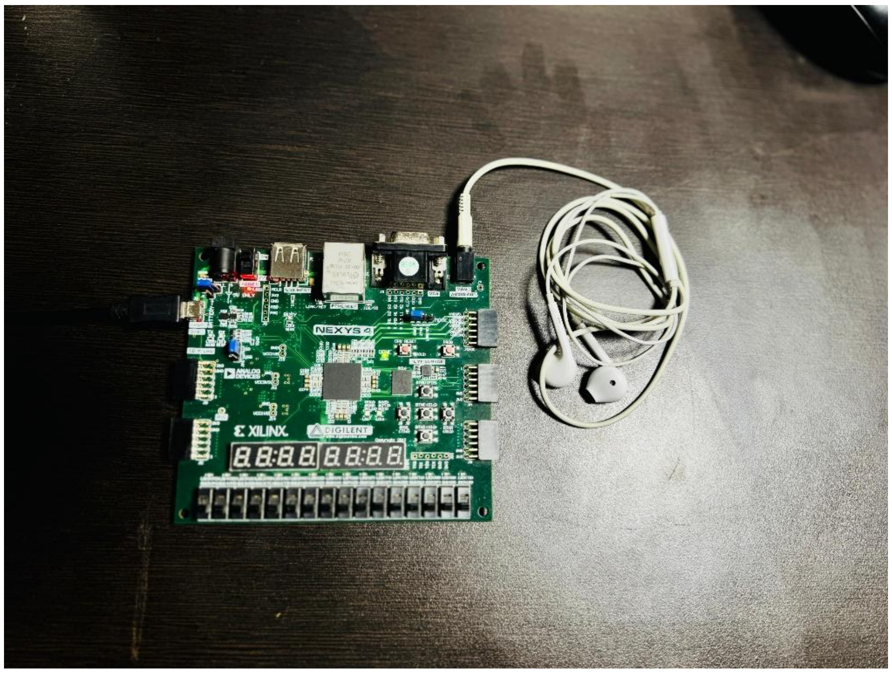
The results show that the design is extremely lightweight and occupies a tiny fraction of the FPGA's available resources.

9. Hardware Implementation and Testing

9.1. Hardware Setup

The hardware was set up as follows:

1. The Digilent Nexys 4 board was connected to the host computer via the USB-JTAG programming port.
2. A 3.5mm audio jack was connected to the FPGA. The tip of the jack was connected to the aud_pwm output pin, and the sleeve was connected to a ground (GND) pin.
3. A Xilinx Design Constraints (.xdc) file was created to map the Verilog ports (clk, aud_pwm, aud_sd) to the physical pins of the Nexys 4 board.
4. Headphones or an amplified speaker were plugged into the audio jack.



9.2. Programming and Verification

The generated bitstream was downloaded to the FPGA using the Vivado Hardware Manager. Immediately upon successful programming, the Verilog module began executing. The audio stored in the BRAM started playing through the connected headphones.

The output was clear, and the melody was easily recognizable, confirming that the entire workflow—from MATLAB data conversion to Verilog PWM generation—was successful. The audio looped back to the beginning after reaching the end of the BRAM content, as designed.

10. Conclusion

10.1. Summary of Achievements

This project successfully achieved all its primary objectives. A complete, standalone digital audio playback system was implemented on an FPGA. The project demonstrated:

- A robust workflow for converting software-processed audio into a hardware-compatible format.
- The successful use of on-chip Block RAM for storing digital media samples.
- The design of a precise clock divider for generating standard audio sampling rates.
- The effective implementation of Pulse-Width Modulation as a low-cost, resource-efficient Digital-to-Analog Converter.
- The final hardware test validated the entire system, producing clear and correct audio output.

11. Appendix: Source Code

A.1. Verilog Code (top_audio.v)

```
module top_audio (
    input wire clk,       // Nexys 4's 100 MHz system clock
    output wire aud_pwm,   // PWM output to the audio amplifier
    output wire aud_sd     // Audio amplifier shutdown/enable pin
);

    // --- Internal Signal Declarations ---
    // These signals connect the different logic blocks and the IP core.
    reg [15:0] audio_addr = 0;       // 16-bit address counter for the BRAM
    wire [7:0] audio_sample;         // 8-bit audio data read from the BRAM

    reg audio_clk_reg = 0;           // A slow clock signal for sampling
audio
    reg [31:0] clk_divider_counter = 0; // Counter to generate the audio
clock

    reg [7:0] pwm_counter = 0;       // Counter for the PWM generator
    // --- Parameters for Clock Division ---
    // Calculates the division ratio to get a 44.1 kHz sample rate from 100
MHz.
    // The division value is for a toggling clock, so we divide by 2.
    parameter AUDIO_SAMPLE_RATE = 44100;
    parameter SYSTEM_CLOCK_RATE = 100_000_000;
    parameter DIVIDER_VALUE = (SYSTEM_CLOCK_RATE / AUDIO_SAMPLE_RATE) / 2;

    // --- Instance of the Block Memory IP ---
    // This is where your IP core is connected to the top-level module.

   blk_mem_gen_0 blk_mem_gen_instance (
  .clka(audio_clk_reg),    // input wire clka
  .ena(1'b1),      // input wire ena
  .wea(1'b0),      // input wire [0 : 0] wea
  .addra(audio_addr),  // input wire [15 : 0] addra
  .dina(8'd0),    // input wire [7 : 0] dina
  .douta(audio_sample)  // output wire [7 : 0] douta
);

    // --- Behavioral Logic ---
    // All the sequential logic driven by the main system clock.
    always @(posedge clk) begin
        // Clock Divider Logic: Generates the 44.1 kHz audio clock.
        if (clk_divider_counter == DIVIDER_VALUE) begin
            audio_clk_reg <= ~audio_clk_reg;
            clk_divider_counter <= 0;
        end else begin
            clk_divider_counter <= clk_divider_counter + 1;
        end
        // Address Counter Logic: Reads samples from the BRAM on each audio
clock edge.
        if (audio_clk_reg) begin
            audio_addr <= audio_addr + 1;
            // Loop the audio by resetting the address at the end of the
memory.
            // 65536 samples is 16'hFFFF
            if (audio_addr == 16'hFFFF) begin
                audio_addr <= 16'h0000;
            end
        end

        // PWM Counter Logic: Continuously counts up to 255.
        // This counter is the basis for the pulse width.
        pwm_counter <= pwm_counter + 1;
    end

    // --- Combinational Logic ---
    // These assignments are always active and happen instantaneously.

    // PWM Output: The heart of the DAC.
    // The PWM signal is high as long as the PWM counter value is less than
the audio sample value.
    assign aud_pwm = (pwm_counter < audio_sample);

    // Audio Amplifier Enable: Drives the `aud_sd` pin high to enable the
audio output.
    assign aud_sd = 1'b1;

endmodule
```

A.2. MATLAB COE File Generator (wav_to_coe.m)

```
% Input file (full path)
[data, fs] = audioread('C:\Verilog
Labs\FPGA\Audio_jack\audio_4s.wav');

% If stereo, take only one channel
if size(data,2) > 1
    data = data(:,1);
end

% === Convert to 16-bit signed integer ===
% audioread gives normalized [-1,1], so scale to int16 range [-
32768, 32767]
data16 = int16(data * (2^15 - 1));

% Output file (same folder as input)
fid = fopen('C:\Verilog Labs\FPGA\Audio_jack\audio_4s.coe','w');

if fid == -1
    error('? Cannot create COE file. Check path or
permissions.');
end

% Write COE header
fprintf(fid, 'memory_initialization_radix=16;\n');
fprintf(fid, 'memory_initialization_vector=\n');

% Write samples in hex (4 hex digits = 16 bits)
for i = 1:length(data16)
    v = data16(i);
    if v < 0
        v = v + 2^16;  % convert to two's complement
    end
    fprintf(fid, '%04X', v);   % 4-digit uppercase hex

    if i ~= length(data16)
        fprintf(fid, ',\n');
    else
        fprintf(fid, ';\n');
    end
end

fclose(fid);
disp('? 16-bit COE file "audio16.coe" generated successfully');
```

A.3. Python FIR Filter Script (fir.py)

```
import numpy as np
import scipy.signal as signal
import scipy.io.wavfile as wavfile
# Matplotlib is no longer needed in this script, as MATLAB will do the
plotting.

def process_audio_and_return_for_plot():
    """
    This function contains your original, unchanged logic for
processing the audio file.
    It now returns the original and filtered data so MATLAB can plot
it.
    """

    # --- YOUR ORIGINAL CODE STARTS HERE (LOGIC IS UNCHANGED) ---

    input_path = r'C:\Verilog
Labs\FPGA\Real_Time_3Band_Audio_Equalizer\samples\orginal_samples_for_
testing\audio.wav'

    # Read the input .wav file
    sample_rate, data = wavfile.read(input_path)

    # Create own sample (This code was present but commented out in
your original script)
    #sample_rate = 44100
    #freq_cl = 1000
    #freq_ns = 150
    #t = np.linspace(0, 1.0, sample_rate)
    #data = 10*np.sin(2*freq_cl*np.pi*t) + 5*np.cos(2*freq_ns*np.pi*t)
    # Design a FIR bandpass filter using the Hamming window
    numtaps = 30  # Number of filter taps (adjust as needed)
    low_cutoff = 4000  # Low cutoff frequency in Hz
    high_cutoff = 20000  # High cutoff frequency in Hz
    nyquist_rate = sample_rate / 2.0
    cutoff = [low_cutoff / nyquist_rate, high_cutoff / nyquist_rate]
    fir_coeff = signal.firwin(numtaps, cutoff, window='hamming',
pass_zero='bandpass')

    # Print the coefficients
    for i in range (0, numtaps):
        print(fir_coeff[i])

    print(f'Writing file successfully')

    # Apply the filter to the data
    filtered_data = signal.lfilter(fir_coeff, 1.0, data)

    output_path = r'C:\Verilog
Labs\FPGA\Real_Time_3Band_Audio_Equalizer\samples\Matlab_Output\audio.
wav'

    # Write the filtered data to a new .wav file
    wavfile.write(output_path, sample_rate,
filtered_data.astype(np.int16))

    # --- YOUR ORIGINAL CODE ENDS HERE ---

    # Return the data needed for plotting
    return data, filtered_data
```

A.4. Python IIR Filter Script (iir.py)

```
import numpy as np
import scipy.signal as signal
import scipy.io.wavfile as wavfile
# Matplotlib is no longer needed as MATLAB will handle the
plotting.

def process_iir_and_return_for_plot():
    """
    This function contains your original, unchanged logic for the
IIR filter.
    It returns the original and filtered data so MATLAB can
create the plot.
    """

    # --- YOUR ORIGINAL CODE STARTS HERE (LOGIC IS UNCHANGED) ---

    input_path = r'C:\Verilog
Labs\FPGA\Real_Time_3Band_Audio_Equalizer\samples\orginal_samples
_for_testing\audio.wav'

    # Read the input .wav file
    sample_rate, data = wavfile.read(input_path)

    print(f'sample rate = {sample_rate}')

    # Design an IIR lowpass filter
    numtaps = 4  # Number of filter taps
    high_cutoff = 500  # High cutoff frequency in Hz
    nyquist_rate = sample_rate / 2.0
    cutoff = high_cutoff / nyquist_rate
    b, a = signal.iirfilter(numtaps, cutoff, btype='lowpass',
analog=False, ftype='butter')

    print(f'Writing file successfully')

    # Apply the filter to the data
    filtered_data = signal.lfilter(b, a, data)

    print(f'a = {a}')
    print(f'b = {b}')

    output_path = r'C:\Verilog
Labs\FPGA\Real_Time_3Band_Audio_Equalizer\samples\Matlab_Output\i
ir_audio.wav'

    # Write the filtered data to a new .wav file
    wavfile.write(output_path, sample_rate,
filtered_data.astype(np.int16))

    # --- YOUR ORIGINAL CODE ENDS HERE ---

    # Return the data needed for plotting
    return data, filtered_data
```

A.5. Python FFT Analysis Script (view_fft.py)

```
import numpy as np
# Matplotlib is no longer needed as MATLAB will handle the plotting.

def calculate_fft_and_return_for_plot():
    """
    This function reads a .hex file, calculates the FFT,
    and returns the frequency and amplitude data for MATLAB to plot.
    """
    # --- YOUR ORIGINAL CODE STARTS HERE (LOGIC IS UNCHANGED) ---

    # Path to the .hex file you want to analyze
    # This path is updated to match your project structure
    hex_path = r'C:\Verilog
Labs\FPGA\Real_Time_3Band_Audio_Equalizer\samples\orginal_samples_for_t
esting\audio.hex'

    # Read the hex file
    with open(hex_path, 'r') as file:
        hex_data = file.read().splitlines()

    # Convert hex data to integers
    data = [int(x, 16) for x in hex_data]

    # Ensure the data is a numpy array
    data = np.array(data)

    # Compute the FFT
    sample_rate = 44100  # Given sample rate
    n = len(data)
    fft_data = np.fft.fft(data)
    fft_data = np.abs(fft_data)[:n//2]  # Take the positive half of the
FFT

    # Create the frequency axis
    frequencies = np.fft.fftfreq(n, d=1/sample_rate)[:n//2]

    # --- YOUR ORIGINAL CODE ENDS HERE ---
```

```
        # Return the data needed for plotting
        return frequencies, fft_data
```

A.6. Xilinx Design Constraints File (top_audio.xdc)

```
## Main System Clock (100 MHz from pin E3)
set_property PACKAGE_PIN E3 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]

## Audio Interface Pins for Nexys 4
# These pins connect to the onboard audio amplifier and jack
set_property PACKAGE_PIN A11 [get_ports aud_pwm]
set_property IOSTANDARD LVCMOS33 [get_ports aud_pwm]

set_property PACKAGE_PIN D12 [get_ports aud_sd]
set_property IOSTANDARD LVCMOS33 [get_ports aud_sd]
```