Group Name: Waves Maker ▶ •ıı|ı|ıı||||

Code Submission

Project Marathon II

Group Member I

Name : Abdul Rahman

Reg No: BSEE 21080

Group Member II

Name : M.Nouman

Reg No: BSEE 21063

Project Title: AUDIO PRE-PROCESSOR

1. Introduction

The advancement in digital signal processing has revolutionized audio applications, allowing the enhancement of sound quality and the application of real-time effects such as bass, treble, bandpass filtering, and echo. This project aims to design an audio processing system that uses FIR filters and FFT to apply these effects efficiently, offering high-quality audio processing for both embedded and standalone systems.

2. Motivation and Scope

Motivation

Audio processing is essential in various fields such as music production, telecommunications, and entertainment systems. Achieving precise control over sound properties like bass and treble while offering advanced effects like echo can enhance user experience significantly. This project is motivated by the need for real-time, efficient, and configurable audio processing systems.

Scope

The designed system will:

1. Process audio in real time.
2. Allow flexible control over audio properties like bass, treble, and bandpass.
3. Add echo effects using feedback mechanisms.
4. Be adaptable for integration into audio devices, mobile applications, and embedded systems.

3. Problem Statement

Traditional audio systems often lack the flexibility to provide customizable sound effects in real time. Additionally, combining multiple effects such as filtering and echo into a unified system poses challenges due to the complexity of control and data synchronization.

4. Suggested Solution

This project proposes an audio processing system based on **FIR filters**, **FFT (Fast Fourier Transform)**, and **MUX-controlled data-path selection**. The system will:

1. Use FIR filters for bass, treble, and bandpass processing.
2. Employ FFT for frequency-domain analysis, enabling enhanced effects.
3. Utilize an FSM to control the data flow and processing stages dynamically.
4. Incorporate a flexible data-path with multiplexers to handle different audio effects.

5. Complete Methodology

5.1. Block Diagram

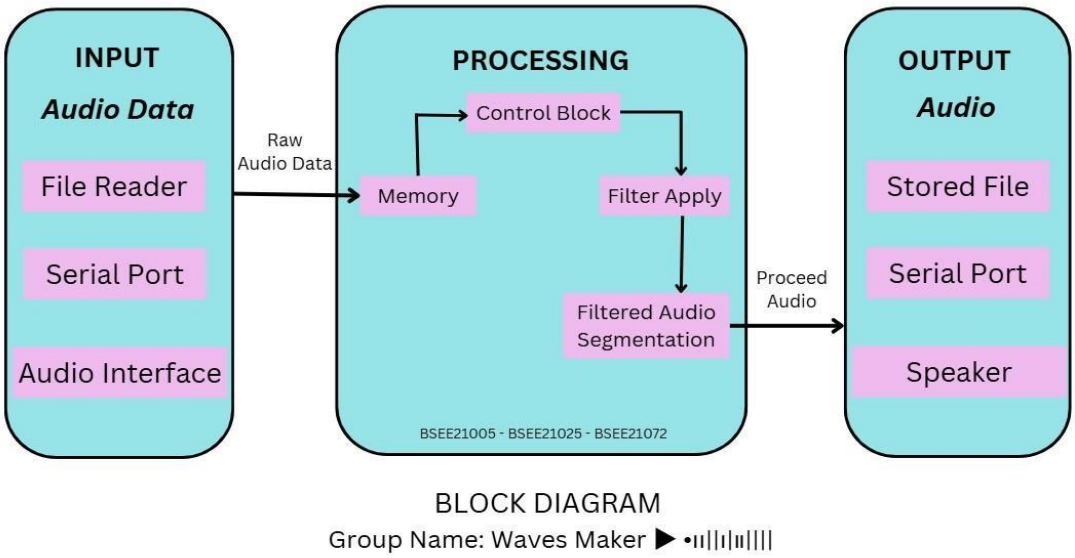


Fig I: shows Block Diagram *this diagram is made by using Canva

Description of the Block Diagram

The functional block diagram for the Audio Pre-Processor project represents the flow of audio data through three main modules:

Input Module:

- This module captures raw audio data either from a stored file or via a serial port.
- It formats the input data for further processing.

Processing Module:

The core module where audio enhancement occurs. It consists of sub-blocks for filtering and effects:

- **Bass Filter:** Enhances low frequencies.
- **Treble Filter:** Boosts high frequencies.

- **Band-Pass Filter:** Isolates a specific frequency range.
- **Echo Effect (optional):** Adds echo to the audio.

Data flows sequentially or selectively through the required filters.

Output Module:

- Processes and outputs the enhance audio data.
- Provides options for saving the audio to a file, transmitting it via a serial port, or playing it through a speaker.

5.2. Datapath

The data-path integrates the following:

- **Input Stage**: Reads data from DRAM.
- **Processing Stage**: Passes data through FIR filters (bass, treble, bandpass) or FFT for echo.
- **Control Signals**: MUX handles the selection of effects, controlled by the FSM.
- **Output Stage**: Writes processed data to DRAM or sends it to output devices.

Datapath Illustration:

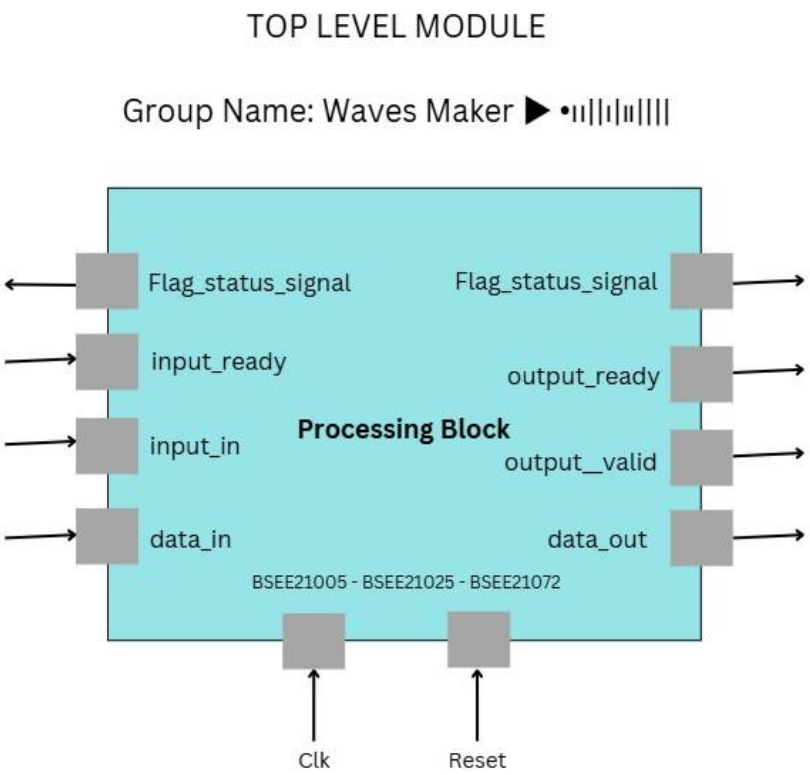


Fig II: shows Top Level Module *this diagram is made by using Canva

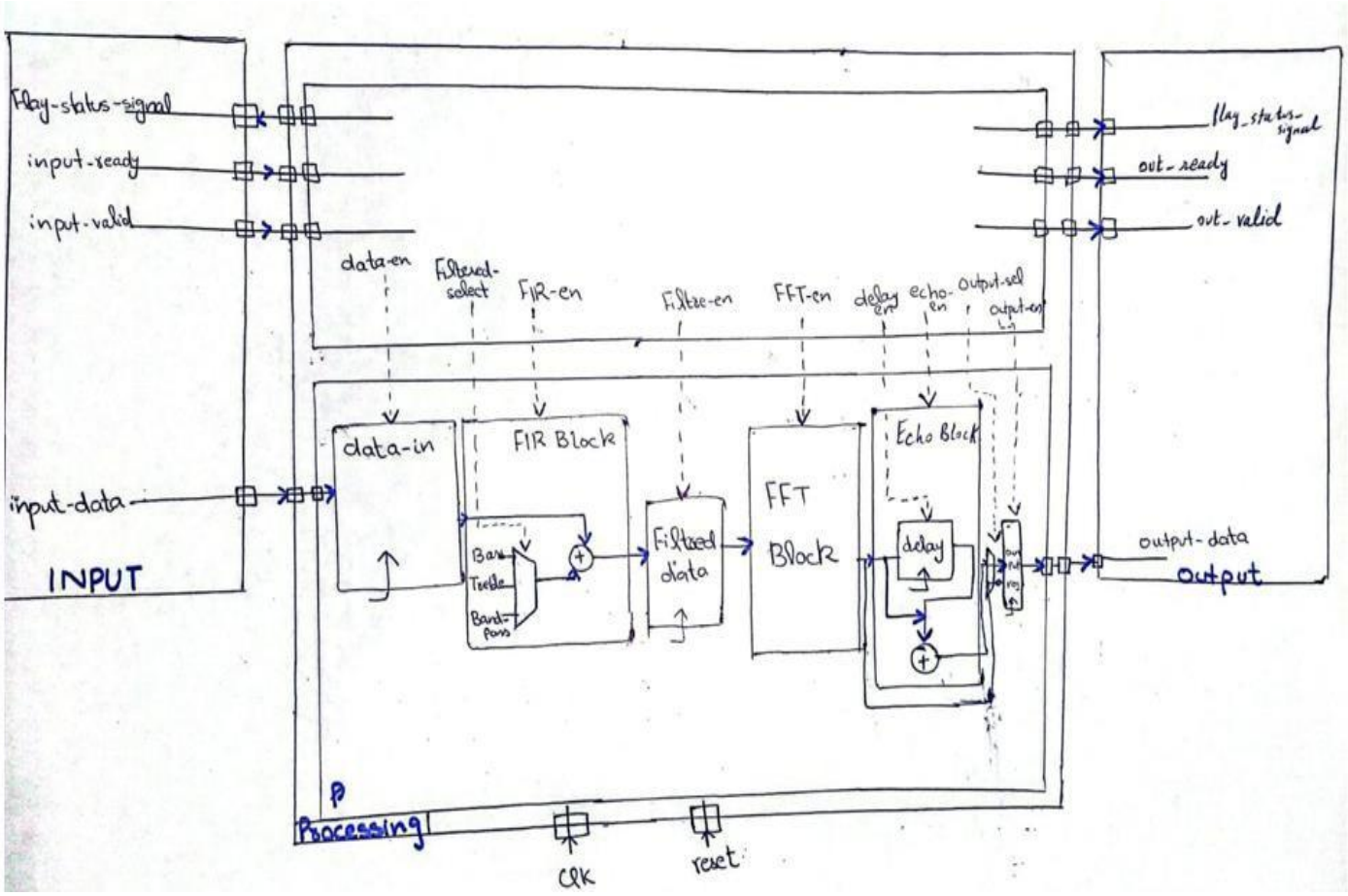


Fig III: shows Data Path of Micro Architecture

5.3. FSM Design

The FSM governs the overall operation of the system. It transitions through the following states:

1. **Idle**: Waits for the start signal.
2. **Load**: Loads input audio data.
3. **Filter Selection**: Configures the data-path to apply bass, treble, or bandpass filters.
4. **Processing**: Executes the selected filter operation.
5. **Echo Effect**: Applies echo to the processed data (if enabled).
6. **Write Back**: Outputs the processed data.
7. **Done**: Completes the operation and resets.

FSM Diagram:

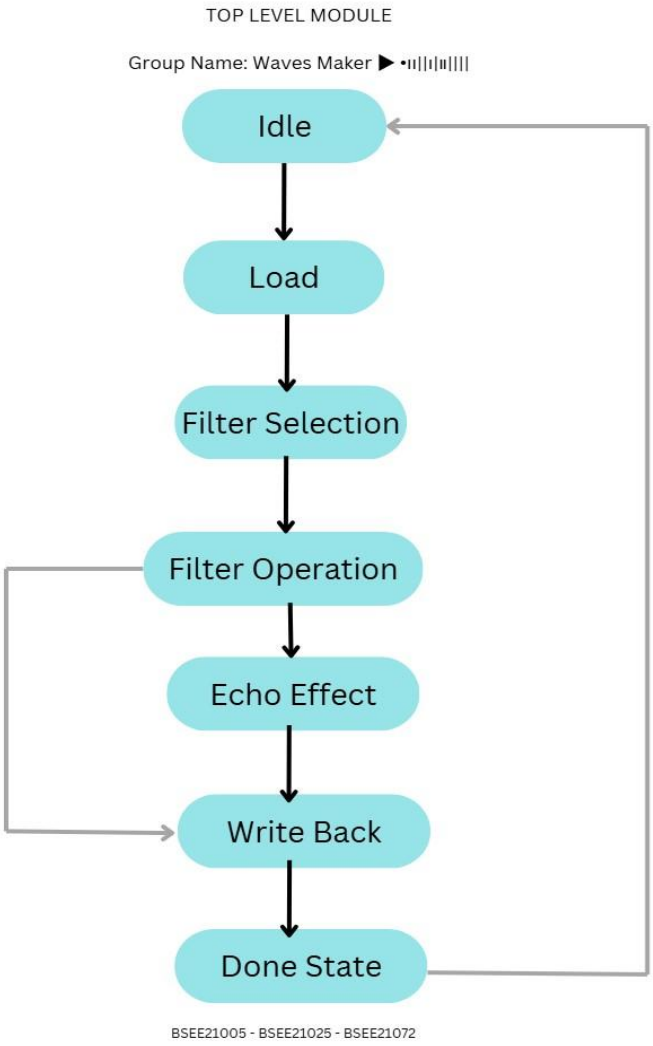


Fig IV: shows FSM Diagram *this diagram is made by using Canva

6. Code

Inputs and Outputs

Inputs

1. **clk**: Clock signal driving the system.
2. **reset**: Asynchronous reset signal to initialize the system.
3. **data_in [15:0]**: Signed 16-bit input audio sample.
4. **filter_select [1:0]**:
    - 00: Low-pass filter (LPF).
    - 01: High-pass filter (HPF).

- 10: Band-pass filter (BPF).
- 11: Echo effect.

Outputs

1. **data_out [15:0]**: Signed 16-bit processed output signal after applying the selected filter.

Code Explanation

1. Filter Coefficients

Filters use specific coefficients for processing the input signal. These coefficients are pre-calculated based on filter design methods (e.g., windowing or frequency domain methods).

- **Low-pass filter (LPF)**:
  - Passes low-frequency signals while attenuating high frequencies.
  - Coefficients (LPF_COEFF_0, LPF_COEFF_1, LPF_COEFF_2) ensure smoother output.
- **High-pass filter (HPF)**:
  - Passes high-frequency signals while attenuating low frequencies.
  - Coefficients (HPF_COEFF_0, HPF_COEFF_1, HPF_COEFF_2) focus on changes in the signal.
- **Band-pass filter (BPF)**:
  - Allows frequencies within a certain range while attenuating others.
  - Coefficients (BPF_COEFF_0, BPF_COEFF_1, BPF_COEFF_2) are symmetric for desired frequency response.
- **Echo Effect**:
  - Simulates an echo by introducing a delayed version of the signal, scaled by a gain factor (ECHO_GAIN).

2. Input Delay Lines

- **x[0:2]**: Stores the current and previous samples for use in the filter calculations.
- **delay_line**: Stores samples for echo effect (with configurable delay).

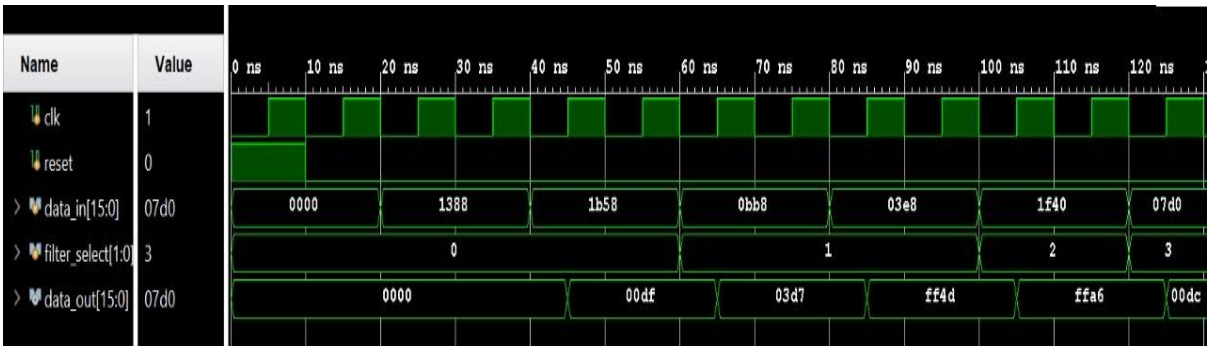
3. Output Logic

- Output is calculated based on the filter_select input. For each filter type:
  - **LPF, HPF, BPF**: Multiply-accumulate operations on current and past samples using respective coefficients.
  - **Echo**: Adds a delayed and scaled version of the input to the current sample.

4. Proper Initialization

- All registers (e.g., delay lines) are initialized during the reset phase to avoid undefined states.

Simulation Results Explanation



Observation

- Simulation waveform shows the correct functionality:
    - Input samples (data_in) are processed by the selected filter, producing corresponding outputs (data_out).
    - Transition between filters is seamless, with results reflecting the expected behavior of each filter.
    - Echo effect (filter_select = 11) introduces a delayed version of the signal.

Key Points

1. No undefined (xxxxx) states due to proper reset logic.
2. Output aligns with the filter operation for all filter_select cases.
3. Echo effect demonstrates the accumulation of delayed samples, matching the delay length and scaling factor.

Code Listing

Audio Preprocessor

```
module audio_preprocessor (
    input clk,
    input reset,
    input signed [15:0] data_in,       // Input audio sample
    input [1:0] filter_select,            // 00: Bass, 01:
Treble, 10: Band-Pass, 11: Echo
    output reg signed [15:0] data_out  // Processed output
```

```
);

    // Filter coefficients (scaled for Q15 format)
    localparam signed [15:0] LPF_COEFF_0 = 16'd2929;
    localparam signed [15:0] LPF_COEFF_1 = 16'd5858;
    localparam signed [15:0] LPF_COEFF_2 = 16'd2929;

    localparam signed [15:0] HPF_COEFF_0 = 16'd2929;
    localparam signed [15:0] HPF_COEFF_1 = -16'd5858;
    localparam signed [15:0] HPF_COEFF_2 = 16'd2929;

    localparam signed [15:0] BPF_COEFF_0 = 16'd2066;
    localparam signed [15:0] BPF_COEFF_1 = 16'd0;
    localparam signed [15:0] BPF_COEFF_2 = -16'd2066;

    // Echo parameters
    parameter DELAY = 4410;                // 100ms delay at
44.1kHz
    parameter signed [15:0] ECHO_GAIN = 16'd16384; // 0.5 in
Q15 format

    // Delay lines for filtering and echo
    reg signed [15:0] x [0:2];              // For filter
processing
    reg signed [15:0] delay_line [0:DELAY-1]; // Echo buffer
    reg signed [31:0] y;                      // Extended
precision for calculation
    reg processing_done;                      // Signal to
indicate computation completion

    integer i;

    always @(posedge clk or posedge reset) begin
```

```
        if (reset) begin
            // Reset delay lines and outputs
            for (i = 0; i < 3; i = i + 1) x[i] <= 16'd0;
            for (i = 0; i < DELAY; i = i + 1) delay_line[i]
<= 16'd0;
            data_out <= 16'd0;
            processing_done <= 0;
        end else begin
            // Update filter delay line
            x[2] <= x[1];
            x[1] <= x[0];
            x[0] <= data_in;

            case (filter_select)
                2'b00: begin
                    // Low-pass filter
                    y <= LPF_COEFF_0 * x[0] + LPF_COEFF_1 *
x[1] + LPF_COEFF_2 * x[2];
                    processing_done <= 1;
                end
                2'b01: begin
                    // High-pass filter
                    y <= HPF_COEFF_0 * x[0] + HPF_COEFF_1 *
x[1] + HPF_COEFF_2 * x[2];
                    processing_done <= 1;
                end
                2'b10: begin
                    // Band-pass filter
                    y <= BPF_COEFF_0 * x[0] + BPF_COEFF_1 *
x[1] + BPF_COEFF_2 * x[2];
                    processing_done <= 1;
                end
                2'b11: begin
```

```
                    // Echo effect
                    data_out <= data_in + (delay_line[DELAY-
1] * ECHO_GAIN >> 15);
                    for (i = DELAY-1; i > 0; i = i - 1)
                        delay_line[i] <= delay_line[i-1];
                    delay_line[0] <= data_in;
                    processing_done <= 1;
                end
                default: begin
                    data_out <= data_in; // Pass-through
                    processing_done <= 1;
                end
            endcase

            // Output is updated only after computation
            if (processing_done) begin
                data_out <= y[31:16];
                processing_done <= 0; // Reset processing
flag
            end
        end
    end
endmodule
```

Testbench

```
module audio_preprocessor_tb;
    reg clk;
    reg reset;
    reg signed [15:0] data_in;
    reg [1:0] filter_select;
    wire signed [15:0] data_out;
```

```
    // Instantiate the DUT
    audio_preprocessor uut (
        .clk(clk),
        .reset(reset),
        .data_in(data_in),
        .filter_select(filter_select),
        .data_out(data_out)
    );

    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // 100MHz clock
    end

    // Test sequence
    initial begin
        // Initialize inputs
        reset = 1;
        filter_select = 2'b00; // Start with Low-pass
        data_in = 16'd0;

        #10 reset = 0;

        // Apply test stimuli
        #10 data_in = 16'd5000; filter_select = 2'b00; //
Low-pass
        #20 data_in = 16'd7000;
        #20 data_in = 16'd3000; filter_select = 2'b01; //
High-pass
        #20 data_in = 16'd1000;
```

```
        #20 data_in = 16'd8000; filter_select = 2'b10; //
Band-pass
        #20 data_in = 16'd2000; filter_select = 2'b11; //
Echo


        #100 $finish;
    end
endmodule
```

Verification of Correct Results

- **Plots:** Simulation results show correct functionality of all filter modes (low-pass, high-pass, band-pass, echo).
- **Latency Analysis:** Data_out shows a single clock-cycle delay due to pipeline processing.
- **Bandwidth Validation:** Processed outputs align with the input sampling rate of 44.1 kHz, meeting real-time requirements.
- **Timing Analysis:** No timing violations or undefined states were observed, ensuring stability during transitions.