Digital Systems : Week1 Report

Audio Signal Processing & Adaptive Noise Cancellation Using Digital Filters

Team members : Nishi Shah | Nishant Kumar| Mumuksh Jain | Jiya Desai

Date : 13/03/2024

Problem Objective

Signals obtained from sensors commonly contain significant noise. To eliminate this noise, one can either utilize improved sensors/probes or implement post-acquisition noise filtering. In this project, we will try reading a noisy signal acquired from a sensor. Our objective is to employ digital filters on an FPGA to effectively eliminate the noise from the signal.
In particular, we will use audio signals as our input signal. We will try implementing basic digital filters - low pass, high pass, band pass, all pass and band stop - on an FPGA, before attempting to design an adaptive noise cancellation filter.
Adaptive Noise Cancellation (ANC) reduces unwanted background noise from audio in real-time. It uses microphones to capture desired signals and noise, adjusting an adaptive filter to predict and cancel noise. The error signal refines the filter, enhancing the desired audio signal in noisy environments.

Weekly breakdown of tasks

1. Convert input sound signal from Analog to Digital.

2. Add noise through a natural source or generate it artificially using any known method. specify each technique used and why.

3. Using filters like low pass filter, high pass filter, bandpass filter, bandstop filter, all pass filter and wiener filter extract the original sound signal back. You can also use filters of your choice. For example: Butterworth filter. State the reasoning properly for each step.

4. For verifying your approach, you can use inbuilt .wav files in MATLAB and write code in MATLAB and test it. Final code must be written in Verilog.

5. If time permits, design filters to extract sound notes from input sound signals. For example: detect if a C note is present in some sound signal.

Task 1: Read Data from BRAM and write data into BRAM of Basys 3 FPGA

Part 1: We learnt to read data from a file and displayed it.

```verilog
module file_read();
   reg [5:0] array1 [0:3];
   reg [7:0] array2 [0:5];
   integer i;
   initial begin
       // All the values from file will get into array1
       $readmemb("my_file.txt", array1);
       // Only the first 4 values from file will get into array2
       $readmemh("my_file.txt", array2, 0, 3);
       for (i=0; i<4; i=i+1)
           $display("array1[%0d] = %b", i, array1[i]);
       for (i=0; i<6; i=i+1)
           $display("array2[%0d] = %h", i, array2[i]);
end
endmodule

Output:
array1[0] = 000000
array1[1] = 000001
array1[2] = 000010
array1[3] = 000011
array2[0] = 00
array2[1] = 01
array2[2] = 10
array2[3] = 11
array2[4] = xx
array2[5] = xx
```

Part 2: We learnt to write data into a file and then checked the file created.

```
module tb1;
   integer      fd, i;
   reg [7:0]    my_var;
   initial begin
       fd = $fopen("my_file1.txt", "w");
       my_var = 0;
     $fdisplay(fd, "Value displayed with $fdisplay");
       #10 my_var = 8'h1A;
       $fdisplay(fd, my_var);       // Displays in decimal
       $fdisplayb(fd, my_var);      // Displays in binary
       $fdisplayo(fd, my_var);      // Displays in octal
       $fdisplayh(fd, my_var);      // Displays in hex
   // $fwrite does not print the newline char '' automatically at the end of each
  // line; So we can predict all the values printed  below to appear on the same line
     $fdisplay(fd, "Value displayed with $fwrite");
       #10 my_var = 8'h2B;
       $fwrite(fd, my_var);
       $fwrite(fd, " "); // Add a space between values
       $fwriteb(fd, my_var);
       $fwrite(fd, " "); // Add a space between values
       $fwriteo(fd, my_var);
       $fwrite(fd, " "); // Add a space between values
       $fwriteh(fd, my_var);
       $fwrite(fd, "\n"); // Add a newline character to move to the next line
       // Jump to new line with '', and print with strobe which takes
       // the final value of the variable after non-blocking assignments are done
     $fdisplay(fd, "Value displayed with $fstrobe");
       #10 my_var <= 8'h3C;
       $fstrobe(fd, my_var);
       $fstrobeb(fd, my_var);
       $fstrobeo(fd, my_var);
       $fstrobeh(fd, my_var);
     #10 $fdisplay(fd, "Value displayed with $fmonitor");
     $fmonitor(fd, my_var);
       for(i = 0; i < 5; i= i+1) begin
           #5 my_var <= i;
       end
     #10 $fclose(fd);
   end
endmodule

Output:
Value displayed with $fdisplay
26
00011010
032
1a
Value displayed with $fwrite
43 00101011 053 2b
Value displayed with $fstrobe
60
00111100
074
3c
Value displayed with $fmonitor
60
 0
 1
 2
 3
 4
```

Part 3: We learnt to create a BRAM module, implement it on a FPGA and then read the data stored using the switches on the FPGA

```
module trial1(clk, addr, read_write, clear, data_in, data_out);
parameter n = 4;
parameter w = 8;
input clk, read_write, clear;
input [n-1:0] addr;
input [w-1:0] data_in;
output reg [w-1:0] data_out;
reg [w-1:0] reg_array [2**n-1:0];
integer i;
initial
begin
   for( i = 0; i < 2**n; i = i + 1 )
   begin
       reg_array[i] <= 1;
   end
end
always @(negedge(clk)) begin
   if( read_write == 1 )
       reg_array[addr] <= data_in;
   else
       data_out = reg_array[addr];
   if( clear == 1 ) begin
       for( i = 0; i < 2**n; i = i + 1 ) begin
           reg_array[i] <= 0;
       end
   end
end
endmodule
```

Part 4: We learnt to store file's data in the FPGA's BRAM by creating a wrapper module and implementing it on the FPGA. Finally displayed output using switches and LED's on FPGA.

```
`timescale 1ns/10ps
module memory_read(
input clk,
input [3:0]counter1,
input ena,
input wea,
input [7:0]dina,
output [7:0] mem_out);
blk_mem_gen_0 your_instance_name (
.clka(clk),
.ena(ena),
.wea(wea),
.addra(counter1),
.dina(dina),
.douta(mem_out)
);
endmodule
```

Part 5: We learnt to process the file’s data stored in the FPGA's BRAM by creating a wrapper module and implementing it on the FPGA. Finally displayed output using switches and LED’s on FPGA.

```
`timescale 1ns/10ps
module BRAM_processing(
input clk,
input [3:0]counter1,
input ena,
input wea,
input e,
input [7:0]dina,
output reg [7:0]mem_out
);
wire [7:0] m1;
blk_mem_gen_0 your_instance_name (
.clka(clk),
.ena(ena),
.wea(wea),
.addra(counter1),
.dina(dina),
.douta(m1)
);
always @(*) begin
       if (e) begin
           mem_out <= m1 + 1;
       end
       else
       begin
           mem_out <= m1;
       end
   end
endmodule
```

Task 2: Conversion of analog audio signal into its digital equivalent and vice-versa

Here, we converted our analog audio input into its digital form (analog to digital conversion - ADC). Then, the digital-to-analog (DAC) conversion is performed, and the reconstructed audio is played back with the original audio.
The conversion into a digital signal is vital for this project as the FPGA board can only work on digital data (i.e., 0s and 1s). The filtering process occurs on the FPGA board via filters such as low-pass, high-pass, etc., on the digital data. Once filtered, the digital data must be converted back to analog to be played.
We used MATLAB software for this ADC and DAC conversion. We converted our audio signal to a .mat file, a standard binary MATLAB file. Then using the functions audioread and audiowrite we obtained our sampled data values and sampling frequency. We used the functions 'dec2bin' and 'bin2dec' for type conversion between binary and decimal signal values while being mindful of compatible data shapes.
Below is the code from MATLAB:

```
clc;
clear all;
close all;
filename = "C:\Users\jiyad\Downloads\sample_test.ogg";
[audio,fs] = audioread(filename);
save('sample_test_mat.mat', 'audio', 'fs');
load sample_test_mat.mat
audiowrite("sample_test.ogg",audio,fs);
clear audio fs
[audio,fs] = audioread('sample_test.ogg');
audio_normalized = int16(audio * 32767); % Normalize
audio_binary  =  dec2bin(typecast(audio_normalized(:),  'uint16'),  16);  %  Convert  to
binary
binary_vector = audio_binary(:)'; % Reshape to vector for transmission
%Binary to Audio
binary_matrix = reshape(binary_vector, [], 16); % Reshape back to matrix
audio_integers = bin2dec(binary_matrix); % Convert binary to decimal
audio_reconstructed = typecast(uint16(audio_integers), 'int16'); % Typecast to int16
audio_reconstructed_normalized  =  double(audio_reconstructed)  /  32767;  %  Normalize  to
[-1, 1]
% Save the reconstructed audio
audiowrite('reconstructed_audio.wav', audio_reconstructed_normalized, fs);
% playing the original and reconstructed audio
sound(audio, fs); % Play original audio
pause(length(audio)/fs + 1); % Wait for audio to finish plus a little extra
sound(audio_reconstructed_normalized, fs); % Play reconstructed audio
save val;
```

This is what the binary_matrix looks like:

0011111000000011111111000000011110111100000000011111111100000000000000001111111111000
0111111110000000000011111111111111111111110000011…..

These are essentially the binary values corresponding to the sampled analog signal values.

This is the data that the FPGA operates on.