CE-325: Digital System Design
Final Project Report

Huzaifah Tariq Ahmed & Samiya Ali Zaidi

May 10, 2024

1 FPGA

1.1 Filter Design

We used the `MATLAB filterDesigner` to create a filter of the following specifications:

1. **Transition Band: 500 Hz - 1000Hz**
2. **FIR filter**
3. **32 Taps**
4. **Stop band attenuation of more than -20dB**

This was the resultant filter:

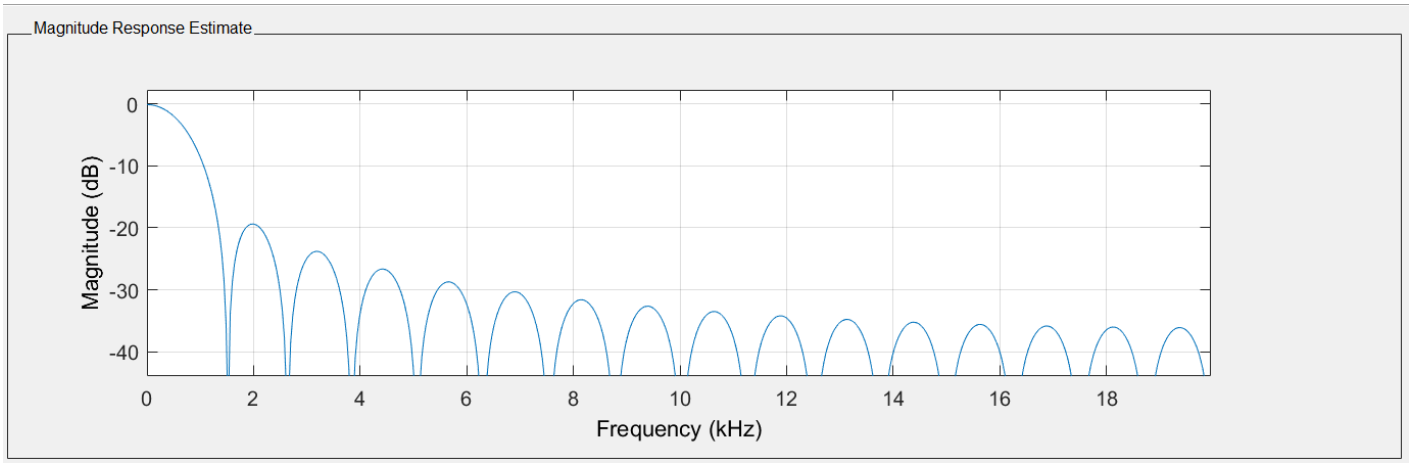

Figure 1: Magnitude Response of the Filter

The parameters used while designing the tool were:

- **Wpass**: 1
- **Wstop**: 1

- **Fs**: 40000
- **Fpass**: 500 Hz
- **Fstop**: 1000 Hz

1.2 Coefficients Conversion

The coefficients extracted from the filter were then converted into a 10-bit signed representation to create a module in Verilog. The following MATLAB script was used for this.

```
% Assuming 'filter' is a matrix containing your filter coefficients

% Define the maximum positive value for 10-bit signed representation
max_positive_value = 511;

% Initialize a matrix to store the converted coefficients
filter_binary = zeros(size(Filter));

% Loop through each coefficient in the 'filter' matrix
for i = 1:numel(Filter)
    % Scale the coefficient to fit within the 10-bit range
    scaled_coefficient = round(Filter(i) * max_positive_value);

    % Check if the coefficient is negative
    if scaled_coefficient < 0
        % Perform sign extension and convert to 10-bit binary
           representation
        filter_binary(i) = bin2dec([repmat('1',1,10-length(dec2bin(abs(
           scaled_coefficient)))) dec2bin(abs(scaled_coefficient))]);
    else
        % Convert positive coefficient to 10-bit binary representation
        filter_binary(i) = bin2dec(dec2bin(scaled_coefficient, 10));
    end
end

% Display the original coefficients and their 10-bit signed binary
   representations
disp(['Original Coefficients: ' num2str(Filter)]);
disp(['Signed Binary Representation: ' num2str(filter_binary)]);
```

Original Coefficients: 0.016774 0.018995 0.021219 0.02342 0.025575 0.027659 0.029645 0.031511 0.033234 0.034792 0.036166 0.037339 0.038295 0.039024 0.039515 0.039762 0.039762 0.039515 0.039024 0.038295 0.037339 0.036166 0.034792 0.033234 0.031511 0.029645 0.027659 0.025575 0.02342 0.021219 0.018995 0.016774

Signed Binary Representation: 9 10 11 12 13 14 15 16 17 18 18 19 20 20 20 20 20 20 20 20 19 18 18 17 16 15 14 13 12 11 10 9

1.3 Verilog Implementation

The coefficients extracted from the filter and then converted to 10-bit signed representation, were then added to our filter implementation in Verilog:

1.3.1 Original FIR Filter Code

```
module fir_low_pass_filter(Data_out,
                           Data_in,
                           clk,
                           rst);

      parameter order = 32;
      parameter word_size_in = 8;
      parameter word_size_out = 2*word_size_in;

      parameter b0 = 10'd9;
      parameter b1 = 10'd10;
      parameter b2 = 10'd11;
      parameter b3 = 10'd12;
      parameter b4 = 10'd13;
      parameter b5 = 10'd14;
      parameter b6 = 10'd15;
      parameter b7 = 10'd16;
      parameter b8 = 10'd17;
      parameter b9 = 10'd18;
      parameter b10 = 10'd18;
      parameter b11 = 10'd19;
      parameter b12 = 10'd20;
      parameter b13 = 10'd20;
      parameter b14 = 10'd20;
      parameter b15 = 10'd20;
      parameter b16 = 10'd20;
      parameter b17 = 10'd20;
      parameter b18 = 10'd20;
      parameter b19 = 10'd20;
      parameter b20 = 10'd19;
      parameter b21 = 10'd18;
      parameter b22 = 10'd18;
      parameter b23 = 10'd17;
      parameter b24 = 10'd16;
      parameter b25 = 10'd15;
      parameter b26 = 10'd14;
      parameter b27 = 10'd13;
      parameter b28 = 10'd12;
      parameter b29 = 10'd11;
      parameter b30 = 10'd10;
      parameter b31 = 10'd9;

      output [word_size_out-1:0] Data_out;
```

```
44
45      input [word_size_in-1:0] Data_in;
46      input clk,rst;
47
48      reg signed [word_size_in-1:0] Samples [1:order];
49
50      integer k;
51
52      assign Data_out = b0  * (Data_in >>> 3) +
53                        b1  * (Samples[1] >>> 3) +
54                        b2  * (Samples[2] >>> 3) +
55                        b3  * (Samples[3] >>> 3) +
56                        b4  * (Samples[4] >>> 3) +
57                        b5  * (Samples[5] >>> 3) +
58                        b6  * (Samples[6] >>> 3) +
59                        b7  * (Samples[7] >>> 3) +
60                        b8  * (Samples[8] >>> 3) +
61                        b9  * (Samples[9] >>> 3) +
62                        b10 * (Samples[10] >>> 3) +
63                        b11 * (Samples[11] >>> 3) +
64                        b12 * (Samples[12] >>> 3) +
65                        b13 * (Samples[13] >>> 3) +
66                        b14 * (Samples[14] >>> 3) +
67                        b15 * (Samples[15] >>> 3) +
68                        b16 * (Samples[16] >>> 3) +
69                        b17 * (Samples[17] >>> 3) +
70                        b18 * (Samples[18] >>> 3) +
71                        b19 * (Samples[19] >>> 3) +
72                        b20 * (Samples[20] >>> 3) +
73                        b21 * (Samples[21] >>> 3) +
74                        b22 * (Samples[22] >>> 3) +
75                        b23 * (Samples[23] >>> 3)+
76                        b24 * (Samples[24] >>> 3) +
77                        b25 * (Samples[25] >>> 3) +
78                        b26 * (Samples[26] >>> 3) +
79                        b27 * (Samples[27] >>> 3) +
80                        b28 * (Samples[28] >>> 3) +
81                        b29 * (Samples[29] >>> 3) +
82                        b30 * (Samples[30] >>> 3) +
83                        b31 * (Samples[31] >>> 3);
84
85      always @ (posedge clk)
86        if(rst == 1)
87          begin
88            for(k=1;k<=order;k=k+1)
89              Samples[k] <= 0;
90          end
91      else
92        begin
93          Samples[1] <= Data_in;
```

```
            for(k=2;k<=order;k=k+1)
              Samples[k] <= Samples[k-1];
          end

endmodule
```

1.3.2 Sign Bit Preservation Logic

So that are signed bit doesn't get affected by the shift right operations performed on the input samples, we altered our original logic to preserve the sign bit as the starting bit of our filtered output.

```
function signed [15:0] multiply_and_attach_sign(input signed [7:0] data,
                                                input signed [9:0] coeff);
    reg signed [6:0] shifted_val; // Shifting only the lower 7 bits
    reg signed [14:0] mult_result; // To store multiplication result
    begin
      shifted_val = data[6:0] >>> 3; // Take the lower 7 bits
      mult_result = shifted_val * coeff; // Multiply by coefficient
      // Re-attach the sign bit
      multiply_and_attach_sign = {data[7], mult_result};
    end
endfunction

assign Data_out = multiply_and_attach_sign(Data_in, b0) +
                  multiply_and_attach_sign(Samples[1], b1) +
                  multiply_and_attach_sign(Samples[2], b2) +
                  multiply_and_attach_sign(Samples[3], b3) +
                  multiply_and_attach_sign(Samples[4], b4) +
                  multiply_and_attach_sign(Samples[5], b5) +
                  multiply_and_attach_sign(Samples[6], b6) +
                  multiply_and_attach_sign(Samples[7], b7) +
                  multiply_and_attach_sign(Samples[8], b8) +
                  multiply_and_attach_sign(Samples[9], b9) +
                  multiply_and_attach_sign(Samples[10], b10) +
                  multiply_and_attach_sign(Samples[11], b11) +
                  multiply_and_attach_sign(Samples[12], b12) +
                  multiply_and_attach_sign(Samples[13], b13) +
                  multiply_and_attach_sign(Samples[14], b14) +
                  multiply_and_attach_sign(Samples[15], b15) +
                  multiply_and_attach_sign(Samples[16], b16) +
                  multiply_and_attach_sign(Samples[17], b17) +
                  multiply_and_attach_sign(Samples[18], b18) +
                  multiply_and_attach_sign(Samples[19], b19) +
                  multiply_and_attach_sign(Samples[20], b20) +
                  multiply_and_attach_sign(Samples[21], b21) +
                  multiply_and_attach_sign(Samples[22], b22) +
                  multiply_and_attach_sign(Samples[23], b23) +
                  multiply_and_attach_sign(Samples[24], b24) +
                  multiply_and_attach_sign(Samples[25], b25) +
                  multiply_and_attach_sign(Samples[26], b26) +
```

```
                multiply_and_attach_sign(Samples[27], b27) +
                multiply_and_attach_sign(Samples[28], b28) +
                multiply_and_attach_sign(Samples[29], b29) +
                multiply_and_attach_sign(Samples[30], b30) +
                multiply_and_attach_sign(Samples[31], b31);
```

1.4 Matlab Generated Audio File

1.4.1 Generating Tones

Two tones were generated using Matlab: 100 Hz and 8000 Hz.

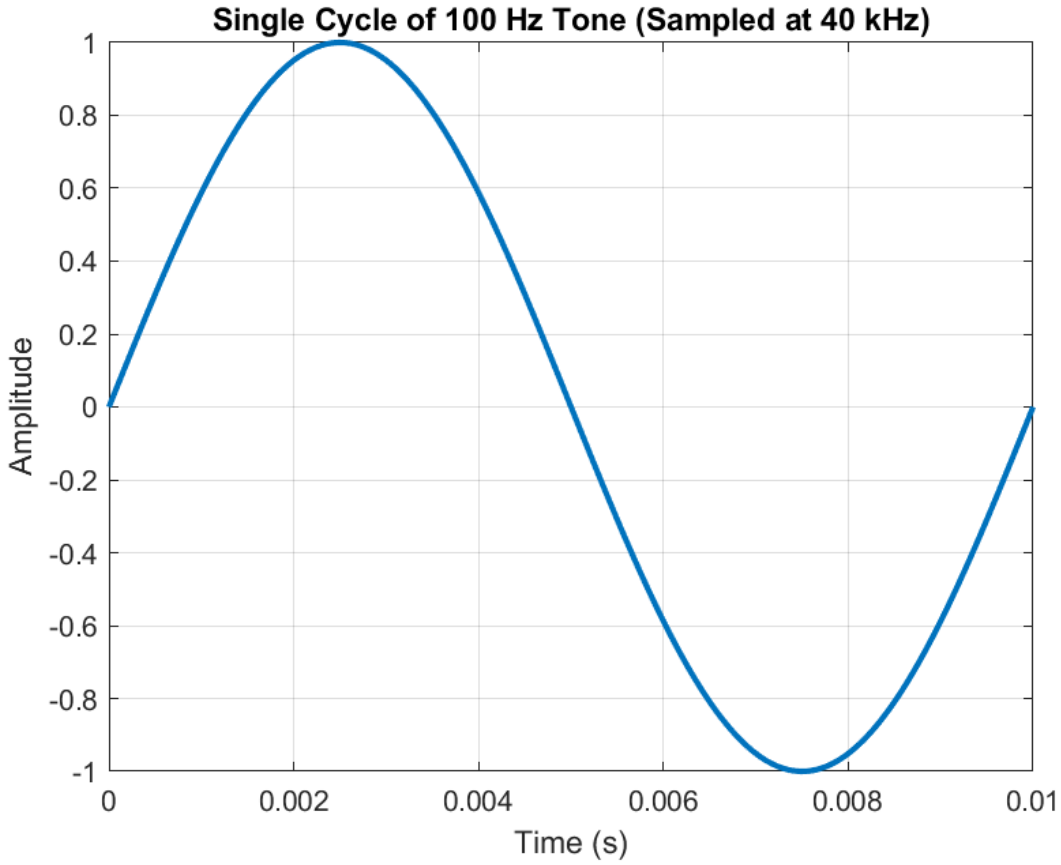

Figure 2: 100 Hz Tone Generated

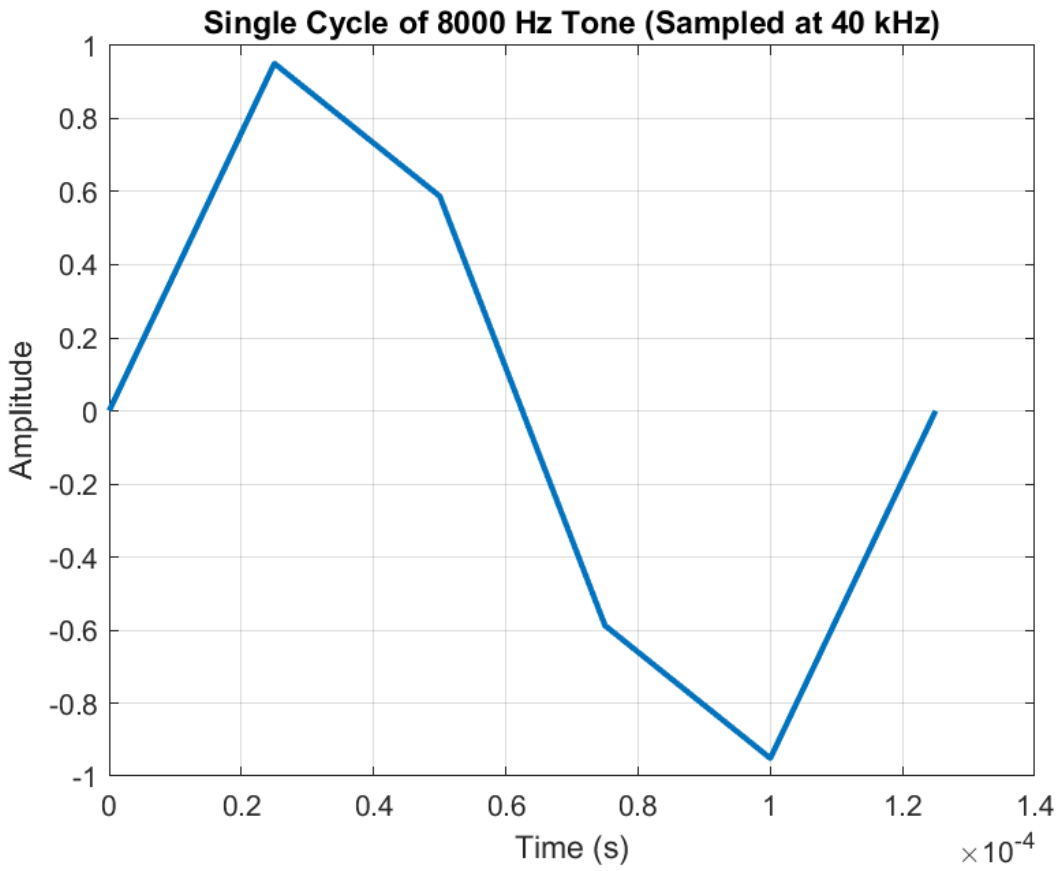

Figure 3: 8000 Hz Tone Generated

1.4.2 Passing the Tones through the Filter In MATLAB

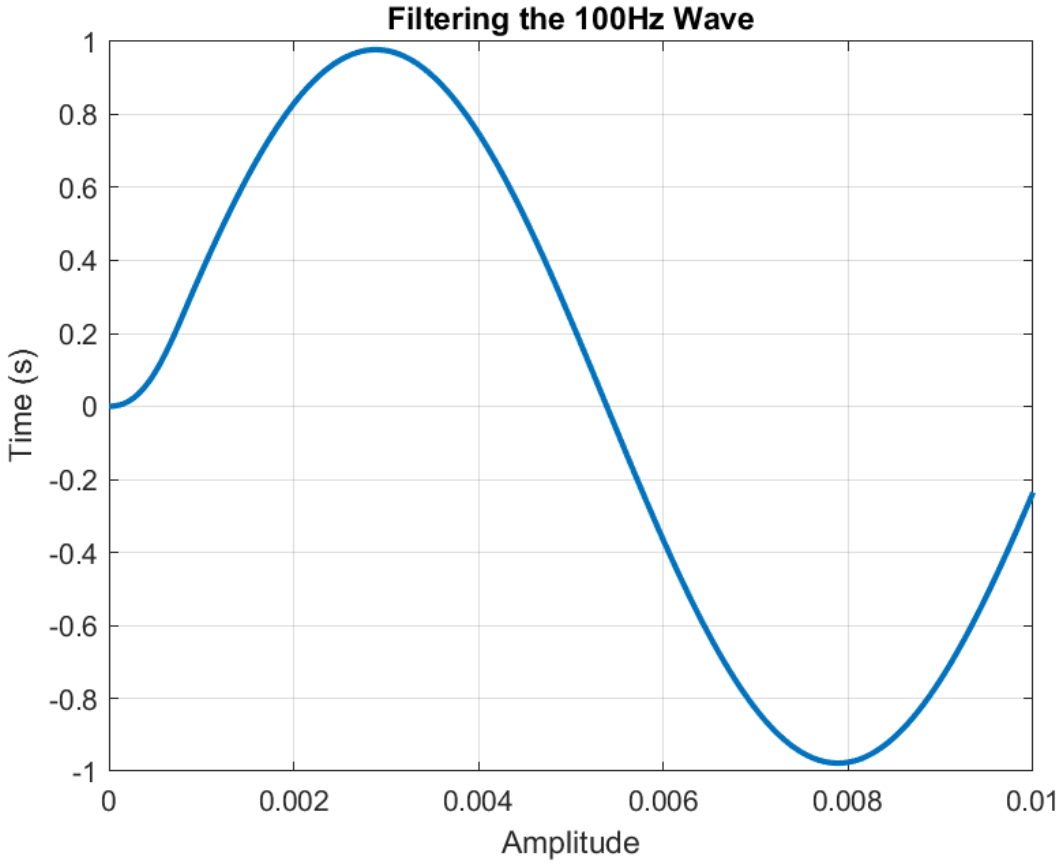

Figure 4: 100Hz filtered using MATLAB

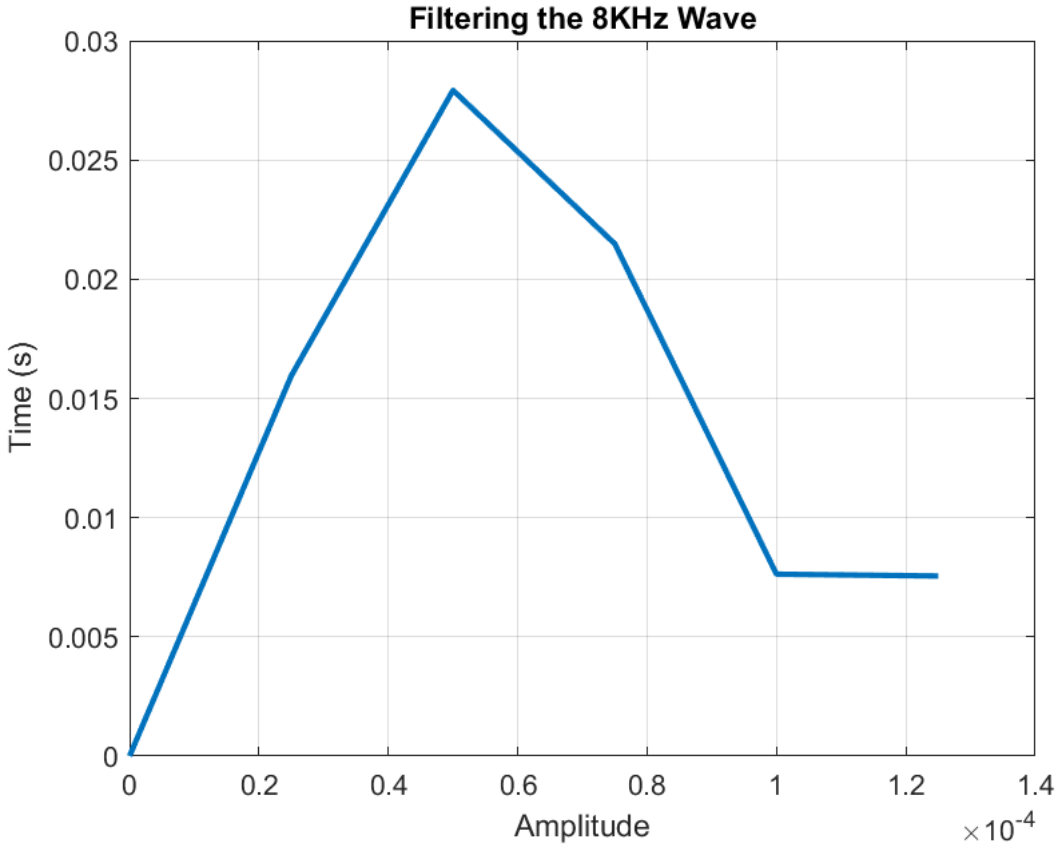

Figure 5: 8000Hz filtered using MATLAB

1.5 Passing the Tones through the Filter Implementation In Verilog

1.5.1 Conversion to Binary

For storing the tones in a ROM, and then subsequently passing them into the LPF, we used this code to convert the decimal numbers into signed 8-bit representation:

```
% read the audio file
audio_filename = '100Hz_Tone.wav';
[audio_data, sample_rate] = audioread(audio_filename); % read the file

% convert to 8-bit Signed Representation
% scale the audio data to fit the -128 to 127 range
audio_data_scaled = audio_data * 127; % scale to 8-bit signed integer
   range

audio_data_int8 = int8(audio_data_scaled); % Convert to 8-bit signed
   integers

% convert to 8-bit binary representation
binary_strings = arrayfun(@(x) dec2bin(typecast(x, 'uint8'), 8),
   audio_data_int8, 'UniformOutput', false);
```

```
% write to a text file
txt_filename = 'audio_data_8bit_100Hz.txt'; % Name of the text file to
   create
file_id = fopen(txt_filename, 'w'); % Open in write mode

% write each binary string on a new line
for i = 1:length(binary_strings)
    fprintf(file_id, '%s\n', binary_strings{i}); % Write the binary
       string
end

% close the text file
fclose(file_id);

fprintf('Audio data saved as 8-bit signed binary representation in %s\n
   ', txt_filename);
```

To verify the converted values:

```
% read the text file
txt_filename = 'audio_data_8bit_100Hz.txt';
file_id = fopen(txt_filename, 'r'); % open the file
binary_strings = textscan(file_id, '%s');
fclose(file_id);

binary_strings = binary_strings{1}; % Get the list of binary strings

% convert the binary strings to unsigned decimal
unsigned_values = cellfun(@(x) bin2dec(x), binary_strings);

% convert to signed 8-bit integers
signed_values = unsigned_values;
% apply 2s complement
signed_values(signed_values > 127) = signed_values(signed_values > 127)
    - 256;

% convert to int8
signed_values = int8(signed_values);

disp('Binary strings and their corresponding signed decimal values:');

for i = 1:length(binary_strings)
    fprintf('Binary: %s, Signed Decimal: %d\n', binary_strings{i},
       signed_values(i));
end
```

1.6 Testing on Verilog FIR

1.6.1 100HZ Signal

We can see from our results below that are filter effectively allowed passing the 100HZ signal as it is, without any attenuation. Which is what the intended result is, as 100 HZ is within our pass band of the filter.

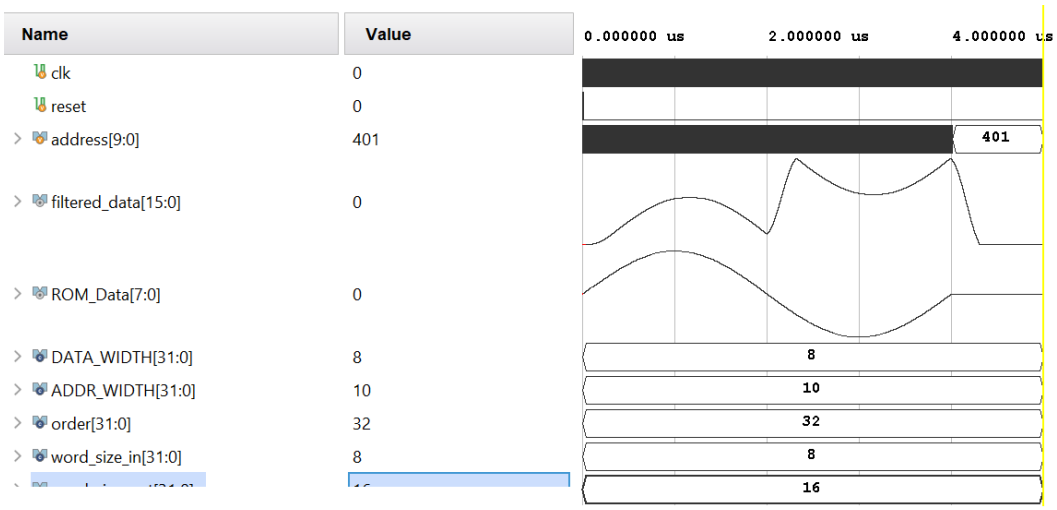

Figure 6: Filtered 100Hz Signal. The signal is passing through the filter.

The problem here is with the overflow handling of the bits, because of which the negative portion of the signal is shifted upwards. We tried to preserve the sign bit by first multiplying the coefficients with right-shifted-inputs (>>> 3), and then concatenating it with the sign bit, however, we were not getting a correct output with it.

1.6.2 8kHZ Signal

We can see here that when we pass 8kHz frequency tone through this so we see our tone getting attenuated and its amplitude reducing by half from 1 originally to 0.5. This can be seen both in the waveform generated in Verilog as well as through the filtered output values plotted in Matlab.

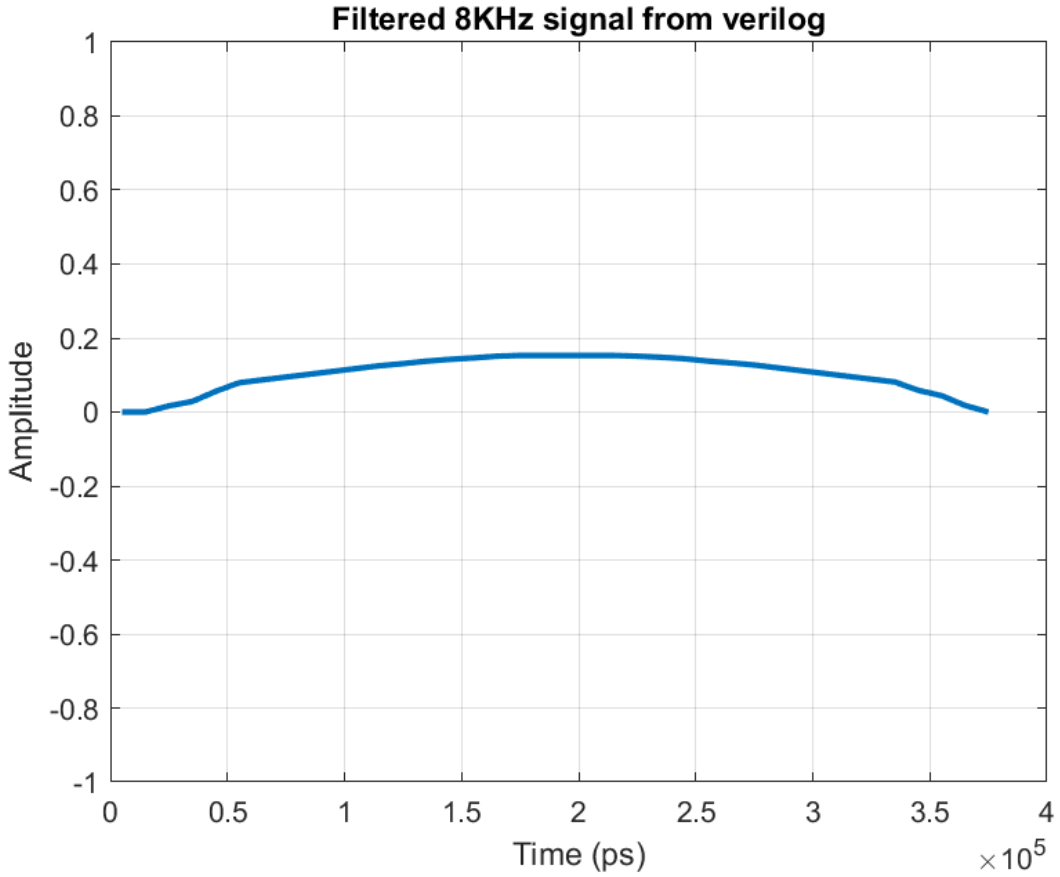

Figure 7: 8KHz filtered signal obtained from Verilog, plotted using MATLAB

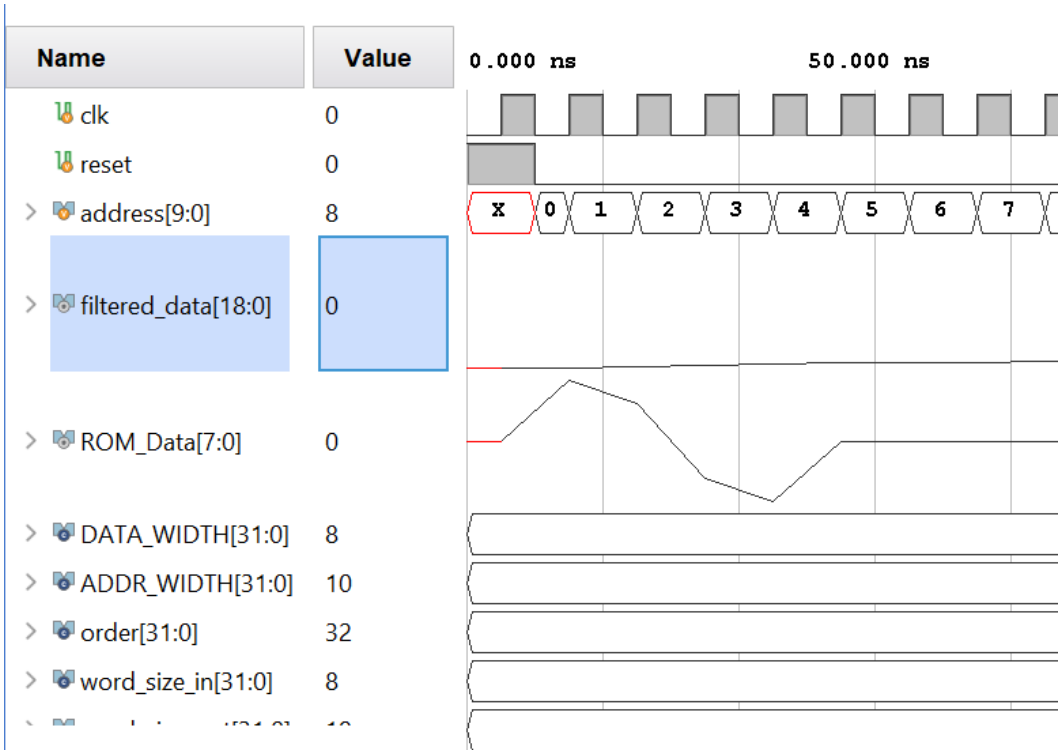

Figure 8: 8KHz signal filtered output in Verilog Waveform