ASSIGNMENT - 4

AIM: Designing FIR Filter using filter coefficients (from Matlab) using Matlab and Verilog. And verifying Verilog values by comparing with Matlab.

FIR Filter:

A Finite Impulse Response (FIR) filter is a type of digital filter characterized by a finite-duration impulse response. FIR filters are widely used in signal processing applications due to their inherent stability and linear-phase properties.

An FIR filter is defined by the convolution sum:

$$y[n] = \sum_{k=0}^{M-1} h[k]x[n-k]$$

where:

y[n] is the output signal,

x[n] is the input signal,

h[k] are the filter coefficients (impulse response),

M is the filter order (number of taps),

k represents discrete time indices.

Unlike Infinite Impulse Response (IIR) filters, FIR filters do not rely on feedback, making them inherently stable.

MATLAB and Verilog Codes:

I've stored some values of band pass filter coefficients from Matlab filter design window. And converted them to Q(2,14) format fixed points.

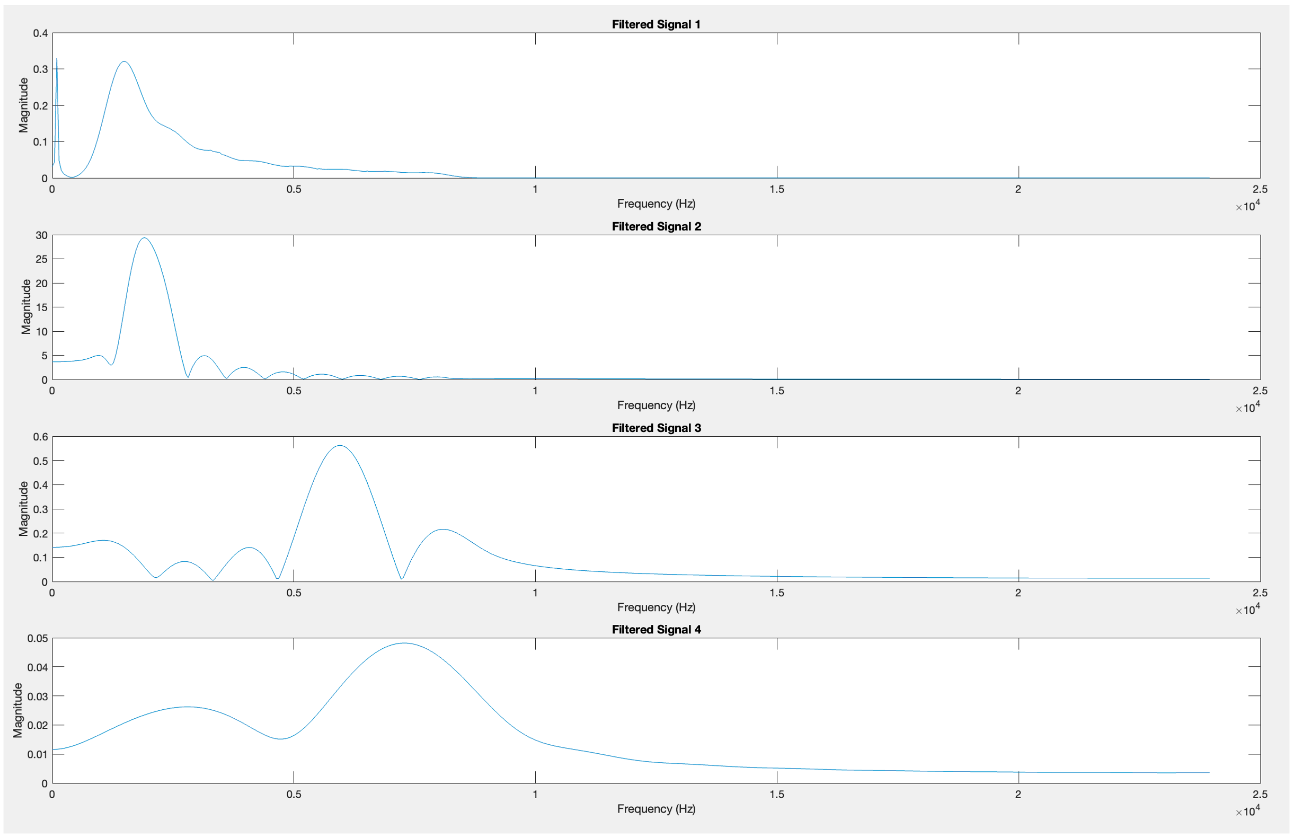
Now, I've generated 4 sine waves of frequencies 100Hz, 2000Hz, 6000Hz, and 11000Hz and converted them to Q(2,14) format fixed points. Then, I'm using the filter coefficients stored before to perform filtering with the sine waves and plot the frequency responses of the filtered waves.

Below is the matlab code to do filter operation,

```matlab
% Storing Filter Coefficients
data = load('Num.mat');
coeffs = data.Num;
a = coeffs;

q_214 = round(a * 2^14)/2^14;
fileID = fopen('fp.txt', 'w');
fprintf(fileID, '%.6f\n', q_214);
fclose(fileID);

% Sine wave generation
Fs = 48000;
f1 = 100; f2 = 2000; f3 = 6000; f4 = 11000;
t1 = 0:1/Fs:(5 * 1/f1);
t2 = 0:1/Fs:(5 * 1/f2);
t3 = 0:1/Fs:(5 * 1/f3);
t4 = 0:1/Fs:(5 * 1/f4);

% Generate sine waves
sine1 = sin(2 * pi * f1 * t1);
sine2 = sin(2 * pi * f2 * t2);
sine3 = sin(2 * pi * f3 * t3);
sine4 = sin(2 * pi * f4 * t4);

% Convert to fixed-point Q(2,14) format
sine1_q214 = round(sine1 * 2^14) / 2^14;
sine2_q214 = round(sine2 * 2^14) / 2^14;
sine3_q214 = round(sine3 * 2^14) / 2^14;
sine4_q214 = round(sine4 * 2^14) / 2^14;

fileID1 = fopen('sine1_fp.txt', 'w');
fprintf(fileID1, '%.6f\n', sine1_q214);
fclose(fileID1);

fileID2 = fopen('sine2_fp.txt', 'w');
fprintf(fileID2, '%.6f\n', sine2_q214);
fclose(fileID2);

fileID3 = fopen('sine3_fp.txt', 'w');
fprintf(fileID3, '%.6f\n', sine3_q214);
fclose(fileID3);

fileID4 = fopen('sine4_fp.txt', 'w');
fprintf(fileID4, '%.6f\n', sine4_q214);
fclose(fileID4);
% Step 3: Apply the filter using 'filter' function
filtered_sine1 = filter(a, 1, sine1_q214);
filtered_sine2 = filter(a, 1, sine2_q214);
filtered_sine3 = filter(a, 1, sine3_q214);
filtered_sine4 = filter(a, 1, sine4_q214);

% Function to save floating-point data line by line
function save_fp_file(filename, data)
    fileID = fopen(filename, 'w');
    for i = 1:length(data)
        fprintf(fileID, '%.6f\n', data(i));  % Write each floating-point value in a new line
    end
    fclose(fileID);
end

% Save the floating-point values back to text files
save_fp_file('filtered_sine1_mat.txt', filtered_sine1);
save_fp_file('filtered_sine2_mat.txt', filtered_sine2);
save_fp_file('filtered_sine3_mat.txt', filtered_sine3);
save_fp_file('filtered_sine4_mat.txt', filtered_sine4);
```

```
% Step 4: Compute FFT of original and filtered signals
N = 1024;
f_axis = (0:N/2-1) * (Fs/N);  % Frequency axis (half spectrum)

fft_filtered1 = fft(filtered_sine1, N);
fft_filtered2 = fft(filtered_sine2, N);
fft_filtered3 = fft(filtered_sine3, N);
fft_filtered4 = fft(filtered_sine4, N);

% Take only first half of FFT (positive frequencies)
fft_filtered1 = abs(fft_filtered1(1:N/2));
fft_filtered2 = abs(fft_filtered2(1:N/2));
fft_filtered3 = abs(fft_filtered3(1:N/2));
fft_filtered4 = abs(fft_filtered4(1:N/2));

% Step 5: Plot Frequency-Domain Representation
figure;
subplot(4,1,1);
plot(f_axis, fft_filtered1);
title('Filtered Signal 1');
xlabel('Frequency (Hz)'); ylabel('Magnitude');

subplot(4,1,2);
plot(f_axis, fft_filtered2);
title('Filtered Signal 2');
xlabel('Frequency (Hz)'); ylabel('Magnitude');

subplot(4,1,3);
plot(f_axis, fft_filtered3);
title('Filtered Signal 3');
xlabel('Frequency (Hz)'); ylabel('Magnitude');

subplot(4,1,4);
plot(f_axis, fft_filtered4);
title('Filtered Signal 4');
xlabel('Frequency (Hz)'); ylabel('Magnitude');
```

<u>Frequency Response of 4 Signals</u>:

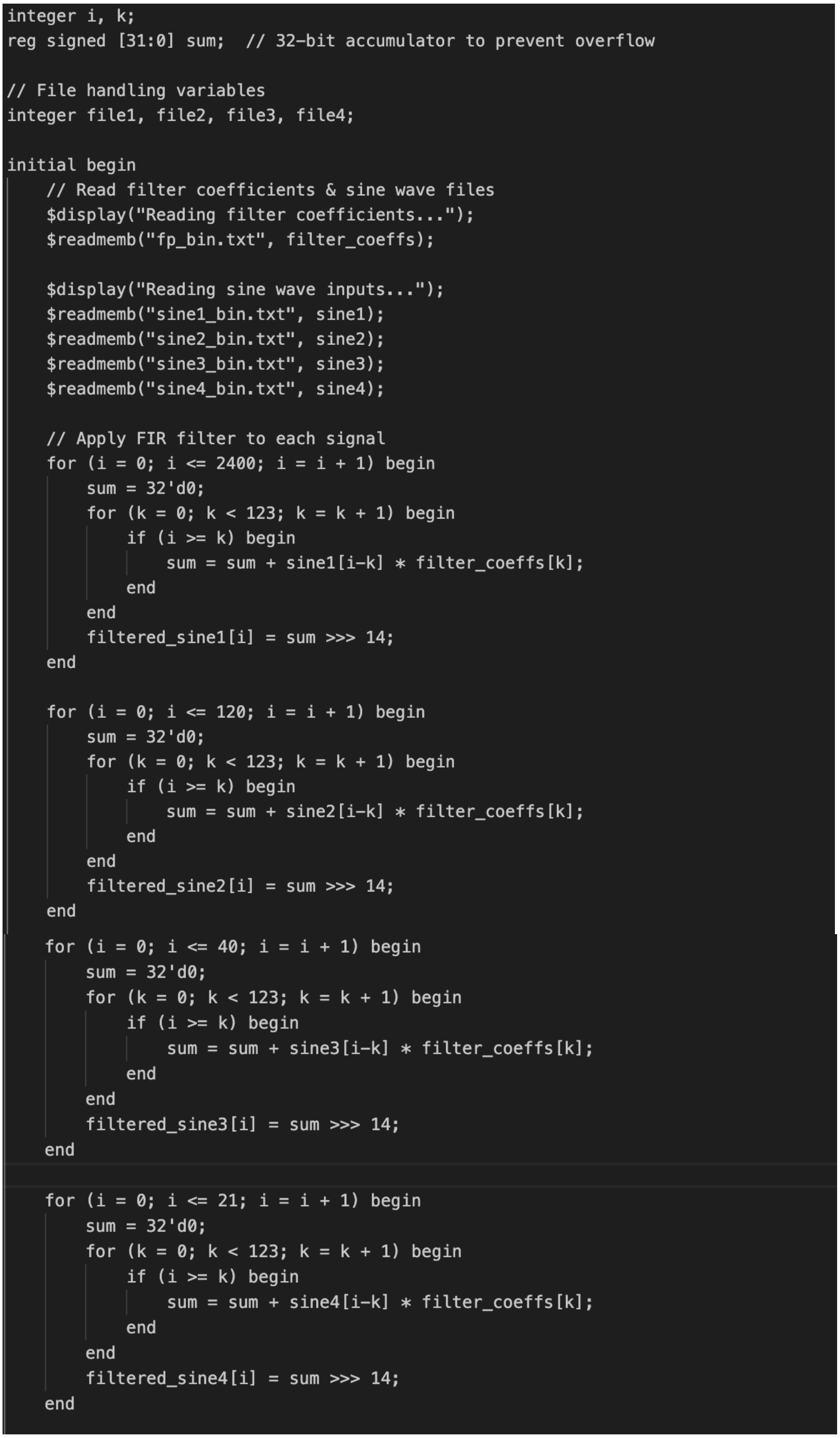


Now, We have to perform the operation in Verilog. To read the input files in verilog we have to convert them into binary numbers. I’ve used matlab to do that. My Matlab code for this conversion is

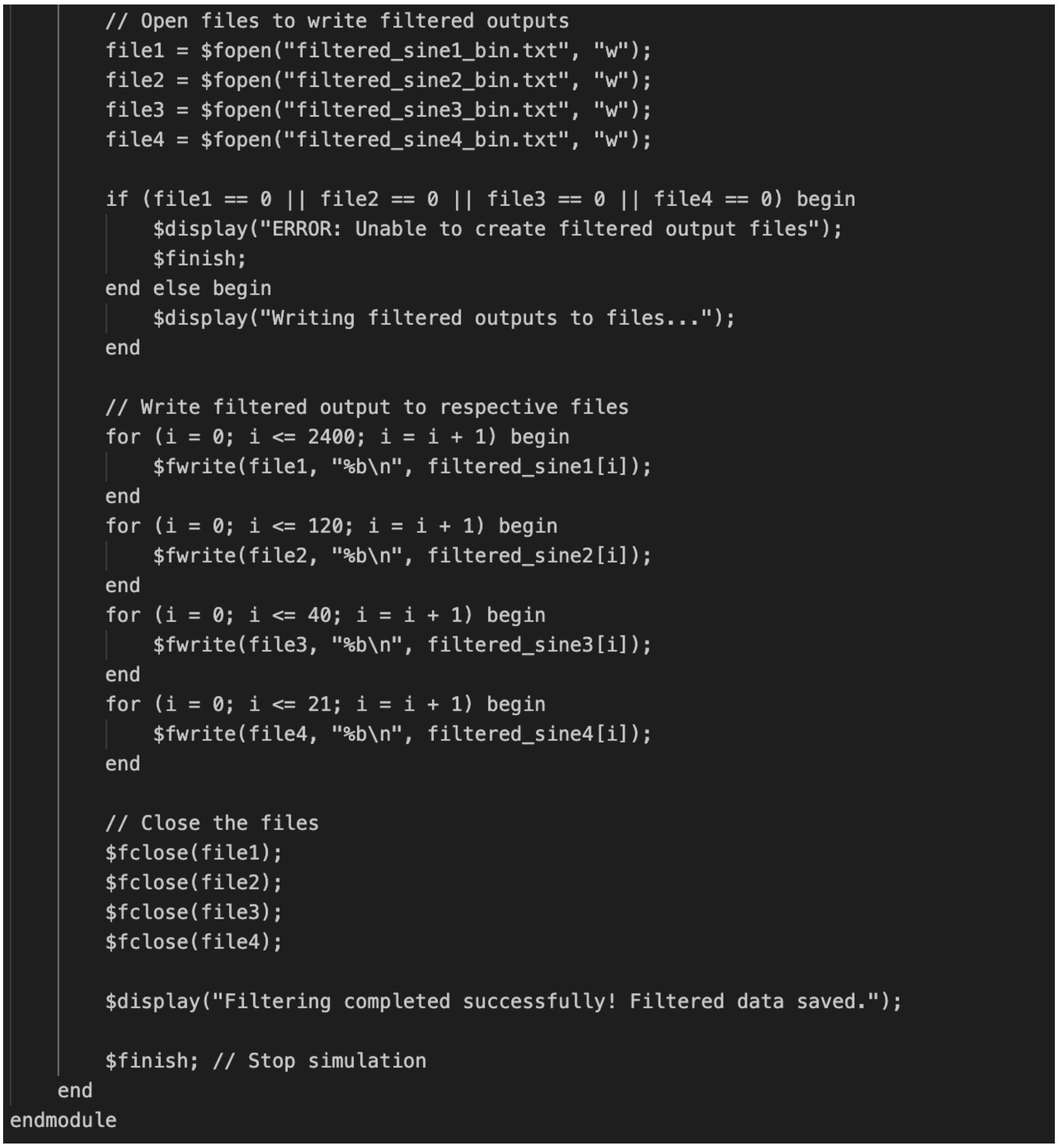
```
% Load floating-point filter coefficients and signals
filter_coeffs = load('fp.txt');  % 20 filter coefficients
sine_100Hz = load('sine1_fp.txt');  % 123 signal samples
sine_2000Hz = load('sine2_fp.txt');
sine_6000Hz = load('sine3_fp.txt');
sine_11000Hz = load('sine4_fp.txt');

% Convert floating-point values to Q2.14 format (scale by 2^14)
filter_coeffs_fixed = round(filter_coeffs * 2^14);
sine_100Hz_fixed = round(sine_100Hz * 2^14);
sine_2000Hz_fixed = round(sine_2000Hz * 2^14);
sine_6000Hz_fixed = round(sine_6000Hz * 2^14);
sine_11000Hz_fixed = round(sine_11000Hz * 2^14);

% Convert to 16-bit two's complement representation (binary format)
filter_coeffs_bin = dec2bin(mod(filter_coeffs_fixed, 2^16), 16);
sine_100Hz_bin = dec2bin(mod(sine_100Hz_fixed, 2^16), 16);
sine_2000Hz_bin = dec2bin(mod(sine_2000Hz_fixed, 2^16), 16);
sine_6000Hz_bin = dec2bin(mod(sine_6000Hz_fixed, 2^16), 16);
sine_11000Hz_bin = dec2bin(mod(sine_11000Hz_fixed, 2^16), 16);

% Function to save binary data line by line
function save_bin_file(filename, data)
    fileID = fopen(filename, 'w');
    for i = 1:size(data, 1)
        fprintf(fileID, '%s\n', data(i, :));  % Write each value as a new line
    end
    fclose(fileID);
end

% Save as text files in binary format for Verilog
save_bin_file('fp_bin.txt', filter_coeffs_bin);
save_bin_file('sine1_bin.txt', sine_100Hz_bin);
save_bin_file('sine2_bin.txt', sine_2000Hz_bin);
save_bin_file('sine3_bin.txt', sine_6000Hz_bin);
save_bin_file('sine4_bin.txt', sine_11000Hz_bin);
```

Since, the files are converted into binary now we can read the files in verilog and do filtering. My Verilog code for doing filtering is

```
module fir_filter_system;
    // Define memory arrays for coefficients and signals
    reg signed [15:0] filter_coeffs [0:122];  // 123 filter coefficients (Q2.14)
    reg signed [15:0] sine1 [0:2400];        // 2401 samples for sine waves
    reg signed [15:0] sine2 [0:120];
    reg signed [15:0] sine3 [0:40];
    reg signed [15:0] sine4 [0:21];

    reg signed [15:0] filtered_sine1 [0:2400]; // Filtered outputs
    reg signed [15:0] filtered_sine2 [0:120];
    reg signed [15:0] filtered_sine3 [0:40];
    reg signed [15:0] filtered_sine4 [0:21];
```

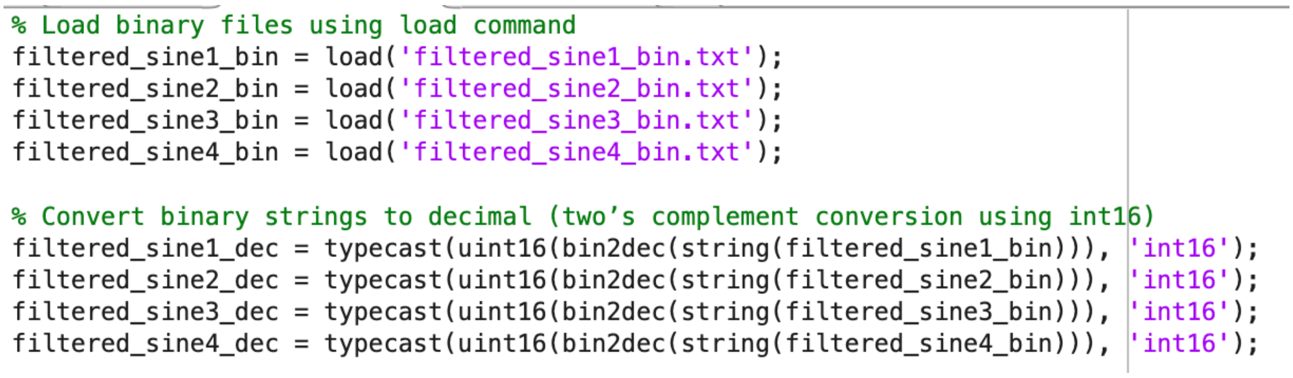


```verilog
integer i, k;
reg signed [31:0] sum;  // 32-bit accumulator to prevent overflow

// File handling variables
integer file1, file2, file3, file4;

initial begin
    // Read filter coefficients & sine wave files
    $display("Reading filter coefficients...");
    $readmemb("fp_bin.txt", filter_coeffs);

    $display("Reading sine wave inputs...");
    $readmemb("sine1_bin.txt", sine1);
    $readmemb("sine2_bin.txt", sine2);
    $readmemb("sine3_bin.txt", sine3);
    $readmemb("sine4_bin.txt", sine4);

    // Apply FIR filter to each signal
    for (i = 0; i <= 2400; i = i + 1) begin
        sum = 32'd0;
        for (k = 0; k < 123; k = k + 1) begin
            if (i >= k) begin
                sum = sum + sine1[i-k] * filter_coeffs[k];
            end
        end
        filtered_sine1[i] = sum >>> 14;
    end

    for (i = 0; i <= 120; i = i + 1) begin
        sum = 32'd0;
        for (k = 0; k < 123; k = k + 1) begin
            if (i >= k) begin
                sum = sum + sine2[i-k] * filter_coeffs[k];
            end
        end
        filtered_sine2[i] = sum >>> 14;
    end
    for (i = 0; i <= 40; i = i + 1) begin
        sum = 32'd0;
        for (k = 0; k < 123; k = k + 1) begin
            if (i >= k) begin
                sum = sum + sine3[i-k] * filter_coeffs[k];
            end
        end
        filtered_sine3[i] = sum >>> 14;
    end

    for (i = 0; i <= 21; i = i + 1) begin
        sum = 32'd0;
        for (k = 0; k < 123; k = k + 1) begin
            if (i >= k) begin
                sum = sum + sine4[i-k] * filter_coeffs[k];
            end
        end
        filtered_sine4[i] = sum >>> 14;
    end
```



```
        // Open files to write filtered outputs
        file1 = $fopen("filtered_sine1_bin.txt", "w");
        file2 = $fopen("filtered_sine2_bin.txt", "w");
        file3 = $fopen("filtered_sine3_bin.txt", "w");
        file4 = $fopen("filtered_sine4_bin.txt", "w");

        if (file1 == 0 || file2 == 0 || file3 == 0 || file4 == 0) begin
            $display("ERROR: Unable to create filtered output files");
            $finish;
        end else begin
            $display("Writing filtered outputs to files...");
        end

        // Write filtered output to respective files
        for (i = 0; i <= 2400; i = i + 1) begin
            $fwrite(file1, "%b\n", filtered_sine1[i]);
        end
        for (i = 0; i <= 120; i = i + 1) begin
            $fwrite(file2, "%b\n", filtered_sine2[i]);
        end
        for (i = 0; i <= 40; i = i + 1) begin
            $fwrite(file3, "%b\n", filtered_sine3[i]);
        end
        for (i = 0; i <= 21; i = i + 1) begin
            $fwrite(file4, "%b\n", filtered_sine4[i]);
        end

        // Close the files
        $fclose(file1);
        $fclose(file2);
        $fclose(file3);
        $fclose(file4);

        $display("Filtering completed successfully! Filtered data saved.");

        $finish; // Stop simulation
    end
endmodule
```

Signals are filtered successfully. Now we have to verify these results using Matlab results. For that we have to convert these binary values again into floating point values. Matlab code for that conversion is



```
% Load binary files using load command
filtered_sine1_bin = load('filtered_sine1_bin.txt');
filtered_sine2_bin = load('filtered_sine2_bin.txt');
filtered_sine3_bin = load('filtered_sine3_bin.txt');
filtered_sine4_bin = load('filtered_sine4_bin.txt');

% Convert binary strings to decimal (two’s complement conversion using int16)
filtered_sine1_dec = typecast(uint16(bin2dec(string(filtered_sine1_bin))), 'int16');
filtered_sine2_dec = typecast(uint16(bin2dec(string(filtered_sine2_bin))), 'int16');
filtered_sine3_dec = typecast(uint16(bin2dec(string(filtered_sine3_bin))), 'int16');
filtered_sine4_dec = typecast(uint16(bin2dec(string(filtered_sine4_bin))), 'int16');
```

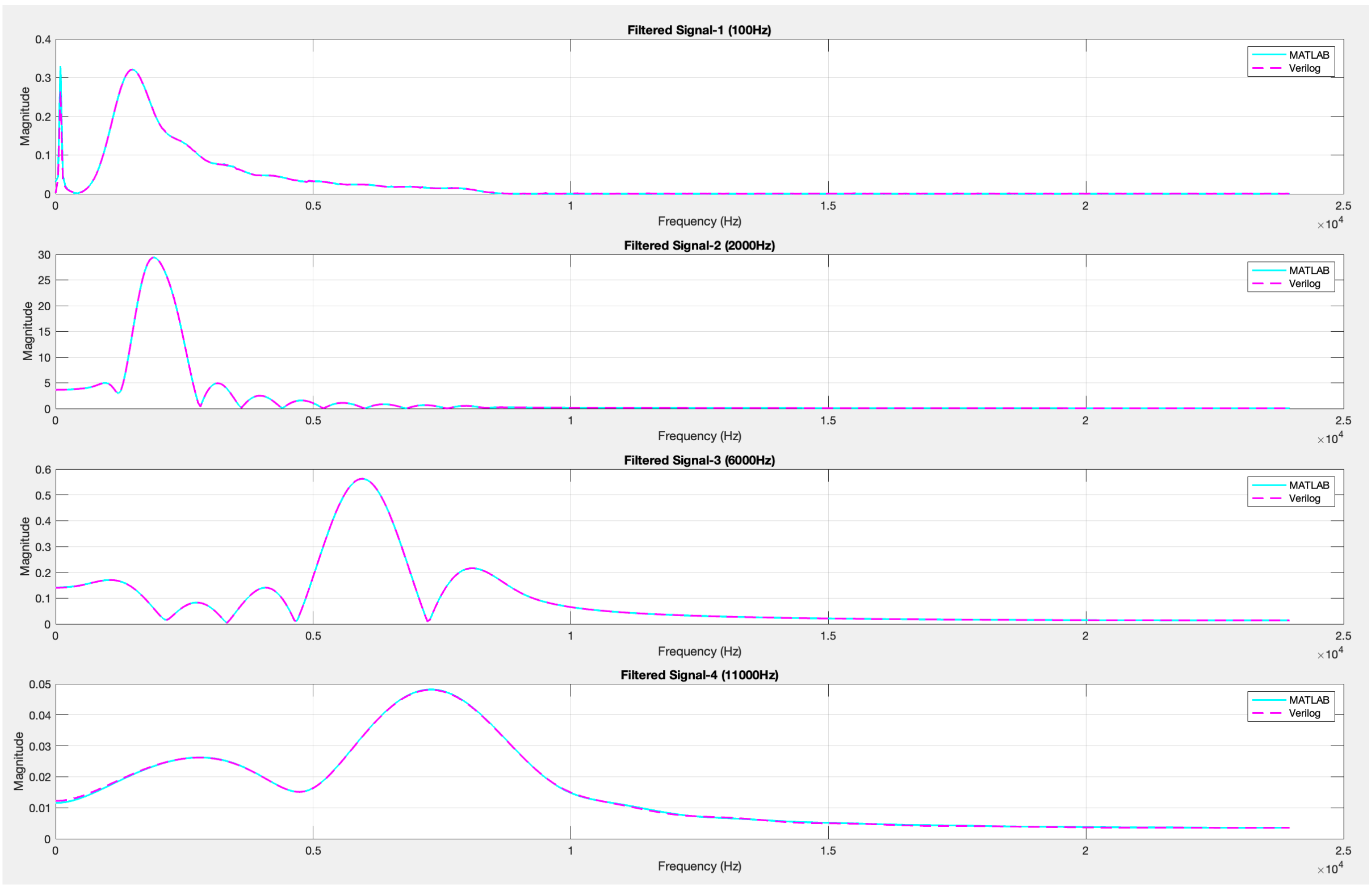
```matlab
% Convert back to floating-point Q2.14 format
filtered_sine1_fp = double(filtered_sine1_dec) / 2^14;
filtered_sine2_fp = double(filtered_sine2_dec) / 2^14;
filtered_sine3_fp = double(filtered_sine3_dec) / 2^14;
filtered_sine4_fp = double(filtered_sine4_dec) / 2^14;

% Function to save floating-point data line by line
function save_fp_file(filename, data)
    fileID = fopen(filename, 'w');
    for i = 1:length(data)
        fprintf(fileID, '%.6f\n', data(i));  % Write each floating-point value in a new line
    end
    fclose(fileID);
end

% Save the floating-point values back to text files
save_fp_file('filtered_sine1_ver.txt', filtered_sine1_fp);
save_fp_file('filtered_sine2_ver.txt', filtered_sine2_fp);
save_fp_file('filtered_sine3_ver.txt', filtered_sine3_fp);
save_fp_file('filtered_sine4_ver.txt', filtered_sine4_fp);
```

Now, we have to verify verilog values with matlab values. For that I have compared plots of frequency responses for all the filtered signals in Verilog and Matlab. My Matlab Code for that is

```matlab
% Step 1: Load Verilog and MATLAB output files
filtered_sine1_ver = load('filtered_sine1_ver.txt');
filtered_sine2_ver = load('filtered_sine2_ver.txt');
filtered_sine3_ver = load('filtered_sine3_ver.txt');
filtered_sine4_ver = load('filtered_sine4_ver.txt');

filtered_sine1_mat = load('filtered_sine1_mat.txt');
filtered_sine2_mat = load('filtered_sine2_mat.txt');
filtered_sine3_mat = load('filtered_sine3_mat.txt');
filtered_sine4_mat = load('filtered_sine4_mat.txt');

% Step 2: Set parameters for FFT
Fs = 48000;
N = 1024;
f_axis = (0:N/2-1) * (Fs/N);

% Step 3: Compute FFT of MATLAB and Verilog outputs
fft_sine1_mat = abs(fft(filtered_sine1_mat, N));
fft_sine2_mat = abs(fft(filtered_sine2_mat, N));
fft_sine3_mat = abs(fft(filtered_sine3_mat, N));
fft_sine4_mat = abs(fft(filtered_sine4_mat, N));

fft_sine1_ver = abs(fft(filtered_sine1_ver, N));
fft_sine2_ver = abs(fft(filtered_sine2_ver, N));
fft_sine3_ver = abs(fft(filtered_sine3_ver, N));
fft_sine4_ver = abs(fft(filtered_sine4_ver, N));

% Take only the first half of the FFT (positive frequencies)
fft_sine1_mat = fft_sine1_mat(1:N/2);
fft_sine2_mat = fft_sine2_mat(1:N/2);
fft_sine3_mat = fft_sine3_mat(1:N/2);
fft_sine4_mat = fft_sine4_mat(1:N/2);

fft_sine1_ver = fft_sine1_ver(1:N/2);
fft_sine2_ver = fft_sine2_ver(1:N/2);
fft_sine3_ver = fft_sine3_ver(1:N/2);
fft_sine4_ver = fft_sine4_ver(1:N/2);
```

```matlab
% Step 4: Calculate Error
error1 = filtered_sine1_mat - filtered_sine1_ver;
error2 = filtered_sine2_mat - filtered_sine2_ver;
error3 = filtered_sine3_mat - filtered_sine3_ver;
error4 = filtered_sine4_mat - filtered_sine4_ver;

% Mean Squared Error (MSE)
mse1 = mean(error1.^2);
disp(mse1)
mse2 = mean(error2.^2);
disp(mse2)
mse3 = mean(error3.^2);
disp(mse3)
mse4 = mean(error4.^2);
disp(mse4)

% Step 5: Plot Frequency-Domain Representation
figure;
subplot(4,1,1);
plot(f_axis, fft_sine1_mat, 'b', 'LineWidth', 1.5); hold on;
plot(f_axis, fft_sine1_ver, 'r--', 'LineWidth', 1.5);
title('Filtered Signal-1 (100Hz)');
xlabel('Frequency (Hz)'); ylabel('Magnitude');
legend('MATLAB', 'Verilog');
grid on;

subplot(4,1,2);
plot(f_axis, fft_sine2_mat, 'b', 'LineWidth', 1.5); hold on;
plot(f_axis, fft_sine2_ver, 'r--', 'LineWidth', 1.5);
title('Filtered Signal-2 (2000Hz)');
xlabel('Frequency (Hz)'); ylabel('Magnitude');
legend('MATLAB', 'Verilog');
grid on;

subplot(4,1,3);
plot(f_axis, fft_sine3_mat, 'b', 'LineWidth', 1.5); hold on;
plot(f_axis, fft_sine3_ver, 'r--', 'LineWidth', 1.5);
title('Filtered Signal-3 (6000Hz)');
xlabel('Frequency (Hz)'); ylabel('Magnitude');
legend('MATLAB', 'Verilog');
grid on;

subplot(4,1,4);
plot(f_axis, fft_sine4_mat, 'b', 'LineWidth', 1.5); hold on;
plot(f_axis, fft_sine4_ver, 'r--', 'LineWidth', 1.5);
title('Filtered Signal-4 (11000Hz)');
xlabel('Frequency (Hz)'); ylabel('Magnitude');
legend('MATLAB', 'Verilog');
grid on;
```

Frequency response plots of Filtered signals in MATLAB and Verilog are



Mean Square Error Between Verilog and Matlab Values for 4 filtered signals

For 100Hz Signal: `9.4222e-09`

For 2000Hz Signal: `3.5928e-09`

For 6000Hz Signal: `1.9178e-09`

For 11000Hz Signal: `2.2493e-09`

<u>OBSERVATIONS</u>:

I noticed that the MATLAB and Verilog outputs look almost the same, which means the filter is working correctly. There are small differences because Verilog uses fixed-point numbers (Q2.14) and works with binary values. The lower frequency signals (100Hz, 2000Hz, and 6000Hz) pass through just fine, while the 11000Hz signal is reduced, just like it should be.

CONCLUSION:

The FIR filter works properly in both MATLAB and Verilog. There are small differences because Verilog uses binary numbers due to which errors occurred during conversion to floating point numbers. The experiment shows that a filter designed in MATLAB can be successfully used in hardware. This proves that the filter works well for real-time applications.