FIR Filter Transposed Structure

Basem Hesham Tawfik

Outlines

1. What is FIR filter ?
2. FIR Implementation
   - FIR Direct architecture
   - FIR Transposed architecture
   - FIR Transposed architecture operation
3. Design Filter Specifications
4. Design of FIR Filters by Windowing
5. MATLAB Modelling
   - Magnitude and Phase Response , Pole-Zero Plot , Filter Coefficients
   - Convert coefficients from decimal to binary
6. Simulatiuon

What is FIR filter ?

- An FIR (Finite Impulse Response) filter is a type of digital filter used in signal processing. It operates by convolving a finite-length input signal with a series of coefficients, which are typically called the filter taps. These coefficients determine how the input signal is weighted and combined to produce the output signal.

- FIR Filters have no feedback constant coefficient and can be expressed as follows:

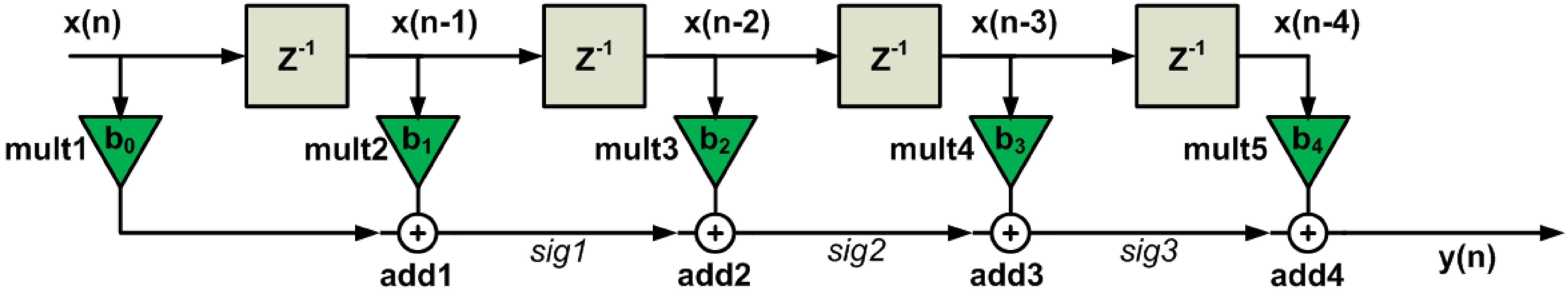
$$H(Z) = \sum_{k=0}^{N-1} h(k)\ Z^{-k}$$

$$= h(0) + h(1)\ Z^{-1} + h(2)\ Z^{-2} + \cdots\cdots$$

- $h(k) \rightarrow$ coefficients of filter
- N $\rightarrow$ Filter length (Number of filter coefficients)

FIR Direct architecture

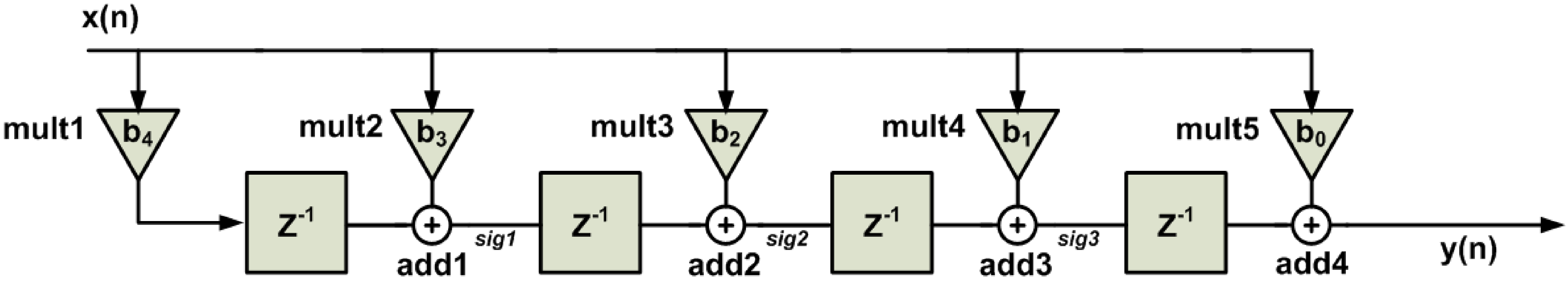
- Digital filters are implemented using the basic building block elements of adders, multipliers, and shift registers. How these elements are arranged and interconnected defines a filter ’ s architecture. In general, a given filter can have multiple architectures that can be used to implement a common transfer function.



The direct form of a five-tap FIR filter.

FIR Transposed architecture

- Another baseline FIR architecture is called the transpose FIR, which is a variation of the direct architecture theme. An FIR, with an impulse response $h[k] = \{h_0\ , h_1\ , \ldots , h_{N-1}\}$ can be implemented as the transpose architecture shown in the following Figure
- Comparing with the direct form we observe that the order of the filter coefficients is reversed, And the input reaches all the multipliers at the same time This is in contrast to the direct form structure where a given input sample reaches the multipliers at different clock cycles.



The transposed form of a five-tap FIR filter

FIR Transposed architecture Operation

- One of the most important features of this structure is its self-pipelined operation. To understand this, let's see how a new sample is processed by the transposed structure. We'll examine the circuit in different clock cycles:
- **The 1st clock:** Assume that, at the first clock edge, a new sample is applied to the filter. After a delay of T_{mult}, the multiplier mult1 will output $b_4\, x(n)$.
- **The 2nd clock:** At the second clock edge, the output of mult1, which is $b_4\, x(n)$, will be stored in the leftmost register. The register introduces a unit delay; hence, the content of the register will be $b_4\, x(n-1)$ .This means that the register stores b_4 multiplied by the previous sample of the input. To further clarify, note that, we are at the second clock cycle and the stored value corresponds to the sample taken in the first clock cycle.
- Besides, with a delay of T_{mult} , mult2 will output b_3 times the current input which is $x(n)$ .Hence, with a delay of $T_{mult} + T_{add}$ after the second clock edge, we have $sig1 = b_4\, x(n-1) + b_3\, x(n)$
- **The 3rd clock:** at the third clock edge, $sig1 = b_4\, x(n-1) + b_3\, x(n)$ will be stored in the corresponding register. Moreover, mult3 will output b_2 times the current input which is $x(n)$. Hence, with a delay of $T_{mult} + T_{add}$ after the third clock edge, we have

$$sig2 = b_4\, x(n-2) + b_3\, x(n-1) + b_2\, x(n)$$

FIR Transposed architecture Operation

- **The 4th clock:** with a delay of $T_{mult} + T_{add}$ after the fourth clock edge, we have

$$sig2 = b_4\, x(n-3) + b_3\, x(n-2) + b_2\, x(n-1) + b_1 x(n)$$

- **The 5th clock:** with a delay of $T_{mult} + T_{add}$ after the fifth clock edge, we have

$$y(n) = b_4\, x(n-4) + b_3\, x(n-3) + b_2\, x(n-2) + b_1 x(n-1) + b_0\, x(n)$$

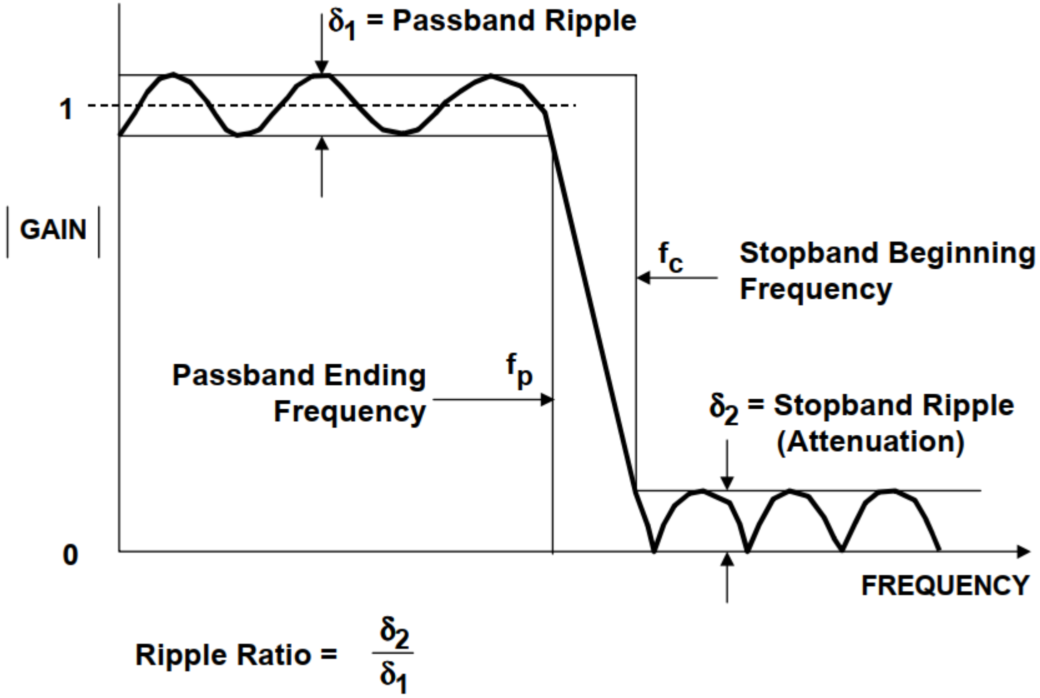
- This is the value of the output during the 5th clock cycle.

- If we consider the transposed structure during different clock cycles, we observe that the registers are storing the final result calculated by all the previous stages. Hence, these previous stages can be used to process new samples the transposed structure is inherently a pipelined implementation.

Filter Specifications

- we will design an audio lowpass filter that operates at a sampling rate of 44.1kHz which are standard sampling rate for audio applications. Let's say you want to design a FIR LPF for audio signal processing with a cutoff frequency of 4 kHz. We must also specify the word length of the coefficients, which in this case is 16 bits, assuming a 16-bit fixed-point DSP is to be used.
- We can design FIR LPF using various methods such as windowing, frequency sampling, or optimization techniques. The filter length and the coefficients will depend on the specific design method you choose and the desired filter characteristics. We will use window method in our design.

- ❑ Filter Type: Lowpass
- ❑ Sampling Frequency: 44,100Hz
- ❑ Cutoff Frequency: 4,000Hz
- ❑ Word length: 16-bits



Design of FIR Filters by Windowing

- In this method, we start from the required or desired frequency response of the filter $H_d[\omega]$ in the frequency domain, and from it we calculate the impulse response of this filter $h_d[n]$, where each of the previous two responses is linked to a Fourier transform relationship as follows:

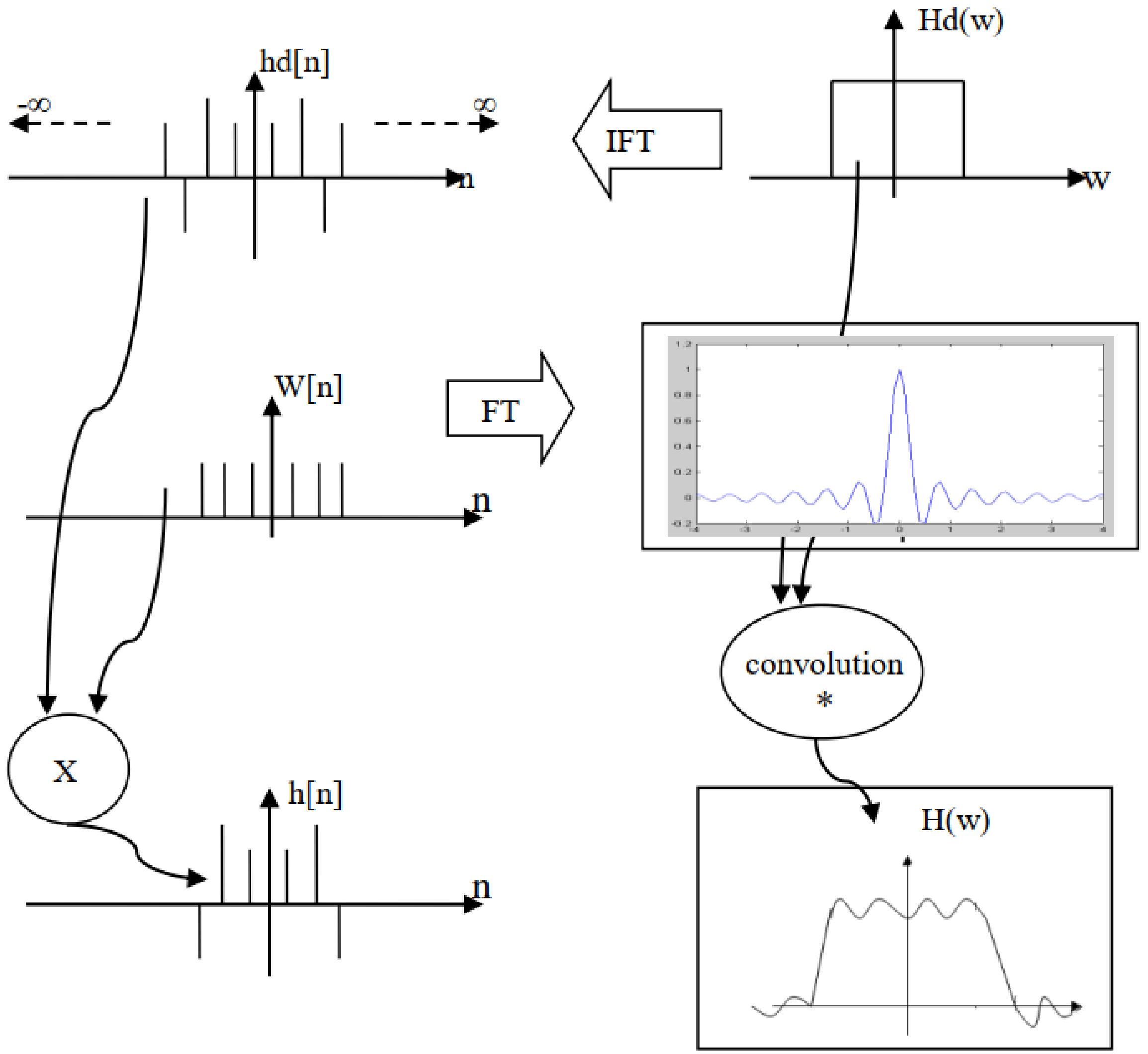
$$H_d[\omega] = \sum_{n=-\infty}^{\infty} h_d[n] e^{-j\omega n} \qquad , \qquad h_d[n] = \frac{1}{2\pi}\int_{-\pi}^{\pi} H(\omega) e^{j\omega n}\, d\omega$$

- Therefore, by knowing the frequency response, the impulse response $h_d[n]$ can be deduced using previous equation. Unfortunately, this response $h_d[n]$ is infinite in length in the positive and negative direction of the variable n. Therefore, to obtain a specific length for the impulse response, we will truncate a number M-1 from Samples from the response $h_d[n]$, and this will be done by multiplying the response $h_d[n]$ in a window W[n] whose length is M-1 of samples as follows :

$$h[n] = h_d[n]\, \mathrm{W}[n]$$

$$h[n] = \begin{cases} h_d[n] & n = 0,1,2,\dots,M-1 \\ 0 & otherwise \end{cases}$$

Window Design Method

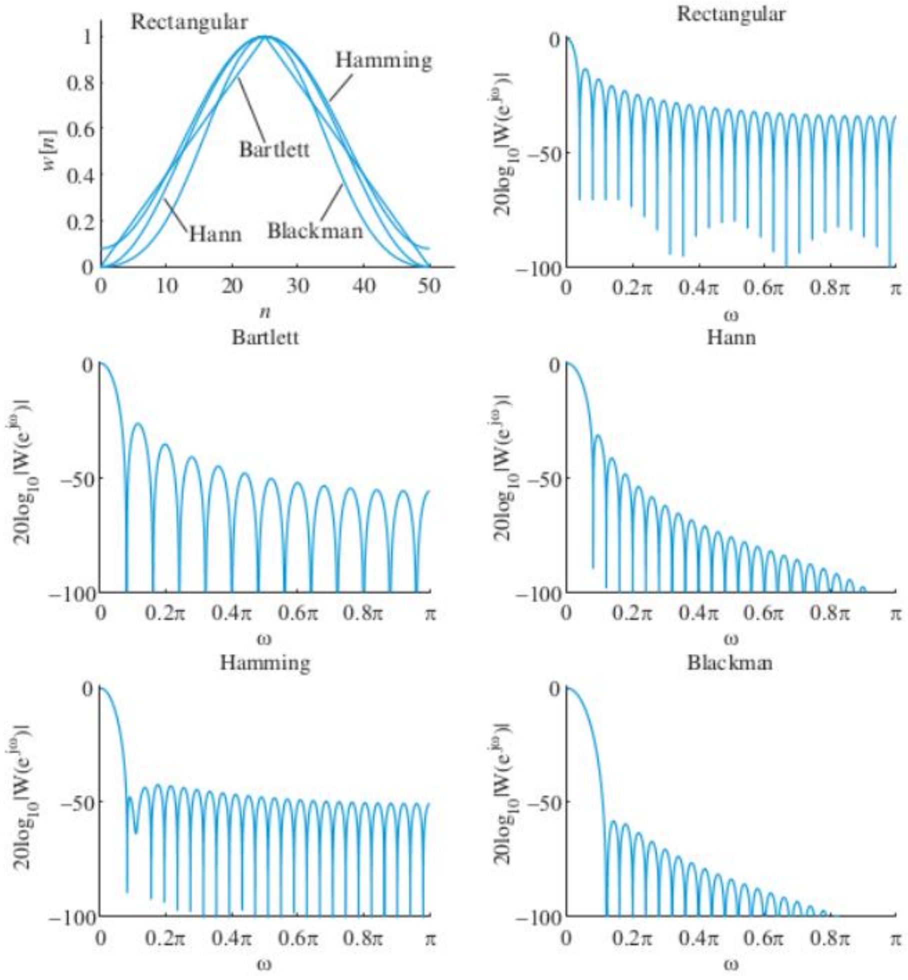


Obtain a FIR filter using a square window [w]

Design of FIR Filters by Windowing

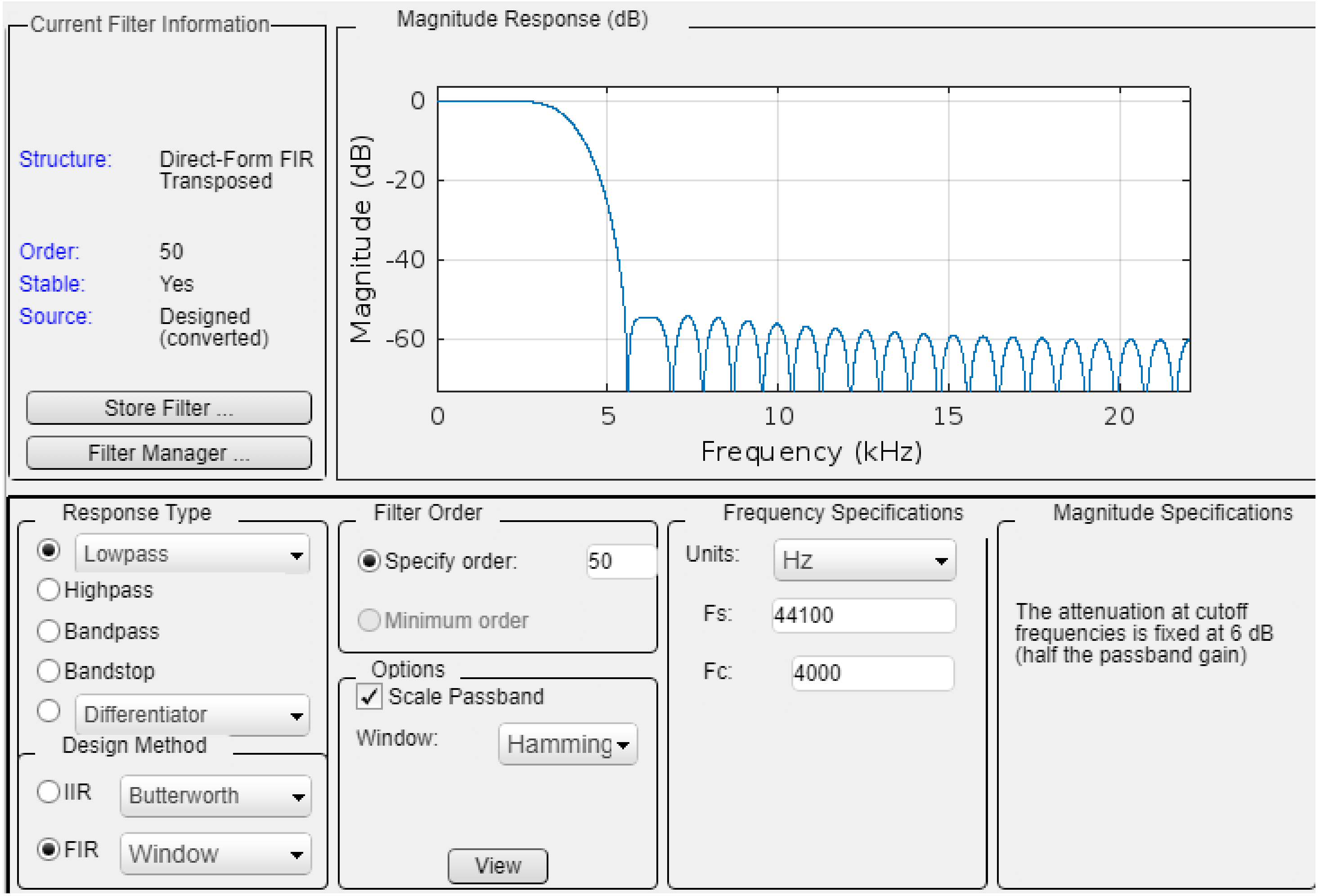
- Using non-rectangular windows to obtain a less abrupt truncation of the impulse response reduces the height of the ripples at the expense of a wider transition band. The most commonly used windows are: Rectangular, Bartlett (triangular), Hann, Hamming, and Blackman.

- In our design we will use Hamming window.

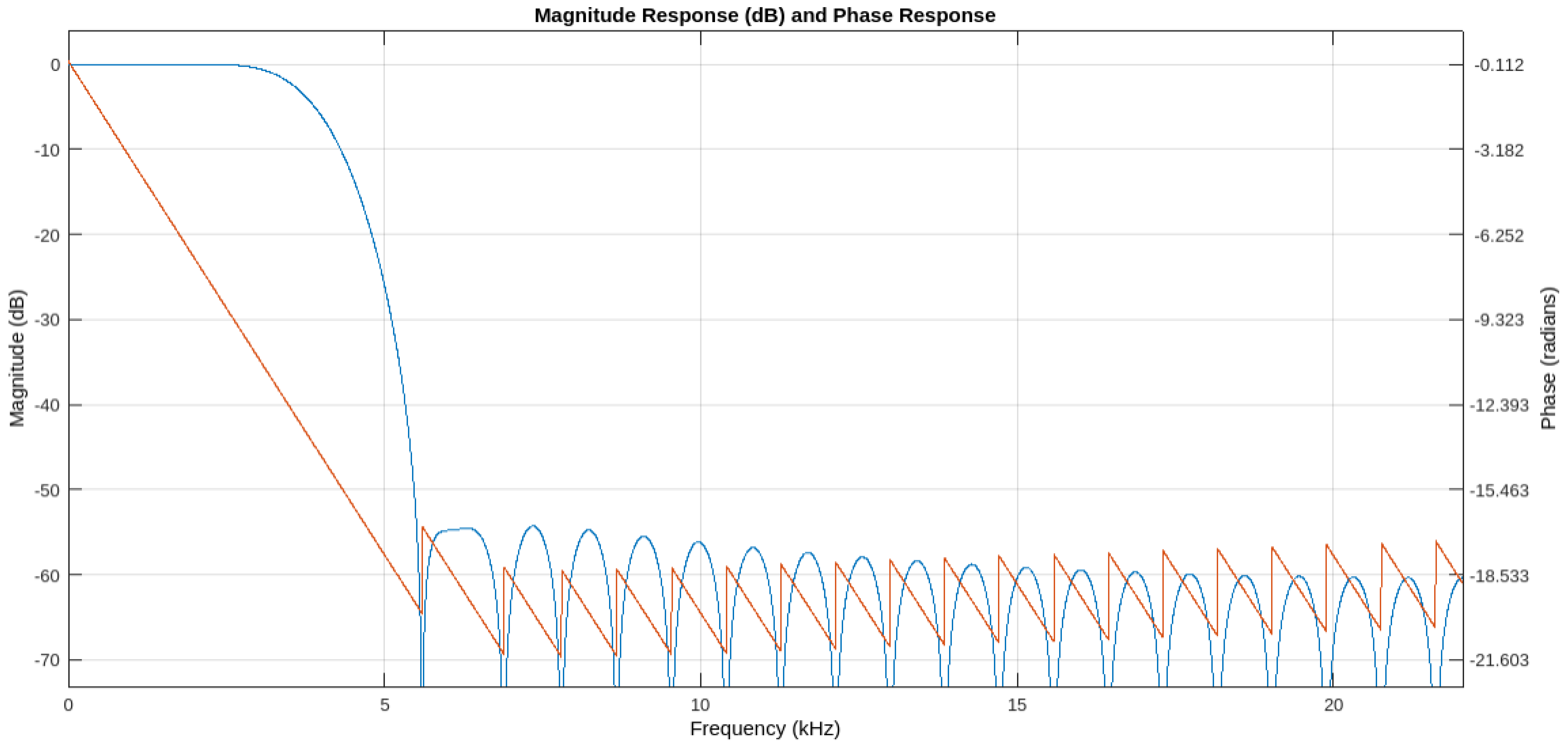


MATLAB Modelling

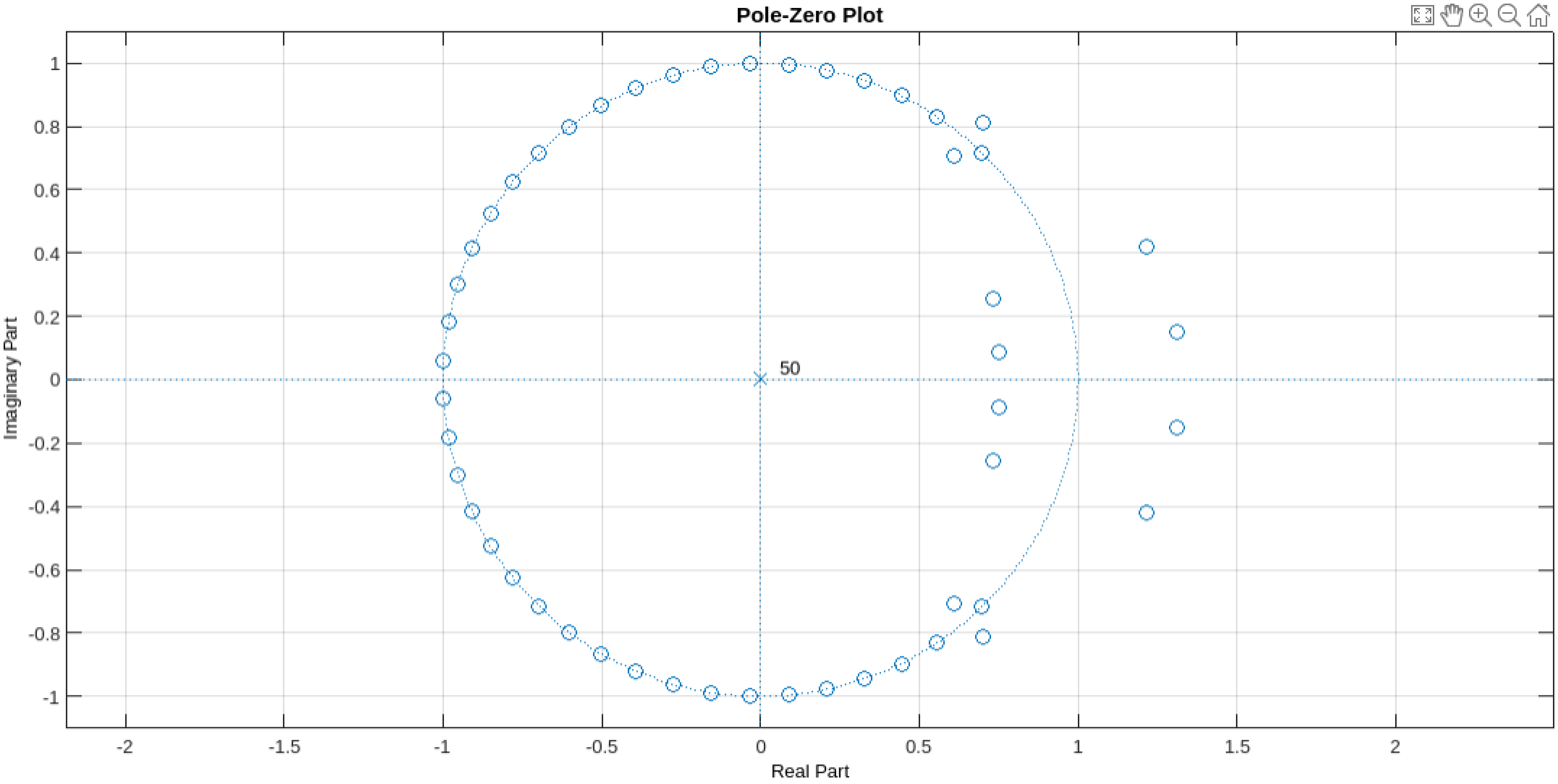
- Now we can design our filter based on the previous specifications using Filter Designer tool by the following command: filterDesigner



Magnitude and Phase Response



Pole-Zero Plot



Filter Coefficients

- The following step is to get the filter coefficients from File>Export>Coefficient File (ASCII) and choose decimal format.

Coff.txt

/MATLAB Drive/Coff.txt

```
% Generated by MATLAB(R) 24.1 and Signal Processing Toolbox 24.1.
% Generated on: 08-May-2024 18:55:22

% Coefficient Format: Decimal

% Discrete-Time FIR Filter (real)
% -------------------------------
% Filter Structure  : Direct-Form FIR Transposed
% Filter Length     : 51
% Stable            : Yes
% Linear Phase      : Yes (Type 1)


Numerator:
 0.0010111344633320566537004747686978589 63
 0.000992870575472371659150883083100325166
 0.000672807298663284625815650397129275007
-0.000046237285137533990799201671917373346
-0.001167475630147992013072033579135222681
-0.002454921774698606146136725669748557266
-0.003376816650393119068063185750361299142
-0.003226657669108042806455083351124812907
-0.001430120806642840632777469167535855377
 0.002062032014757390722958785289620209369
 0.006478726383710022712625331564595398959
 0.010225136057689852148167020118307846133
 0.011280793190062929701178440211606357479
 0.007950180787007817689859834331400634255
-0.000257843171964013562896811393443385896
```

Convert coefficients from decimal to binary

- To represent floating number in binary we use **python script** truncate the numbers to just the 7 digits after the decimal point and write the new values after truncated then converted it to binary.

main.py | FilterDecCoff.txt | binary_coefficients...

```python
# Read the content of the file
with open("FilterDecCoff.txt", "r") as file:
    numbers = file.readlines()

# Process the numbers: truncate to 7 digits after the decimal point
truncated_numbers = [format(float(num), '.7f') + '\n' for num in numbers]

# Write the truncated numbers back to the file, overwriting its contents
with open("FilterDecCoff.txt", "w") as file:
    file.writelines(truncated_numbers)

def float_to_binary(f):
    # Convert floating-point number to 16-bit binary representation
    sign_bit = "1" if f < 0 else "0"
    abs_f = abs(f)

    # Round off the result
    rounded_result = round(abs_f * (2**15))

    # Convert to 15-bit binary representation
    binary_representation = format(rounded_result, "015b")

    # If the number was negative, get 2's complement
    if f < 0:
        binary_representation = bin(
            1
            + int(
                "".join("1" if bit == "0" else "0" for bit in binary_representation), 2
            )
        )[2:]

    return sign_bit + binary_representation

```

```python

def convert_coefficients(input_file, output_file):
    # Read coefficients from the input file
    with open(input_file, "r") as file:
        coefficients = [float(line.strip()) for line in file]
    # print(coefficients)
    # Convert coefficients to binary representation
    binary_coefficients = [float_to_binary(coeff) for coeff in coefficients]
    # print(binary_coefficients) and Save binary coefficients to the output file
    with open(output_file, "w") as file:
        for binary_coeff in binary_coefficients:
            file.write(binary_coeff + "\n")


if __name__ == "__main__":
    # Provide the input and output file names
    input_file_name = "FilterDecCoff.txt"
    output_file_name = "binary_coefficients.txt"

    # Call the function to convert coefficients and save to the output file
    convert_coefficients(input_file_name, output_file_name)
```

Convert coefficients from decimal to binary

in.py | FilterDecCoff.txt

```
0.0010111
0.0009929
0.0006728
-0.0000462
-0.0011675
-0.0024549
-0.0033768
-0.0032267
-0.0014301
0.0020620
0.0064787
0.0102251
0.0112808
0.0079502
-0.0002578
-0.0119599
-0.0237687
-0.0308884
-0.0283730
-0.0127205
0.0166997
0.0569020
```

Coefficients (Floating numbers) after truncated

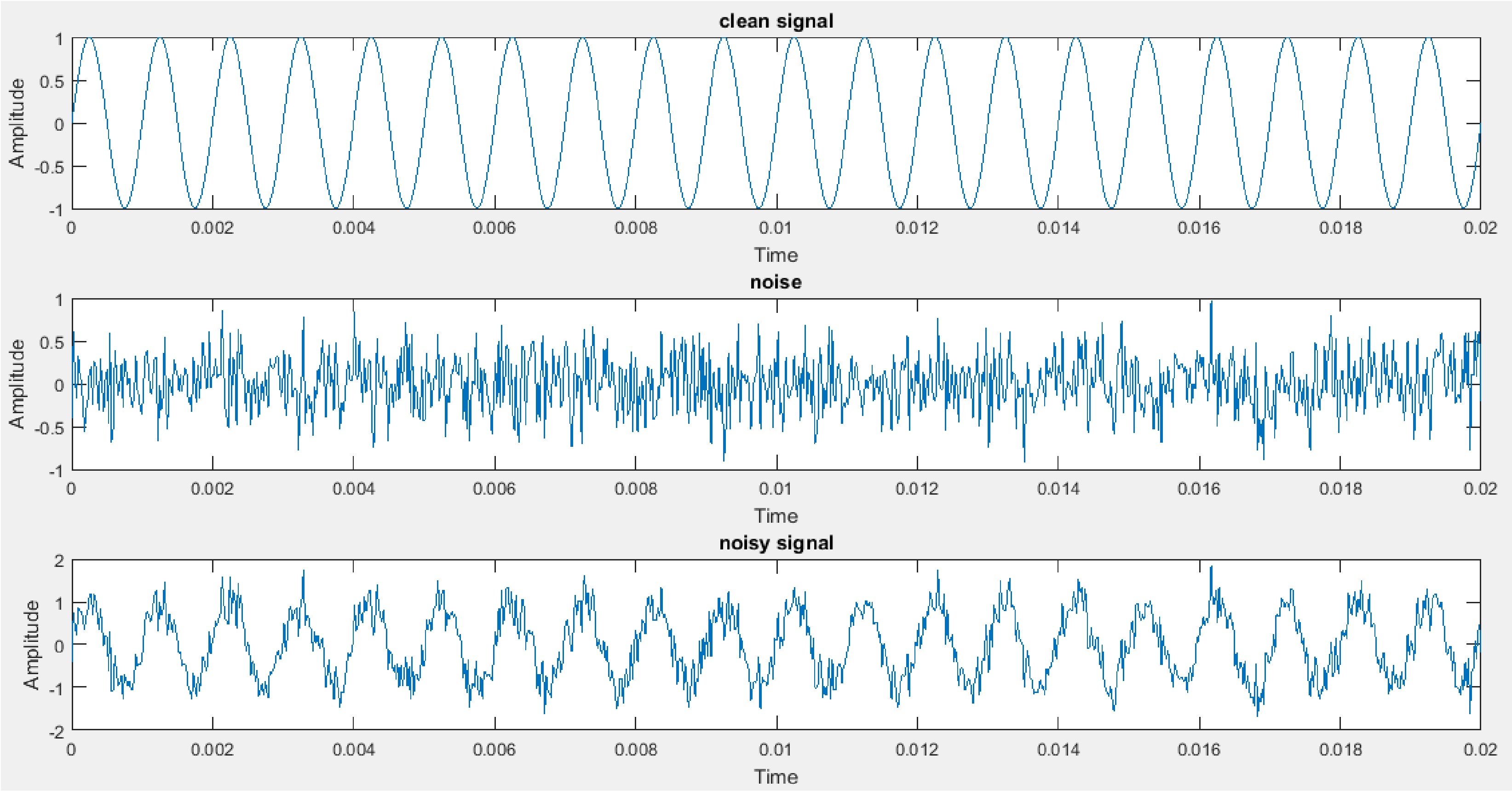
n.py | FilterDecCoff.txt | binary_coefficients.

```
0000000000100001
0000000000100001
0000000000010110
1111111111111110
1111111111011010
1111111110110000
1111111110010001
1111111110010110
1111111111010001
0000000001000100
0000000011010100
0000000101001111
0000000101110010
0000000100000101
1111111111111000
1111111001111000
1111110011110101
1111110000001100
1111110001011110
1111111001011111
0000001000100011
0000011101001001
```

Coefficients after converted to binary

Generating Noisy Signal

- This MATLAB script generates a noisy signal by adding random noise to a clean sinusoidal signal.
- the script first calculates the power of the noise based on the specified SNR, and then generates Gaussian noise samples with that power to add to the clean signal.




```
Fs = 44100;     % Sampling frequency (Hz)
SNR = 10;       % Signal-to-noise ratio (dB)

% Generate time vector
t = 0:1/Fs:0.02;

% Generate clean signal
clean_signal = sin(2*pi*1000*t);
```

```
% Generate noise
noise_power = 10^(-SNR/10);
noise = sqrt(noise_power) * randn(size(t));

% Add noise to the clean signal
noisy_signal = clean_signal + noise;
```

```
% Add noise to the clean signal
noisy_signal = clean_signal + noise;
```

Generating Noisy Signal

- Convert this noisy signal into binary within a range of 16-bit and create file

```matlab
% Scale the signal to fit within the range of a 16-bit integer
scaled_signal = int16(noisy_signal * (2^15 - 1));

% Convert the scaled signal to binary representation
binary_signal = dec2bin(typecast(scaled_signal, 'uint16'), 16);

% Save each 16-bit binary value on a separate line in the text file
fileID = fopen('binary_signal.txt', 'w');
for i = 1:size(binary_signal, 1)
    fprintf(fileID, '%s\n', binary_signal(i, :));
end
fclose(fileID);
```

sin_generator.m   binary_signal.txt

```
1100111001011100
0100010000010100
1110111110101001
0100000000110111
0111111111111111
0111111010010011
0111111111111111
0111100111001001
0111110010111010
0001000110111000
0010110110101111
1100000110000001
1111110010011001
1010111100100011
1011101010001110
1000000000000000
1000111011001110
1010101111000111
1000000000000000
1000000000000000
1000000000000000
```

Output file (binary_signal.txt)

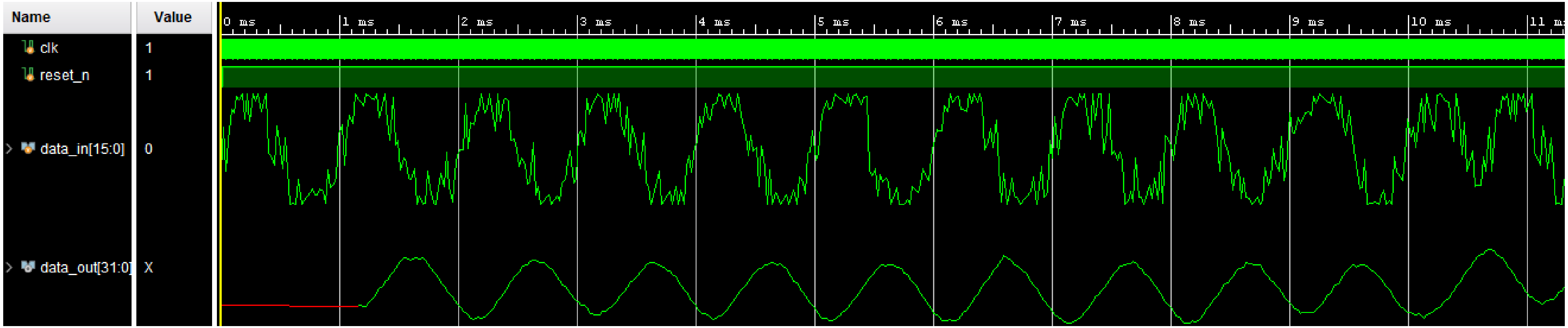
Simulation (Test case 1)

after the fifty-clock edge we have the output

$$y(n) = b_{50}\, x(n-50) + b_{49}\, x(n-49) + \cdots + b_2\, x(n-2) + b_1 x(n-1) + b_0\, x(n)$$

data_in : noisy signal = $(\sin(2\pi * 1000 * t) +$ noise)

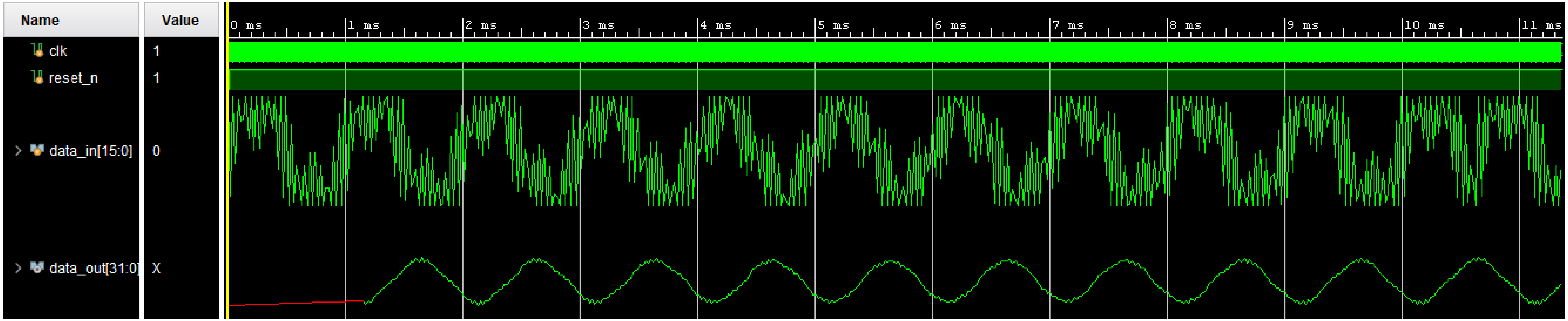
data_out : filtered signal = $\sin(2\pi * 1000 * t)$



We can notice that the original signal is recovered successfully with an acceptable delay that was expected due to the filter hardware stages.

Simulation (Test case 2)

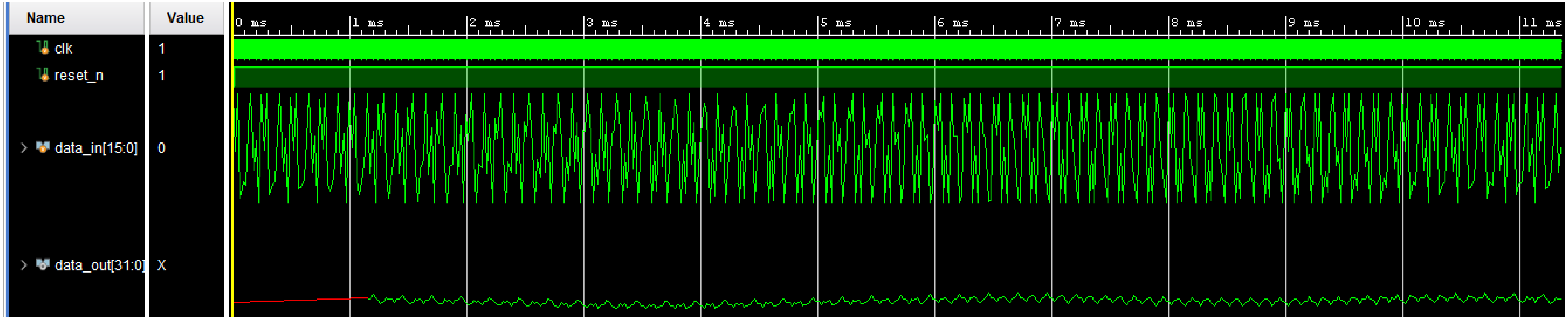
data_in : signal with two frequencies $= \sin(2\pi * 1000t) + \cos(2\pi * 20000t)$;
data_out : filtered signal = $\sin(2\pi * 1000t)$



We can notice that the frequency 20KHz is filtered and frequency 2KHz is passed

Simulation (Test case 3)

data_in : signal with two high frequencies $= \sin(2\pi * 8000t) + \cos(2\pi * 20000t)$;
data_out : filtered signal ≈ 0

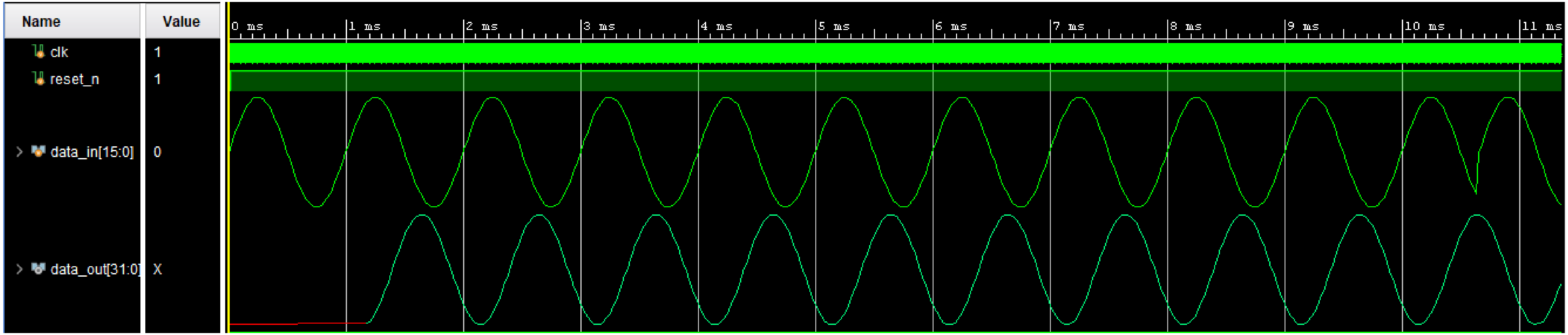


We can notice that the both frequencies 20KHz and 8KHz is filtered

Simulation (Test case 4)

data_in : $\sin(2\pi * 1000t)$

data_out : the same signal $= \sin(2\pi * 1000t)$



We can notice that the frequency 1KHz which is below cutoff frequency is passed

Thank You

The code for this project can found in my GitHub repository

GitHub