FIR FILTERS USING VERILOG

WHAT IS FIR FILTER:

Finite Impulse Response (FIR) Filters – Detailed Explanation

A **Finite Impulse Response (FIR) filter** is a type of digital filter characterized by a **finite number of nonzero terms in its impulse response**. It is widely used in digital signal processing (DSP) due to its stability and linear-phase characteristics.

---

1. Mathematical Formulation of FIR Filters

The output of an FIR filter is computed as a weighted sum of the most recent input samples. Mathematically, an FIR filter of order NNN is expressed as:

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

where:

- y[n] = output signal at time n,
- x[n]= input signal at time n,
- h[k]= filter coefficients (impulse response),
- N= order of the FIR filter (i.e., the number of taps - 1),
- k = index of filter coefficients.

This equation shows that FIR filters **only depend on present and past input values**, unlike IIR filters that also use previous output values.

---

2. Impulse Response of FIR Filter

The impulse response of an FIR filter is simply the sequence of its coefficients:

h={h[0],h[1],...,h[N]}

For an FIR filter, this impulse response **eventually becomes zero** after N samples, ensuring finite response duration.

---

3. Frequency Response of FIR Filters

The **frequency response** of an FIR filter is obtained by taking the Discrete-Time Fourier Transform (DTFT) of its impulse response:

H(ejω)=k=0∑Nh[k]e−jωk

where:

H(ejω) = frequency response,

ω\omegaω = normalized angular frequency.

This response determines how different frequency components of the input signal are attenuated or passed.

---

4. Types of FIR Filters

FIR filters can be designed to serve different purposes:

1. **Low-pass FIR filter** – Allows low-frequency signals and blocks high-frequency ones.
2. **High-pass FIR filter** – Allows high-frequency signals while blocking low-frequency ones.
3. **Band-pass FIR filter** – Allows signals within a specific frequency band.
4. **Band-stop (Notch) FIR filter** – Blocks a specific frequency range.
5. **Differentiator FIR filter** – Used to compute the derivative of a signal.

6. **Hilbert Transformer FIR filter** – Generates a signal with a 90-degree phase shift.

---

5. Design Methods for FIR Filters

There are various methods to design FIR filters, including:

1. Windowing Method

This is the simplest and most common method. The ideal filter response is truncated and multiplied by a window function to reduce artifacts.

Example: Low-pass FIR filter using a Hamming window

$h[k] = h_{ideal}[k] \cdot w[k]$

where:

- $h_{ideal}[k]$ = ideal impulse response,
- $w[k]$ = window function (Hamming, Hanning, Blackman, etc.).

2. Frequency Sampling Method

This method designs the filter directly in the frequency domain by specifying desired frequency response values.

3. Parks-McClellan (Remez Exchange) Algorithm

This is an optimization-based approach that minimizes the maximum error between the designed and ideal filter responses.

---

6. Advantages of FIR Filters

✅ **Always Stable** – Since they don't use feedback, FIR filters are inherently stable.
✅ **Linear Phase** – FIR filters can be designed to have a linear phase response, meaning no phase distortion.
✅ **Easier to Implement in Hardware** – Suitable for FPGA and DSP applications.

---

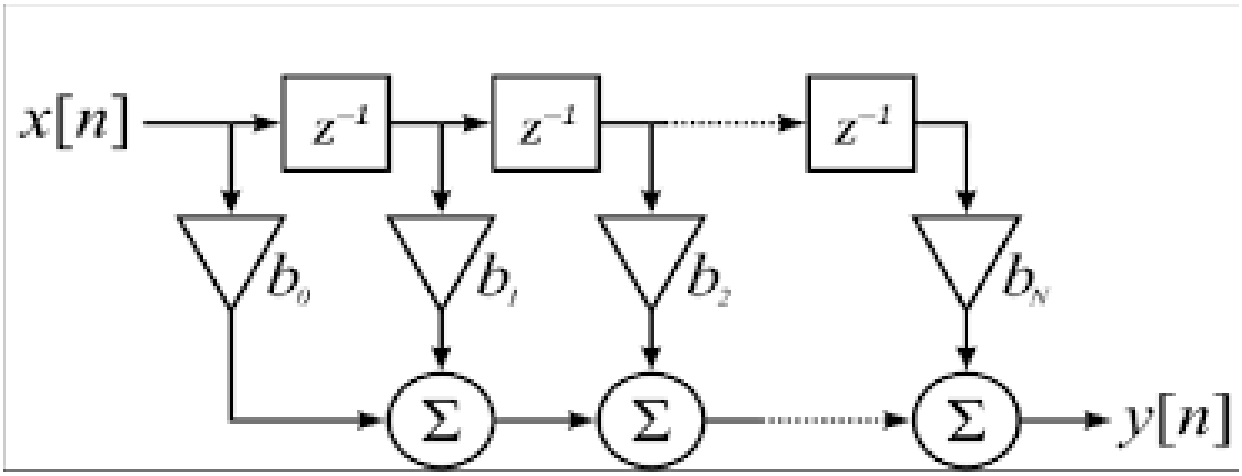
7. Disadvantages of FIR Filters

❌ **Higher Computational Cost** – Requires more memory and computation than IIR filters for similar performance.
❌ **More Coefficients Needed** – FIR filters require a higher filter order to achieve the same sharpness of cutoff as an IIR filter.

---

9. Applications of FIR Filters

- **Audio Processing** – Used in equalizers, noise reduction, and speech enhancement.
- **Image Processing** – Used in edge detection, smoothing, and sharpening filters.
- **Biomedical Signal Processing** – Used in ECG and EEG signal filtering.
- **Communication Systems** – Used in signal modulation, demodulation, and equalization.
- **Radar and Sonar Systems** – Used in detecting objects while filtering noise.
- **Stock Market Analysis** – Used to smooth financial time-series data.



Overview of FIR Filter Implementation

The FIR filter implemented in this Verilog code is a **third-order Moving Average FIR Filter**. The design uses a **clock-driven sequential process** to store previous input values and apply weighted coefficients to generate the filtered output.

Filter Specifications

- **Filter Type**: Moving Average FIR Filter (3rd order)
- **Number of Taps**: 4 (3 delays + 1 current input)
- **Scaling Factor**: 128
- **Coefficient Values**: 0.25 (scaled to integer values using 6-bit representation)
- **Bit Width**: 16 bits for input and output signals

Verilog Code Analysis

1. Module Definition

The FIR filter is implemented in the **firfilter** module. It has the following **inputs and outputs**:

- clk (Clock): Synchronizes the filter operations.
- reset (Reset): Clears stored values when activated.
- data_in (16-bit input): Incoming signal to be filtered.
- data_out (16-bit output): Filtered signal output.

module firfilter(clk, reset, data_in, data_out);

2. FIR Filter Coefficients

The FIR filter uses **four coefficients**, each having a value of 0.25 in floating point. To work with integer arithmetic, the coefficients are scaled by **128**, resulting in a 6-bit representation:

wire [5:0] b0 = 6'b100000;

wire [5:0] b1 = 6'b100000;

wire [5:0] b2 = 6'b100000;

wire [5:0] b3 = 6'b100000;

Each coefficient is applied to its corresponding delayed input.

3. Delay Elements Using D Flip-Flops

To store past input values, **three D Flip-Flops (DFFs)** are used, creating a shift register structure.

DFF DFF0(clk, 0, data_in, x1);

DFF DFF1(clk, 0, x1, x2);

DFF DFF2(clk, 0, x2, x3);

Each D Flip-Flop captures and holds the previous input value at each clock cycle.

4. Multiplication with Coefficients

Each stored input value is multiplied by its corresponding coefficient:

assign Mul0 = data_in * b0;

assign Mul1 = x1 * b1;

assign Mul2 = x2 * b2;

assign Mul3 = x3 * b3;

This operation implements the convolution sum of the FIR filter.

5. Summation of Multiplied Values

The results of the multiplications are summed to produce the final filter output:

assign Add_final = Mul0 + Mul1 + Mul2 + Mul3;

6. Output Register Assignment

The final computed sum is assigned to data_out on each clock cycle:

always@(posedge clk)

data_out <= Add_final;

This ensures that the output is updated synchronously with the clock.

7. D Flip-Flop (DFF) Implementation

The **DFF module** defines how previous input values are stored:

```
module DFF(clk, reset, data_in, data_delayed);
parameter N = 16;
input clk, reset;
input [N-1:0] data_in;
output reg [N-1:0] data_delayed;

always@(posedge clk, posedge reset)
begin
   if (reset)
      data_delayed <= 0;
   else
      data_delayed <= data_in;
end
endmodule
```

This module is responsible for **delaying the input signal** and passing it forward in the FIR filter.

---

8. Hardware Optimization Considerations

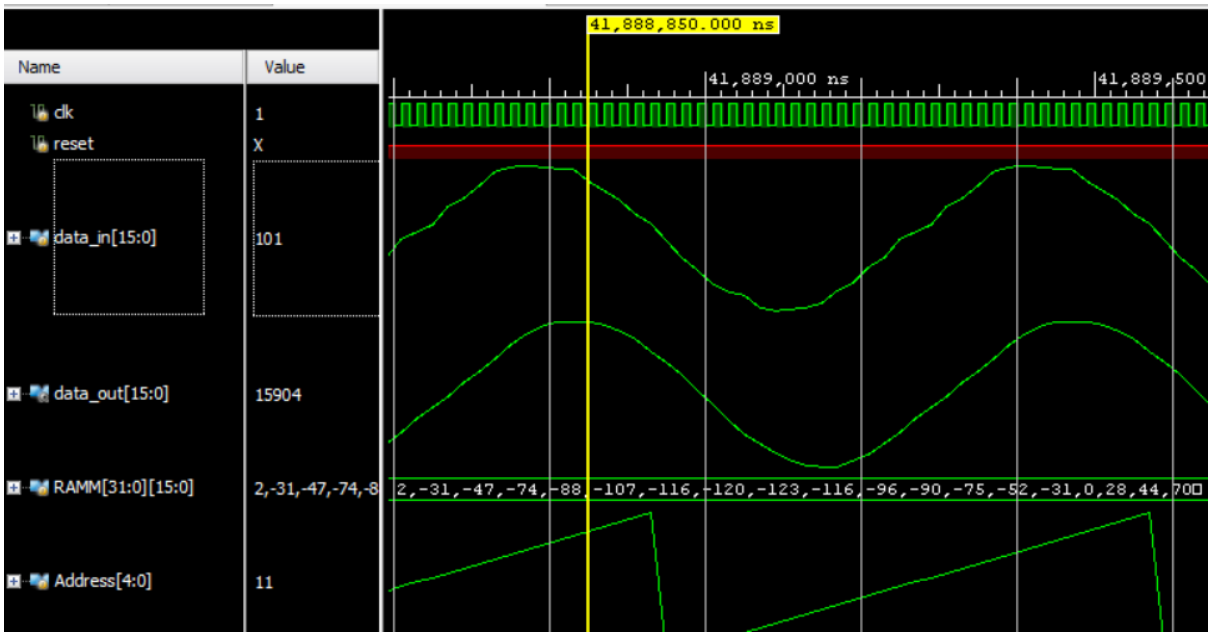
8.1 Fixed-Point Arithmetic and Precision

- Instead of using **floating-point multiplications**, the coefficients are scaled for **efficient integer arithmetic**.
- The **6-bit representation** balances precision and hardware efficiency.

8.2 Pipelining for Parallel Processing

- The design uses **D Flip-Flops as delay registers**, which implement **pipelining** for better throughput.
- This makes the filter **suitable for real-time FPGA applications**.

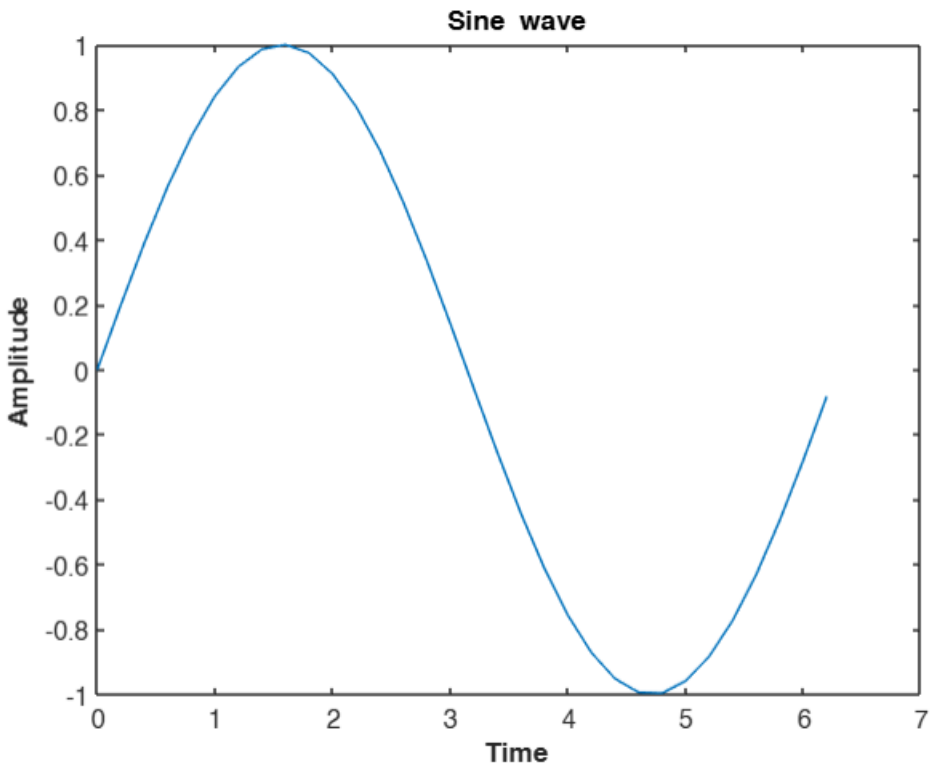
The simulation can be seen from the simulation that it is quite smooth then the input data:

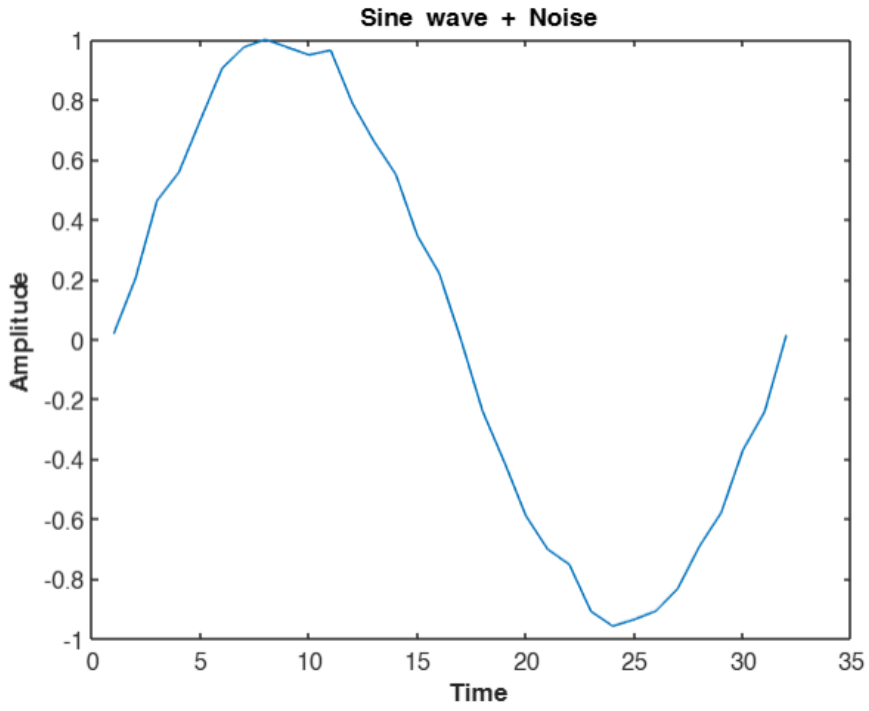


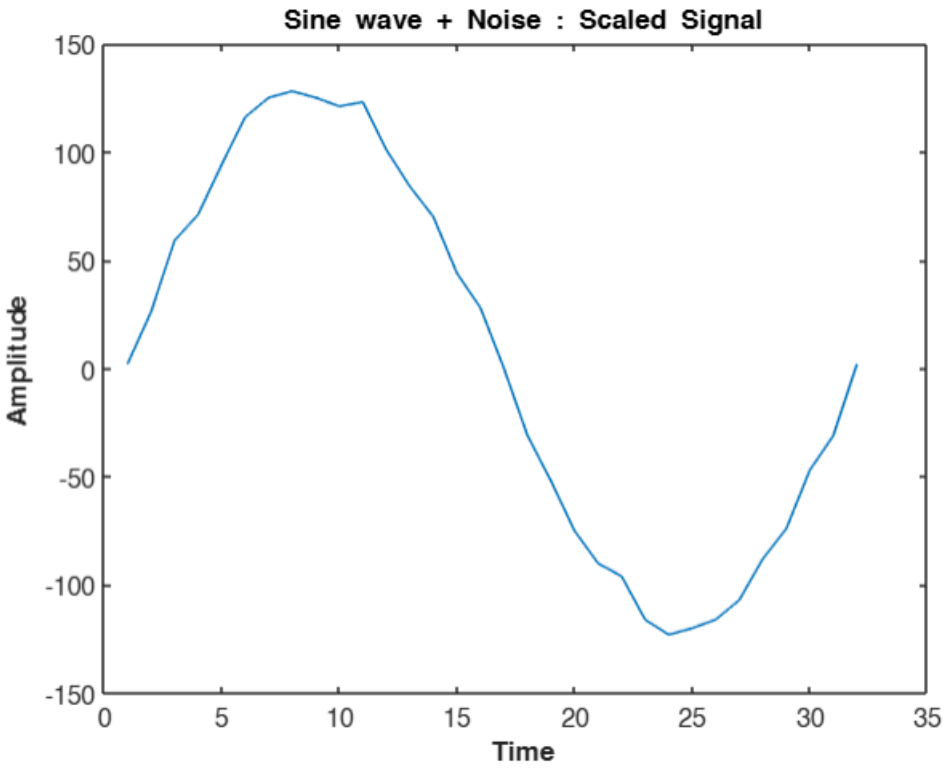
In the code the usage of a external software has been done named Matlab for the generation of the data files that is the binary number file for simulation whose graphs and code are shown below :

```
% Generate a sine wave
close all; clear all;
fs = 5;
Amp = 1;
t = 0:1/fs:2*pi; % time vector
sine_wave = Amp*sin(t);
figure();
plot(t, sine_wave);
xlabel('\bf Time');
```

```
ylabel('\bf Amplitude');
title('\bf Sine wave');

% Add a noise
a = 0.1; % upper limit
b = 0; % lower limit
noise = (b-a).*rand(length(sine_wave),1) + a; noise =
noise';
sine_noise = (sine_wave + noise);
sine_norm = sine_noise / max(abs(sine_noise));
figure();plot(1:length(sine_norm), sine_norm);
xlabel('\bf Time');
ylabel('\bf Amplitude');
title('\bf Sine wave + Noise');

% Convert from real to integers
total_wordlength = 16;
scaling = 7;
sine_noise_integers = round(sine_norm.*(2^scaling));
figure();plot(1:length(sine_noise_integers),
sine_noise_integers);
xlabel('\bf Time');
ylabel('\bf Amplitude');
title('\bf Sine wave + Noise : Scaled Signal');

% Convert from integers to binary
sine_noise_in_binary_signed =
dec2bin(mod((sine_noise_integers),2^total_wordlength),tota
l_wordlength);
yy = cellstr(sine_noise_in_binary_signed);
fid = fopen('signal.data', 'wt');
fprintf(fid, '%8s \n', yy{:});
disp('text file for signal finished');
```


Sine wave
Amplitude
Time
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
1
2
3
4
5
6
7


Sine wave + Noise
Amplitude
Time
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
5
10
15
20
25
30
35



Conclusion

This Verilog-based FIR filter efficiently performs **real-time signal filtering** using **fixed-point arithmetic, D Flip-Flops, and coefficient multiplication**. It provides a **low-pass filtering effect** with a **linear-phase response**, making it suitable for **audio processing, communication systems, and embedded applications**.