



DSP Project

FIR Filter

Verilog Hardware Design



Yaseen Salah
Digital IC Design Undergraduate



Design Flow



Getting Started with FIR Filter Concept



System Modeling Using **MATLAB**



FIR Filter Architecture (Block Diagram)



Hardware Design Using **Verilog**



Testbench and Simulation Using **Vivado**

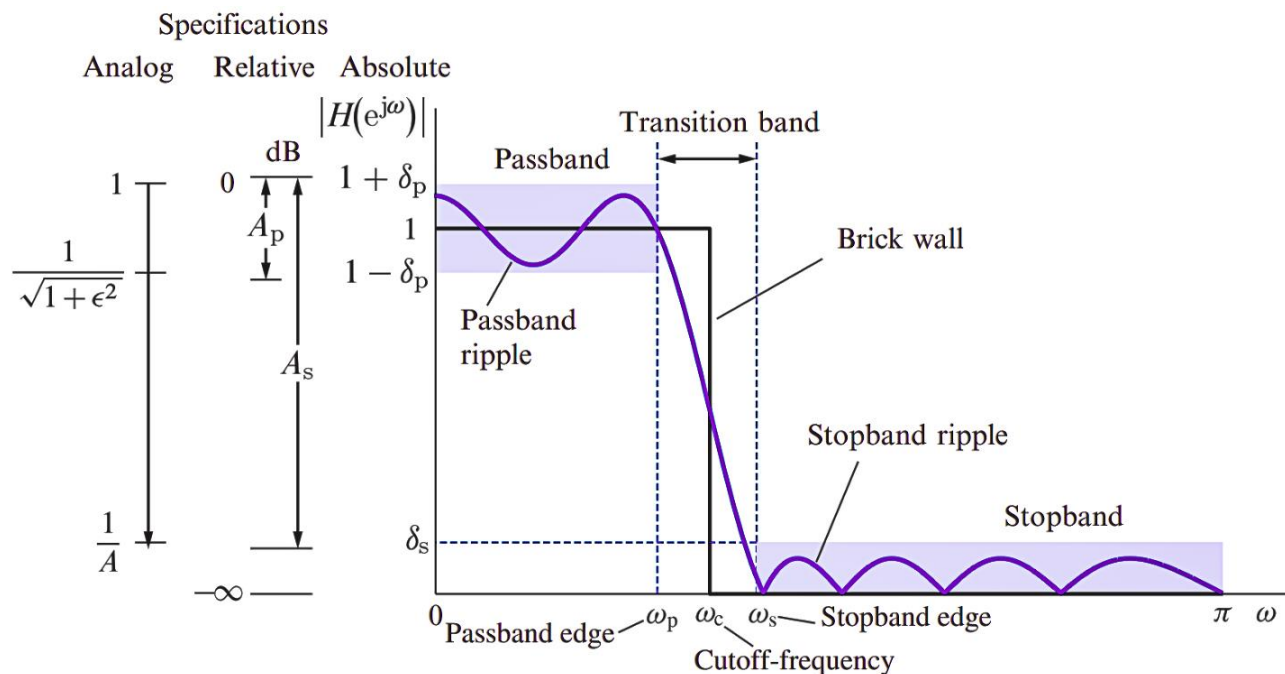


.WAV File Filtering Using **Simulink**



What is FIR Filter ?

Finite Impulse Response (FIR) Filter is a digital LTI system with an impulse response that settles to a zero value over a certain period of time, thus making it finite. It passes a set of desired frequency components from a mixture of desired and undesired components.



Example of tolerance diagram for a lowpass filter - Applied digital signal processing - Dimitris G. Manolakis, Vinay K. Ingle.

FIR Filter is featured by phase linearity and stability which make it suitable for systems that rely on the shape characteristics of a signal such as IoT devices and VUI systems where the shape of a voice signal should be preserved, and Neuron and cell monitoring devices. However, it requires higher order (more hardware) compared to IIR.



What is FIR Filter ?

We can generally describe discrete-time LTI systems through Linear Constant-Coefficient Difference Equation (LCCDE):

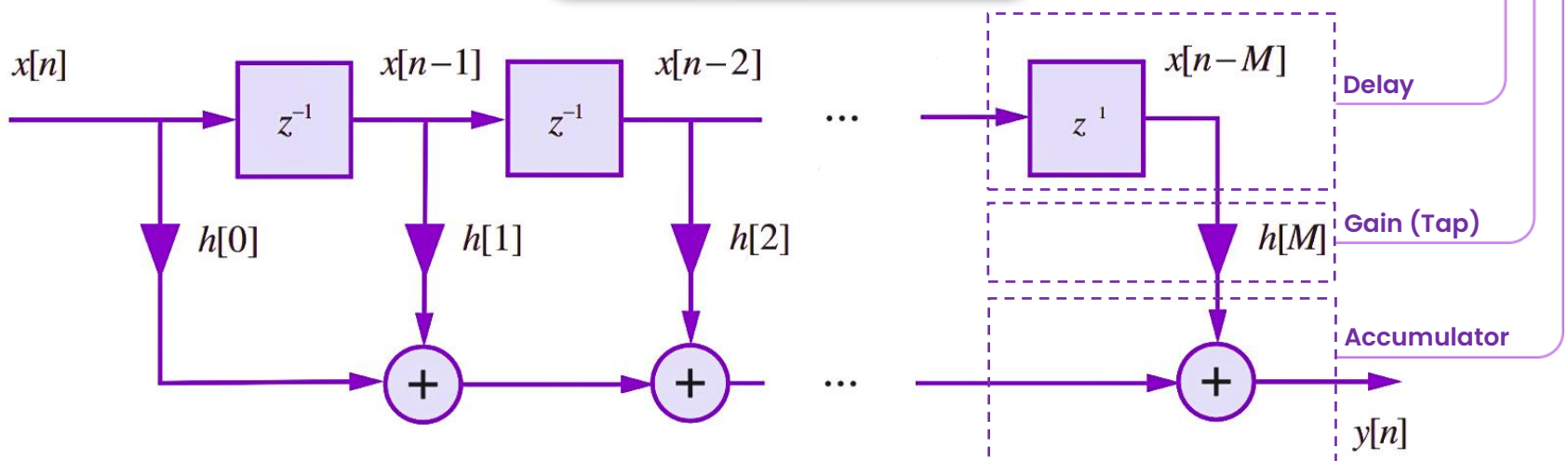
$$y[n] = - \sum_{k=1}^N a_k y[n-k] + \sum_{k=0}^M b_k x[n-k]$$

Feedback
Constant Coefficients

Feedforward
Constant Coefficients

Since the feedback causes the impulse response depends on old outputs and consequentially its non-zero samples extend to infinity such as Infinite Impulse Response (IIR) Filters, FIR Filters have no feedback constant coefficient and can be expressed as follows:

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





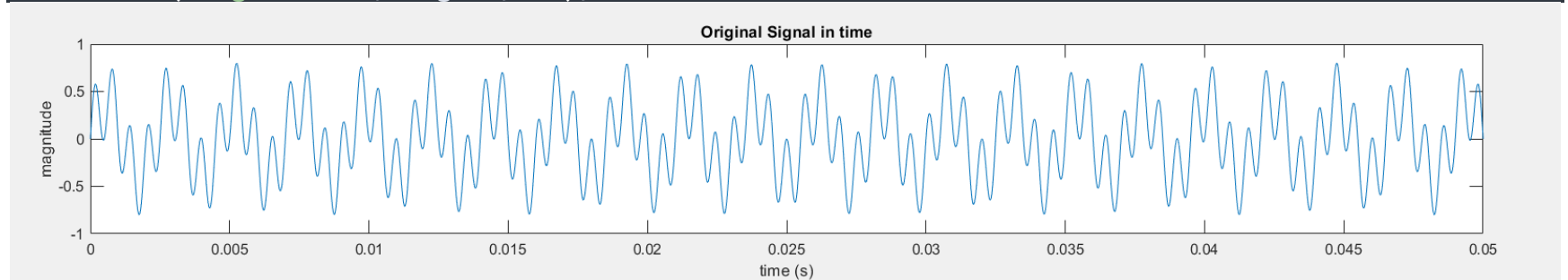
MATLAB Modeling

To determine our design requirements and filter specifications, let's assume we are dealing with audio signal from .wav file which has sampling rate 44.1 KHz.

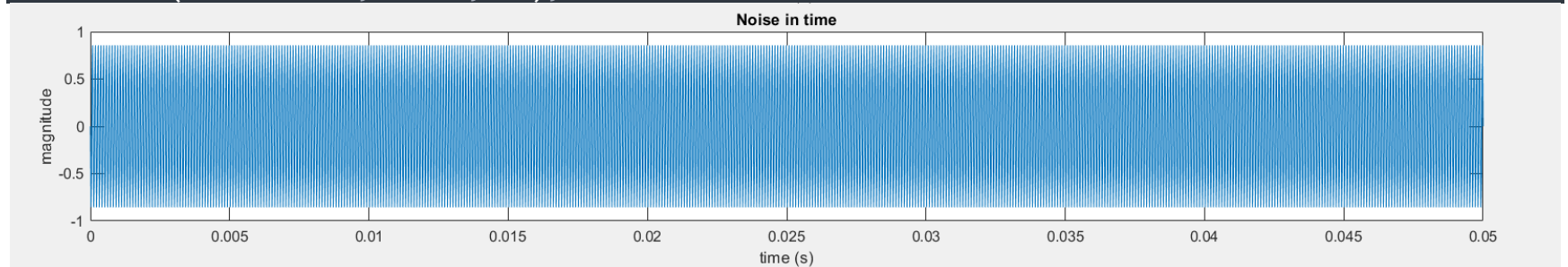
Creating a simple multi-frequency signal with 570 Hz and 1KHz and another signal that represents noise with high frequencies +40KHz.

```
Fs=44100;  
t = 0:1/50000:0.05;
```

```
signal = (sin(2*pi*1000*t)).*(0.8*cos(2*pi*570*t));  
audiowrite('signal.wav', signal, Fs);
```



```
noise = sin(2*pi*40000*t)+0.5*sin(2*pi*50000*t);  
audiowrite('noise.wav', noise, Fs);
```



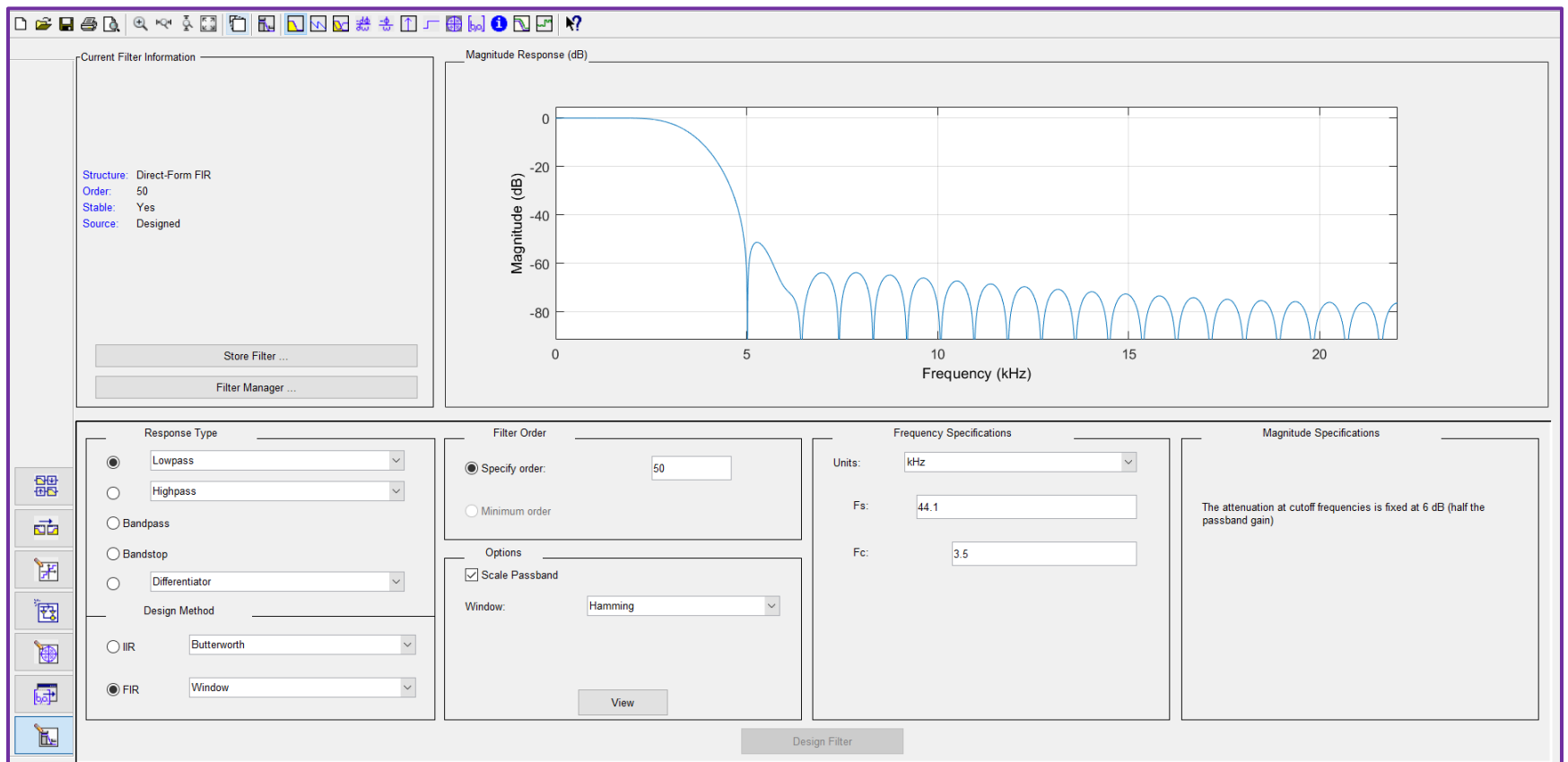


MATLAB Modeling

Now we can design our filter based on the previous ranges of frequencies using Filter Designer tool by the following command:

```
>> filterDesigner
```

Note: This command is supported only in MATLAB newer versions than 2018 instead of fdatool

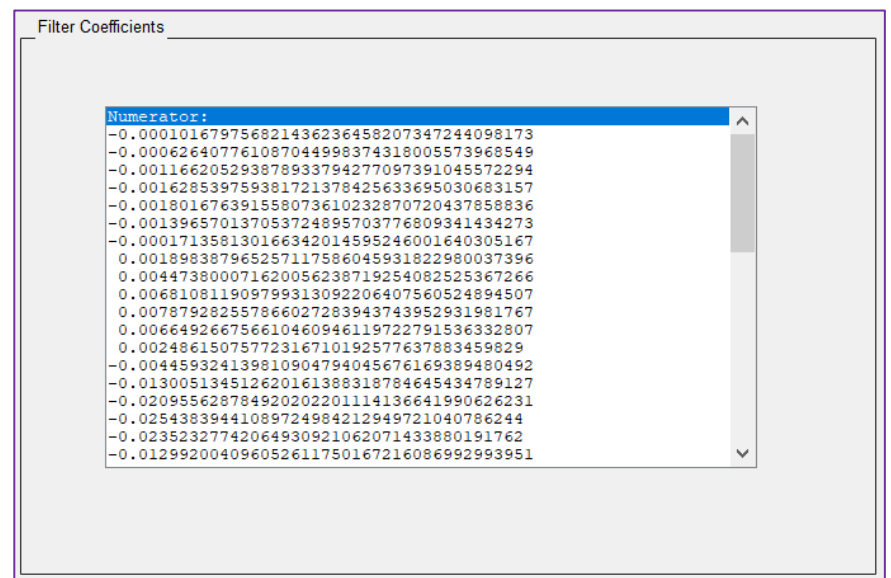
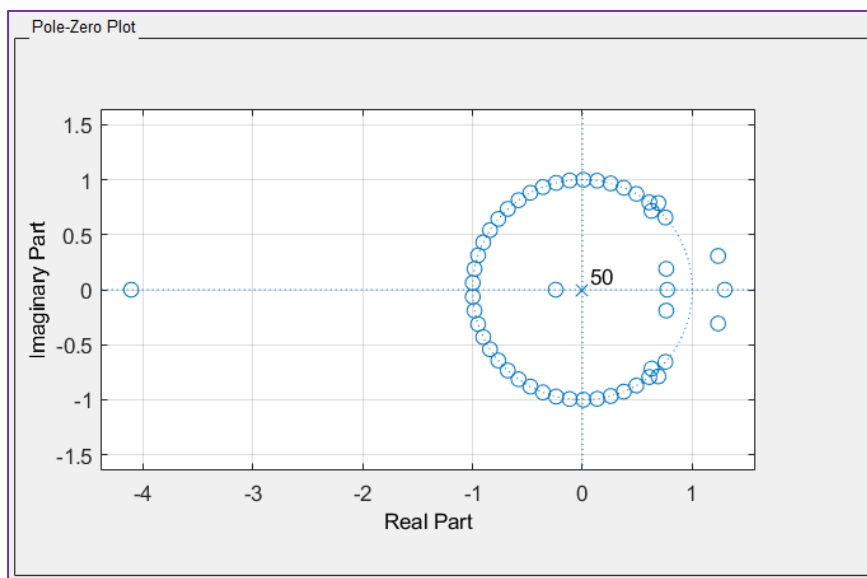
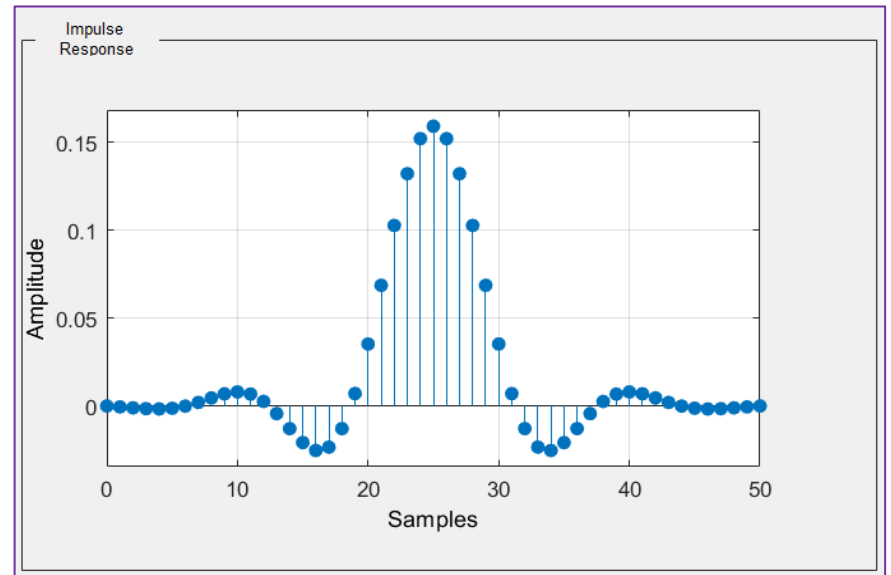
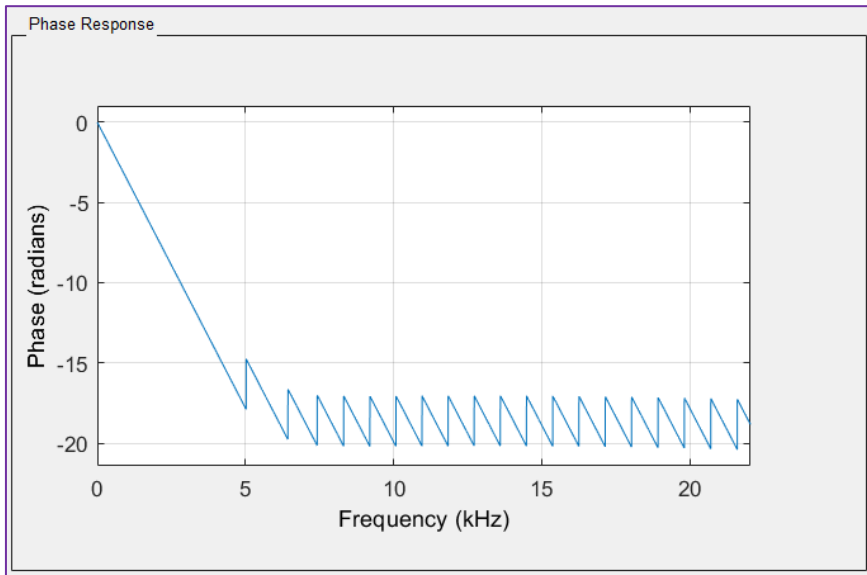


I designed a 50th order low-pass FIR filter with Hamming window for better side lobes reduction and 3.5KHz Cutoff Frequency which is suitable for filtering our 1KHz from the applied +40KHz noise.



MATLAB Modeling

From the toolbar, We can represent our filter from different aspects:



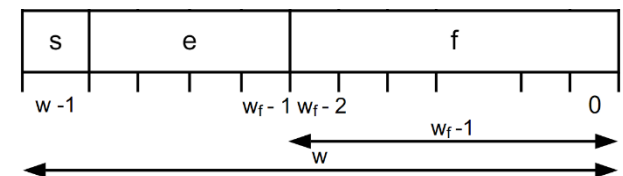
Now, the most critical step is to represent the Filter Coefficients shown above in our design.



MATLAB Modeling

Getting our 51 coefficients From *File* > *Export* > Coefficient File (ASCII)

```
1 % Generated by MATLAB(R) 9.8 and Signal Processing Toolbox 8.4.
2 % Generated on: 08-Feb-2023 03:24:28
3
4 % Coefficient Format: Decimal
5
6 % Discrete-Time FIR Filter (real)
7 % -----
8 % Filter Structure : Direct-Form FIR
9 % Filter Length : 51
10 % Stable : Yes
11 % Linear Phase : Yes (Type 1)
12
13
14 Numerator:
15 -0.000101679756821436236458207347244098173
16 -0.000626407761087044998374318005573968549
17 -0.001166205293878933794277097391045572294
18 -0.001628539759381721378425633695030683157
19 -0.001801676391558073610232870720437858836
20 -0.001396570137053724895703776809341434273
21 -0.000171358130166342014595246001640305167
```



To represent floating number in binary we reserve the MSB as a sign bit, some bits for integer value (exponent), and other bits for the fraction (mantissa). So, let the tap (coefficient register) size is 16-bit including 1 bit for the sign, 0 bit for integer (all coefficients are less than 1) and the rest 15 bits represents the floating part.

```
To represent 15-bit fraction, multiply Coefficient * 2^15
if +ve --> round the result off
if -ve --> round the result off --> get 2's complement (1 + ~(rounded off result))
-0.0001016 --> 1111 1111 1111 1101
-0.0006264 --> 1111 1111 1110 1011
-0.0011662 --> 1111 1111 1101 1010
-0.0016285 --> 1111 1111 1100 1011
-0.0018016 --> 1111 1111 1100 0101
-0.0013965 --> 1111 1111 1101 0010
-0.0001713 --> 1111 1111 1111 1010
0.0018983 --> 0000 0000 0011 1110
```

and so on ...



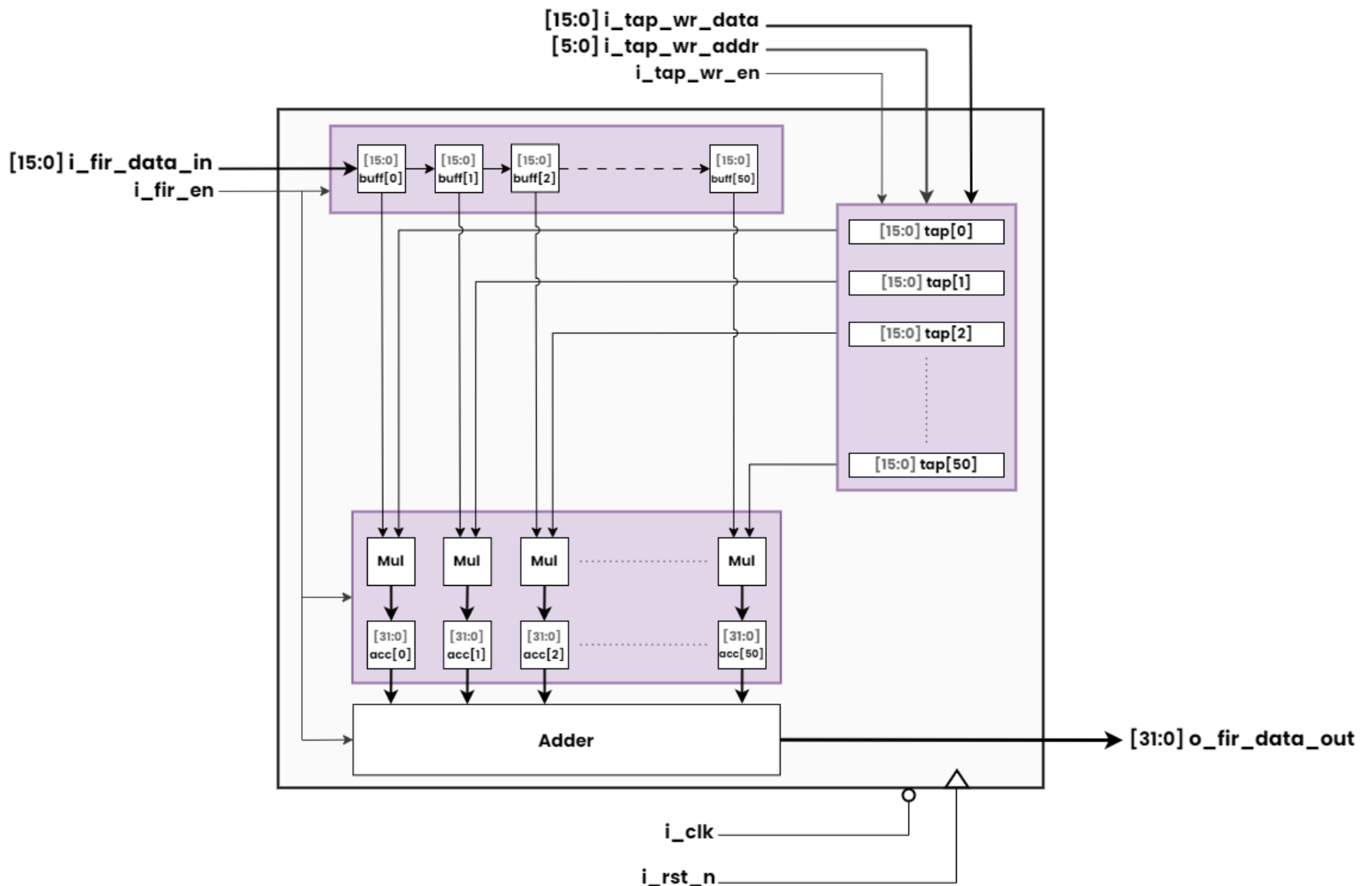
Architecture Design

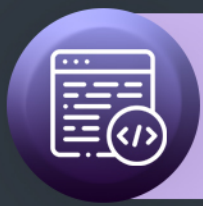
From the block diagram stated before, our main building blocks is:

Buffers act as a shift register that propagates each *16-bit* sample.

Taps hold the *16-bit* filter coefficients (I treated it as a configurable RAM)

Accumulators hold the 16-bit samples \times 16-bit taps *32-bit* results.





RTL Design

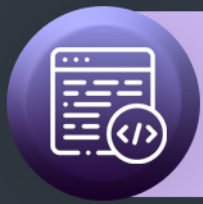
```
module fir_filter
#(
    parameter ORDER          = 50 ,
    parameter DATA_IN_WIDTH = 16 ,
    parameter DATA_OUT_WIDTH = 32 ,
    parameter TAP_DATA_WIDTH = 16 ,
    parameter TAP_ADDR_WIDTH = 6 )
(
    input wire signed [DATA_IN_WIDTH-1 : 0] i_fir_data_in ,
    input wire i_fir_en ,
    input wire i_tap_wr_en ,
    input wire [TAP_ADDR_WIDTH-1 : 0] i_tap_wr_addr ,
    input wire [TAP_DATA_WIDTH-1 : 0] i_tap_wr_data ,
    input wire i_clk ,
    input wire i_rst_n ,
    output reg signed [DATA_OUT_WIDTH-1 : 0] o_fir_data_out
);
```

```
//internal registers declaration as 2D-arrays for optimized hardware description
reg signed [TAP_DATA_WIDTH-1 : 0] tap[ORDER : 0] ;
reg signed [DATA_IN_WIDTH-1 : 0] buffer[ORDER : 0] ;
reg signed [DATA_OUT_WIDTH-1 : 0] accumulator[ORDER : 0] ;
```

```
integer i ;
```

```
//51-taps block logic
always @(posedge i_clk or negedge i_rst_n) begin
    if (!i_rst_n)
        begin
            //50th order fir coefficients
            tap[0]  <= 16'b 1111_1111_1111_1101 ;
            tap[1]  <= 16'b 1111_1111_1110_1011 ;
            tap[2]  <= 16'b 1111_1111_1101_1010 ;
            tap[3]  <= 16'b 1111_1111_1100_1011 ;
            tap[4]  <= 16'b 1111_1111_1100_0101 ;
            tap[5]  <= 16'b 1111_1111_1101_0010 ;
            tap[6]  <= 16'b 1111_1111_1111_1010 ;
            tap[7]  <= 16'b 0000_0000_0011_1110 ;
            tap[8]  <= 16'b 0000_0000_1001_0011 ;
            tap[9]  <= 16'b 0000_0000_1101_1111 ;
            tap[10] <= 16'b 0000_0001_0000_0010 ;
```

[Preview Full Code on GitHub](#)



RTL Design

```
        tap[49] <= 16'b 1111_1111_1110_1011 ;
        tap[50] <= 16'b 1111_1111_1111_1101 ;
    end
else if(i_tap_wr_en && !i_fir_en) //preventing coeff change during the operation
    begin
        tap[i_tap_wr_addr] <= i_tap_wr_data ;
    end
end

//51-buffers block logic
always @(posedge i_clk or negedge i_rst_n) begin
    if (!i_rst_n)
        begin
            for (i=0 ; i<=ORDER ; i=i+1)
                begin
                    buffer[i] <= 16'b0 ;
                end
            end
        else if (i_fir_en)
            begin
                buffer[0] <= i_fir_data_in ;
                for (i=0 ; i<ORDER ; i=i+1)
                    begin
                        buffer[i+1] <= buffer[i] ;
                    end
                end
            end
        end

//51-accumulator block logic
always @(posedge i_clk or negedge i_rst_n) begin
    if (!i_rst_n)
        begin
            for (i=0 ; i<=ORDER ; i=i+1)
                begin
                    accumulator[i] <= 32'b0 ;
                end
            end
        else if (i_fir_en)
            begin
                for (i=0 ; i<=ORDER ; i=i+1)
                    begin
                        accumulator[i] <= buffer[i] * tap[i] ;
                    end
                end
            end
        end
end
```

[Preview Full Code on GitHub](#)



RTL Verification

```
module fir_filter_tb();
reg signed [15 : 0] fir_data_in_tb ;
reg                fir_en_tb      ;
reg                tap_wr_en_tb   ;
reg                [5 : 0] tap_wr_addr_tb ;
reg                [15 : 0] tap_wr_data_tb ;
reg                clk_tb         ;
reg                rst_n_tb       ;
wire signed [31 : 0] fir_data_out_tb ;

parameter CLK_PERIOD = 22675.73696 ; //periodic time of 44.1KHz CLK
parameter SIN1_SAMP  = 37500        ; //sampling time for 550Hz + 3KHz Sin Wave
parameter SIN2_SAMP  = 25000        ; //sampling time for 820Hz + 4.3KHz Sin Wave
parameter SIN3_SAMP  = 10000        ; //sampling time for 2.1KHz + 10.8KHz Sin Wave
parameter SIN4_SAMP  = 1562         ; //sampling time for 13KHz + 70KHz Sin Wave

`include "fir_filter_tasks.v"

reg [15 : 0] sin_sampling_count ;
integer i ;

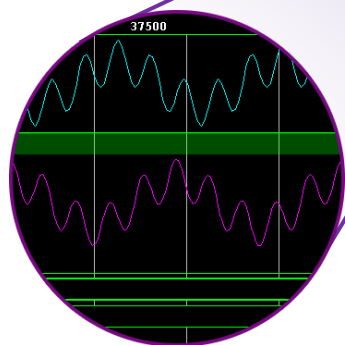
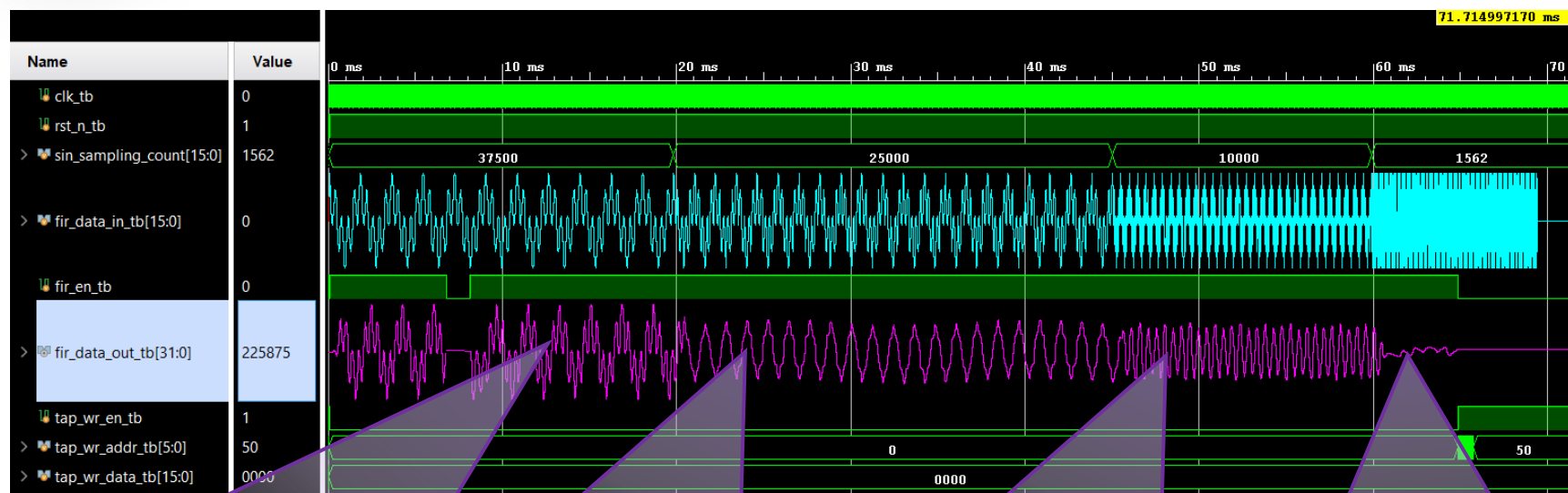
//-----DUT Instantiation
fir_filter DUT (
.i_fir_data_in (fir_data_in_tb) ,
.i_fir_en      (fir_en_tb)      ,
.i_tap_wr_en   (tap_wr_en_tb)   ,
.i_tap_wr_addr (tap_wr_addr_tb) ,
.i_tap_wr_data (tap_wr_data_tb) ,
.i_clk         (clk_tb)         ,
.i_rst_n       (rst_n_tb)       ,
.o_fir_data_out (fir_data_out_tb)
);

//-----Clock Generation
initial
begin
    clk_tb = 1'b1 ;
    forever
    #(CLK_PERIOD*0.5) clk_tb = ~clk_tb ;
end
```

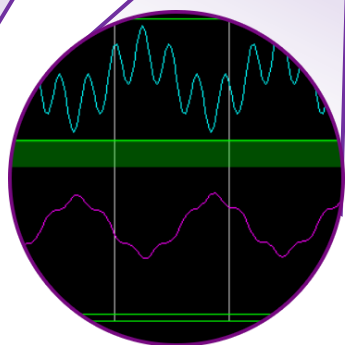
[Preview Full Code on GitHub](#)



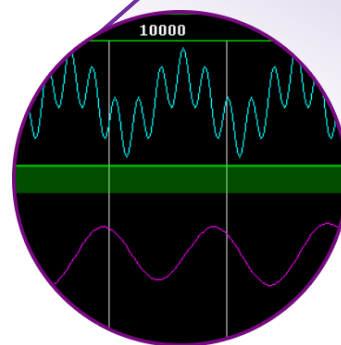
RTL Verification



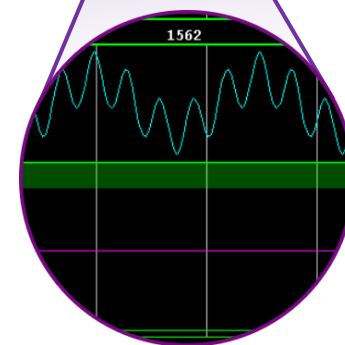
550Hz: Passed



820Hz: Passed



2.1KHz: Passed



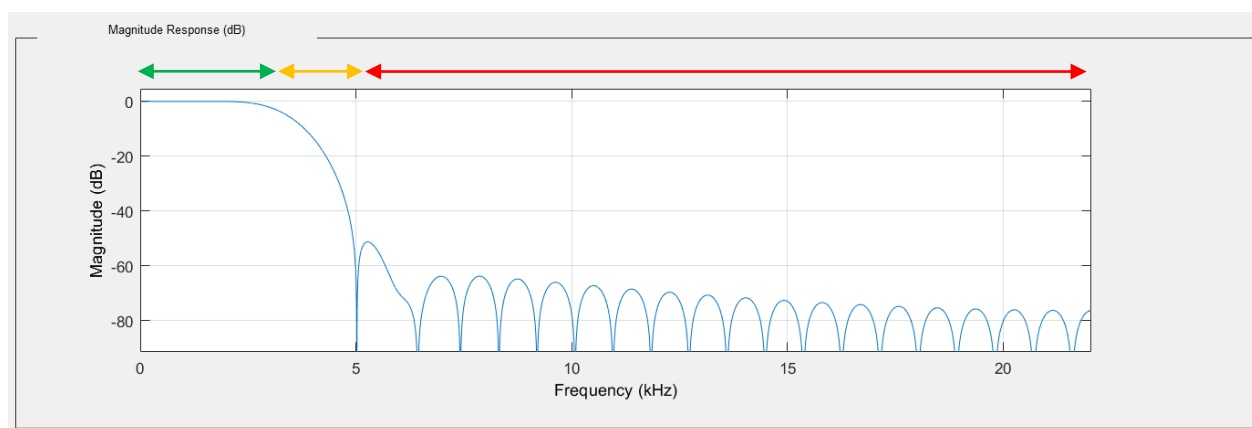
13KHz: Filtered

3KHz: Passed

4.3KHz: attenuated

10.8KHz: Filtered

70KHz: Filtered





Simulink Testing

In the previous testbench, we covered several scenarios of filtering a multi-frequency signal in addition to testing control signals such as disabling *fir_en* and resuming it again and finally writing new filter coefficients (clear all taps in our case).

But for more digging into practical applications, we are going to import our verified RTL to Simulink and apply the more complex audio .wav files we created before and check the filter results.

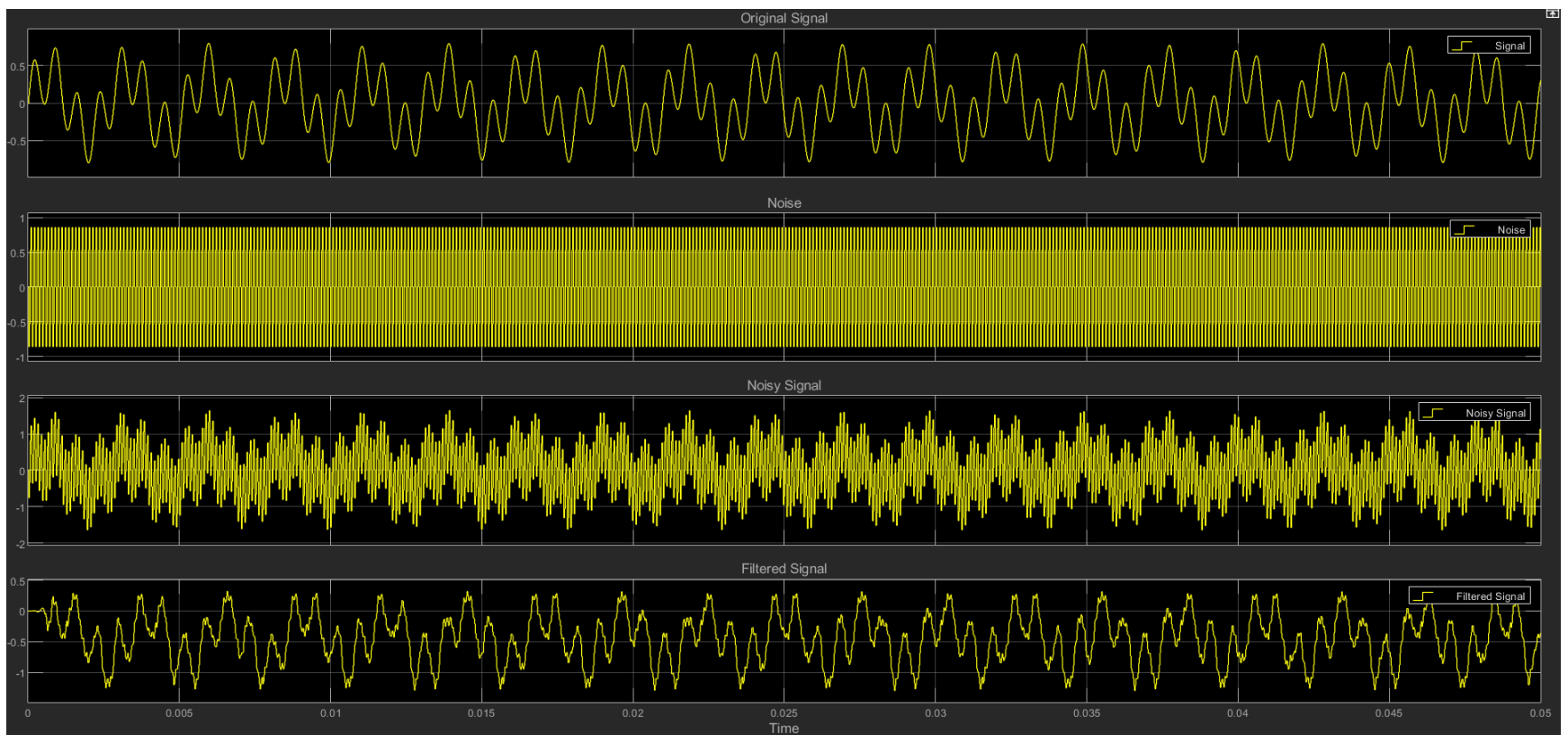
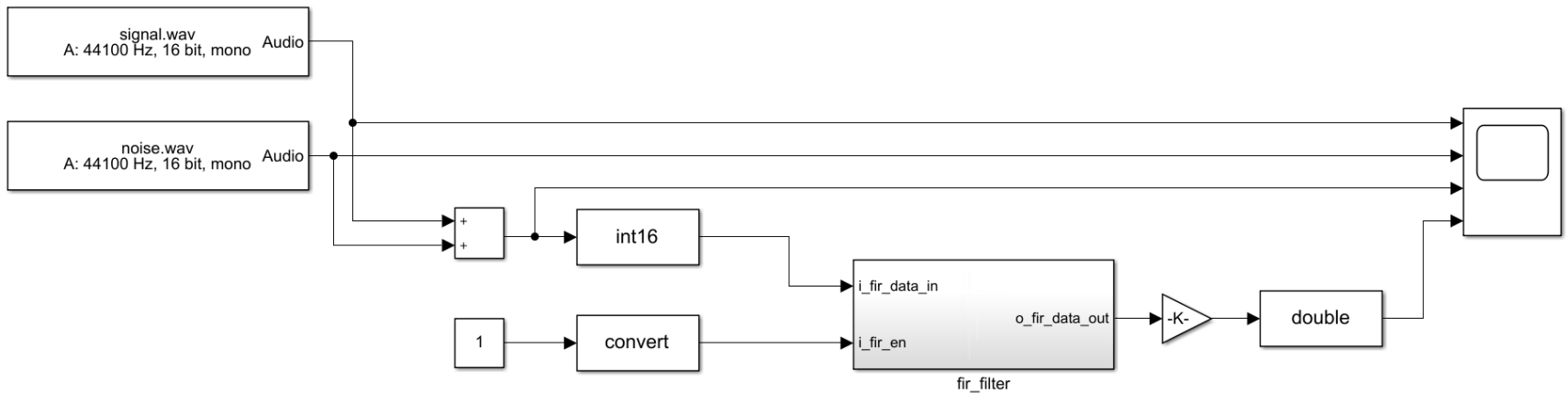
To import HDL code and generate Simulink model:

```
>> importhdl('fir_filter.v')
```

Then, create your top Simulink model and get from Simulink Library Browser>*DSP System Toolbox>Sources>From Multimedia File* and import the .wav files and get other required blocks to construct our experiment system as shown in the following page.



Simulink Testing



We can notice that the original signal is recovered successfully with an acceptable effect of noise in some values of the signal's amplitude.



Thank You !

All code for this project can be found on my [GitHub](#)



Yaseen Salah

Digital IC Design Undergraduate