EE2016 MidSem Part-C
Report

Sharanesh R
EE19B116
Ph: 8973359547

I. Theory on FIR, Convolution, DTFT:-

A finite impulse response (FIR) filter is a filter whose impulse response is of finite duration, because it settles down in finite amount of time. There is no feedback here as in the case of an infinite Impulse response filter.

For a Causal Discrete-time FIR filter of order N, the output of the filter is given by,

$$y[n] = b_0 x[n] + b_1 x[n-1] + \cdots + b_N x[N-1]$$
$$= \sum_{K=0}^{N} b_K x[n-K],$$

Where,

- → $x[n]$ is the input signal
- → $y[n]$ is the output signal
- → N is the order of the filter, (N = number of taps + 1)
- → b_i is value of the impulse response at the i^{th} instant of time for $i \in [0, N]$.

This computation known as the Discrete-time convolution *.

-> The values $x[n-i]$ here are what being referred to as taps, based on the structures of a tapped delay line to find the delayed inputs used in the multiplication.

* Convolution in general on two functions produces a third function that expresses how the shape of one is modified by the other.

Discrete time convolution is defined as,

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]\, g[n-m]$$

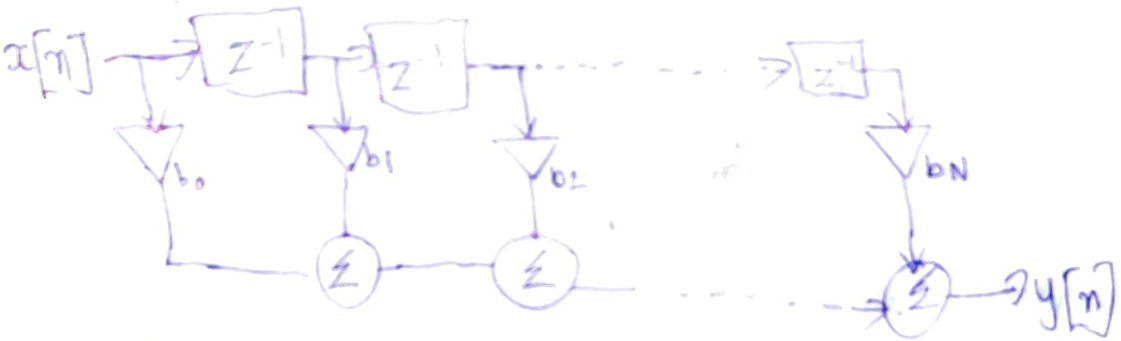
→ Thus in the context of FIR filter, the impulse response is defined over a non-zero domain as

$$h[n] = \sum_{i=0}^{N} b_i \delta[n-i] = \begin{cases} b_n & 0 \le n \le N \\ 0 & \text{elsewhere} \end{cases}$$

where

$\delta[n-i]$ is the impulse function
b_i are the coefficients

The direct form Discrete-time FIR filter is given below,



where N = order of the filter

it has N+1 taps, each with a unit of delay,

the z^{-1} is in the Z-transform notation

→ Frequency response of FIR filter is given below cos,

$$\underbrace{\mathcal{F}\{x * h\}}_{Y(\omega)} = \underbrace{\mathcal{F}\{x\}}_{X(\omega)} \cdot \underbrace{\mathcal{F}\{h\}}_{H(\omega)} \quad , \quad y[n] = x[n] * h[n] = \mathcal{F}^{-1}\{X(\omega) \cdot H(\omega)\}$$

where the operations $\mathcal{F}, \mathcal{F}^{-1}$ denote the Discrete-time Fourier transform (DTFT)

→ The DTFT is defined here as

$$X_{2\pi}(\omega) = \sum_{n=-\infty}^{\infty} x[n] e^{-j\omega n}$$

if the function is periodic with periodicity "2π"

(or)

$$X_{1/T}(f) = X_{2\pi}(2\pi f T) \approx \sum_{k=-\infty}^{\infty} X\left(f - \frac{k}{T}\right)$$

(or)

$$X_{1/T}(f) = \mathcal{F}\left\{\sum_{n=-\infty}^{\infty} x[n]\,\delta(t - nT)\right\}$$ where T is period.

Thus, these all are very much related to each other and are used in this implementation.

Section-II : Fixed point Arithmetic used here

→ Here basically we have all the inputs and the coefficients as signed fractional numbers.

→ We take a fixed resolution of input and coefficients as 8-bits.

→ Among the 8 bits, we use the (1.7) notation where the floating point is fixed between the MSB and the next bit.

→ we can represent any fractional number in the range [-1 to 1) using this format.

→ When we are using this notation and then perform the multiplication, we use the FMUL operation that produces a 16-bit result which will be in the format (1.15).

Converting numbers to this format :-

Given the decimal number to convert it to the fixed point format as shown above we use the following

i) if the fraction is positive then multiply that with 2^7 (128) and get the rounded integer which will be the representation in decimal form

ii) if the fraction is negative then add 2 to that decimal number and multiply that with 2^7(128) and get the rounded integer which will be the representation in decimal form.

→ Here we use 8-bit representation (signed) so they can range from -128 to 127 (-2^{8-1} to $2^{8-1}-1$), and what is being done for both inputs and the coefficients.

→ Multiplying the numbers and accumulating them,

* the multiplication of any two numbers represented in the above format will produce a 16 bit result in (1.15) format.
* we need accumulate such results to get the final output of the FIR filter.

Size of accumulator :-

$$N \geq 2 \cdot K + \log_2^M + 1$$

(K → bit resolution excl. sign bit)
(M → number of additions)

for 5-tap case :- $M=5, K=7$

$$N \geq 2(7) + \log_2^5 + 1 \Rightarrow N \geq 17 //$$

for 32-tap case :- $M=32, K=7$

$$N \geq 2(7) + \log_2^{32} + 1 \Rightarrow N \geq 19 //$$

→ So, we need a maximum of 24 bits to store the accumulated result.

→ But there are certain cases that must be considered while processing the generated 24-bits results because whenever we add two signed numbers there is a possibility of overflow being generated which should be taken care of.

→ There are two cases in which the overflow can occur, if we add two positive numbers and the result is not fitting in the resolution and the other is that two negative numbers sum can't be fitted into the resolution.

→ For the case that happens with +ve numbers, we just assume that as carry and include that result also while descaling the output.

→ But in the case of overflow occuring with the negative numbers addition, we need to take care of the sign bit being flipped and this can be tackled by counting all such additions ((i.e) additions of negative numbers) and multiply it with two and Substract it from the integral part of accumulated result. This is done because in that case we are adding extra two's since the whole computation is done in two's complement format.

→ By the above we make sure that the final result is correct without any extra result bearing embedded in it.

→ Implementation of above method:

* Keep a register and inc. it by 2 everytime we add -ve numbers. (1)
* Do the accumulation as normal 24-bits addition.
* Substract the value (1) from the integral part of accumulated result.

* The integral part of the accumulated result is the third byte along with MSB of the middle byte.

→ In the code this has been done by copying the two bytes (MSB and middle byte) to auxiliary registers and using ROR and ROL and subtraction.

Downscaling the Result :-

→ Our result that has been generated is of 24 bits.
→ we initially scaled the ~~results~~ inputs and coefficients by 2^7, 2^7 respectively and so we need to downscale the accumulated result by 2^{14}.

Interpretation of the result :-

→ we get a 24-bit result and if the high byte (MSB) is '1' then the result is -ve and if it is zero then it is positive

→ only if the result is negative, the high byte will be affected and we need to subtract 2^{14} from it and then divide it by 2^{15} to get the final decimal answer. (because 1.15 is the final format).

→ But in case of the result being positive there is no issue and we just divide the result by 2^{15} to get the final decimal answer (same as above the 1.15 format)

Section - III : DTFT Implementation

→ After calculating all the values of the output and downscaling them and getting the final result, we do the DTFT to get frequency response / magnitude of output in the frequency domain over a range of frequencies.

→ For implementing this, we basically use the built-in function in python `Scipy.fft`, that implements the algorithm of Fast fourier transform to get the frequency response.

→ This basically takes the time domain output as its input alongwith the number of samples and ~~the frequency~~ gives us the DFT of the output. We then plot them over the range of frequency and interpret our results.

Section - IV : Actual implementation details

1. Input into AVR and storing the output

→ The input values need to given are for DC signal, sinusoidal signal of frequency and a white noise with all matched power.

→ for the normalized inputs, following the instructions given taking a variance of 0.33 we generate the white noise signal, so almost 99% of the value will be in -1 to 1 range, if any are out this range we neglect them / make it as zero). for the other cases DC and sine, we keep the value of DC signal as 127 (largest possible +ve number) and for sine it is already in the range -1 to 1.

→ All the preprocessing of inputs to bring it to the fixed point arithmetic notation is being done in python.

→ The 1000 input values can't be stored in the SRAM because the memory won't sufficient them to store the coefficients and Output, so I stored the inputs in flash memory itself for sine and white noise cases and for DC I hard code that value and use LDI to that because the value is same irrespective of the value of 'n'.

→ So the inputs are stored in the program (flash memory) that are loaded using num: .db after ~~pasting~~ manually copy pasting them.

→ for storing outputs:-

* output is 24 bit long and so if we need to calculate 1000 values we need to have 3000 bytes but Atmega8 has only 1000 bytes so I just calculate the output for 250 values of 'n' starting from n=100 to n=349.

* The calculated output are stored in SRAM, starting from the location(0x0090), from this onwards every 3 bytes corresponds to output of one value of 'n'.

→ Since we are using LPM to load the input value, the Z-pointer is used for that, Y-pointer is used for storing the results and X-pointer for loading the coefficients.

→ Note:- There is no use of circular buffer here, for accessing the correct inputs I just used two registers that ~~does~~ increments the pointer each time after the calculation of the result to that of the previous start location, this way the inputs are loaded without circular buffer

2. Storing the Coefficients

→ Before the start of the algorithm we just load the coefficients from flash memory (defined using num.db) and then store them in SRAM starting from location (0x0060)

→ for accessing the coefficients the x-pointer is used which decrements each time for each multiply and accumulate operation, once the output for one 'n' value is calculated it resets again to the 0x0060 location.

3. Multiplication details (Multiply and accumulate)

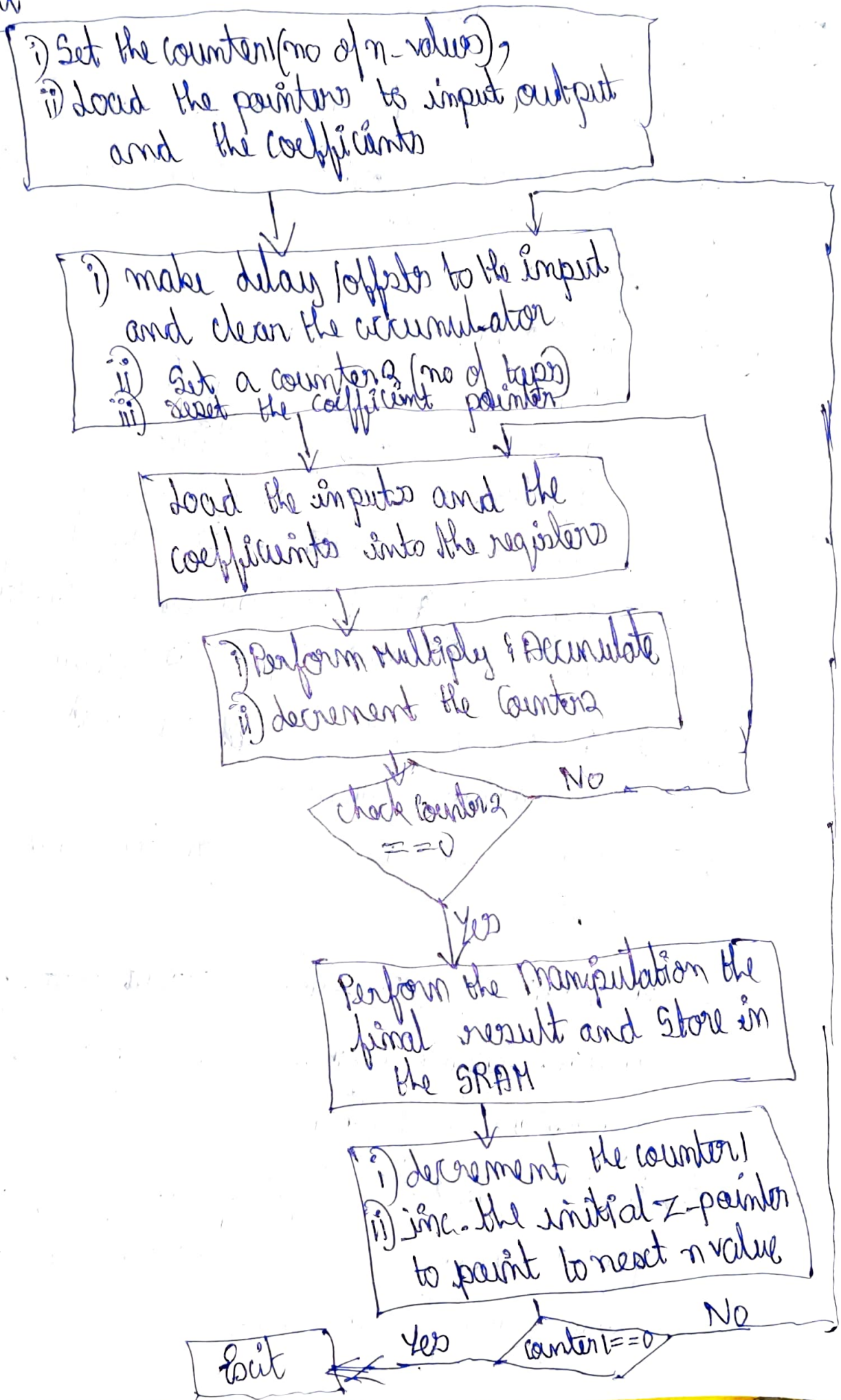
→ we need to an accumulator of size 24 bits to store final result.

→ here r10 : r9 : r8 acts as the accumulator, and the register r11 stores the 2* (no. of negative number add^n's made)

→ After the couple MAC process we subtract the r11 register the integer part of result (i.e) r10 along with the MSB of r9, this done by moving them to separate registers, using shifts and finally storing them back in r10 and r9 itself.

→ Accumulate is done simply by adding 24 bit number and a 16 bit number, if any carry is generated it gets added to third byte.

4) flow chart


i) Set the counter1(no of n-values),
ii) Load the pointers to input, output and the coefficients
i) make delay /offsets to the input and clear the accumulator
ii) Set a counter2 (no of taps)
iii) Reset the coefficient pointer
Load the inputs and the coefficients into the registers
i) Perform multiply & Accumulate
ii) decrement the Counter2
Check Counter2 == 0
No
Yes
Perform the manipulation the final result and Store in the SRAM
i) decrement the counter1
ii) inc. the initial z-pointer to point to next n value
counter1 == 0
No
Yes
Exit

5) Some Specific things done :-

a) The coefficients are initially loaded with the help of Z-pointer from the flash memory and then stored in the SRAM. After this only the inputs are stored in flash memory

b) there are 4 num.dat's defined.

- num1 :- 5 tap coefficients
- num2 :- 32 tap coefficients
- num3 :- sine signal input
- num4 :- white noise input

c) So as per (b) we need to load proper input and coefficients while running the program

d) to use DC signal just give MOVLDI with 0x7f because it is a constant signal.

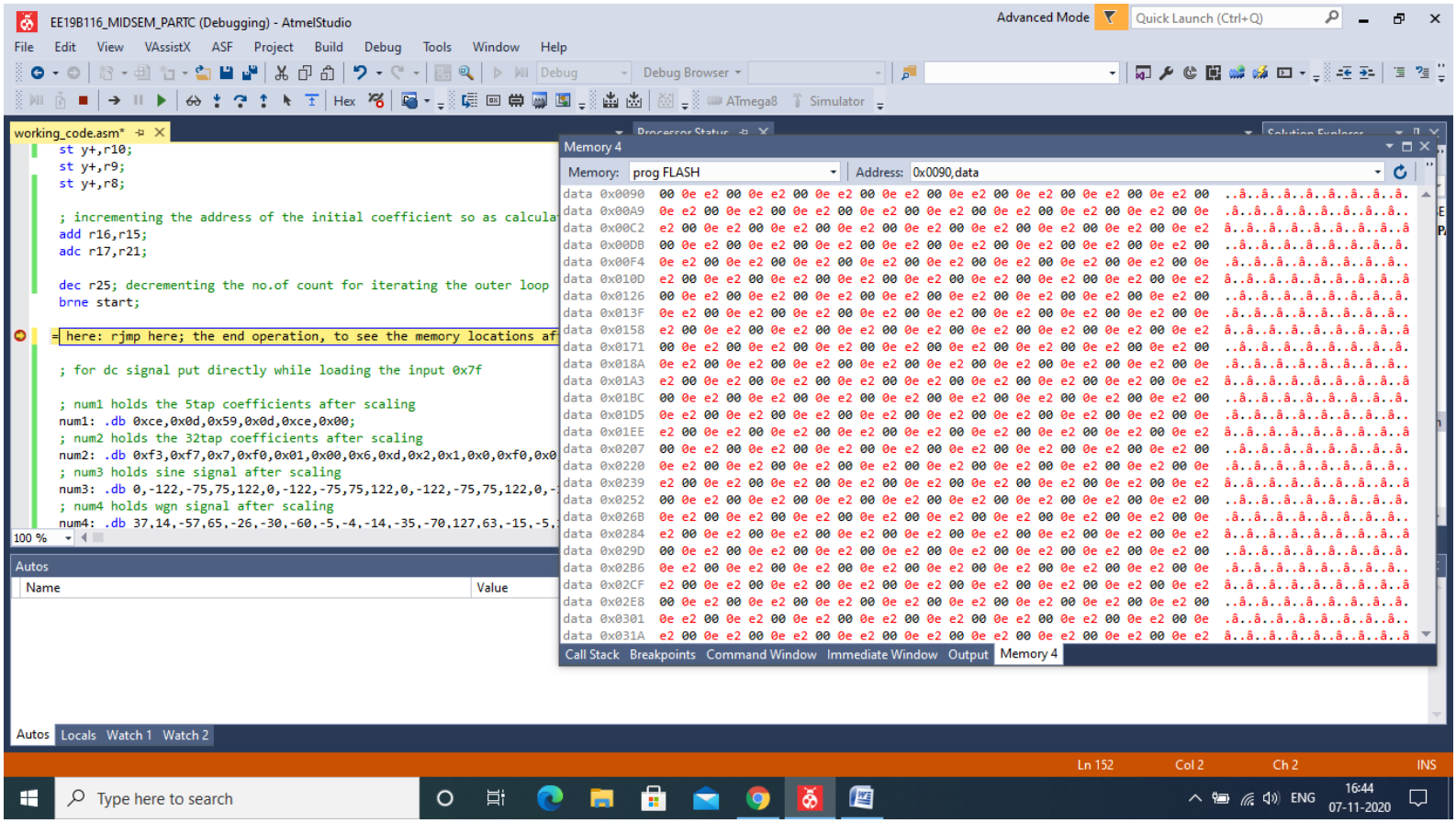
e) The output calculated are for the range n=100 to n=349.

f) the input (pre-processing), output (post-processing) is done using python and plots are also generated in python.

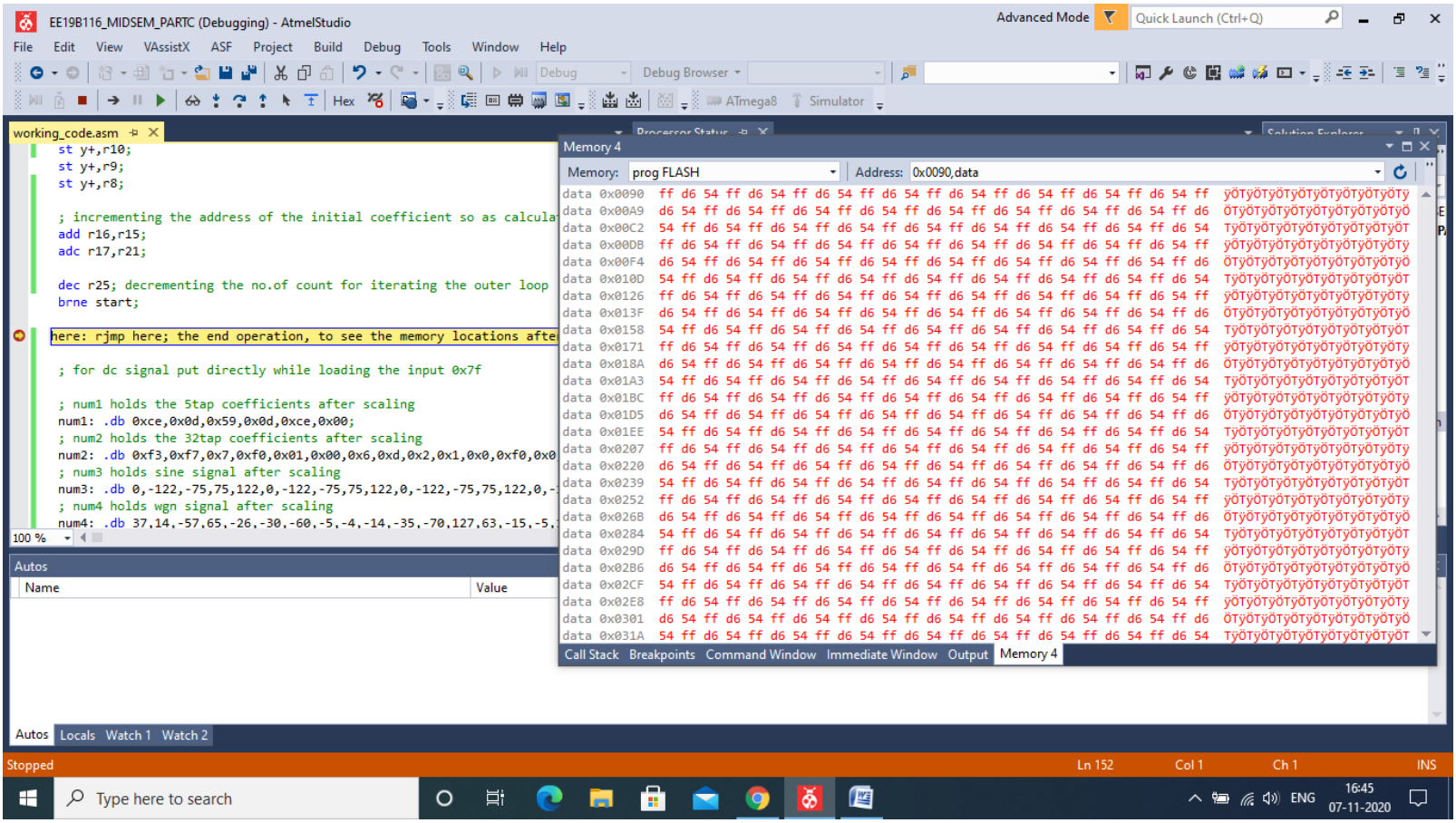
g) the FFT (DTFT to get freq. response) is done in python using scipy.fft module and plots using matplotlib.pyplot.

Register Outputs Pictures

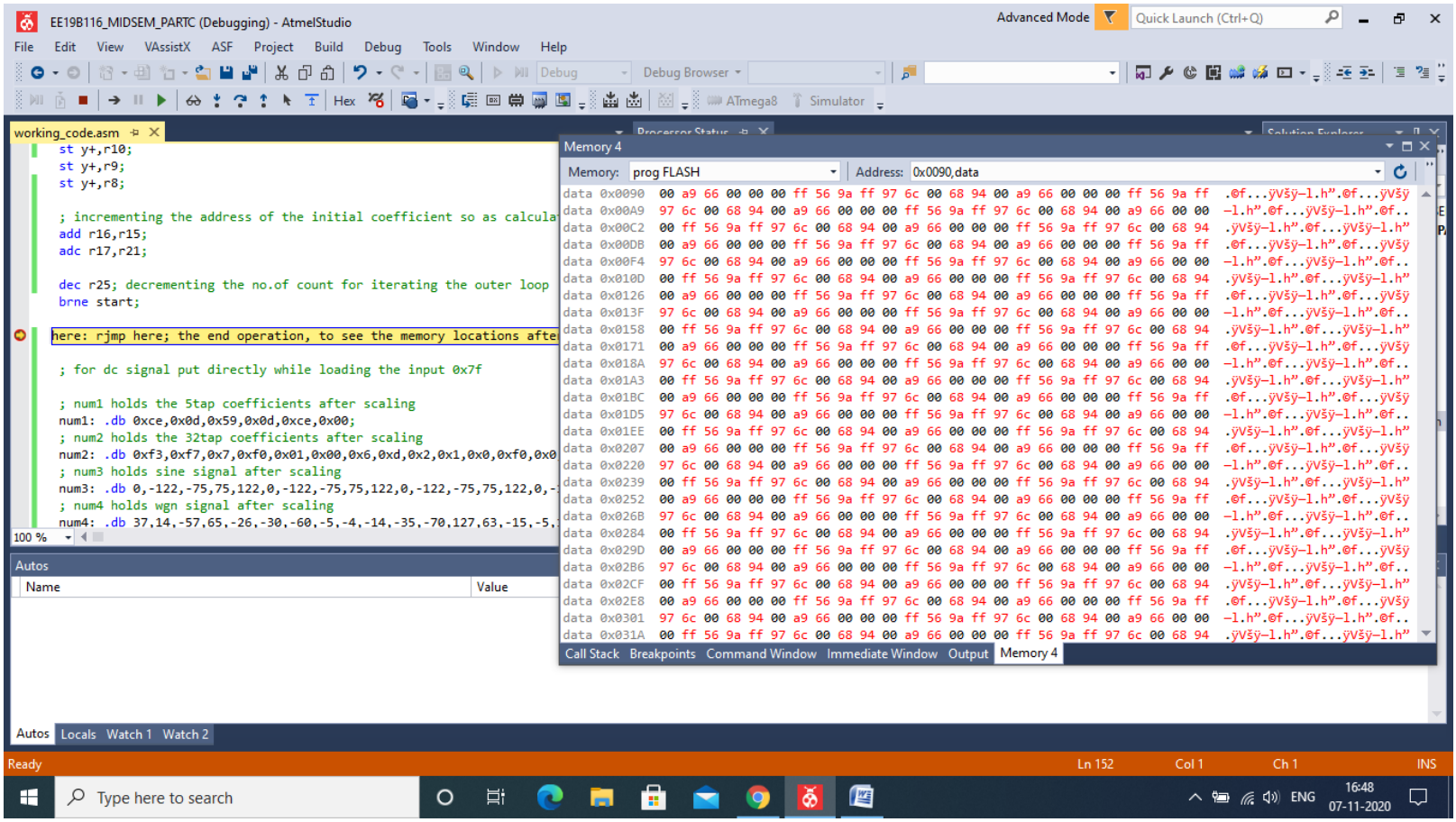
a)5-tap output for DC input being stored at 0x0090



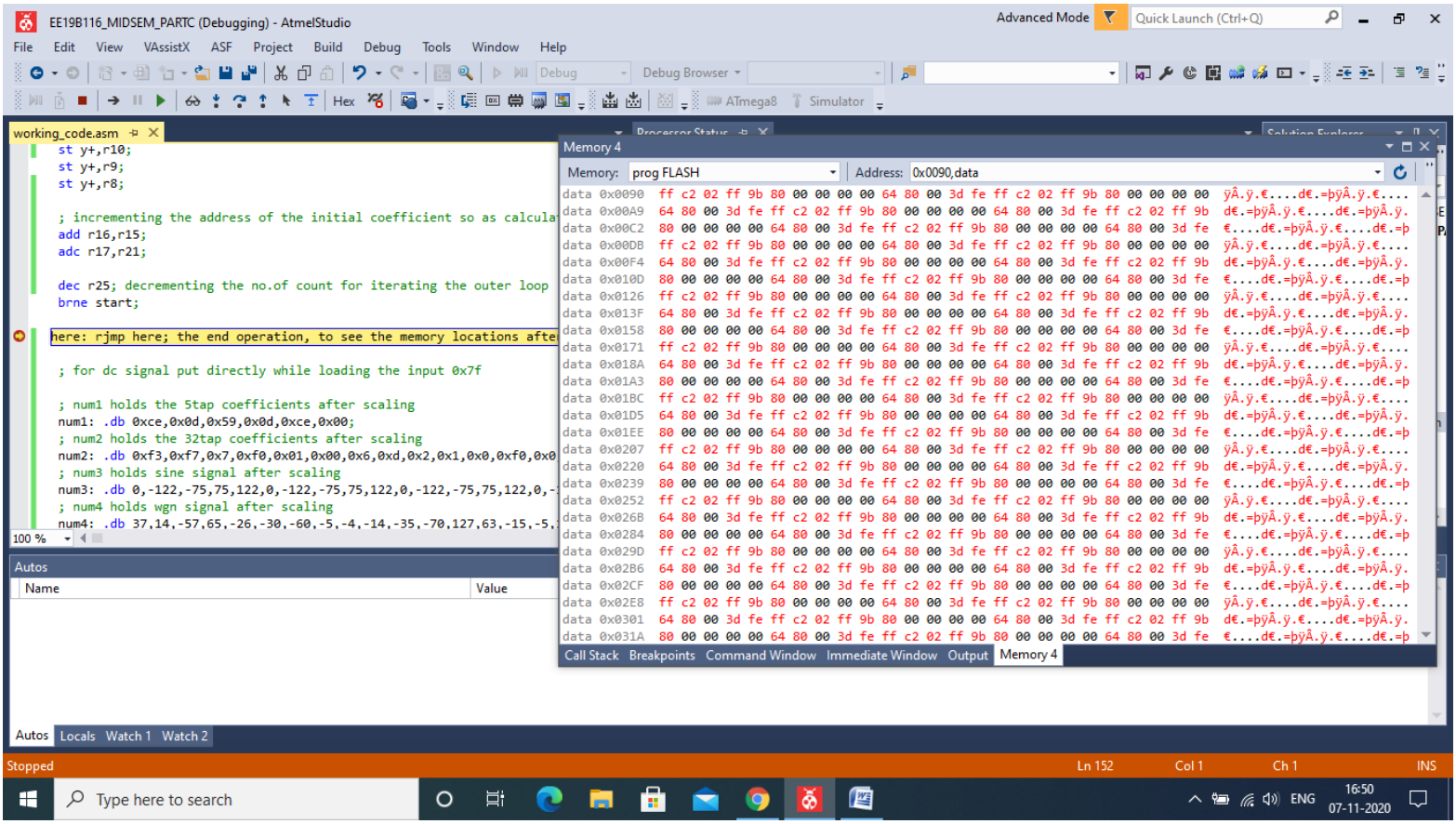
b) 32-tap output for the DC input stored at 0x0090



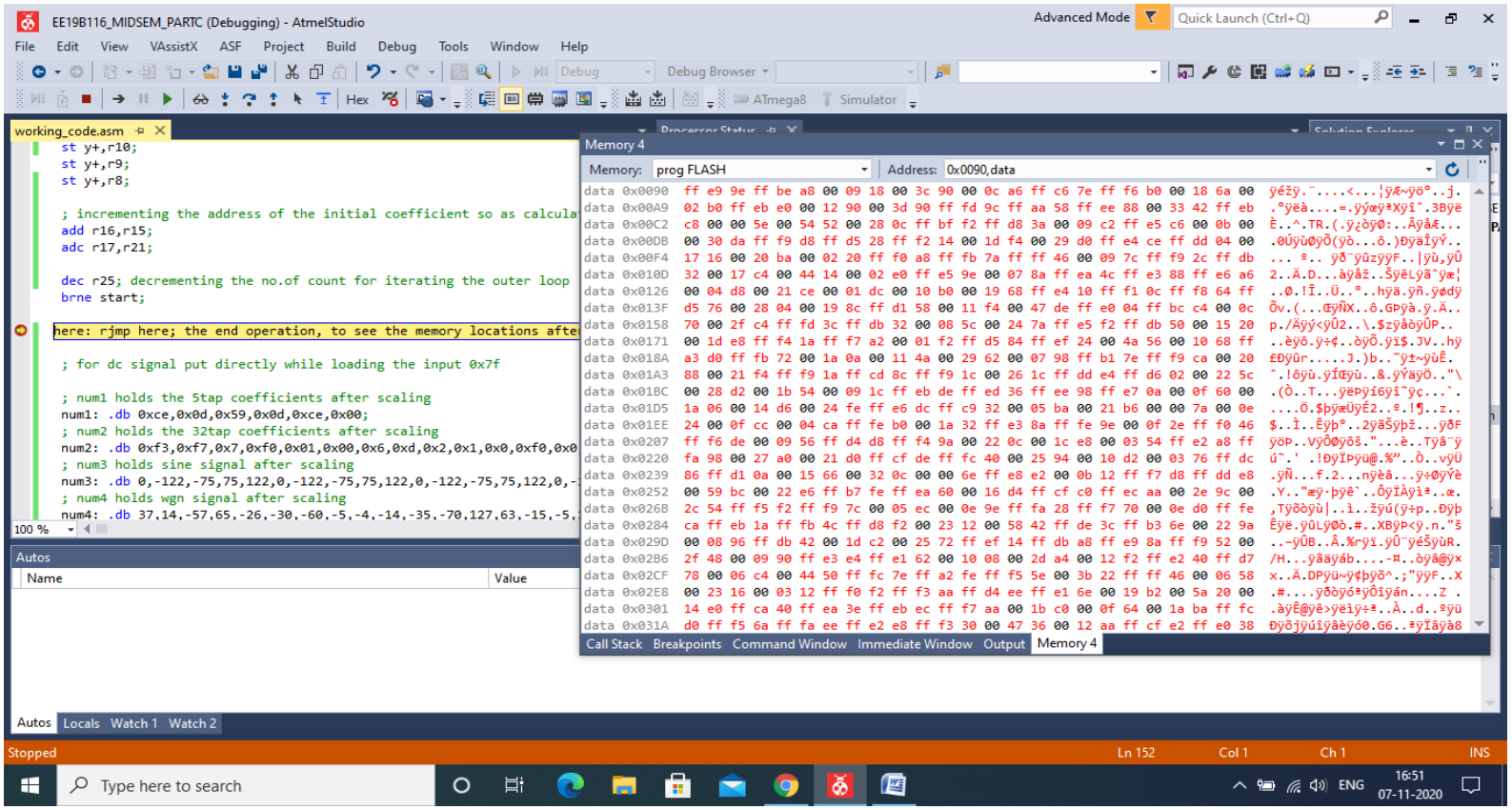
c) 5-tap output for sinusoidal input being stored at 0x0060



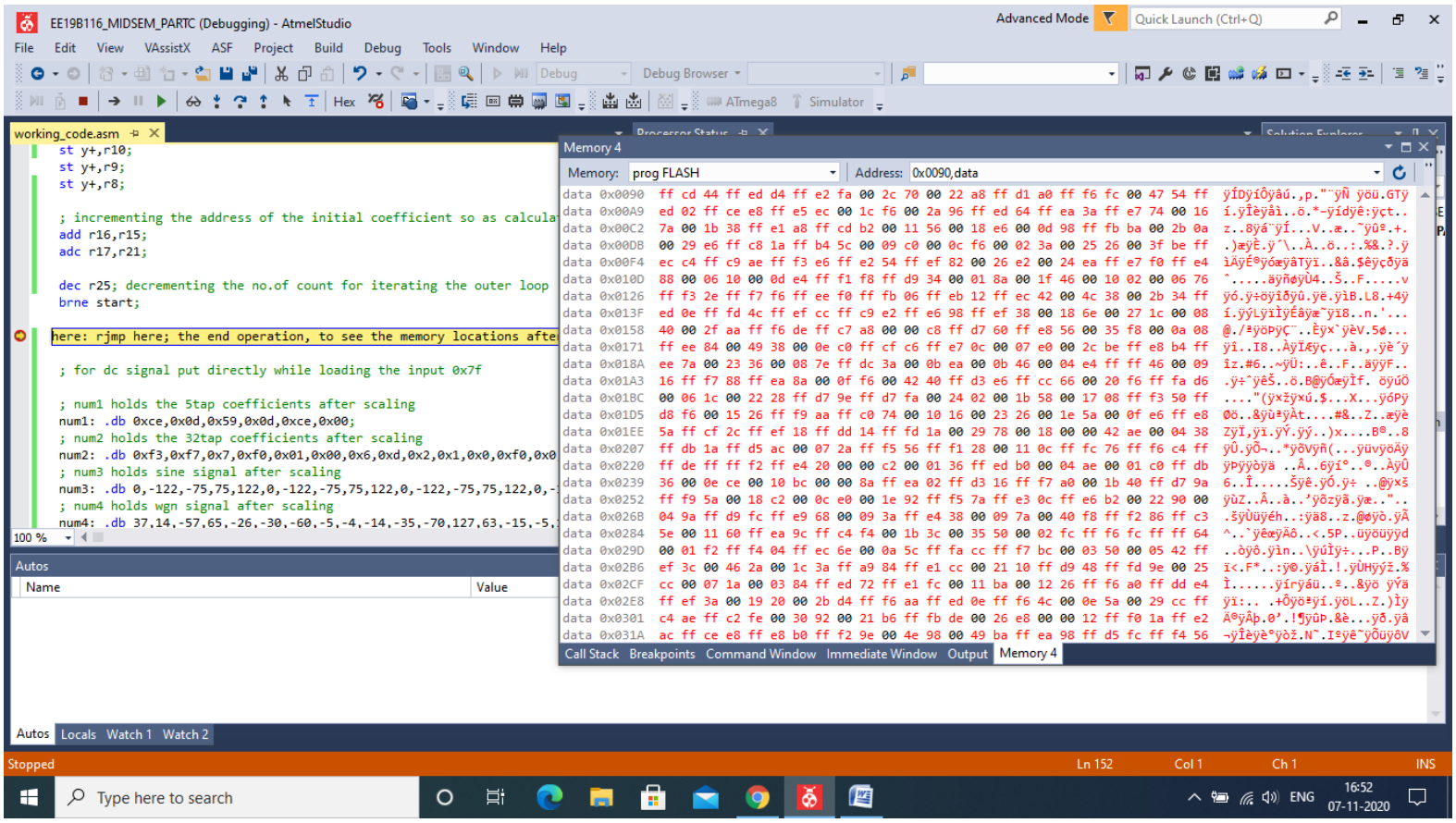
d) 32-tap output for sinusoidal input being stored at 0x0090



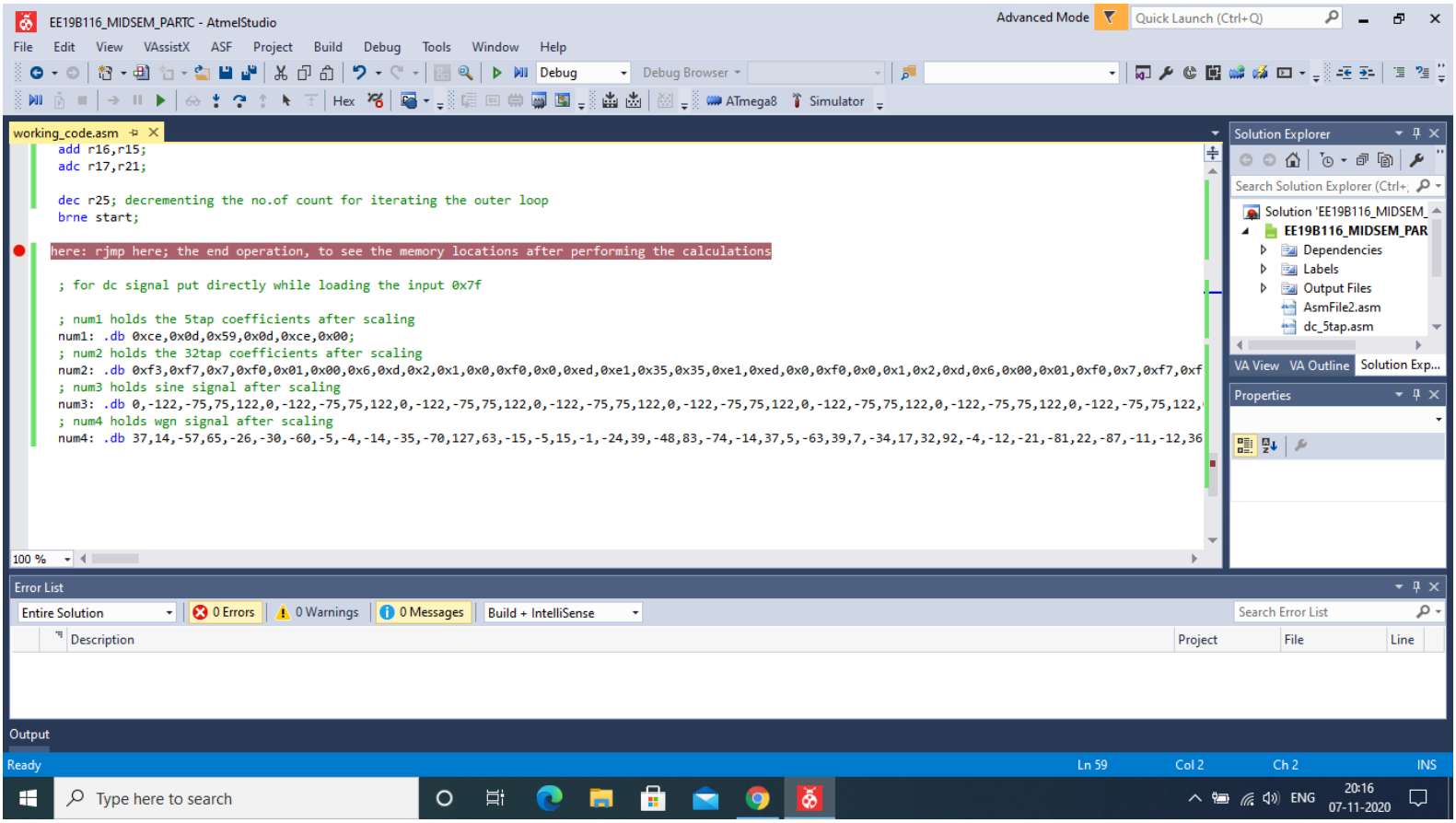
e) 5-tap output for a white noise input being stored at 0x0090



f) 32-tap output for a white noise input being stored at 0x0090

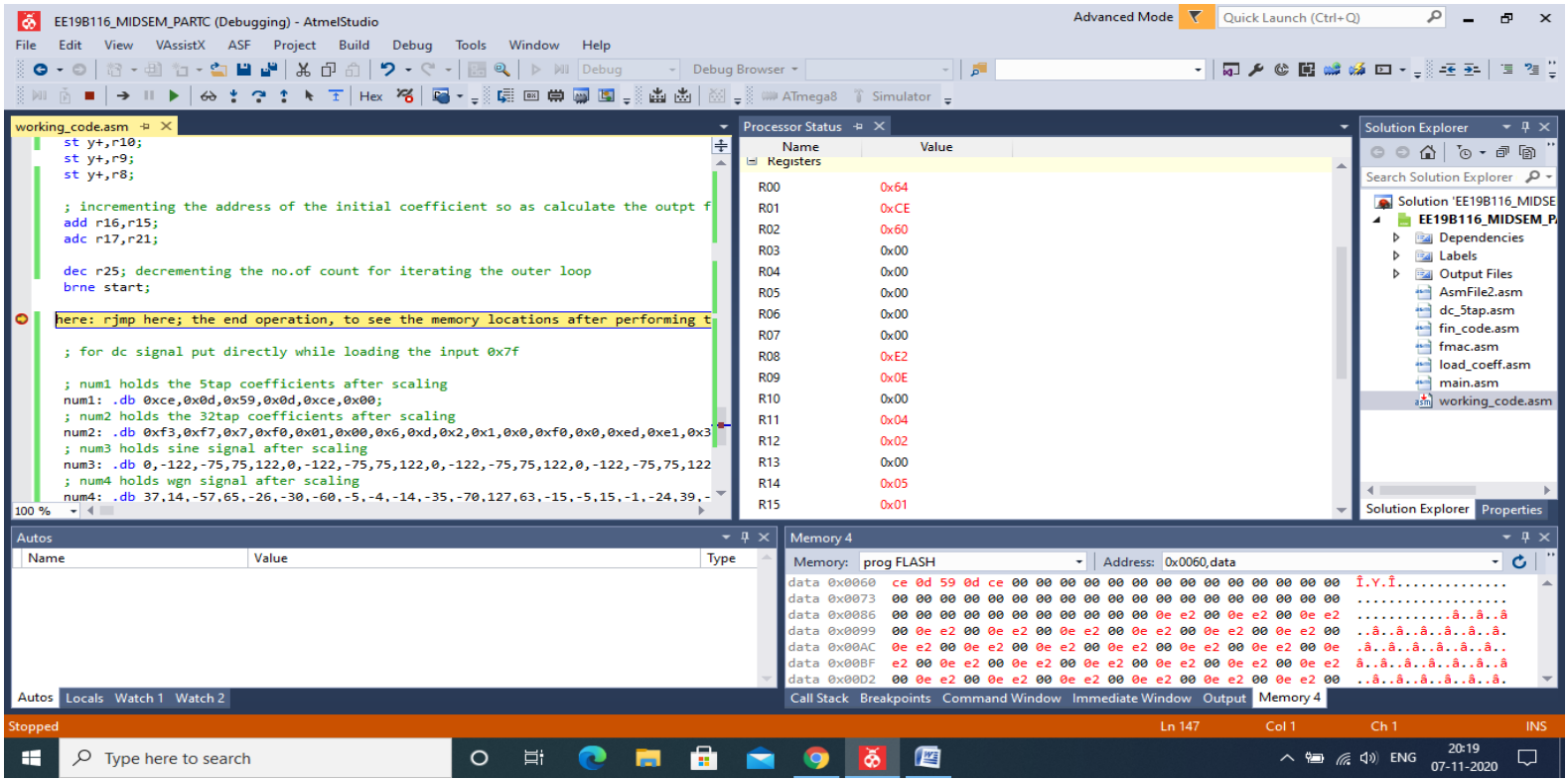


Giving the inputs using num: .db

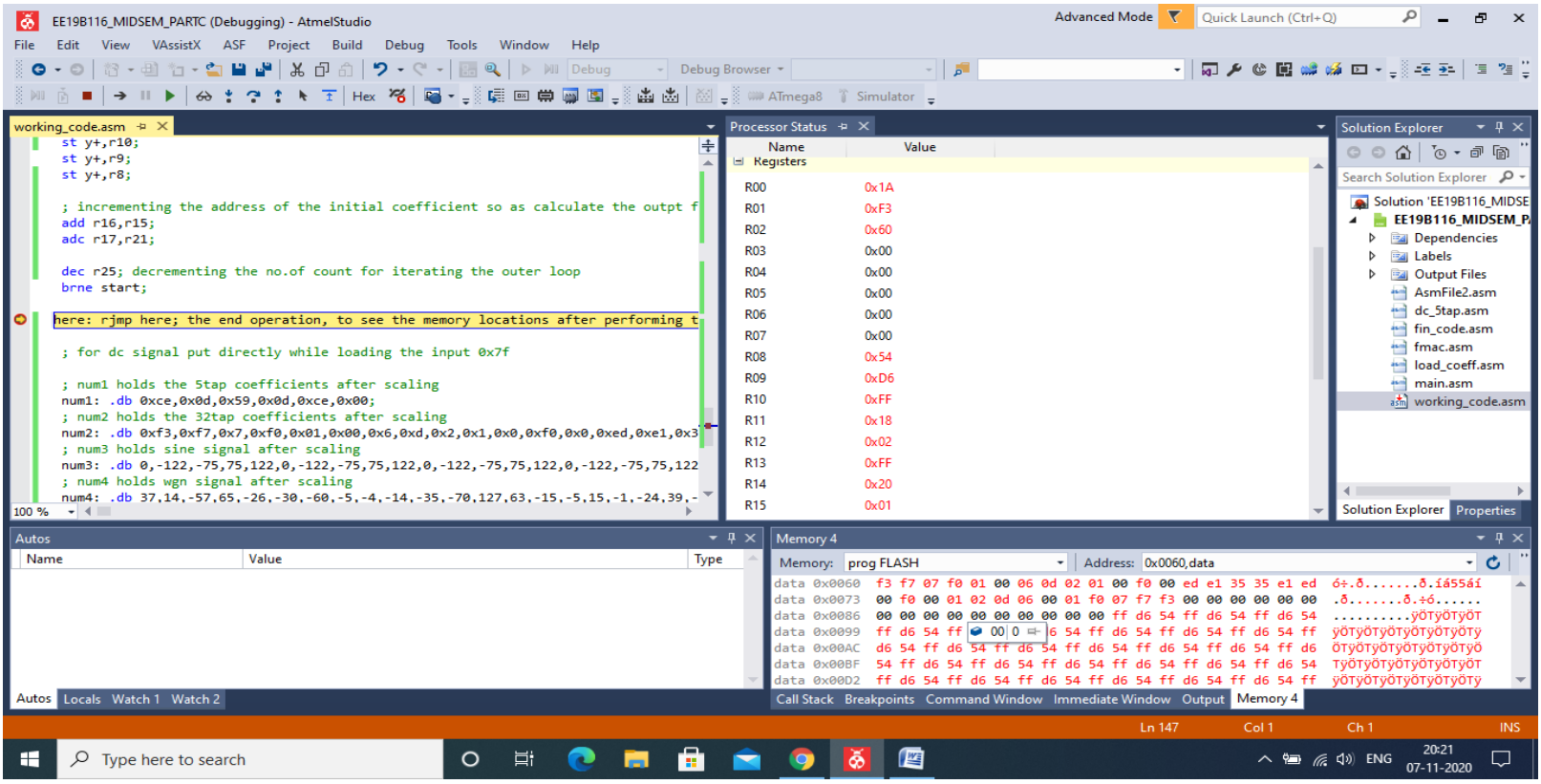


The coefficients are loaded using num: .db and store it in the SRAM at 0x0060 at the location:

a)for 5-tap case



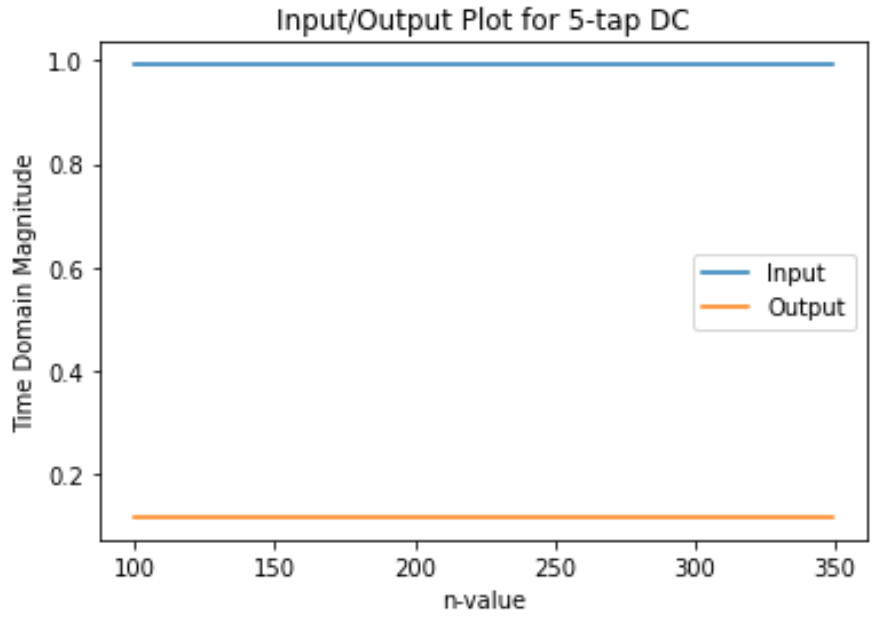
b)for 32-tap case



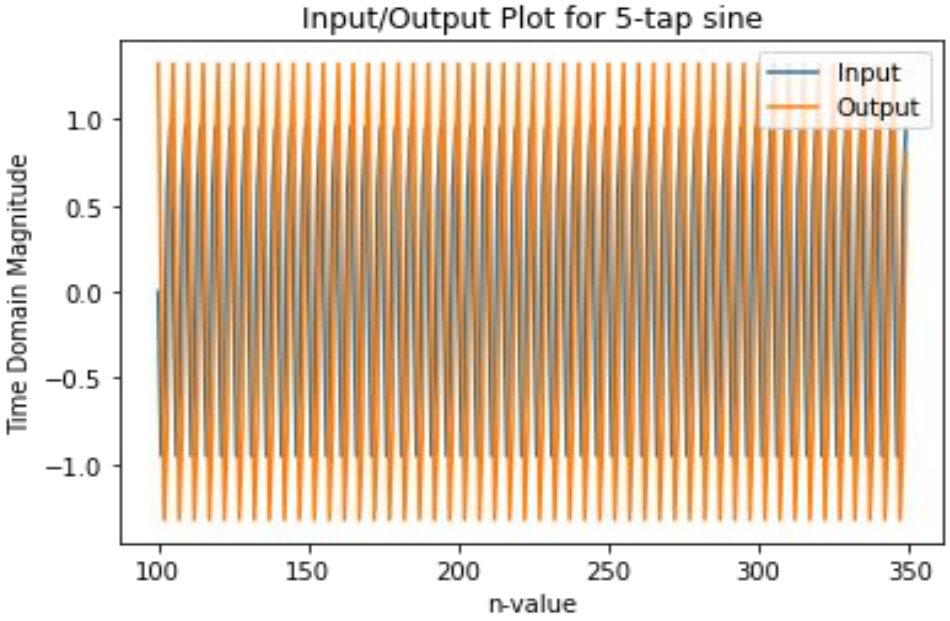
Results

Input-Output plots for the 5-tap case:

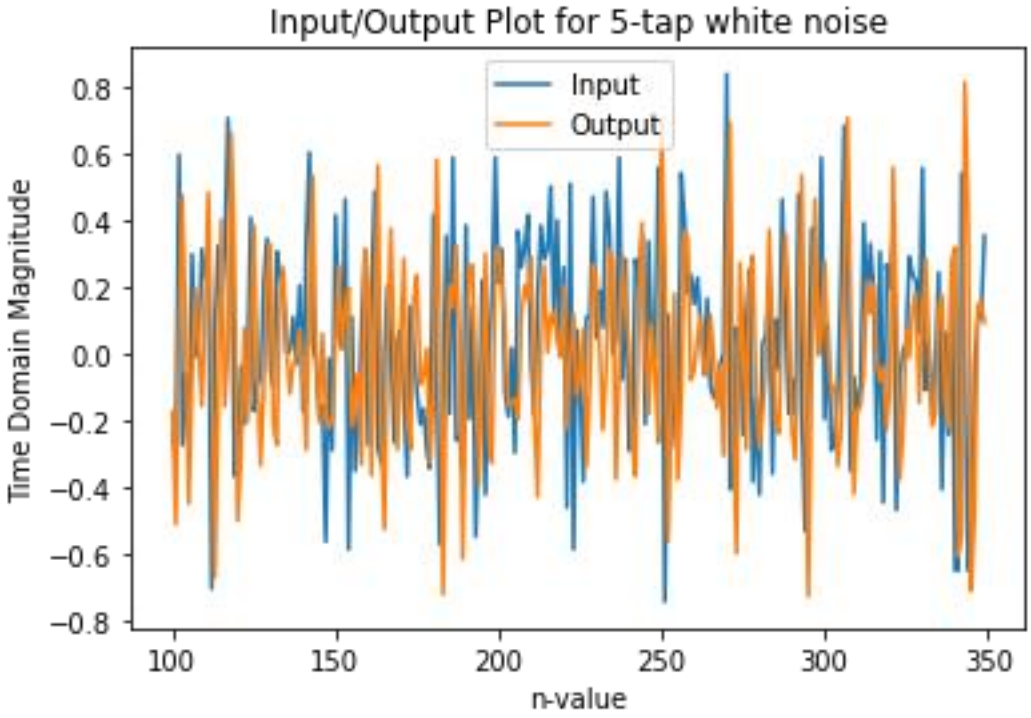
a)Input: DC Signal



b) Input: Sinusoidal signal of frequency 1800Hz

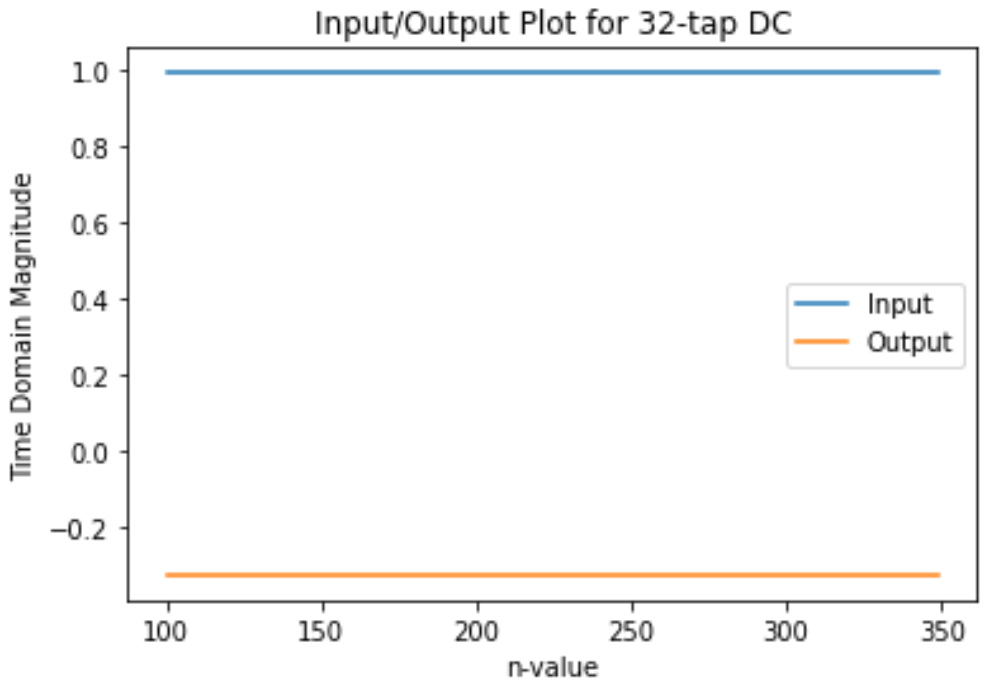


c) Input: white noise signal of variance 0.33

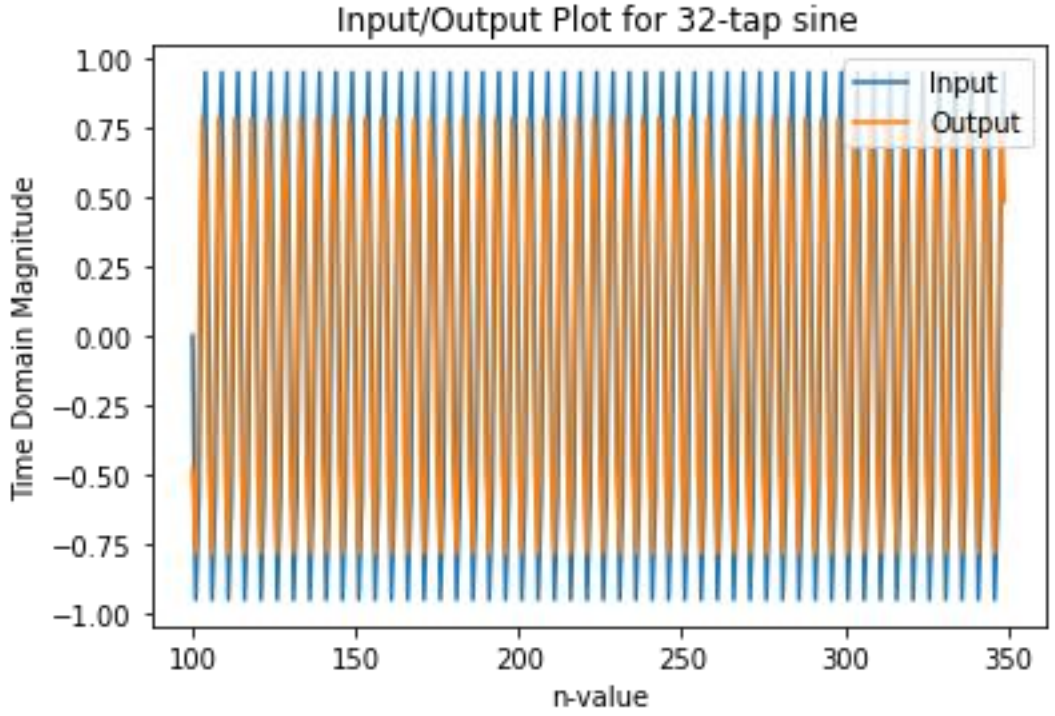


Input-Output plots for the 32-tap case:

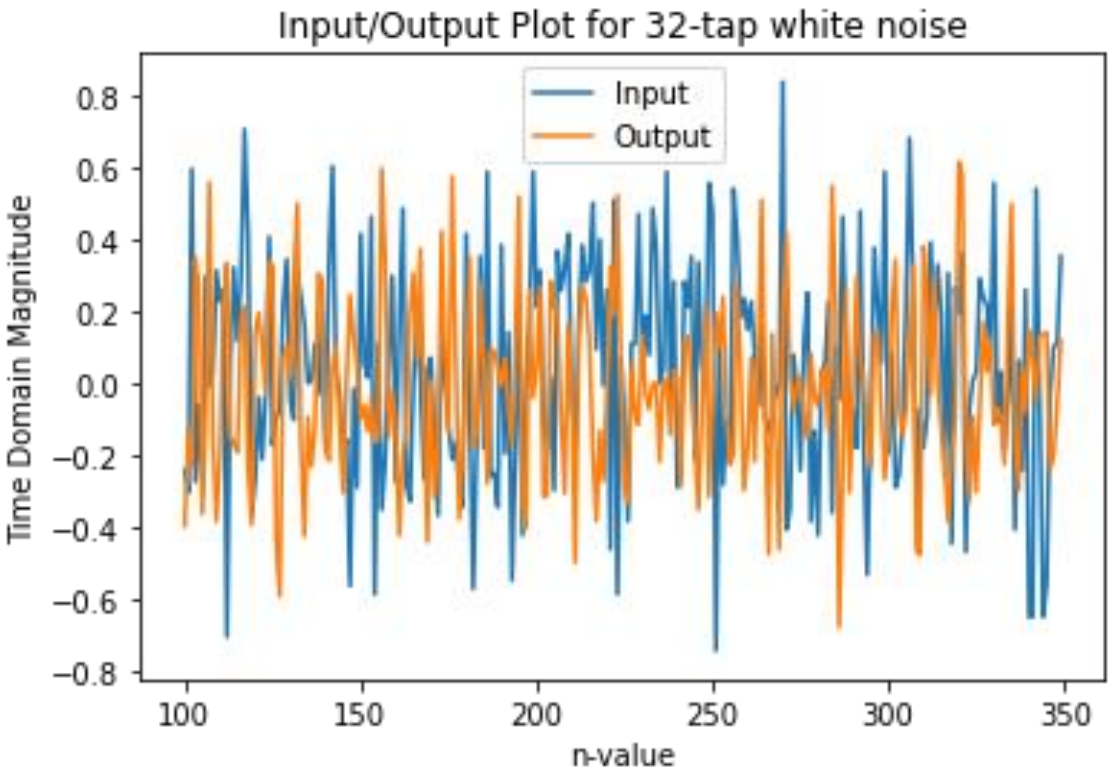
a)Input : DC signal



b)Input: Sinusoidal signal at frequency 1800Hz

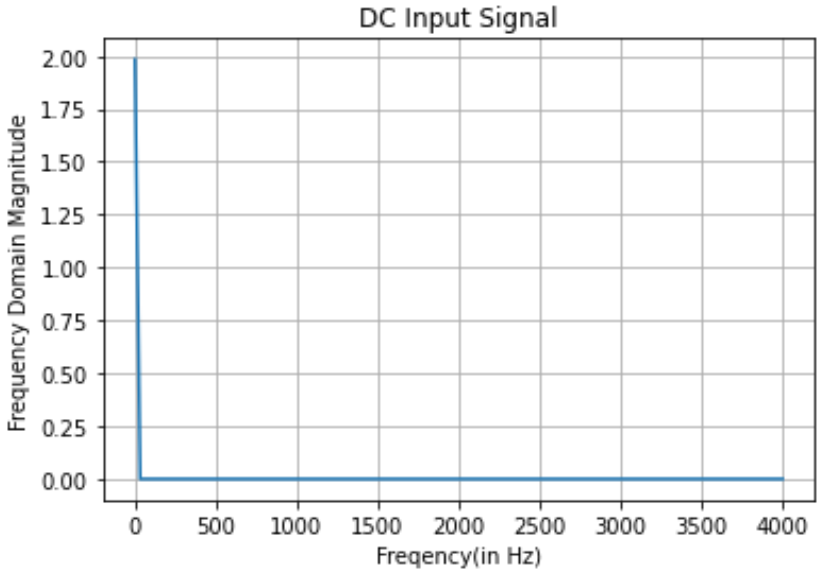


c)Input: White noise signal of variance 0.33

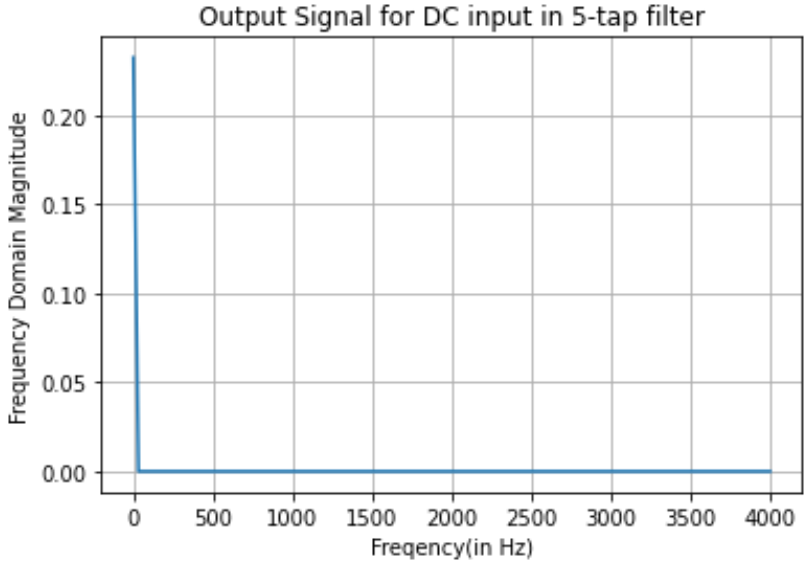


Frequency Domain representation of Output (for 5-tap case):

a)DC Input

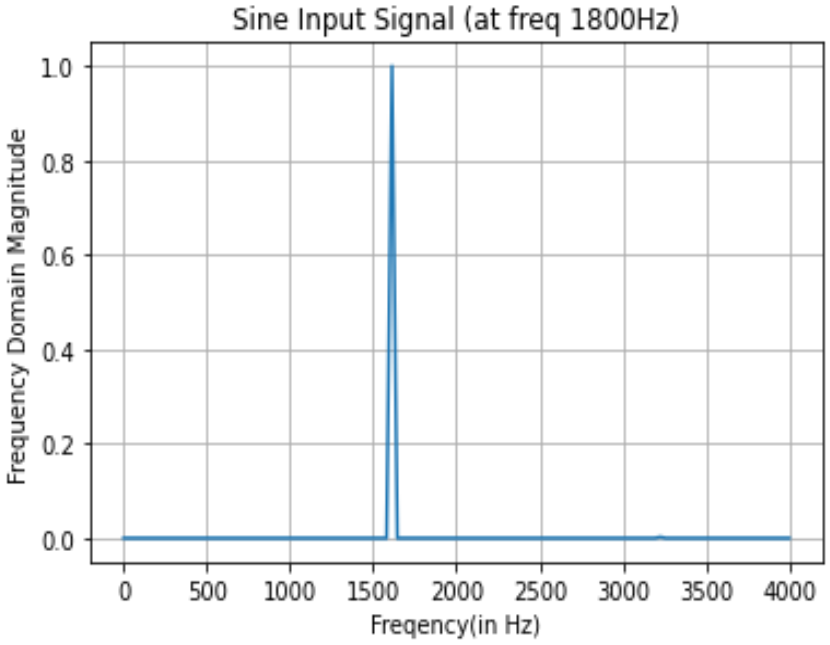


In this signal we see a peak corresponding to the zero frequency, since the DC corresponds to zero frequency.

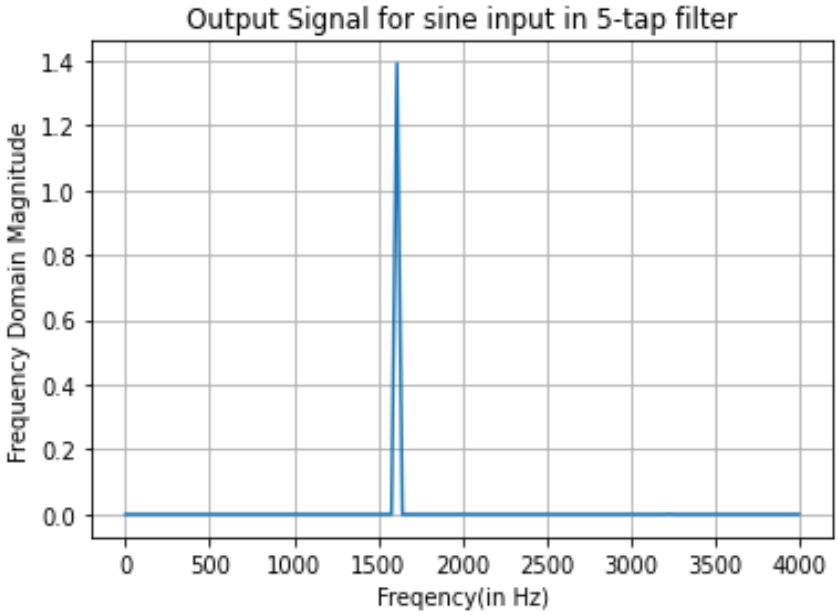


Even in the output we see the same variation, that is the peak at zero frequency, except that there is a scaling of the output, since we multiply the input signal with the coefficients of the filter.

b)Sinusoidal Input of 1800 Hz



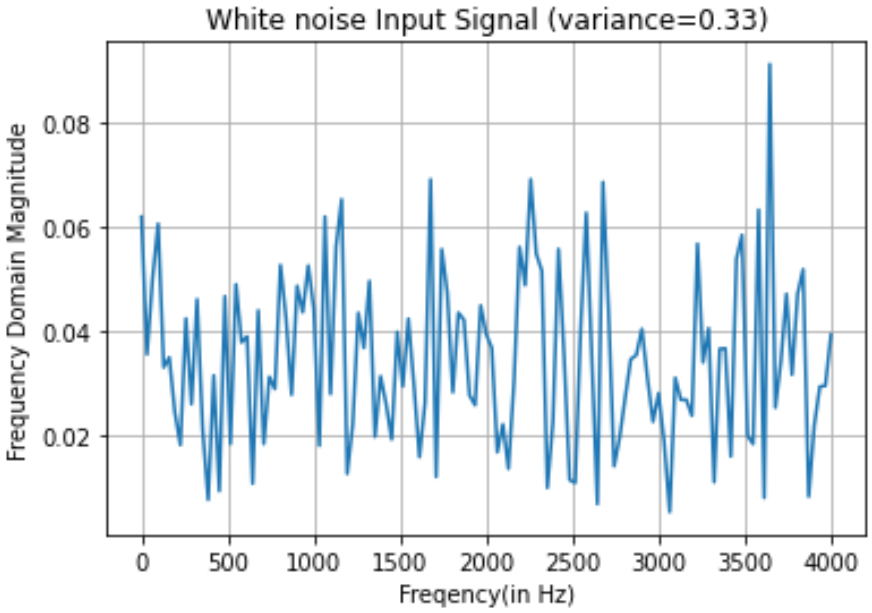
In this signal we see a peak corresponding to the input frequency, since the sine value is of 1800Hz frequency



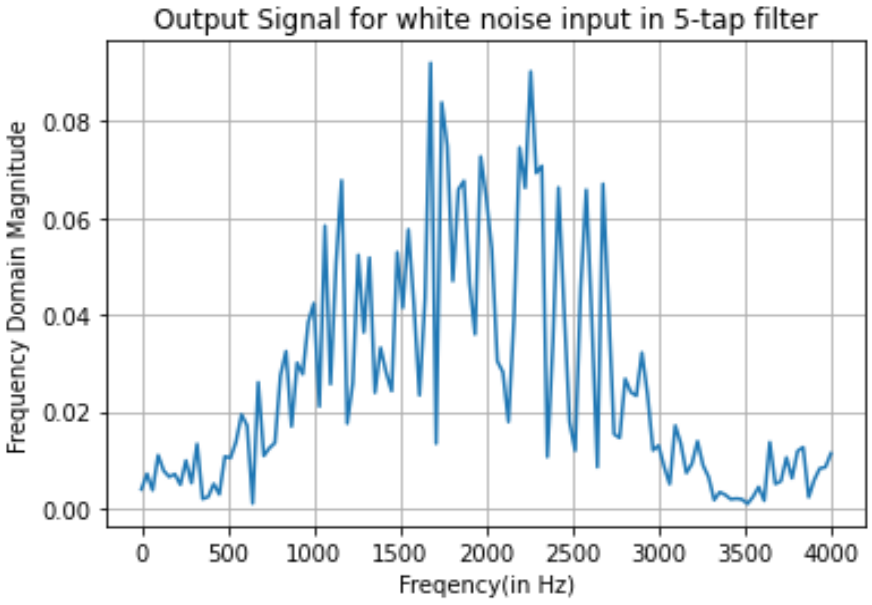
Even in the output we see the same variation, that is the peak at 1800Hz frequency, except that there is a scaling of the output, since we multiply the input signal with the coefficients of the filter.

Also, since the frequency is in the range of the pass-band, it is getting passed on completely.

c) White noise signal of variance 0.33



Here since the input is noise, we see a combination of multiple frequencies in the input, before applying filter.

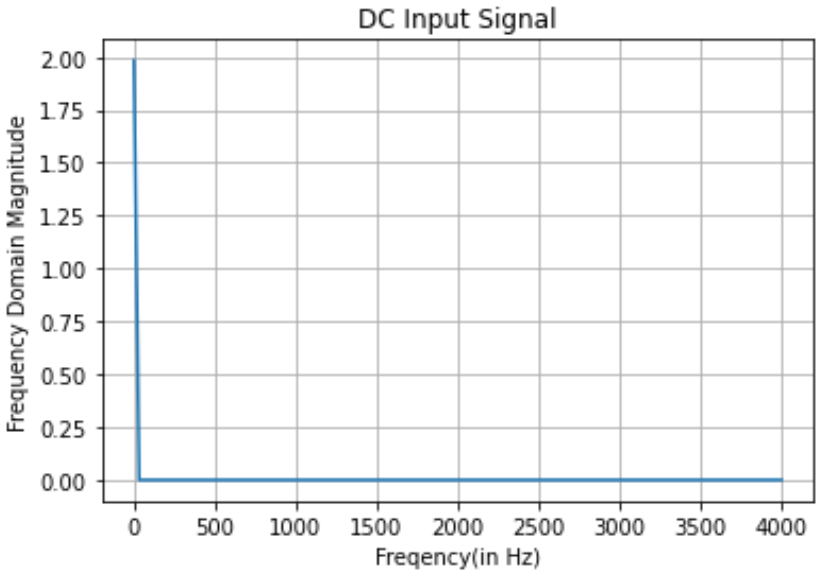


Here in the output plot we see that only the frequencies in the passband range of 800Hz to 3400Hz are passed through completely and the other frequencies outside it are attenuated strongly.

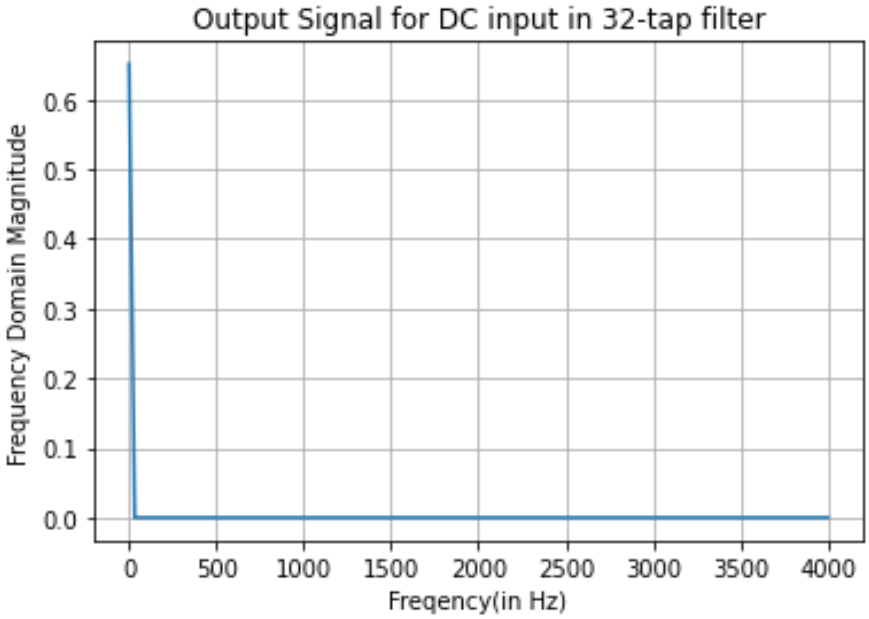
Hence we see a clear working of the bandpass filter from the above plots.

Frequency Domain representation of Output (for 32-tap case):

a)DC Input

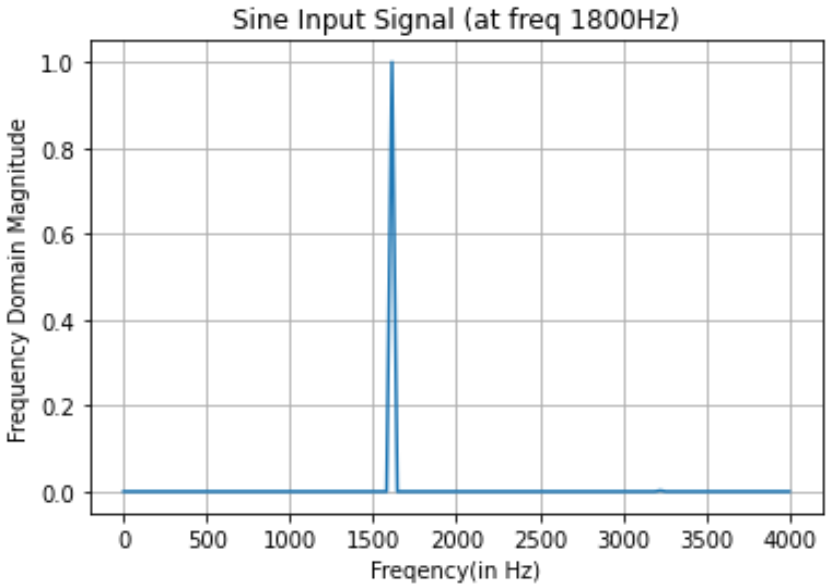


In this signal we see a peak corresponding to the zero frequency, since the DC corresponds to zero frequency.

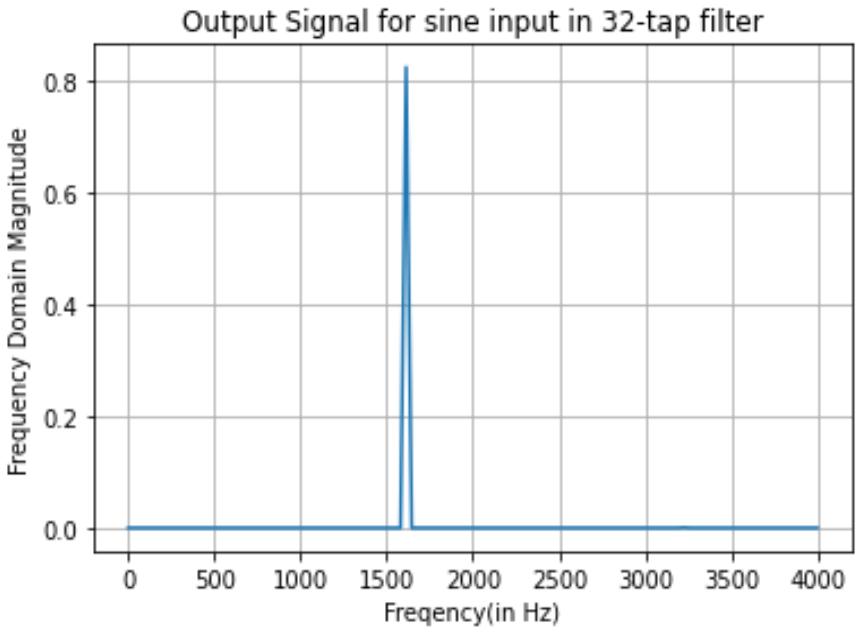


Even in the output we see the same variation, that is the peak at zero frequency, except that there is a scaling of the output, since we multiply the input signal with the coefficients of the filter.

b)Sinusoidal Input of 1800 Hz



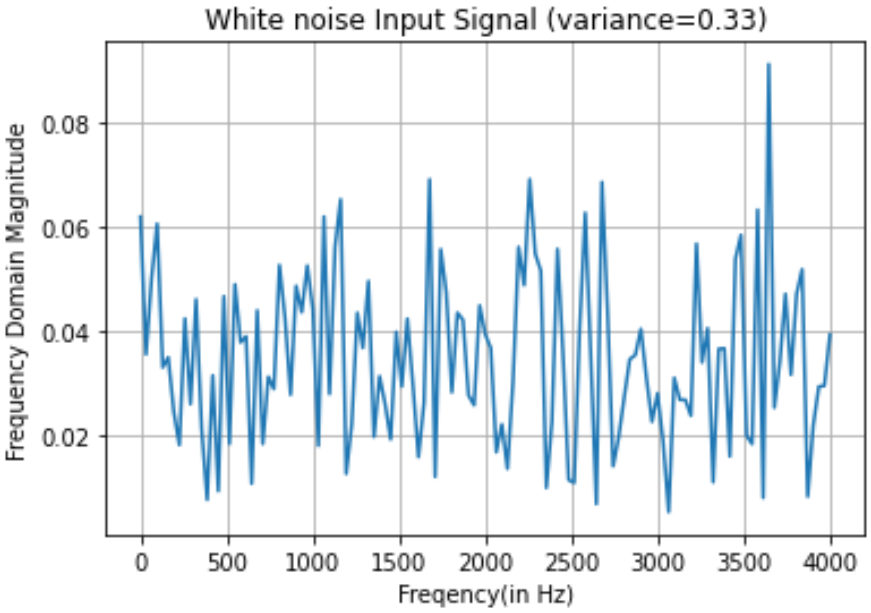
In this signal we see a peak corresponding to the input frequency, since the sine value is of 1800Hz frequency



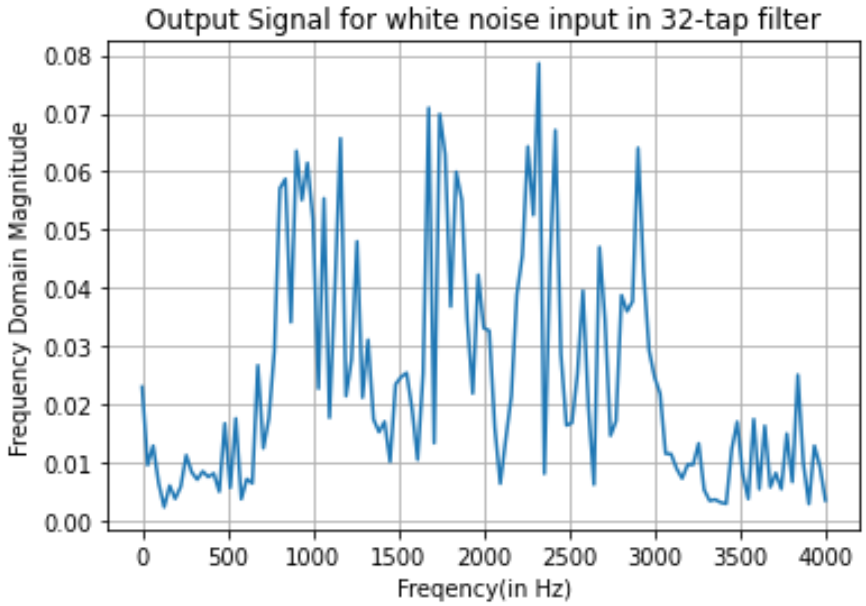
Even in the output we see the same variation, that is the peak at 1800Hz frequency, except that there is a scaling of the output, since we multiply the input signal with the coefficients of the filter.

Also, since the frequency is in the range of the pass-band, it is getting passed on completely.

c) White noise signal of variance 0.33



Here since the input is noise, we see a combination of multiple frequencies in the input, before applying filter.

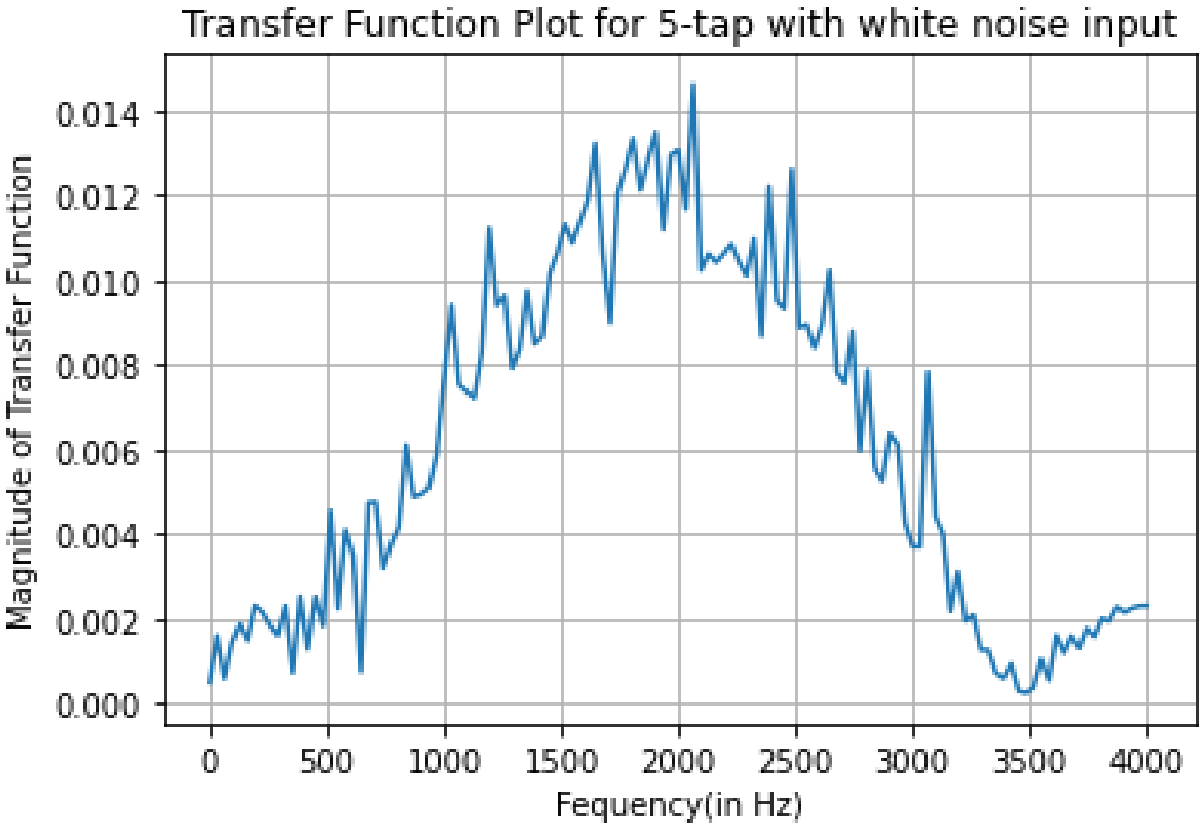


Here in the output plot we see that only the frequencies in the passband range of 800Hz to 3400Hz are passed through completely and the other frequencies outside it are attenuated strongly.

Hence we see a clear working of the bandpass filter from the above plots.

Transfer function plots for the White noise signal:

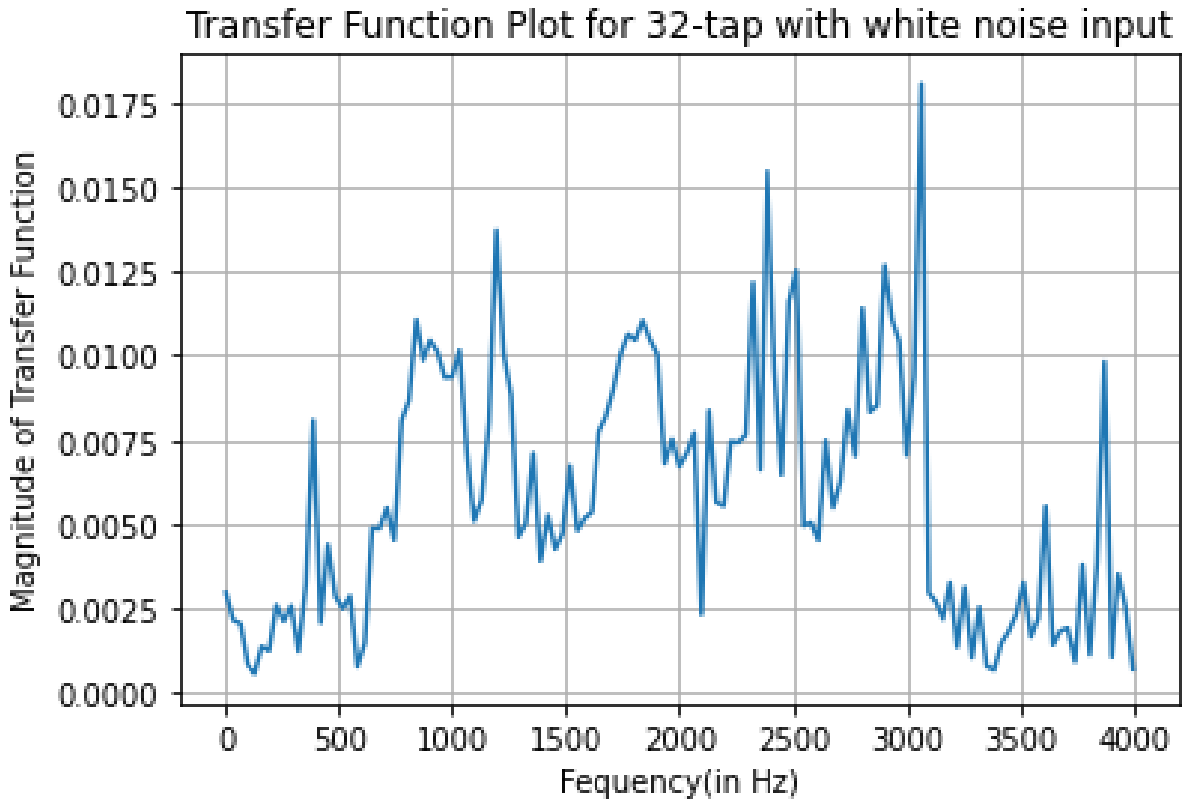
a)For the 5-tap filter case:



Here we can see a clear peak at the frequency around 2100 Hz.

This shows the perfect working of the band-pass filter.

b)For the 32-tap filter case:



The same as in the above case occurs, except that there are multiple peaks, since the number of taps increases.