

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,

$$y[n] = b_0 x[n] + b_1 x[n-1] + \dots + b_N x[N-1]$$

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

Where,

$\rightarrow x[n]$ is the input signal

$\rightarrow y[n]$ is the output signal

$\rightarrow N$ is the order of the filter, ($N = \text{number of taps} + 1$)

$\rightarrow b_i$ 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.

\rightarrow The values $x[n-i]$ here are what being referred to as taps, based on the structure 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

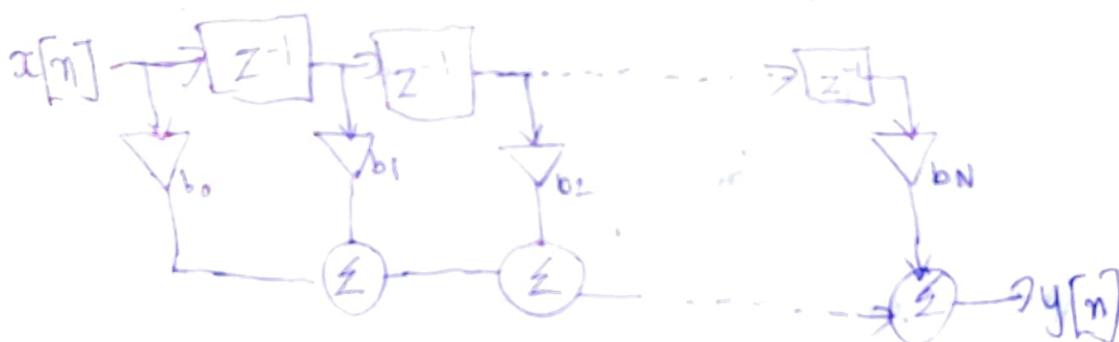
$$h[n] = \sum_{i=0}^N b_i \delta[n-i] = \begin{cases} b_n & 0 \leq n \leq N \\ 0 & \text{elsewhere} \end{cases}$$

where

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

b_i are the coefficients

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



where
 $N = \text{order of the filter}$

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 as,

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

$$Y(\omega) \quad X(\omega) \quad H(\omega)$$

where the operations 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"

(a)

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

(b)

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

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 ~~numbers~~, we use the (1.7) notation where the floating point is placed 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 FHAL 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 ~~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}), 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:

$$N \geq 2 \cdot K + \log_2 M + 1 \quad (K \rightarrow \text{bit resolution ex. 6 sign bit})$$

(M → number of addition)

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-bit 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 the numbers, we just assume that as carry and include that result also while dealing the output.
- But in the case of overflow occurring 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 subtract 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 being embedded in it.

→ Implementation of above method:

- * Keep a register and increase it by 2 every time we add -ve numbers, $\alpha^2(i)$
- * Do the accumulation as normal 24-bit addition.
- * Subtract the α value (i) from the integral part of accumulated result.

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

→ In the code this has been done by copying the two bytes (HSB and middle bytes) 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 ~~inputs~~ inputs and coefficients by $2^7, 2^7$ respectively and so we need to downscale the accumulated result by 2^4 .

Interpretation of the result :-

→ we get a 24-bit result and if the high byte is HSB 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 need and we just divide the result by 2^{15} to get the final decimal answer (same as above the 1.15 format)

Section-II : 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 along with the number of samples and ~~the output~~ gives us the DTFT of the output. We then plot them over the range of frequency and interpret our results.

Section-II : 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 fixe number) and for sin 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 ~~put~~ 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 (x0090), from this onwards every 3 bytes corresponds to output of one value of n .
- Since we are using LPN 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 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. 2) 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 'm' value is calculated it resets again to the 0x0060 location.

3. Multiplication details (multiply and accumulate)

- we need to an ~~80~~ accumulator of size 24 bits to store final result.
- here x_{10}, x_9, x_8 acts as the accumulator, and the register x_{11} stores the 2^k (no. of negative number add's made)
- After the complete MAC process we subtract the x_{11} register the integer part of result (i.e.) x_{10} along with the MSB of x_9 , this done by moving them to separate registers, using shifts and finally storing them ~~through~~ ~~in~~ back in x_{10} and x_9 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 types)
- iii) reset the coefficient pointer

Load the inputs and the coefficients into the registers

- i) Perform Multiply & Accumulate
- ii) decrement the Counter2

check Counter2
 $\neq 0$

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
Yes
No
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. db's defined.
 - num1 : \Rightarrow 5 tap coefficients
 - num2 : \Rightarrow 32 tap coefficients
 - num3 : \Rightarrow Sinl Signal input
 - num4 : \Rightarrow 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 ~~L~~ LDI with 0xFF because it is a constant signal.
- e) The output calculated are for the range $n=100$ to $n=249$.
- f) The input (pre-processing), output (post-processing) is done using python and plots are also generated in python.
- g) The FFT (~~DTFT~~ 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

The screenshot shows the AtmelStudio interface with the following details:

- Title Bar:** EE19B116_MIDSEM_PARTC (Debugging) - AtmelStudio
- Menu Bar:** File, Edit, View, VAssistX, ASF, Project, Build, Debug, Tools, Window, Help
- Toolbar:** Includes icons for file operations, search, and various debug tools.
- Code Editor:** Displays assembly code for `working_code.asm`. A yellow box highlights the instruction `rjmp here`, with a note: "here: rjmp here; the end operation, to see the memory locations after".
- Memory View:** Shows the memory dump starting at address 0x0090. The data is mostly zeros (0x00) with some non-zero values at higher addresses. The memory type is listed as `prog FLASH`.
- Autos View:** Shows the current values of variables in the Autos window.
- Windows Taskbar:** Shows the Windows taskbar with the AtmelStudio icon, a search bar, and system status indicators.

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

The screenshot shows the Atmel Studio interface with the following details:

- Title Bar:** EE19B116_MIDSEM_PARTC (Debugging) - AtmelStudio
- Menu Bar:** File, Edit, View, VAssistX, ASF, Project, Build, Debug, Tools, Window, Help
- Toolbar:** Includes icons for file operations, build, debug, and simulation.
- Code Editor:** Displays assembly code for `working_code.asm`. The current line is highlighted with a red border:


```
here: rjmp here; the end operation, to see the memory locations after
```
- Memory View:** Shows a dump of memory starting at address 0x0090. The data is mostly zeros (ff 6c 00 68 94 00) with some non-zero values (a9 66 00 00 00 ff). The memory dump is truncated at the bottom.

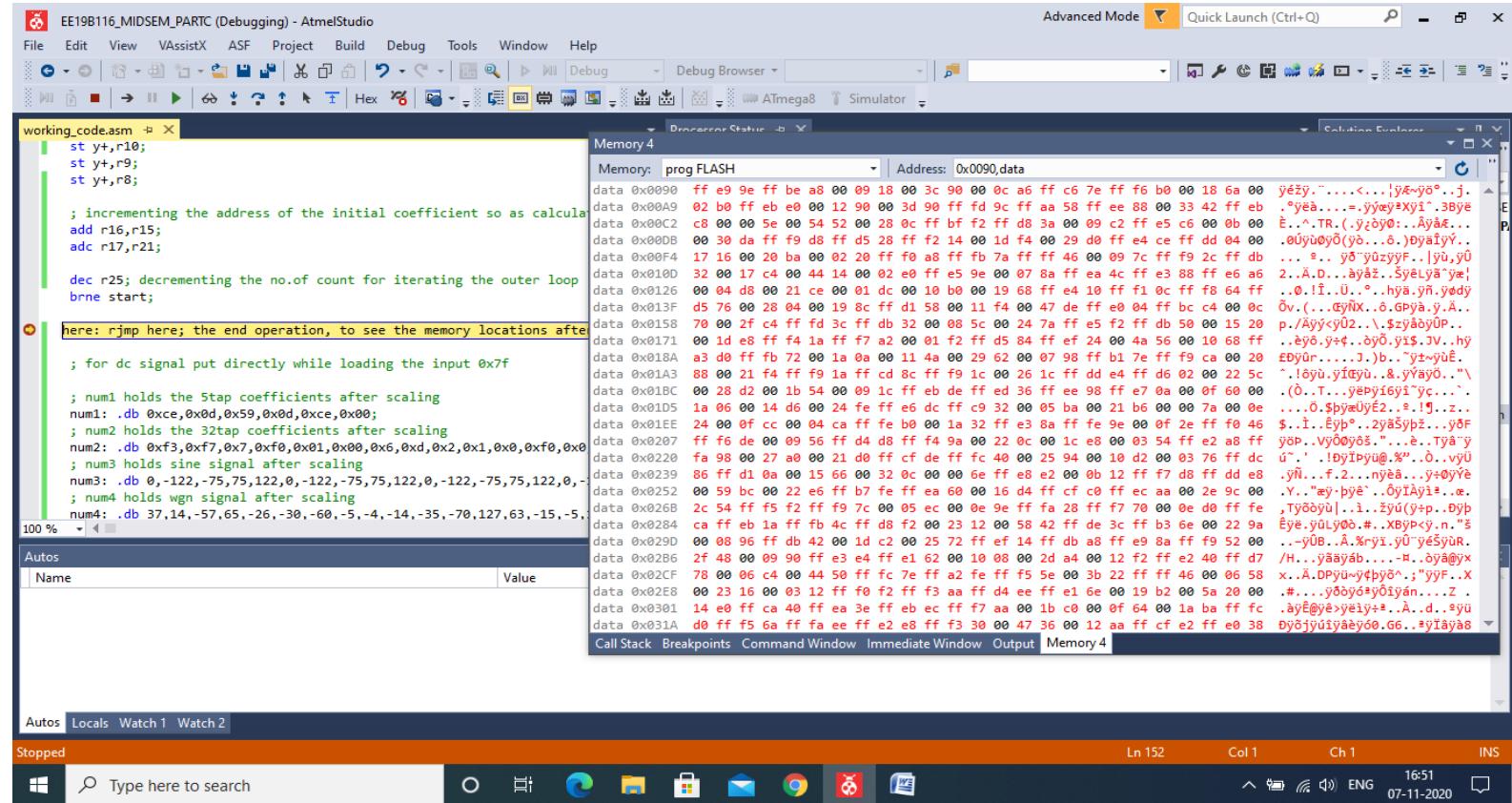
Address	Value	Comment
0x0090	00 a9 66 00 00 00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff .of...yvšy-1.h".of...yvšy	
0x00A9	97 6c 00 68 94 00 a9 66 00 00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff .of...yvšy-1.h".of...yvšy	
0x00C2	00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff 97 6c 00 68 94 .of...yvšy-1.h".of...yvšy	
0x00D8	00 a9 66 00 00 00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff .of...yvšy-1.h".of...yvšy	
0x00F4	97 6c 00 68 94 00 a9 66 00 00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff .of...yvšy-1.h".of...yvšy	
0x010D	00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff 97 6c 00 68 94 .of...yvšy-1.h".of...yvšy	
0x0126	00 a9 66 00 00 00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff .of...yvšy-1.h".of...yvšy	
0x013F	97 6c 00 68 94 00 a9 66 00 00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff .of...yvšy-1.h".of...yvšy	
0x0158	00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff 97 6c 00 68 94 .of...yvšy-1.h".of...yvšy	
0x0171	00 a9 66 00 00 00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff .of...yvšy-1.h".of...yvšy	
0x018A	97 6c 00 68 94 00 a9 66 00 00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff .of...yvšy-1.h".of...yvšy	
0x01A3	00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff 97 6c 00 68 94 .of...yvšy-1.h".of...yvšy	
0x01BC	00 a9 66 00 00 00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff .of...yvšy-1.h".of...yvšy	
0x01D5	97 6c 00 68 94 00 a9 66 00 00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff .of...yvšy-1.h".of...yvšy	
0x01EE	00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff 97 6c 00 68 94 .of...yvšy-1.h".of...yvšy	
0x0207	00 a9 66 00 00 00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff .of...yvšy-1.h".of...yvšy	
0x0220	97 6c 00 68 94 00 a9 66 00 00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff .of...yvšy-1.h".of...yvšy	
0x0239	00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff 97 6c 00 68 94 .of...yvšy-1.h".of...yvšy	
0x0252	00 a9 66 00 00 00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff .of...yvšy-1.h".of...yvšy	
0x026B	97 6c 00 68 94 00 a9 66 00 00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff .of...yvšy-1.h".of...yvšy	
0x0284	00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff 97 6c 00 68 94 .of...yvšy-1.h".of...yvšy	
0x029D	00 a9 66 00 00 00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff .of...yvšy-1.h".of...yvšy	
0x02B6	97 6c 00 68 94 00 a9 66 00 00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff .of...yvšy-1.h".of...yvšy	
0x02CF	00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff 97 6c 00 68 94 .of...yvšy-1.h".of...yvšy	
0x02E8	00 a9 66 00 00 00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff .of...yvšy-1.h".of...yvšy	
0x0301	97 6c 00 68 94 00 a9 66 00 00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff .of...yvšy-1.h".of...yvšy	
0x031A	00 ff 56 9a ff 97 6c 00 68 94 00 a9 66 00 00 00 ff 56 9a ff 97 6c 00 68 94 .of...yvšy-1.h".of...yvšy	
- Toolbars:** Call Stack, Breakpoints, Command Window, Immediate Window, Output, Memory 4.
- Bottom Status Bar:** Ready, Ln 152, Col 1, Ch 1, 16:48, 07-11-2020, INS.

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

The screenshot shows the AtmelStudio interface with the following details:

- Title Bar:** EE19B116 MIDSEM PARTC (Debugging) - AtmelStudio
- Menu Bar:** File, Edit, View, VAssistX, ASF, Project, Build, Debug, Tools, Window, Help
- Toolbar:** Includes icons for Open, Save, Build, Run, Stop, and various debug tools.
- Code Editor:** Displays assembly code for `working_code.asm`. The code includes instructions for storing values into memory, incrementing addresses, and decrementing loop counters. A specific line is highlighted: `here: rjmp here; the end operation, to see the memory locations after`.
- Memory View:** A large window titled "Memory 4" showing memory starting at address 0x0090. The data is displayed in hex format, primarily consisting of FF and 00 values, indicating a repeating pattern of zeros and ones.
- Autos Watch:** Shows the current state of variables in the Autos window.
- Bottom Status Bar:** Displays the status bar with information like "Ln 152", "Col 1", "Ch 1", "INS", and the date/time "07-11-2020 16:50".

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

The screenshot shows the Atmel Studio interface with the following details:

- Title Bar:** EE19B116_MIDSEM_PARTC (Debugging) - AtmelStudio
- File Menu:** File, Edit, View, VAssistX, ASF, Project, Build, Debug, Tools, Window, Help
- Toolbar:** Standard toolbar with icons for file operations, search, and debug.
- Memory View:**
 - Address: 0x0090, data
 - Memory: prog FLASH
 - Content: A dump of memory starting at address 0x0090, showing data words in hex format. The data is mostly zeros with some scattered non-zero values, representing the 32-tap output for a white noise input.
- Code Editor:**

```

working_code.asm
st y+,r10;
st y+,r9;
st y+,r8;

; incrementing the address of the initial coefficient so as calculate
add r16,r15;
adc r17,r21;

dec r25; decrementing the no.of count for iterating the outer loop
brne start;

here: rjmp here; the end operation, to see the memory locations after
      ; for dc signal put directly while loading the input 0x7f

; num1 holds the 5tap coefficients after scaling
num1: .db 0xce,0x0d,0x59,0xd,0xce,0x0;
; num2 holds the 32tap coefficients after scaling
num2: .db 0xf3,0xf7,0x7,0xf0,0x01,0x00,0x6,0xd,0x2,0x1,0x0,0xf0,0x0
; num3 holds sine signal after scaling
num3: .db 0,-122,-75,75,122,0,-122,-75,75,122,0,-122,-75,75,122,0,-
; num4 holds wgn signal after scaling
num4: .db 37,14,-57,65,-26,-30,-60,-5,-4,-14,-35,-70,127,63,-15,-5,
```
- Autos View:** Shows the current values of variables in the Autos window.
- Locals View:**
- Watch View:**
- Status Bar:** Shows the current line (Ln 152), column (Col 1), character (Ch 1), and date/time (16:52 07-11-2020).

Giving the inputs using num: .db

The screenshot shows the Atmel Studio interface with the following components:

- Title Bar:** EE19B116_MIDSEM_PARTC - AtmelStudio
- Menu Bar:** File, Edit, View, VAssistX, ASF, Project, Build, Debug, Tools, Window, Help
- Toolbar:** Includes icons for file operations, search, and various development tools.
- Code Editor:** Displays assembly code for `working_code.asm`. The code includes instructions like `add r16,r15`, `adc r17,r21`, and a loop with a jump instruction. It also defines memory variables using the `num` directive, such as `num1: .db 0xce,0x0d,0x59,0xd,0xce,0x00` and lists of coefficients for various taps.
- Solution Explorer:** Shows the project structure with files like `EE19B116_MIDSEM_PAR`, `Dependencies`, `Labels`, `Output Files`, `AsmFile2.asm`, and `dc_5Tap.asm`.
- Error List:** Shows 0 Errors, 0 Warnings, and 0 Messages.
- Output:** Shows the status as "Ready".
- System Tray:** Includes icons for battery, signal strength, volume, language (ENG), date (07-11-2020), and time (20:16).

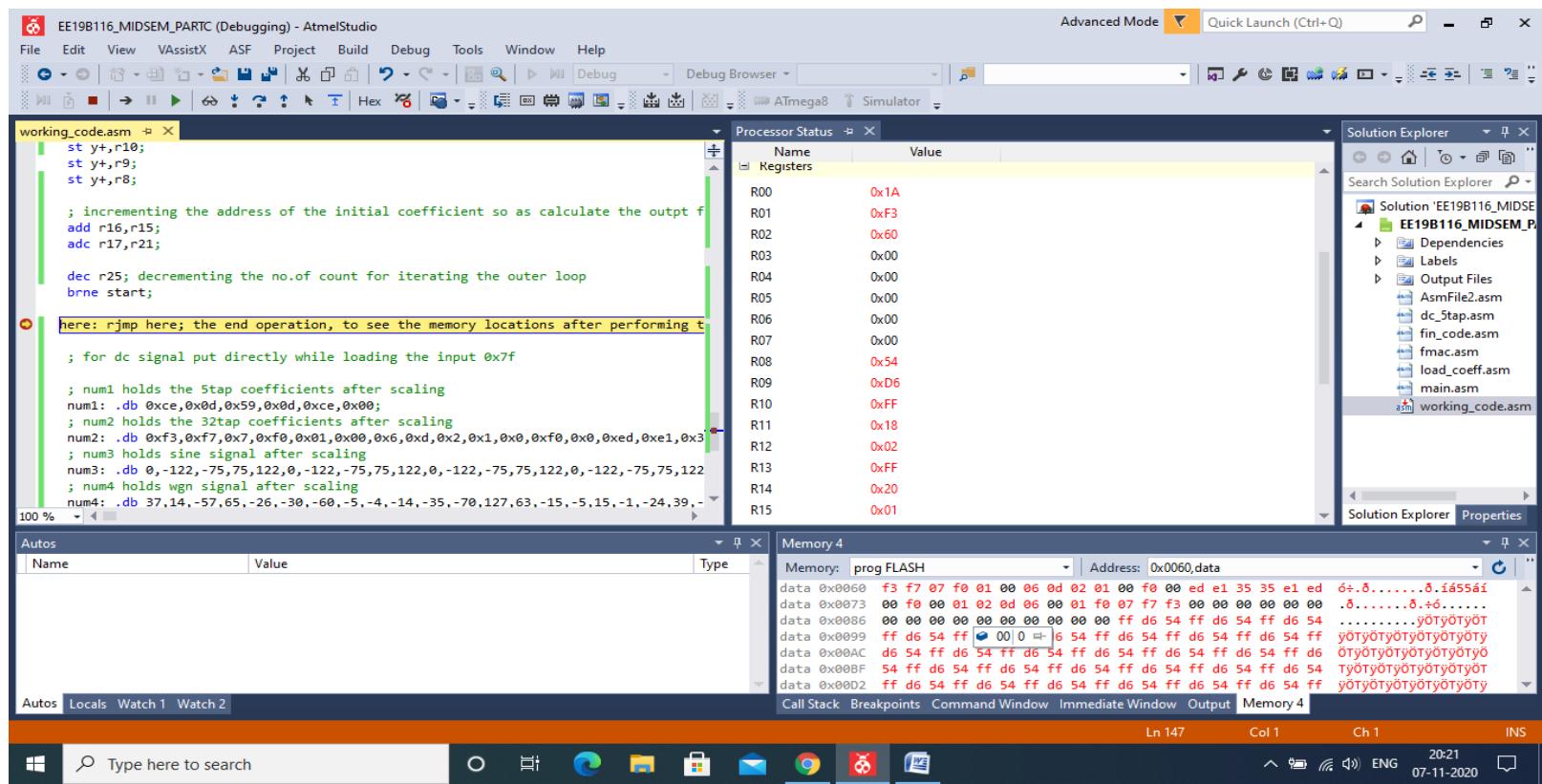
The coefficients are loaded using num: .db and store it in the SRAM at 0x0060 at the location:

a)for 5-tap case

The screenshot shows the Atmel Studio interface with the following details:

- File Explorer:** Shows the project structure for 'EE19B116_MIDSEM_P' with files like AsmFile2.asm, dc_Stap.asm, fin_code.asm, fmac.asm, load_coeff.asm, main.asm, and working_code.asm.
- Processor Status:** Registers window showing values for R00 to R15.
- Memory Dump:** Memory 4 window showing the content of prog FLASH starting at address 0x0060, which contains the loaded coefficients.
- Code Editor:** working_code.asm file containing assembly code for a 5-tap case, including comments about scaling and coefficient loading.
- Solution Explorer:** Shows the project files and their locations.
- Taskbar:** Includes the Windows Start button, search bar, and pinned application icons for File Explorer, File Manager, Task View, Edge, File History, Mail, Google Chrome, and FileZilla.

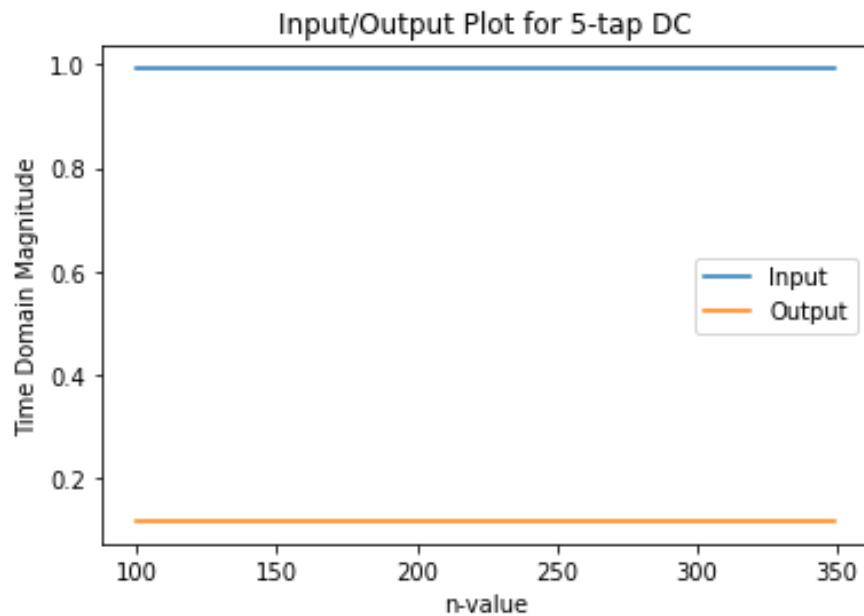
b)for 32-tap case



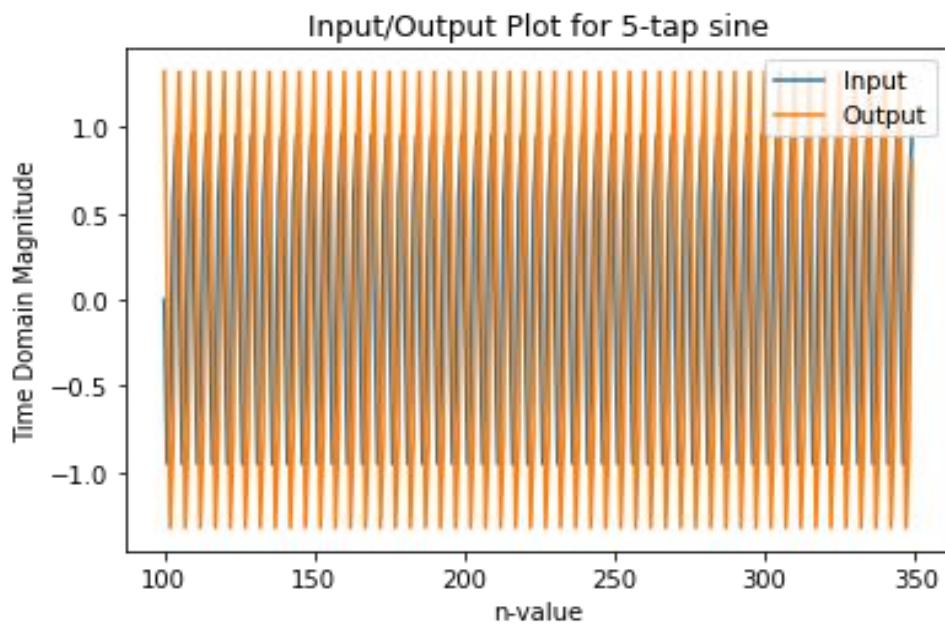
Results

Input-Output plots for the 5-tap case:

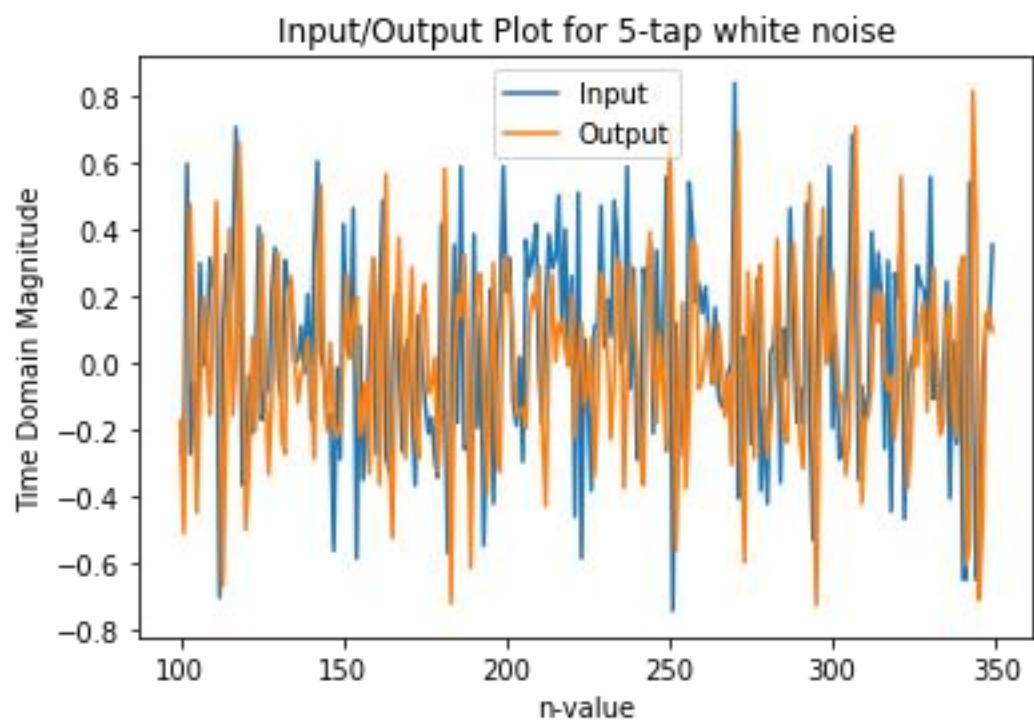
a) Input: DC Signal



b) Input: Sinusoidal signal of frequency 1800Hz

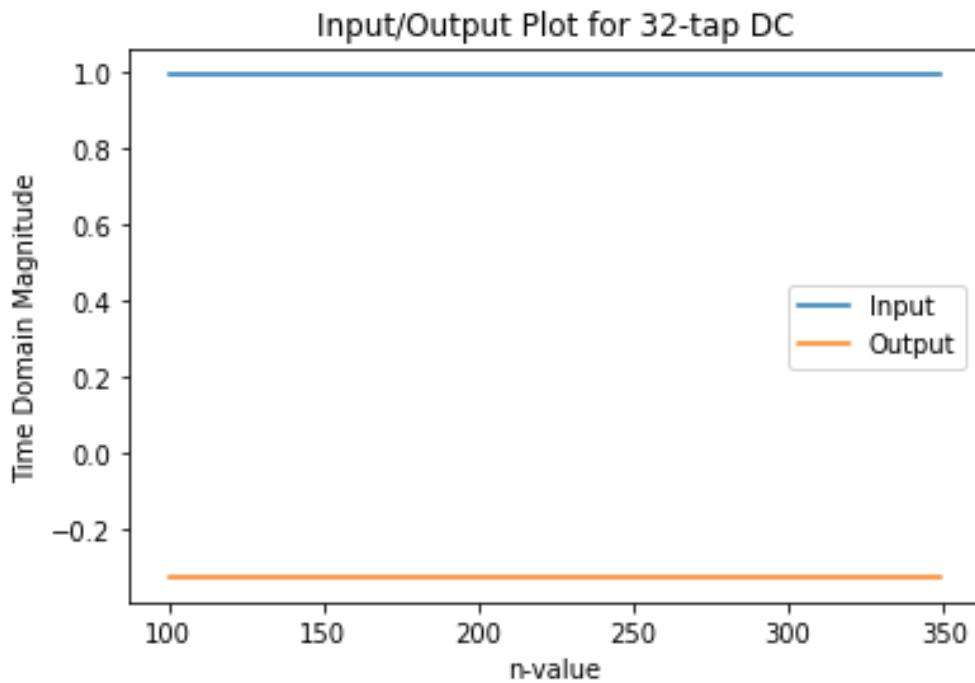


c) Input: white noise signal of variance 0.33

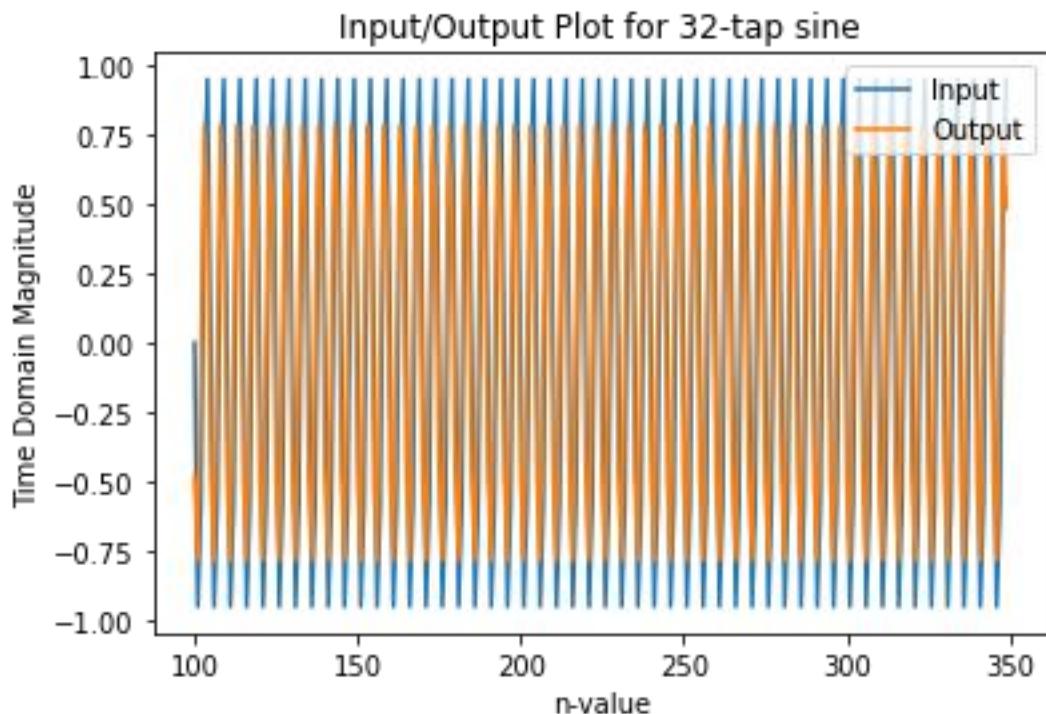


Input-Output plots for the 32-tap case:

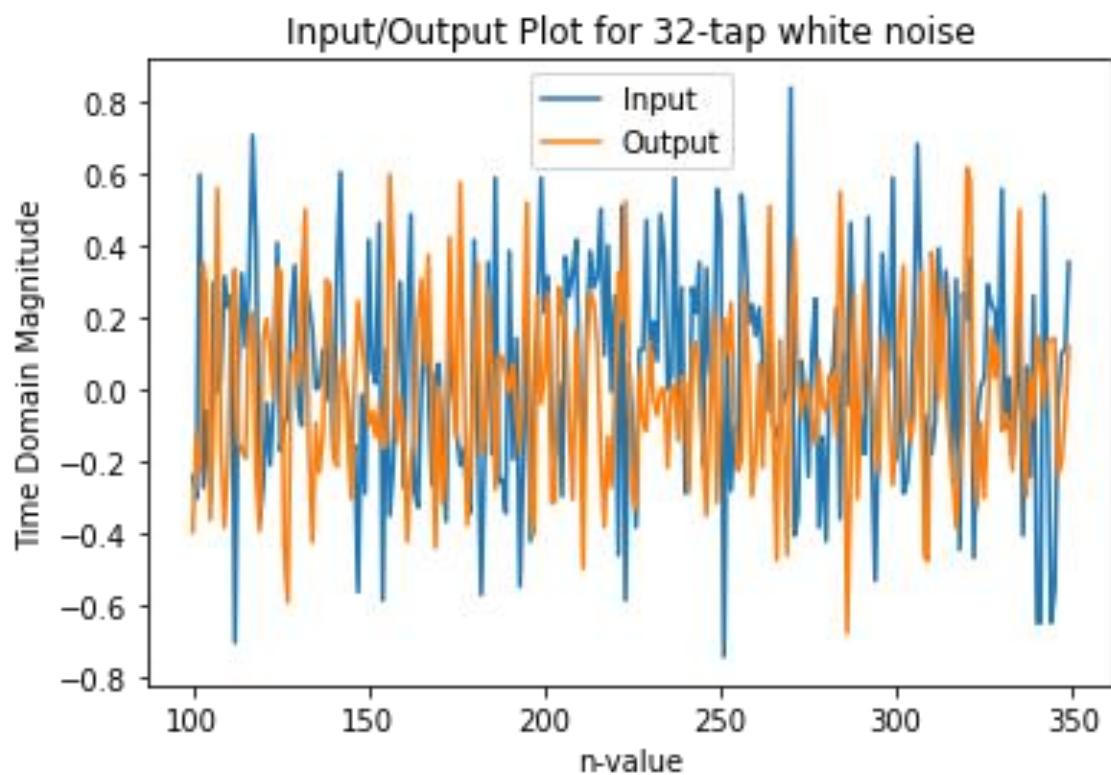
a) Input : DC signal



b) Input: Sinusoidal signal at frequency 1800Hz

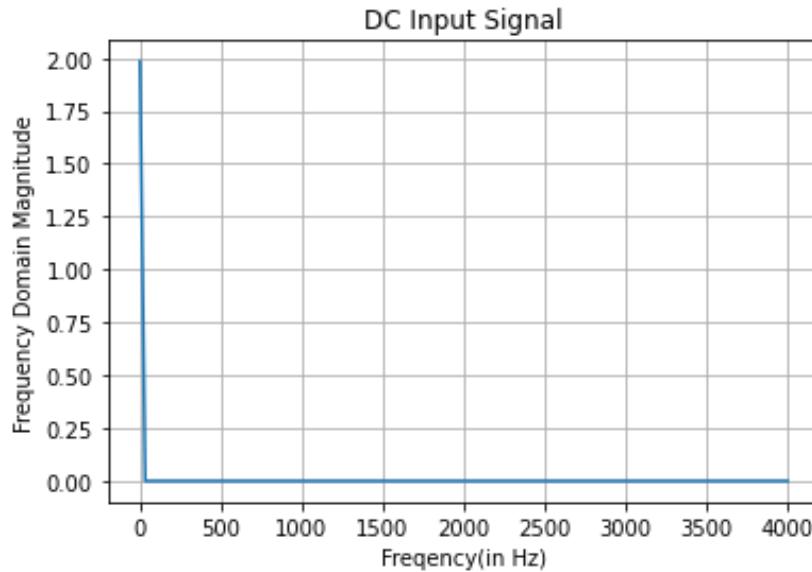


c) Input: White noise signal of variance 0.33

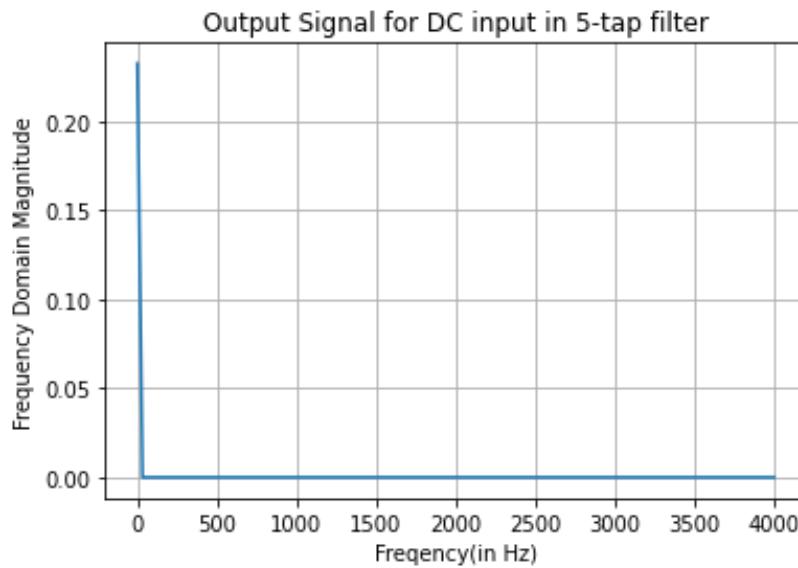


Frequency Domain representation of Output (for 5-tap case):

a)DC Input

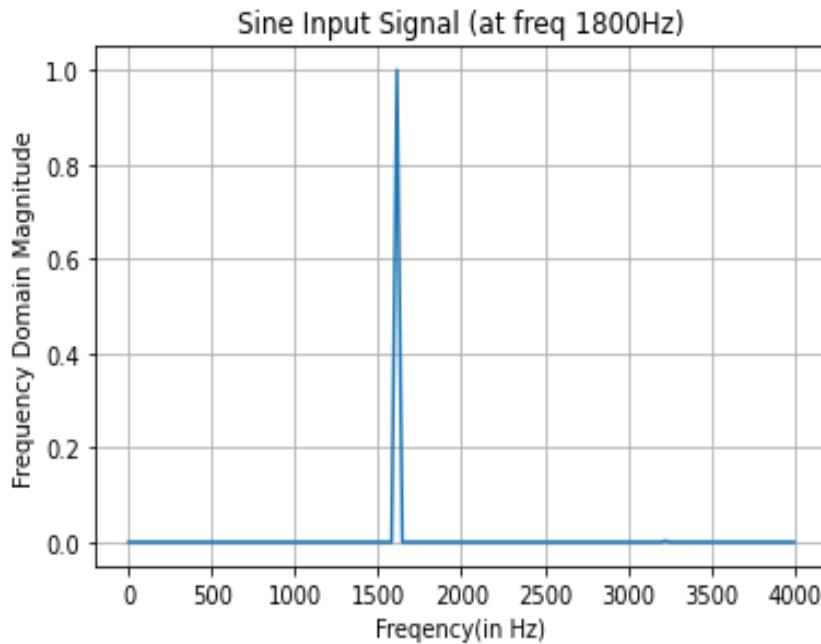


In this signal we see a peak corresponding to the zero frequency, since the DC corresponds to zero frequency.

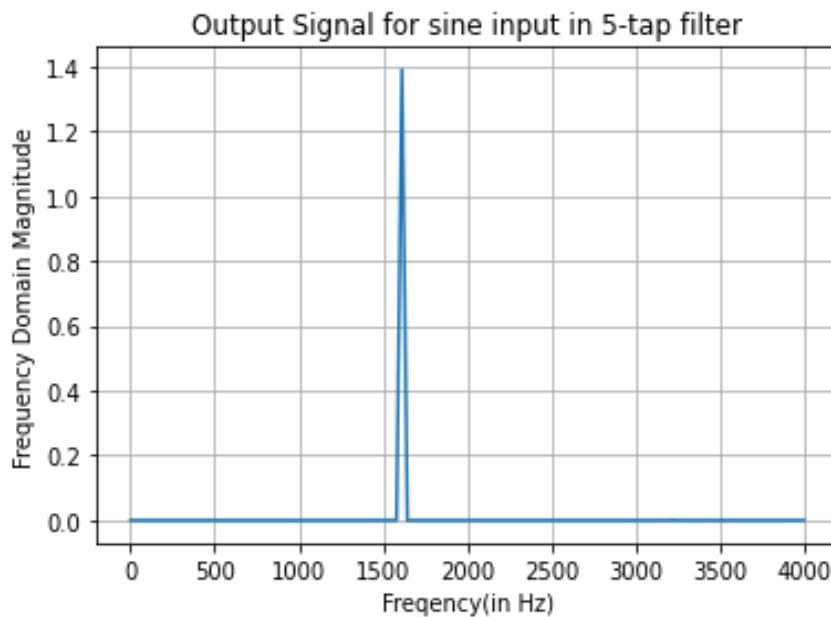


Even in the output we see the same variation, that is the peak at zero frequency, except that there is a scaling of the output, since we multiply the input signal with the coefficients of the filter.

b) Sinusoidal Input of 1800 Hz



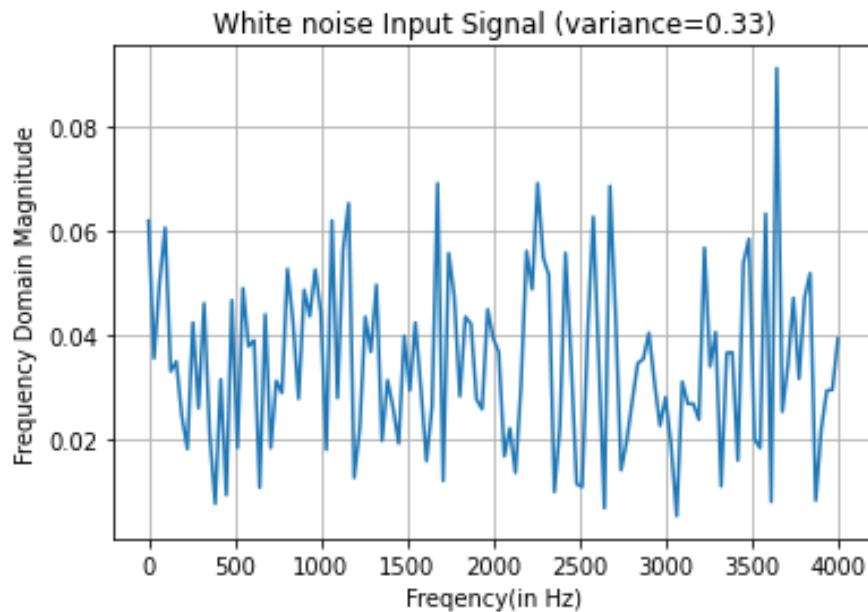
In this signal we see a peak corresponding to the input frequency, since the sine value is of 1800Hz frequency



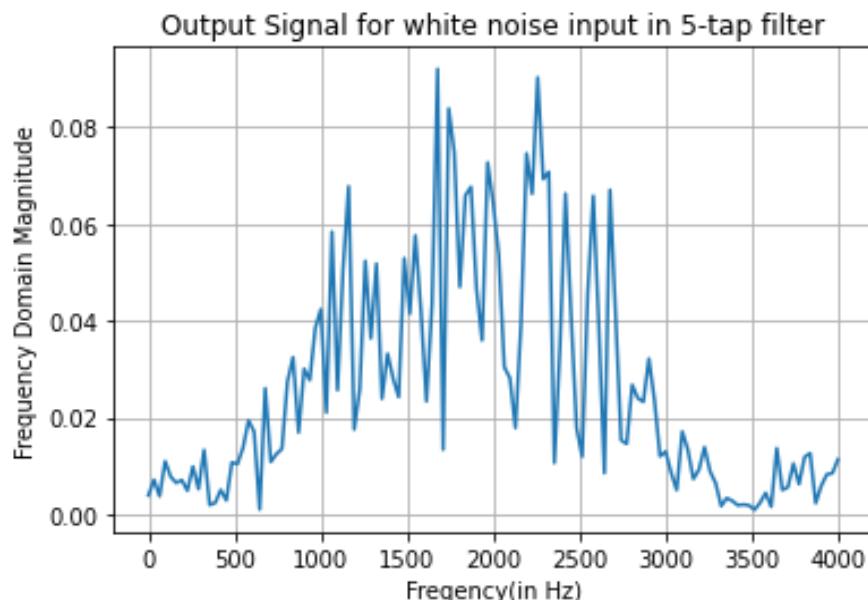
Even in the output we see the same variation, that is the peak at 1800Hz frequency, except that there is a scaling of the output, since we multiply the input signal with the coefficients of the filter.

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.

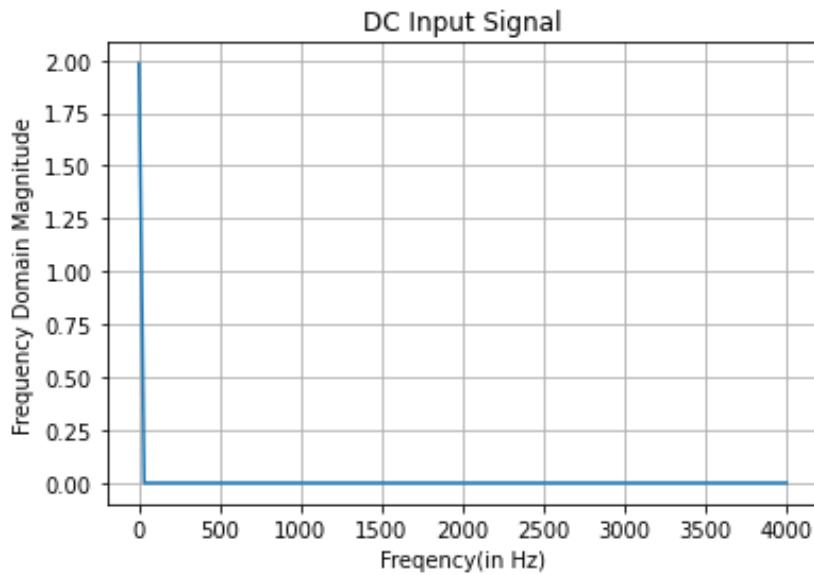


Here in the output plot we see that only the frequencies in the passband range of 800Hz to 3400Hz are passed through completely and the other frequencies outside it are attenuated strongly.

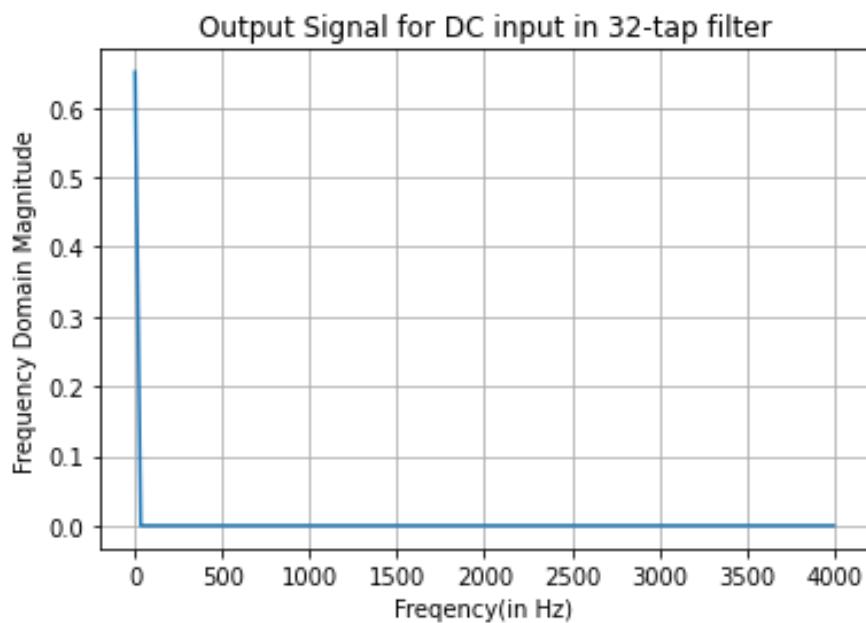
Hence we see a clear working of the bandpass filter from the above plots.

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

a)DC Input

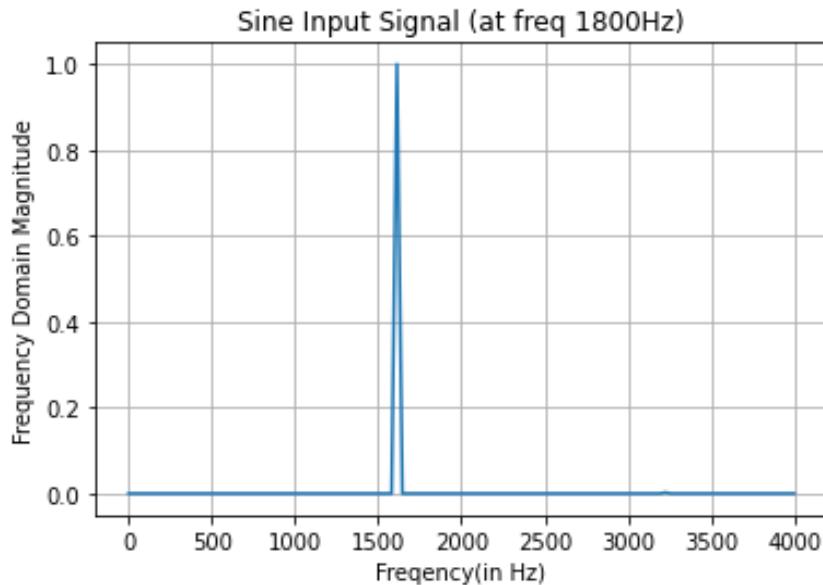


In this signal we see a peak corresponding to the zero frequency, since the DC corresponds to zero frequency.

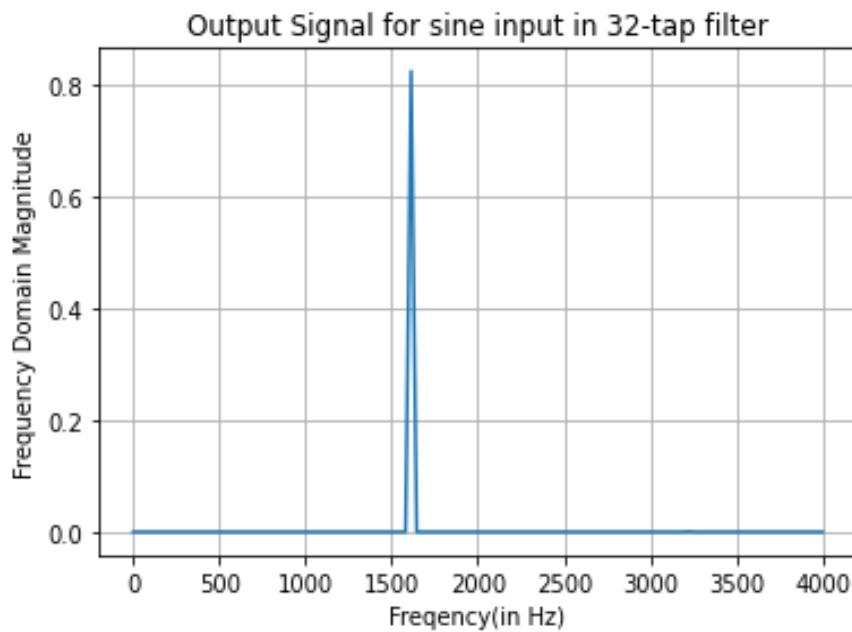


Even in the output we see the same variation, that is the peak at zero frequency, except that there is a scaling of the output, since we multiply the input signal with the coefficients of the filter.

b)Sinusoidal Input of 1800 Hz



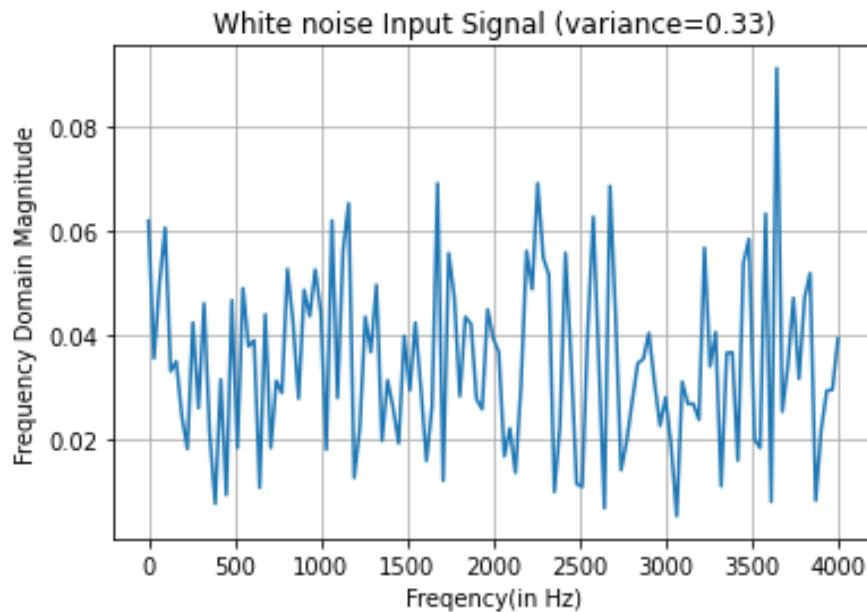
In this signal we see a peak corresponding to the input frequency, since the sine value is of 1800Hz frequency



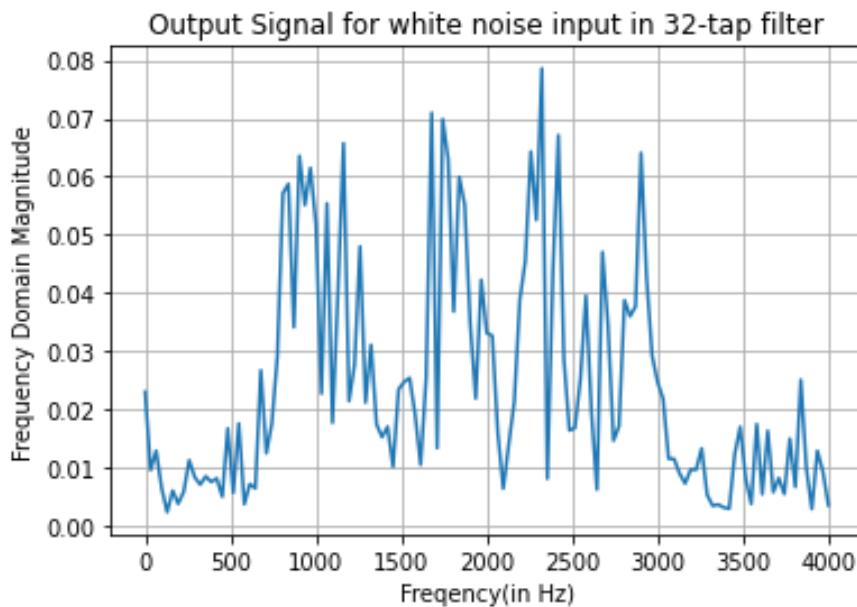
Even in the output we see the same variation, that is the peak at 1800Hz frequency, except that there is a scaling of the output, since we multiply the input signal with the coefficients of the filter.

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.

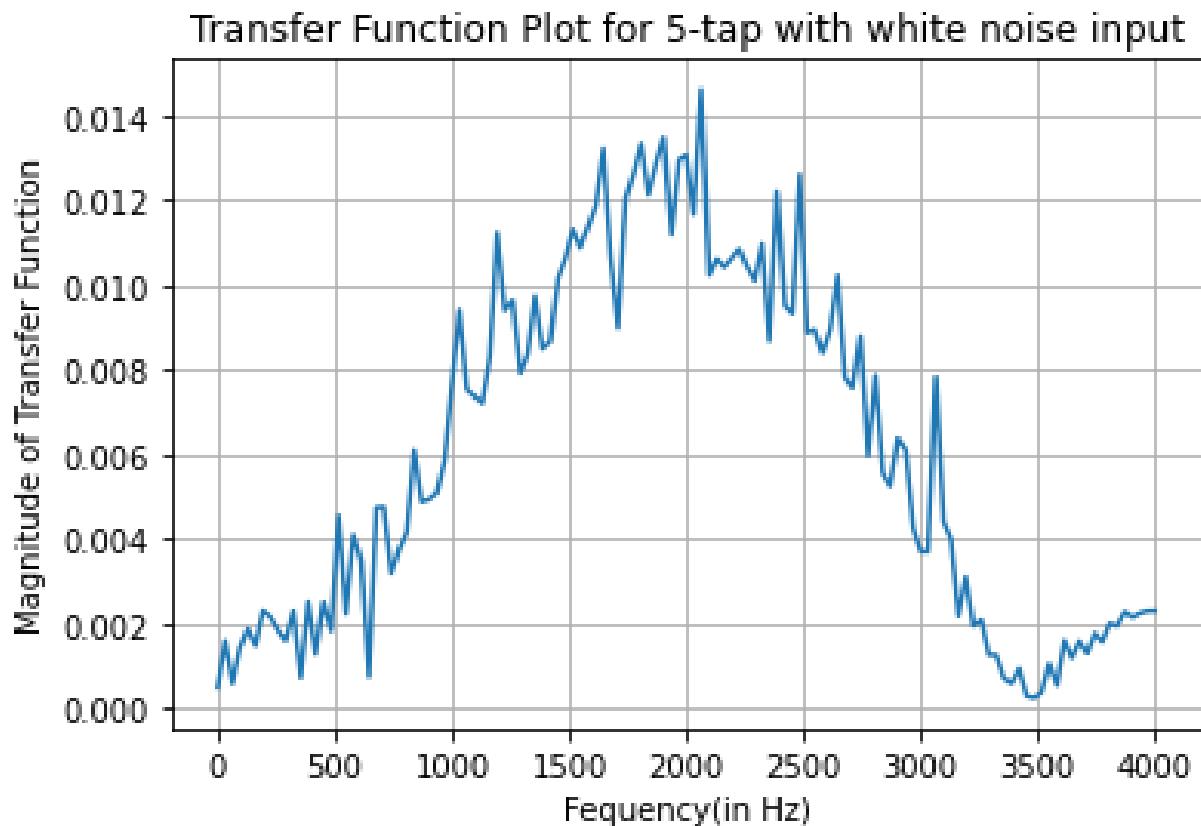


Here in the output plot we see that only the frequencies in the passband range of 800Hz to 3400Hz are passed through completely and the other frequencies outside it are attenuated strongly.

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.