Implement a FIR filter co-processor in a simulation environment

Karan Kabbur Hanumanthappa Manjunatha(1236383) Ali Bavarchee(1219425) Edoardo Antonaci(1234431)

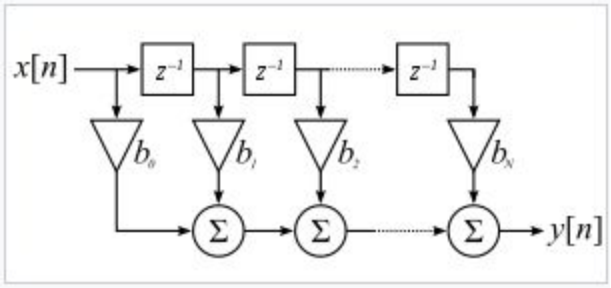
Introduction:

We design a FIR filter both VHDL and in python and compare the results from the two implementations (i.e. Frequency analysis). The FIR filter for input sine wave, square wave and audio wave file has 5 taps whereas for the ECG data, it was implemented with 11 taps. All of them are of low pass type. Adding to the FIR filter, DPRAM has also been implemented as an array as a simulation. The FIR filter has been interfaced (which typically has a serial interface, the input data has to be loaded one sample per clock cycle), with the blockram. So, a dedicated state machine able to perform read and write operations has been implemented. In the case of implementing fir filter using python, the scipy library has a method firwin that helps to calculate the coefficients related to a specified frequency behavior.

Theory and concepts:

For a causal discrete time FIR filter of order *N*, each value of the output sequence is a weighted sum of the most recent input values**:**

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



where:

- x [n] is the input signal,
- y [n] is the output signal,
- N is the filter order; an Nth-order filter has (N + 1) terms on the right-hand side
- b_i is the value of the impulse response/coefficient for $0 \leq i \leq N$ of an Nth-order FIR filter.

The coefficients to the FIR filter are obtained using scipy firwin package of python by giving the inputs which are number of taps as 5 and cut-off frequency as 0.1. The coefficients are in floating points and so to work with them in vivado, we multiply by 1000 and truncate all the remaining floating point numbers and so convert them into integers. These integers are easily

converted to hexadecimals format and used as coefficients in vivado. Note that after doing this step we have to rescale the results data obtained in the output text file by dividing them by 1000.

Coefficients	Floating point	Integer format	Hex format
b0	0.193353	193	0xc1
b1	0.203304	203	0xcb
b2	0.206687	206	0xce
b3	0.203304	203	0xcb
b4	0.193353	193	0xc1

The frequency analysis of this 5 tap FIR filter with 0.1 Hz cutoff frequency and we get the following plots as shown below:

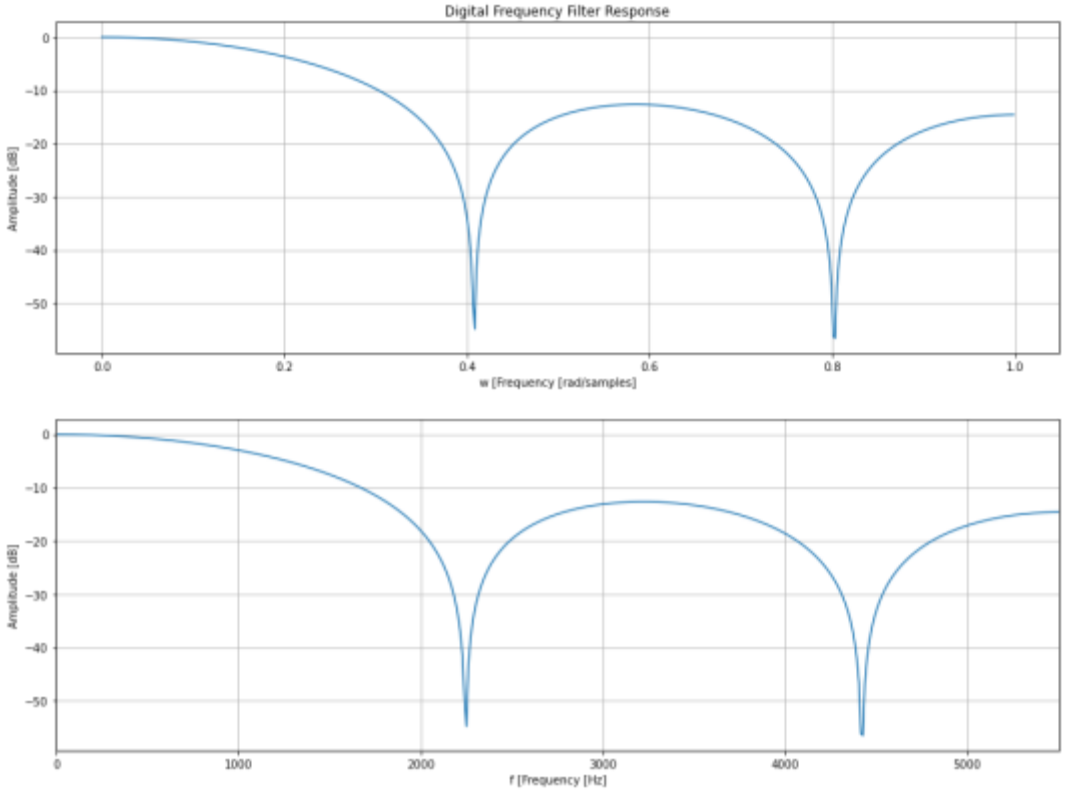


Figure 1: Digital frequency filter response for 5 tap filter and 0.1Hz cutoff frequency

A schematic image Figure 2, related to our project is represented below and in synthesis and test bench we have the following steps:

-read the input data from text file

-write the data into DPRAM represented as an array in vhdl code from files

-read the data from the memory of DPRAM and pass them into FIR filter.

-write the “filtered” data from filter into output text file.

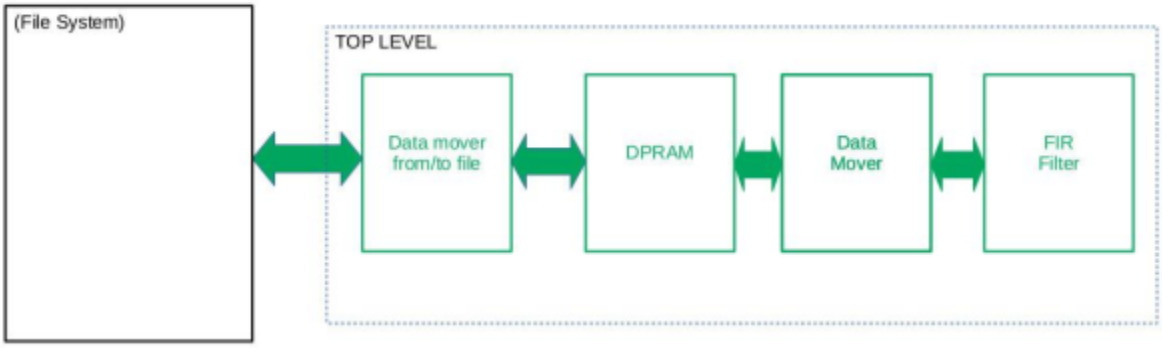


Figure 2: Implementation of simulation project in VHDL.

Methods and code:

VHDL design code:

- **Import library:** First we import the required library i.e. IEEE and use the STD_LOGIC_1164.all, NUMERIC_STD.all and Std_logic_signed.all module.
- **Entity “FIR_RI”:** The width of the input data and of the coefficients is taken as 32 each. So, the output width which is sum of those two widths is simply 64. Clk and rclk are the two clocks used for our simulated model and DPRAM respectively. Din is the data which is given as input to FIR filter and Dout is the output of data from FIR filter. ‘we’ is called write enable which is used in DPRAM. When we=’0’ we read from the DPRAM otherwise we write into it.

```
entity FIR_RI is
generic (
          input_width      : integer:=32;-- set input width by user
          output_width     : integer:=64;-- set output width by user
          coef_width       : integer:=32;-- set coefficient width by user
          tap             : integer :=5);-- set filter order

port(
     Din      : in std_logic_vector(input_width-1 downto 0);-- input data
     Clk      : in std_logic                               ;-- input clk
     Dout     : out std_logic_vector(output_width-1 downto 0); --output data from FIR filter
     rclk: in std_logic;
     we: in std_logic);--write enable

end FIR_RI;
```

- **Architecture of FIR_RI:** Here we define the coefficients for our 5 tap filter as 32 bit where C1, CB, CE in hexadecimals corresponds to integers 193, 203, 206. We have also created 3 arrays where shift_reg_type is of dimension 5x32 while mult_type and ADD_type are of dimensions 5x64. These arrays are used for the FIR filter calculations.

```
type Coeficient_type is array (0 to tap-1) of std_logic_vector(coef_width-1 downto 0);
----------------------------------FIR filter coefficients-----------------------------------
constant coeficient: coeficient_type :=
                              (     X"000000C1",  --193
                                    X"000000CB",  --203
                                    X"000000CE", --206
                                    X"000000CB",  --203
                                    X"000000C1"  --193
                                    );
-------------------------------------------------------------------------------------------------
type shift_reg_type is array (0 to tap-1) of std_logic_vector(input_width-1 downto 0);
signal shift_reg : shift_reg_type;
type mult_type is array (0 to tap-1) of std_logic_vector(input_width+coef_width-1 downto 0);
signal mult : mult_type;
type ADD_type is array (0 to tap-1) of std_logic_vector(input_width+coef_width-1 downto 0);
signal ADD: ADD_type;
signal filter_input :      std_logic_vector(31 downto 0)     ;
```

The working of FIR filter has been implemented with the following code where we make use of N-bit register to keep track of multiplying the coefficients with the input data in accordance with FIR filter formula as described earlier. 'For' loop has been implemented for filter multiplication and addition. Dout represents the output of FIR filter while Din is the input to the FIR filter.

```
begin

          shift_reg(0) <= Din;
          mult(0) <= Din*coeficient(tap-1);
          ADD(0) <= Din*coeficient(tap-1);

          GEN_FIR:
          for i in 0 to tap-2 generate
          begin
                -- N-bit reg unit
                N_bit_Reg_unit : N_bit_Reg generic map (input_width => 32)
                port map ( Clk => Clk,
                                  --reset => reset,
                                  D => shift_reg(i),
                                  Q => shift_reg(i+1)
                                  );
               -- filter multiplication
               mult(i+1)<= shift_reg(i+1)*coeficient(tap-1-i-1);
               -- filter combinational addition
               ADD(i+1)<=ADD(i)+mult(i+1);
          end generate GEN_FIR;
          Dout <= ADD(tap-1);

end behavioral;
```

Our FIR filter makes use of N bit register to store the most recent input values since the formula is based on a weighted sum of the most recent input values. So, at every rising edge of the clock pulse Q takes the value of D and is used in the FIR filter code as shown above.

VHDL Testbench code:

In the testbench we import again the library IEEE and also the use the required modules from the library same as in the previous VHDL design code. We also use the STD.TEXTIO.all module to read and write into text files. The entity cell should be left empty always while coding the testbench.

Architecture "TB_FIR": The architecture of testbench consists of components whose contents are exactly the same as the entity of the design code described earlier.

- **Signals:** The signals described below are used for our simulation. Most of the signals are as explained earlier. 'd' is data signal and 'q' is the data which is written into our ram array. We simulate DPRAM by creating an array of size 32 with 0 to 60000 rows, and then we call it simply as 'ram'. Finally, we create my_input which is the square wave data read from a text file while we create 'my_output' to store the FIR filtered data into another text file.

```
Library IEEE;
USE IEEE.Std_logic_1164.all;

-- N-bit Register in VHDL used for FIR filter calculation
entity N_bit_Reg is
generic (
          input_width : integer     :=32
          );
  port(
   Q : out std_logic_vector(input_width-1 downto 0);
   Clk :in std_logic;
   D :in std_logic_vector(input_width-1 downto 0)
  );
end N_bit_Reg;

architecture Behavioral of N_bit_Reg is
begin
     process(Clk)
     begin
       if ( rising_edge(Clk) ) then
                Q <= D;
      end if;
     end process;
end Behavioral;
```

```
signal Din       :      std_logic_vector(31 downto 0)     ;
signal Clk       :      std_logic:='0'                              ;
signal output_ready      :      std_logic:='0';
signal Dout      :      std_logic_vector(63 downto 0)     ;

signal rclk:  std_logic:='0';
signal we:  std_logic := '1';
signal d:  std_logic_vector(31 downto 0) := (others => '0');
signal q:  std_logic_vector(31 downto 0);

-- here we simulate dpram by creating an array where index is used as address for it
type ram_array is array( 0 to 60000) of std_logic_vector(31 downto 0);
shared variable ram: ram_array;

file my_input : TEXT open READ_MODE is "/home/karan/Desktop/Physics of Data/MAPD Mod A/fir_simulation_karan/input_sq.txt";
file my_output : TEXT open WRITE_MODE is "/home/karan/Desktop/Physics of Data/MAPD Mod A/fir_simulation_karan/output_sq2.tx
```

- Next we provide a clk and rclk which are clock pulse each of which are 10ns high and 10ns low. And then we also provide ‘we’(write enable) signal with 20ns high and 20ns low. Next we create a process where on the rising edge of rclk and when we=’1’ we read a line from the input text file and then write into the dpram known as ram. Along with that we increment the index where the index represents the location of the ram into which data is written. ‘q’ signal gives us data written into that particular address of the ram.

```
        --clk is 10ns high and 10 ns low
        process(clk)
        begin
        Clk          <= not Clk after 10 ns;
        end process;

        --rclk is 10ns high and 10ns low
        process(rclk)
        begin
        rclk  <= not rclk after 10 ns;
        end process;

        --we is 20 ns high and 20 ns low
        we <= not we after 20ns ; --we is already 1
```

```
-- Here we write into the dpram
process(rclk)
variable my_input_line : LINE;
variable input1: integer;
variable index : integer range 0 to 60000;

begin
    if rising_edge(rclk) then
      if (we='1') then -- when we = '1', we write into dpram
            readline(my_input, my_input_line);
            read(my_input_line,input1);
            d <= std_logic_vector(to_signed(input1, 32));
            ram(index) := d; --index represents address of simulated dpram which is array
            q <= ram(index); --q represents the data written into array at location 'index'
            index := index+1;
         end if;
      end if;
   end process;
```

- We create another process which reads the data from the ram array at every rising edge of rclk and when we=’0’. After reading from the ram(index) where index represents the address of the array, we input it to Din which is sent as an input to FIR filter. After this step, our output is ready to be written into the output text file. We read the next data from the ram in a similar way by incrementing the ‘index’.

```
-- Here we read from DPRAM and send the data to FIR filter
process(rclk)
 variable my_input_line : LINE;
 variable my_output_line : LINE;
 variable index : integer range 0 to 60000;

  begin
     if rising_edge(rclk) then --read from DPRAM to FIR
       if (we='0') then -- when we='0' we read the data from ram array and send it to FIR
        Din <= ram(index); --via Din
        output_ready <= '1';
        index := index+1;
       end if;
     end if;
     --end loop;
 end process;
```

- Finally, we end our testbench with a final process where at every falling edge of ‘clk’ pulse and whenever the output_ready=’1’, we write the Dout i.e. the output data from the FIR filter into the output text file. Note that our whole simulation is run only for 20250ns by changing the simulation run time in the ‘Settings’ of Vivado.

```
 -- Here we obtain the output from FIR filter and write it into output file.
 process(clk)
 variable my_output_line : LINE;
-- variable input1: integer;
 begin
      if falling_edge(clk) then
           if output_ready='1' then
                write(my_output_line, to_integer(signed(Dout)));--output of FIR filter
                writeline(my_output,my_output_line);
           end if;
      end if;
 end process;
```

Results:

Generally, to evaluate the design, one safe and easy way is using testbench. By running the simulation of design, the bugs, faults and errors outcrop. Apart from that, this exercise meant to simulate FIR fIlter communicating with dual-port RAM and PC respectively. The output of

testbench of VHDL code could be compared with output of simulation which has been executed by python.

Here is a function defined in python to implement fir filter. ‘x’ represents the input data array and b represents an array of coefficients.

```
# Implementing the above formula of FIR filter
def fir_filter(x,b):
    filtered_wv = []
    for i in range(len(b)-1, len(x)):
        y = 0
        for j in range(len(b)):
            y += b[j]*x[i-j]
        filtered_wv.append(y)
    return(filtered_wv)
```

Square Wave as input: Square wave data was generated in python and stored in a text file. Next, the same data was given as input to vhdl as well as to fir_filter function as described above. From the Fig.3 we notice that the output filtered data for square wave for both vhdl implementation as well as python implementation exactly matches. However, the data is scaled since our coefficients are scaled by 1000 as explained earlier. To rescale the data, we simply divide it by 1000 and so we get the filtered data in the range 0 to 5 instead of 0 to 5000.

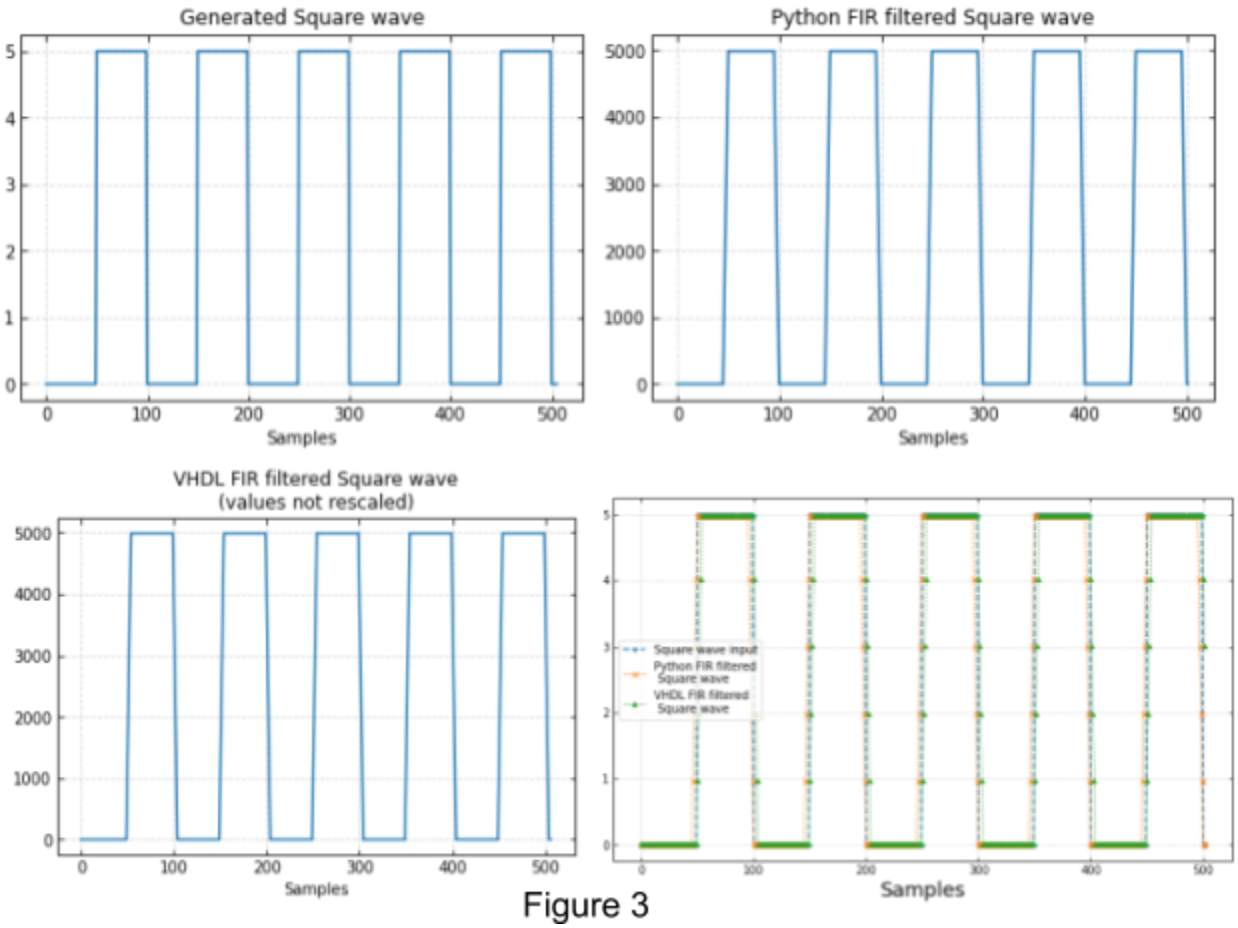


Figure 3

Sine wave as input: We have similarly generated sine wave data using python and stored them in text file. After FIR filtering using both vhdl and python code we get the same results and shows that FIR filtered data using vhdl is correct indeed. Note that the output filtered data are scaled by 1000.

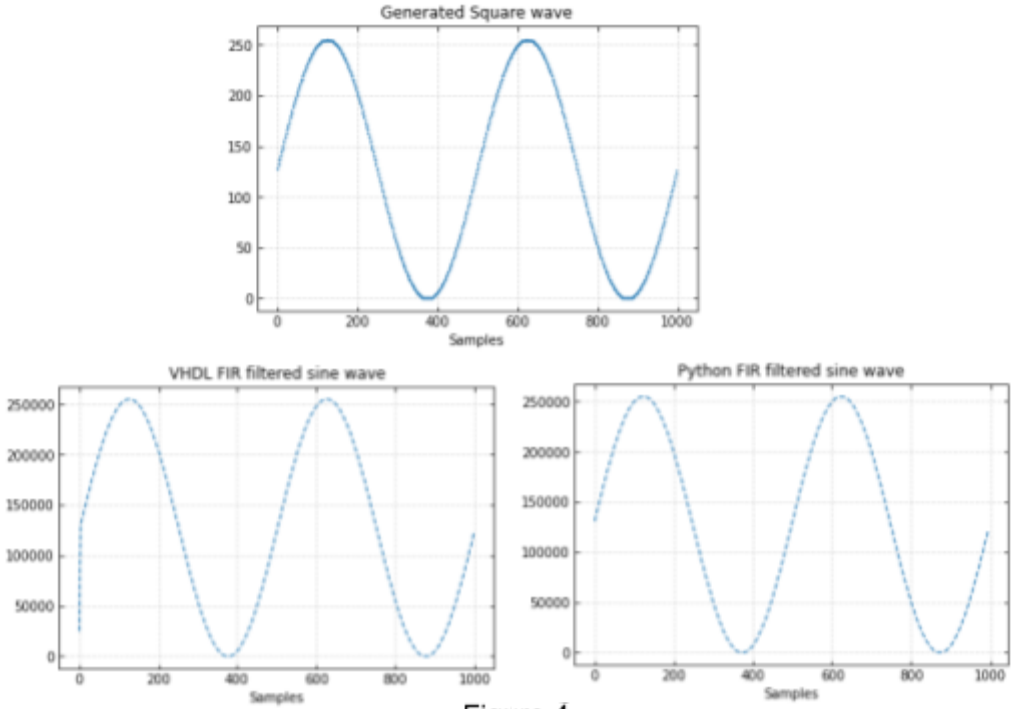


Figure 4

ECG data as input: We obtained ECG data from online website and using VHDL we have performed FIR filter with 11 taps. The advantage of using FIR filter on the ECG data is to

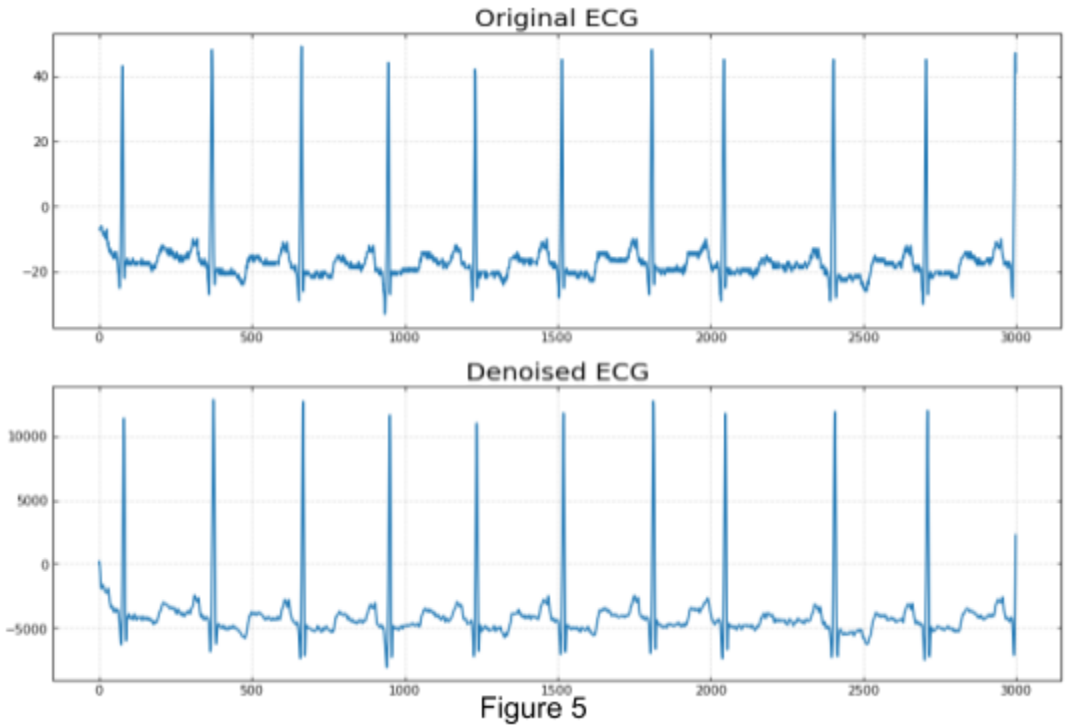


Figure 5

denoise the data and finally obtaining a clear signal. The coefficients used for this filtering are [F1, F3, 07, 26, 42, 4E, 42, 26, 07, F3, F1].

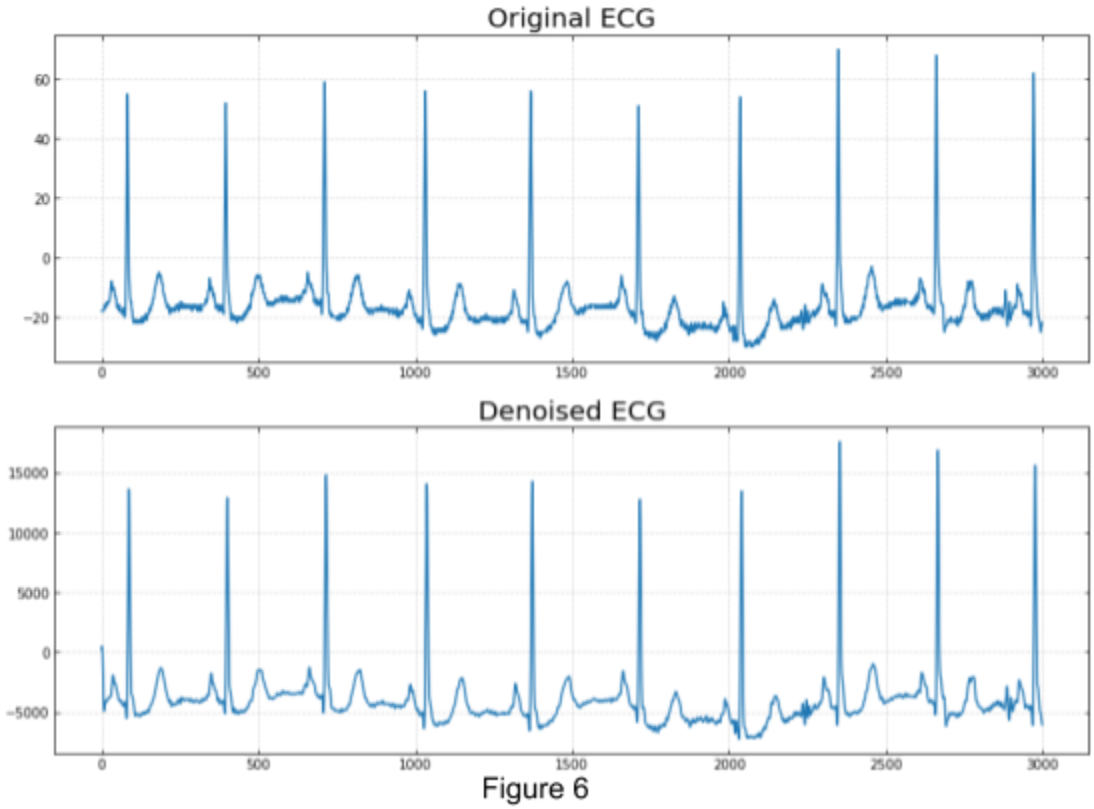


Figure 6

FIR filter on audio files: Audio file named "FrequencyTest.wav" which was given to us has been converted to text file using wavfile.read() function of python. This text file again with the help of VHDL was FIR filtered with 5 taps and 0.1 Hz cut-off frequency. The filtered data present in output text file has to be rescaled by dividing it by 1000. This output text file was again reconverted back to wav file using wavfile.write(). The input text file and the rescaled filtered output text file were used to plot time domain graphs. For comparison, we have also filtered the audio input file using python code also.

Since we have a low pass filter, it passes signals with a frequency lower than a selected cutoff frequency which in our case is 0.1Hz and attenuates signals with frequencies higher than the cutoff frequency. We observe from the below plotted images that VHDL FIR filtered plot and python FIR filtered plot are the same. These plots represent amplitudes of signals in time domain. In addition to time domain amplitude plots, we have also plotted the frequency spectrum of the audio files. The x axis represents frequency in log scale while the y axis represents the FFT of the input and output signal data for the two plots.

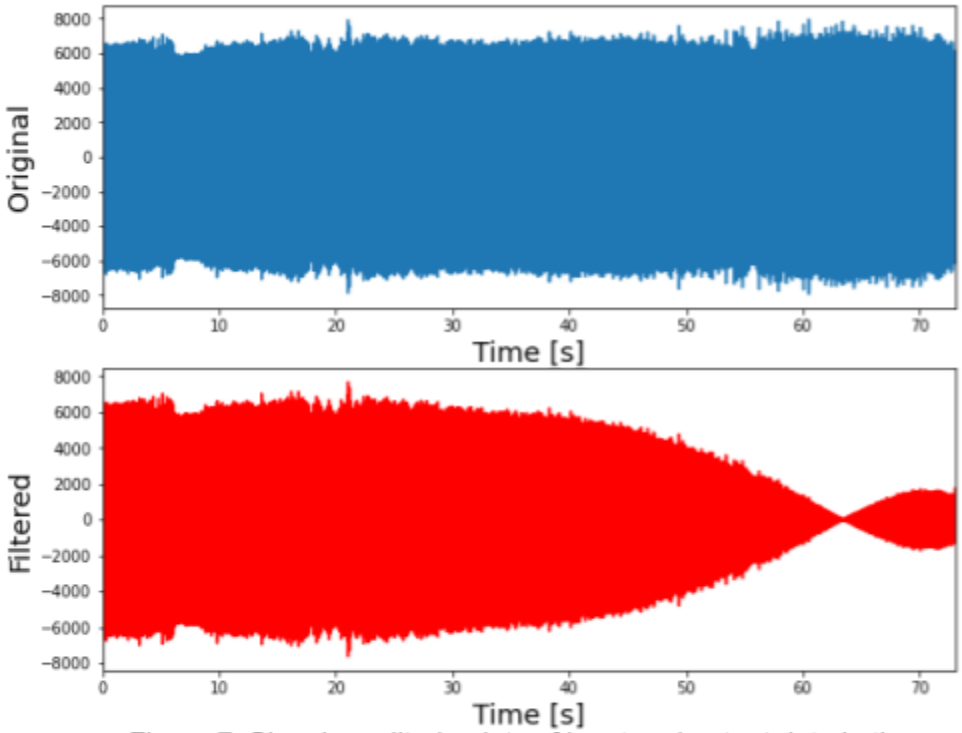


Figure 7: Signal amplitude plots of input and output data in time domain.

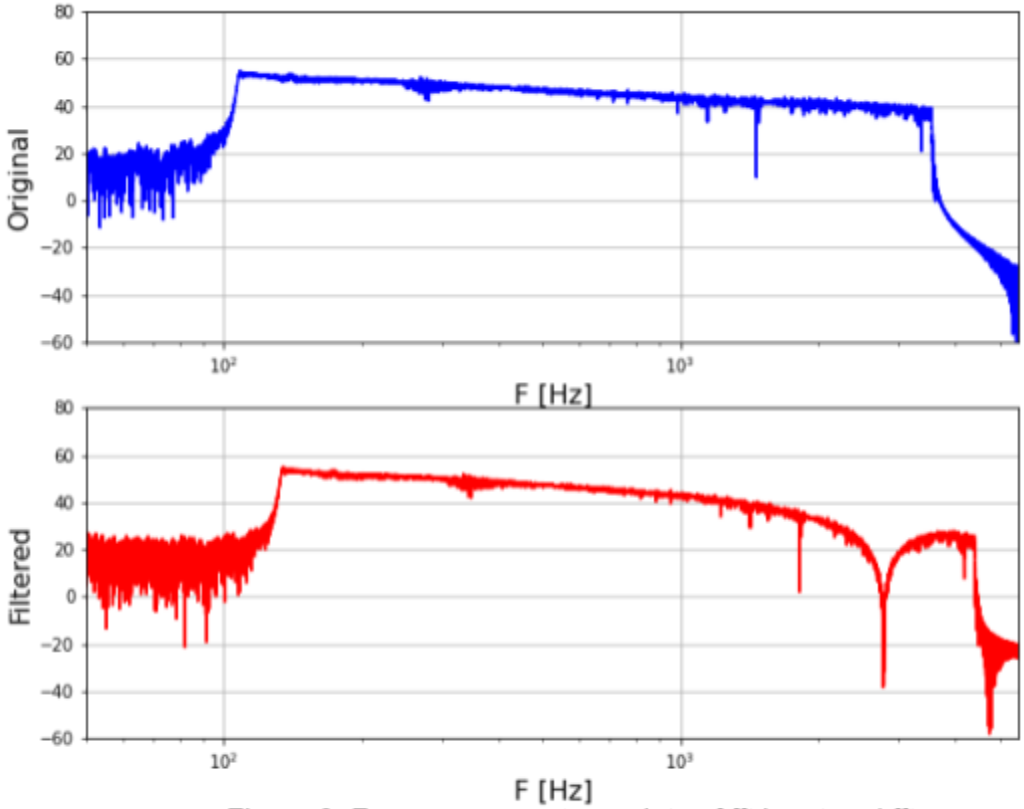


Figure 8: Frequency spectrum plots of fft input and fft filtered signals.

In Figure 8, we observe in the Filtered plot on the right hand side, there is a huge drop in the FFT value of the output signal data for frequency greater than 1000Hz.

In addition to the above, we have also performed FIR filter with 5 taps and 0.1Hz cutoff frequency on music “Still Dr. Dre” with chosen time length of 25 seconds. After converting the wav file to text file using wavfile.read(), we obtain a text file containing 2d array which has two columns. The left column represents left stereo speaker and the right column represents right stereo speaker. Each of the two columns were separately FIR filtered using VHDL and then re-combined to get matrix of two columns. These again with the help of wavfile.write() were reconverted back to audio wav file. In the audio wave file, it was clearly heard that all the high frequencies were removed in other words they were filtered out.

The amplitudes of both the input and output filtered signals were plotted in time domain, this again was done for both left speaker as well as right speaker signal data.

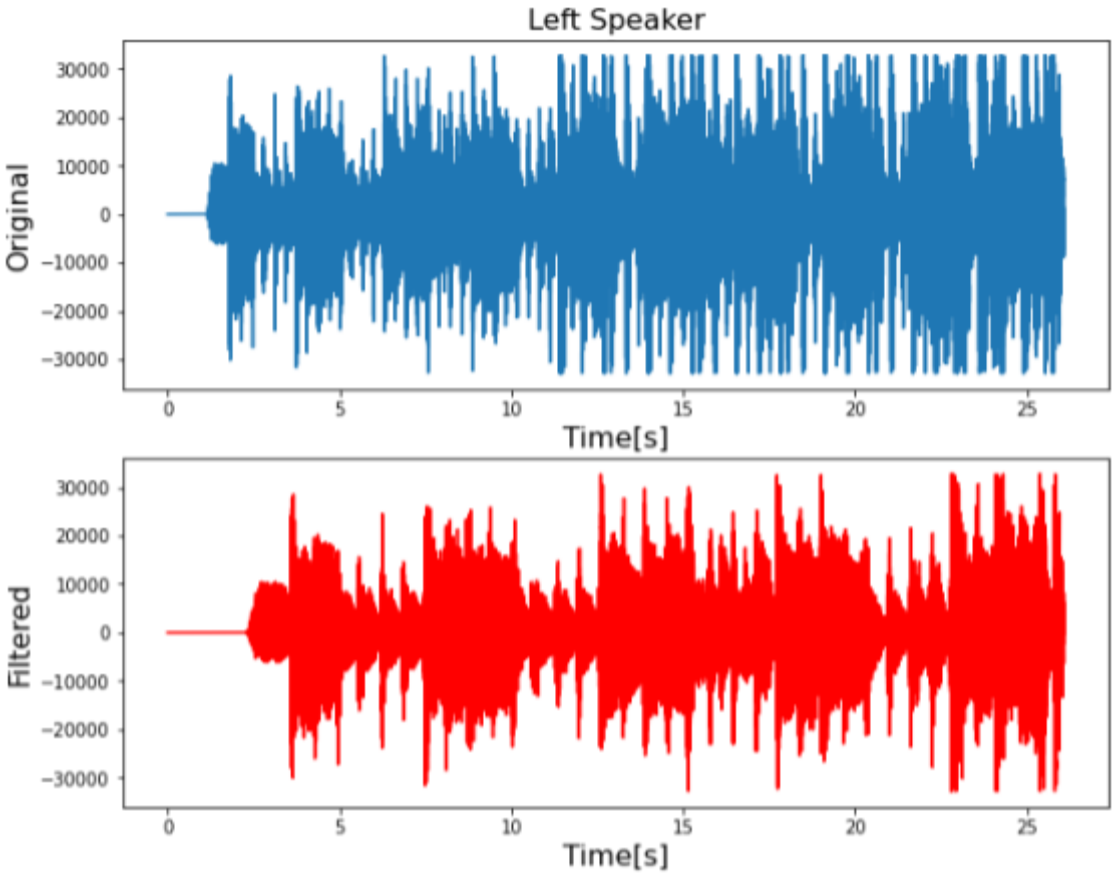


Figure 9: Input and output filtered amplitude signals in time domain for the Left Stereo.

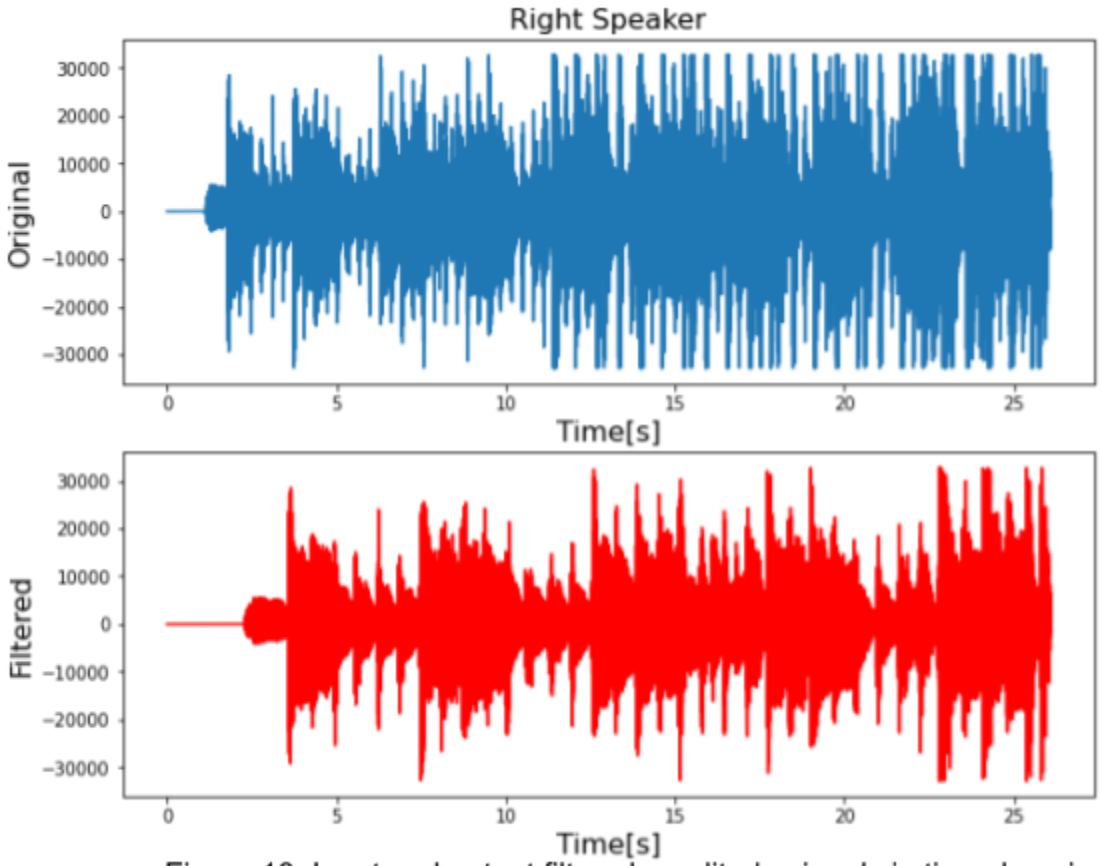


Figure 10: Input and output filtered amplitude signals in time domain for the Right Stereo.

Conclusion: We have successfully implemented DPRAM as an array in VHDL code interfacing it with FIR filter of low pass type. 5 tap FIR filter with cutoff frequency 0.1Hz has been used to filter the input sine wave and square wave. The obtained filtered output were same as the result that was obtained using Python language. Using 11 tap FIR filter, denoising of ECG data were performed through VHDL. Also, the audio files which were in .wav format were converted to text files and then given as input signal to the FIR filter implemented in VHDL. Again the output filtered data which was obtained were the same as that of Python. This implies that our DPRAM and FIR implementation are correctly working.