NOISE CANCELLATION USING WINDOWING TECHNIQUE

A Project Report Presented to
The faculty of the Department of Electrical Engineering
San José State University

In Partial Fulfillment of the Requirements for the Degree
Master of Science

By

Lekhya Tata

SJSU ID# 012493598

EE 297B Section # 11 , Spring 2019
lekhya.tata@sjsu.edu

Phone # 669-294-5476

Karthik Govinda Raju

SJSU ID# 012462294

EE 297B Section # 11 , Spring 2019

karthik.govindaraju@sjsu.edu

Phone # 408-520-8649

Department Approval

Lili He
Project Advisor : DocuSigned by: Lili He D488DCB476DD4AA... Date: *5/15/2019*
lili.he@sjsu.edu

Morris Jones
Project Co-Advisor : DocuSigned by: Morris Jones 251AA457416D415... Date: *5/16/2019*
morris.jones@sjsu.edu

Birsen Sirkeci
Graduate Coordinator : DocuSigned by: Birsen Sirkeci 3DE1954E6EDB405... Date: *5/16/2019*
birsen.sirkeci@sjsu.edu

Department of Electrical Engineering
Charles W. Davidson College of Engineering
San José State University
San Jose, CA 95192-0084

Upload Files

Project Report	Turn-it-in report	Project Poster
		

Submission Date: 5/14/2019

Lekhya Tata

5/14/2019

DocuSigned by:
Lekhya Tata
EB8F74F1CF4044C...

Karthik Govinda Raju

5/15/2019

DocuSigned by:
Karthik Govinda Raju
2B6E9E7EEE7749F...

NOISE CANCELLATION USING WINDOWING TECHNIQUE

by

Lekhya Tata

Karthik Govinda Raju

Abstract

Motivation:

Many methods have been implemented for noise cancellation in audio signals. Mostly in software like MATLAB. Filters that are already in Verilog are confined to only small class of noise removal and no study proves that each method is completely efficient.

Tasks:

Project aim is to filter audio signal with noise in MATLAB and get the results. By taking MATLAB results as a reference, filter is designed in System Verilog for the audio file with constrains that are used in MATLAB code. Compare both the results. To get faster results and reduce hardware mathematical computations in filter are improved. FFT, windowing and IFFT are used to filter noise signal.

Significance:

There is no formal study that shows which transforms method works better, that are employed in both speed and accuracy. The results of the design in this project are much cleaner than the MATLAB result

Table of Contents

List of figures

List of tables:

1. Introduction

ASIC chips are leading in the industry of semiconductor technology because they are working at a quicker rate and as a result for better operation and task involving great technology, design is becoming more complex. It includes many challenges and trade-offs.

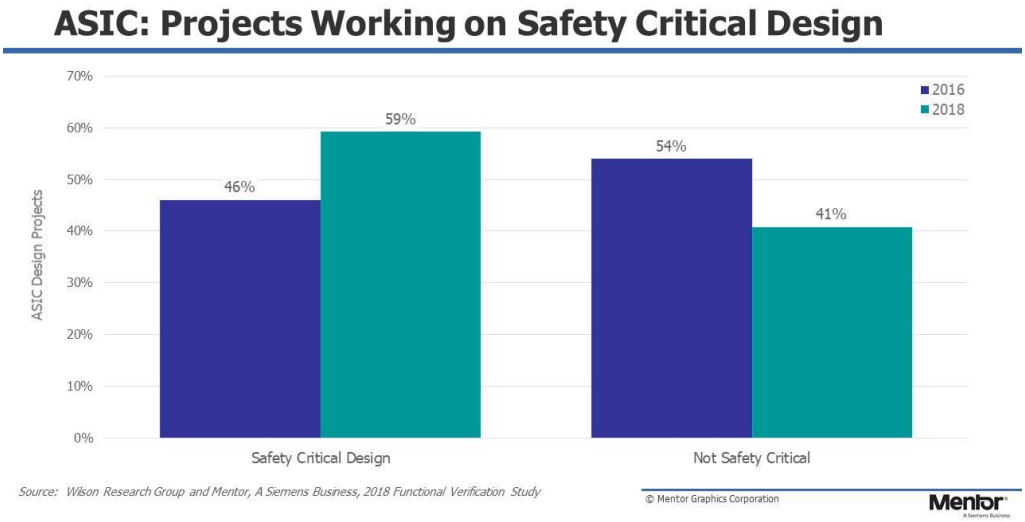


Figure 1. Importance of ASIC projects [1]

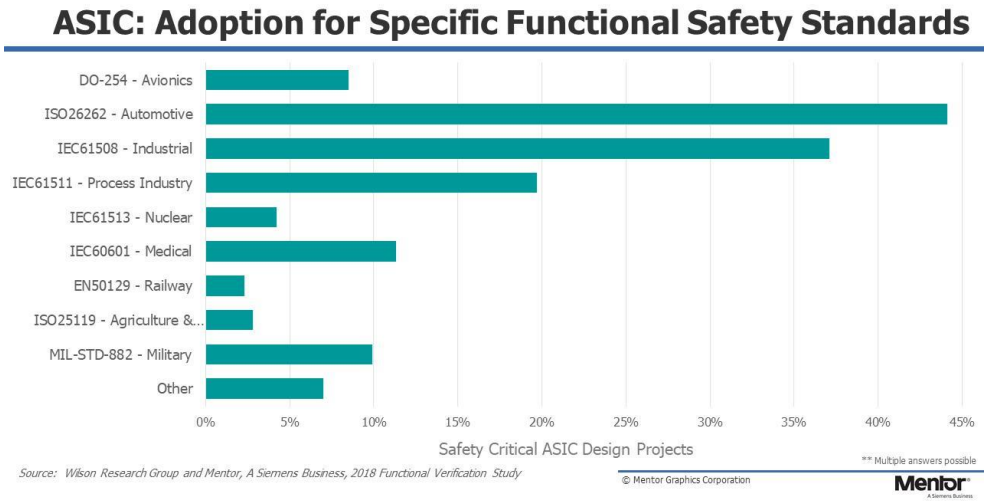


Figure 2. Role of ASIC adoption in various industries [1]

From the figure 2, it is clear, that ASIC is leading in various industries. Medical is the fourth leading industry. This project involves noise removal in the ultrasound of fetal heartbeat.

Aim of the project is to design a filter that performs better and reduce maximum noise in a faster way. Generally, noise can be eliminated in time domain and frequency domain. In this project design is made for frequency domain as speech and noise signals may be better separated in that space, which enables better filter estimation and noise reduction performance. Noise can be categorized in many ways. Additive noise (white noise) is selected for filtering.

When the audio signal with noise is been fed to MATLAB and cut off frequency of noise signal is known from magnitude spectrum plot. This cutoff frequency is used to calculate digital cut off using sampling frequency and analog cut off frequency.

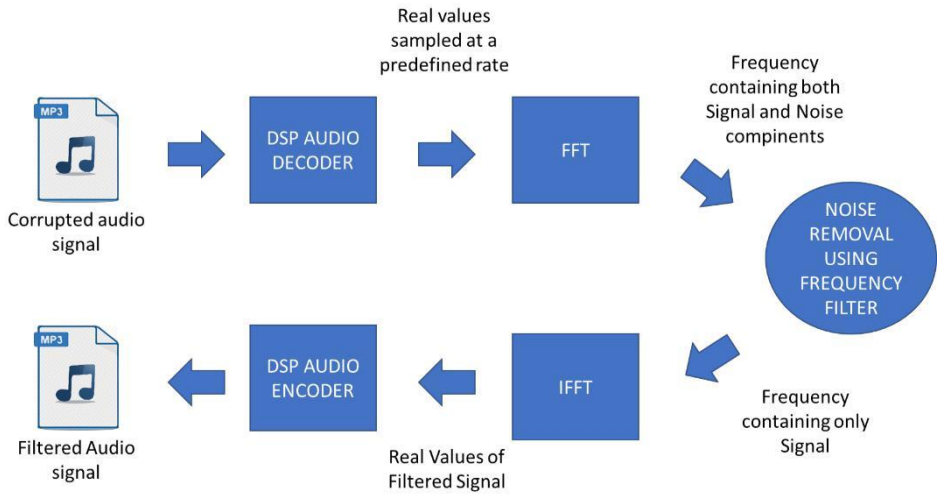


Figure 3. Flow of the project

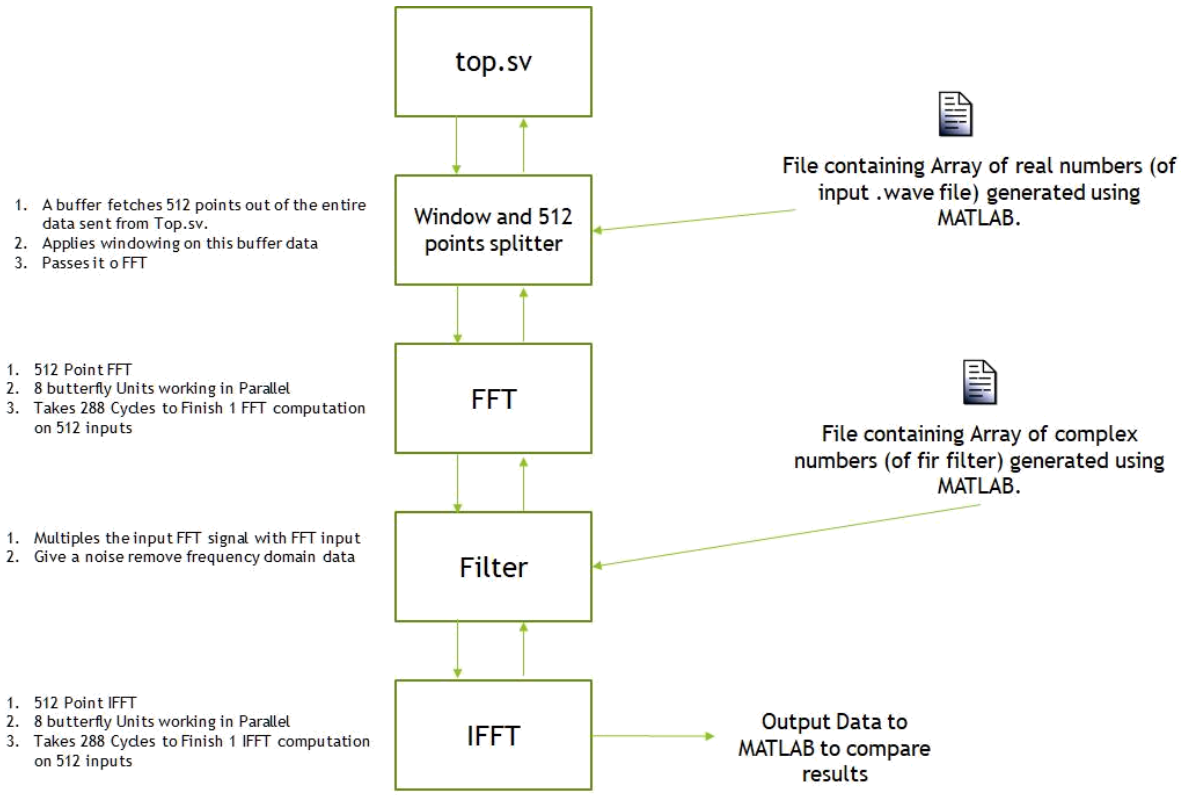


Figure 4. Flow of the project in System Verilog

2. Noise

Noise comes in all shapes, sizes and sounds. This is project concentrates on noise in the form of sound. Sound wave can be broken down to two fundamental characteristics, frequency and amplitude Any unwanted sound that is added to the original audio file is considered as a sound noise. During audio recording, noise gets included in Analog tapes or digital recordings. Among many files section of "hiss" noise found throughout the recording. Addition of noise into the recording also depends on the environment of recording and equipment. Below is the list of four major kinds of audio noises.

2.1. Hum

The sound which is at low frequency than the actual sound and whose frequency ranges between 40 – 80hz [2] is considered as hum. For example, whirring of low-pitched motor. Hum usually occurs by electrical interferences or the ground wiring of recording equipment is not done properly.

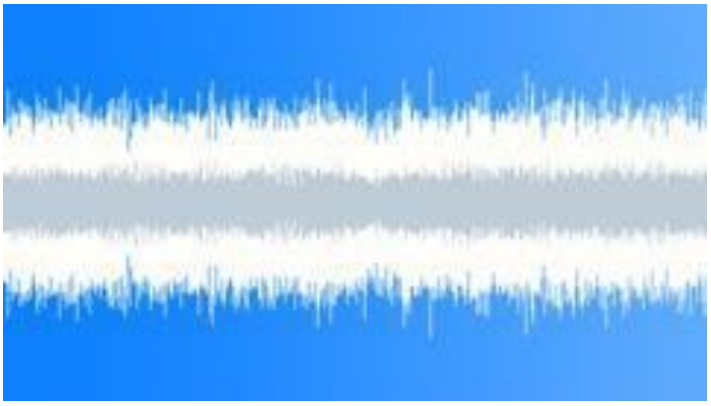


Figure 5. Low frequency hum noise Hiss

This can be seen in devices that includes electronic components. Due to rise in temperature (compared to room temperature) path of electrons in the device is deviated as a results output is disturbed. This disturbance in the distorted output is considered as “hiss”. Hiss noise depends on the quality of recording equipment. This noise can also be a result of environmental factors like A/C, fans etc.,

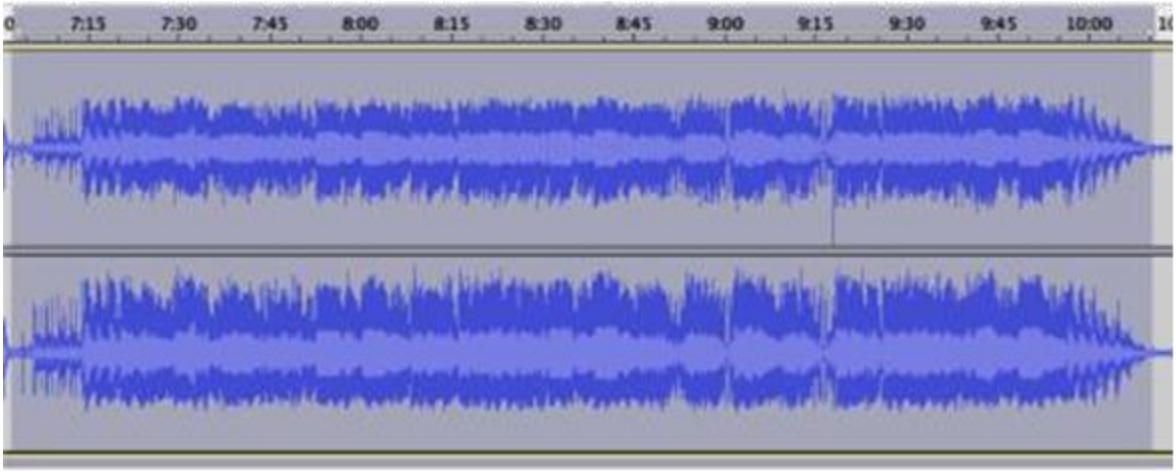


Figure 6. Hiss noise plot

2.2. Rumble

This is very common audio noise. Like hum noise this is also at low frequency. This can be seen only between specific points in a given bandwidth. It is most observable noise in mechanical devices. For example, ball bearings at a joint, gear rotation in a gear box, mechanical parts moving in the audio recording equipment creates this noise. There are filters that are inbuilt in a device to eliminate this kind of noise.

Figure 7 Rumble noise

2.4. Crackle

These are kind of discontinuous and non-musical. This kind of noise is like the sound heard during burning of wood. This is caused by explosion of air pockets inside the object. These are of two types fine crackles which are high pitched and last for less duration of time, coarse crackles which are of low pitched and last for longer duration.

Noise can be categorized by giving color names. This project involves an audio file that contains “white noise”. White noise generally comes to hiss noise category. It is present everywhere in the entire audio file. To put entire concept in a nutshell, it is a layer of sound that is considered as noise floor while processing it.

Figure 8 Gramophone crackling noise

2.5. White noise

White noise can be stated as the noise that contains many frequencies. sometimes white noise is used to mask the other unwanted frequencies.

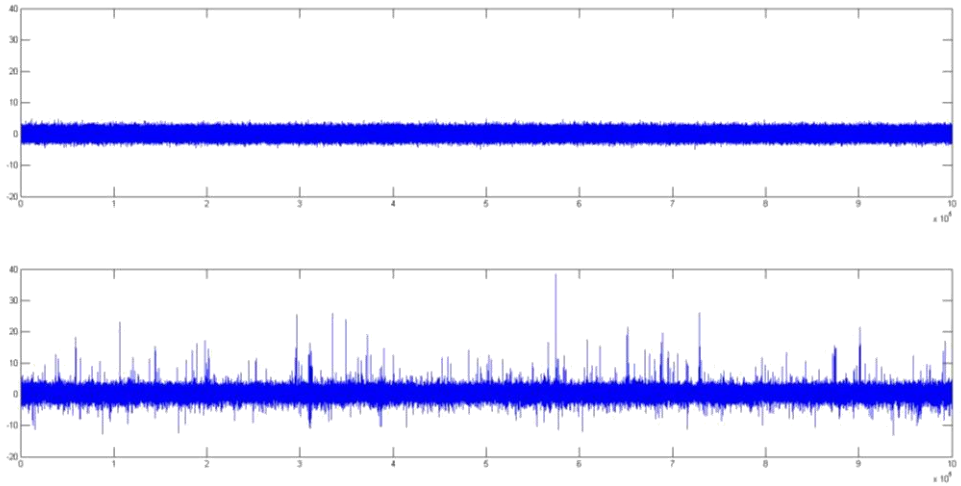


Figure 9. Plot showing white noise in an audio file

3. Filters

The main function of a filter is to pass desired frequencies and attenuate the unwanted frequencies. The four primary types of filters that can be listed down as

- Low-pass filter
- High-pass filter
- Band-pass filter
- Notch filter (band-stop filter)

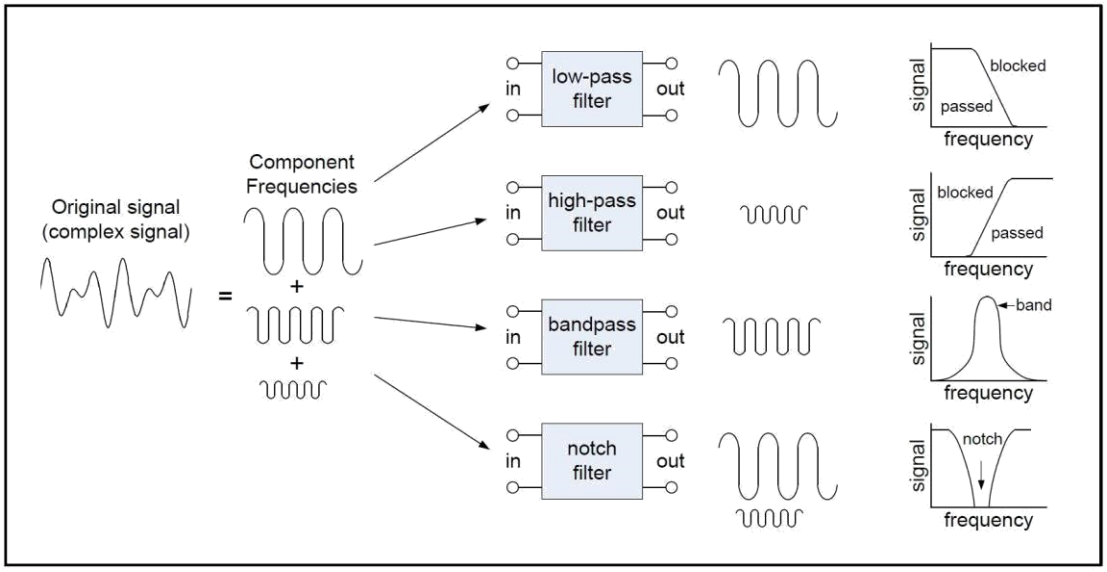


Figure 10. A basic depiction of four major filter types

There are two major digital filters. FIR filters and IIR filters.

3.1. FIR filter

The Fourier filter is a type of filtering function that is based on manipulation of specific frequency components of a signal. It works by taking the Fourier transform of the signal, then attenuating or amplifying specific frequencies, and finally inverse transforming the result. The main advantage of FIR filter is that they can easily be designed to be "linear phase". FIR filter works better with 2^N sample points. Stability of FIR filter is very good which can also be framed as finite output can be obtained for every finite input.

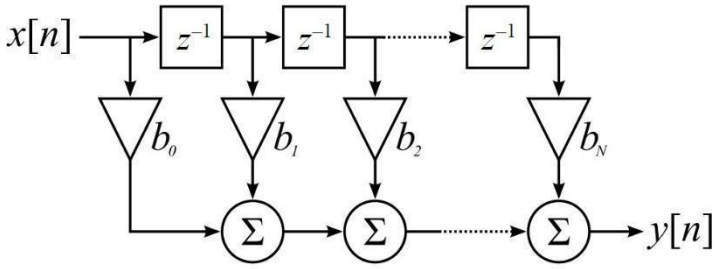


Figure 11. Butterworth filter representation [4]

$$y[n] = \sum_{k=0}^{N-1} h[k] \cdot x[n-k]$$

FIR filter formula [5]

3.2. IIR filter

Infinite impulse response filter. This filter is infinite because of presence of feedback in the filter. Desired filtering characteristics can be achieved using IIR filter consuming low memory and calculations when compared to an FIR filter. The main disadvantage is that implementation of filtering using fixed point arithmetic becomes slower and harder. They don't offer the computational advantages of FIR filters for multi rate applications.

FIR filters are preferable over IIR filters as the response is a linear phase and non-recursive. IIR filter performance is good when dealing with analog filter responses. This project uses FIR filter for the input wave file selected.

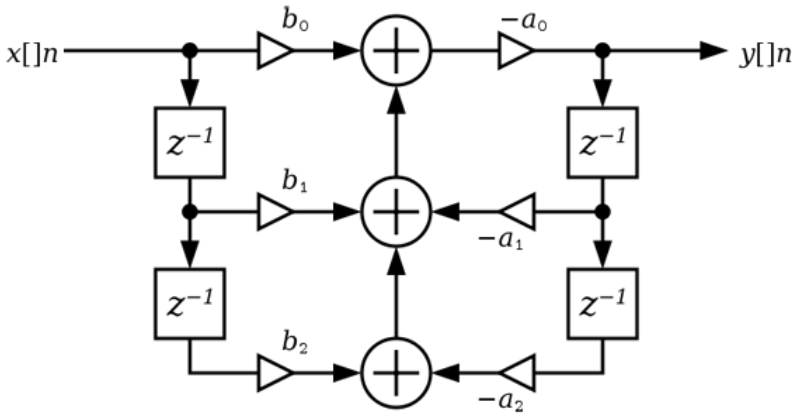


Figure 12. IIR Filter Z Transform [6]

4. Windowing

In most of the signal measurements, the signal doesn't contain integer number of periods. Due to this finiteness is not perfect and result in loss of some part of waveform and characteristics of whole signal changes. This finiteness introduces sharp transition changes in the signal that is measured.

When the waveform's period is non-integer, discontinuities are seen at the endpoints. These unwanted discontinuous components become high frequency harmonics in FFT which are not actually present in the original signal. The frequency of these unwanted components is higher than the Nyquist frequency value. Because of thse distortion the FFT spectrum is not the actual spectrum of actual signal. It appears as if there is a leakage of energy from one frequency component to other frequency component.

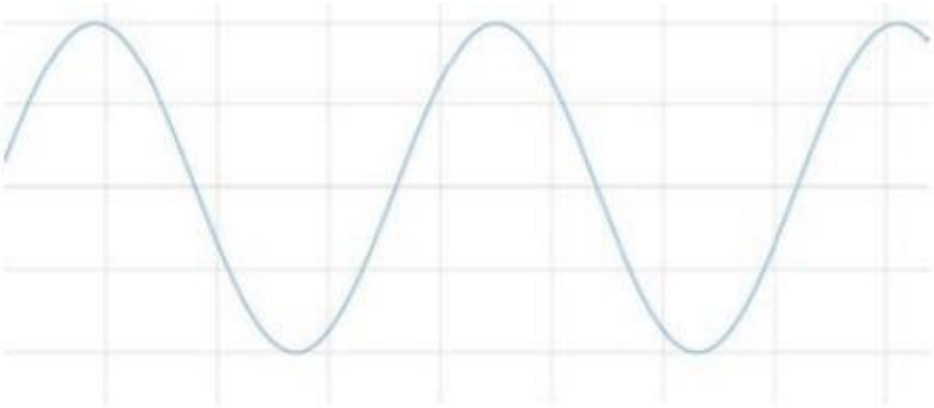


Figure 13. Signal with non-integer number of periods

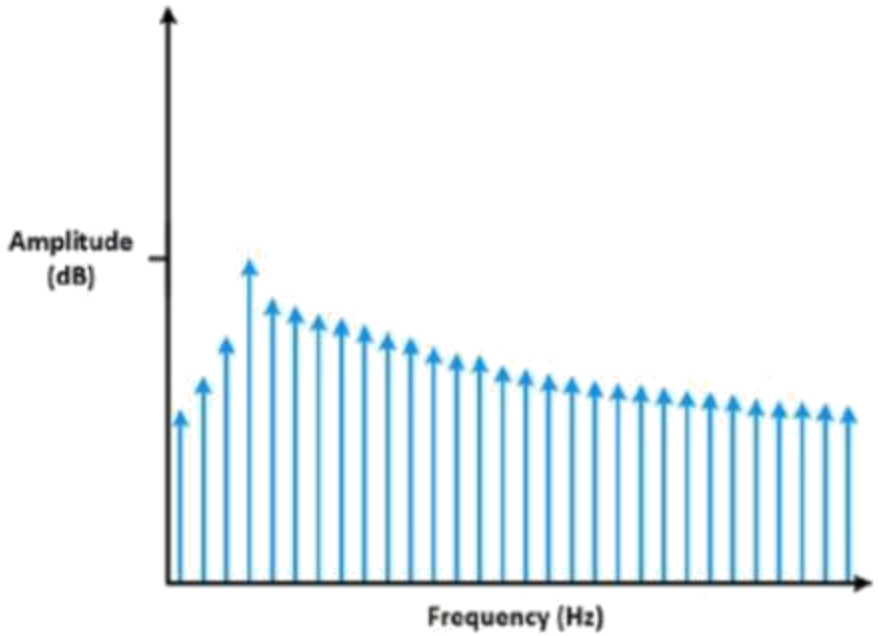


Figure 14. Spectral leakage to the FFT of signal in the figure

In a signal that contains non-integer periods, this distortion in FFT can be minimized by using a technique called “windowing”. Windowing basically reduce the amplitude of these discontinuities at starting and ending and gradually rolls of to zero of given boundary for a finite length.

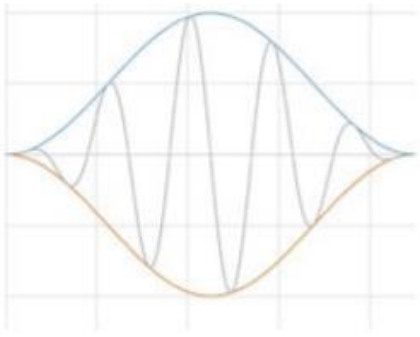


Figure 15. Result of windowing

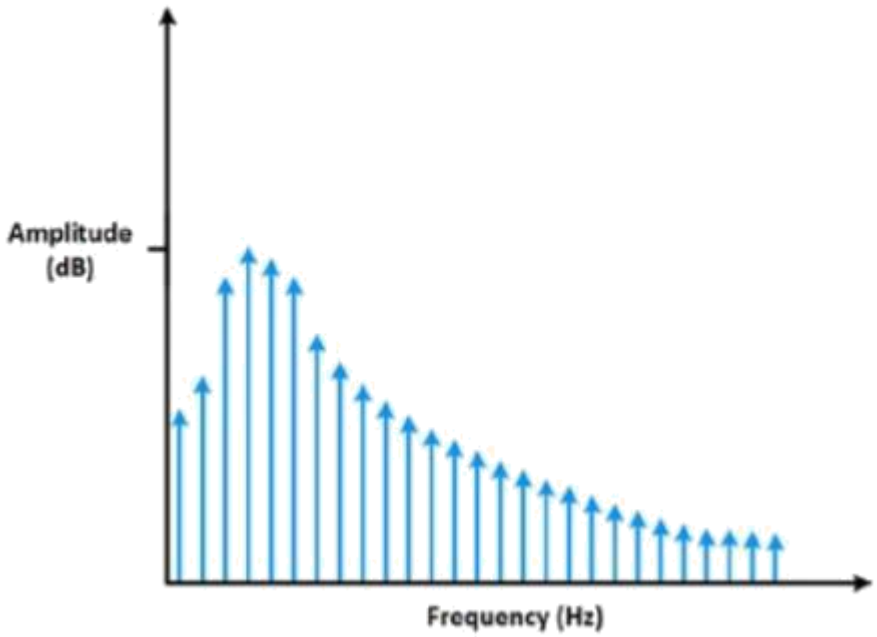


Figure 16. Applying a window minimizes spectral leakage.

There are several windowing mechanisms that are possible like the hann window, hamming window and so on. The below window is used in our project as it has a better damping characteristic as desired for the FFT design.

4.1. Hann window

This window is used for a signal whose characteristics are unknown. Instead of doing tradeoff between amplitude and frequency accuracy a good compromise is provided between the both by using this windowing technique.

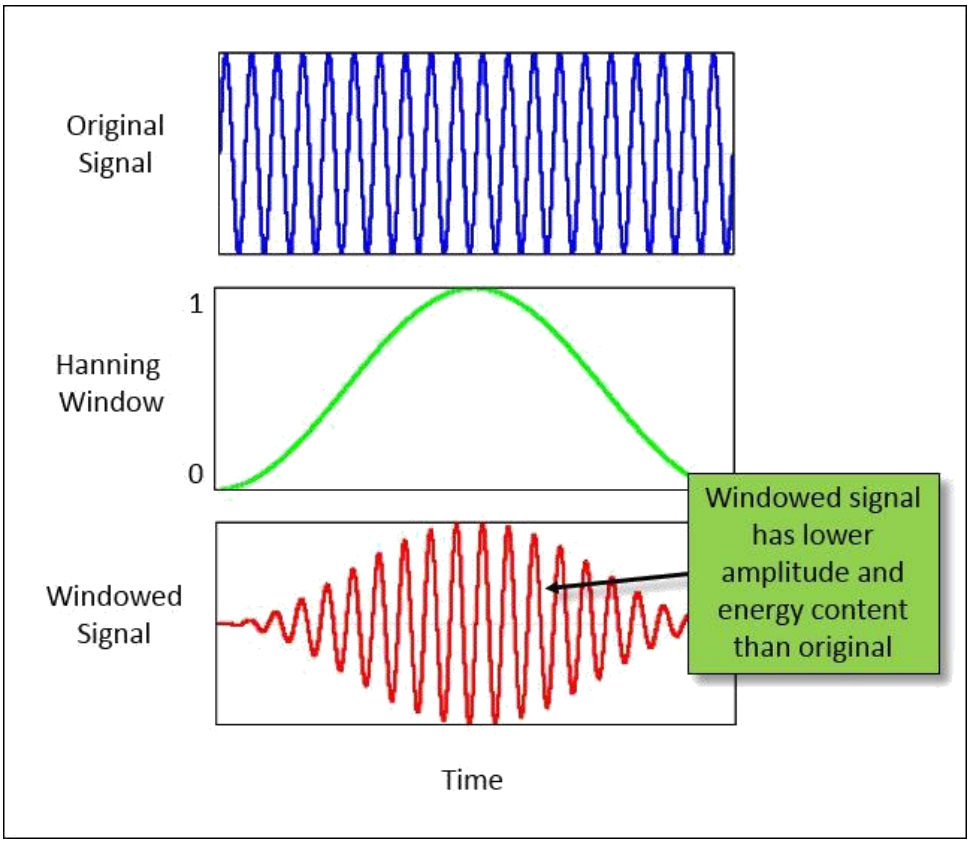


Figure 17. Multiplying original signal by hann window reduces the amplitude and energy in the signal [7]

5. MATLAB

This software is excessively used for computing. Integration of computation, visualization and programming is possible in this platform.

MATLAB software environment is user friendly. Mathematical notation type is used to express all kinds of problems and solutions. Main usage of MATLAB is application development. It makes usage of signal processing techniques easy by providing many predefined functions. It provides unified work flow for the development of embedded systems and streaming applications.

MathWorks provides design apps, DSP algorithm libraries, and I/O interfaces for real-time processing of streaming signals in MATLAB and Simulink. You can rapidly design and simulate streaming algorithms for audio, video, instrumentation, smart sensors, wearable devices, and other electronic systems

Entire filter design is first made in MATLAB and checked for output. Based on the results obtained cutoff frequency is determined and it is used in filter designing in Verilog.

5.1. Generating plots

The Generation of Graphs are critical part of simulation as it presents the expected output in a graphical manner for ease of understanding. In our scenario, we need to generate graphs for time and frequency domain at every point in simulation. Such as

1. When the input signal arrives
2. When the signal is chopped into 512 bits
3. After windowing
4. Post FFT and Filtering
5. After IFFT
6. Once the whole signal of merged data is obtained back

The below code presents two subplots, one for Time and the other for frequency. The frequency domain x axis is calculated as below

```
(-0.5:1/lengthOfSignal : 0.5-1/lengthOfSignal)*fs
```

The reason is because the signal has negative and positive axis corresponding to imaginary parts of the FFT output

The other parts of the code are self-explanatory.

* generateGraph.m

```
function generateGraph(name,signal,fs,figureCOunt)
    persistent n;
    if isempty(n)
        n = 0;
        num=0;
    end
    if(n==0)
      figure()
    end
    num=n*2+1;
    lengthOfSignal=length(signal);
    subplot(figureCOunt,2,num)
  %TIME PLOT
    t = 0:lengthOfSignal-1;
    plot(t,signal);title(['Time Domain ' name]);xlabel('Time,s');ylabel(name);
  %FREQUENCY PLOT
    subplot(figureCOunt,2,num+1)
    plot((-0.5:1/lengthOfSignal:0.5-1/lengthOfSignal)*fs,20*log10(abs(fftshift(fft(signal,lengthOfSignal)))))
    title(['Frequency Domain ' name]);xlabel('frequency,f');ylabel(name);
    %axis ([-2000 2000 -100 100]);
    n=n+1;
    if(n==figureCOunt)
      n=0;
    end
end
```

Figure 18. Code for generating plots

To maintain good clarity in the actual code, function “generategraph” is called into actual code. This gives plots in both time domain and frequency domain. Input signal is taken in the variable “signal” and it is plotted in time domain. After processing in the final stage frequency plot of filtered signal is obtained. True or false is set to generate graph function according the requirement. True is to generate plot and false is not to access the function.

5.2. Initialization

In the “audioread” function is used to fetch the input signal which is ultrasound heartbeat of a fetus. The input is in the wave format. Sampling frequency is given as 22050 hz. BS (512) is the block size for which the filtering is performed. This is nothing but 512 point FFT which is further designed in Verilog. The length of input file is 13782600.

```
clear all;
clc;
close all;


% **********FETCHING INPUT (FIGURE 1)**********
inputSignal = audioread('Input.wav');
expectedSignal = audioread('Expected.wav');


% **********INITIALIZATION*********
fs=22050;               %Samplng Rate of sound = 22050 samples/sec
BS = 512;
pkg load signal
outputSignal(1:length(inputSignal))=0;
```

Figure 19. Code for reading input and setting FFT points

```
generateplot=true;
loopEnd=(length(inputSignal)-mod(length(inputSignal),BS))

```

Figure 20. For loop constraints

```
% **********LOOPING THROUGH (FIGURE 2)**********
for i=1:(BS/2):loopEnd
%i=1;
    % **********SELECTING A RANGE OF INPUT SIGNAL*********
    if((i+BS-1)<=length(inputSignal))
      selectedRange  =   inputSignal(i:i+BS-1);
      windowLength = BS;
    else
      selectedRange  =   inputSignal(i:length(inputSignal)-1);
      windowLength = length(inputSignal)-i;
    end
```

Figure 21. Looping Through Each 512-point chunk

A for loop is made which includes which gives the portion of input signal which starts from 1 and end at 512. The next block starts from 512. For loop constraints are selected in such a way that only one fourth portion of first block sized input signal is removed and last one fourth portion of last block sized input signal. Due to application of hann window on the input signal, in everyone block sized input signal first one fourth and last one fourth signal is removed to reduce the spectral leakage. This leakage is considered as spectral leakage. Due to this lot of input signal is removed and there is a chance of losing the original signal. To avoid such condition, after every block size filtering last one fourth of the signal is added to next block size signal and then rest all computations are performed.

```
29    % **********Filter Design**********
30    filter = firl(900,0.05,'low');
31    fftFilter = fft(filter,BS);
```

Figure 22. Filter design

FIR filter of 900 order and 0.05 roll off which is a low pass filter is used. So, the entire signal length is divided into chunks whose length is of 512.

```
43      % **********APPLYING WINDOWING FUNCTION**********
44
45      window=hanning(windowLength);
46      windowedSignal    =    selectedRange.* window;
47
```

Figure 23. Hann window

Hann window is applied to window length which is of block size (initial size of window length is block size). Windowed signal is magnitude of vector product of selected range of input signal and the window. To perform this operation array and order size of both the elements should match.

```
48      % *********APPLYING FILTER*********
49      fftSignal = fft(selectedRange,BS);
50      fftFilteredOUtput = fftSignal .* transpose(fftFilter);
51      filteredResponse=ifft(fftFilteredOUtput,BS);
52      %filteredResponse      =    filter(hc,1,selectedRange);
53
```

Figure 24. Applying filter

Predefined function “FFT” is used to perform 512 point FFT. This function needs two parameters. “select range” is the range of processed signal and “BS” specifies the number of FFT points. When multiplying FFT signal of FFT of filter transpose is taken to match the order of the two variables.

```
% *********APPLYING INERSE FOURIER TRANSFORM*********

slicedOutput  = filteredResponse(BS/4:3*(BS/4));

 highCutOff=0.1;
 lowCutOff=-0.1;
 multiplicationFactor=1.63;
 reduction=2;
 DCoffset=0.18;

 for j=1:length(slicedOutput)
    slicedOutput1(j,1)=(real(slicedOutput(j)))-DCoffset+(1i*(imag(slicedOutput(j))-DCoffset));
end

for j=1:length(slicedOutput1)
  if(real(slicedOutput1(j))>highCutOff || real(slicedOutput1(j))
  <lowCutOff || imag(slicedOutput1(j))>highCutOff || imag(slicedOutput1(j))<lowCutOff)
    slicedOutput2(j,1)=multiplicationFactor*real(slicedOutput1(j))
    +1i*multiplicationFactor*imag(slicedOutput1(j));
  else
    %slicedOutput2(j,1)=0+1i*0;
    slicedOutput2(j,1)=(1/reduction)*real(slicedOutput1(j))+1i*(1/reduction)*imag(slicedOutput1(j));
  end
end
```

Figure 25. Inverse fourier transform

FFT is performed for the windowed signal and because of windowing, DC offset is introduced, and amplitude of original signal is reduced. So, multiplication factor is selected as 1.63 [3] to bring back the amplitude of filtered signal to the original.

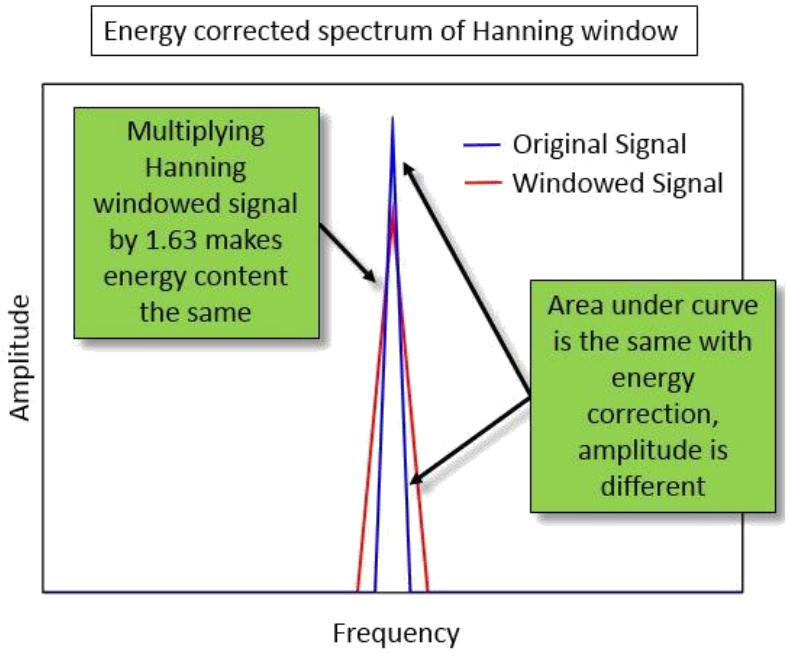


Figure 26. Selection of multiplication factor for hann window [7]

```
80      % *********ADDING *********
81      outputSignal(i:i+BS/2-1)=slicedOutput2(1:BS/2);
82
83      if(false)
84          figureCOunt=6;
85          generateGraph('selected Range',selectedRange,fs,figureCOunt);
86          generateGraph('windowed Signal',windowedSignal,fs,figureCOunt);
87          generateGraph('filtered Response',filteredResponse,fs,figureCOunt);
88          generateGraph('sliced Output',slicedOutput,fs,figureCOunt);
89          generateGraph('sliced Output 1',slicedOutput1,fs,figureCOunt);
90          generateGraph('output Output',outputSignal,fs,figureCOunt);
91      end
92  netx=1;
93
94  end
95
96  if(true)
97      figureCOunt=2;
98      generateGraph('Input Signal',inputSignal,fs,figureCOunt);
99      %generateGraph('Expected Signal',expectedSignal,fs,figureCOunt);
100     generateGraph('output Signal',outputSignal,fs,figureCOunt);
101 end
102
103 if(false)
104     sound(inputSignal,fs);
105     %sound(expectedSignal,fs);
106     sound(outputSignal,fs);
107 end
108
109 if(false)
110     audiowrite('output.wav',outputSignal,fs);
111 end
```

Figure 27. Producing final filtered output

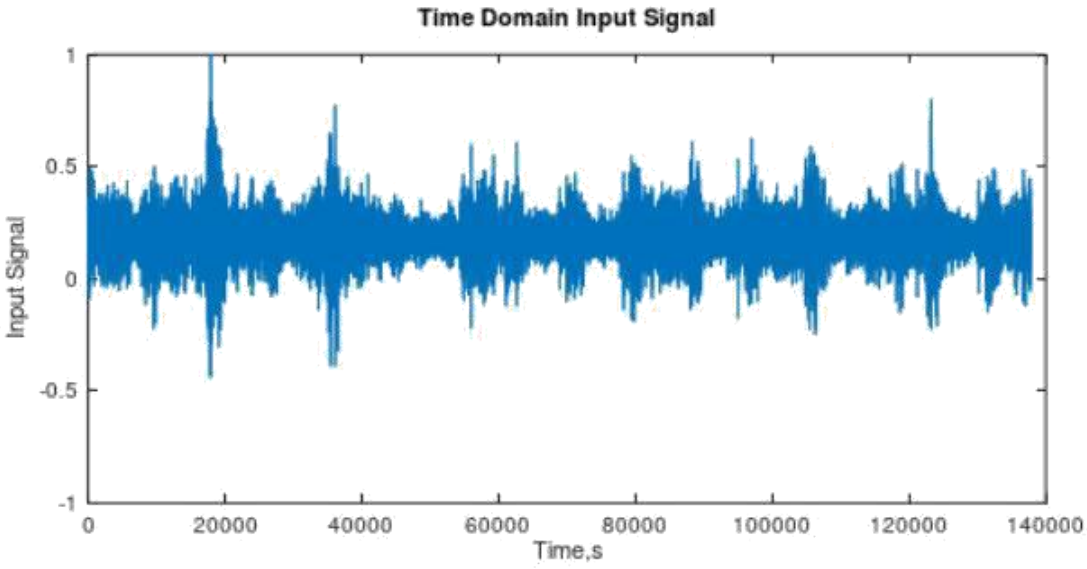


Figure 28. Plot of Input signal in time domain

Input signal plot in time domain. This contains lot of noise whose frequency can't be determined from time domain plot. The spikes indicate heartbeat and as you the signal is low frequency (Occurs at a lesser rate than the noise signal). Main aim is to remove all other frequency components which all are considered as noise components.

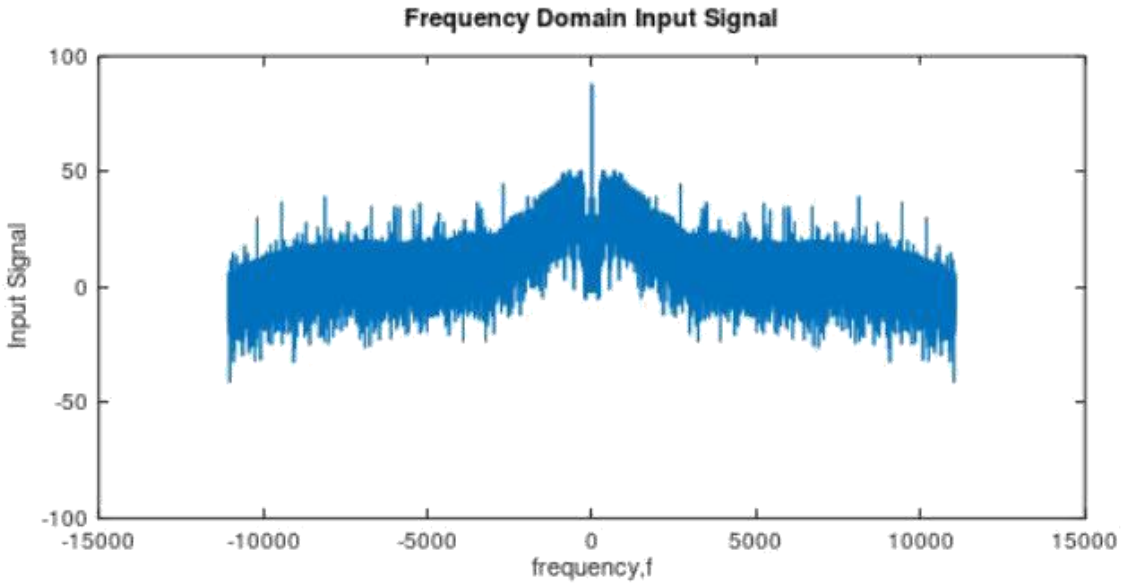


Figure 29. Plot of input signal in frequency domain

In figure 27 the frequency component at zero frequency determines the Doppler heartbeat frequency. The maximum noise floor ranges from -30db to 50db. To eliminate those frequencies low pass FIR filter of order 900 and roll off 0.05 is selected.

Few Considerations for Noise Removal

1. The signal is low frequency. So, we need to design a low pass filter with a cutoff frequency (which is to be selected based on trial and error and brute force method)
2. An assumption of an error on Window to be 50 percent was made. Meaning, if we window 512 samples, only the mid 256 Samples will be valid data. Remaining should be dumped as they will be grounded to zero.

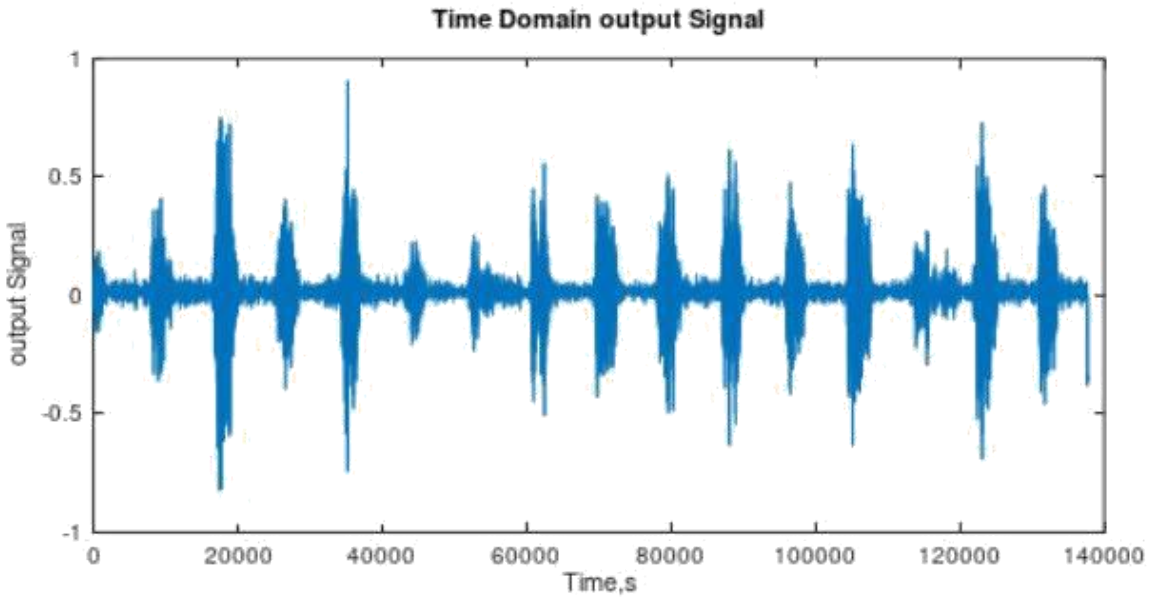


Figure 30. Plot of output signal in time domain

Before getting plot in figure 29, DC offset of 0.18db is observed and amplitude was reduced to nearly 2. So appropriate code is made to overcome that error and the desired output is obtained.

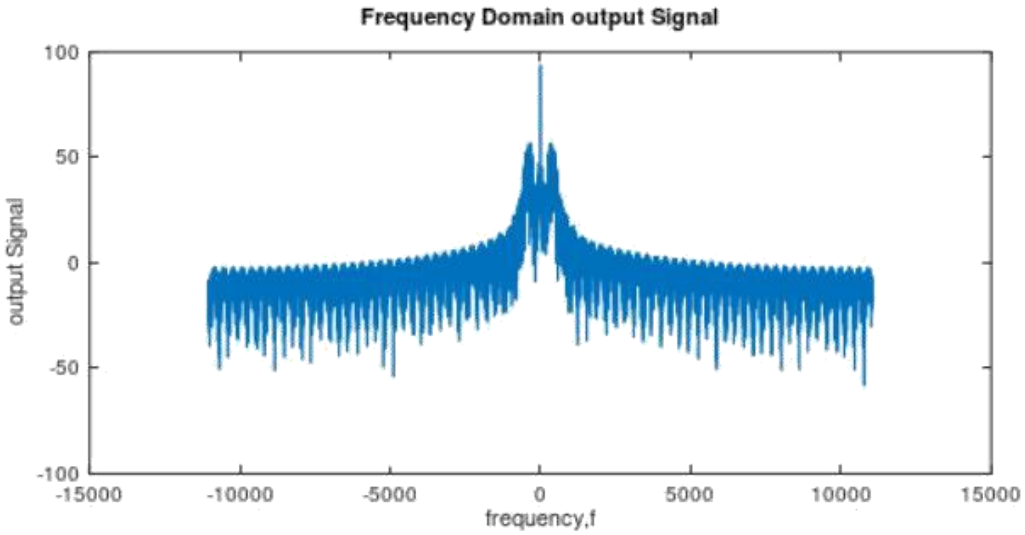


Figure 31. Plot of output signal in frequency domain

The noise floor in the above figure is pushed down to 0db. The output of this is heard and the noise is eliminated.

6. System Verilog

System Verilog is a good platform to provide a detailed design specifications of a digital circuit. Creation of design specifications is trivial when compared to the amount of work required to translate the specification to a schematic based structural description needed to fabricate a device.

Because of system Verilog design engineer's productivity is increased in very few years. CAD tools that are used in digital design can be categorized into two as "front end" tools that allows capturing and simulating the design and "back end" tools that are used to synthesize a design, link it to a technology and analyzes its performance

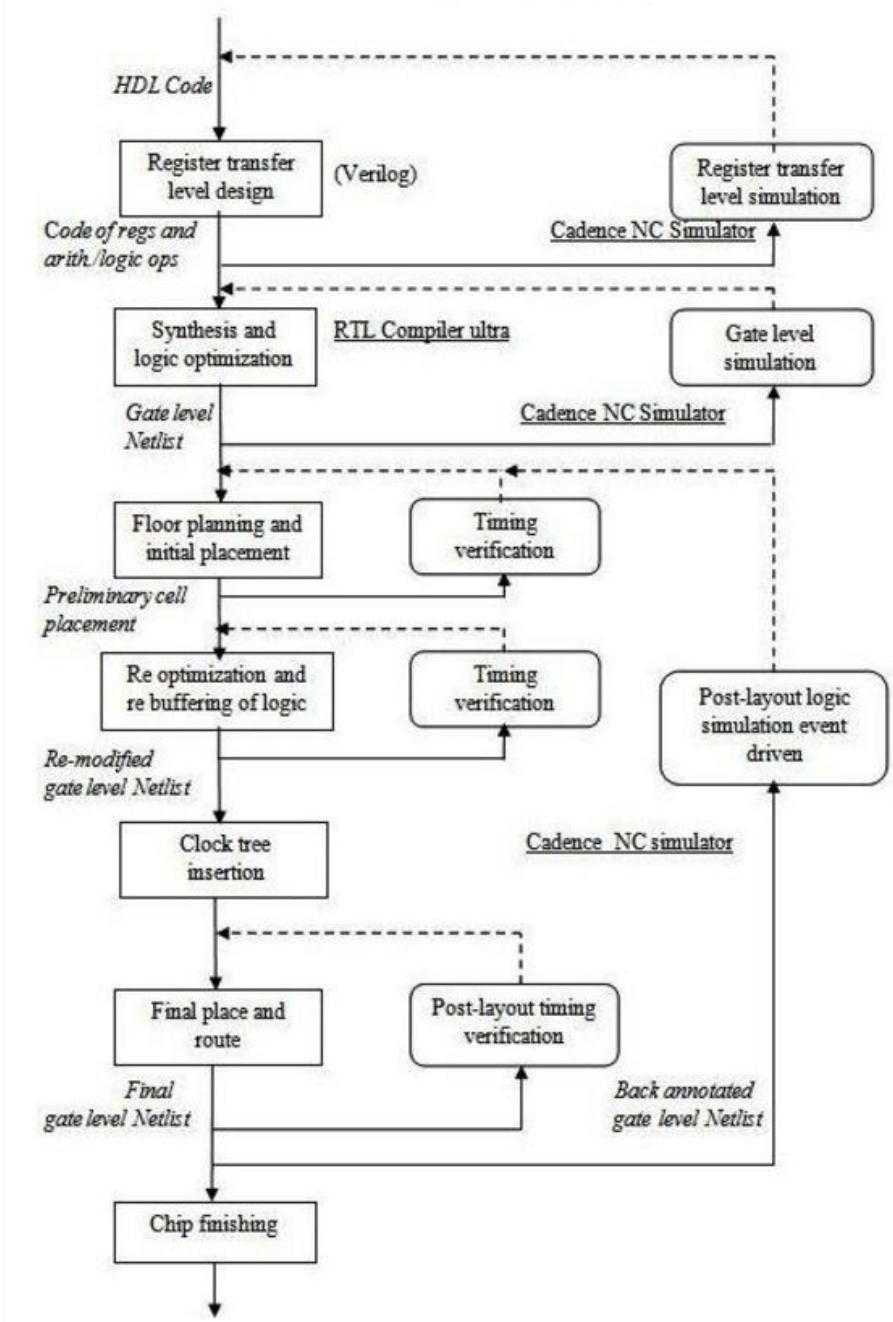


Figure 32. Digital design flow

7. Design

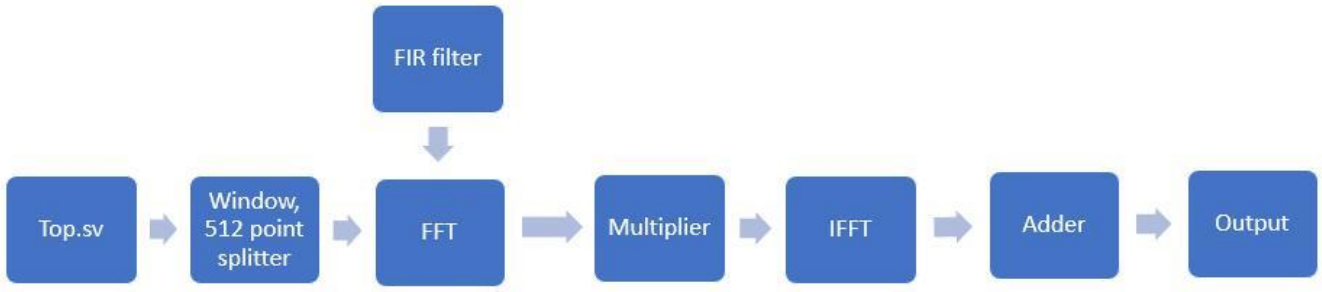


Figure 33. Design flow in System Verilog

After getting filtered results of input signal from MATLAB for ultrasound of heartbeat of fetus, the input.wav file is converted into text file, hann window is converted to text file and fir filter coordinates is converted into text file. All these files are used to perform filtering of input and resultant output values matches with the output from MATLAB.

7.1. Test bench

```
initial begin
    inputDataFile = $fopen("signalInput.txt","r");
    pushin=0;
    rst=1;
    repeat(3) @(posedge(clk)) #1;
    rst=0;
    @(posedge(clk)) #1;
    inputCount=0;
    pushin=1;
    endOfFile=0;
    $display("start");
    while(endOfFile==0) begin
        if(!$feof(inputDataFile)) begin
            if(stopOutWindow==0) begin
                $display(inputCount);
                //readDataReal=$fscanf(inputDataFile,"%f\n",realin);
                readDataReal=$fscanf(inputDataFile,"%f\n",realin);
                imagin= 0;
                inputCount=inputCount+1;
            end
        end
        else
            endOfFile=1;
        @(posedge clk) #1;
```

Figure 34. Fetching input

Input “fetal doppler” is in .wav format. to convert this into digital form, which is a vector representation, MATLAB is used. This is of size 1307800. Audio file in .wav format is easy to convert to digital form. So, any audio file which any other format is first converted into .wav file for representing it in digital format.

After converting input into text file that contains real and imaginary values, at every posedge of the clock real values are read in a variable called “realdata” and imaginary values are read in a variable “imagindata”.

```
initial begin
    forever begin
        if(pushoutFFt) begin
            filterRealDataFile = $fopen("filterInputReal.txt","r");
            filterImagDataFile = $fopen("filterInputImag.txt","r");
            while(pushoutFFt) begin

                if(!$feof(filterRealDataFile) && !$feof(filterImagDataFile)) begin
                    if(pushoutFFt) begin
                        //readDataReal=$fscanf(filterDataFile,"%f\n",realin);
                        readDataFilterReal=$fscanf(filterRealDataFile,"%f\n",
                            realinFilter);
                        readDataFilterImagin=$fscanf(filterImagDataFile,"%f\n",
                            imaginFilter);
                    end
                end
                else begin
                    if(pushoutFFt) begin
                        //readDataReal=$fscanf(filterDataFile,"%f\n",realin);
                        realinFilter=0;
                        imaginFilter=0;
                    end
                end
                @(posedge clk) ;
            end
            $fclose(filterRealDataFile);
            $fclose(filterImagDataFile);
        end
        @(posedge clk) ;
    end
end
```

Figure 35. fetching filter data

Like how the real and imaginary parts of input data is read, in the same manner low pass FIR filter vector representation text file is loaded into design using similar code in system verilog. Here FIR filter is of order 900, roll 0ff 0.05 and its a low pass filter. These numbers are selected by trial and error method. The white noise floor is way below the frequency level of required heart beat. So low pass Filter is selected for filtering.

7.2. Windowing

Windowing technique is used to eliminate the spectral leakage in the signals. Usually the signals which are not periodic contains frequency harmonics in FFT which are not actually present in the signal. These are of high frequency components. the signal frequencies seem like as if the energy been transferred from one harmonic to other harmonic. To eliminate this spectral leakage windowing is applied to the signal which rolls down the signal at starting and at ending to zero, this is been performed because spectral leakage is usually observed at the starting and ending of the recorded non-periodic signal. Hann window is used. The hann window's vector representation is obtained using the MATLAB.

Hann window

$$w(k) = 0.5 - 0.5\cos\left(\frac{2\pi k}{N-1}\right)$$

Hann window formula [8]

7.3. FFT

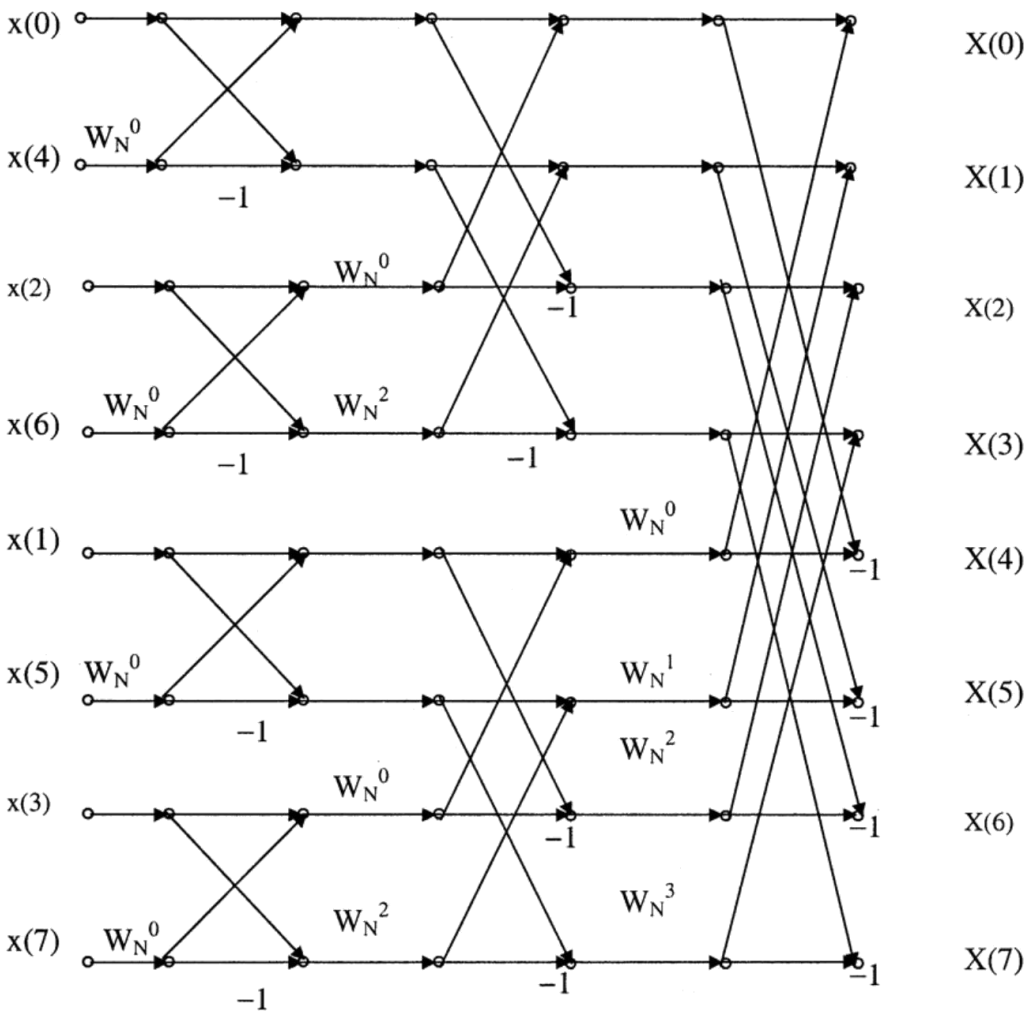
Fast Fourier transform concept is used to convert time domain input signal into frequency domain. If the input to FFT are two to the race any natural number, then it works better. Input signal is a continuous signal in time vs amplitude. In frequency domain signal is plotted between normalized frequency and magnitude response (dB) of signal. The harmonic that is at higher magnitude is determined as heartbeat of fetus.

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}, \qquad k = 0,1,\ldots,N-1$$
$$= \sum_{n \text{ even}} x(n) W_N^{kn} + \sum_{n \text{ odd}} x(n) W_N^{kn}$$
$$= \sum_{m=0}^{(N/2)-1} x(2m) W_N^{2mk} + \sum_{m=0}^{(N/2)-1} x(2m+1) W_N^{k(2m+1)}$$

FFT formula [9]

The formula in above figure is used to compute FFT of filter co-ordinates and hann window co-ordinates. The weight components that are used in computation are called twiddle factors or butterflies. These twiddles contain both real and imaginary values. Real values and imaginary values are of floating-point format and they are saved in arrays in System Verilog.

The data is first saved to input to a buffer which is of size 512. This takes 512 clock cycles. Once this data is transferred, the data is moved completely to a working buffer which is used to compute FFT. The figure 36 shows how data is transferred into input buffer



$W_N^0 = 1,\ W_N^1 = (1-j)/\sqrt{2},\ W_N^2 = -j,\ W_N^3 = -(1+j)/\sqrt{2}$

```
if(inputDataStartFLag) begin
    reversedAddress= ({inputAddressCounter[0],inputAddressCounter[1],inputAddressCounter[2
        ],inputAddressCounter[3],inputAddressCounter[4],inputAddressCounter[5],
        inputAddressCounter[6],inputAddressCounter[7],inputAddressCounter[8]});
end
else
     reversedAddress=0;
```

Figure 36. Saving 512 inputs to buffer

As it is seen in the figure 36, the input address is reversed and stored in reversed order. The table 1 shows an example for 8-bit FFT (with 3 bits used for representing address values)

Input Count	Binary Count	Reversed count Value	Address in which the data is stored
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Table 1. Address locations of 8bit FFT

In short, the first input is stored to address 1, the second input are stored to address 4 (001 reversed will be 100), third input to address 6 (011 reversed will be 110) and so on.

Now, since the data is stored accordingly, once the calculation starts, data is taken serially from address 1, 2 ,3 till 512. But calculations are performed as seen in figure 37.

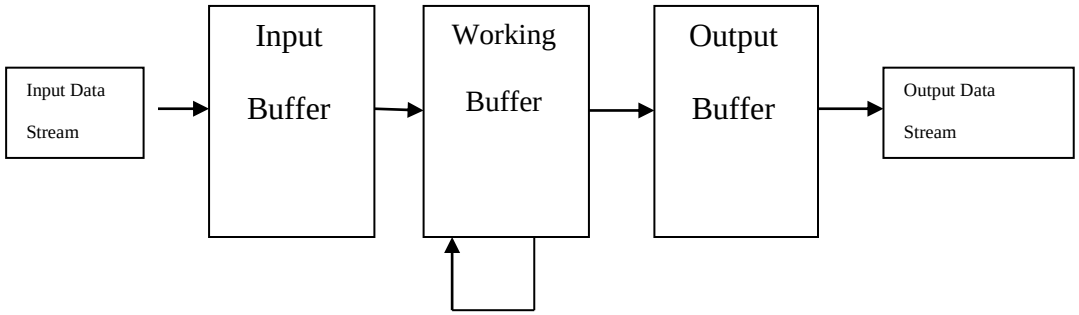
```
always @(posedge clk or posedge rst) begin
    if(rst) begin
        stopout=0;
        realOut=0;
        imagOut=0;
        pushout=0;
        inputCount=0;
        outputCount=0;
    end
    else begin
        if(pushin && !stopout && inputCount<510) begin
            realBuffer[inputCount]<=#1 realInput;
            imagBuffer[inputCount]<=#1 imagInput;
            inputCount<=#1 inputCount+1;
        end
        else if(inputCount>=510) begin
            realBuffer[511]<=#1 realInput;
            imagBuffer[511]<=#1 imagInput;
            stopout<=#1 1;
            inputCount<=#1 0;
        end
        else if(!stopIn && stopout && outputCount<=511) begin
            if(outputCount==0)
                pushout<=#1 1;
            else
                pushout<=#1 0;
            realOut<=#1 realBuffer[outputCount];
            imagOut<=#1 imagBuffer[outputCount];
            outputCount<=#1 outputCount+1;
        end
        else if(outputCount>511) begin
            stopout<=#1 0;
            pushout<=#1 0;
            outputCount<=#1 0;
        end
    end
end
```

Figure 37. Saving 512 inputs to buffer

Figure 36 is a snippet of code which performs the operation of storage and retrieval of data from input buffer and output buffer.

7.4. Buffers

There are in total three buffers acting in a pipelined fashion as seen below



Input Buffer receives data and stores them in reversed addresses as explained before. Once we get all 512 points, the data is moved to working buffer. The data from this buffer is sent to butterflies (16 at a time) to 8 butterflies working in parallel. Once the computation is done, the data is put back to the same working buffer such that they overlap the data which is already used.

```
Butterfly_fft buff1(lastStageFlag,clock,reset,pushinB0,pushoutB0,realIp1B0,realIp2B0,imgIp1B0,
    imgIp2B0,twiddleFactorRealB0,twiddleFactorimgB0,stateIp1B0,stateIp2B0,ReOp1B0,ReOp2B0,imgOp1B0
    ,imgOp2B0,stateOp1B0,stateOp2B0,lastStageFlagOp);

Butterfly_fft buff2(lastStageFlag,clock,reset,pushinB1,pushoutB1,realIp1B1,realIp2B1,imgIp1B1,
    imgIp2B1,twiddleFactorRealB1,twiddleFactorimgB1,stateIp1B1,stateIp2B1,ReOp1B1,ReOp2B1,imgOp1B1
    ,imgOp2B1,stateOp1B1,stateOp2B1,lastStageFlagOp);

Butterfly_fft buff3(lastStageFlag,clock,reset,pushinB2,pushoutB2,realIp1B2,realIp2B2,imgIp1B2,
    imgIp2B2,twiddleFactorRealB2,twiddleFactorimgB2,stateIp1B2,stateIp2B2,ReOp1B2,ReOp2B2,imgOp1B2
    ,imgOp2B2,stateOp1B2,stateOp2B2,lastStageFlagOp);

Butterfly_fft buff4(lastStageFlag,clock,reset,pushinB3,pushoutB3,realIp1B3,realIp2B3,imgIp1B3,
    imgIp2B3,twiddleFactorRealB3,twiddleFactorimgB3,stateIp1B3,stateIp2B3,ReOp1B3,ReOp2B3,imgOp1B3
    ,imgOp2B3,stateOp1B3,stateOp2B3,lastStageFlagOp);

Butterfly_fft buff5(lastStageFlag,clock,reset,pushinB4,pushoutB4,realIp1B4,realIp2B4,imgIp1B4,
    imgIp2B4,twiddleFactorRealB4,twiddleFactorimgB4,stateIp1B4,stateIp2B4,ReOp1B4,ReOp2B4,imgOp1B4
    ,imgOp2B4,stateOp1B4,stateOp2B4,lastStageFlagOp);
```

Figure 38. Butterfly units running in parallel

This process of computation and overlapping goes on until the final state of state machine is reached. During the final state, the output is directly written to output buffer. Once all 512 data are written, output flag is set high and data is sent in a streamlined fashion.

```
else if(inputBufferFlag) begin
    for(i=0;i<512;i=i+1) begin
        realWorkingBuffer[i] <= #1 realInputBuffer[i];
        imagWorkingBuffer[i] <= #1 imagInputBuffer[i];
    end
    inputBufferFlag<=#1 0;
        else if(computationFlag) begin
            if(state==0)
                state<=#1 1;
            else if(state==288) begin
                state<=#1 0;
                computationFlag<=#1 0;
                lastStageFlag<=#1 0;
            end
            else
                state<=#1 (state+1);
```

Figure 39. Transferring data to working buffer and increment states of FFT calculation thereafter

The code in figure 39 snippet shows the input to working buffer and incrementing states of FFT. Each state of FFT contains inputs to 8 buffers and twiddles to these butterflies. All these states are generated using a Java program which is written such that the entire states are generalized. On changing constants to 1024, the states for 1024-point FFT can also be generated.

7.5. IFFT

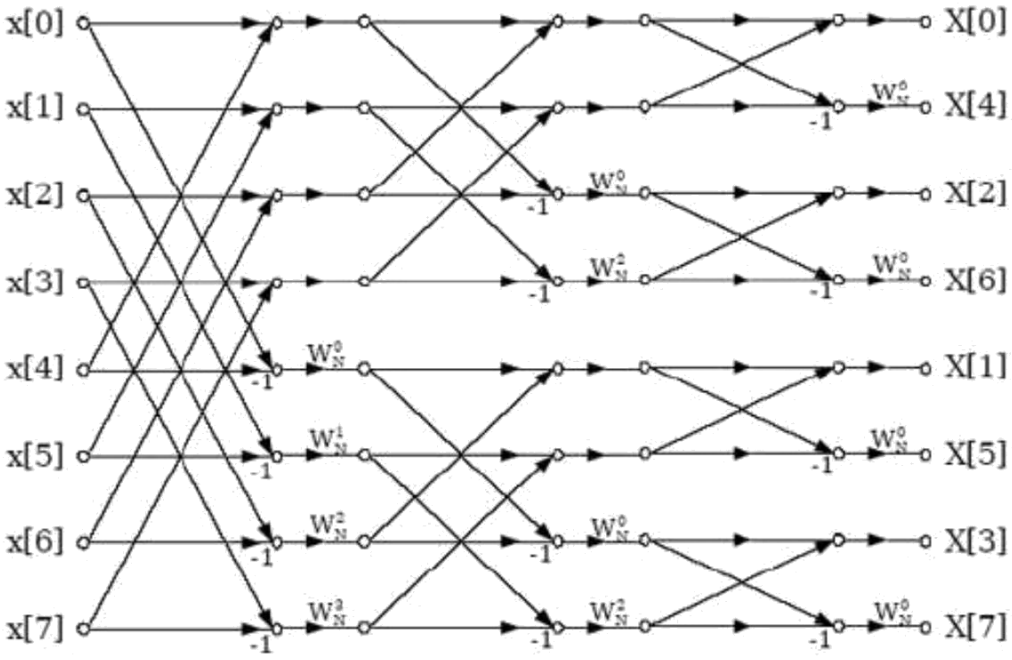
Inverse Fast Fourier transform concept is used to convert frequency domain input signal into time domain. If the input to IFFT are two to the race N, then it works better. Input signal is a Frequency Signal vs Frequency spectrum. In frequency domain signal is plotted between normalized frequency and magnitude response (dB) of signal. The harmonic that is at higher magnitude is determined as heartbeat of fetus.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i 2\pi k n / N} \qquad k = 0, \ldots, N-1,$$

IFFT formula [10]

The formula in above figure is used to compute IFFT of Frequency response of filtered signal. The weight components that are used in computation are called twiddle factors or butterflies. These twiddles contain both real and imaginary values. Real values and imaginary values are of floating-point format and they are saved in arrays in system Verilog.

Data is saved in input buffer, which is of size 512. This takes 512 clock cycles. Once this data transferred, the data is moved completely to a working buffer, which is used to compute FFT. The figure 40 shows how data is transferred into input buffer




```
else if(inputBufferFlag) begin
    for(i=0;i<512;i=i+1) begin
        realWorkingBuffer[i] <= #1 realInputBuffer[i];
        imagWorkingBuffer[i] <= #1 imagInputBuffer[i];
    end
    inputBufferFlag<=#1 0;

        else if(computationFlag) begin
            if(state==0)
                state<=#1 1;
            else if(state==288) begin
                state<=#1 0;
                computationFlag<=#1 0;
                lastStageFlag<=#1 0;
            end
            else
                state<=#1 (state+1);
```

Figure 40. Transferring data to working buffer and increment states of FFT calculation thereafter

Once the data is transferred to working buffer the states of FFT are incremented as shown below. Each state of FFT contains inputs to 8 buffers and twiddles to these butterflies. All these states are generated using a java program which is written such that the entire states are generalized. On changing constants to 1024, the states for 1024-point IFFT can also be generated. The code in figure 41, shows the first state of the entire 288 states.

```
always@(*) begin
      M1_d=ReIp2*twiddleFactorReal;
      M2_d=ReIp2*twiddleFactorComplx;
      M3_d=CmpIp2*twiddleFactorReal;
      M4_d=CmpIp2*twiddleFactorComplx;

      S1_d<=M1-M4;
      S2_d<=M2+M3;

      ReOp1_d<=ReIp1_2d+S1;
      ReOp2_d<=ReIp1_2d-S1;
      CmpOp1_d<=CmpIp1_2d+S2;
      CmpOp2_d<=CmpIp1_2d-S2;
end

endmodule
```

Figure 41. Writing data from working buffer to butterfly inputs

7.6. Comparing Design vs Simulation

After taking constraints from MATLAB that are employed in the filtering of "input.wav" Design is made in the system verilog to obtain same results. Some of the results are more similar to that MATLAB results and some results from System Verilog are much cleaner than that of from MATLAB.

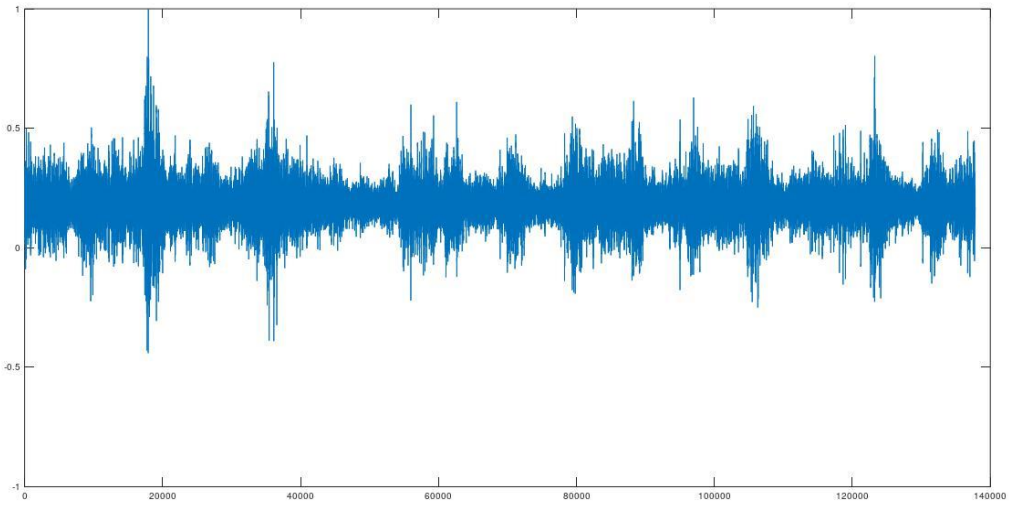


Figure 42. Input signal (with noise)

In figure 42 input audio file which contains much of white noise is plotted in time domain. Because of white noise the low frequency components of heartbeat are unheard and it's difficult to obtain the actual heartbeat sound.

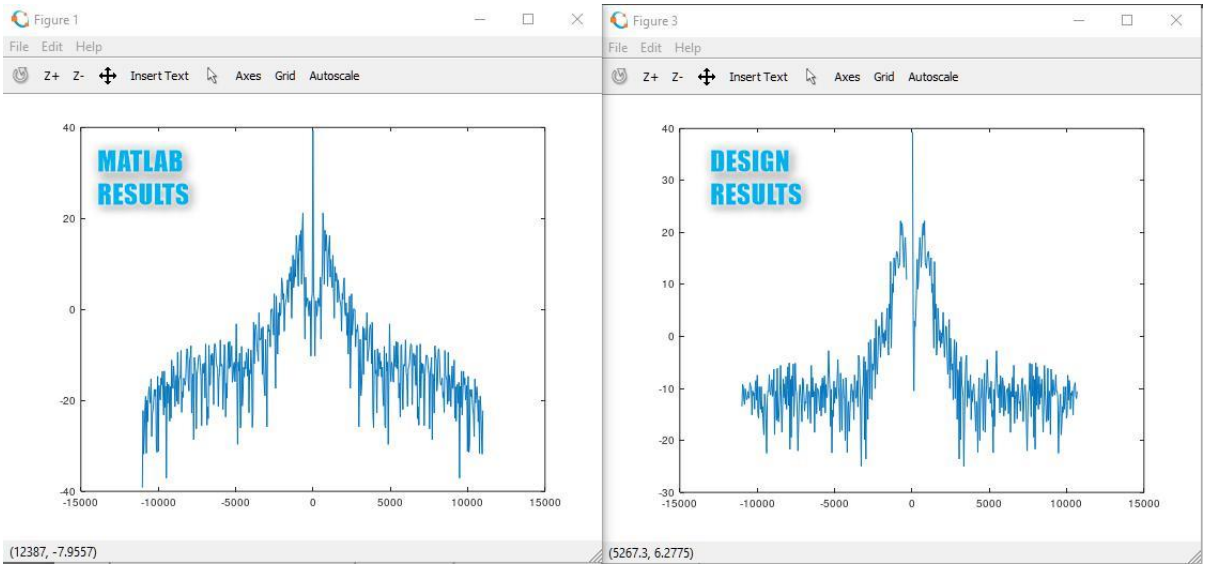


Figure 43. FFT output of the first 512 points

FFT is performed in for each block of input signal whose size is 512. FFT plot of first 512 samples of input signal is plotted and shown on left side of figure 43 and the result of 512-point FFT performed using FFT design in System Verilog is plotted and is shown on the right side of figure 48. It is observed that results from FFT design in System Verilog are much cleaner. The same fashion of output clarity is seen for filtered output of first 512 samples of input signal.

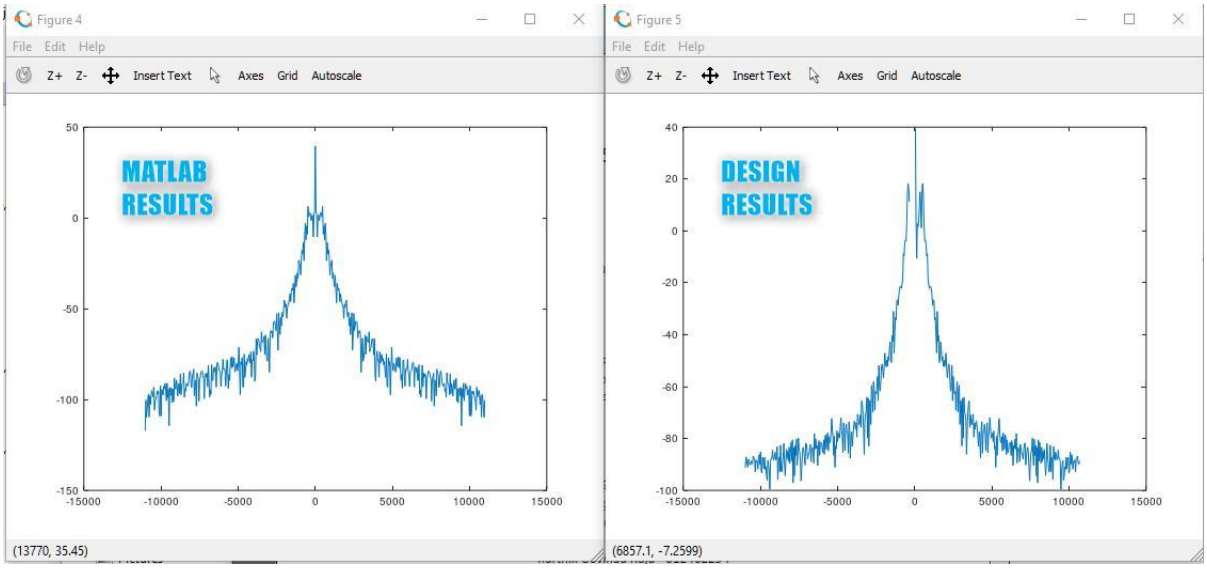


Figure 44. Filtered output of the first 512 points

After filtering entire input signal, it is converted back to wave format and plotted in MATLAB. It is observed in figure 45 that much of noise is removed and actual heartbeat dominates the noise floor.

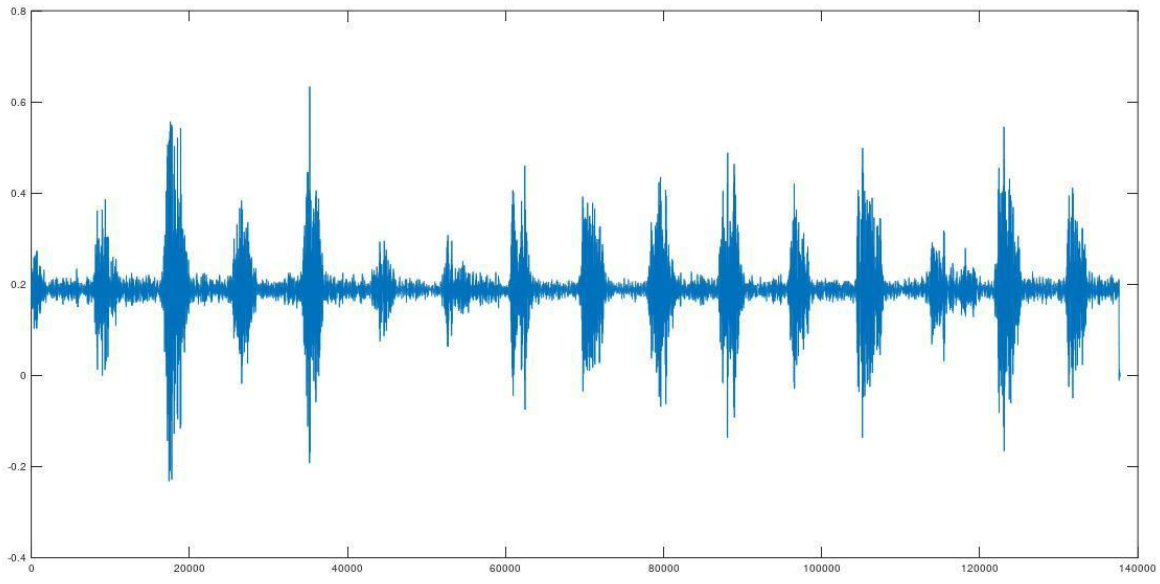


Figure 45. Final output of entire signal after IFFT

8. Java Code for generating states of FFT and IFFT

The coding for this project involved trial and error method with various FFT designs such as 256 points, 512 points, 1024 points and so on. Writing the code for all of these was a tedious job. So instead we designed a java application which was intelligent enough to generate System Verilog files customized for each FFT and IFFT design.

The inputs to the java program defined the design parameters. The same are as below

1. Number of Multipliers

The Design contains several multiplier units working in parallel to get the FFT work done. The number of multipliers high the rate of output generation. But this comes at a cost of larger synthesized area. Thus, as a tradeoff different numbers needed to be tried.

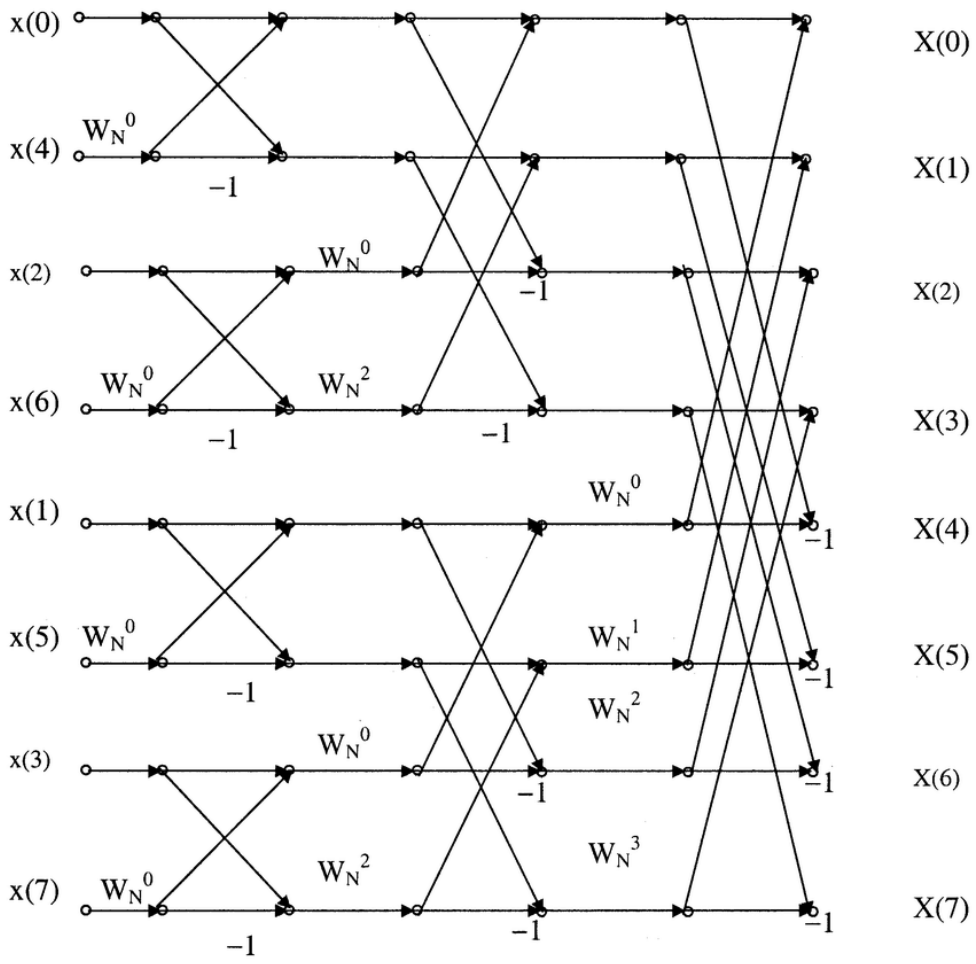
2. Number of points (256, 512, …)

As explained in the start of this page, we need to design a system where we can easily switch between different FFT and IFFT designs and do a trial and error method to obtain the same precision of noise cancellation as seen in simulation. Thus, we designed a java program which was robust such that we could easily generate the entire code for different points as per the input.

The number of points combined with Number of multipliers decided the max number of states of the design state diagram

3. FFT or IFFT

FFT and IFFT design are only a small change with twiddles varying as shown below



$W_N^0 = 1$, $W_N^1 = (1-j)/\sqrt{2}$, $W_N^2 = -j$, $W_N^3 = -(1+j)/\sqrt{2}$

Figure 46. FFT butterflies

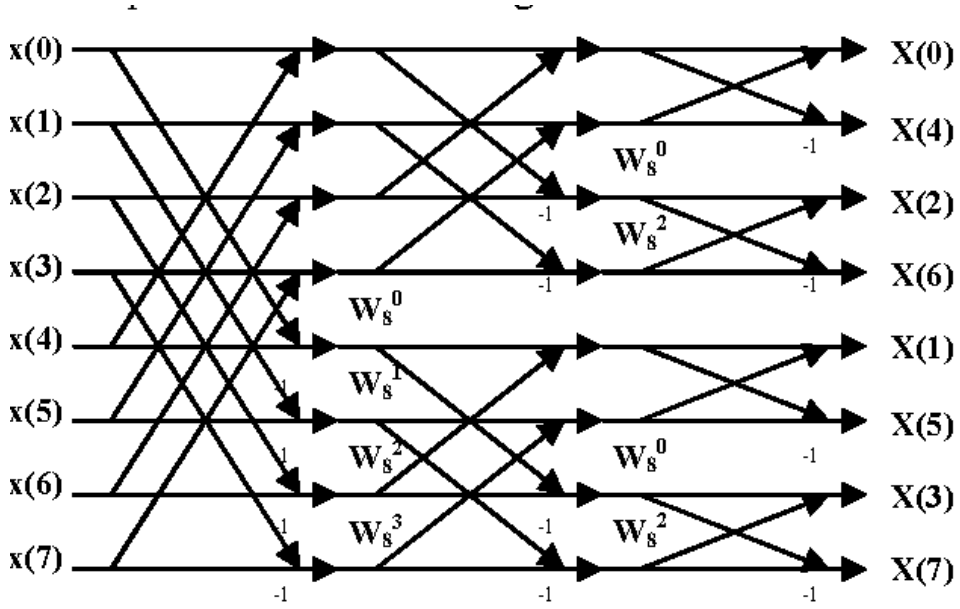


Figure 47. IFFT butterflies

The main differences between an FFT and IFFT are

1. Twiddles are calculated before the execution of the butterfly operation for a fft. Whereas the twiddles are computed after butterfly operation of IFFT
2. Twiddle factors a slight calculation change. In FFT, the imaginary part come with negative sign whereas in IFFT the imaginary part come with positive sign.
3. The initial set of addresses as you see above are reversed for FFT. Whereas for IFFT the initial is sequential, whereas the output is reversed.

Figure 48 shows all the above points in code

```
package txtFileGenerator;
public class Twiddles_512_ifft {
    public static double generate(boolean imag,int currentStage, int level){
        switch (currentStage){
        case 0:
            switch(level){
            case 0:if(!imag) return 1.000000;
            else return 0.000000;
            case 1:if(!imag) return 0.999925;
            else return 0.012272;
            case 2:if(!imag) return 0.999699;
            else return 0.024541;
            case 3:if(!imag) return 0.999322;
            else return 0.036807;
            case 4:if(!imag) return 0.998795;
            else return 0.049068;
            case 5:if(!imag) return 0.998118;
            else return 0.061321;
```

```
                else return 0.024541;
                case 255:if(!imag) return -0.999925;
                else return 0.012272;
                }
            case 1:
                switch(level){
                case 0:if(!imag) return 1.000000;
                else return 0.000000;
                case 1:if(!imag) return 0.999699;
                else return 0.024541;
                case 2:if(!imag) return 0.998795;
                else return 0.049068;
                case 3:if(!imag) return 0.997290;
                else return 0.073565;
                case 4:if(!imag) return 0.995185;
                else return 0.098017;

            }
        case 2:
            switch(level){
            case 0:if(!imag) return 1.000000;
            else return 0.000000;
            case 1:if(!imag) return 0.998795;
            else return 0.049068;
            case 2:if(!imag) return 0.995185;
            else return 0.098017;
```

Figure 48. Twiddle Generation showing different stages of Twiddle

The above twiddles were further generated using another Octave snippet shown below

```
for mm = 0:1:(fft_length-1)
    theta = (-2*pi*mm*1/fft_length);
    twiddle(mm+1) = cos(theta) + (1i*(sin(theta)));

    real_twiddle = real(twiddle);
    im_twiddle = imag(twiddle);
    printf (' case %d:\n',mm);
    printf ('   switch(level){\n');
    for c=1 : length(real_twiddle)

      printf('      case %d:if(!imag) return %f;\n',c-1,real_twiddl
      printf('        else return %f;\n',im_twiddle(c));
      count=count+1;
    end
    printf('    }\n');
    count=count+1;
end
printf('   }\n');
printf('  }\n');
printf('}\n');
```

Figure 49. Twiddle Value Generation

These twiddle values were further used to generate the states of the FFT and IFFT as below

```
//boolean stage0flag;
public void caseStatementGenerator() throws IOException{
    BufferedWriter writer = new BufferedWriter(new FileWriter("ifftCalculator.v"));
    printHardCodeStatements(writer);
    printDefaultCaseStatements(writer);
    for (int tstage = 0; tstage < numberOfStages; tstage++) {
        int stage=numberOfStages-1-tstage;
//      if(stage==0) {stage0flag=true;}
//      else {stage0flag=false;}
        double end=Npoint/Math.pow ( 2, (stage+1));
        for(int p=0; p<end;p++){
            int firstVariable=(int) ((Math.pow ( 2, (stage+1)))*p);
            int end2=(int)Math.pow ( 2, (stage));
            for(int q=0;q<end2;q++){
                int input1=firstVariable+q;
                int input2=firstVariable+q+(int)Math.pow ( 2, (stage));
                printCaseStatements(writer,input1,input2,tstage,q,(tstage+1==numberOfStages && p==0 && q==0),(tstage==0 &&
                System.out.println(input1+" : "+input2+" ::stage =  "+tstage);

            }
        }
    }
//      writer.write("\t\t\t "+(numberOfBitsToRepresetAllStates)+"'d"+257+" : begin\n");
```

Figure 50. Calculation of number of states

Once the number of states are known, code generation for each state is required. And this takes the looping logic shown in figure 50.

```
for(int p=0; p<end;p++){
    int firstVariable=(int) ((Math.pow ( 2, (stage+1)))*p);
    int end2=(int)Math.pow ( 2, (stage));
    for(int q=0;q<end2;q++){
        int input1=firstVariable+q;
        int input2=firstVariable+q+(int)Math.pow ( 2, (stage));
        printCaseStatements(writer,input1,input2,tstage,q,(tstage+1==numberOfStages && p==0 && q==0),(tstage
        System.out.println(input1+" : "+input2+" ::stage =  "+tstage);

    }
}
```

Figure 51. Looping conditions

Here the looping condition takes care of the below points

1. The number of states
2. Number of multipliers in each state
3. Calculating the number of stage in each state
4. Then once we drill down to one stage of one state, we generate the state variable which include
    a. Twiddles from twiddle generator
    b. Variable address from memory

Once all the data is available, generate the statement is next step. This is done by printing the data into a text file and saving the same as with ".sv" extension.

```
if(count%noOfMultipliers ==0){
    writer.write("\t\t\t "+caseVariable+++" : begin\n");
}

if(firstStage) {
    writer.write("\n\t\t\t\t\t lastStageFlag<=1'b0;\n");
}

writer.write("\n\t\t\t\t\t //("+input1+","+input2+")\n");

//writer.write("\t\t\t\t pushinB"+count%noOfMultipliers+" <= 1'b1;\n");
writer.write("\t\t\t\t imgIp1B"+count%noOfMultipliers+" <=#1 imagWorkingBuffer["+input1+"];\n");
writer.write("\t\t\t\t realIp1B"+count%noOfMultipliers+" <=#1 realWorkingBuffer["+input1+"];\n");
writer.write("\t\t\t\t imgIp2B"+count%noOfMultipliers+" <=#1 imagWorkingBuffer["+input2+"];\n");
writer.write("\t\t\t\t realIp2B"+count%noOfMultipliers+" <=#1 realWorkingBuffer["+input2+"];\n");

double real = 0;
double imag = 0;
switch (Npoint) {
case 256:
    real=Twiddles_256.generate(false,currentStage,q);
    imag=Twiddles_256.generate(true,currentStage,q);
    break;
case 512:
    real=Twiddles 512 ifft.generate(false,currentStage,q);
```

Figure 52. Generating twiddles

The above code in the figure 52 shows the same, where it is observed writer object writing System Verilog code into text file.

This approach is similar made as it is easy in java to system Verilog converter. But in simpler manner, the java script generates System Verilog design files which are compilation free and easily customizable.

9. Summary

The ultrasound heartbeat of fetus is taken as input signal which contains lot of noise. This is first filtered in MATLAB. Using constrains from MATLAB ASIC design is made in System Verilog that contains FFT, hann window, IFFT, FIR filter. After filtering the same signal using System Verilog design code, the results are compared with MATLAB results which are observed as much cleaner than that from MATLAB. This project can be further extended in designing a generalized FIR filter that is suitable for any type of input with noise.

10. References

[1] "Part 7: The 2018 Wilson Research Group Functional Verification Study." *Verification Horizons BLOG RSS*, blogs.mentor.com/verificationhorizons/blog/2019/01/22/part-7-the-2018-wilson-research-group-functional-verification-study

[2] 2019, https://en.wikipedia.org/wiki/The_Hum, accessed 10 May 2019

[3] 2019, https://www.crystalinstruments.com/dynamic-signal-analysis-basics, accessed 10 May 2019

[4] 2018, http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html, accessed 10 May 10, 2019

[5] 2019, http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html. accessed 10 May 2019.

[6] 2017, https://electronics.stackexchange.com/questions/15206/iir-filters-what-does-infinite-mean, accessed May 10, 2019.

[7] 2019, https://community.plm.automation.siemens.com/t5/Testing-Knowledge-Base/Window-Correction-Factors/ta-p/431775, accessed May 10, 2019

[8] 2019, https://community.plm.automation.siemens.com/t5/Testing-Knowledge-Base/Window-Correction-Factors/ta-p/431775, accessed May 10, 2019

[9] 2019, http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html. Accessed 10 May 2019.

[10] 2019, http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html. Accessed 10 May 2019.

[11] Petrakieva, Simona, Oleg Garasym, and Ina Taralova. "http://ieeexplore. ieee. org/stamp/stamp. jsp? tp= &arnumber= 7038771." (2014).

[12] Y. Ephraim and D. Malah "Speech enhancement using a minimum-mean square error

[13] short-time spectral amplitude estimator" Acoustics Speech and Signal Processing IEEE Transactions on vol. 32 no. 6 pp. 1109-1121 1984.

[14] R. McAulay and M. Malpass "Speech enhancement using a soft-decision noise suppression filter" Acoustics Speech and Signal Processing IEEE Transactions on vol. 28 no. 2 pp. 137-145 1980. (Pubitemid 10487373)

[15] P. J. Wolfe and S. J. Godsill "Efficient alternatives to the Ephraim and Malah suppression rule for audio signal enhancement" EURASIP Journal on Advances in Signal Processing vol. 2003 no. 10 p. 910167 2003.

[16] R. Martin "Speech enhancement based on minimum meansquare error estimation and supergaussian priors" Speech and Audio Processing IEEE Transactions on vol. 13 no. 5 pp. 845-856 2005. (Pubitemid 41558900)

[17] S. Doclo and M. Moonen "On the output SNR of the speechdistortion weighted multichannel Wiener filter" Signal Processing Letters IEEE vol. 12 no. 12 pp. 809-811 Dec. 2005. (Pubitemid 41800439)

[18] T. Den Bogaert J. Wouters S. Doclo and M. Moonen "Binaural cue preservation for hearing aids using an interaural transfer function multichannel Wiener filter" in ICASSP 2007 Proceedings vol. 4 April pp. IV-565-IV-568.

[19] B. Cornelis M. Moonen and J. Wouters "Performance analysis of multichannel Wiener filter-based noise reduction in hearing aids under second order statistics estimation errors" Audio Speech and Language Processing IEEE Transactions on vol. 19 no. 5 pp. 1368-1381 2011.

Appendix

MATLAB code for filtering “input.wav”:

```
clear all;
clc;
close all;


% *********FETCHING INPUT (FIGURE 1)*********
inputSignal = audioread('Input.wav');
expectedSignal = audioread('Expected.wav');


% *********INITIALIZATION*********
fs=22050;               %Samplng Rate of sound = 22050 samples/sec
BS = 4096;
pkg load signal
outputSignal(1:length(inputSignal))=0;

if(false)
  figureCOunt=2;
  generateGraph('input Signal',inputSignal,fs,figureCOunt);
  generateGraph('expected Signal',expectedSignal,fs,figureCOunt);
end

%sound(inputSignal,fs);
%sound(expectedSignal,fs);

generateplot=true;
loopEnd=(length(inputSignal)-mod(length(inputSignal),BS))
% *********LOOPING THROUGH (FIGURE 2)*********
for i=1:(BS/2):loopEnd
%i=1;
    % *********SELECTING A RANGE OF INPUT SIGNAL*********
    if((i+BS-1)<=length(inputSignal))
      selectedRange  =   inputSignal(i:i+BS-1);
      windowLength = BS;
    else
      selectedRange  =   inputSignal(i:length(inputSignal)-1);
      windowLength = length(inputSignal)-i;
    end
    % *********APPLYING WINDOWING FUNCTION*********

    window=hanning(windowLength);
    windowedSignal   =   selectedRange.* window;

    % *********Filter Design*********

    f1 = 500;
    f2 = 750;
    delta_f = f2-f1;
    dB  = 40;
```

```
    N = dB*fs/(22*delta_f);

    f =  [f1 ]/(fs/2);
%    hc = fir1(round(N)-1, f,'low');
    hc = fir1(900,0.05,'low');
%    [b,a]=butter ( 1, (50*2) / fs );

    % *********APPLYING FILTER*********

    filteredResponse      =   filter(hc,1,selectedRange);


    % *********APPLYING INERSE FOURIER TRANSFORM*********


    slicedOutput  = filteredResponse(BS/4:3*(BS/4));

    % *********ADDING *********
    outputSignal(i:i+BS/2-1)=slicedOutput(1:BS/2);

   if(false)
      figureCOunt=5;
      generateGraph('selected Range',selectedRange,fs,figureCOunt);
      generateGraph('windowed Signal',windowedSignal,fs,figureCOunt);
      generateGraph('filtered
Response',filteredResponse,fs,figureCOunt);
      generateGraph('sliced Output',slicedOutput,fs,figureCOunt);
      generateGraph('output Output',outputSignal,fs,figureCOunt);
   end
netx=1;

end

if(true)
  figureCOunt=3;
  generateGraph('Input Signal',inputSignal,fs,figureCOunt);
  generateGraph('Expected Signal',expectedSignal,fs,figureCOunt);
  generateGraph('output Signal',outputSignal,fs,figureCOunt);
end

if(true)
  #sound(inputSignal,fs);
  #sound(expectedSignal,fs);
  sound(outputSignal,fs);
  audiowrite('output.wav',outputSignal,fs);
  outputSignal = audioread('output.wav');%wave file
  #sound(outputSignal,fs);
end
```

MATLAB code to generate graphs:

```
function generateGraph(name,signal,fs,figureCOunt)
    persistent n;
    if isempty(n)
        n = 0;
        num=0;
    end
    if(n==0)
      figure()
    end
    num=n*2+1;
    lengthOfSignal=length(signal);
    subplot(figureCOunt,2,num)
  %TIME PLOT
    t = 0:lengthOfSignal-1;
    plot(t,signal);title(['Time Domain '
name]);xlabel('Time,s');ylabel(name);
  %FREQUENCY PLOT
    subplot(figureCOunt,2,num+1)

    %{
    fftSelectedSignal = fft(signal,lengthOfSignal);
    fftSelectedSignal =
fftshift(abs(fftSelectedSignal/lengthOfSignal))(1:(lengthOfSignal)/2+1)
;
    fftLength=length(fftSelectedSignal);
    fftSelectedSignal(2:fftLength-1)=2*fftSelectedSignal(2:fftLength-
1);
    f = fs*(0:(lengthOfSignal/2))/lengthOfSignal;
    plot(f, fftSelectedSignal);
    %axis ([4000 8000 0 1000]);
    %}

    plot((-0.5:1/lengthOfSignal:0.5-
1/lengthOfSignal)*fs,20*log10(abs(fftshift(fft(signal,lengthOfSignal)))
))
    title(['Frequency Domain '
name]);xlabel('frequency,f');ylabel(name);
    %axis ([-2000 2000 -100 100]);
    n=n+1;
    if(n==figureCOunt)
      n=0;
    end
end
```

FFT design:

```
`include "fftCalculator.v"

module fft();

reg clk,reset;
//dataMem

reg startin,startout;
integer data_file;
integer scan_file;
real captured_data,realin,imagin, realout, imagout;
fftCalculator
obj(clk,reset,realin,imagin,startin,startout,realout,imagout);


always @(posedge clk or posedge reset) begin

    if(reset) begin


    end
    else begin
       scan_file = $fscanf(data_file, "%f\n", captured_data);
        $display(scan_file);
        realin<=#1 captured_data;
        imagin<=#1 0.0;
    end
end

initial begin
    data_file = $fopen("inputData.txt", "r");
    clk=0;
    reset=1;
    #5
    reset=0;
    repeat(50) begin
        clk=#1 ~clk;
    end
end

initial begin
    $dumpfile("fft.vpd");
    $dumpvars();
end

endmodule
```

final report.pdf

by Lekhya Tata

Submission date: 14-May-2019 09:21AM (UTC-0700)
Submission ID: 1130388083
File name: 0190514-9875-1x5gl7a_attachment_5403639420190514-9875-cycgb6.pdf (3.56M)
Word count: 6351
Character count: 39274

NOISE CANCELLATION USING WINDOWING TECHNIQUE

by

Lekhya Tata

Karthik Govinda Raju

Abstract

Motivation:

Many methods have been implemented for noise cancellation in audio signals. Mostly in software like MATLAB. Filters that are already in Verilog are confined to only small class of noise removal and no study proves that each method is completely efficient.

Tasks:

Project aim is to filter audio signal with noise in MATLAB and get the results. By taking MATLAB results as a reference, filter is designed in System Verilog for the audio file with constrains that are used in MATLAB code. Compare both the results. To get faster results and reduce hardware mathematical computations in filter are improved. FFT, windowing and IFFT are used to filter noise signal.

Significance:

There is no formal study that shows which transforms method works better, that are employed in both speed and accuracy. The results of the design in this project are much cleaner than the MATLAB result

Table of Contents

List of figures

List of tables:

1. Introduction

ASIC chips are leading in the industry of semiconductor technology because they are working at a quicker rate and as a result for better operation and task involving great technology, design is becoming more complex. It includes many challenges and trade-offs.

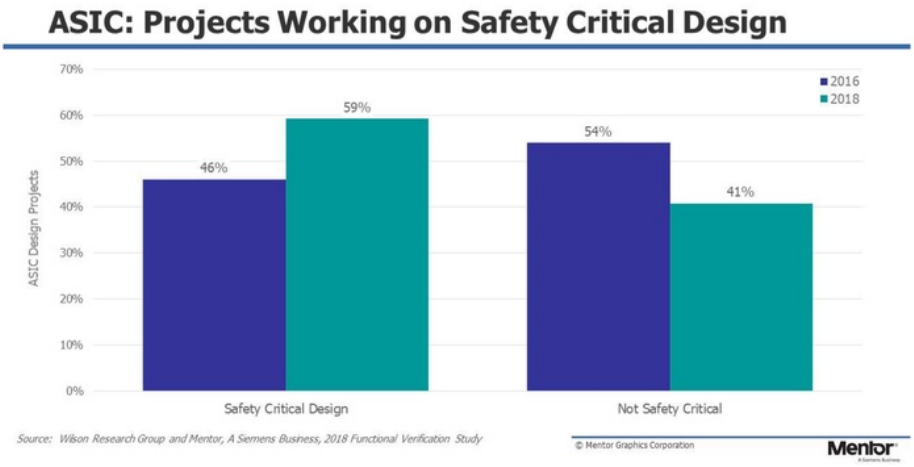


Figure 1. Importance of ASIC projects [1]

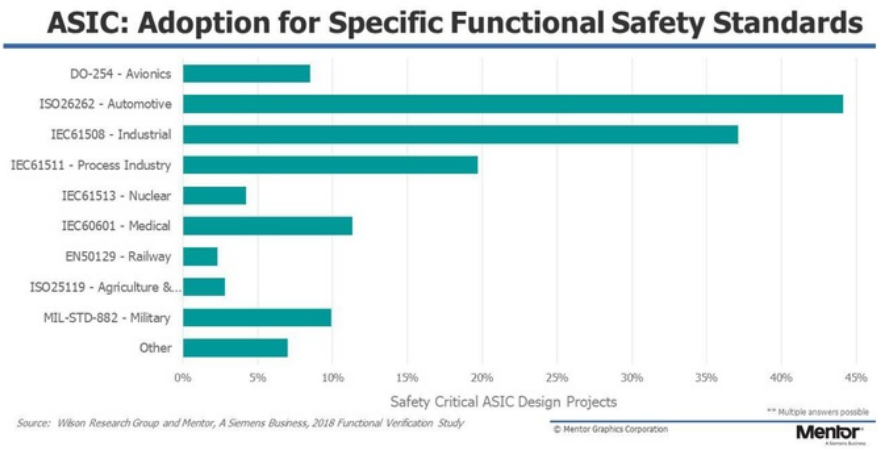


Figure 2. Role of ASIC adoption in various industries [1]

From the figure 2, it is clear, that ASIC is leading in various industries. Medical is the fourth leading industry. This project involves noise removal in the ultrasound of fetal heartbeat.

Aim of the project is to design a filter that performs better and reduce maximum noise in a faster way. Generally, noise can be eliminated in time domain and frequency domain. In this project design is made for frequency domain as speech and noise signals may be better separated in that space, which enables better filter estimation and noise reduction performance. Noise can be categorized in many ways. Additive noise (white noise) is selected for filtering.

When the audio signal with noise is been fed to MATLAB and cut off frequency of noise signal is known from magnitude spectrum plot. This cutoff frequency is used to calculate digital cut off using sampling frequency and analog cut off frequency.

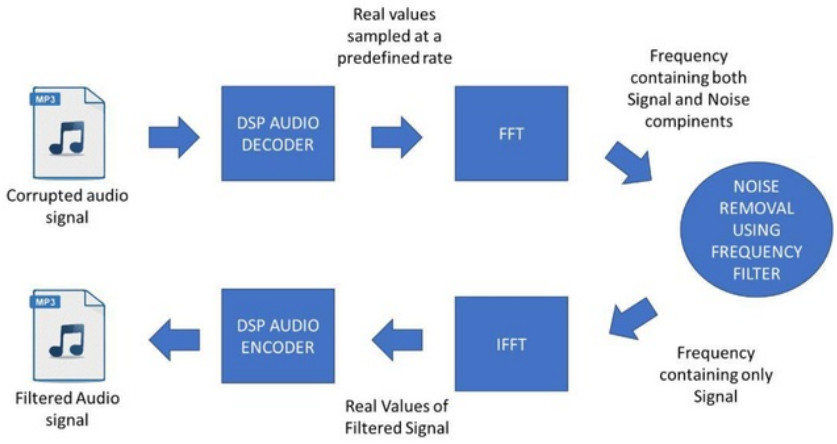


Figure 3. Flow of the project

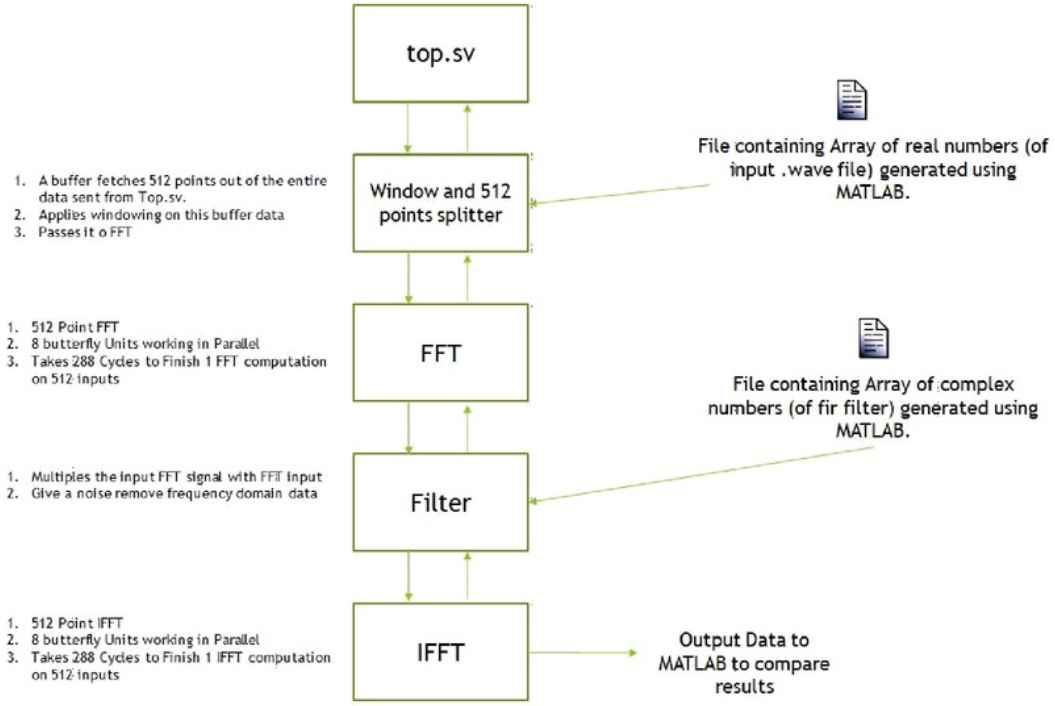


Figure 4. Flow of the project in System Verilog

2. Noise

Noise comes in all shapes, sizes and sounds. This is project concentrates on noise in the form of sound. Sound wave can be broken down to two fundamental characteristics, frequency and amplitude Any unwanted sound that is added to the original audio file is considered as a sound noise. During audio recording, noise gets included in Analog tapes or digital recordings. Among many files section of "hiss" noise found throughout the recording. Addition of noise into the recording also depends on the environment of recording and equipment. Below is the list of four major kinds of audio noises.

2.1. Hum

The sound which is at low frequency than the actual sound and whose frequency ranges between 40 – 80hz [2] is considered as hum. For example, whirring of low-pitched motor. Hum usually occurs by electrical interferences or the ground wiring of recording equipment is not done properly.

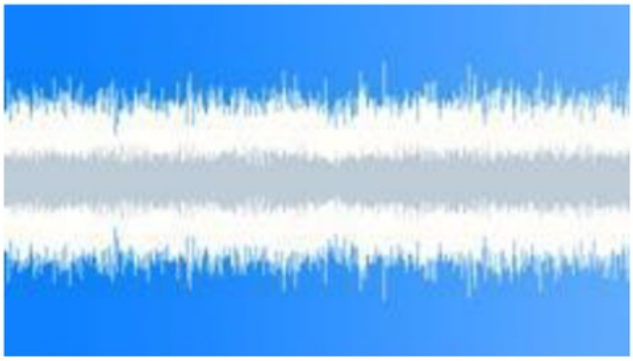


Figure 5. Low frequency hum noise Hiss

This can be seen in devices that includes electronic components. Due to rise in temperature (compared to room temperature) path of electrons in the device is deviated as a results output is disturbed. This disturbance in the distorted output is considered as "hiss". Hiss noise depends on the quality of recording equipment. This noise can also be a result of environmental factors like A/C, fans etc.,

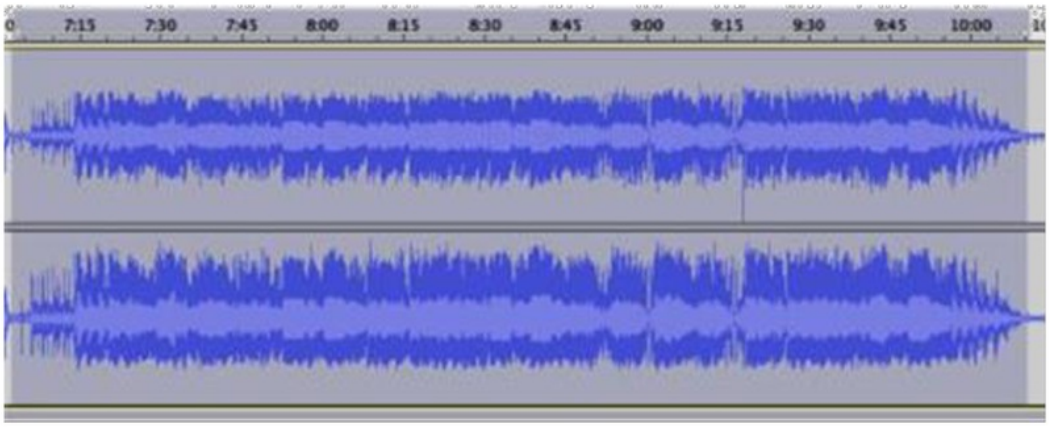


Figure 6. Hiss noise plot

2.2. Rumble

This is very common audio noise. Like hum noise this is also at low frequency. This can be seen only between specific points in a given bandwidth. It is most observable noise in mechanical devices. For example, ball bearings at a joint, gear rotation in a gear box, mechanical parts moving in the audio recording equipment creates this noise. There are filters that are inbuilt in a device to eliminate this kind of noise.

Figure 7 Rumble noise

2.4. Crackle

These are kind of discontinuous and non-musical. This kind of noise is like the sound heard during burning of wood. This is caused by explosion of air pockets inside the object. These are of two types fine crackles which are high pitched and last for less duration of time, coarse crackles which are of low pitched and last for longer duration.

Noise can be categorized by giving color names. This project involves an audio file that contains "white noise". White noise generally comes to hiss noise category. It is present everywhere in the entire audio file. To put entire concept in a nutshell, it is a layer of sound that is considered as noise floor while processing it.

Figure 8 Gramophone crackling noise

2.5. White noise

White noise can be stated as the noise that contains many frequencies. sometimes white noise is used to mask the other unwanted frequencies.

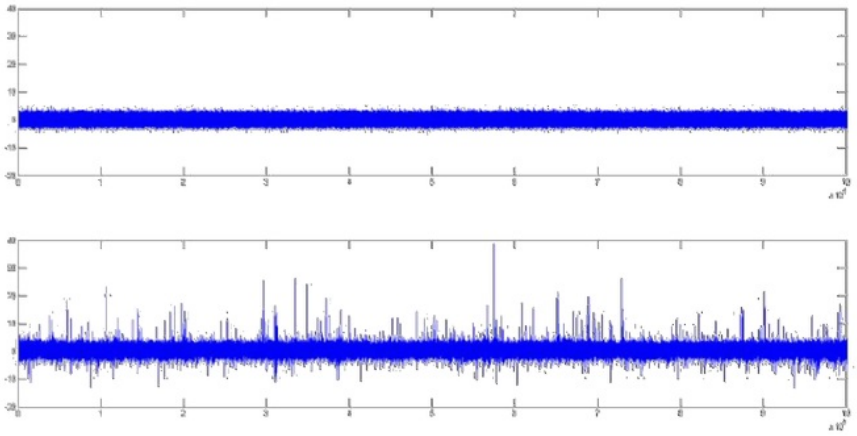


Figure 9. Plot showing white noise in an audio file

3. Filters

The main function of a filter is to pass desired frequencies and attenuate the unwanted frequencies. The four primary types of filters that can be listed down as

- Low-pass filter
- High-pass filter
- Band-pass filter
- Notch filter (band-stop filter)

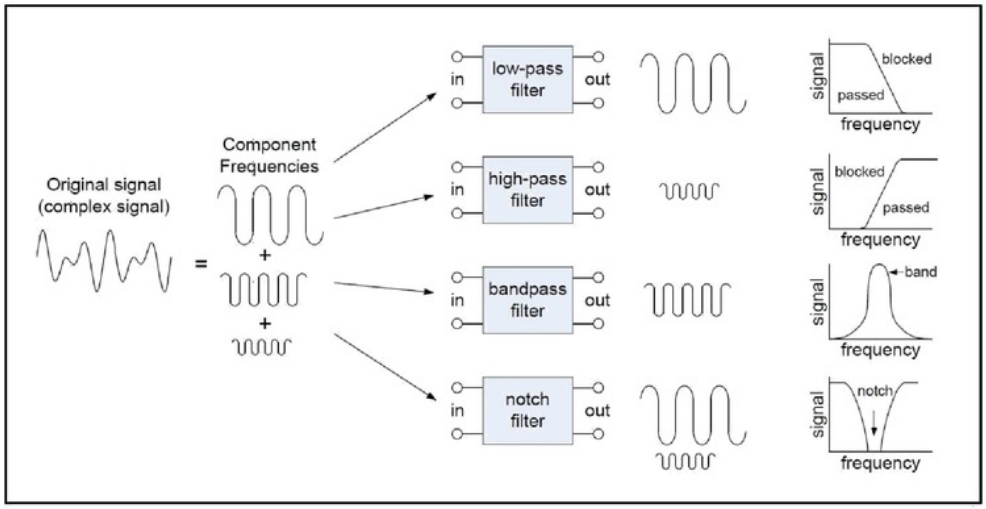


Figure 10. A basic depiction of four major filter types

There are two major digital filters. FIR filters and IIR filters.

3.1. FIR filter

The Fourier filter is a type of filtering function that is based on manipulation of specific frequency components of a signal. It works by taking the Fourier transform of the signal, then attenuating or amplifying specific frequencies, and finally inverse transforming the result. The main advantage of FIR filter is that they can easily be designed to be "linear phase". FIR filter works better with 2^N sample points. Stability of FIR filter is very good which can also be framed as finite output can be obtained for every finite input.

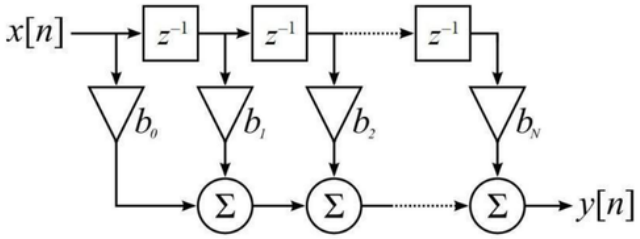


Figure 11. Butterworth filter representation [4]

$$y[n] = \sum_{k=0}^{N-1} h[k] \cdot x[n-k]$$

FIR filter formula [5]

3.2. IIR filter

Infinite impulse response filter. This filter is infinite because of presence of feedback in the filter. Desired filtering characteristics can be achieved using IIR filter consuming low memory and calculations when compared to an FIR filter. The main disadvantage is that implementation of filtering using fixed point arithmetic becomes slower and harder. They don't offer the computational advantages of FIR filters for multi rate applications.

FIR filters are preferable over IIR filters as the response is a linear phase and non-recursive. IIR filter performance is good when dealing with analog filter responses. This project uses FIR filter for the input wave file selected.

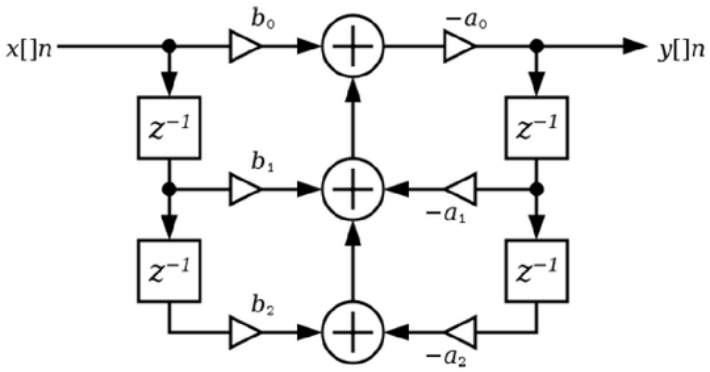


Figure 12. IIR Filter Z Transform [6]

4. Windowing

In most of the signal measurements, the signal doesn't contain integer number of periods. Due to this finiteness is not perfect and result in loss of some part of waveform and characteristics of whole signal changes. This finiteness introduces sharp transition changes in the signal that is measured.

When the waveform's period is non-integer, discontinuities are seen at the endpoints. These unwanted discontinuous components become high frequency harmonics in FFT which are not actually present in the original signal. The frequency of these unwanted components is higher than the Nyquist frequency value. Because of thse distortion the FFT spectrum is not the actual spectrum of actual signal. It appears as if there is a leakage of energy from one frequency component to other frequency component.

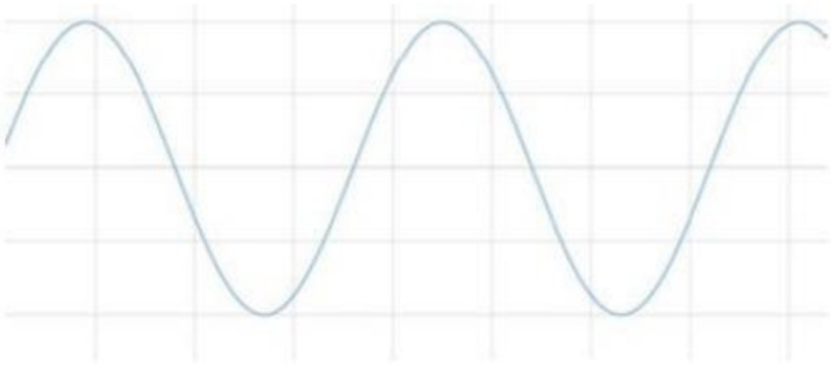


Figure 13. Signal with non-integer number of periods

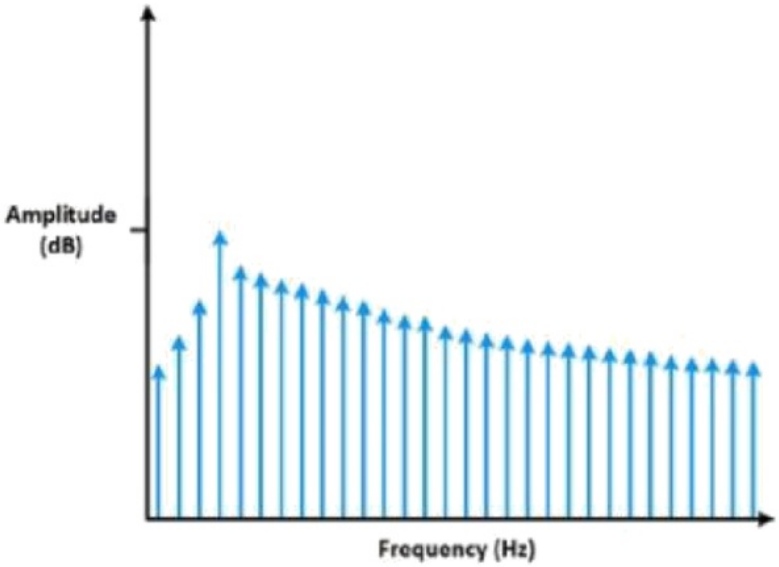


Figure 14. Spectral leakage to the FFT of signal in the figure

In a signal that contains non-integer periods, this distortion in FFT can be minimized by using a technique called "windowing". Windowing basically reduce the amplitude of these discontinuities at starting and ending and gradually rolls of to zero of given boundary for a finite length.

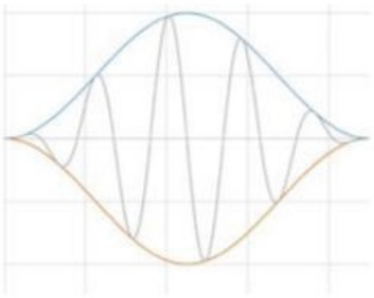


Figure 15. Result of windowing

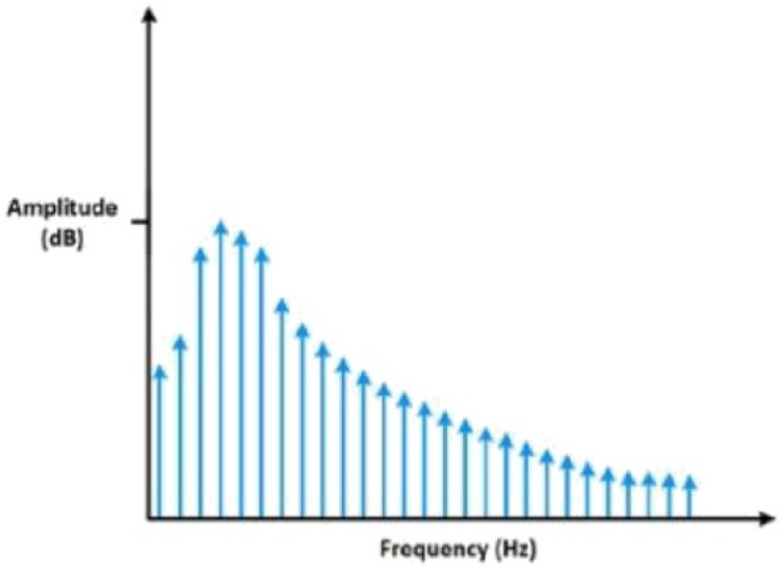


Figure 16. Applying a window minimizes spectral leakage.

There are several windowing mechanisms that are possible like the hann window, hamming window and so on. The below window is used in our project as it has a better damping characteristic as desired for the FFT design.

4.1. Hann window

This window is used for a signal whose characteristics are unknown. Instead of doing tradeoff between amplitude and frequency accuracy a good compromise is provided between the both by using this windowing technique.

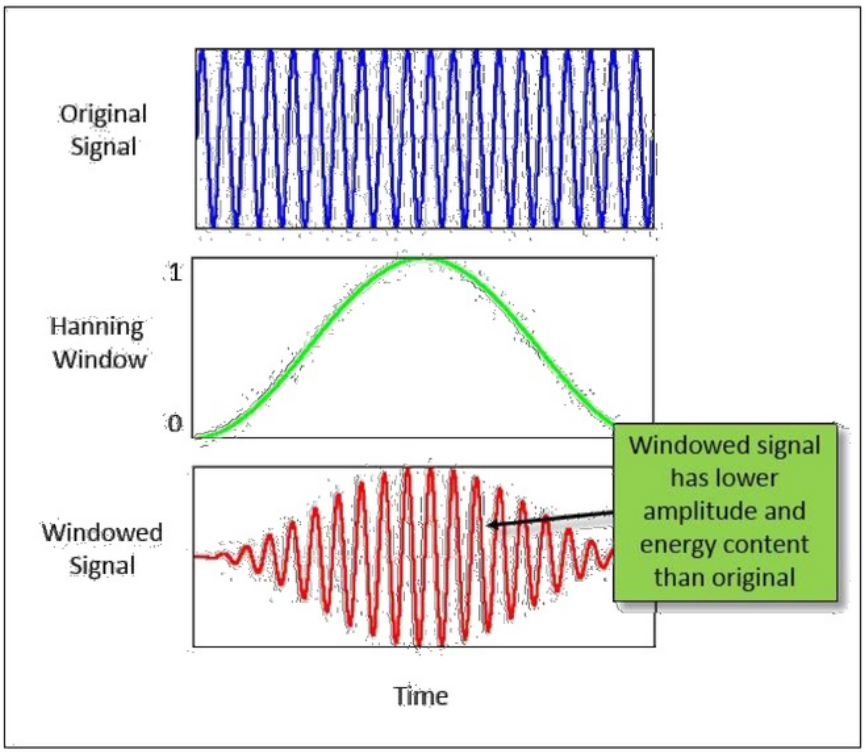


Figure 17. Multiplying original signal by hann window reduces the amplitude and energy in the signal [7]

5. MATLAB

This software is excessively used for computing. Integration of computation, visualization and programming is possible in this platform.

MATLAB software environment is user friendly. Mathematical notation type is used to express all kinds of problems and solutions. Main usage of MATLAB is application development. It makes usage of signal processing techniques easy by providing many predefined functions. It provides unified work flow for the development of embedded systems and streaming applications.

MathWorks provides design apps, DSP algorithm libraries, and I/O interfaces for real-time processing of streaming signals in MATLAB and Simulink. You can rapidly design and simulate streaming algorithms for audio, video, instrumentation, smart sensors, wearable devices, and other electronic systems

Entire filter design is first made in MATLAB and checked for output. Based on the results obtained cutoff frequency is determined and it is used in filter designing in Verilog.

5.1. Generating plots

The Generation of Graphs are critical part of simulation as it presents the expected output in a graphical manner for ease of understanding. In our scenario, we need to generate graphs for time and frequency domain at every point in simulation. Such as

1. When the input signal arrives
2. When the signal is chopped into 512 bits
3. After windowing
4. Post FFT and Filtering
5. After IFFT
6. Once the whole signal of merged data is obtained back

The below code presents two subplots, one for Time and the other for frequency. The frequency domain x axis is calculated as below

```
(-0.5:1/lengthOfSignal : 0.5-1/lengthOfSignal)*fs
```

The reason is because the signal has negative and positive axis corresponding to imaginary parts of the FFT output

The other parts of the code are self-explanatory.

* generateGraph.m

```
function generateGraph(name,signal,fs,figureCOunt)
    persistent n;
    if isempty(n)
        n = 0;
        num=0;
    end
    if(n==0)
      figure()
    end
    num=n*2+1;
    lengthOfSignal=length(signal);
    subplot(figureCOunt,2,num)
  %TIME PLOT
    t = 0:lengthOfSignal-1;
    plot(t,signal);title(['Time Domain ' name]);xlabel('Time,s');ylabel(name);
  %FREQUENCY PLOT
    subplot(figureCOunt,2,num+1)
    plot((-0.5:1/lengthOfSignal:0.5-1/lengthOfSignal)*fs,20*log10(abs(fftshift(fft(signal,lengthOfSignal)))))
    title(['Frequency Domain ' name]);xlabel('frequency,f');ylabel(name);
    %axis ([-2000 2000 -100 100]);
    n=n+1;
    if(n==figureCOunt)
      n=0;
    end
end
```

Figure 18. Code for generating plots

To maintain good clarity in the actual code, function "generategraph" is called into actual code. This gives plots in both time domain and frequency domain. Input signal is taken in the variable "signal" and it is plotted in time domain. After processing in the final stage frequency plot of filtered signal is obtained. True or false is set to generate graph function according the requirement. True is to generate plot and false is not to access the function.

5.2. Initialization

In the "audioread" function is used to fetch the input signal which is ultrasound heartbeat of a fetus. The input is in the wave format. Sampling frequency is given as 22050 hz. BS (512) is the block size for which the filtering is performed. This is nothing but 512 point FFT which is further designed in Verilog. The length of input file is 13782600.

```
clear all;
clc;
close all;


% *********FETCHING INPUT (FIGURE 1)*********
inputSignal = audioread('Input.wav');
expectedSignal = audioread('Expected.wav');


% *********INITIALIZATION*********
fs=22050;                %Samplng Rate of sound = 22050 samples/sec
BS = 512;
pkg load signal
outputSignal(1:length(inputSignal))=0;
```

Figure 19. Code for reading input and setting FFT points

```
generateplot=true;
loopEnd=(length(inputSignal)-mod(length(inputSignal),BS))

```

Figure 20. For loop constraints

```
% *********LOOPING THROUGH (FIGURE 2)*********
for i=1:(BS/2):loopEnd
%i=1;
    % *********SELECTING A RANGE OF INPUT SIGNAL*********
    if((i+BS-1)<=length(inputSignal))
      selectedRange  =   inputSignal(i:i+BS-1);
      windowLength = BS;
    else
      selectedRange  =   inputSignal(i:length(inputSignal)-1);
      windowLength = length(inputSignal)-i;
    end
```

Figure 21. Looping Through Each 512-point chunk

A for loop is made which includes which gives the portion of input signal which starts from 1 and end at 512. The next block starts from 512. For loop constraints are selected in such a way that only one fourth portion of first block sized input signal is removed and last one fourth portion of last block sized input signal. Due to application of hann window on the input signal, in everyone block sized input signal first one fourth and last one fourth signal is removed to reduce the spectral leakage. This leakage is considered as spectral leakage. Due to this lot of input signal is removed and there is a chance of losing the original signal. To avoid such condition, after every block size filtering last one fourth of the signal is added to next block size signal and then rest all computations are performed.

```
% **********Filter Design**********
filter = firl(900,0.05,'low');
fftFilter = fft(filter,BS);
```

Figure 22. Filter design

FIR filter of 900 order and 0.05 roll off which is a low pass filter is used. So, the entire signal length is divided into chunks whose length is of 512.

```
43      % **********APPLYING WINDOWING FUNCTION**********
44
45      window=hanning(windowLength);
46      windowedSignal    =    selectedRange.* window;
47
```

Figure 23. Hann window

Hann window is applied to window length which is of block size (initial size of window length is block size). Windowed signal is magnitude of vector product of selected range of input signal and the window. To perform this operation array and order size of both the elements should match.

```
48      % **********APPLYING FILTER*********
49      fftSignal = fft(selectedRange,BS);
50      fftFilteredOUtput = fftSignal .* transpose(fftFilter);
51      filteredResponse=ifft(fftFilteredOUtput,BS);
52      %filteredResponse       =    filter(hc,1,selectedRange);
53
```

Figure 24. Applying filter

Predefined function “FFT” is used to perform 512 point FFT. This function needs two parameters. “select range” is the range of processed signal and “BS” specifies the number of FFT points. When multiplying FFT signal of FFT of filter transpose is taken to match the order of the two variables.

```
% *********APPLYING INERSE FOURIER TRANSFORM*********

slicedOutput  = filteredResponse(BS/4:3*(BS/4));

 highCutOff=0.1;
 lowCutOff=-0.1;
 multiplicationFactor=1.63;
 reduction=2;
 DCoffset=0.18;

 for j=1:length(slicedOutput)
    slicedOutput1(j,1)=(real(slicedOutput(j)))-DCoffset+(1i*(imag(slicedOutput(j))-DCoffset));
end

for j=1:length(slicedOutput1)
  if(real(slicedOutput1(j))>highCutOff || real(slicedOutput1(j))
  <lowCutOff || imag(slicedOutput1(j))>highCutOff || imag(slicedOutput1(j))<lowCutOff)
    slicedOutput2(j,1)=multiplicationFactor*real(slicedOutput1(j))
    +1i*multiplicationFactor*imag(slicedOutput1(j));
  else
    %slicedOutput2(j,1)=0+1i*0;
    slicedOutput2(j,1)=(1/reduction)*real(slicedOutput1(j))+1i*(1/reduction)*imag(slicedOutput1(j));
  end
end
```

Figure 25. Inverse fourier transform

FFT is performed for the windowed signal and because of windowing, DC offset is introduced, and amplitude of original signal is reduced. So, multiplication factor is selected as 1.63 [3] to bring back the amplitude of filtered signal to the original.

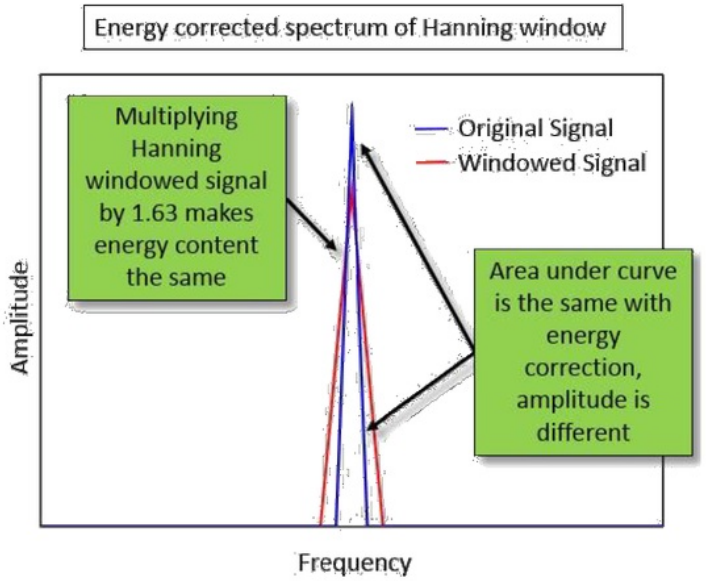


Figure 26. Selection of multiplication factor for hann window [7]

```
80      % **********ADDING **********
81      outputSignal(i:i+BS/2-1)=slicedOutput2(1:BS/2);
82
83      if(false)
84          figureCOunt=6;
85          generateGraph('selected Range',selectedRange,fs,figureCOunt);
86          generateGraph('windowed Signal',windowedSignal,fs,figureCOunt);
87          generateGraph('filtered Response',filteredResponse,fs,figureCOunt);
88          generateGraph('sliced Output',slicedOutput,fs,figureCOunt);
89          generateGraph('sliced Output 1',slicedOutput1,fs,figureCOunt);
90          generateGraph('output Output',outputSignal,fs,figureCOunt);
91      end
92  netx=1;
93
94  end
95
96  if(true)
97      figureCOunt=2;
98      generateGraph('Input Signal',inputSignal,fs,figureCOunt);
99      %generateGraph('Expected Signal',expectedSignal,fs,figureCOunt);
100     generateGraph('output Signal',outputSignal,fs,figureCOunt);
101 end
102
103 if(false)
104     sound(inputSignal,fs);
105     %sound(expectedSignal,fs);
106     sound(outputSignal,fs);
107 end
108
109 if(false)
110     audiowrite('output.wav',outputSignal,fs);
111 end
```

Figure 27. Producing final filtered output

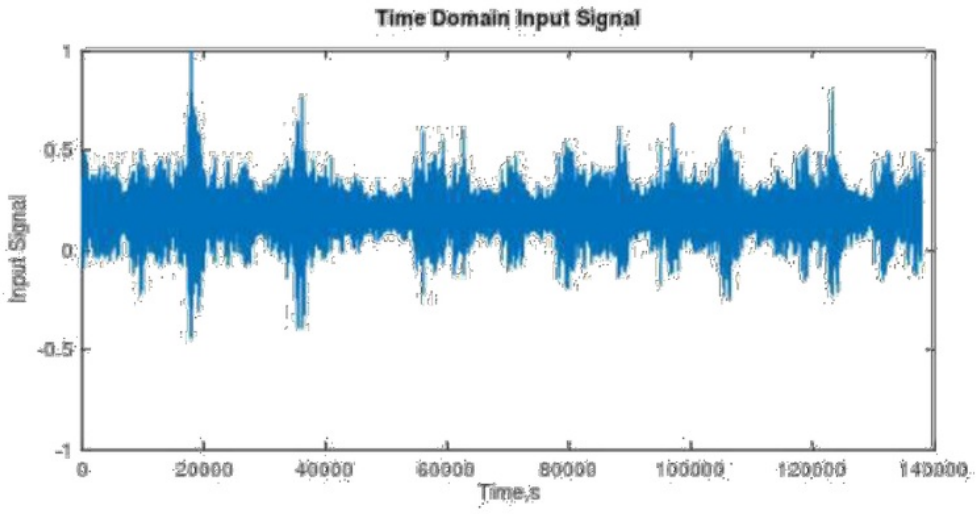


Figure 28. Plot of Input signal in time domain

Input signal plot in time domain. This contains lot of noise whose frequency can't be determined from time domain plot. The spikes indicate heartbeat and as you the signal is low frequency (Occurs at a lesser rate than the noise signal). Main aim is to remove all other frequency components which all are considered as noise components.

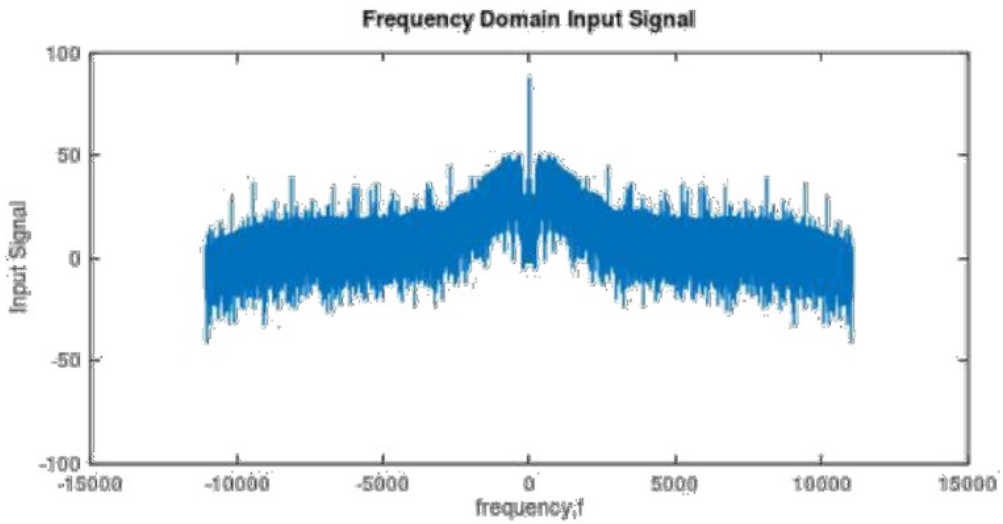


Figure 29. Plot of input signal in frequency domain

In figure 27 the frequency component at zero frequency determines the Doppler heartbeat frequency. The maximum noise floor ranges from -30db to 50db. To eliminate those frequencies low pass FIR filter of order 900 and roll off 0.05 is selected.

Few Considerations for Noise Removal

1. The signal is low frequency. So, we need to design a low pass filter with a cutoff frequency (which is to be selected based on trial and error and brute force method)
2. An assumption of an error on Window to be 50 percent was made. Meaning, if we window 512 samples, only the mid 256 Samples will be valid data. Remaining should be dumped as they will be grounded to zero.

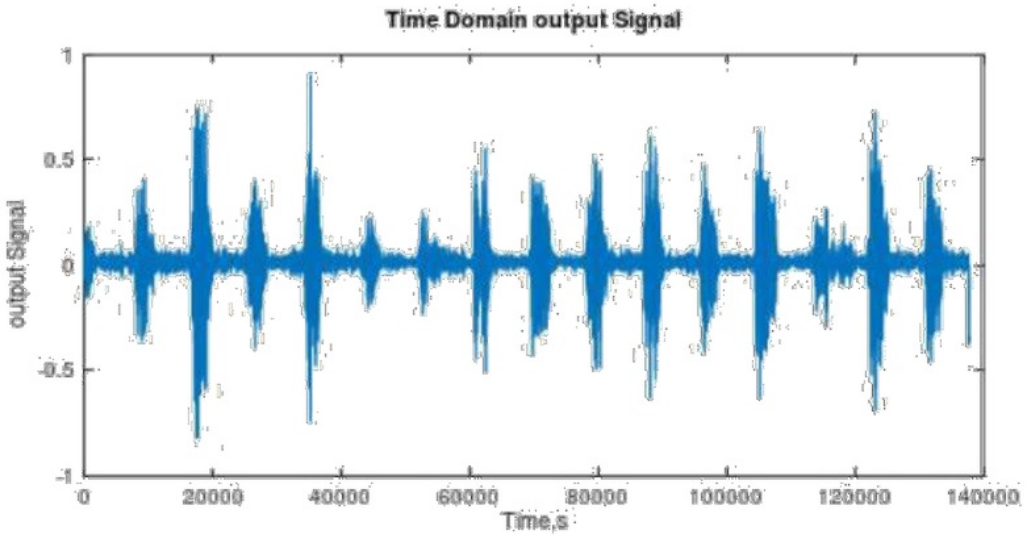


Figure 30. Plot of output signal in time domain

Before getting plot in figure 29, DC offset of 0.18db is observed and amplitude was reduced to nearly 2. So appropriate code is made to overcome that error and the desired output is obtained.

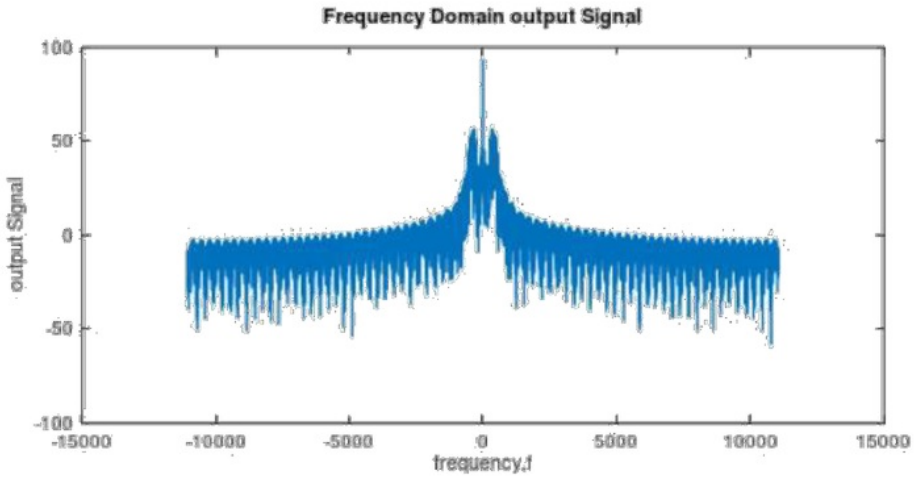


Figure 31. Plot of output signal in frequency domain

The noise floor in the above figure is pushed down to 0db. The output of this is heard and the noise is eliminated.

6. System Verilog

System Verilog is a good platform to provide a detailed design specifications of a digital circuit. Creation of design specifications is trivial when compared to the amount of work required to translate the specification to a schematic based structural description needed to fabricate a device.

Because of system Verilog design engineer's productivity is increased in very few years. CAD tools that are used in digital design can be categorized into two as "front end" tools that allows capturing and simulating the design and "back end" tools that are used to synthesize a design, link it to a technology and analyzes its performance

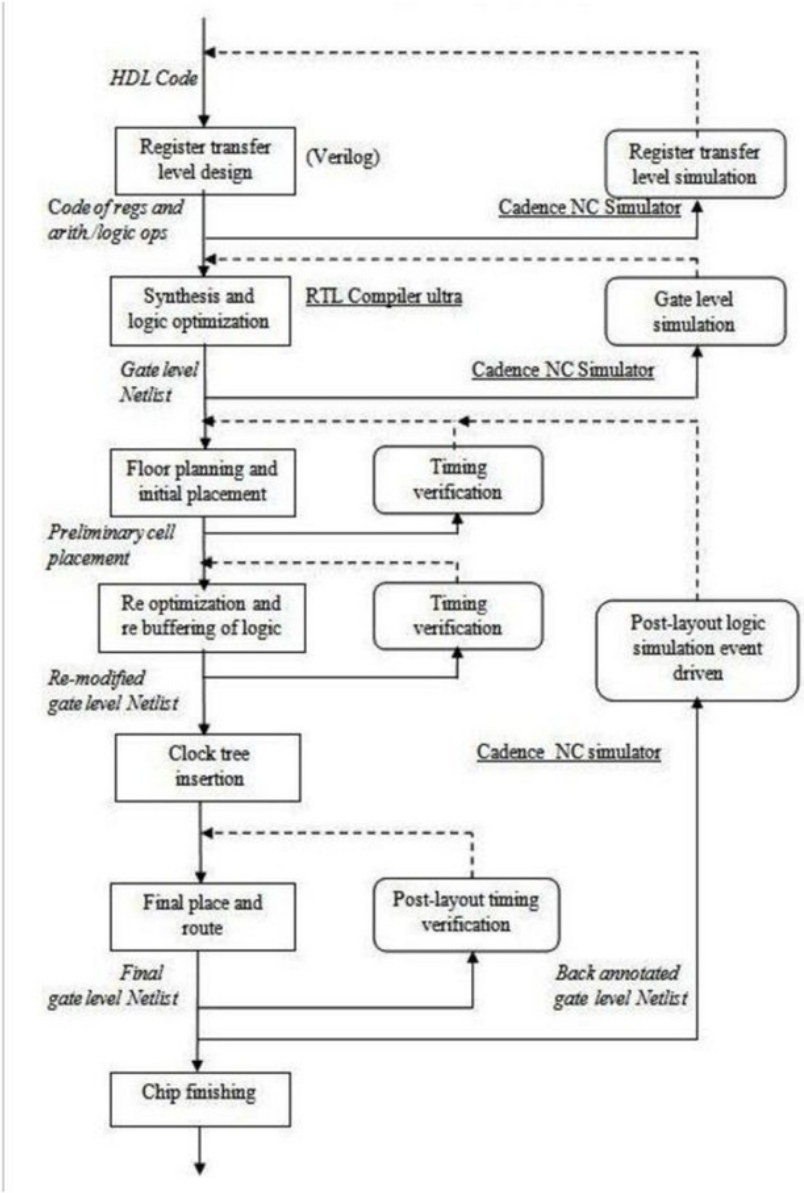


Figure 32. Digital design flow

7. Design

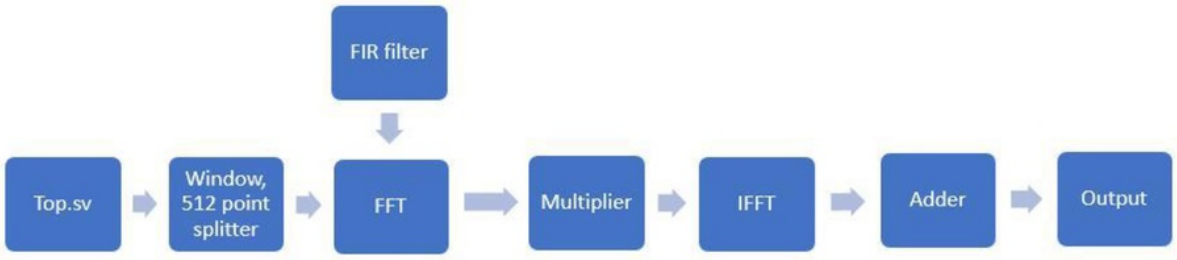


Figure 33. Design flow in System Verilog

After getting filtered results of input signal from MATLAB for ultrasound of heartbeat of fetus, the input.wav file is converted into text file, hann window is converted to text file and fir filter coordinates is converted into text file. All these files are used to perform filtering of input and resultant output values matches with the output from MATLAB.

7.1. Test bench

```
initial begin
    inputDataFile = $fopen("signalInput.txt","r");
    pushin=0;
    rst=1;
    repeat(3) @(posedge(clk)) #1;
    rst=0;
    @(posedge(clk)) #1;
    inputCount=0;
    pushin=1;
    endOfFile=0;
    $display("start");
    while(endOfFile==0) begin
        if(!$feof(inputDataFile)) begin
            if(stopOutWindow==0) begin
                $display(inputCount);
                //readDataReal=$fscanf(inputDataFile,"%f\n",realin);
                readDataReal=$fscanf(inputDataFile,"%f\n",realin);
                imagin= 0;
                inputCount=inputCount+1;
            end
        end
        else
            endOfFile=1;
        @(posedge clk) #1;
```

Figure 34. Fetching input

Input "fetal doppler" is in .wav format. to convert this into digital form, which is a vector representation, MATLAB is used. This is of size 1307800. Audio file in .wav format is easy to convert to digital form. So, any audio file which any other format is first converted into .wav file for representing it in digital format.

After converting input into text file that contains real and imaginary values, at every posedge of the clock real values are read in a variable called "realdata" and imaginary values are read in a variable "imagindata".

```
initial begin
    forever begin
        if(pushoutFFt) begin
            filterRealDataFile = $fopen("filterInputReal.txt","r");
            filterImagDataFile = $fopen("filterInputImag.txt","r");
            while(pushoutFFt) begin

                if(!$feof(filterRealDataFile) && !$feof(filterImagDataFile)) begin
                    if(pushoutFFt) begin
                        //readDataReal=$fscanf(filterDataFile,"%f\n",realin);
                        readDataFilterReal=$fscanf(filterRealDataFile,"%f\n",
                            realinFilter);
                        readDataFilterImagin=$fscanf(filterImagDataFile,"%f\n",
                            imaginFilter);
                    end
                end
                else begin
                    if(pushoutFFt) begin
                        //readDataReal=$fscanf(filterDataFile,"%f\n",realin);
                        realinFilter=0;
                        imaginFilter=0;
                    end
                end
                @(posedge clk) ;
            end
            $fclose(filterRealDataFile);
            $fclose(filterImagDataFile);
        end
        @(posedge clk) ;
    end
end
```

Figure 35. fetching filter data

Like how the real and imaginary parts of input data is read, in the same manner low pass FIR filter vector representation text file is loaded into design using similar code in system verilog. Here FIR filter is of order 900, roll 0ff 0.05 and its a low pass filter. These numbers are selected by trial and error method. The white noise floor is way below the frequency level of required heart beat. So low pass Filter is selected for filtering.

7.2. Windowing

Windowing technique is used to eliminate the spectral leakage in the signals. Usually the signals which are not periodic contains frequency harmonics in FFT which are not actually present in the signal. These are of high frequency components. the signal frequencies seem like as if the energy been transferred from one harmonic to other harmonic. To eliminate this spectral leakage windowing is applied to the signal which rolls down the signal at starting and at ending to zero, this is been performed because spectral leakage is usually observed at the starting and ending of the recorded non-periodic signal. Hann window is used. The hann window's vector representation is obtained using the MATLAB.

Hann window

$$w(k) = 0.5 - 0.5\cos\left(\frac{2\pi k}{N-1}\right)$$

Hann window formula [8]

7.3. FFT

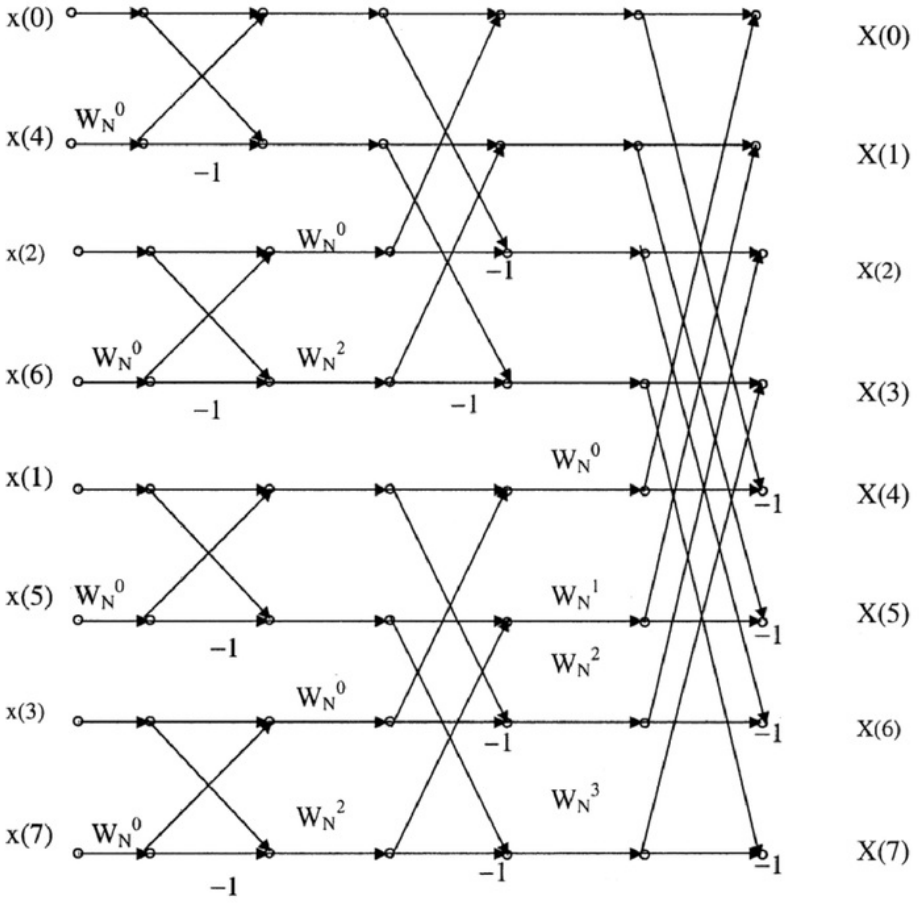
Fast Fourier transform concept is used to convert time domain input signal into frequency domain. If the input to FFT are two to the race any natural number, then it works better. Input signal is a continuous signal in time vs amplitude. In frequency domain signal is plotted between normalized frequency and magnitude response (dB) of signal. The harmonic that is at higher magnitude is determined as heartbeat of fetus.

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}, \qquad k = 0,1,\ldots, N-1$$
$$= \sum_{n \text{ even}} x(n) W_N^{kn} + \sum_{n \text{ odd}} x(n) W_N^{kn}$$
$$= \sum_{m=0}^{(N/2)-1} x(2m) W_N^{2mk} + \sum_{m=0}^{(N/2)-1} x(2m+1) W_N^{k(2m+1)}$$

FFT formula [9]

The formula in above figure is used to compute FFT of filter co-ordinates and hann window co-ordinates. The weight components that are used in computation are called twiddle factors or butterflies. These twiddles contain both real and imaginary values. Real values and imaginary values are of floating-point format and they are saved in arrays in System Verilog.

The data is first saved to input to a buffer which is of size 512. This takes 512 clock cycles. Once this data is transferred, the data is moved completely to a working buffer which is used to compute FFT. The figure 36 shows how data is transferred into input buffer



$W_N^0 = 1$, $W_N^1 = (1-j)/\sqrt{2}$, $W_N^2 = -j$, $W_N^3 = -(1+j)/\sqrt{2}$

```
if(inputDataStartFLag) begin
    reversedAddress= ({inputAddressCounter[0],inputAddressCounter[1],inputAddressCounter[2
        ],inputAddressCounter[3],inputAddressCounter[4],inputAddressCounter[5],
        inputAddressCounter[6],inputAddressCounter[7],inputAddressCounter[8]});
end
else
    reversedAddress=0;
```

Figure 36. Saving 512 inputs to buffer

As it is seen in the figure 36, the input address is reversed and stored in reversed order. The table 1 shows an example for 8-bit FFT (with 3 bits used for representing address values)

Input Count	Binary Count	Reversed count Value	Address in which the data is stored
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Table 1. Address locations of 8bit FFT

In short, the first input is stored to address 1, the second input are stored to address 4 (001 reversed will be 100), third input to address 6 (011 reversed will be 110) and so on.

Now, since the data is stored accordingly, once the calculation starts, data is taken serially from address 1, 2 ,3 till 512. But calculations are performed as seen in figure 37.

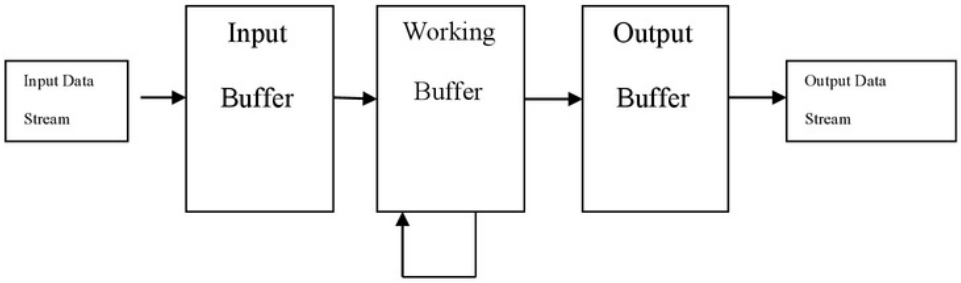
```
always @(posedge clk or posedge rst) begin
    if(rst) begin
        stopout=0;
        realOut=0;
        imagOut=0;
        pushout=0;
        inputCount=0;
        outputCount=0;
    end
    else begin
        if(pushin && !stopout && inputCount<510) begin
            realBuffer[inputCount]<=#1 realInput;
            imagBuffer[inputCount]<=#1 imagInput;
            inputCount<=#1 inputCount+1;
        end
        else if(inputCount>=510) begin
            realBuffer[511]<=#1 realInput;
            imagBuffer[511]<=#1 imagInput;
            stopout<=#1 1;
            inputCount<=#1 0;
        end
        else if(!stopIn && stopout && outputCount<=511) begin
            if(outputCount==0)
                pushout<=#1 1;
            else
                pushout<=#1 0;
            realOut<=#1 realBuffer[outputCount];
            imagOut<=#1 imagBuffer[outputCount];
            outputCount<=#1 outputCount+1;
        end
        else if(outputCount>511) begin
            stopout<=#1 0;
            pushout<=#1 0;
            outputCount<=#1 0;
        end
    end
end
```

Figure 37. Saving 512 inputs to buffer

Figure 36 is a snippet of code which performs the operation of storage and retrieval of data from input buffer and output buffer.

7.4. Buffers

There are in total three buffers acting in a pipelined fashion as seen below



Input Buffer receives data and stores them in reversed addresses as explained before. Once we get all 512 points, the data is moved to working buffer. The data from this buffer is sent to butterflies (16 at a time) to 8 butterflies working in parallel. Once the computation is done, the data is put back to the same working buffer such that they overlap the data which is already used.

```
Butterfly_fft buff1(lastStageFlag,clock,reset,pushinB0,pushoutB0,realIp1B0,realIp2B0,imgIp1B0,
    imgIp2B0,twiddleFactorRealB0,twiddleFactorimgB0,stateIp1B0,stateIp2B0,ReOp1B0,ReOp2B0,imgOp1B0
    ,imgOp2B0,stateOp1B0,stateOp2B0,lastStageFlagOp);

Butterfly_fft buff2(lastStageFlag,clock,reset,pushinB1,pushoutB1,realIp1B1,realIp2B1,imgIp1B1,
    imgIp2B1,twiddleFactorRealB1,twiddleFactorimgB1,stateIp1B1,stateIp2B1,ReOp1B1,ReOp2B1,imgOp1B1
    ,imgOp2B1,stateOp1B1,stateOp2B1,lastStageFlagOp);

Butterfly_fft buff3(lastStageFlag,clock,reset,pushinB2,pushoutB2,realIp1B2,realIp2B2,imgIp1B2,
    imgIp2B2,twiddleFactorRealB2,twiddleFactorimgB2,stateIp1B2,stateIp2B2,ReOp1B2,ReOp2B2,imgOp1B2
    ,imgOp2B2,stateOp1B2,stateOp2B2,lastStageFlagOp);

Butterfly_fft buff4(lastStageFlag,clock,reset,pushinB3,pushoutB3,realIp1B3,realIp2B3,imgIp1B3,
    imgIp2B3,twiddleFactorRealB3,twiddleFactorimgB3,stateIp1B3,stateIp2B3,ReOp1B3,ReOp2B3,imgOp1B3
    ,imgOp2B3,stateOp1B3,stateOp2B3,lastStageFlagOp);

Butterfly_fft buff5(lastStageFlag,clock,reset,pushinB4,pushoutB4,realIp1B4,realIp2B4,imgIp1B4,
    imgIp2B4,twiddleFactorRealB4,twiddleFactorimgB4,stateIp1B4,stateIp2B4,ReOp1B4,ReOp2B4,imgOp1B4
    ,imgOp2B4,stateOp1B4,stateOp2B4,lastStageFlagOp);
```

Figure 38. Butterfly units running in parallel

This process of computation and overlapping goes on until the final state of state machine is reached. During the final state, the output is directly written to output buffer. Once all 512 data are written, output flag is set high and data is sent in a streamlined fashion.

```
else if(inputBufferFlag) begin
    for(i=0;i<512;i=i+1) begin
        realWorkingBuffer[i] <= #1 realInputBuffer[i];
        imagWorkingBuffer[i] <= #1 imagInputBuffer[i];
    end
    inputBufferFlag<=#1 0;

        else if(computationFlag) begin
            if(state==0)
                state<=#1 1;
            else if(state==288) begin
                state<=#1 0;
                computationFlag<=#1 0;
                lastStageFlag<=#1 0;
            end
            else
                state<=#1 (state+1);
```

Figure 39. Transferring data to working buffer and increment states of FFT calculation thereafter

The code in figure 39 snippet shows the input to working buffer and incrementing states of FFT. Each state of FFT contains inputs to 8 buffers and twiddles to these butterflies. All these states are generated using a Java program which is written such that the entire states are generalized. On changing constants to 1024, the states for 1024-point FFT can also be generated.

7.5. IFFT

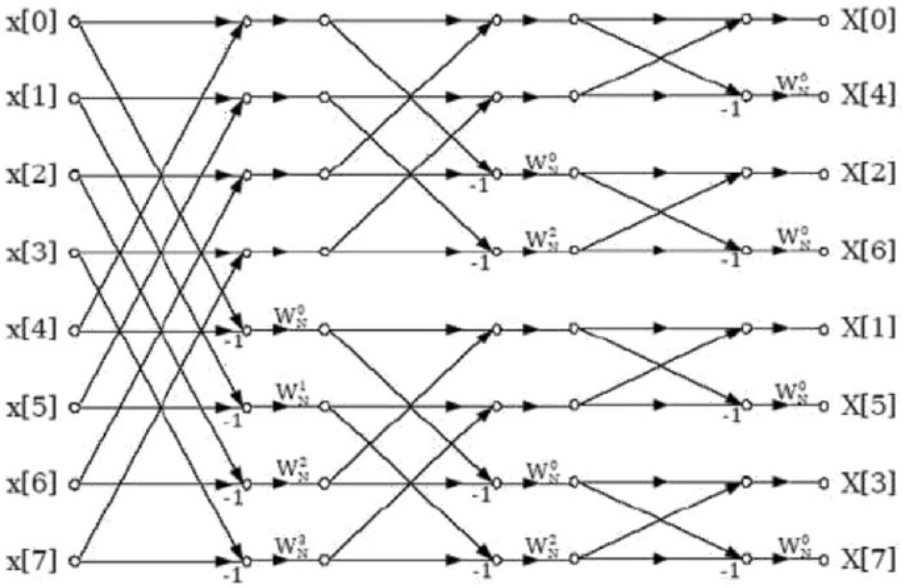
Inverse Fast Fourier transform concept is used to convert frequency domain input signal into time domain. If the input to IFFT are two to the race N, then it works better. Input signal is a Frequency Signal vs Frequency spectrum. In frequency domain signal is plotted between normalized frequency and magnitude response (dB) of signal. The harmonic that is at higher magnitude is determined as heartbeat of fetus.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N} \qquad k = 0, \ldots, N-1,$$

IFFT formula [10]

The formula in above figure is used to compute IFFT of Frequency response of filtered signal. The weight components that are used in computation are called twiddle factors or butterflies. These twiddles contain both real and imaginary values. Real values and imaginary values are of floating-point format and they are saved in arrays in system Verilog.

Data is saved in input buffer, which is of size 512. This takes 512 clock cycles. Once this data transferred, the data is moved completely to a working buffer, which is used to compute FFT. The figure 40 shows how data is transferred into input buffer




```
else if(inputBufferFlag) begin
    for(i=0;i<512;i=i+1) begin
        realWorkingBuffer[i] <= #1 realInputBuffer[i];
        imagWorkingBuffer[i] <= #1 imagInputBuffer[i];
    end
    inputBufferFlag<=#1 0;

        else if(computationFlag) begin
            if(state==0)
                state<=#1 1;
            else if(state==288) begin
                state<=#1 0;
                computationFlag<=#1 0;
                lastStageFlag<=#1 0;
            end
            else
                state<=#1 (state+1);
```

Figure 40. Transferring data to working buffer and increment states of FFT calculation thereafter

Once the data is transferred to working buffer the states of FFT are incremented as shown below. Each state of FFT contains inputs to 8 buffers and twiddles to these butterflies. All these states are generated using a java program which is written such that the entire states are generalized. On changing constants to 1024, the states for 1024-point IFFT can also be generated. The code in figure 41, shows the first state of the entire 288 states.

```
always@(*) begin
      M1_d=ReIp2*twiddleFactorReal;
      M2_d=ReIp2*twiddleFactorComplx;
      M3_d=CmpIp2*twiddleFactorReal;
      M4_d=CmpIp2*twiddleFactorComplx;

      S1_d<=M1-M4;
      S2_d<=M2+M3;

      ReOp1_d<=ReIp1_2d+S1;
      ReOp2_d<=ReIp1_2d-S1;
      CmpOp1_d<=CmpIp1_2d+S2;
      CmpOp2_d<=CmpIp1_2d-S2;
end

endmodule
```

Figure 41. Writing data from working buffer to butterfly inputs

7.6. Comparing Design vs Simulation

After taking constraints from MATLAB that are employed in the filtering of "input.wav" Design is made in the system verilog to obtain same results. Some of the results are more similar to that MATLAB results and some results from System Verilog are much cleaner than that of from MATLAB.

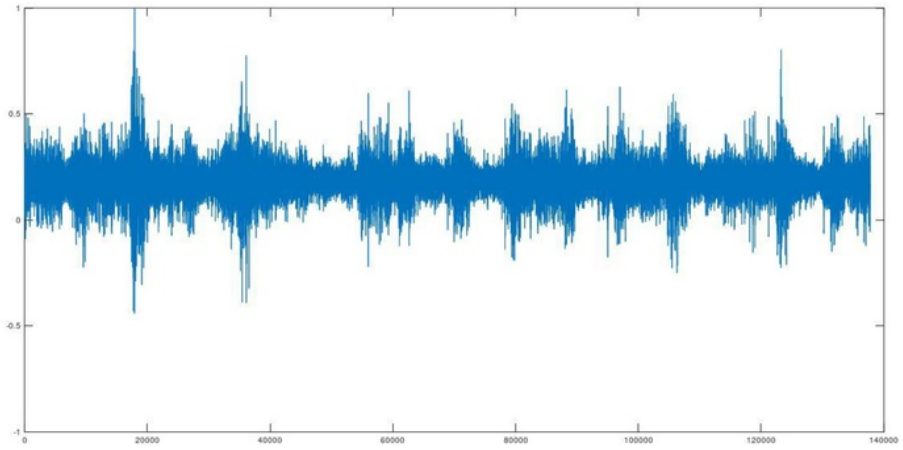


Figure 42. Input signal (with noise)

In figure 42 input audio file which contains much of white noise is plotted in time domain. Because of white noise the low frequency components of heartbeat are unheard and it's difficult to obtain the actual heartbeat sound.

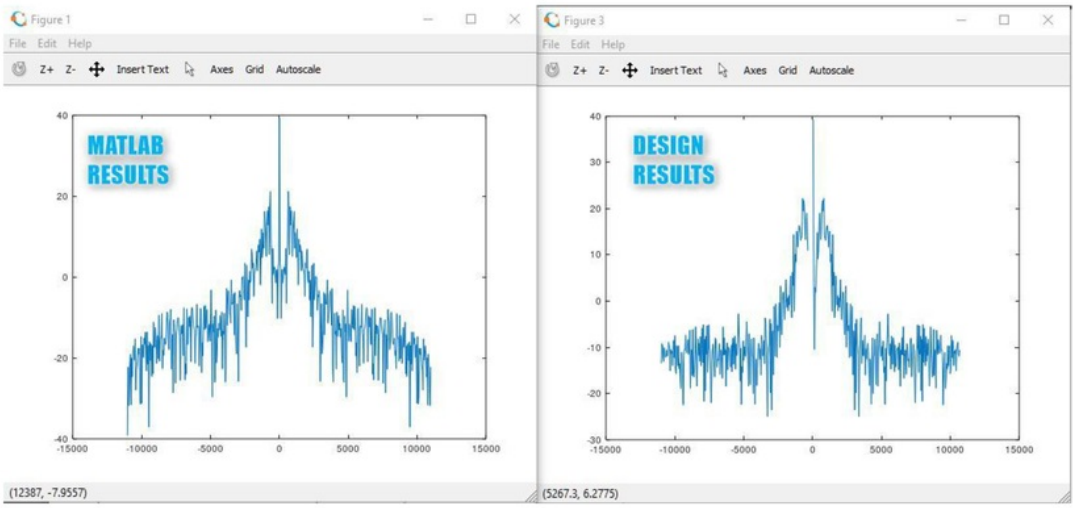


Figure 43. FFT output of the first 512 points

FFT is performed in for each block of input signal whose size is 512. FFT plot of first 512 samples of input signal is plotted and shown on left side of figure 43 and the result of 512-point FFT performed using FFT design in System Verilog is plotted and is shown on the right side of figure 48. It is observed that results from FFT design in System Verilog are much cleaner. The same fashion of output clarity is seen for filtered output of first 512 samples of input signal.

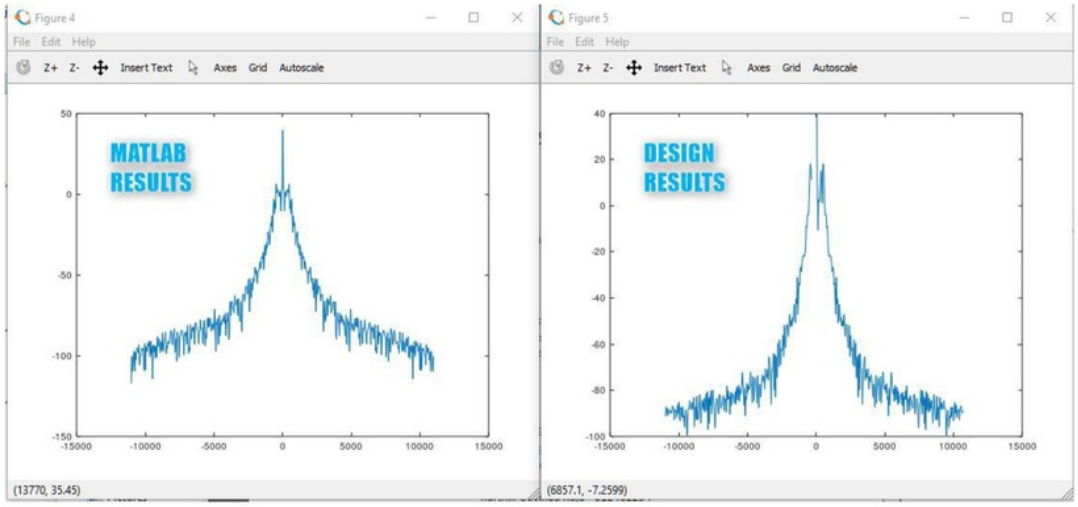


Figure 44. Filtered output of the first 512 points

After filtering entire input signal, it is converted back to wave format and plotted in MATLAB. It is observed in figure 45 that much of noise is removed and actual heartbeat dominates the noise floor.

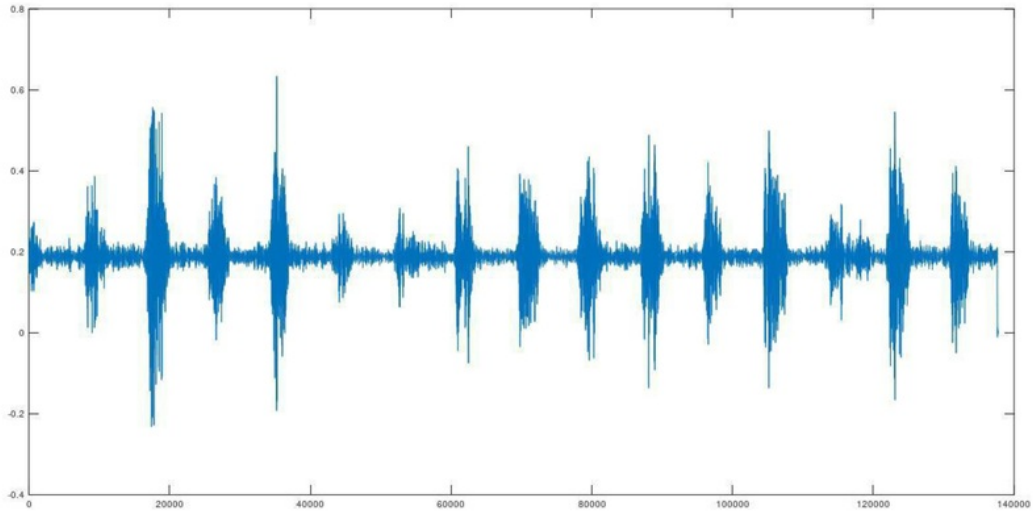


Figure 45. Final output of entire signal after IFFT

8. Java Code for generating states of FFT and IFFT

The coding for this project involved trial and error method with various FFT designs such as 256 points, 512 points, 1024 points and so on. Writing the code for all of these was a tedious job. So instead we designed a java application which was intelligent enough to generate System Verilog files customized for each FFT and IFFT design.

The inputs to the java program defined the design parameters. The same are as below

1. Number of Multipliers

The Design contains several multiplier units working in parallel to get the FFT work done. The number of multipliers high the rate of output generation. But this comes at a cost of larger synthesized area. Thus, as a tradeoff different numbers needed to be tried.

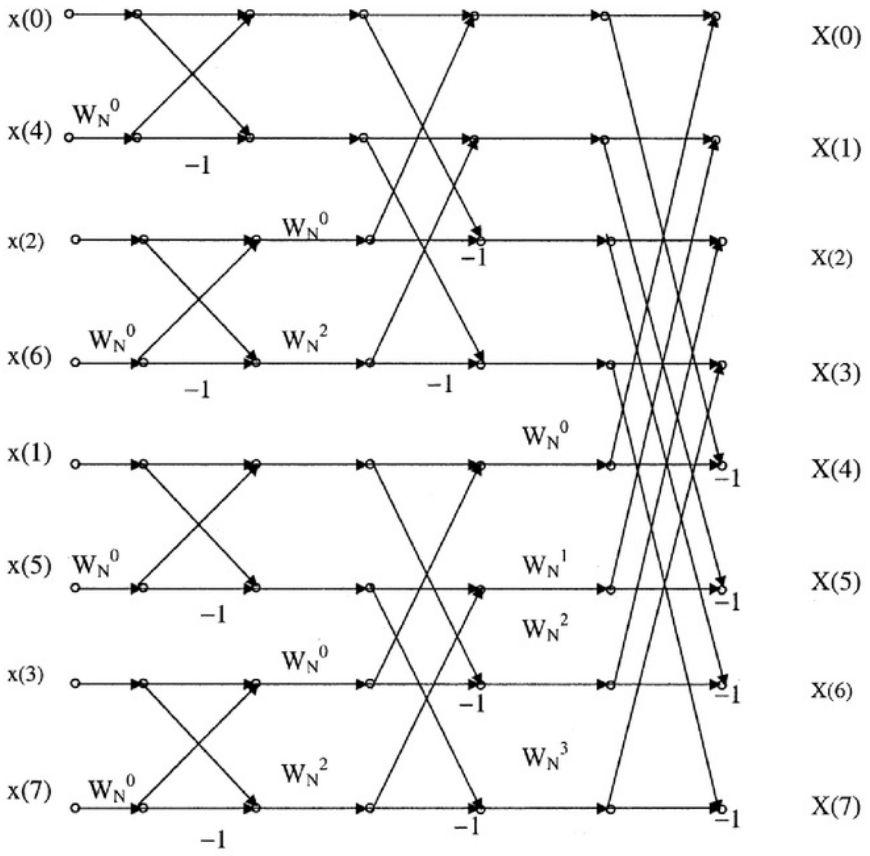
2. Number of points (256, 512, …)

As explained in the start of this page, we need to design a system where we can easily switch between different FFT and IFFT designs and do a trial and error method to obtain the same precision of noise cancellation as seen in simulation. Thus, we designed a java program which was robust such that we could easily generate the entire code for different points as per the input.

The number of points combined with Number of multipliers decided the max number of states of the design state diagram

3. FFT or IFFT

FFT and IFFT design are only a small change with twiddles varying as shown below



$W_N^0 = 1$, $W_N^1 = (1-j)/\sqrt{2}$, $W_N^2 = -j$, $W_N^3 = -(1+j)/\sqrt{2}$

Figure 46. FFT butterflies

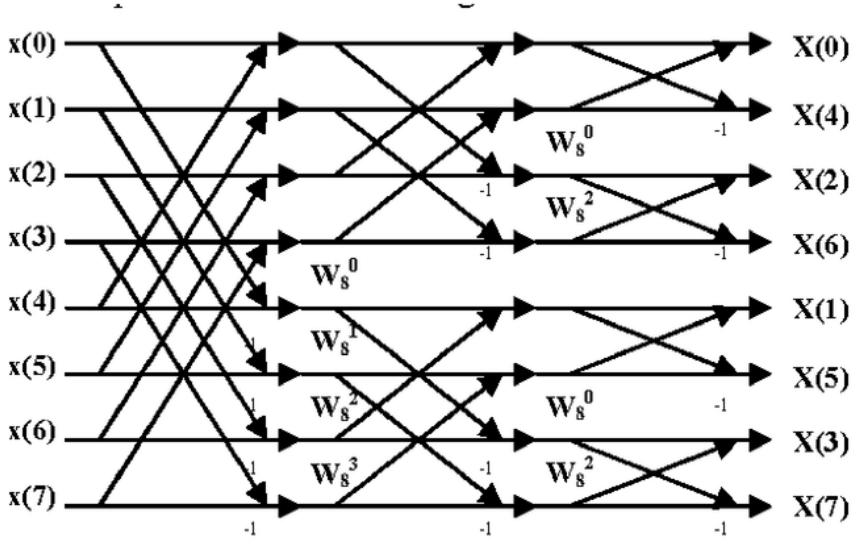


Figure 47. IFFT butterflies

The main differences between an FFT and IFFT are

1. Twiddles are calculated before the execution of the butterfly operation for a fft. Whereas the twiddles are computed after butterfly operation of IFFT
2. Twiddle factors a slight calculation change. In FFT, the imaginary part come with negative sign whereas in IFFT the imaginary part come with positive sign.
3. The initial set of addresses as you see above are reversed for FFT. Whereas for IFFT the initial is sequential, whereas the output is reversed.

Figure 48 shows all the above points in code

```
package txtFileGenerator;
public class Twiddles_512_ifft {
    public static double generate(boolean imag,int currentStage, int level){
        switch (currentStage){
        case 0:
            switch(level){
            case 0:if(!imag) return 1.000000;
            else return 0.000000;
            case 1:if(!imag) return 0.999925;
            else return 0.012272;
            case 2:if(!imag) return 0.999699;
            else return 0.024541;
            case 3:if(!imag) return 0.999322;
            else return 0.036807;
            case 4:if(!imag) return 0.998795;
            else return 0.049068;
            case 5:if(!imag) return 0.998118;
            else return 0.061321;
```

```
                else return 0.024541;
                case 255:if(!imag) return -0.999925;
                else return 0.012272;
                }
            case 1:
                switch(level){
                case 0:if(!imag) return 1.000000;
                else return 0.000000;
                case 1:if(!imag) return 0.999699;
                else return 0.024541;
                case 2:if(!imag) return 0.998795;
                else return 0.049068;
                case 3:if(!imag) return 0.997290;
                else return 0.073565;
                case 4:if(!imag) return 0.995185;
                else return 0.098017;

            }
        case 2:
            switch(level){
            case 0:if(!imag) return 1.000000;
            else return 0.000000;
            case 1:if(!imag) return 0.998795;
            else return 0.049068;
            case 2:if(!imag) return 0.995185;
            else return 0.098017;
```

Figure 48. Twiddle Generation showing different stages of Twiddle

The above twiddles were further generated using another Octave snippet shown below

```
for mm = 0:1:(fft_length-1)
    theta = (-2*pi*mm*1/fft_length);
    twiddle(mm+1) = cos(theta) + (1i*(sin(theta)));

    real_twiddle = real(twiddle);
    im_twiddle = imag(twiddle);
    printf (' case %d:\n',mm);
    printf ('   switch(level){\n');
    for c=1 : length(real_twiddle)

      printf('      case %d:if(!imag) return %f;\n',c-1,real_twiddl
      printf('        else return %f;\n',im_twiddle(c));
      count=count+1;
    end
    printf('    }\n');
    count=count+1;
end
printf('   }\n');
printf('  }\n');
printf('}\n');
```

Figure 49. Twiddle Value Generation

These twiddle values were further used to generate the states of the FFT and IFFT as below

```
    //boolean stage0flag;
    public void caseStatementGenerator() throws IOException{
        BufferedWriter writer = new BufferedWriter(new FileWriter("ifftCalculator.v"));
        printHardCodeStatements(writer);
        printDefaultCaseStatements(writer);
        for (int tstage = 0; tstage < numberOfStages; tstage++) {
            int stage=numberOfStages-1-tstage;
//          if(stage==0) {stage0flag=true;}
//          else {stage0flag=false;}
            double end=Npoint/Math.pow ( 2, (stage+1));
            for(int p=0; p<end;p++){
                int firstVariable=(int) ((Math.pow ( 2, (stage+1)))*p);
                int end2=(int)Math.pow ( 2, (stage));
                for(int q=0;q<end2;q++){
                    int input1=firstVariable+q;
                    int input2=firstVariable+q+(int)Math.pow ( 2, (stage));
                    printCaseStatements(writer,input1,input2,tstage,q,(tstage+1==numberOfStages && p==0 && q==0),(tstage==0 &&
                    System.out.println(input1+" : "+input2+" ::stage =  "+tstage);

                }
            }
        }
        //      writer.write("\t\t\t  "+(numberOfBitsToRepresentAllStates)+"'d"+2574+" : begin\n");
```

Figure 50. Calculation of number of states

Once the number of states are known, code generation for each state is required. And this takes the looping logic shown in figure 50.

```
for(int p=0; p<end;p++){
    int firstVariable=(int) ((Math.pow ( 2, (stage+1)))*p);
    int end2=(int)Math.pow ( 2, (stage));
    for(int q=0;q<end2;q++){
        int input1=firstVariable+q;
        int input2=firstVariable+q+(int)Math.pow ( 2, (stage));
        printCaseStatements(writer,input1,input2,tstage,q,(tstage+1==numberOfStages && p==0 && q==0),(tstage
        System.out.println(input1+" : "+input2+" ::stage =  "+tstage);

    }
}
```

Figure 51. Looping conditions

Here the looping condition takes care of the below points

1. The number of states
2. Number of multipliers in each state
3. Calculating the number of stage in each state
4. Then once we drill down to one stage of one state, we generate the state variable which include
    a. Twiddles from twiddle generator
    b. Variable address from memory

Once all the data is available, generate the statement is next step. This is done by printing the data into a text file and saving the same as with ".sv" extension.

```
if(count%noOfMultipliers ==0){
    writer.write("\t\t\t "+caseVariable+++" : begin\n");
}

if(firstStage) {
    writer.write("\n\t\t\t\t\t lastStageFlag<=1'b0;\n");
}

writer.write("\n\t\t\t\t\t //("+input1+","+input2+")\n");

//writer.write("\t\t\t\t\t pushinB"+count%noOfMultipliers+" <= 1'b1;\n");
writer.write("\t\t\t\t\t imgIp1B"+count%noOfMultipliers+" <=#1 imagWorkingBuffer["+input1+"];\n");
writer.write("\t\t\t\t\t realIp1B"+count%noOfMultipliers+" <=#1 realWorkingBuffer["+input1+"];\n");
writer.write("\t\t\t\t\t imgIp2B"+count%noOfMultipliers+" <=#1 imagWorkingBuffer["+input2+"];\n");
writer.write("\t\t\t\t\t realIp2B"+count%noOfMultipliers+" <=#1 realWorkingBuffer["+input2+"];\n");

double real = 0;
double imag = 0;
switch (Npoint) {
case 256:
    real=Twiddles_256.generate(false,currentStage,q);
    imag=Twiddles_256.generate(true,currentStage,q);
    break;
case 512:
    real=Twiddles 512 ifft.generate(false,currentStage,q);
```

Figure 52. Generating twiddles

The above code in the figure 52 shows the same, where it is observed writer object writing System Verilog code into text file.

This approach is similar made as it is easy in java to system Verilog converter. But in simpler manner, the java script generates System Verilog design files which are compilation free and easily customizable.

9. Summary

The ultrasound heartbeat of fetus is taken as input signal which contains lot of noise. This is first filtered in MATLAB. Using constrains from MATLAB ASIC design is made in System Verilog that contains FFT, hann window, IFFT, FIR filter. After filtering the same signal using System Verilog design code, the results are compared with MATLAB results which are observed as much cleaner than that from MATLAB. This project can be further extended in designing a generalized FIR filter that is suitable for any type of input with noise.

10. References

[1] "Part 7: The 2018 Wilson Research Group Functional Verification Study." *Verification Horizons BLOG RSS*, blogs.mentor.com/verificationhorizons/blog/2019/01/22/part-7-the-2018-wilson-research-group-functional-verification-study

[2] 2019, https://en.wikipedia.org/wiki/The_Hum, accessed 10 May 2019

[3] 2019, https://www.crystalinstruments.com/dynamic-signal-analysis-basics, accessed 10 May 2019

[4] 2018, http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html, accessed 10 May 10, 2019

[5] 2019, http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html. accessed 10 May 2019.

[6] 2017, https://electronics.stackexchange.com/questions/15206/iir-filters-what-does-infinite-mean, accessed May 10, 2019.

[7] 2019, https://community.plm.automation.siemens.com/t5/Testing-Knowledge-Base/Window-Correction-Factors/ta-p/431775, accessed May 10, 2019

[8] 2019, https://community.plm.automation.siemens.com/t5/Testing-Knowledge-Base/Window-Correction-Factors/ta-p/431775, accessed May 10, 2019

[9] 2019, http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html. Accessed 10 May 2019.

[10] 2019, http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html. Accessed 10 May 2019.

[11] Petrakieva, Simona, Oleg Garasym, and Ina Taralova. "http://ieeexplore. ieee. org/stamp/stamp. jsp? tp= &arnumber= 7038771." (2014).

[12] Y. Ephraim and D. Malah "Speech enhancement using a minimum-mean square error

[13] short-time spectral amplitude estimator" Acoustics Speech and Signal Processing IEEE Transactions on vol. 32 no. 6 pp. 1109-1121 1984.

[14] R. McAulay and M. Malpass "Speech enhancement using a soft-decision noise suppression filter" Acoustics Speech and Signal Processing IEEE Transactions on vol. 28 no. 2 pp. 137-145 1980. (Pubitemid 10487373)

[15] P. J. Wolfe and S. J. Godsill "Efficient alternatives to the Ephraim and Malah suppression rule for audio signal enhancement" EURASIP Journal on Advances in Signal Processing vol. 2003 no. 10 p. 910167 2003.

[16] R. Martin "Speech enhancement based on minimum meansquare error estimation and supergaussian priors" Speech and Audio Processing IEEE Transactions on vol. 13 no. 5 pp. 845-856 2005. (Pubitemid 41558900)

[17] S. Doclo and M. Moonen "On the output SNR of the speechdistortion weighted multichannel Wiener filter" Signal Processing Letters IEEE vol. 12 no. 12 pp. 809-811 Dec. 2005. (Pubitemid 41800439)

[18] T. Den Bogaert J. Wouters S. Doclo and M. Moonen "Binaural cue preservation for hearing aids using an interaural transfer function multichannel Wiener filter" in ICASSP 2007 Proceedings vol. 4 April pp. IV-565-IV-568.

[19] B. Cornelis M. Moonen and J. Wouters "Performance analysis of multichannel Wiener filter-based noise reduction in hearing aids under second order statistics estimation errors" Audio Speech and Language Processing IEEE Transactions on vol. 19 no. 5 pp. 1368-1381 2011.

Appendix

MATLAB code for filtering "input.wav":

```
clear all;
clc;
close all;


% *********FETCHING INPUT (FIGURE 1)*********
inputSignal = audioread('Input.wav');
expectedSignal = audioread('Expected.wav');


% *********INITIALIZATION*********
fs=22050;               %Samplng Rate of sound = 22050 samples/sec
BS = 4096;
pkg load signal
outputSignal(1:length(inputSignal))=0;

if(false)
  figureCOunt=2;
  generateGraph('input Signal',inputSignal,fs,figureCOunt);
  generateGraph('expected Signal',expectedSignal,fs,figureCOunt);
end

%sound(inputSignal,fs);
%sound(expectedSignal,fs);

generateplot=true;
loopEnd=(length(inputSignal)-mod(length(inputSignal),BS))
% *********LOOPING THROUGH (FIGURE 2)*********
for i=1:(BS/2):loopEnd
%i=1;
    % *********SELECTING A RANGE OF INPUT SIGNAL*********
    if((i+BS-1)<=length(inputSignal))
      selectedRange  =   inputSignal(i:i+BS-1);
      windowLength = BS;
    else
      selectedRange  =   inputSignal(i:length(inputSignal)-1);
      windowLength = length(inputSignal)-i;
    end
    % *********APPLYING WINDOWING FUNCTION*********

    window=hanning(windowLength);
    windowedSignal   =   selectedRange.* window;

    % *********Filter Design*********

    f1 = 500;
    f2 = 750;
    delta_f = f2-f1;
    dB  = 40;
```

```
    N = dB*fs/(22*delta_f);

    f =  [f1 ]/(fs/2);
%    hc = fir1(round(N)-1, f,'low');
    hc = fir1(900,0.05,'low');
%    [b,a]=butter ( 1, (50*2) / fs );

    % *********APPLYING FILTER*********

    filteredResponse      =   filter(hc,1,selectedRange);


    % *********APPLYING INERSE FOURIER TRANSFORM*********


    slicedOutput  = filteredResponse(BS/4:3*(BS/4));

    % *********ADDING *********
    outputSignal(i:i+BS/2-1)=slicedOutput(1:BS/2);

   if(false)
      figureCOunt=5;
      generateGraph('selected Range',selectedRange,fs,figureCOunt);
      generateGraph('windowed Signal',windowedSignal,fs,figureCOunt);
      generateGraph('filtered
Response',filteredResponse,fs,figureCOunt);
      generateGraph('sliced Output',slicedOutput,fs,figureCOunt);
      generateGraph('output Output',outputSignal,fs,figureCOunt);
   end
netx=1;

end

if(true)
  figureCOunt=3;
  generateGraph('Input Signal',inputSignal,fs,figureCOunt);
  generateGraph('Expected Signal',expectedSignal,fs,figureCOunt);
  generateGraph('output Signal',outputSignal,fs,figureCOunt);
end

if(true)
  #sound(inputSignal,fs);
  #sound(expectedSignal,fs);
  sound(outputSignal,fs);
  audiowrite('output.wav',outputSignal,fs);
  outputSignal = audioread('output.wav');%wave file
  #sound(outputSignal,fs);
end
```

MATLAB code to generate graphs:

```
function generateGraph(name,signal,fs,figureCOunt)
    persistent n;
    if isempty(n)
        n = 0;
        num=0;
    end
    if(n==0)
      figure()
    end
    num=n*2+1;
    lengthOfSignal=length(signal);
    subplot(figureCOunt,2,num)
  %TIME PLOT
    t = 0:lengthOfSignal-1;
    plot(t,signal);title(['Time Domain '
name]);xlabel('Time,s');ylabel(name);
  %FREQUENCY PLOT
    subplot(figureCOunt,2,num+1)

    %{
    fftSelectedSignal = fft(signal,lengthOfSignal);
    fftSelectedSignal =
fftshift(abs(fftSelectedSignal/lengthOfSignal))(1:(lengthOfSignal)/2+1)
;
    fftLength=length(fftSelectedSignal);
    fftSelectedSignal(2:fftLength-1)=2*fftSelectedSignal(2:fftLength-
1);
    f = fs*(0:(lengthOfSignal/2))/lengthOfSignal;
    plot(f, fftSelectedSignal);
    %axis ([4000 8000 0 1000]);
    %}

    plot((-0.5:1/lengthOfSignal:0.5-
1/lengthOfSignal)*fs,20*log10(abs(fftshift(fft(signal,lengthOfSignal)))
))
    title(['Frequency Domain '
name]);xlabel('frequency,f');ylabel(name);
    %axis ([-2000 2000 -100 100]);
    n=n+1;
    if(n==figureCOunt)
      n=0;
    end
end
```

FFT design:

```
`include "fftCalculator.v"

module fft();

reg clk,reset;
//dataMem

reg startin,startout;
integer data_file;
integer scan_file;
real captured_data,realin,imagin, realout, imagout;
fftCalculator
obj(clk,reset,realin,imagin,startin,startout,realout,imagout);


always @(posedge clk or posedge reset) begin

    if(reset) begin


    end
    else begin
       scan_file = $fscanf(data_file, "%f\n", captured_data);
        $display(scan_file);
        realin<=#1 captured_data;
        imagin<=#1 0.0;
    end
end

initial begin
    data_file = $fopen("inputData.txt", "r");
    clk=0;
    reset=1;
    #5
    reset=0;
    repeat(50) begin
        clk=#1 ~clk;
    end
end

initial begin
    $dumpfile("fft.vpd");
    $dumpvars();
end

endmodule
```

final report.pdf

ORIGINALITY REPORT

54%	4%	1%	52%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to CSU, San Jose State University Student Paper	51%
2	www.mathworks.com Internet Source	1%
3	reference.digilentinc.com Internet Source	1%
4	www.allaboutcircuits.com Internet Source	1%
5	Submitted to Glasgow Caledonian University Student Paper	<1%
6	Submitted to 8779 Student Paper	<1%
7	epdf.tips Internet Source	<1%

Exclude quotes	On	Exclude matches	Off
Exclude bibliography	On		

Electrical Engineering Department

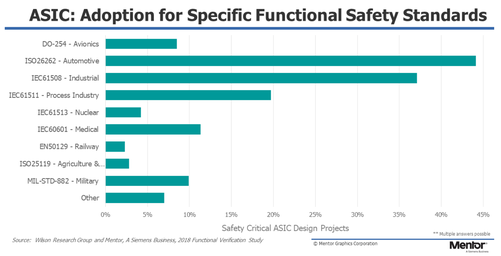
Noise cancellation using windowing technique

Project Advisors: Prof. Lili He, Prof. Morris Jones

Tata, Lekhya (MS Electrical Engineering)
Govinda Raju, Karthik (MS Electrical Engineering)

Introduction

ASIC chips are leading in the industry of semiconductor technology because they are working at a quicker rate and as a result, for better operation and task involving great technology, design becomes more complex. It includes many challenges and trade-offs.

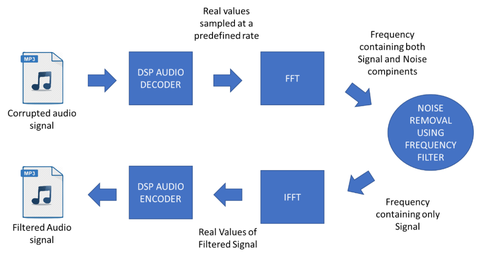


Aim of the project is to design a filter that performs better and reduce maximum noise in a faster way. Generally, noise can be eliminated in time domain and frequency domain. In this project, design is made for frequency domain as speech and noise signals may be better separated in that space, which enables better filter estimation and noise reduction performance. Noise can be categorized in many ways. Additive noise (white noise) is selected for filtering

Methodology

Design flow

When the audio signal with noise is been fed to MATLAB, cut off frequency of noise signal is known from magnitude spectrum plot. This cutoff frequency is used to calculate digital cut off using sampling frequency and analog cut off frequency



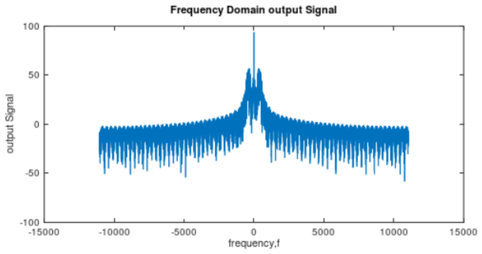
Cut off frequency obtained from the spectrum plot is used to design low pass FIR filter for the particular input

Methodology

Design in MATLAB.

In MATLAB filtering is done in time-domain and using predefined functions for filter, window and FFT are used.

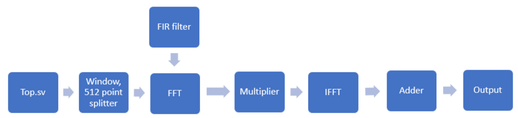
Initial step is to load the input file. Input is in wave format. “audioread” function is used to load input. Total length of the signal is determined. Entire length is made into blocks and each block is size 512. To each block Hann window is applied, which removes signal of 25% at the starting and ending of each block. FFT is applied to this windowed signal. After filtering IFFT is done combine all these blocks into one audio file which is wave format.



Designing in System Verilog

Constraints from MATLAB are used to perform filtering using System Verilog.

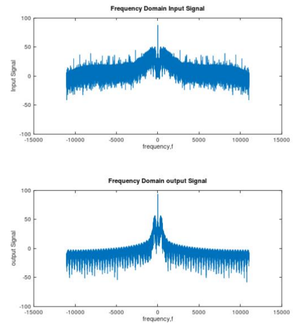
“Input.wav” file is converted into text file that contains real and imaginary values which is nothing but vector representation of signal in digital form. 512 point FFT logic is designed. This logic is used to perform FFT on windowing function and low pass FIR filter. The digital representation text file of hann window and low pass FIR filter is taken using MATLAB. After performing FFT on both, they are multiplied using multiplier design. IFFT is performed to this product and all the results from each for loop is added and produced an output file in vector form.



The vector format of output obtained through system verilog is converted back to wave format and plotted. outputs from MATLAB and System verilog are compared. They both are nearlt similar to each other.

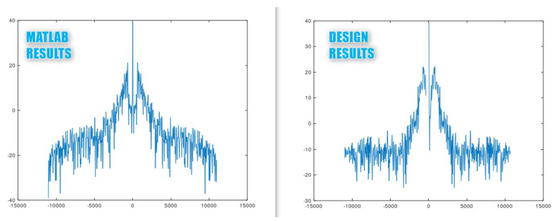
Analysis and Results

FFT spectrum of input signal and filtered output signal. Noise floor above 0db is removed. Output signal is much cleaner compared to input.



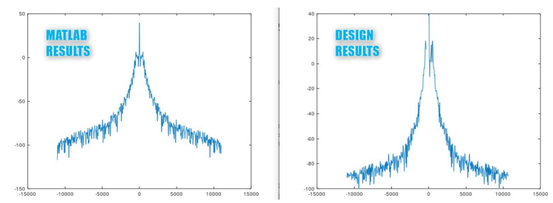
Plot of input signal and filtered signal in frequency domain

As the process goes FFT of first 512 points is done in MATLAB. With the appropriate design in System Verilog FFT of first 512 points is done. The resultant values are converted to text file and plotted in MATLAB. When both the results are compared, the results from system Verilog are much cleaner.



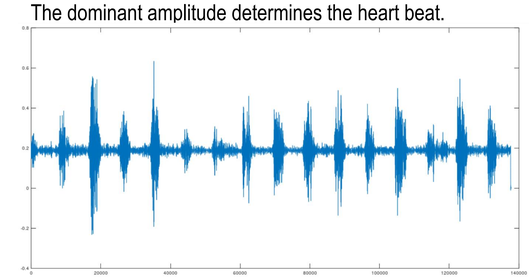
FFT of 512 samples of filtered signal

By applying low pass FIR filter of order 900 and roll off as 0.05 is applied to the windowed signal. Results from both MATLAB and System Verilog is compared which proves System Verilog output is more cleaner.



FFT of filtered signal

The dominant amplitude determines the heart beat.



Filtered output signal in time domain

Summary/Conclusions

Though the focus of project is to eliminate few defined real time noise categories, the design can be further extended in designing a generalized FIR filter. Also, the windowing mechanism limits the number of samples processed per time to a smaller range which are limited by the size, power and time constraints. This can be further enhanced using a wider configuration range which can boost the performance of the system.

Key References

[1] Y. Ephraim and D. Malah "Speech enhancement using a minimum-mean square error short-time spectral amplitude estimator" Acoustics Speech and Signal Processing IEEE Transactions on vol. 32 no. 6 pp. 1109-1121 1984.

[2] R. McAulay and M. Malpass "Speech enhancement using a soft-decision noise suppression filter" Acoustics Speech and Signal Processing IEEE Transactions on vol. 28 no. 2 pp. 137-145 1980. (Pubitemid 10487373)

[3] P. J. Wolfe and S. J. Godsill "Efficient alternatives to the Ephraim and Malah suppression rule for audio signal enhancement" EURASIP Journal on Advances in Signal Processing vol. 2003 no. 10 p. 910167 2003

[4] R. Martin "Speech enhancement based on minimum meansquare error estimation and supergaussian priors" Speech and Audio Processing IEEE Transactions on vol. 13.

Acknowledgements

We are greatly thankful to Dr. Lili He for giving a chance to work on the project under her guidance. Thanks to project co-advisor Prof. Morris Jones for his continuous help and co-operation.

We are also thankful to San Jose state university Electrical engineering department for providing all resources without which the project work would be inadequate