DOKUZ EYLÜL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

DIGITAL FILTERING FOR COMMUNICATION SIGNALS USING FPGA TECHNOLOGY

by
Selçuk BELEN

October, 2010
İZMİR

DIGITAL FILTERING FOR COMMUNICATION SIGNALS USING FPGA TECHNOLOGY

A Thesis Submitted to the
Graduate School of Natural and Applied Sciences of Dokuz Eylül University
In Partial Fulfillment of the Requirements for the Degree of Master of Science in Electrical and Electronics Engineering

by

Selçuk BELEN

October, 2010

İZMİR

M.Sc THESIS EXAMINATION RESULT FORM

We have read the thesis entitled **"DIGITAL FILTERING FOR COMMUNICATION SIGNALS USING FPGA TECHNOLOGY"** completed by **SELÇUK BELEN** under supervision of **Asst. Prof. Dr. ÖZGE ŞAHİN** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

.....

Supervisor

…… ……

(Jury Member) (Jury Member)

Prof. Dr. Mustafa SABUNCU

Director

Graduate School of Natural and Applied Sciences

ACKNOWLEDGEMENTS

I am heartily thankful to my supervisor, Özge ŞAHİN, whose encouragement, guidance and support from the initial to the final level enabled me to develop an understanding of the subject.

Also I would like to thank to my lovely family members. This thesis would not have been possible without spiritual support of them.

Lastly, I offer my regards and blessings to all of my friends who supported me in any respect during the completion of the project.

Selçuk BELEN

DIGITAL FILTERING FOR COMMUNICATION SIGNALS USING FPGA TECHNOLOGY

ABSTRACT

Base element of digital systems which are used in a very wide area between daily life and or hi-tech research applications is ICs (Integrated Circuits). There are several technologies for designing and producing these IC's which are used for several applications.

FPGA (Field Programmable Gate Array) is a new technology which provides a different new way to develop these integrated circuits. It is produced by a few companies around the world and common property of these FPGA's is they can be configured by the hardware descriptive languages Verilog and VHDL (Verilog Hardware Descriptive Language) which have base standards. Filtering of digital signals on implemented hardware using these languages is explained in this thesis.

Together with explanation of developed filter which is created by VHDL command lines, algorithm of serial communication modules which are needed for data communication in the project like transmitter, receiver and delayer are also explained in this thesis.

To test the constructed modules in FPGA digital data is transmitted and received through serial port of FPGA. Audio files which have specific sampling frequency and bit depth are used for testing filtering characteristics.

In addition, subjects like algorithm of VHDL modules, digital filter construction, module instantiation, differences of filter architectures and design process are explained and results are compared. Also several software are used in the design process and it is explained how to use them in this thesis.

Keywords: FPGA, VHDL, digital filters, instantiation, implementation

FPGA KULLANARAK HABERLEŞME SAYISAL İŞARETLERİNİN SÜZÜLMESİ

ÖZ

Günlük ihtiyaçlarda kullanılan cihazlardan, yüksek teknoloji araştırmalarına kadar geniş bir yelpazede kullanılan sayısal sistemlerin temel elemanı entegre devrelerdir. Değişik amaçlar için kullanılan ve farklı tipleri bulunan bu entegre devrelerin, bilinen birden fazla tasarım ve üretim teknolojileri vardır.

FPGA sayısal tasarımda entegre devrelerin üretimi için farklı bir yol sunan yeni bir teknolojidir. Dünyada FPGA üreten bir kaç şirket bulunmaktadır ve bu FPGA'ların ortak noktası belirli yazılım standartlarına sahip Verilog ve VHDL olarak bilinen donanım tabanlı komut dilleridir. Bu tezde sayısal sinyallerin VHDL kullanılarak oluşturulan entegre devre ile süzülmesi anlatılmaktadır.

VHDL kullanılarak sayısal süzme için tasarlanan entegre kısmına ilaveten verilerin iletiminde rol oynayan ardışık alıcı, ardışık gönderici, erteleme gibi işlemlere yarayan çevresel modüller de VHDL kullanılarak oluşturulmuş ve algoritmaları anlatılmıştır.

Haberleşme sayısal işaretlerinin FPGA içinde süzülmesi için seri haberleşme modülleri aracılığıyla veri gönderilmiş ve alınmıştır. Süzme için belirli örnekleme frekansına ve bit genişliğine sahip ses dosyaları kullanılmıştır.

Ayrıca tasarımda kullanılan VHDL komutlarının mantık haritaları, sayısal süzgeç tasarımı, modül instantiation, değişik süzme mimarilerinin birbirinden farkları ve tasarım boyunda izlenen yol, açıklamalarla birlikte sonuçlar kıyaslanarak yorumlanmıştır.

Anahtar Sözcükler: FPGA, VHDL, sayısal süzgeç, dijital filtre, implementasyon

CONTENTS

Page

M.Sc THESIS EXAMINATION RESULT FORM ii

ACKLOWNEDGEMENTS iii

ABSTRACT iv

ÖZ v

CHAPTER ONE - INTRODUCTION 1

1.1 Overview of Thesis 3

CHAPTER TWO - FILTERS & FPGA's 5

2.1 FPGA Architecture 5
2.2 Filters in Communication 5
2.3 Why Digital Filter? 6
2.4 Which Type of Filter to Use? 7
2.4.1 FIR Filter Structure 8
2.4.2 IIR Filter Structure 9
2.4.3 Filter Performance on FPGA 10

CHAPTER THREE - IMPLEMENTATION OF DIGITAL FILTER 12

3.1 Architecture of the Filter 12
3.1.1 FIR Filter Structure Types 13
3.2 Implementation Methods 15
3.2.1 IP Generator Software 15
3.2.2 User Defined Coding 16

3.2.3 HDL Code Generators 16
3.3 Chosen Method for Implementation 17

CHAPTER FOUR - DESIGN TEMPLATE 18

4.1 Basic Specifications of the Template 18
4.2 Design Template Overview 19
4.3 Design of the Receiver 21
4.3.1 Pseudo Code for Receiver 24
4.4 Design of the Delayer 26
4.4.1 Pseudo Code for Delayer 28
4.5 Design of the Transmitter 29
4.5.1 Pseudo Code for Transmitter 31
4.6 Design of FIR Filters 33

CHAPTER FIVE - PRE-IMPLEMENTATION 35

5.1 Overview of the FPGA Board 35
5.2 Board Features 36
5.2.1 Memory 38
5.2.2 Clock Sources 38
5.2.3 10/100/1000 Ethernet PHY 38
5.2.4 RS-232 39
5.2.5 VGA Output 39
5.2.6 Miscellaneous I/O 40
5.3 Utilization of Resources on FPGA Board 40

CHAPTER SIX - DATA COMPARISON 46

6.1 PSTN Signals 46
6.1.1 Fundamentals of PCM in PSTN 47

6.1.1.1 Sampling 47
6.1.1.2 Quantizing 48
6.1.1.3 Encoding 49
6.1.1.4 Multiplexing 49
6.1.1.5 Demultiplexing 49
6.1.1.6 Decoding 49
6.2 Telephony Signals vs. Wav Files 50
6.2.1. Wav File 50

CHAPTER SEVEN - APPLICATION AND RESULTS 52

7.1 Pre-Implementation 52
7.2 Filter Design with FDATool 53
7.3 Implementation on FPGA 56
7.4 Transmit and Receive Data 56
7.5 Expected and Derived Results 58
7.5.1 Results of Direct Form Architecture FIRs 59
7.5.2 Results of Direct Form Transposed Architecture FIRs 68
7.5.3 Results of Direct Form Symmetric Architecture FIRs 78

CHAPTER EIGHT - CONCLUSION 91

8.1 Futureworks 92

REFERENCES 93

APPENDIX - CODES 95

9.1 VHDL Codes 95
9.1.1 VHDL Codes for Transmitter 95

9.1.2 VHDL Codes for Receiver 97
9.1.3 VHDL Codes for Delayer 100
9.2 C# Codes 102
9.2.1 Write Read Serial Port Codes 102
9.3 Filter Equation Tables 107

CHAPTER ONE
INTRODUCTION

Electronics has made a great progress in the past century and continues to grow in the 21st century. The start and growth of the computer industry has been responsible for the recent growth of electronics. Thirty years before now, in the 1980s, electronic systems were created by connecting basic components such as microprocessors and memory chips with digital logic components on PCBs (Printed Circuit Boards). As electronics grew, producing stable PCBs became more complex and harder, due to increase in the number of transistors inside ICs increase in the number of I/O (Input/Output) pins also the development of multi-layer boards which has more than 20 separate layers. Figure1.1 shows the increase in the number of transistors from 1970s to 2000s. Because of this huge progress, the probability of error increased. So it became harder to design and to test a working system before producing it. Increase of transistor number per IC is shown in figure below.

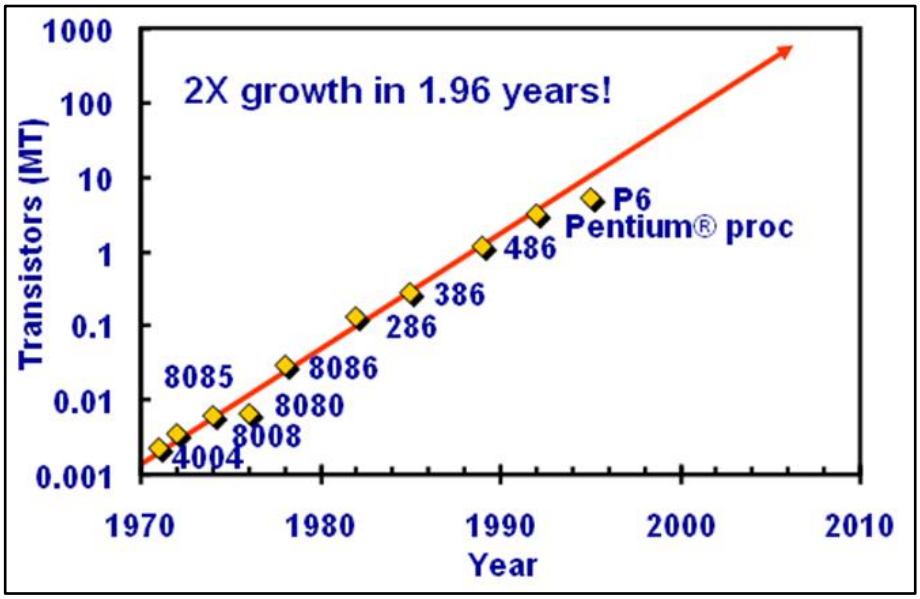

Figure1.1 Number of transistor per IC for past decade (Berkeley University)

Need for designing and testing a working system before production caused the new solution technology called FPGA.

FPGA can be described as simple ‘logic gate’ technology which is based on either antifuse, EPROM (Erasable Programmable Read Only Memory) or SRAM (Static Random Access Memory) technologies and provides programmable connectivity between desired components. This functionality helps designer to catch errors at the development stage before production. These errors can be corrected by simply reprogramming the FPGA. Reprogramming is provided by the array of logic blocks and routing channels. Thus using FPGA avoids a costly and time-consuming board redesign procedure and reduces the design risks.

Al Mahdi Eshtawie & Bin Othman, (2007) has obtained a better solution about representing coefficients of FIR filters for reducing area used to implement FIR filter architectures.

Vaidyanathan, (2004) has reviewed filter banks used in communication and filter bank predecoders which are very important for channel equalization applications.

Elhossini, Areibi & Dony, (2006) have worked on implementing LMS adaptive filtering on FPGA to compare software performance to the implemented hardware.

Wang, (2005) has worked on implementation of digital filters on FPGA’s for better performance-area ratios.

Bicakci, Çetinkaya & Karaboğa, (2005) have worked about designing digital filters for FPGA’s.

Dikmeşe, (2007) has worked on implementation of FPGA for wireless communication systems, especially for mobile wireless communications which require faster response in signal process.

Tamer, (2007) has worked about FPGA based smart antenna implementation to improve distinct processes better signal processing performance with processor arrays which are provided by FPGA’s.

These works have been very useful for understanding the FPGA's in a lot of perspectives and have been guidelines of the design process of this work. Such a work for the same purpose could not be found including implementations and applications as this project.

The objective of this thesis is to gather the VHDL code generation knowledge and FPGA implementation techniques to construct different hardware designs to achieve very high SNR values. Design bit depth can be enhanced with simply changing the specific codes on specific lines on VHDL codes.

There are different ways to implement a digital filter. There will be several FIR filters with different architectures and orders inside the designed system. That increases the number of hardware systems used for this project. The purpose is to identify different approaches and architectures, compare and contrast different methods.

With the help of results, FPGA resource usage results are a useful report to identify the best performance on hardware. To design and change the order of the filters, MATLAB is a useful tool which simply calculates the filter internals and coefficients for designer to save time. After implementing hardware on FPGA, audio data will be sent to the input of the filter system as serial bits. Pure input will be analyzed before it was sent and filtered data will be analyzed after the data is received. These analyses will be compared to make an analytical conclusion about the filter order, type and architecture to achieve best SNR (Signal to Noise Ratio) and sampling rates with optimum use of resources on FPGA.

1.1 Overview of Thesis

Chapter 2 covers a brief overview of FPGA, digital filters, digital filter types and performance calculation of these filters.

Chapter 3 covers a brief overview of FIR (Finite Impulse Response) filter architectures, MAC (Multiply Accumulate) units of FPGA's and describes digital filter implementation methods.

Chapter 4 describes the specifications of the top design, includes explanation of algorithms and comparisons of simulation results of communication modules implemented on FPGA.

Chapter 5 describes specifications and features of the used FPGA board. Also this chapter includes simulation results of digital filters' utilization results for Spartan 3A-DSP 1800A FPGA development board.

Chapter 6 gives brief information of signals used in communication, PCM (Pulse Code Modulation) process, and describes the estimation of signals to audio signals.

Chapter 7 describes the filter design method and application procedure, and then compares results of the implementations.

Chapter 8 concludes the thesis and gives advises for future works.

CHAPTER TWO
FILTERS & FPGA's

2.1 FPGA Architecture

There are four main categories of FPGAs available in the market: symmetrical-array, row-based, hierarchical PLD and sea-of-gates.

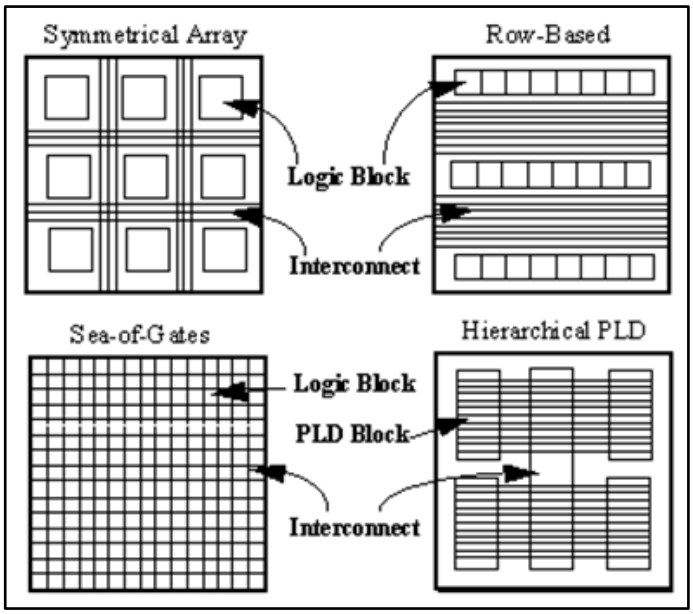

Figure 2.1 Base producing architecture types of FPGA's

Different designs are considered by main FPGA trademarks. Xilinx uses static RAM (Random Access Memory) programming and QuickLogic uses *antifuse* programming with *symmetrical array* architecture. Actel uses *antifuse* programming with *row-based* architecture. Algotronix uses static RAM programming with *sea-of-gates* architecture. ALTERA uses EPROM programming with *hierarchical PLD* (Programmable Logic Device) architecture.

2.2 Filters in Communication

As in most of the electronics, filters are often used in communication systems. They are used for several purposes as spectrum shaping, channel detecting and FFT (Fast Fourier Transform) etc. in communication. For example telephone lines which

were considered to carry speech signals are today used to carry many megabits of data per second. This became possible with efficient use of high frequency regions. As a result of using high frequency regions on telephone line, high speed internet traffic such as ADSL (Asymmetric Digital Subscriber Line) became available for use of the public. Several signals through a single channel became possible with the efficient use of filters located on start and end of the channel.

Generally communication channels are handled in two parts as wireless and wireline. In both cases, signals face with distortion and noise caused by environmental sources which affect the channel. Figure 2.2 below shows the noise affect on signals passing through the channel.

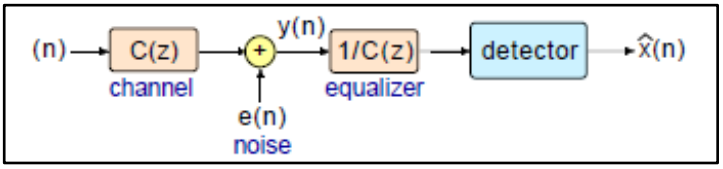

Figure 2.2 Noise effect on signal x(n) as passing through a physical channel

Noise reduction and spectrum shaping will be dealt with digital filters implemented on FPGA in this paper. Purpose of the thesis is to verify mathematical expressions of digital filters by implementing on FPGA and test these filters characteristics like efficiency and architecture for any kind of signal. And then compare these results to bring up a total report which may enlighten designers' future works about digital filter applications implemented on FPGA for several types of signals which are used in communication or any other areas.

2.3 Why Digital Filter?

Digital filters are used in all areas of electronics. They are mostly used for digital signal processing for audio and video applications. This is because digital filters can achieve better SNRs than analog filters. Ability of better SNR comes from the noiseless mathematical operations, but this property is not true for the analog filters

because the analog filter adds more noise to the signal. As a result digital filters became preferred option for removing noise and shaping spectrum in communication systems. Digital filters have become popular because they have precise reproducibility and allow designers to achieve high performance levels which are difficult to obtain with analog filters.

2.4 Which Type of Filter to Use?

There are two common filter forms which are FIR and IIR (Infinite Impulse Response). We need to decide if the desired filter should be IIR or FIR. The following table summarizes different factors that could be considered when making this decision.

Table 2.1 Differences between FIR and IIR filters (MIT EECS Dept 6.341: Discrete Time Signal Processing OpenCourseWare 2006)

	IIR Filters	FIR Filters
Phase	difficult to control, no particular techniques available	Linear phase always possible
Stability	can be unstable, can have limit cycles	always stable, no limit cycles
Order	Less	more
History	derived from analog filters	no analog history
Others		Polyphase implementation possible, can always be made casual

The number of calculations done per unit time is directly related to filter order. It affects number of logic gates usage on the FPGA board. FIR filters have only numerators according to their mathematical expression, whereas IIR filters have both numerator and denominator coefficients. So IIR filters are more complex to FIR filters and they use much more resources than FIR filters even if they are in the same order. According to the above analysis, the FIR type is chosen for the design.

2.4.1 FIR Filter Structure

Transfer function of N tap filter is shown in equation (2.0.1). In digital FIR filter for Nth order, tap is N+1.

$$H(z) = a_0 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_{(N-1)} z^{-(N-1)} \qquad (2.4.1)$$

Structure can be realized in many forms, such as canonical, pipelined or inverted as shown in Figure 2.3.

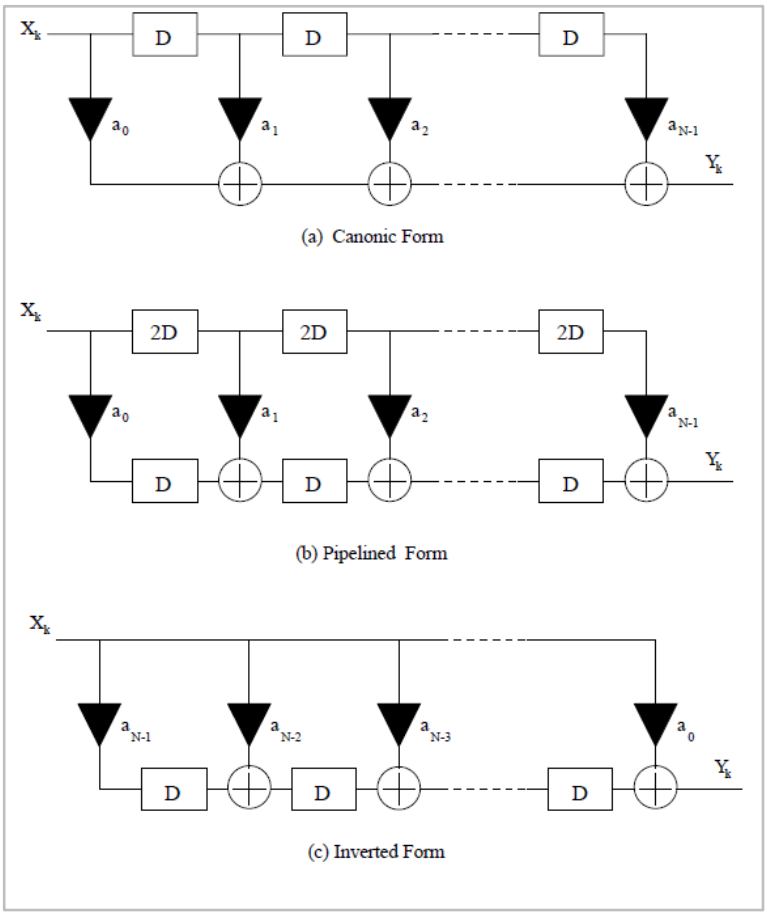

Figure 2.3 Realizations of FIR Filters in various forms

2.4.2 IIR Filter Structure

Transfer function of N[th] tap filter is expressed as;

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_{(N-1)} z^{-(N-1)}}{(1 - a_1 z^{-1} - a_2 z^{-2} - \cdots - a_{(N-1)} z^{-(N-1)})} \quad (2.4.2)$$

Two of possible realizations are direct form I and direct form II. For reducing the delay the realization shown in Figure 2.4 can be used.

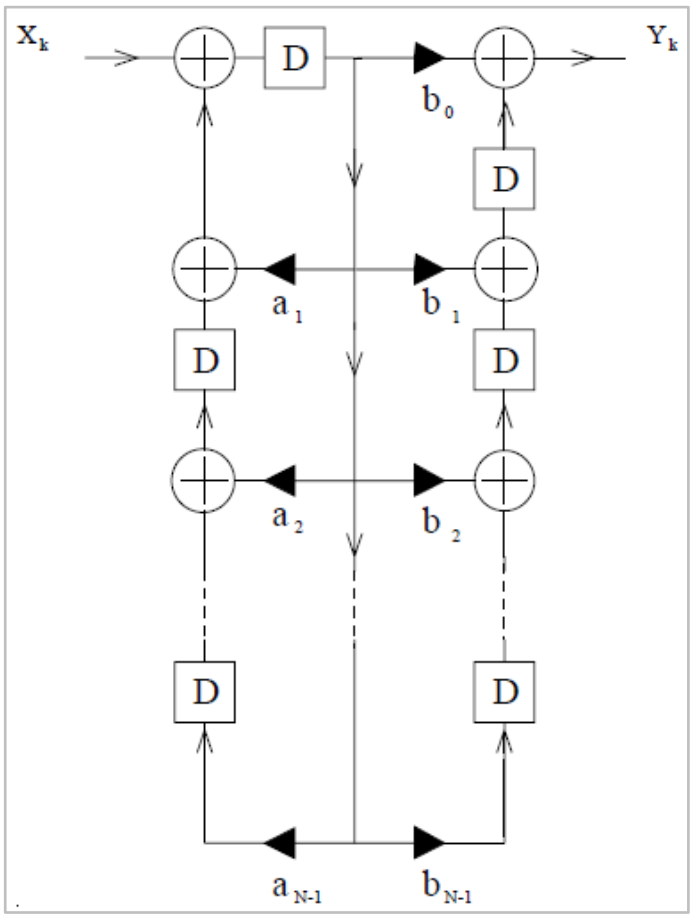

Figure 2.4 Modified Canonical Form realization of IIR Filter

It is cascade of an AR (Autoregressive) filter and MA (Moving Average) filter. There is only one path with a multiplier and two adders. This realization makes implementation easier and reduces the routing delays between CLBs (Configurable Logic Blocks).

2.4.3 Filter Performance on FPGA

If the size of the chip is constant and resources are low, the performance has to be considered before designing filter. The structure in Figure 2.5 is a MAC unit with four multipliers and four adders.

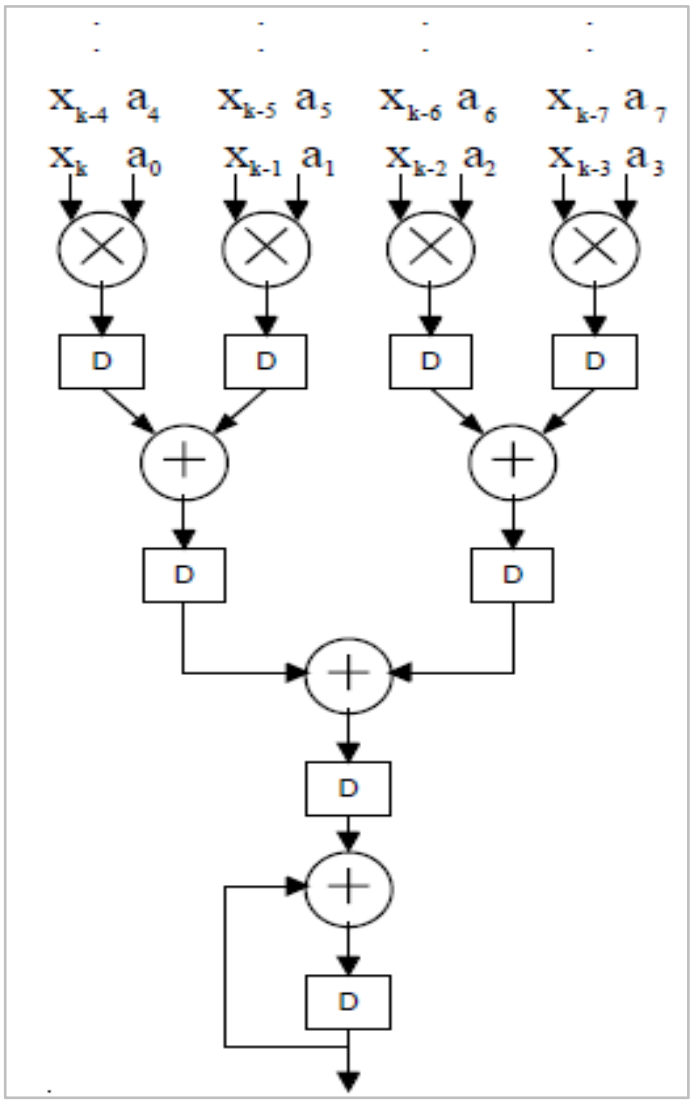

Figure 2.5 FIR Filter realization using MAC with four multipliers

Inputs are multiplied with the filter coefficients in the multipliers and summation is done with adders. Since the multipliers have greater delay than adders, clock frequency of the filter is dependent to the multipliers delay. There are four multipliers in this MAC unit and four outputs of multiplications are summed in every clock cycle. So if a four tap filter is implemented, the sampling frequency of the filter

will be same as the clock frequency, or eight tap filter with sampling frequency which is half of the clock frequency.

It can be said generally that if there are M multipliers in a chip and the delay of the multiplier is T sec, then N tap filter can operate at a sampling frequency f_s, given by;

$$f_s = \frac{1}{T(N/M)} \tag{2.4.3}$$

For considering that system has 100MHz clock frequency, multiplier delay is 100ns, adder delay is 25ns, an N tap filter can operate at a sampling rate of 40/N MHz where N is multiple of 4. For example a 16 tap filter can operate at 2.5 MHz sampling frequency with these specifications.

CHAPTER THREE
IMPLEMENTATION OF DIGITAL FILTER

3.1 Architecture of the Filter

Based on the resource capacity of your FPGA or the price limitation of the chip to be produced, area efficient serial filter architecture with low number of MACs or faster parallel architecture with higher number of MACs decision is up to the designer. Serial architectures are very area efficient and ideal for low price and low performance applications. Fully parallel architectures are generally chosen for high performance applications but they are not that area efficient as serial architecture. Distributed arithmetic architecture can be chosen something in between serial and parallel architecture for optimum performance and area applications. Serial architecture shares one single MAC unit and parallel architectures use many MACs dependent to the order of filter as shown in Figure 3.1.

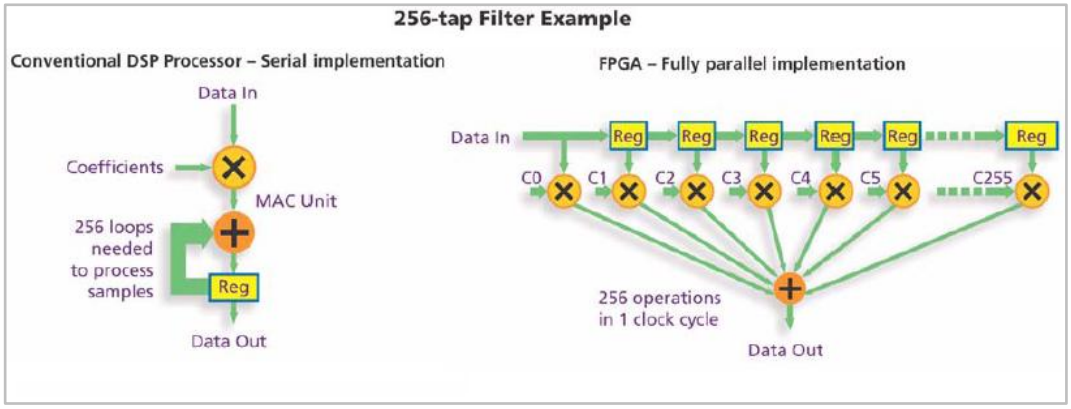

Figure 3.1 FPGAs parallel advantage to DSP for higher computational throughput. (*DSP Co-Processing in FPGAs: Embedding High-Performance, Low-Cost DSP Function, Steve Zack, Signal Processing Engineer XILINX* WP212 (v1.0) March 18, 2004)

Since new FPGA devices provide many numbers of MACs and DSP (Digital Signal Processing) blocks on board, high performance filters are easy to realize and implement.

3.1.1 FIR Filter Structure Types

Different types of architectures are used for reducing area dependent on the number of mathematical calculations in the mathematical expression of filter. These architecture types are respectively named as “direct form”, “transposed”, “symmetric” and “asymmetric”. Below are the analyses of these architectures for the following transfer function (3.1.1) given next line.

$$H(z) = Gain \times \sum_{n=0}^{M} h[n]\, z^{-n} \qquad (3.1.1)$$

Direct form is the closest structure to the mathematical expression of the filter. The number of delays are equal to the filter order M. Figure 3.2 represents the direct form structure below.

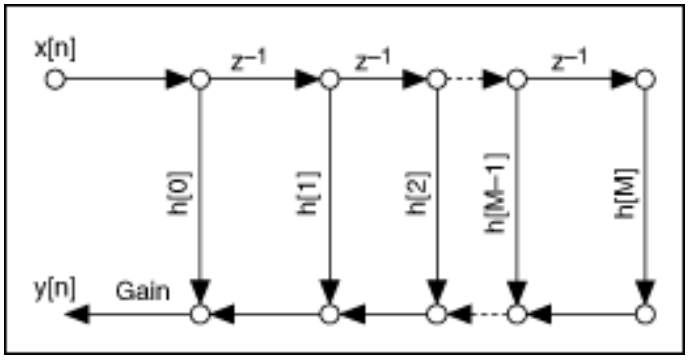

Figure 3.2 Direct Form FIR Filter structure

Transposed form structure is an alternate way to implement direct form structure. Figure 3.3 represents the transposed form structure below.

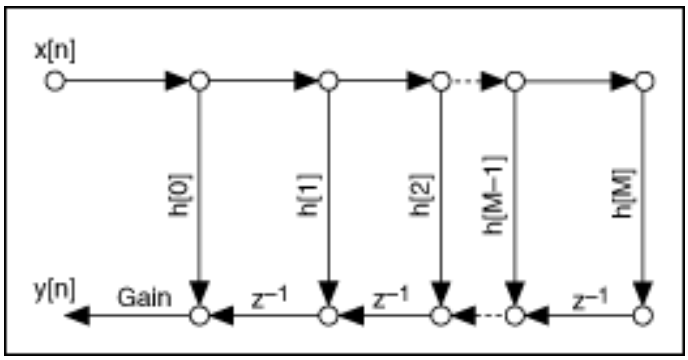

Figure 3.3 Transposed Form FIR Filter structure

Both direct form and direct form transposed structures contain the same number of delays. Simply the direct form structure has a total accumulation at the output but transposed form structure has many small additions between delay elements. Direct form structure needs extra pipeline registers between the adders to reduce the delay of the adder tree, but transposed structure does not need this pipeline registers because it already has these registers in the self structure. In the light of this information, direct form structure is more area efficient than transposed structure. If there is no limit in the area, direct form would be the better choice for the design. These structures will be analyzed in fifth chapter.

Symmetric structure is used for symmetric phase FIR filters to reduce the number of multipliers. New number of multipliers reduced from $M+1$ to $(M+1)/2$ when filter order is an odd number, and $M/2$ when filter order is an even number. Figure 3.4 represents the FIR symmetric structure when the filter order is an even number on the left, when the filter order is odd number on the right.

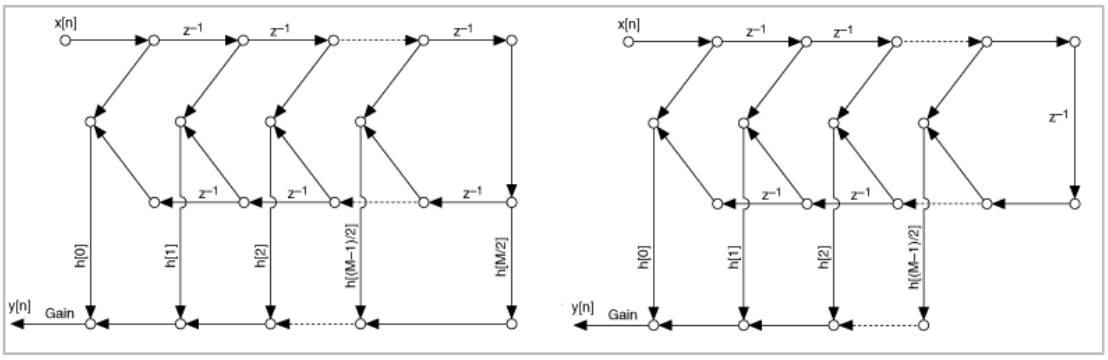

Figure 3.4 Symmetric Form FIR Filter structures with filter order number is even (left) and odd (right)

Antisymmetric structure is used for antisymmetric phase FIR filters to reduce the number of multipliers as symmetric structure. New number of multipliers reduced from $M+1$ is $M/2$ when filter order is even number, and $(M+1)/2$ when filter order is odd number. Figure 3.5 represents the FIR antisymmetric structure when the filter order is an even number on the left, when the filter order is odd number on the right.

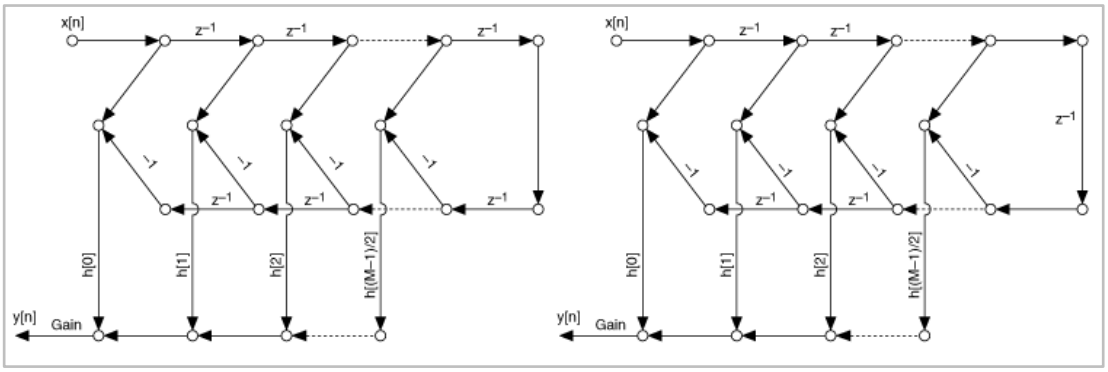

Figure 3.5 Antisymmetric Form FIR Filter structures with filter order number is even (left) and odd (right)

3.2 Implementation Methods

There is more than one way for implementing a digital filter. The important point is choosing the right one suitable for the design which is directly related to the background knowledge of the designer to save a lot of work and time.

3.2.1 IP Generator Software

IP (Intellectual Property) generators are designed to construct specific IP cores for related FPGA board. Limitations and the benefits come with these IP cores. FIR IP cores are available with most of producers support software. Generally each IP core is designed for specific board to arrange device utilization of used FPGA and very area efficient because of being specific for target device. However these IP cores do not provide user modification and implementation on another device. As an example, Xilinx System Generator or AccelDSP with MatLab Simulink could be used for IP core generating and implementing on FPGA. Figure 3.6 Illustrates the relation and design flow of using Xilinx System Generator with MatLab.

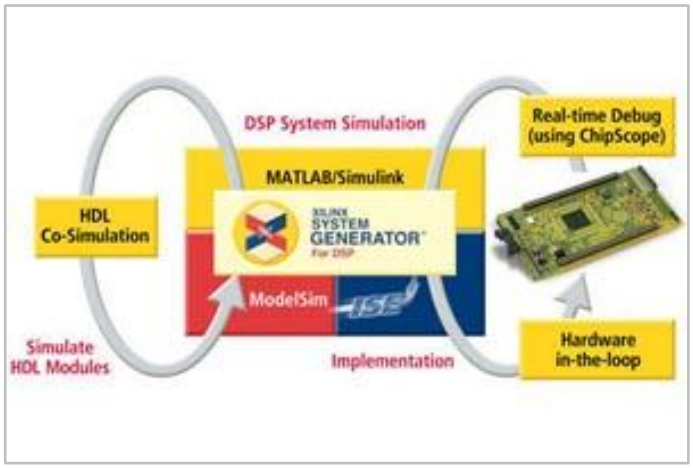

Figure 3.6 Design flow of using Xilinx System Generator with MatLab

3.2.2 User Defined Coding

Area efficiency and filter performance for specific properties makes user-defined coding better than pre-defined IP cores in specific designs. Writing own code for FIR filter allows designer to specify every property in the architecture such as symmetry, unique impulse response, coefficient type, coefficient length etc. Some IP cores provide modification to a few of these properties but not all.

User defined coding with VHDL or Verilog provides wide flexibility on the RTL (Real Time Logic) design but it costs designer a very long time. A simply 6 tap filter code written in VHDL can range 100 - 200 lines. Also specific block RAM (Random Access Memory) or DSP blocks or pipeline registers has to be considered during the coding process.

3.2.3 HDL Code Generators

HDL coders are used to generate VHDL or Verilog source code from any code written in another language or from specific inputs and selected options in the graphical user interface (GUI) for a filter. Filter Design and Analysis Tool (FDATool) or the MatLab command line can be used to design a filter and generate VHDL or Verilog code.

The Filter Design Tool can generate VHDL or Verilog codes for several filter structures such as: Direct Form or Transposed FIR, Symmetric or Anti-symmetric FIR, Direct Form I or II SOS (Second Order Section) IIR and Direct Form I Transposed SOS IIR

These IIR and FIR structures can be modeled with fixed-point and floating-point (single/double precision) realizations. Also FIR structures support unsigned fixed-point coefficients.

3.3 Chosen Method for Implementation

Designing a hardware and implement it in FPGA is a hard and time consuming task. There is a lot of way to implement a digital filter in FPGA. Important point is to specify design properties and chose the right way to get in work. There are several tools for designing process to make it easy and quick for designers. MatLab is one of these tools that can save designers time when used for designing several type of filters for implementing. Pre-defined mathematical algorithm of filter functions and HDL generation property of MatLab reduces the time to generate target architecture. This gained time can be used to implement different types of filters to analyze area and efficiency of the designs.

CHAPTER FOUR
DESIGN TEMPLATE

4.1 Basic Specifications of the Template

Design is called template because the design has a good flexibility of the bit depth of filter, filter order and filter types. Objective of the design template is to test several filter types with different orders on the FPGA, then statistically collect and compare these results. Several voice files which are considered as used in communication systems will be used with different bit depths and sampling rates. These signals are estimations of generally used or specific communication signals. Template has four modules named Receiver, Transmitter, Filter and Delayer on the FPGA side. Transmitter and Receiver can be considered one unique module as UART (Universal Asynchronous Receiver Transmitter). Design template has following tasks and features below:

1. At first the design template is considered for Spartan-3 DSP 1800A FPGA development board.

2. Communicate with PC through the RS-232 (Recommended Standard 232) serial interface.

3. Receive a voice signal from the PC each sample after previous one. Data will be in binary form so data can be applied to the FIR filter in parallel form through base register of the RS-232. Pure signal can have 5, 6, 7 or 8-bit samples.

4. Again template will have several types of FIR filters. Filtered data will be returned to the PC and then analyzed with proper software. Data from the board to the PC will be binary too.

5. To send data file to the board and receive outputs simultaneously, TeraTerm, MatLab, or HyperTerminal can be used. Baud rate of the serial port can be arranged by the clock dividing constant (CD) inside Transmitter, Delayer and

Receiver. Our board has a 125 MHz on board clock source. For the target baud rate, the clock divider constant (CD) can be calculated as:

$$CD = \frac{125000000}{Baudrate} \tag{4.1.1}$$

Design template has a flexible baud rate with data bits between 1 start bit and 1 stop bit without parity and flow control.

4.2 Design Template Overview

Several types of filters will be implemented on Spartan-3A DSP 1800A edition FPGA board of producer Xilinx. Below figure is an illustration of the design template. VHDL codes of Transmitter, Receiver, Delayer and the FIR Filter have to be changed for different baud rates and bit depths.

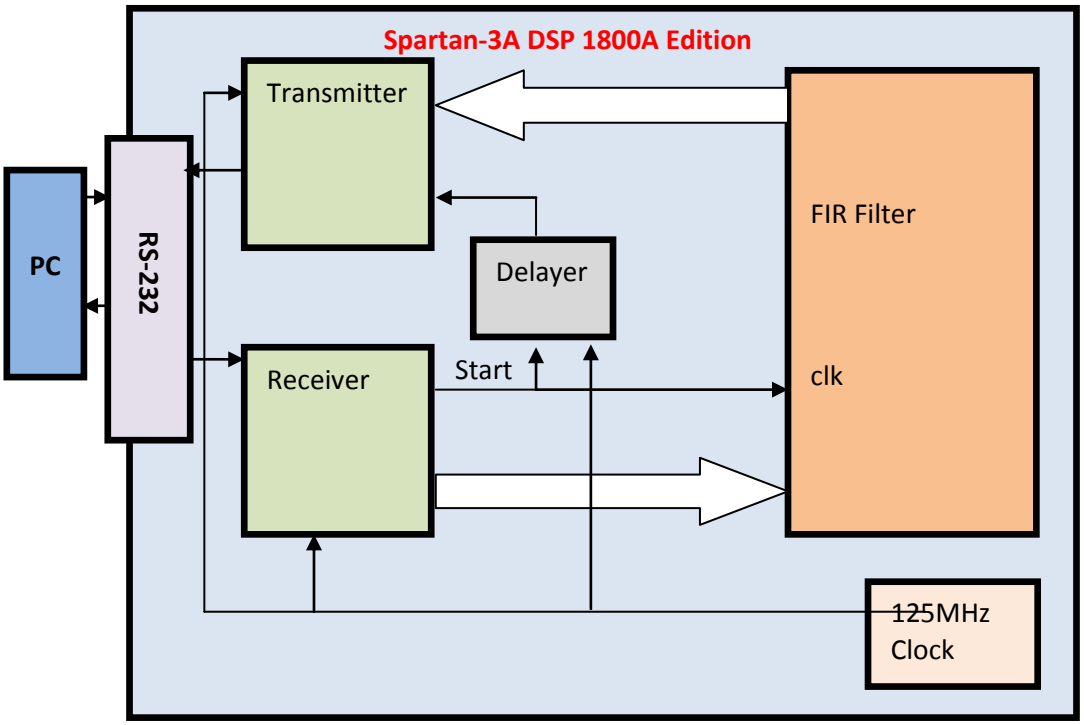

Figure 4.1 Design template overview including sub modules

In the figure above FIR filter codes will be generated with the help of FdaTool of MatLab, rest of the template, Transmitter, Receiver and Delayer modules are

constructed with user defined coding method. Actually the Transmitter and Receiver hardware can be considered together as a UART. Delayer module helps the stabilization of transmitting and receiving, by delaying the start signal *start* which interrupts transmitter for starting to transmit data, for 2 bauds. FPGA board has an on-board oscillator which produces 125MHz system clock.
Receiver has a FSM (Finite State Machine) inside to be capable of listening the port if the start of the communication bit has arrived or not. System performs following algorithms:

1- Transmission starts with the start bit which has a value of 0 on the *receive* (RX) port on Receiver.

2- After detecting first bit, data bits are collected in the temporary register of Receiver by shifting function. Register has to be at the same length of data according to the number of data bits.

3- After data bits are received, RX port is started being listened for the stop bit which has a value of 1.

4- When stop bit is detected on RX port, then data in the temporary register is shifted to the output register and the *start* output set to 1. This signal will work directly as the clock of the FIR Filter simultaneously, and interrupts the transmitter after 2 bauds with the help of Delayer module.

5- In the meaning time between *start* is 1 and detected by transmitter, FIR filter takes the N-bit data to the input register, applies filter algorithm according to the self architecture and gives the output on the output register.

6- After 2 bauds, transmitter detects the *start* signal on the *start* input ad receives the N-bit data to input register, then start and stop bits are added to this data.

7- Send the framed data in binary form and wait for next *start* signal.

Delayer is designed to arrange the data flow and clocks to stabilize the system. It gives a delay for 2 bauds to *start* signal which interrupts Transmitter in the design. This delay is considered for the FIR filter parallel computation duration, and also used for transmitter to wait until the Receiver outputs next data sample to the filter. Delayer has a FSM which has 3 states to realize the delay algorithm on the FPGA. Also delayer has a clock input and takes the clock from FPGA on-board clock source. Delayer uses clock divider constant "CD" same as Transmitter and Receiver to reduce self clock down to the target baud rate.

4.3 Design of the Receiver

For the RS-232 standard, data flow has a maximum length of 8, actually it can have a length of 5, 6, 7 bits too. There are other bits named start bit, stop bit and parity bit used to control flow of data. Start bit is the first bit which declares that transmission is started, and stop bit declares the end of the frame. Parity bit is used for controlling mechanism if any error occurred during transmission, but it is not used in the design. In addition "Baud Rate" is the number of signals sent through TX in one second Figure 4.2 below shows the data, start bit and the stop bit without parity and error bits.

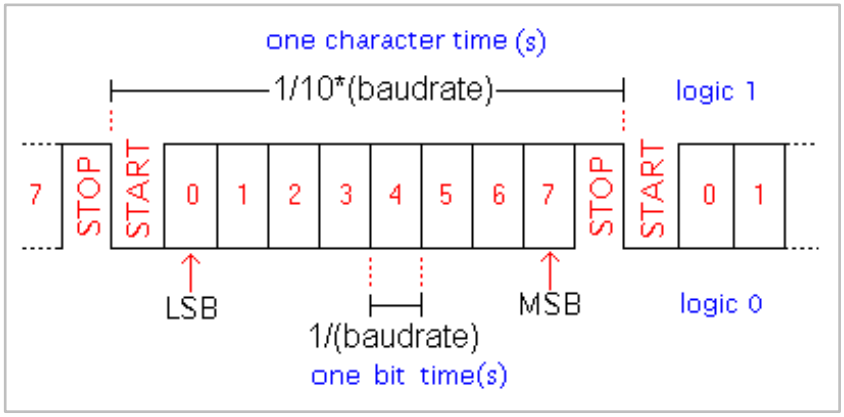

Figure 4.2 Baudrate and character time on serial port

As seen in the figure start bit is logic0 and stop bit is logic1. Length of the stop bit can be arranged 1, 1.5 or 2 bit length. In the design it is considered as 1 bit length. First bit sent after the start bit is always least significant bit. Time elapsed for a data

frame to be sent can be calculated by dividing 1 to the multiply of baud rate and number of bits. As an example for 8 bit frame length and 1 bit start, 1 bit stop and no parity at 9600 baud rate communication; time period for 1 bit will be 1/9600 = 104.1µs.

In the design of the receiver, specifications are considered as 8 bit data frame, 1 start bit, 1 stop bit and no parity control. Baud rate is not specified as a constant, it can be changed manually by changing the *CD* in the VHDL codes of the design. Baud rate for the system is gathered by dividing the on-board clock of the FPGA board. However if the division has a value in the fraction, it is rolled to an integer and the synchronization loss is unavoidable. This is because of the basic principle of synchronization, clock sources should be same but PC has its own clock generator and FPGA board too. Solution for the unavoidable synchronization loss lays under considering both hardware and software practically. Based on the FSM of the receiver, if there is no start bit for 1 baud after stop bit is received, then state counter sets itself to zero and returns the idle state again. So if there exists at least 1 baud delay before sending the next frame in the PC side, synchronization can never be lost. This solution is considered in the software part of the design. The flowchart of receiver algorithm is shown in Figure 4.3.

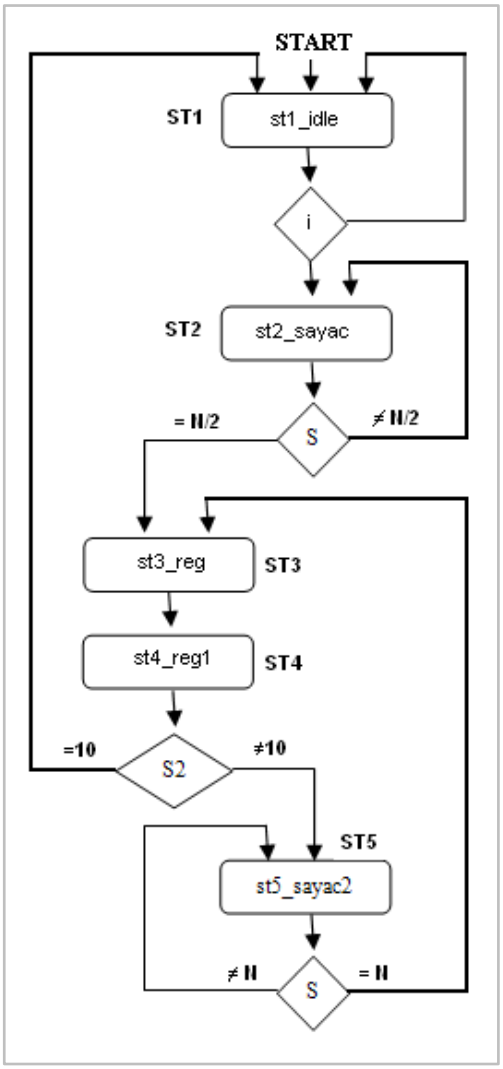

Figure 4.3 Flowchart of the Receiver

In the figure some of the constants are named shorter because of figure area. In VHDL codes of the Receiver, N is actually clock divider constant, S and S2 are start and start2 respectively. In the system start-up, the Receiver starts in the st1_idle state. In this state the RX port is listened if there is a start bit detected system goes to the st2_sayac state, if not returns to the st1_idle state again. In the st2_sayac state, counter named sayac counts from 0 to the CD/2 and jumps to the st3_reg state. In this state s_reg_en register is triggered to 1 and sayac2 is increased by 1. Then next state is st4_reg1 and in this state s_reg_en register is triggered to 0 and 0 value is given to sayac. Then system goes to st5_sayac and in this state sayac is increased by 1. When sayac is equal to CD system jumps to the st3_reg, otherwise it returns back to st5_sayac2 again. With this state machine structure, data bits are received and

stored in a specific register which has the same length as the data frame. Then the data is shifted to the output register. The simulation result for the Receiver is shown in Figure 4.4 below. Simulation is produced by ISIM which is a software of Xilinx company.

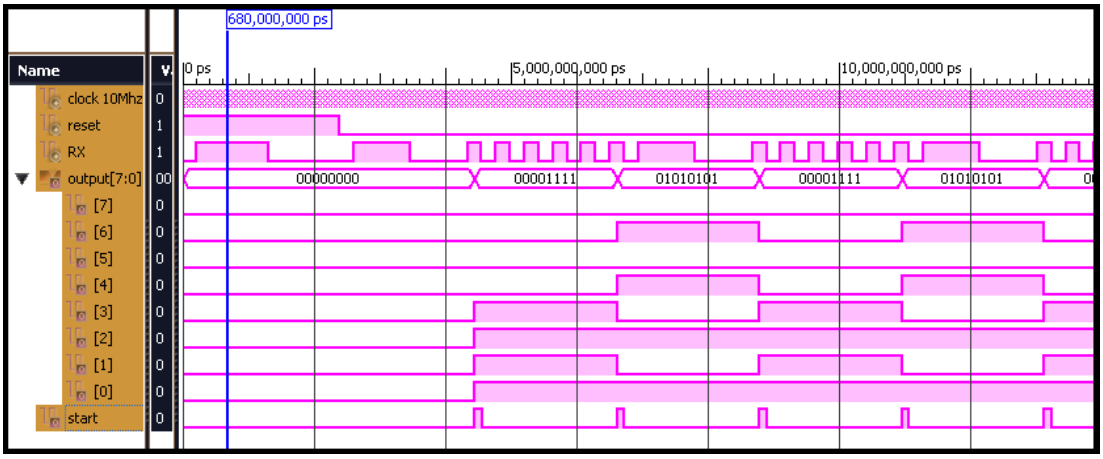

Figure 4.4 Simulation results from ISIM for the Receiver

Clock frequency is arranged 10 MHz for simulation, and it can be easily understood that reset is considered active-high. When reset level is high, there is no output even though there is a serial input on RX. After reset level is low. Then system starts to work and waits for the start bit which is 0. After detecting the first start bit and collecting 10 bits which has a value as “1000011110”, the frame without start bit and the stop bit “00001111” is sent through the output at the same time the start signal is set to 1for ½ baud which interrupts the Fir Filter and Transmitter.

4.3.1 Pseudo Code for Receiver

Set start with the value of stop_en

Set the output1(0:9) with the value of register reg_en(9:0)

Set output(7:0) with the register c(7:0)

If there is a clock event and clock is equal to 1

If reset is equal to 1

Set current state is st1_idle

Else

Set current state to next state

If there is a clock event and clock is equal to 1 and reset is equal to 0

If s_reg_en is equal to 1

Set the MSB of reg_n with incoming bit and shift reg_n through LSB

If there is a clock event and clock is equal to 1

If stop_en is equal to 1

Increment sayac_stop by 1

Else

Set sayac_stop to 0

If there is a clock event and clock is equal to 1

If current state is st1_idle

Set C(0 to7) with the output1(1 to 8)

If the current state is state st1_idle and If there is start bit 0 has been detected

Go to the state st2_sayac

Else

Stay in the state st1_idle wait for the start bit to be detected

Case of states are

When the current state is st2_sayac

Increment the sayac by 1

If sayac_stop is equal to half of CD

Set the stop_en register to 0

If sayac is equal to half of CD

Go to st3_reg state

When current state is st3_reg

Set st3_reg_en register to 1

Increment the sayac2 counter by 1

If sayac_stop is greater than half of CD

Set the stop_en register to 0

Go to st4_reg1 state

When current state is st4_reg1

Set s_reg_en to 0

Set sayac to 0

If sayac_stop is greater than half of CD

Set stop_en to 0

If sayac2 is equal to 10

Set m to 1

Set stop_en to 1

Go to st1_idle

Else

Go to st5_sayac2

When current state is st5_sayac2

Increment sayac by 1

If sayac is equal to CD

Go to is st3_reg state

If there is a clock event and clock is equal to 1

If sayac2 is smaller than eight

Set enable to 1

Else

Set enable to 0

4.4 Design of the Delayer

As it is mentioned before, delayer works for the synchronization of receiver and transmitter modules to prevent data loss. It has two inputs named *clock* and *rec_out*, and has an output named *tra_in.* The pin *rec_out* is connected to the *start* which is output pin of the Receiver. The output signal *start* which comes from the receiver is delayed by 2 bauds with the Delayer module. Delayer has its own processes and also has a FSM which has three states. These states are st1_gor, st2_bekle and st3_ver. Delayer also arranges its baud rate by dividing the board clock by CD with the counter inside which holds value of CD inside. Flowchart of the Delayer is shown in Figure 4.5 below.

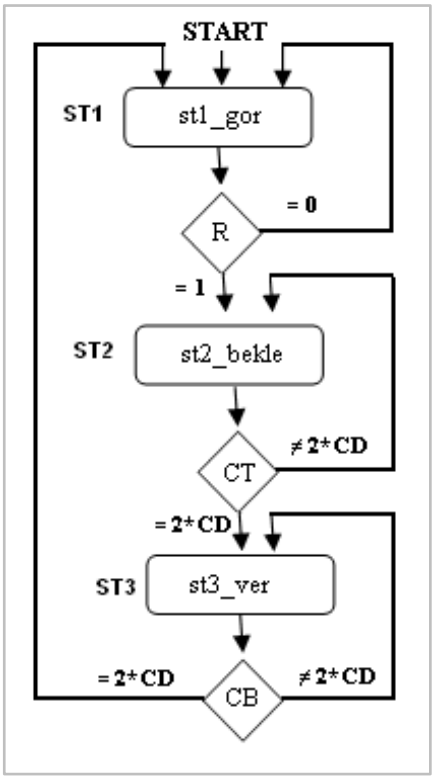

Figure 4.5 Flowchart of the Delayer

In the Figure some coefficients are named shorter because of the area of figure. "R" represents *rec_out* pin, "CT" represents *counttut* and "CB" represents *countbekle* which are all in the VHDL code of the Delayer.

System starts at the *st1_gor* state and in this state system listens to the *rec_out* pin if there is any rising edge of *start* signal. If there is not, then it returns to *st1_gor* state again, if there is a rising edge of the *rec_out* signal, then system goes to *st2_bekle* state. In this state system waits for two times the baud rate and then system goes to *st3_ver*. In the *st3_ver* state system waits for two times the baud rate. In this duration *tra_in* signal is set to 1. Then system goes to *st1_gor* state and *tra_in* signal is taken down to 0 again. Simulation result obtained from ISIM is shown in Figure 4.6.

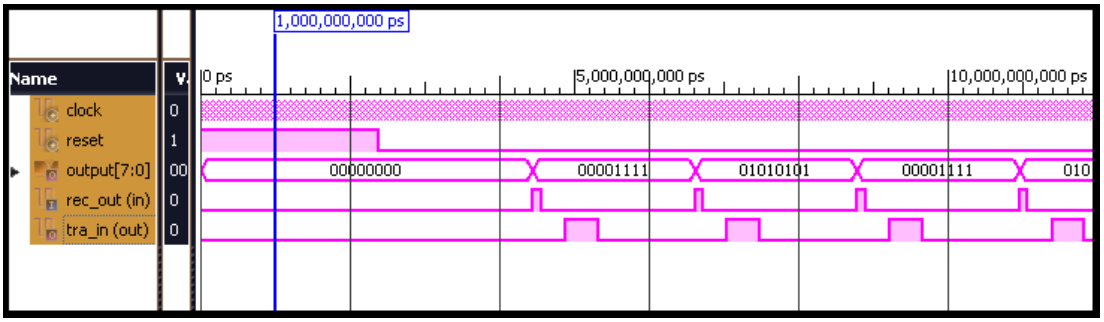

Figure 4.6 Simulation results of the Delayer

In the Figure 4.6 above *rec_out* is input of the delayer and *tra_in* is the output of the delayer. The signal *output(7:0)* shows the output of Receiver. It is obviously seen that the behavior of the Delayer is exactly matches with the simulation results. There is a delay of 2 bauds between the input and output; also the duration of output is increased to 2 bauds.

4.4.1 Pseudo Code for Delayer

Set tra_in with the valu of counttut_en

Set rec_out_ctrl with the value od rec_out

If there is a clock event and clock is equal to 1

If countbekle_en is equal to 1

Increment coutbekle by 1

Else

Set countbekle to 0

If counttut_en is equal to 1

Increment counttut by 1

Else

Set counttut to 0

If there is a clock event and clock is equal to 1

Case of states are

When current state is st1_gor

Set counttut_en and countbekle_en both to 0

If rec_out is equal to 1

Go to st2_bekle

When current state is st2_bekle

Set countbekle_en to 1

If countbekle is equal to two times the CD

Go to st3_ver

When current state is st3_ver

Set countbekle_en to 0 and set counttut_en to 1

If counttut is equal to to times th CD

Go to st1_bekle

4.5 Design of the Transmitter

In the design the Transmitter data frame length has been set the same as receivers to make a working design. It is easier to create transmitter because is works simpler than the Receiver. Transmitter takes whole bunch of 8 bits, adds start and stops bits to this frame and sends it one by one through *TX* pin of the design. Transmitter understands time to take parallel data from its input register with the help of the input pin *start* which is connected to the output pin *tra_in* of the Delayer. Transmitter has a FSM like Receiver and Delayer inside too. States of the FSM in the Transmitter are *st1_a*, *st2_b*, *st3_c*, *st4_d.* Design waits in the *st1_a* state until there is a rising edge on the input *start*. Then system goes to the next state *st2_b* and starts to rotate between four states with the help of counters and registers until all bits are sent through TX one by one. Then system returns to the *st1_a* state where everything has started. System flowchart of the Transmitter is shown in Figure 4.7.

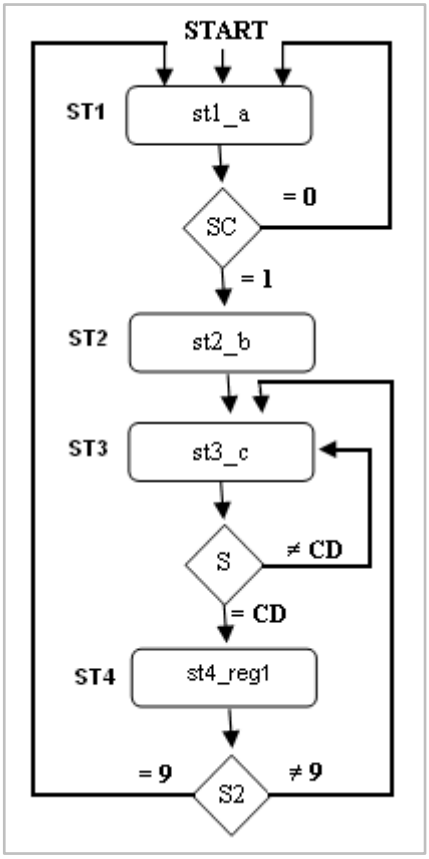

Figure 4.7 Flowchart of the Transmitter

In the figure above, some names are written shorter because of the figure area. "SC" represents the *start_ctrl* register in the VHDL code. "S" and "S2" represent *start* and *start2* respectively. As already known CD is the clock divider constant which is same in every part of the whole design template. In the Transmitter algorithm system goes from *st2_b* to *st3_c* without a query. This is because there is no need for a query between these states. These states are constructed because a few counters are needed in separate states.

System starts in the *st1_a* state and waits for the rising edge on the input pin *start.* When the rising edge of the *start* is detected, system goes to *st2_b* state. In this state *sayac* starts to count from zero then goes to the next state *st3_c*. In *st3_c* state, *load_enable* register is set to 0 and system goes to the next state *st4_d*. In *st4_d* state, *sayac2* starts to count from zero, *sayac* is set to 0 and *s_reg_en* register is set to 1. In this state, system always returns to *st3_c* state until *sayac2* reaches 9. When *sayac2*

is equal to 9, then system returns to *st1_a* state. Simulation result for Transmitter obtained from ISIM is shown in Figure 4.8.

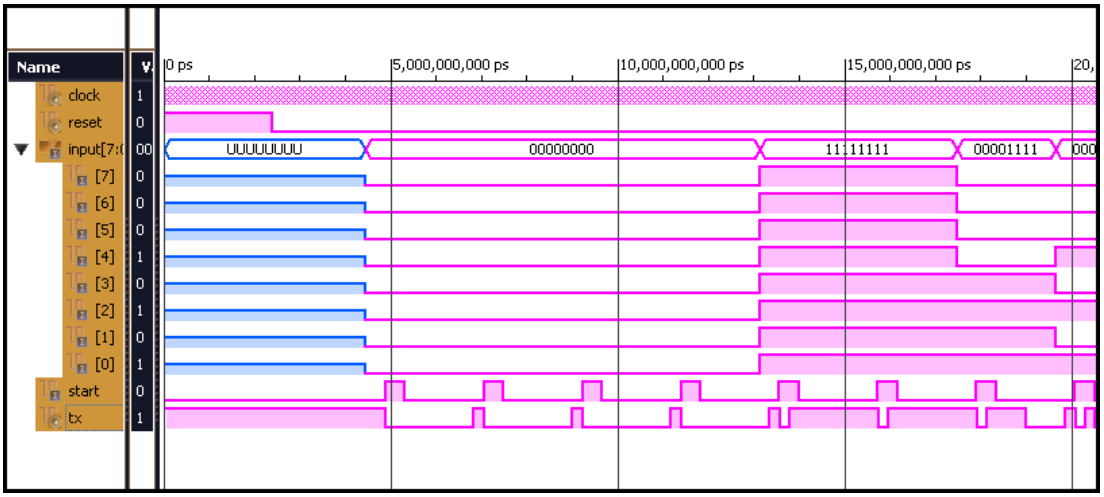

Figure 4.8 Simulation results of the Transmitter

In the figure *tx* is the serial output and *input[7:0]* is the parallel input of Transmitter. Whole system does not start to work until *reset* gets 0. It is seen that after system wtarts to work, Transmitter is interrupted with signal *start* and sends the parallel input one by one after adding start and stop bits to the frame. It is considered for Transmiter to send LSB (Least Significant Bit) of the frame at first always.

4.5.1 Pseudo Code for Transmitter

Set reg(7:0) with the input(0:7)
If there is a clock event and clock is equal to 1
    Set output with the reg_n(0)
If there is a clock event and clock is equal to 1
    If reset is equal to 1
        Go to st1_a state
    Else
        Current state is next state
If there is a clock event and clock is equal to 1
    If start is equal to 1

Set start_ctrl to1

If sayac2 is equal to 9

Set start_ctrl to 0

If there is a clock event and clock is equal to 1

If start_ctrl is equal to 1

If load_enable is equal to 1

Set reg_n(9) with 1

Set reg_n(1:8) with reg(7:0)

Set reg_n(0) with 0

If reg_en is equal to 1

Shift reg_n to through LSB one bit and set MSB with 0

Else

Set reg_n(0) with 1

If there is a clock event and clock is equal to 1

Case of states are

When current state is st1_a

Set load_enable to 0

Set sayac to 0

Set sayac2 to 0

Set s_reg_n to 0

If start_ctrl is equal to 1

Go to st2_b state

When current state is st2_b

Set load_enable to 1

Increment sayac by 1

Go to st3_c state

When current state is st3_c

Set load_enable to 0

Set s_reg_en to 0

Increment sayac by 1

If sayac is equal to CD

Go to st4_d state

When current state is st4_d

Set sayac to 0

Set s_reg_en to 1

If sayac2 is equal to 9

Go to st1_a

Else

Go to st3_c

4.6 Design of FIR Filters

For the complete design, FIR Filter is considered to have 8-bit input data length and 8-bit output respectively. For the system to work and MatLab calculate correct coefficients for the desired filter, sampling frequency must be exactly same with the sampling frequency of the data. Other variables of the FIR filter like order, fraction length etc. can be changed for several examinations. As FIR filter is considered fully parallel architecture for the whole design, so its behavior can be explained easily. Clock signal and clk_enable signal of FIR filter is connected to the start signal which is output of Receiver. This signal interrupts FIR filter to take new 8 –bit frame from temporary register. Fir filter takes this 8-bit frame in its pipeline, multiplies each sample with coefficients and gices an output from its output register. These operations take one clock of clock which is connected start pin of Receiver.

Simulation tested for a FIR filter with following specifications:

Response Type:	Low-pass;
Characteristic:	Equiripple;
Filter Order:	5;
Architecture:	Direct-Form 1;
Density Factor:	20;
Pass band frequency:	1600 Hz;
Stop band frequency:	2000 Hz;
Sampling Frequency:	8000 Hz;
Input Data Length:	8-bit;

Output Data Length: 8-bit;

Simulation result obtained from ISIM for the FIR Filter with above specifications is shown in Figure 4.9 below.

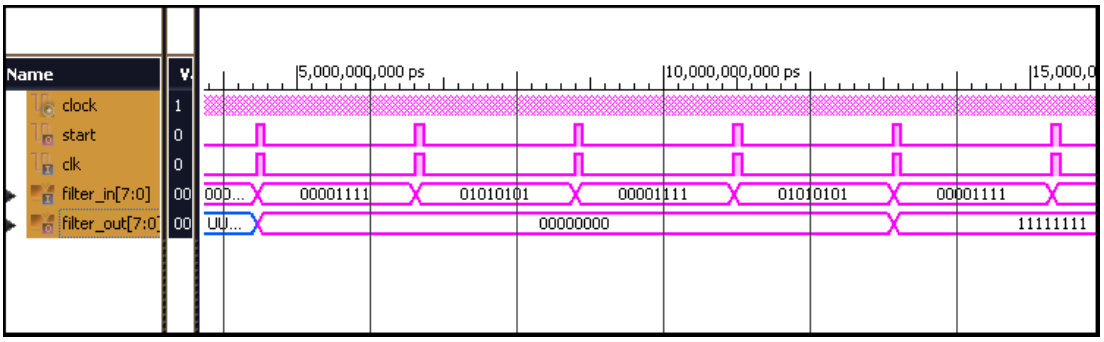

Figure 4.9 Simulation results for FIR Filter with specifications on previous page

In the figure above it is clearly seen that signal *start* and *clk* of FIR Filter is connected. *Start* works as clock source of the Filter. In the time axis, first few outputs are not reliable, precise outputs are given after all taps are fully loaded with inputs. Because FIR filter use windowing method and for reliable results; window elements must not be empty.

CHAPTER FIVE
PRE-IMPLEMENTATION

5.1 Overview of the FPGA Board

Before implementing the whole design, FPGA board which will be used should be deeply analyzed. Spartan-3A DSP 1800A type Spartan-3A DSP family FPGA will be used for implementation. FPGAs in this family solve the designs which need cost-sensitive, high-volume and high-performance DSP applications. Spartan-3A DSP 1800A has 1.8 million system gates inside. Package marking of the FPGA is shown below in Figure 5.1.

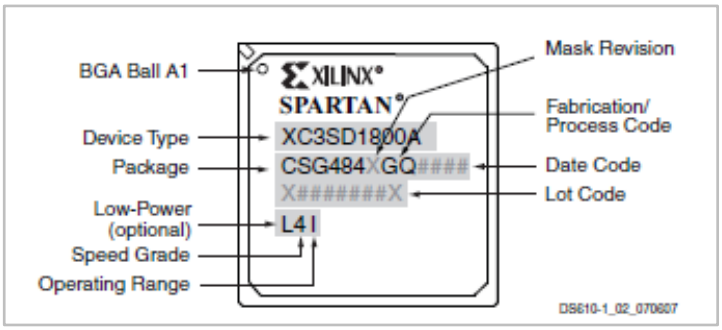

Figure 5.1 Spartan-3A DSP FPGA family package marking codes

The Spartan-3A DSP family which has a new feature, XtremeDSP™ DSP48A slices, and combined with proven 90 nm technology, it delivers more functionality and bandwidth per dollar than ever before which results a new standard in the programmable logic and DSP processing industry.

The XC3SD1800A is reconstructed for DSP applications and has additional block RAM and XtremeDSP DSP48A slices. The XtremeDSP DSP48A slices included in the Spartan-3A devices are based on the DSP48 blocks used in the Virtex®-4 devices. Both the block RAM and DSP48A slices in the Spartan-3A DSP devices run at 250 MHz in the lowest cost, standard -4 speed grade.

5.2 Board Features

The Spartan-3A DSP Starter Platform provides the following features:

- Xilinx 3SD1800A-FG676 FPGA
- Clocks
  - LVTTL oscillator socket
  - 125 MHz LVTTL (Low Voltage Transistor-Transistor Logic) SMT oscillator
  - 25.175 MHz LVTTL SMT (Surface Mount) oscillator (video clock)
- Memory
  - 32M x 32 (128 MB) DDR2 (Double Data Rate) SDRAM (Synchronous Dynamic Random Access Memory)
  - 16Mx8 parallel and BPI (Byte-wide Peripheral Interface) configuration flash
  - 64Mb SPI (Serial Peripheral Interface) Configuration and Storage Flash (with 4 extra SPI selects)
- Interfaces
  - 10/100/1000 PHY (Physical Layer)
  - JTAG (Joint Test Action Group) programming and configuration port
  - RS232 Port
  - Low-cost VGA (Video Graphics Array)
- Buttons and switches
  - 8 User LEDs (Light Emitting Diode)
  - 8-position user DIP (Dual In-line Package) switch
  - 4 User push button switches
  - Reset push button switch
- User I/O (Input/Output) and expansion
  - Digilent 6-pin header (2)
  - EXP expansion connector (2)
- Configuration and debug
  - JTAG
  - SystemACE™ module connector

- Eridon debug connector (SATA (Serial Advanced Technology Attachment))

A high-level block diagram of the Spartan-3A DSP Starter Platform is shown in Figure 5.2. Sections shown as boxes describe details of the board design.

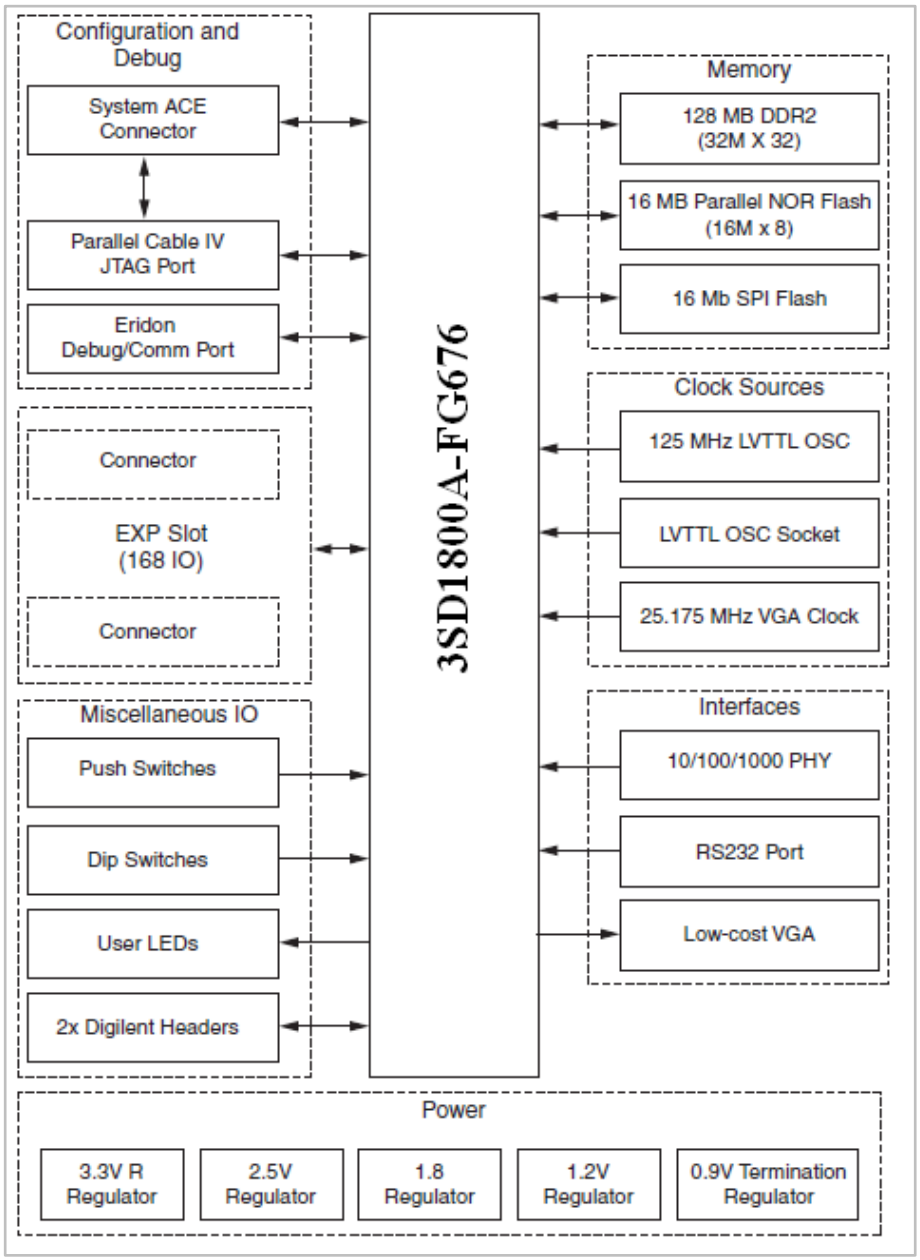

Figure 5.2 Spartan-3A DSP Starter Platform block diagram

5.2.1 Memory

The Spartan-3A DSP Starter Platform includes high-speed RAM (128Mbytes DDR2) and non-volatile ROM (Read Only Memory) (16Mbytes parallel, and 64Mbit serial). Additionally, a 50-pin connector is provided for SystemACE interface (not included) that can be used to configure the Spartan-3A DSP FPGA, and to provide storage for A/V (Audio/Video) media files from removable Compact Flash cards.

5.2.2 Clock Sources

There are four clock sources provided on the Spartan-3A DSP Starter Platform:

- A 125 MHz oscillator connected to GCLK7 (Global Clock) (Used in design).
- A 25.175 MHz oscillator (primarily for VGA timing) connected to RHCLK2 (Right Half Clock).
- A socket for a half-can oscillator connected to GCLK14.
- The user must install this oscillator.
- An SMA connector footprint (J1) connected to GCLK4. The user must install this connector*.

The clock sources are listed and described in Table 5.1.

Table 5.1 Clock sources of Spartan-3A DSP Starter Platform

Clock Source	FPGA Pin No	Part Number
125 MHz oscillator (U7)	F13	Fox FXO-HC535-125.000
25.175 MHz oscillator (U4)	P26	Fox FXO-HC530-25.175
Socket	AE13	Populate with Fox 350LF osc.
SMA connector J1*	K14	Tyco-AMP part #221789-3

5.2.3 10/100/1000 Ethernet PHY

The Spartan-3A DSP development board provides a 10/100/1000 Ethernet port for network connection. The PHY is a low power version of the National Gig PHYTER V with a 1.8V core voltage and 2.5V I/O voltage. The PHY is connected to a RJ-45

jack, which is connected to two LEDs and their relevant resistors. An external logic is used to indicate 10, 100 and 1000 Mb/s to drive a LED on the RJ-45 jack. This external logic may not work if the configuration of strap is changed. Four more LEDs are provided on the board for status indication. These LEDs indicate speed at 10 Mb/s, speed at 100 Mb/s, speed at 1000 Mb/s and Full Duplex operation. PHY has its own clock generated from its own 25 MHz crystal.

5.2.4 RS-232

The board provides an RS-232 connector for serial communication. The RS-232 transceiver is a Texas Instruments MAX3221 device. It operates at 3.3V with an internal charge pump. The RS-232 interface is carried out by DB9 connector P2. Only null-modem serial cables are supported by this RS-232 interface. A male-to-female serial cable should be used to connect J11 with PC serial port (male DB9). Table 5.2 shows the FPGA pin-out for the RS232 interface.

Table 5.2 RS232 signals relevant pin codes on FPGA

Net Name	Description	FPGA Pin Number
FPGA_RS232_Rx	Receive data, RD	N21
FPGA_RS232_Tx	Transmit data, TD	P22

5.2.5 VGA Output

The Spartan-3A DSP Starter Platform includes a VGA video output port which uses a resistor-divider network and 4-bits for each RGB color. This resistor-divider network is constructed with 510, 1K, 2K, & 4K ohms for each color. A 25.175 MHz clock (VGA-resolution) is included on the board, connected to the FPGA at RHCLK2 (P26) pin. For timing the output and generating the image and syncs, this clock should be used in the FPGA controller.

5.2.6 Miscellaneous I/O

Spartan-3A DSP Starter Platform has an 8-position DIP switch, 4 user Pushbuttons, and 8 user LEDs. The connections of these devices to the FPGA pins are shown in Table 5.3. Each DIP switch is pulled low in the "OFF" position. Turning the switch "ON" causes the corresponding FPGA pin to be pulled up to VCC_0 (Common Collector Voltage) of the FPGA bank. The four pushbuttons are also initially low and pressing any button will cause the corresponding FPGA pin to be driven to the value of VCCO_0. Driving a "High" to the LEDs will cause them to light.

Table 5.3 Relevant pin assignments for DIP Switch, Push Buttons and LEDs on FPGA

Device	Name	FPGA Pin
DIP Switch	SW3.1, SW3.2	A7, G16
	SW3.3, SW3.4	E9, D15
	SW3.5, SW3.6	D19, B24
	SW3.7, SW3.8	A5, A23
Push Buttons	SW5(PB1), SW6(PB2)	J17, J15
	SW7(PB3), SW8(PB4)	J13, J10
LEDs	LED1(D14), LED2(D13)	P18, P25
	LED3(D12),LED4(D11)	N19, K22
	LED5(D10), LED6(D9)	H20, G21
	LED7(D8), LED8(D7)	D24, D25

5.3 Utilization of Resources on FPGA Board

As mentioned on chapter three, this utilization results are used to choose right filter type for the corresponding purpose. To investigate the resources used inside FPGA, simulation result from ISE software is a useful tool. For Spartan-3A DSP 1800A FPGA board, the detailed usage of resources results after implementing low pass equiripple FIR filter for several orders and structures shown in following tables.

Table 5.4 Device utilization numbers and ratios for 5th order FIR filters for three different architectures

Number of Resources in Spartan-3A DSP 1800	Available	Architecture of 5th order FIR Filter		
		Direct	Transposed	Symmetric
		Used	Used	Used
Slice Flip Flops	33,280	80 – (1%)	169 – (1%)	112 – (1%)
4 input LUTs	33,280	80 – (1%)	186 – (1%)	93 – (1%)
Occupied Slices	16,640	72 – (1%)	111 – (1%)	96 – (1%)
Bonded IOBs	519	35 – (6%)	35 – (6%)	35 – (6%)
BUFGMUXs	24	1 – (4%)	1 – (4%)	1 – (4%)
DSP48As	84	6 – (7%)	3 – (3%)	3 – (3%)

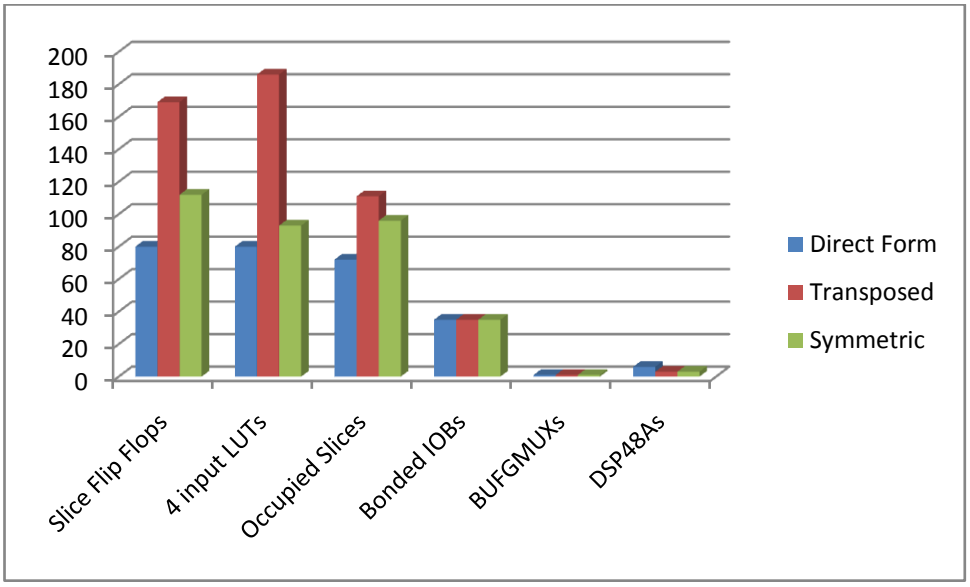

Figure 5.3 Compare of filter architectures according to device utilizations in Table 5.4

It is seen in Figure 5.3 that direct form architecture uses most DSP48 blocks than other architectures. But on other resources, direct form architecture uses minimum amount. Usage of bounded IOBs and BUFGMUXs (Global Clock Multiplexer) is same for all structures. These results are just for fifth order low-pass FIR filter and can not be generalized.

Table 5.5 Device utilization numbers and ratios for 10th order FIR filters for three different architectures

Number of Resources in Spartan-3A DSP 1800	Available	Architecture of 10th order FIR Filter		
		Direct	Transposed	Symmetric
		Used	Used	Used
Slice Flip Flops	33,280	160 – (1%)	323 – (1%)	192 – (1%)
4 input LUTs	33,280	338 – (1%)	341 – (1%)	186 – (1%)
Occupied Slices	16,640	246 – (1%)	190 – (1%)	184 – (1%)
Bonded IOBs	519	35 – (6%)	35 – (6%)	35 – (519)
BUFGMUXs	24	1 – (4%)	1 – (4%)	1 – (4%)
DSP48As	84	11 – (13%)	6 – (7%)	6 – (7%)

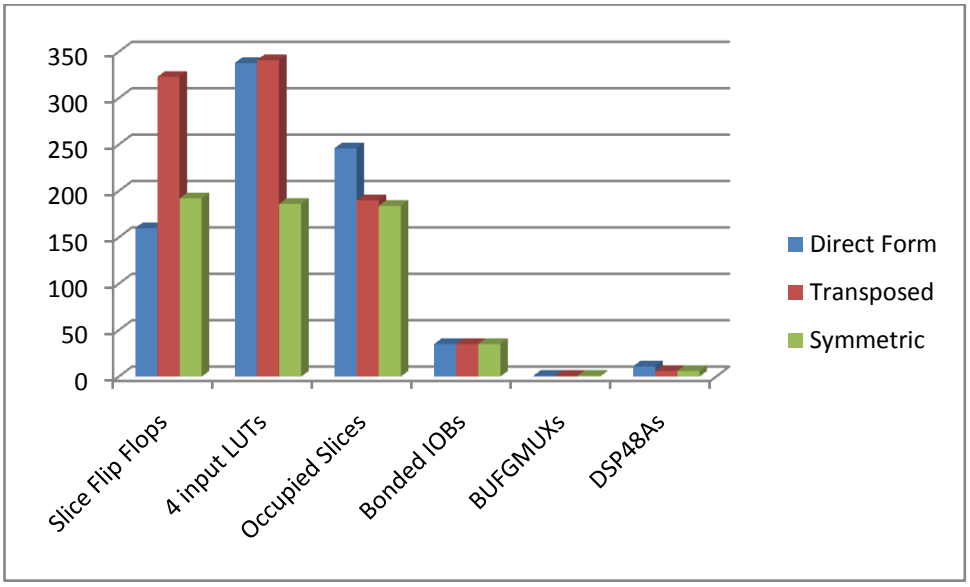

Figure 5.4 Compare of filter architectures according to device utilizations in Table 5.5

It is seen in Figure 5.4 that direct form and transposed architectures are closely use resources inside FPGA. The clear difference between them is usage of DSP48 blocks. Symmetric form seems most homogeneous structure on usage of resources. It also uses the same amount of DSP48 blocks as transposed structure. Usage of bounded IOBs and BUFGMUXs is same for all structures. These results are just for tenth order low-pass FIR filter and can not be generalized.

Table 5.6 Device utilization numbers and ratios for 20th order FIR filters for three different architectures

Number of Resources in Spartan-3A DSP 1800	Available	Architecture of 20th order FIR Filter		
		Direct	Transposed	Symmetric
		Used	Used	Used
Slice Flip Flops	33,280	320 – (1%)	630 – (1%)	352 – (1%)
4 input LUTs	33,280	641 – (1%)	653 – (1%)	341 – (1%)
Occupied Slices	16,640	482 – (2%)	351 – (2%)	344 – (2%)
Bonded IOBs	519	35 – (6%)	35 – (6%)	35 – (6%)
BUFGMUXs	24	1 – (4%)	1 – (4%)	1 – (4%)
DSP48As	84	21 – (25%)	11 – (13%)	11 – (13%)

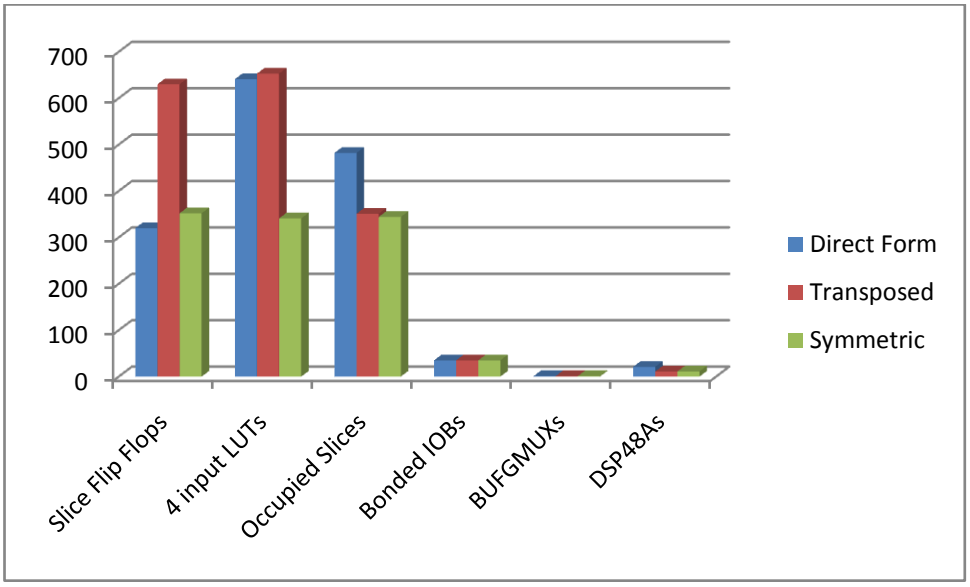

Figure 5.5 Compare of filter architectures according to device utilizations in Table 5.6

In the Figure 5.5 above it is seen that direct form and transposed architectures are closely use resources inside FPGA. The clear difference between them is usage of DSP48 blocks. Symmetric form seems most homogeneous structure on usage of resources. It also uses the same amount of DSP48 blocks as transposed structure. Usage of bounded IOBs and BUFGMUXs is same for all structures. These results are just for twentieth order low-pass FIR filter and can not be generalized.

Table 5.7 Device utilization numbers and ratios for 40th order FIR filters for three different architectures

Num. of Resources in Spartan-3A DSP 1800	Available	Architecture of 40th order FIR Filter		
		Direct	Transposed	Symmetric
		Used	Used	Used
Slice Flip Flops	33,280	672 – (2%)	1,246 - (3%)	672 – (2%)
4 input LUTs	33,280	1,273 – (3%)	1,281 - (3%)	745 – (2%)
Occupied Slices	16,640	986 – (5%)	686 – (4%)	714 – (4%)
Bonded IOBs	519	35 – (6%)	35 – (6%)	35 – (6%)
BUFGMUXs	24	1 – (4%)	1 – (4%)	1 – (4%)
DSP48As	84	33 – (39%)	17 – (20%)	17 – (20%)

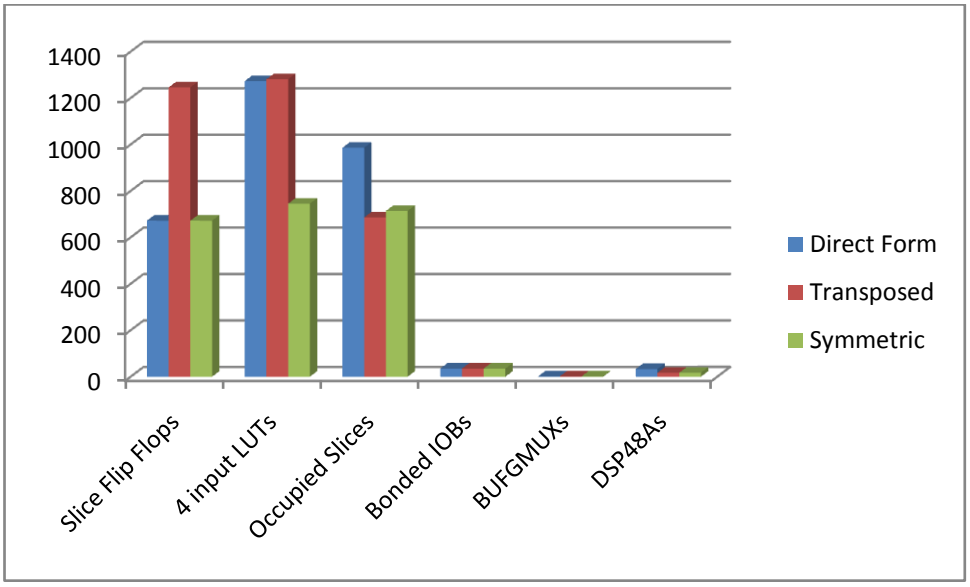

Figure 5.6 Compare of filter architectures according to device utilizations in Table 5.7

In the above it is seen that direct form and transposed architectures are closely use resources inside FPGA. The clear difference between them is usage of DSP48 blocks. Symmetric form seems most homogeneous structure on usage of resources. It also uses the same amount of DSP48 blocks as transposed structure. Usage of bonded IOBs and BUFGMUXs is same for all structures. These results are just for fortieth order low-pass FIR filter and can not be generalized

Thus far the structures are compared for several orders and number of used BUFGMUXs and bonded IOBs came out same for all. Without these resources, results are compared again all in one graph which can be seen in Figure 5.7.

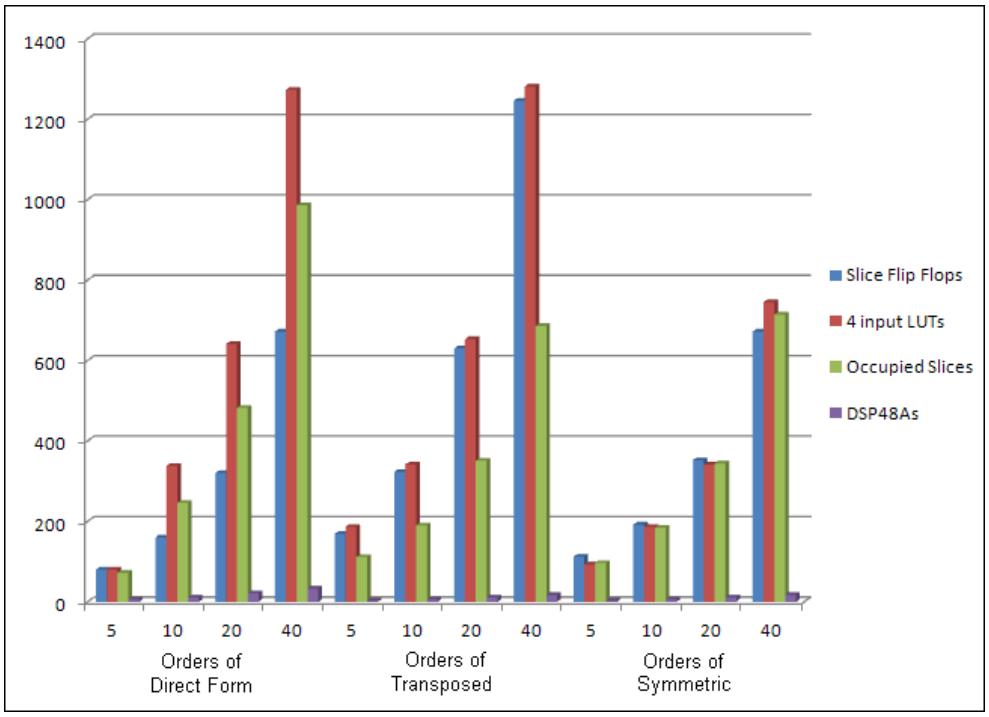

Figure 5.7 Compare of filter architectures and orders according to device utilizations in Table 5.4, Table 5.5, Table 5.6 and Table 5.7

By analyzing the simulation result it can be said that for the same filter order, transposed architecture takes larger resources than the direct architecture except DSP48 blocks. Symmetric hardware architecture takes lower resources than the transposed and direct architectures. That's because symmetric architecture shares some component in common and number of coefficients in it is almost half of the direct architecture. Also it is obviously seen that resource usage increases as the order of filter increases.

CHAPTER SIX
DATA COMPARISON

6.1 PSTN Signals

PSTN (Public Switched Telephone Network) is a circuit-switched network that is used generally for voice communications. At first was a fixed-line analog telephone systems network. But now PSTN is almost digital worldwide.

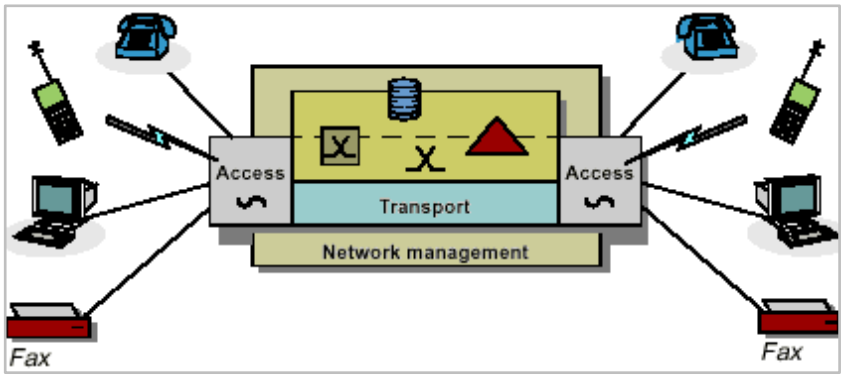

Figure 6.1 Public Switched Telephone Network

The basic frame in the PSTN is a 64 kbps channel, which is "DS0" or Digital Signal 0. DS0's are actually timeslots because they are multiplexed together in a time-division frame. The audio sound in PSTN is changed to digital at 8 kHz sample rate and it is received out by 8-bit PCM.

Multiple DS0's can be multiplexed together on higher capacity circuits, such that 32 DS0's make an E1.

These time slots are transported from the initial access point to last access point over a group of electronic equipment which is known as the access network. This network uses SDH (Synchronous Digital Hierarchy) or SONET (Synchronous Optical Networking) technology. In some areas older PDH (Plesiochronous Digital Hierarchy) technology is still used.

For a long time period, the PSTN was the only available network for telephony. Now other services are integrated with PSTN, such as ISDN (Integrated Service Digital Network), DSL (Digital Subscriber Line), ATM (Asynchronous Transfer Mode), frame relay and the Internet VOIP (Voice Over Internet Protocol).

6.1.1 Fundamentals of PCM in PSTN

As mentioned in previous page, the audio sound in PSTN is changed to digital at 8 kHz sample rate using 8-bit PCM. PCM stands on some techniques.

6.1.1.1 Sampling

The sampling theorem is used to determine the minimum rate (Nyquist Rate) at which an analog signal can be sampled without corruption when recovered to the original signal. The sampling frequency (f_S) must be at least two times the highest frequency in the analog signal (f_A).

$$f_S > 2 * f_A \tag{6.1.1}$$

Sampling frequency (f_S) of 8 kHz is used internationally in telephone systems, and the telephone signal is sampled 8000 times per second. The interval between two sequential samples (sampling interval = T_S) can be calculated as:

$$T_S = \frac{1}{f_S} = \frac{1}{8000\ Hz} = 125\mu s \tag{6.1.2}$$

In the Figure 6.2 it is shown how the telephone signal is sampled. First it is passed on a low-pass filter then fed to an electronic switch. Low-pass filter limits the frequency band; it blocks high frequency components which are higher than half of the f_S. The electronic switch takes samples from audio signal with sampling rate of 8 kHz. Then a PAM (Pulse Amplitude Modulation) signal is obtained at the output.

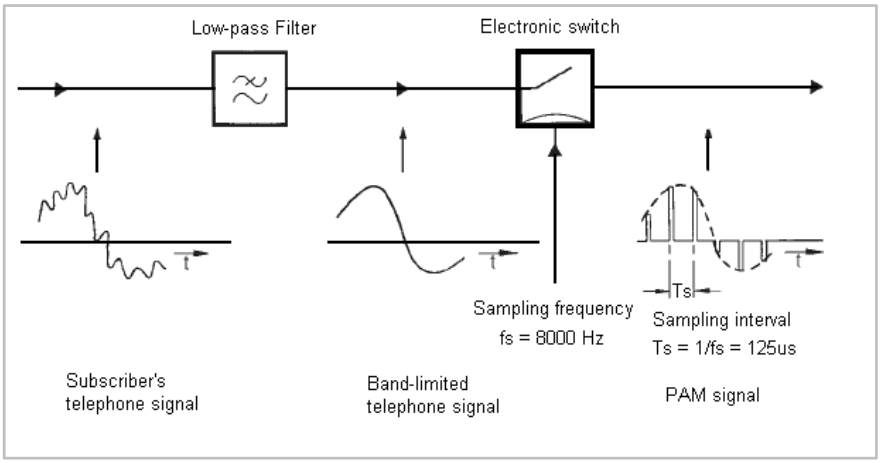

Figure 6.2 Generation of PAM signal

The PAM signal represents the audio in analog form. The samples can be transmitted more easily in digital form. For this the signal has to bi converted to digital PCM signal by quantizing.

6.1.1.2 Quantizing

It is shown in Figure 6.3 how quantization works. In order to make it simpler, only 4 equal quantizing intervals are indicated. The quantizing intervals are numbered 00 to 11 in the signal range. When recovering the signal, signal level corresponding to the middle of two quantizing intervals is recovered for one of these intervals. This causes small differences between original signal and recovered signal. This difference for each sample can be up to half of a quantizing interval. This distortion occurred in quantization can be decreased by increasing the number of quantizing intervals.

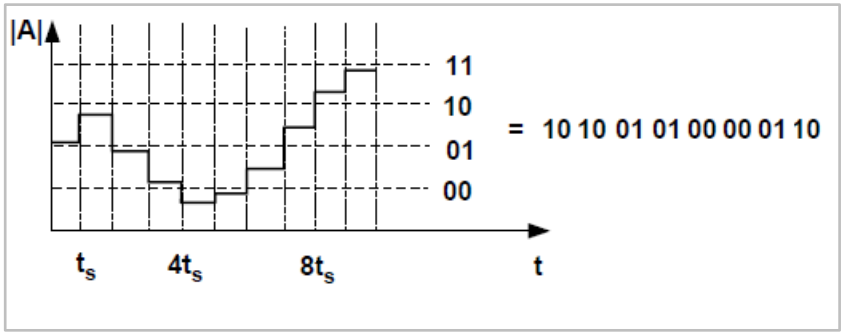

Figure 6.3 Example of quantizing an analog signal rolled-up

6.1.1.3 Encoding

The PCM is obtained by encoding the quantized signal into digital bits. The encoder writes an 8-bit PCM word for each sample determined in quantizing intervals. An 8-digit binary code is used for the $256 = 2^8$ intervals which are 128 for positive and 128 for negative quantizing intervals. Therefore PCM words have 8 bits. Signed representation is used for PCM. The first bit for the positive intervals is a "1", for the negative intervals is a "0".

6.1.1.4 Multiplexing

The 8-bit PCM words of a group can be transmitted in a wider frame synchronously. One word of is followed by next word, and all are arranged in sequential order. This technique is called PCM time-division multiplexing. Figure 6.4 shows a multiplexed 32 channels into E1 frame below.

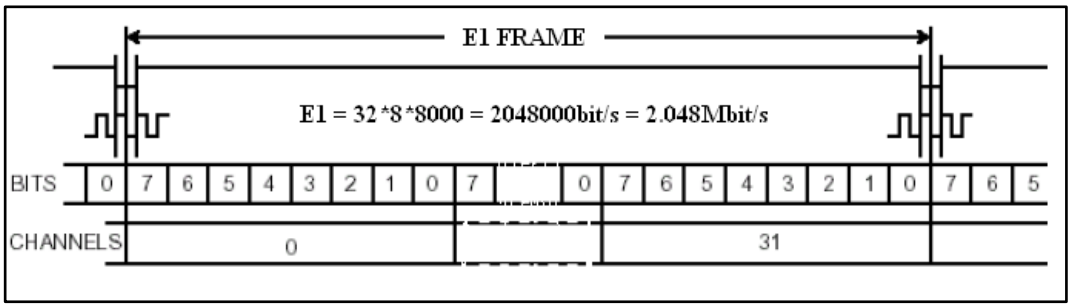

Figure 6.4 Multiplexed 32 channels into E1 frame

6.1.1.5 Demultiplexing

On the receive side each PCM signal is recovered from the time-division multiplexed signal. Then the process is continues through decoding.

6.1.1.6 Decoding

As in quantizing and encoding parts, on the receive side a signal amplitude is allocated to every 8-bit PCM word. It corresponds to the midpoint of the particular quantizing interval. The characteristic for decoding should be the same as encoding

on the transmit side. The PCM words are decoded in the receive order and converted to a PAM signal. Finally these PAM signals are passed on a low-pass filter, which reproduces the original analog voice signal. Figure 6.5 shows the reproduction procedure of the digital signal down below.

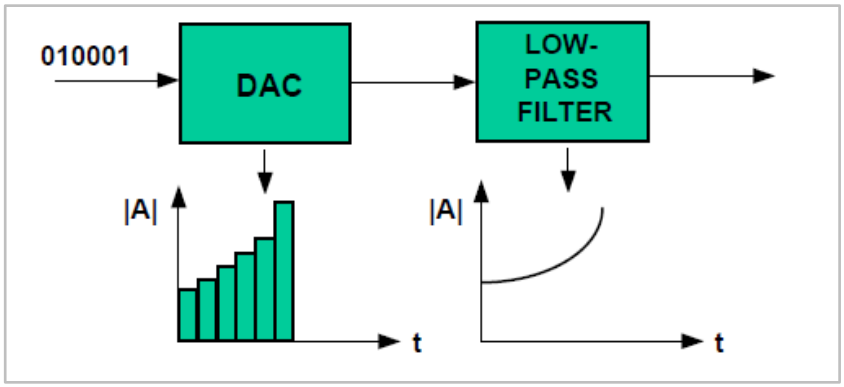

Figure 6.5 Reproduction of digital signal with low-pass filter

6.2 Telephony Signals vs. Wav Files

Telephony signals in PSTN include voice signals and these are filtered through analogue filters. In this project the main issue is to implement digital filters in a FPGA and apply spectrum shaping on voice signals especially digitally. For this aim, data subchunk of the *wav* files can be used as a voice source. Actually 8-bit wav file which is sampled with 8 kHz has almost the same data as the telephony signal which is just encoded to digital in the PCM procedure.

6.2.1. Wav File

The WAV (Waveform Audio File Format) is a subtype of Microsoft's RIFF (Resource Interchange File Format) specification and used or storing an audio signal on PCs. A RIFF file has a header in the start, and it is followed by data chunks. A *wav* file is generally just a RIFF file contains a single *wav* chunk. This *wav* chunk consists of two sub chunks; a "FMT" chunk specifies the data format and a "data" chunk contains the sampled audio. This type of wav file is called "Canonical form".

Basically a *wav* file has three major chunks which are *riff, fmt* and *data* chunks. First 12 bytes are used for riff chunk and contains “chunked” “chunk size” and “chunk format” descriptions. Second part is the fmt sub-chunk which is 24 bytes long and it contains the description of several properties such as “sample rate”, “audio format” etc. of the audio signal. The last chunk is data sub-chunk which starts with 37th byte and continues till end of the file. First 8 bytes of this chunk contains ID and size information, and the rest is pure sampled audio. So it can be said that first 44 bytes of this type of file contains information and the rest contains pure sampled audio.

Data chunks of this type of files can be used to test digital filters in the whole design. Figure 6.6 shows the structure of a wav file.

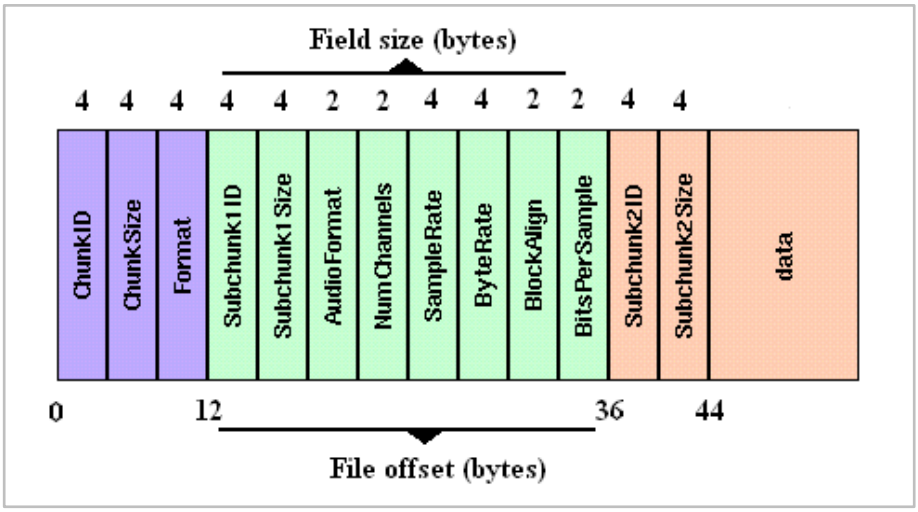

Figure 6.6 Data structure of a wav file

CHAPTER SEVEN
APPLICATION AND RESULTS

7.1 Pre-Implementation

As mentioned in Chapter.3, whole system is prepared for implementation using Xilinx ISE Design Suite software. Pin connections are constructed with Plan Ahead software. In Figure 7.1 RTL schematic of the system is shown below.

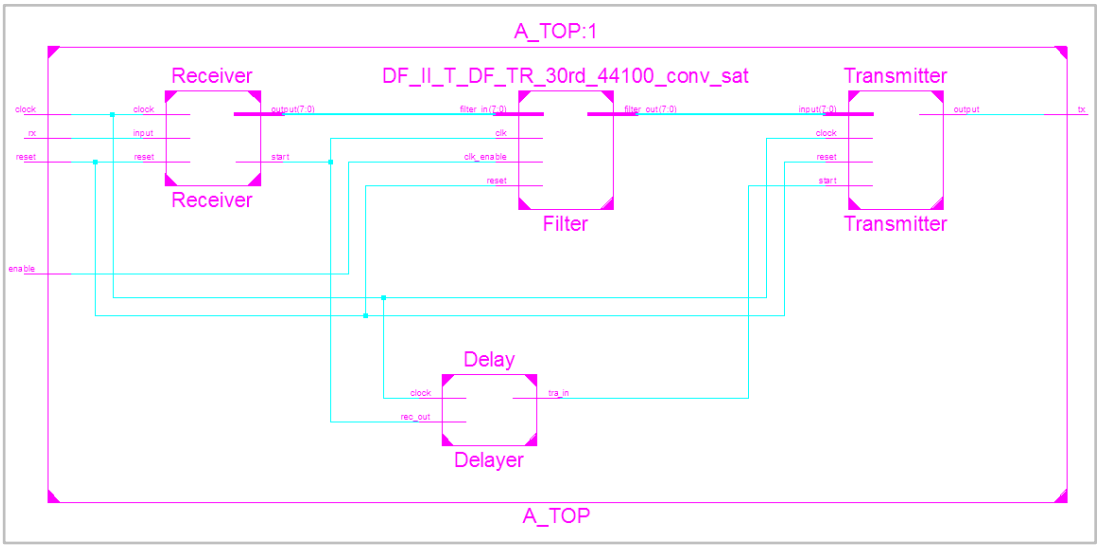

Figure 7.1 RTL schematic of the designed system

Receiver, Transmitter, Delayer and Filter modules are seen in the Figure 7.1. Each of these modules has its own VHDL source file. These files are instantiated in a top module named A_TOP. For more information about instantiation, refer to the Xilinx ISE / Help.

After passing “Synthesize” and “Implement” steps without any errors on ISE Design suite, a *.bit file is created. This file is implemented on the board by Xilinx iMPACT software.

7.2 Filter Design with FDATool

FDATool is used for creating several types of filters with different specifications. The software work as a sub software of MatLab and starts executing the "fdatool" command in the MatLab command window. FDATool is displayed as shown in Figure 7.2 below.

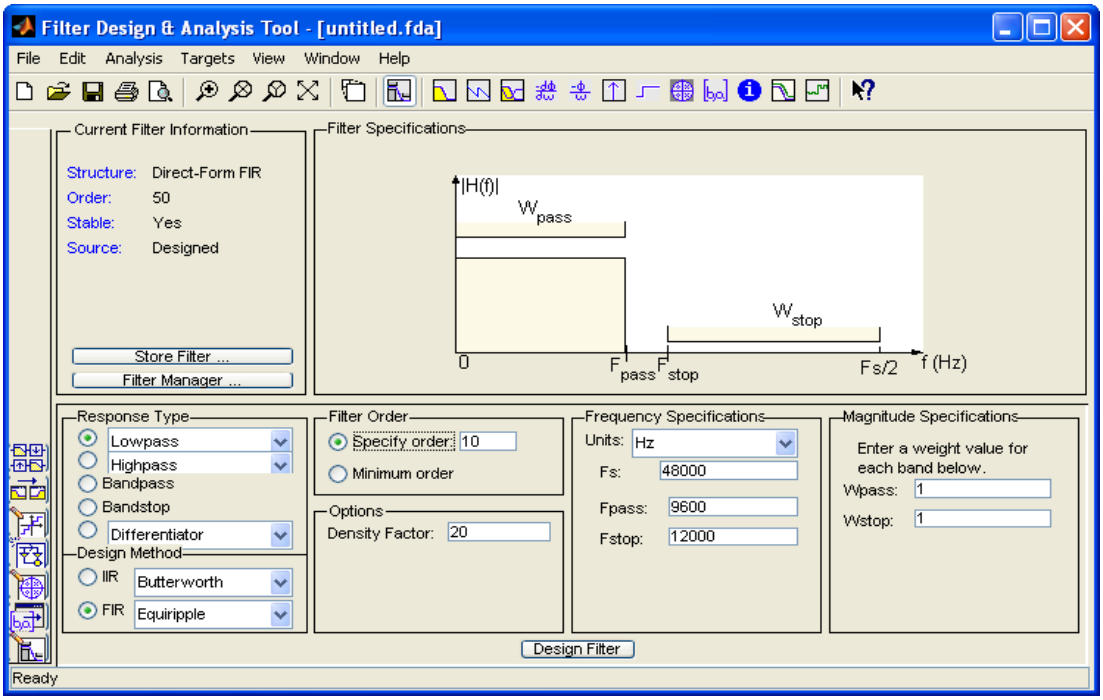

Figure 7.2 FDAToool main window with filter design specifications

As designing the filter required, filter specifications like Response Type, Design Method, Filter Order, Frequency Specifications were checked and entered correctly in the Design Filter tab. Next step is the Set Quantization Parameters tab. It is shown in Figure 7.3.

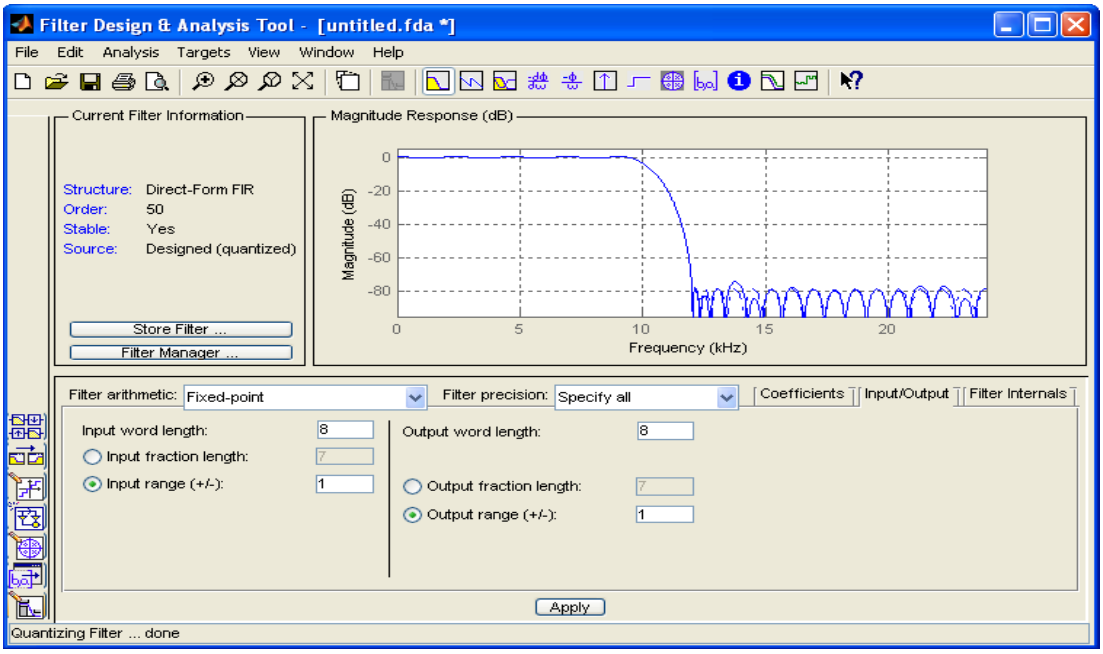

Figure 7.3 Set quantization parameters tab in FDATool

In this tab important specification is the Filter Arithmetic selection. Only Fixed-Point Arithmetic option is valid for HDL generation. In Filter Precision option, Specify all option should be selected for specifying the output and accumulator width lengths in the filter. Otherwise FDATool calculates an optimum width length of accumulator and output of the filter according to other specifications of the filter. And filter output always has wider width length than filter input when this option is not Specify all. All the specifications like Numerator word length, Input word length, Output word length, Rounding mode, Overflow mode, Accumulator word length are checked and entered according to the filter structure in this tab.

After all the specifications are checked and entered, the next step is generating the VHDL code of the designed filter. Selecting Targets > Generate HDL… from the upper menu starts the HDL Coder which is shown in Figure 7.4.

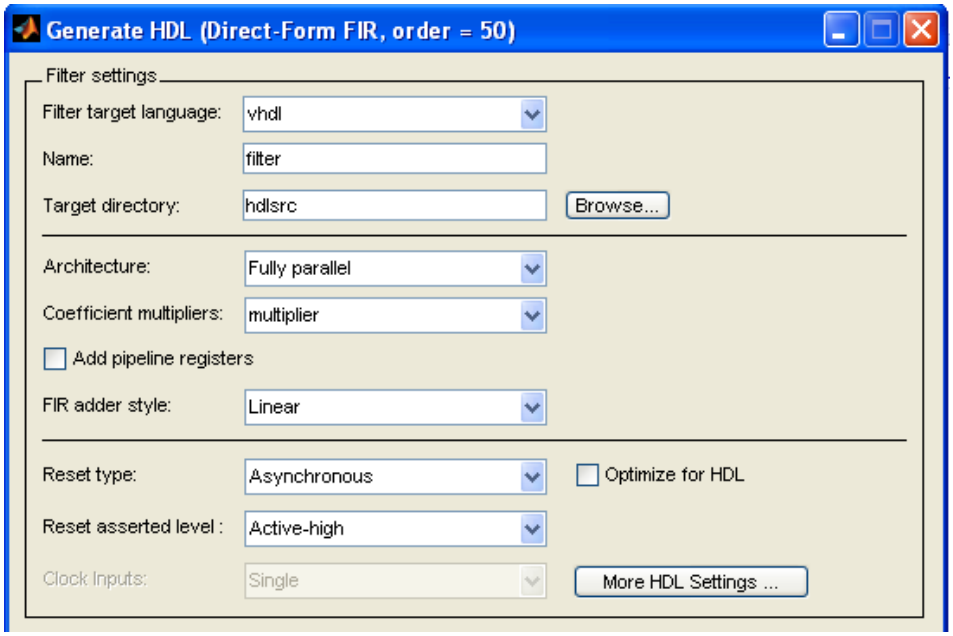

Figure 7.4 HDL generation window in FDATool

In the Generate HDL window specifications like Add pipeline registers, Coefficient multipliers, FIR adder style, Reset type and Reset asserted level are checked and selected according to the top design.

It is seen that an option named “More HDL Settings…” in the Figure 7.4 above. By clicking this button a window appears as shown in Figure 7.5.

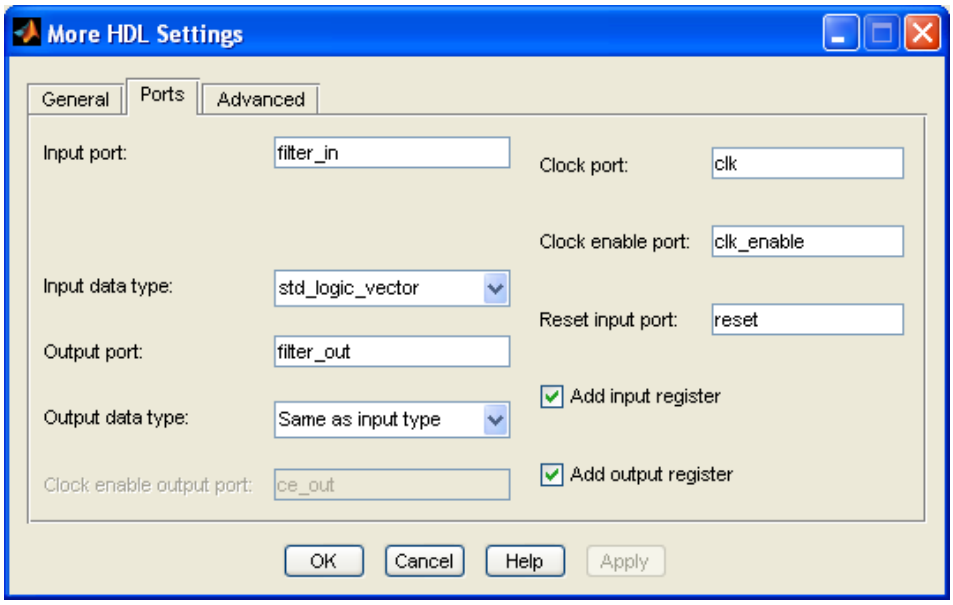

Figure 7.5 More HDL settings window in FDATool

This window has three tabs as General, Ports and Advanced. Ports tab includes preferences like input port, clock port, input data type, output port etc. shown in figure above. These preferences are checked and filled according to the top design.

VHDL code file of the designed filter is created in the target directory just after "Generate" button clicked which is at the right bottom of the Generate HDL window.

7.3 Implementation on FPGA

Constructed VHDL files are instantiated in the top module as mentioned in section 6.1. Then this top module is passed through Synthesize, I/O Pin Planning (PlanAhead) Post Synthesize, Implement Design and Generate Programming File operations. Then the created *.bit file in the project directory of ISE is implemented on the board by the software Xilinx iMPACT.

Following results are given from the command window of iMPACT after implementing the *.bit file on the board and Program Succeeded icon is shown on screen.

```
----------------------------------------------------------------------------------
INFO:iMPACT:2219 - Status register values:
INFO:iMPACT - 0011 1111 1010 1100
INFO:iMPACT:579 - '1': Completed downloading bit file to device.
INFO:iMPACT - '1': Programing completed successfully.
PROGRESS_END - End Operation.
----------------------------------------------------------------------------------
```

For more information about using iMPACT, refer to the Xilinx ISE / Help.

7.4 Transmit and Receive Data

By implementing the design on FPGA, it is ready to test the design and check the results. A connection is needed for transmitting and receiving data between PC and FPGA board in order to test the system if it is working or not. This requirement is

supplied by a serial communication between serial port of PC and RS-232 of FPGA board.

Software like HyperTerminal or TeraTerm can be used to test if any data returns from the FPGA. TeraTerm is used for testing design just other modules together without filter module.

Serial port has maximum 8-bit data width length for each baud as known. System is designed for 8-bit data input and 8-bit data output for each sample to make hardware simpler. 8-bit *.wav files are used for testing the system with several sampling rates. Wav file structure is mentioned in chapter six.

Simple executable software named “write_read_comport” is prepared for this communication. It is coded in C#. Algorithm is basically to remove headers from wav file and send each sample to the serial port starting from LSB; then, to listen the serial port if there is any data on the port. The data processed on FPGA is received from serial port is stored in memory and next sample of wav file is sent to the serial port. After all the samples sent through serial port, new wav file is created in the same directory with original wav file by adding the removed headers to the stored data with the last processed byte received from the serial port. Software GUI looks like Figure 7.6.

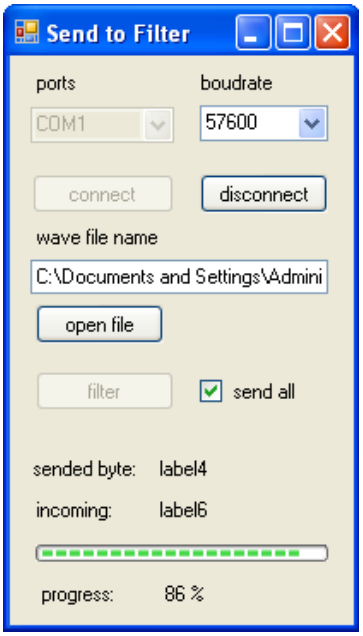

Figure 7.6 Data flow progress

In the software window there are options and selections like ports, baud rate, connect, disconnect, open file, filter, send all. At the bottom of the window it shows the percentage of data processed and received. Source code of the software is given in appendices.

First the serial port which FPGA is connected is selected from ports option. Then the baud rate is selected from baud rate option and clicked to connect button. Now connection between PC and FPGA is established and system is ready to send data. Then wav file is selected by clicking open file option and found from the located directory. Data sending and receiving process starts after checking send all option and clicking on the filter button.

7.5 Expected and Derived Results

The system is constructed and implemented on FPGA and now it is ready to test several types of filters with different orders created by FDATool are ready to test and compare results.

7.5.1 Results of Direct Form Architecture FIRs

First implementation and test has experienced with FIR filter which has following specifications.

Table 7.1 Sixth order Direct Form FIR filter specifications

Direct Form FIR 6th Order Filter FDATool Specifications			
Option	**Value**	**Option**	**Value**
Response Type	Low Pass	Input Word Length	8
Design Method	FIR Equir.	Input Fraction Length	7
Filter Order	6	Filter Precision	Specify All
Fs	44100Hz	Output Word Length	8
Fpass	3500Hz	Output Fraction Length	7
Fstop	4500Hz	Rounding Mode	Nearest (Co.)
Numerator Word Length	16	Overflow Mode	Saturate
Best-Prec. Frac. Lengths	Selected	Product W. Length	16
Use Unsig. Represent.	Cleared	Product Frac. Length	16
Scale Numerator Coeff.	Cleared	Accum. Word Length	18
Filter Arithmetic	Fixed-Point	Accum Frac. Length	16
Numerator Word Length	8		

Expected magnitude and phase responses of the filter given in the below table are shown in Figure 7.6 and Figure 7.7.

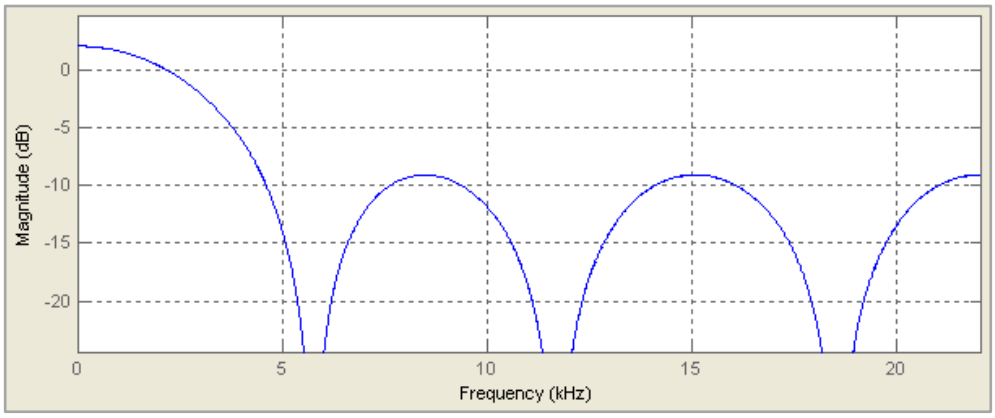

Figure 7.6 Magnitude response of FIR filter according to specifications in Table 7.1

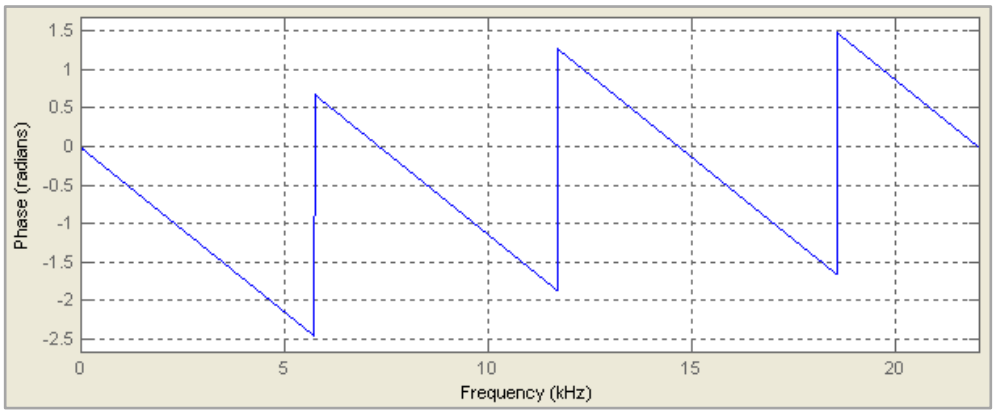

Figure 7.7 Magnitude response of FIR filter according to specifications in Table 7.1

Samples of original wav file which are between the range +127 and -128 are viewed in time domain 0 to 2.2s is analyzed with software Power Sound Editor Free and shown in the Figure 7.8.

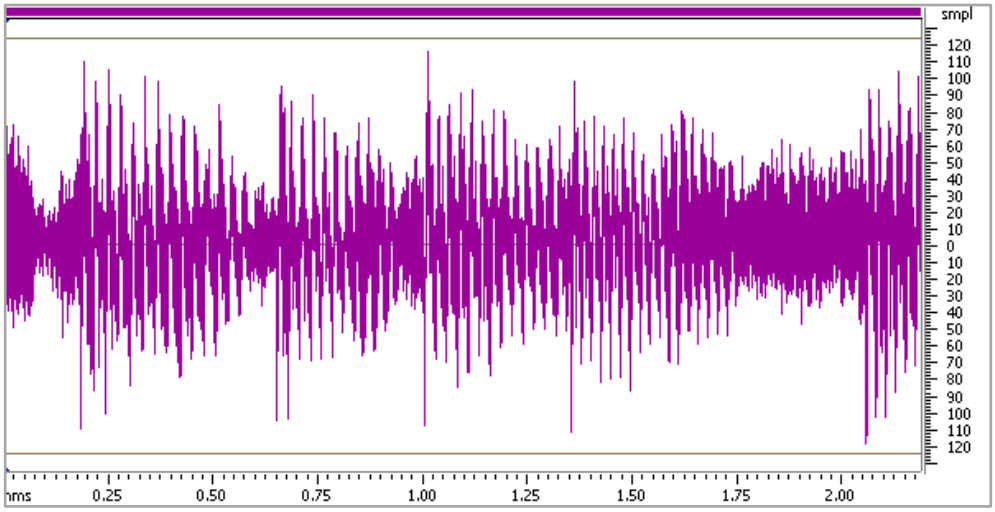

Figure 7.8 Samples of wav file in time domain

The wav file *I_44100.wav* has 44100Hz sampling rate, 8-bit depth, 96633 samples and almost 2.25s length. This file is stored in the relevant directory in the attachment CD. On the right side of the figure, sample range is shown between +127 and -128.

Frequency analysis of the I_44100.wav file with specifications above is shown in Figure 7.9.

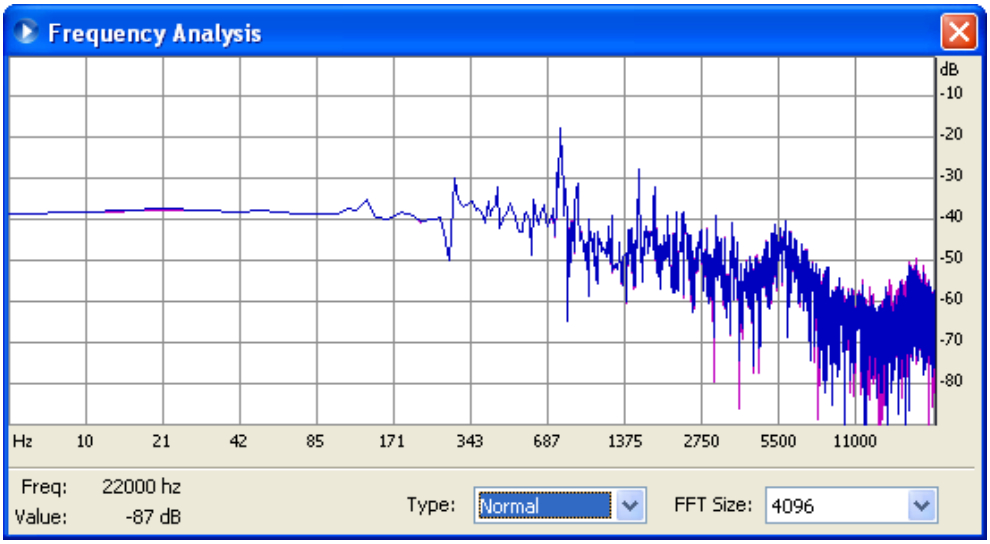

Figure 7.9 Frequency analysis of the I_44100.wav

After sending I_44100.wav through designed system, following sample view Figure 7.10 is taken from the processed audio file DF06_out.wav. This file is stored in the relevant directory of the attachment CD.

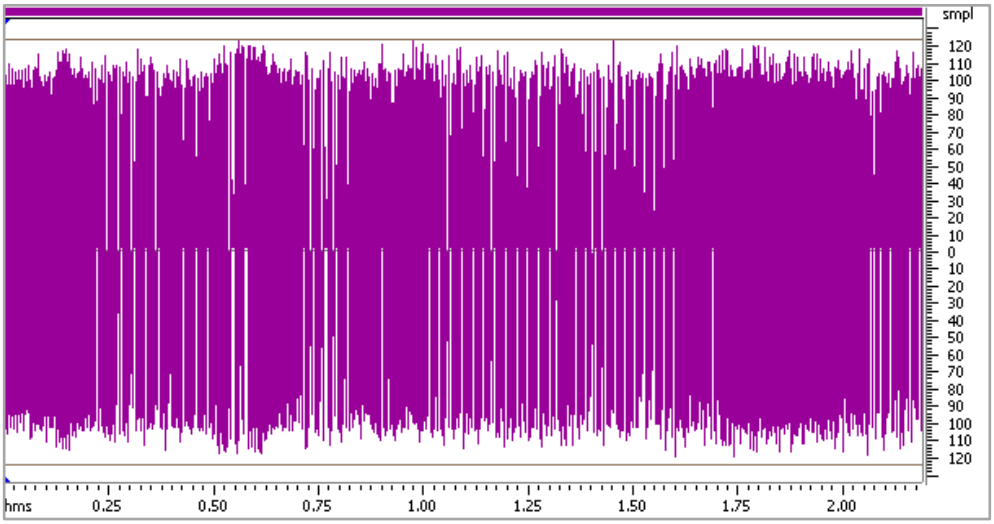

Figure 7.10 Samples of DF06_out.wav from 0 to 2.20s

To make the above figure simpler and see it detailed, it is zoomed into the first 7ms and shown in Figure 7.11 below.

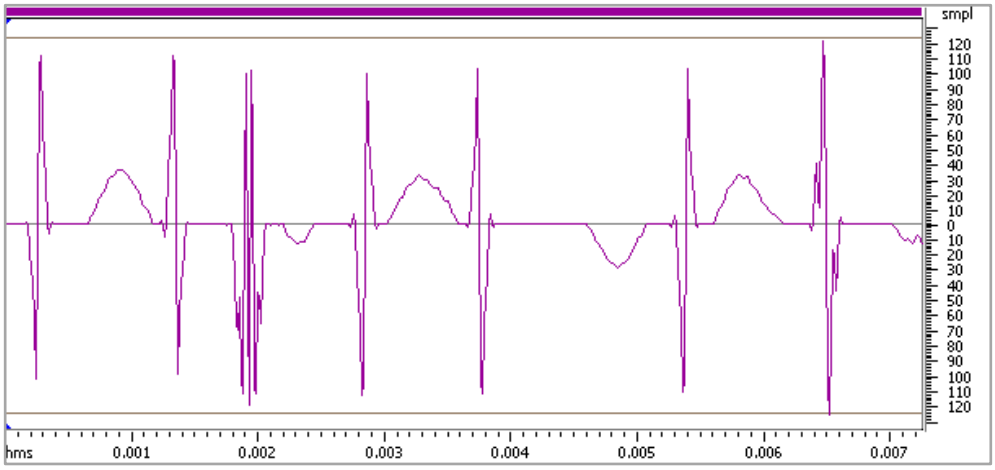

Figure 7.11 Samples of DF06_out.wav from 0 to 7ms

Frequency analysis of the DF06_out.wav file with shown specifications is shown in Figure 7.12.

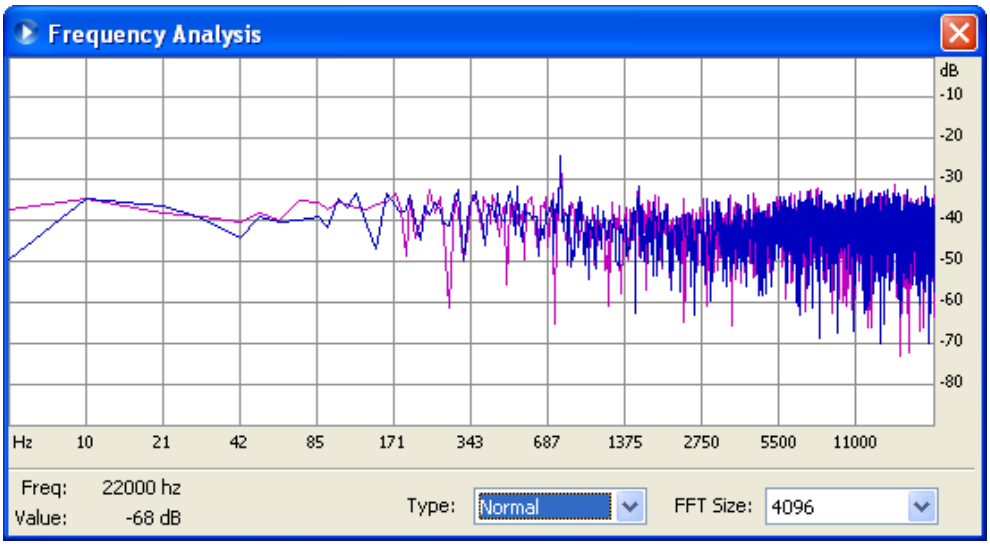

Figure 7.12 Frequency analysis of the DF06_out.wav

Next implementation and test has experienced with FIR filter which has following specifications.

Table 7.2 Twentieth order Direct Form FIR filter specifications

Direct Form FIR 20th Order Filter FDATool Specifications			
Option	**Value**	**Option**	**Value**
Response Type	Low Pass	Input Word Length	8
Design Method	FIR Equir.	Input Fraction Length	7
Filter Order	20	Filter Precision	Specify All
Fs	44100Hz	Output Word Length	8
Fpass	3500Hz	Output Fraction Length	7
Fstop	4500Hz	Rounding Mode	Nearest (Co.)
Numerator Word Length	16	Overflow Mode	Saturate
Best-Prec. Frac. Lengths	Selected	Product W. Length	16
Use Unsig. Represent.	Cleared	Product Frac. Length	16
Scale Numerator Coeff.	Cleared	Accum. Word Length	18
Filter Arithmetic	Fixed-Point	Accum Frac. Length	16
Numerator Word Length	8		

Expected magnitude and phase responses of the filter given in the below Table 7.2 are shown in Figure 7.13 and Figure 7.14 below.

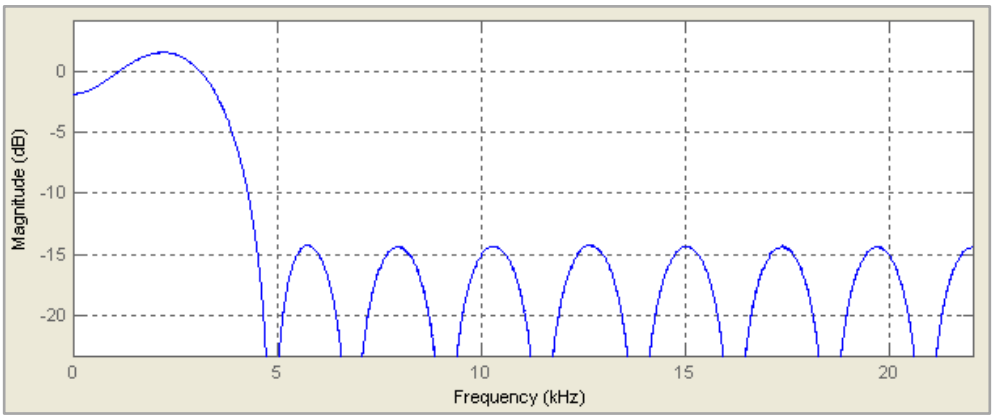

Figure 7.13 Magnitude response of FIR filter according to specifications in Table 7.2

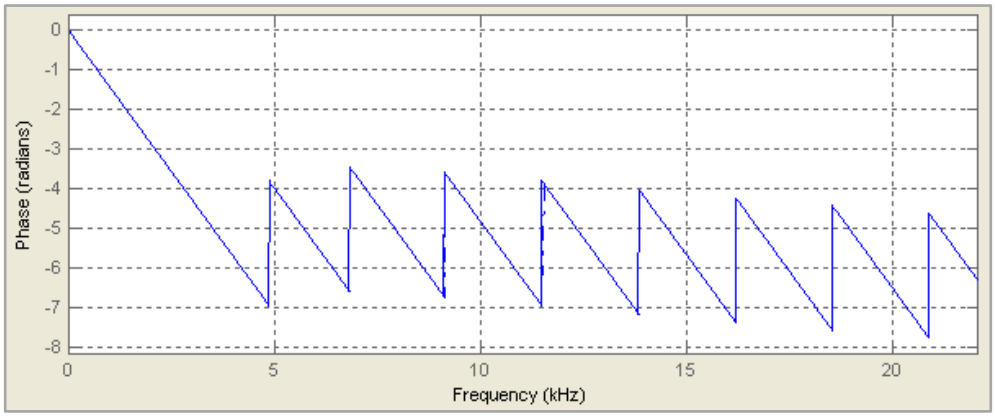

Figure 7.14 Phase response of FIR filter according to specifications in Table 7.2

Samples of original wav file I_44100.wav which are between the range +127 and -128 are viewed in time domain is analyzed with software Power Sound Editor Free and shown in the Figure 7.8.

Frequency analysis of the I_44100.wav file with shown specifications on figure itself is shown in Figure 7.9.

After sending I_44100.wav through designed system, following sample view Figure 7.15 is taken from the processed audio file DF20_out.wav. This file is stored in the relevant directory of the attachment CD.

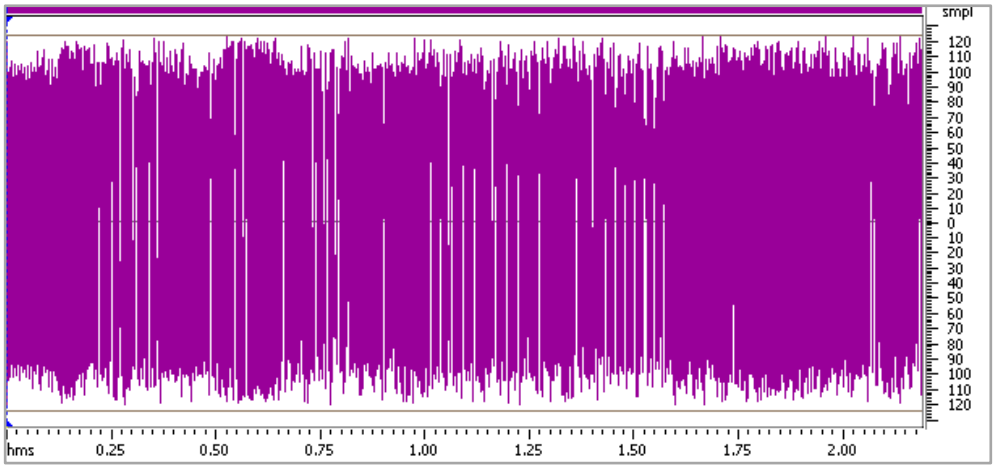

Figure 7.15 Samples of DF20_out.wav from 0 to 2.20s

To make the above figure simpler and see it detailed, it is zoomed into the first 7ms and shown in Figure 7.16 below.

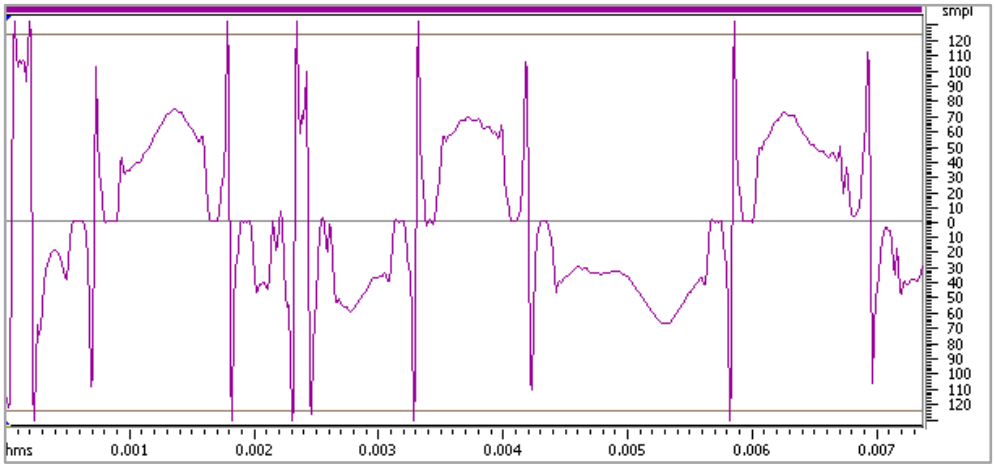

Figure 7.16 Samples of DF20_out.wav from 0 to 7ms

Frequency analysis of the DF20_out.wav file with shown specifications is shown in Figure 7.17.

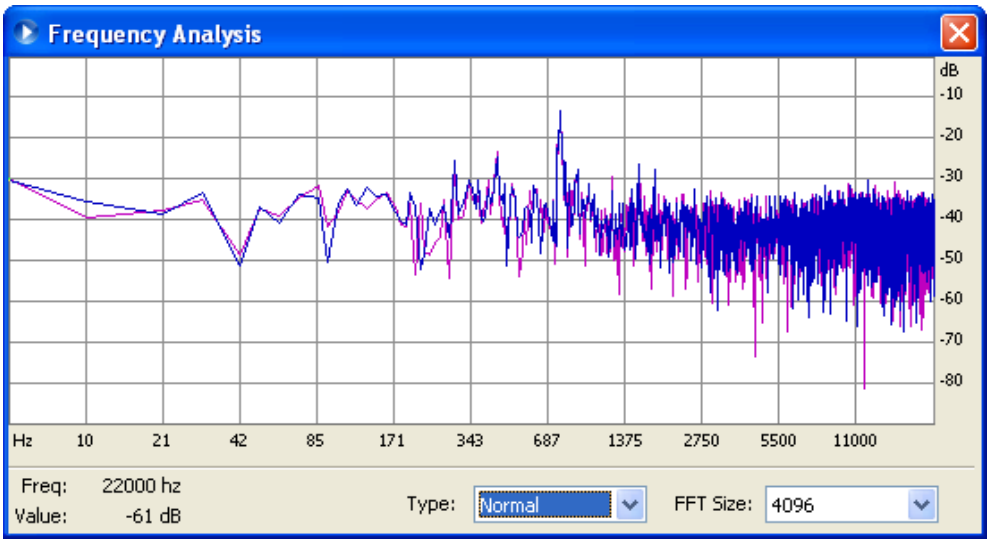

Figure 7.17 Frequency analysis of the DF20_out.wav

Next implementation and test has experienced with FIR filter which has following specifications.

Table 7.3 Thirtieth order Direct Form FIR filter specifications

Direct Form FIR 30th Order Filter FDATool Specifications			
Option	**Value**	**Option**	**Value**
Response Type	Low Pass	Input Word Length	8
Design Method	FIR Equir.	Input Fraction Length	7
Filter Order	30	Filter Precision	Specify All
Fs	44100Hz	Output Word Length	8
Fpass	3500Hz	Output Fraction Length	7
Fstop	4500Hz	Rounding Mode	Nearest (Co.)
Numerator Word Length	16	Overflow Mode	Saturate
Best-Prec. Frac. Lengths	Selected	Product W. Length	16
Use Unsig. Represent.	Cleared	Product Frac. Length	16
Scale Numerator Coeff.	Cleared	Accum. Word Length	18
Filter Arithmetic	Fixed-Point	Accum Frac. Length	16
Numerator Word Length	8		

Expected magnitude and phase responses of the filter with given specifications in Table 7.3 are shown in Figure 7.18 and Figure 7.19.

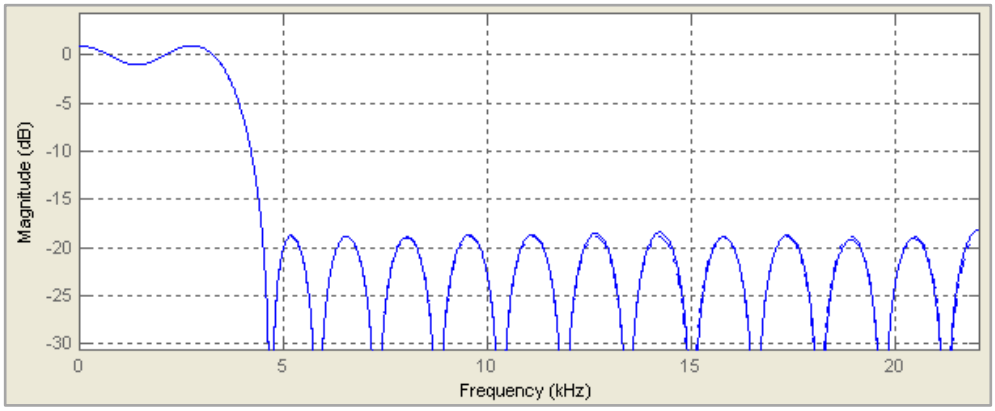

Figure 7.18 Magnitude response of FIR filter according to specifications in Table 7.3

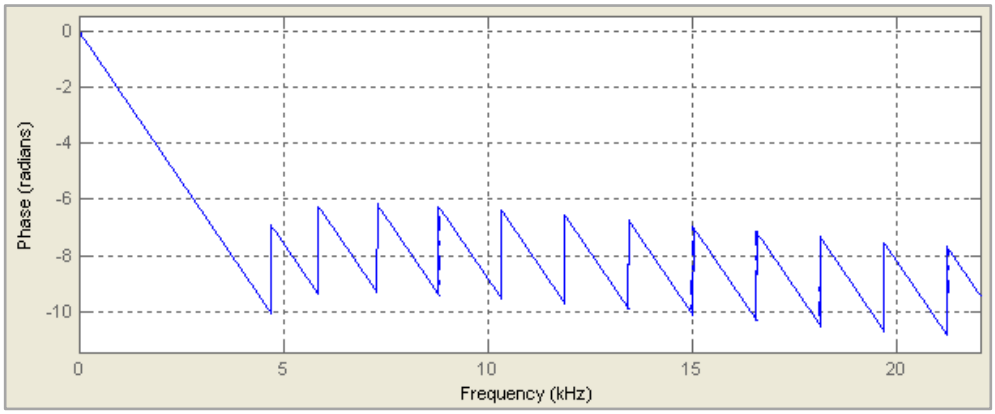

Figure 7.19 Phase response of FIR filter according to specifications in Table 7.3

Received I_44100.wav is processed through designed system, and following sample view Figure 7.20 is taken from the processed audio file DF30_out.wav. This file is stored in the relevant directory of the attachment CD.

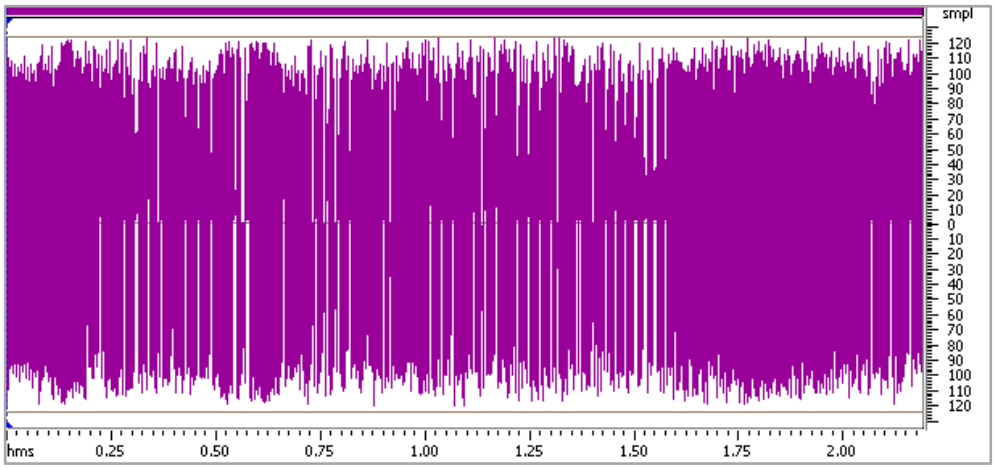

Figure 7.20 Samples of DF30_out.wav from 0 to 2.20s

To make the above figure simpler and see it detailed, it is zoomed into the first 7ms and shown in Figure 7.21.

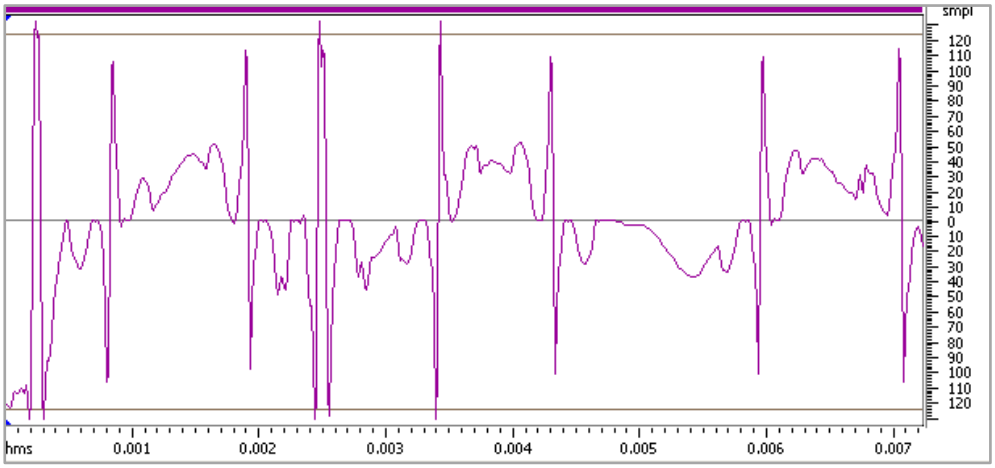

Figure 7.21 Samples of DF30_out.wav from 0 to 7ms

Frequency analysis of the DF30_out.wav file with shown specifications is shown in Figure 7.22.

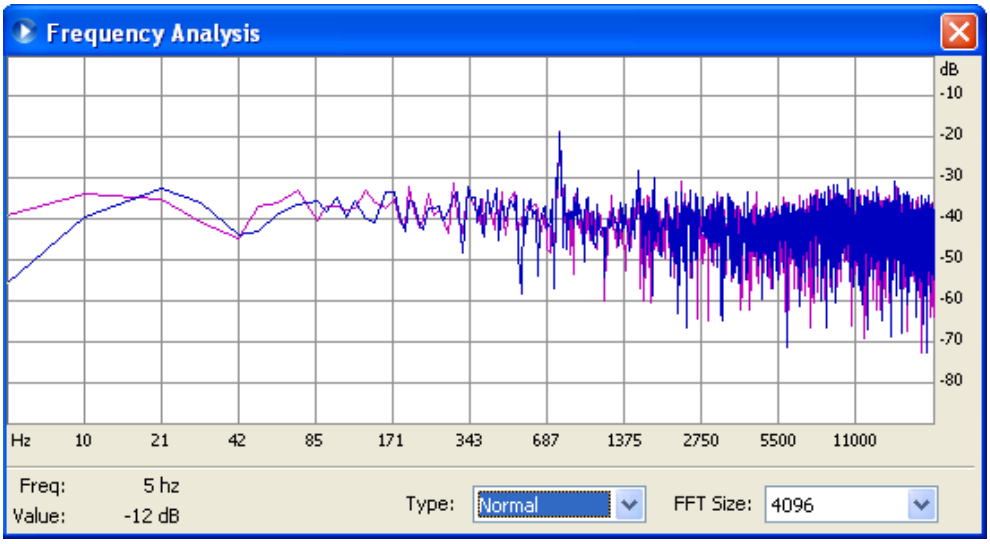

Figure 7.22 Frequency analysis of the DF30_out.wav

7.5.2 Results of Direct Form Transposed Architecture FIRs

To change the architecture of the designed FIR filter, Edit > Convert Stucture… option is selected. This selection opens a new window which has three options shown in Figure 7.23 below. Structure is changed by these steps. It is important that Set Quantization Parameters tab has to be reconfigured according to the design after changing the structure.

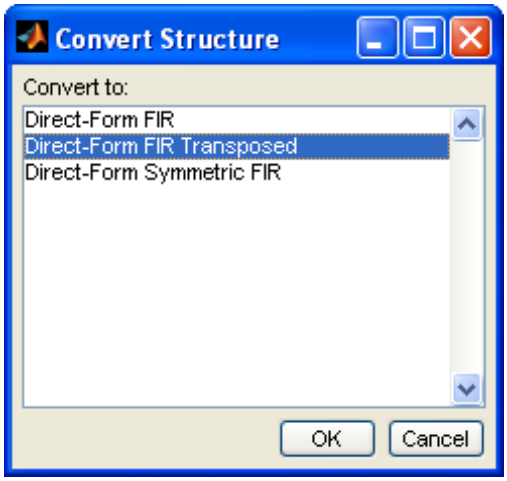

Figure 7.23 Convert structure window

First implementation and test has experienced with FIR filter which has following specifications.

Table 7.4 Tenth order Direct Form Transposed FIR filter specifications

Direct Form FIR Transposed 10th Order Filter FDATool Specifications			
Option	**Value**	**Option**	**Value**
Response Type	Low Pass	Input Word Length	8
Design Method	FIR Equir.	Input Fraction Length	7
Filter Order	10	Filter Precision	Specify All
Fs	44100Hz	Output Word Length	8
Fpass	3500Hz	Output Fraction Length	7
Fstop	4500Hz	Rounding Mode	Nearest (Co.)
Numerator Word Length	16	Overflow Mode	Saturate
Best-Prec. Frac. Lengths	Selected	Product W. Length	16
Use Unsig. Represent.	Cleared	Product Frac. Length	16
Scale Numerator Coeff.	Cleared	Accum. Word Length	18
Filter Arithmetic	Fixed-Point	Accum Frac. Length	16
Numerator Word Length	8		

Expected magnitude response of the filter given specifications in Table 7.4 is shown in Figure 7.24 below.

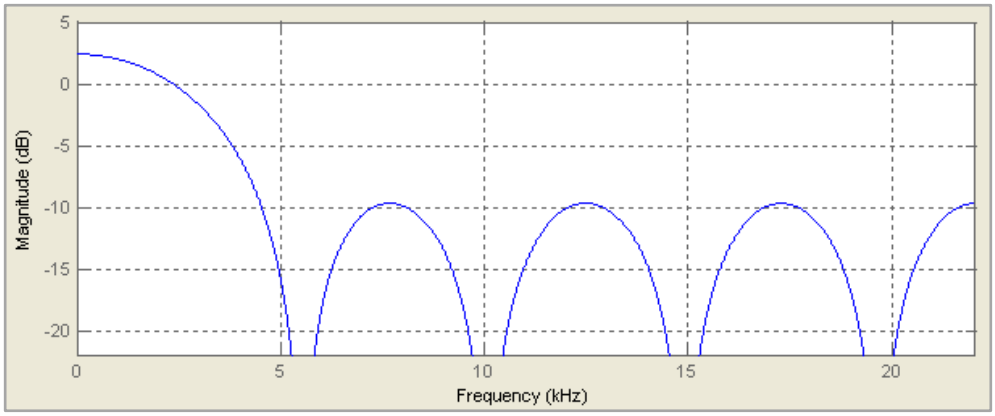

Figure 7.24 Magnitude response of FIR filter according to specifications in Table 7.4

Expected phase response of the filter given specifications in Table 7.4 is shown in Figure 7.25.

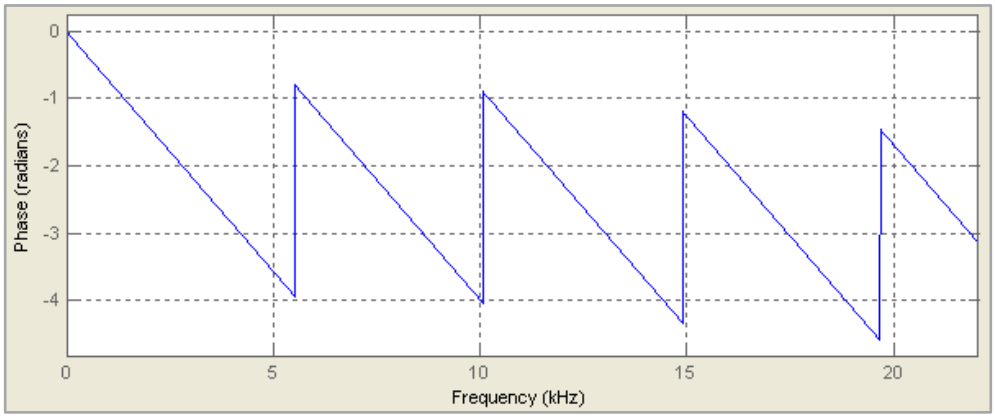

Figure 7.25 Phase response of FIR filter according to specifications in Table 7.4

Original file I_44100.wav file is filtered through designed system, and following sample view Figure 7.26 is taken from the processed audio file TR10_out.wav. This file is stored in the relevant directory of the attachment CD.

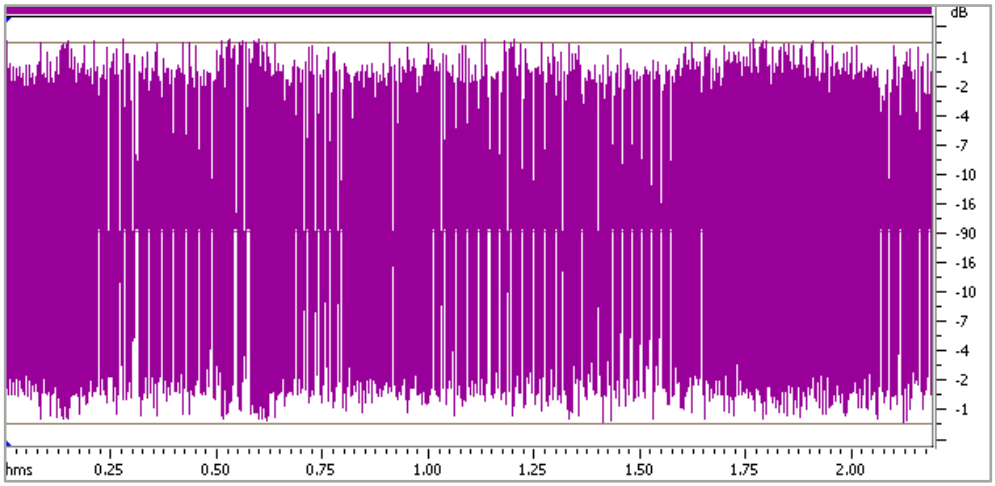

Figure 7.26 Samples of TR10_out.wav from 0 to 2.20s

To make the above figure simpler and see it detailed, it is zoomed into the first 7ms and shown in Figure 7.27.

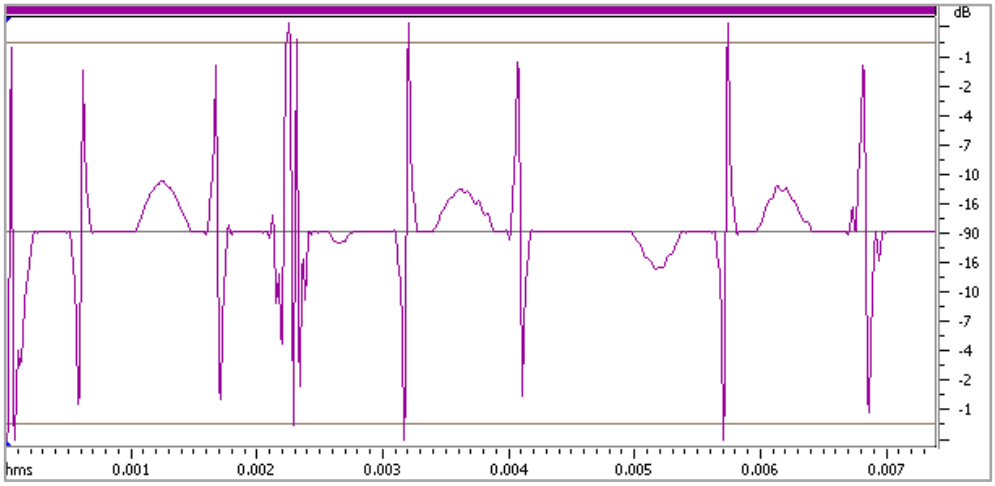

Figure 7.27 Samples of TR10_out.wav from 0 to 7ms

Frequency analysis of the TR10_out.wav file with shown options is shown in Figure 7.28.

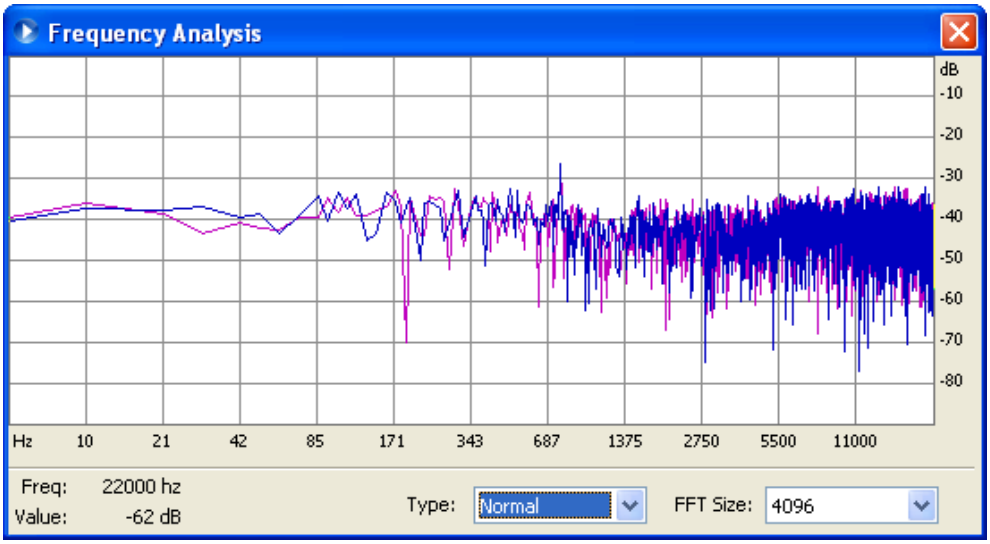

Figure 7.28 Frequency analysis of the TR10_out.wav

Next implementation and test has experienced with FIR filter which has following specifications.

Table 7.5 Twentieth order Direct Form Transposed FIR filter specifications

Direct Form FIR Transposed 20th Order Filter FDATool Specifications			
Option	**Value**	**Option**	**Value**
Response Type	Low Pass	Input Word Length	8
Design Method	FIR Equir.	Input Fraction Length	7
Filter Order	20	Filter Precision	Specify All
Fs	44100Hz	Output Word Length	8
Fpass	3500Hz	Output Fraction Length	7
Fstop	4500Hz	Rounding Mode	Nearest (Co.)
Numerator Word Length	16	Overflow Mode	Saturate
Best-Prec. Frac. Lengths	Selected	Product W. Length	16
Use Unsig. Represent.	Cleared	Product Frac. Length	16
Scale Numerator Coeff.	Cleared	Accum. Word Length	18
Filter Arithmetic	Fixed-Point	Accum Frac. Length	16
Numerator Word Length	8		

Expected frequency and phase responses of the filter with given specifications in Table 7.5 are shown in Figure 7.29 and Figure 7.30.

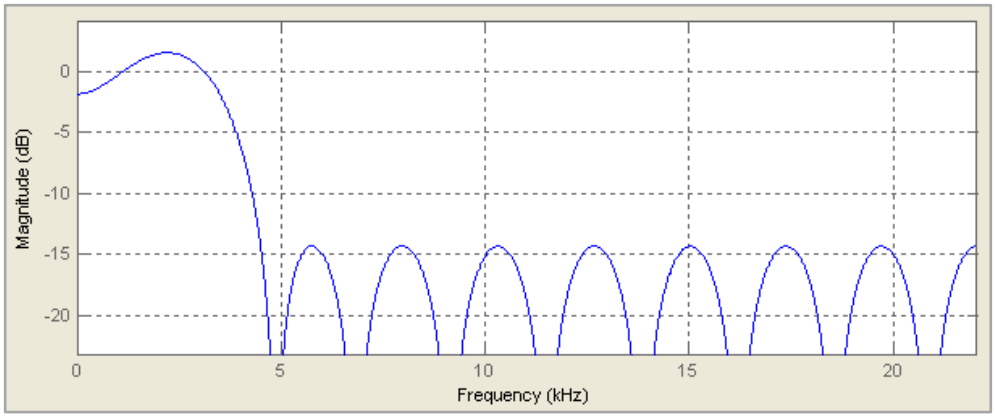

Figure 7.29 Magnitude response of FIR filter according to specifications in Table 7.5

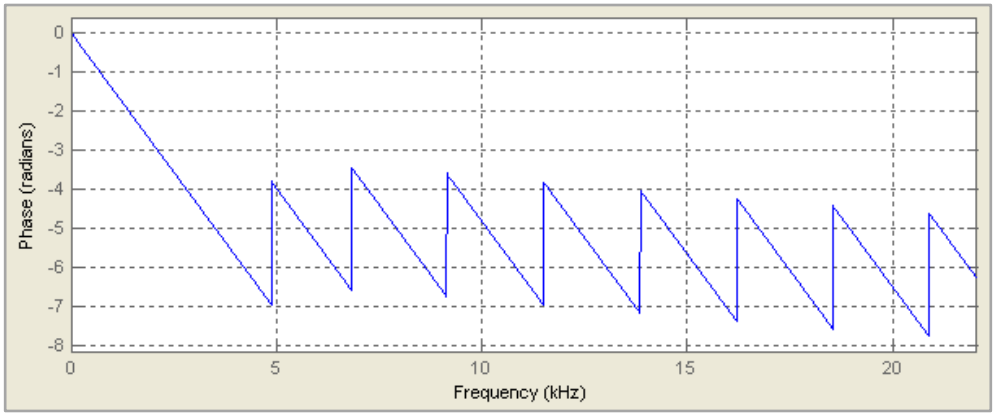

Figure 7.30 Phase response of FIR filter according to specifications in Table 7.5

By filtering I_44100.wav in FPGA with designed system, following sample view Figure 7.31 is taken from the processed audio file TR20_out.wav. This file is stored in the relevant directory of the attachment CD.

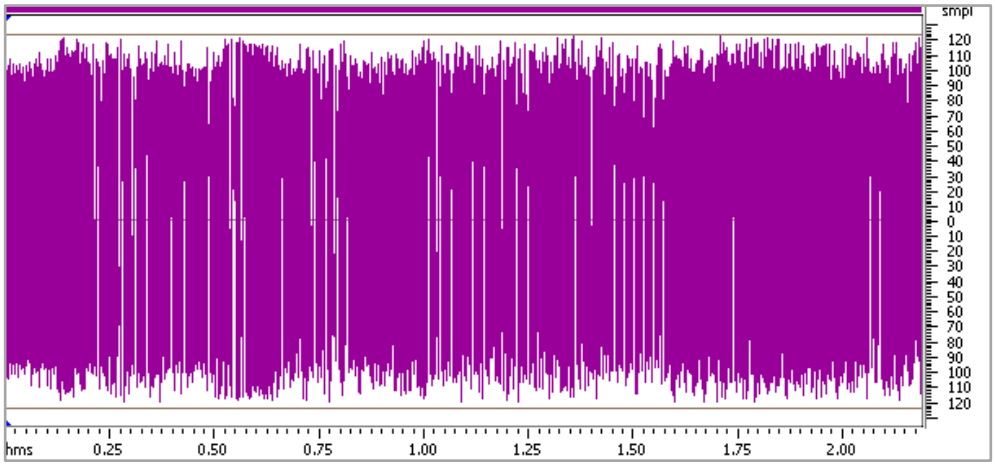

Figure 7.31 Samples of TR20_out.wav from 0 to 2.20s

To make the above figure simpler and see it detailed, it is zoomed into the first 7ms and showed in Figure 7.32.

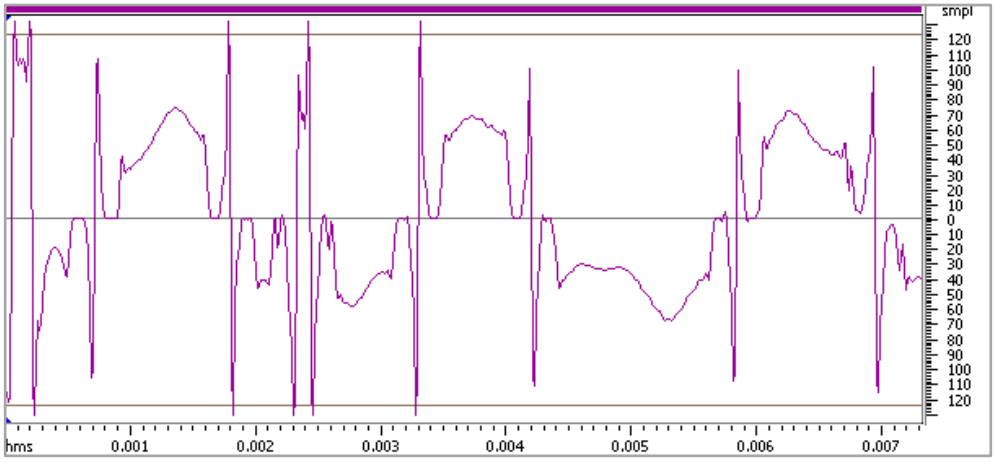

Figure 7.32 Samples of TR20_out.wav from 0 to 7ms

Frequency analysis of the TR20_out.wav file with selected options is shown in Figure 7.33.

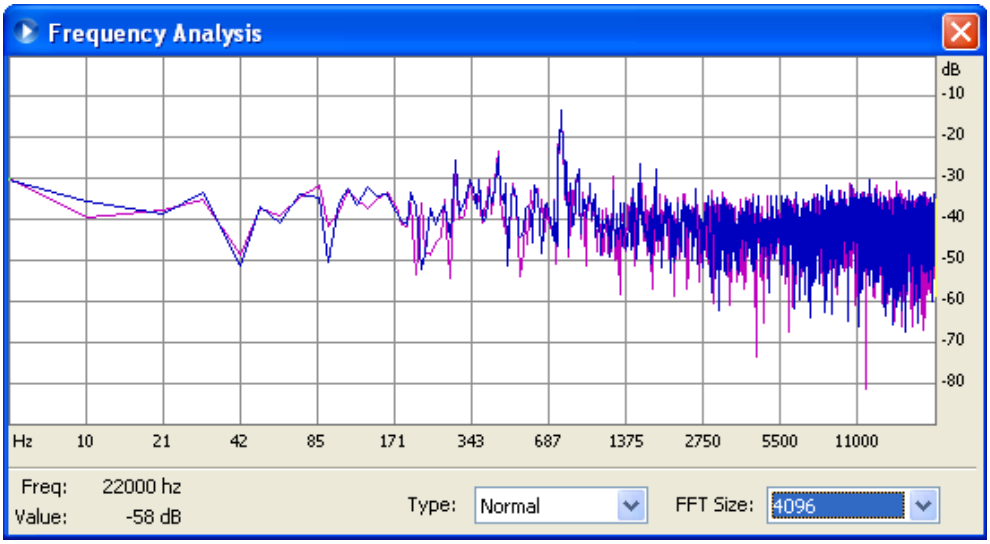

Figure 7.33 Frequency analysis of the TR20_out.wav

Next implementation and test has experienced with FIR filter which has following specifications.

Table 7.6 Thirtieth order Direct Form Transposed FIR filter specifications

Direct Form FIR Transposed 30th Order Filter FDATool Specifications			
Option	**Value**	**Option**	**Value**
Response Type	Low Pass	Input Word Length	8
Design Method	FIR Equir.	Input Fraction Length	7
Filter Order	30	Filter Precision	Specify All
Fs	44100Hz	Output Word Length	8
Fpass	3500Hz	Output Fraction Length	7
Fstop	4500Hz	Rounding Mode	Nearest (Co)
Numerator Word Length	16	Overflow Mode	Saturate
Best-Prec. Frac. Lengths	Selected	Product W. Length	16
Use Unsig. Represent.	Cleared	Product Frac. Length	16
Scale Numerator Coeff.	Cleared	Accum. Word Length	18
Filter Arithmetic	Fixed-Point	Accum Frac. Length	16
Numerator Word Length	8		

Expected magnitude and phase responses of the filter with given specifications in Table 7.6 are shown in Figure 7.34 and Figure 7.35

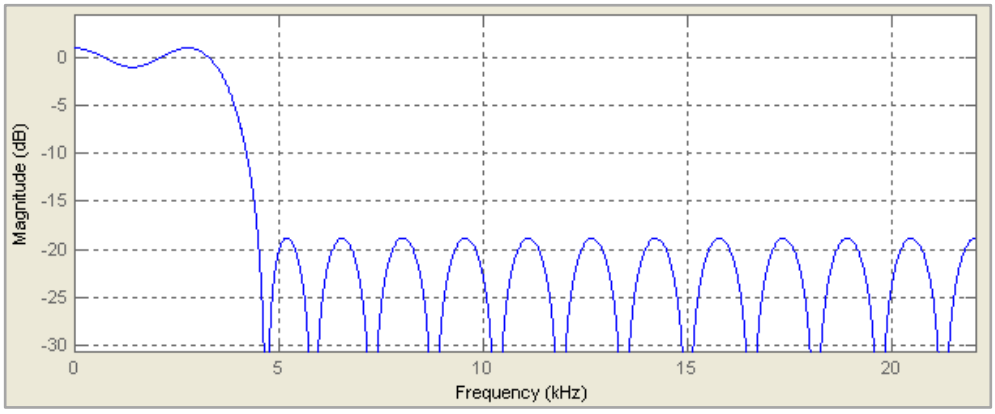

Figure 7.34 Magnitude response of FIR filter according to specifications in Table 7.6

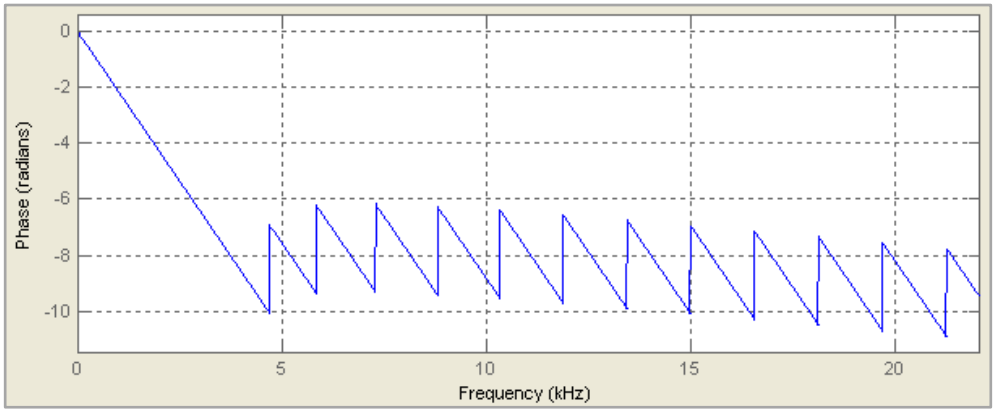

Figure 7.35 Phase response of FIR filter according to specifications in Table 7.6

After sending I_44100.wav through designed system in FPGA, following sample view Figure 7.36 is taken from the processed audio file TR30_out.wav. This file is stored in the relevant directory of the attachment CD.

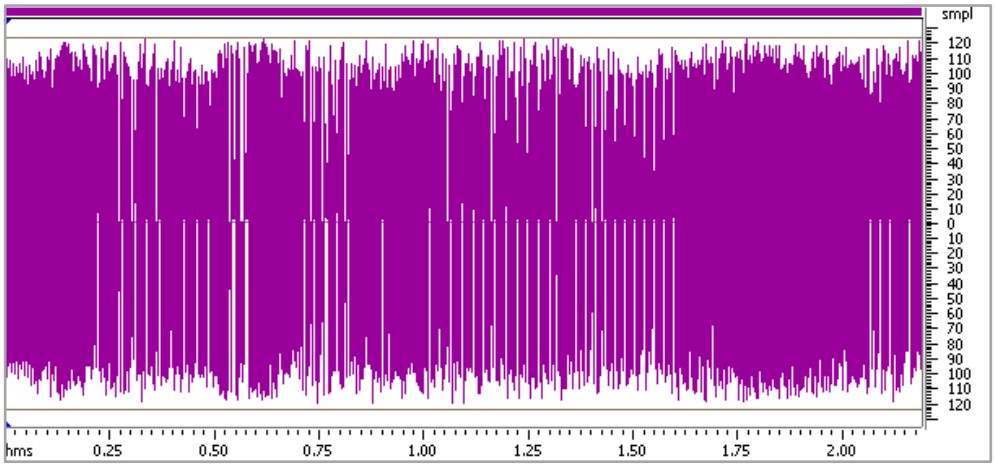

Figure 7.36 Samples of TR30_out.wav from 0 to 2.20s

To make the above figure simpler and see it detailed, it is zoomed into the first 7ms and shown in Figure 7.37.

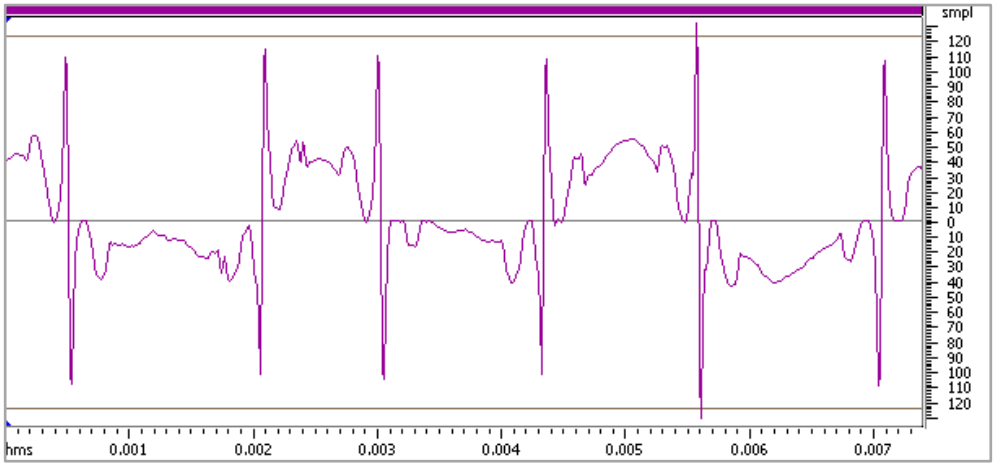

Figure 7.37 Samples of TR30_out.wav from 0 to 7ms

Frequency analysis of the TR30_out.wav file with selected options is shown in Figure 7.38.

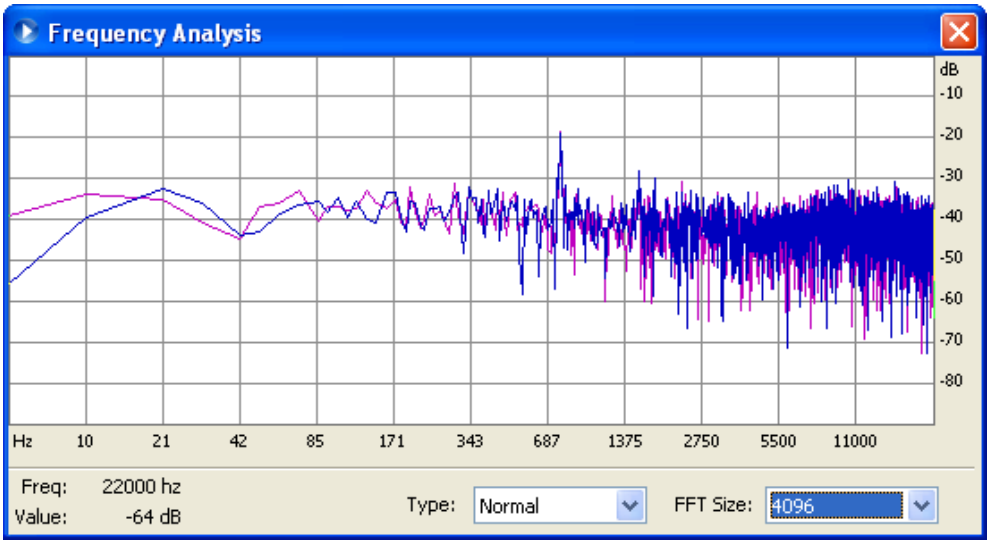

Figure 7.38 Frequency analysis of the TR30_out.wav

7.5.3 Results of Direct Form Symmetric Architecture FIRs

To change the architecture of the designed FIR filter, Edit > Convert Stucture… option is selected. This selection opens a new window which has three options shown in Figure 7.23. Structure is changed by choosing the option in the selection window. It is important that Set Quantization Parameters tab has to be reconfigured according to the design after changing the structure.

First implementation and test has experienced with FIR filter which has following specifications.

Table 7.7 Tenth order Direct Form Symmetric FIR filter specifications

Direct Form FIR Symmetric 10th Order Filter FDATool Specifications			
Option	**Value**	**Option**	**Value**
Response Type	Low Pass	Input Word Length	8
Design Method	FIR Equir.	Input Fraction Length	7
Filter Order	10	Filter Precision	Specify All
Fs	44100Hz	Output Word Length	8
Fpass	3500Hz	Output Fraction Length	7
Fstop	4500Hz	Rounding Mode	Nearest (Co.)

Numerator Word Length	16	Overflow Mode	Saturate
Best-Prec. Frac. Lengths	Selected	Product W. Length	16
Use Unsig. Represent.	Cleared	Product Frac. Length	16
Scale Numerator Coeff.	Cleared	Accum. Word Length	18
Filter Arithmetic	Fixed-Point	Accum Frac. Length	16
Numerator Word Length	8		

Expected magnitude and phase responses of the filter with given specifications in Table 7.7 are shown in Figure 7.39 and Figure 7.40.

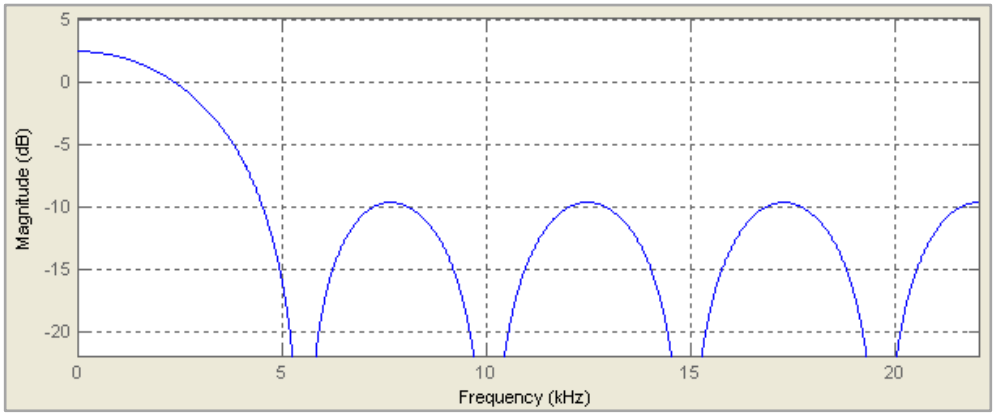

Figure 7.39 Magnitude response of FIR filter according to specifications in Table 7.7

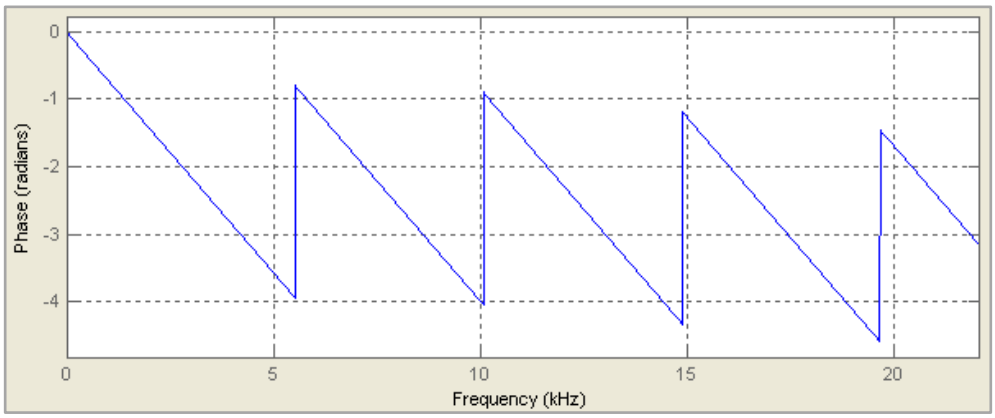

Figure 7.40 Phase response of FIR filter according to specifications in Table 7.7

Following sample view Figure 7.41 is taken from the processed audio file SYM10_out.wav after sending it through designed system. This file is stored in the relevant directory of the attachment CD.

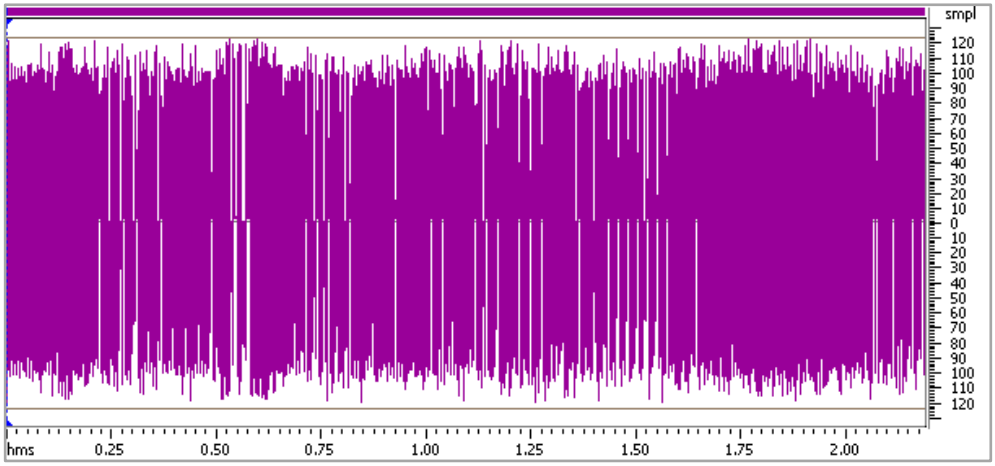

Figure 7.41 Samples of SYM10_out.wav from 0 to 2.20s

To make the above figure simpler and see it detailed, it is zoomed into the first 7ms and shown in Figure 7.42.

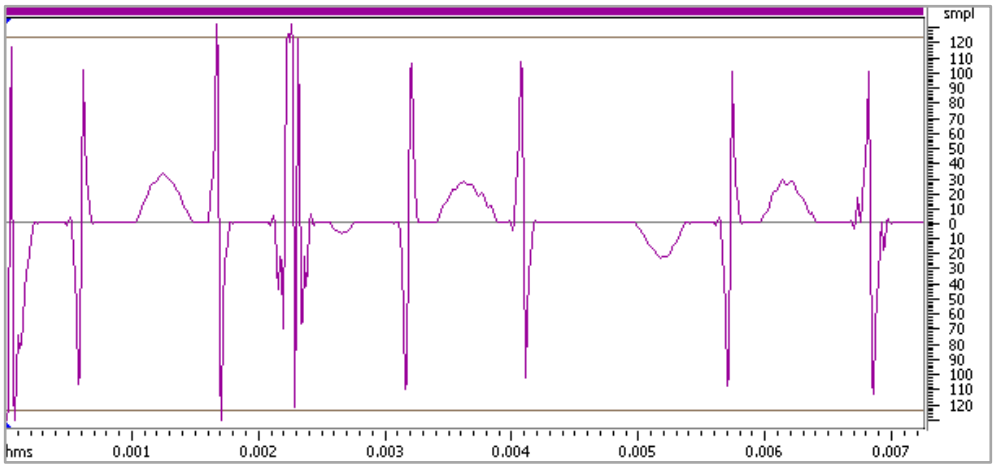

Figure 7.42 Samples of SYM10_out.wav from 0 to 7ms

Frequency analysis of the SYM10_out.wav file with shown specifications is shown in Figure 7.43.

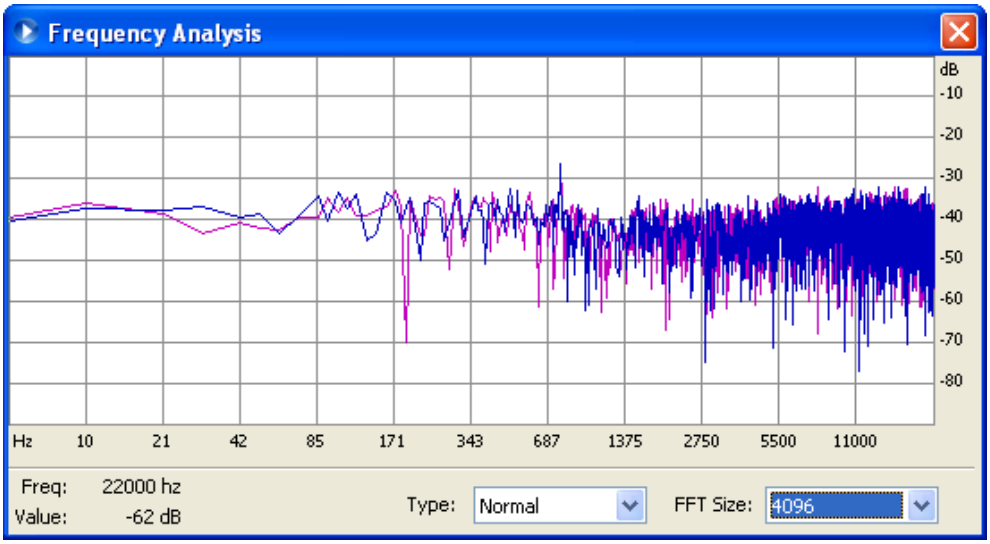

Figure 7.43 Frequency analysis of the SYM10_out.wav

Next implementation and test has experienced with FIR filter which has following specifications.

Table 7.8 Twentieth order Direct Form Symmetric FIR filter specifications

Direct Form FIR Symmetric 20th Order Filter FDATool Specifications			
Option	**Value**	**Option**	**Value**
Response Type	Low Pass	Input Word Length	8
Design Method	FIR Equir.	Input Fraction Length	7
Filter Order	20	Filter Precision	Specify All
Fs	44100Hz	Output Word Length	8
Fpass	3500Hz	Output Fraction Length	7
Fstop	4500Hz	Rounding Mode	Nearest (Co.)
Numerator Word Length	16	Overflow Mode	Saturate
Best-Prec. Frac. Lengths	Selected	Product W. Length	16
Use Unsig. Represent.	Cleared	Product Frac. Length	16
Scale Numerator Coeff.	Cleared	Accum. Word Length	18
Filter Arithmetic	Fixed-Point	Accum Frac. Length	16
Numerator Word Length	8		

Expected frequency and phase responses of the filter according to given specifications in Table 7.8 are shown in Figure 7.44 and Figure 7.45.

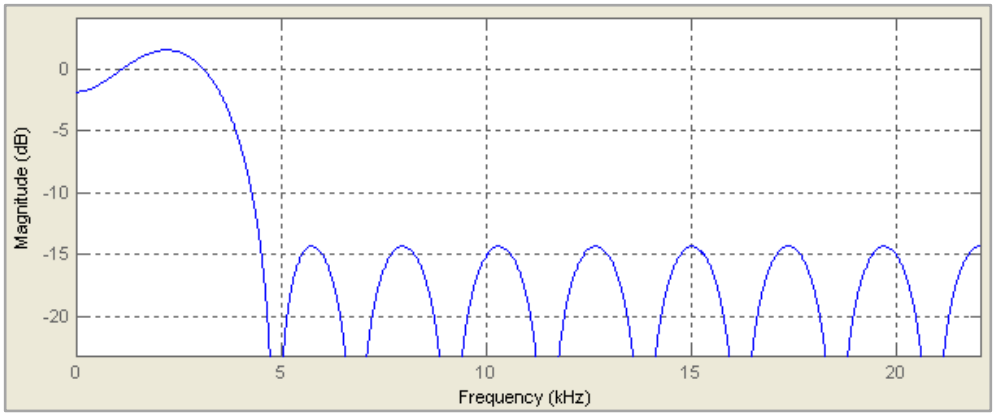

Figure 7.44 Magnitude response of FIR filter according to specifications in Table 7.8

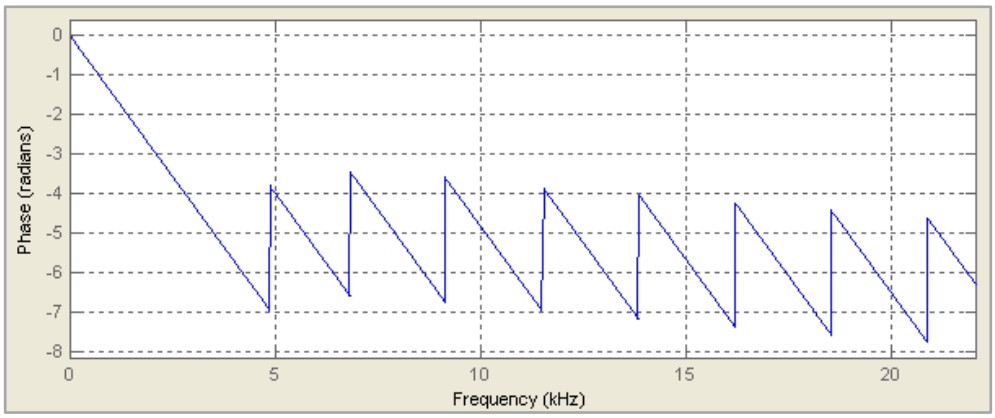

Figure 7.45 Phase response of FIR filter according to specifications in Table 7.8

Audio file I_44100.wav is sent through designed system, and following sample view Figure 7.46 is taken from the processed audio file SYM20_out.wav. This file is stored in the relevant directory of the attachment CD.

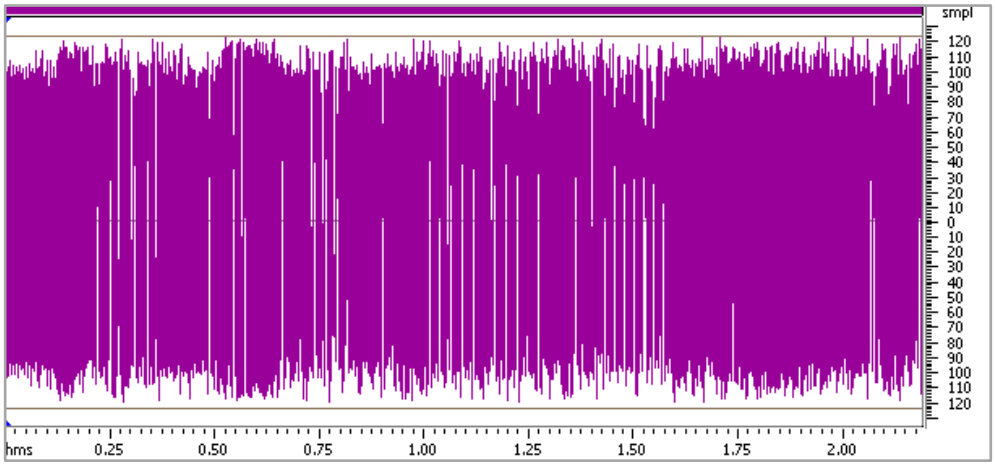

Figure 7.46 Samples of SYM20_out.wav from 0 to 2.20s

To make the above figure simpler and see it detailed, it is zoomed into the first 7ms and shown in Figure 7.47.

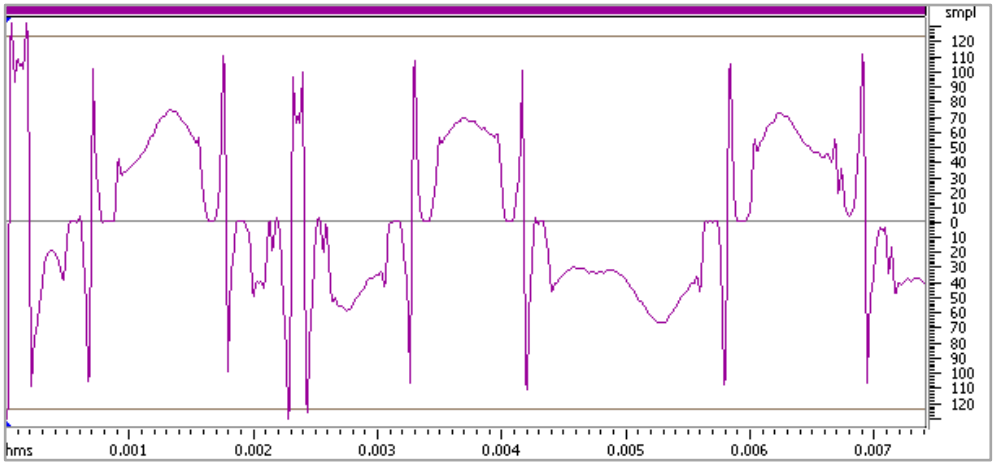

Figure 7.47 Samples of SYM20_out.wav from 0 to 7ms

Frequency analysis of the SYM20_out.wav file with shown specifications is shown in Figure 7.48.

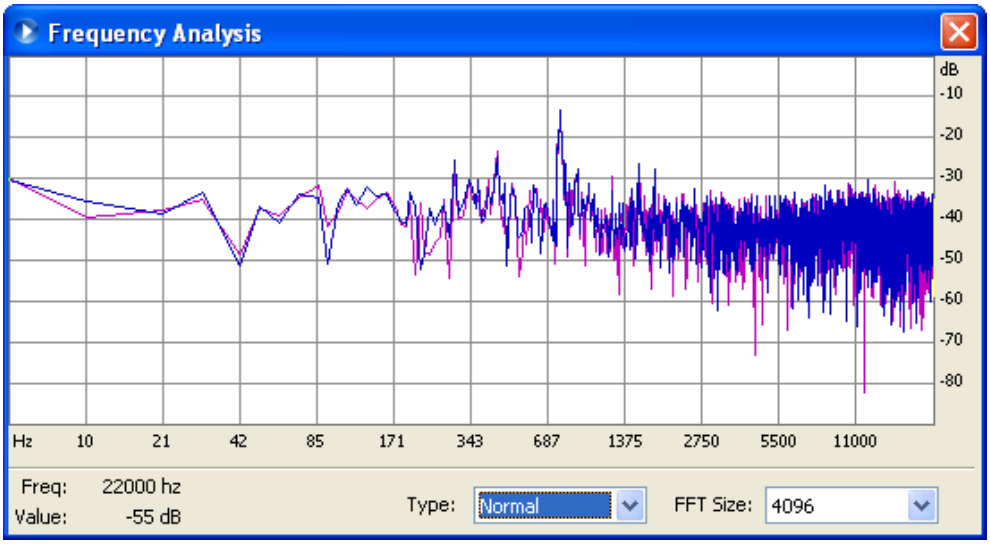

Figure 7.48 Frequency analysis of the SYM20_out.wav

Next implementation and test has experienced with FIR filter which has following specifications.

Table 7.9 Fortieth order Direct Form Symmetric FIR filter specifications

Direct Form FIR Symmetric 40th Order Filter FDATool Specifications			
Option	**Value**	**Option**	**Value**
Response Type	Low Pass	Input Word Length	8
Design Method	FIR Equir.	Input Fraction Length	7
Filter Order	40	Filter Precision	Specify All
Fs	44100Hz	Output Word Length	8
Fpass	3500Hz	Output Fraction Length	7
Fstop	4500Hz	Rounding Mode	Nearest (Co.)
Numerator Word Length	16	Overflow Mode	Saturate
Best-Prec. Frac. Lengths	Selected	Product W. Length	16
Use Unsig. Represent.	Cleared	Product Frac. Length	16
Scale Numerator Coeff.	Cleared	Accum. Word Length	18
Filter Arithmetic	Fixed-Point	Accum. Frac. Length	16
Numerator Word Length	8		

Expected frequency and phase responses of the filter according to given specifications in Table 7.9 are shown in Figure 7.49 and Figure 7.50.

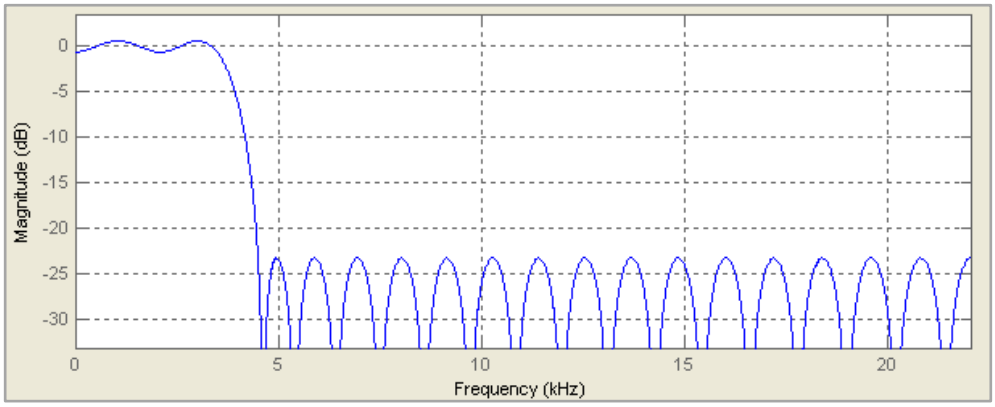

Figure 7.49 Magnitude response of FIR filter according to specifications in Table 7.9

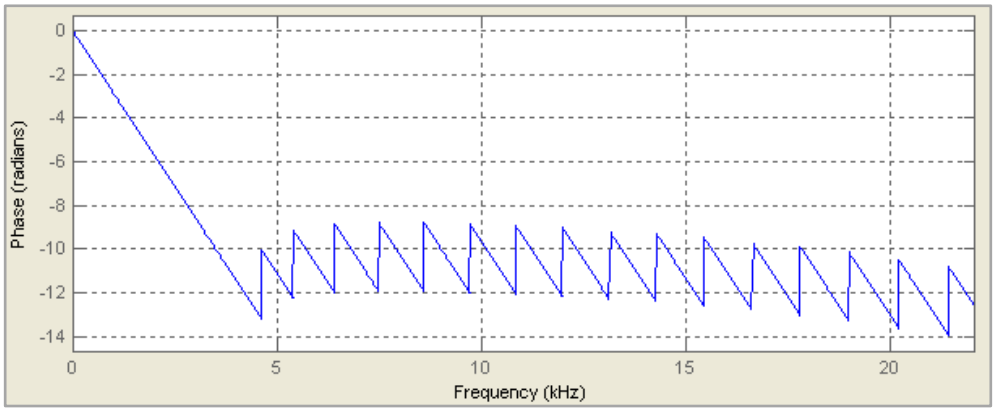

Figure 7.50 Phase response of FIR filter according to specifications in Table 7.9

Following sample view Figure 7.51 is taken from the processed audio file SYM40_out.wav after sending I_44100.wav through designed system. This file is stored in the relevant directory of the attachment CD.

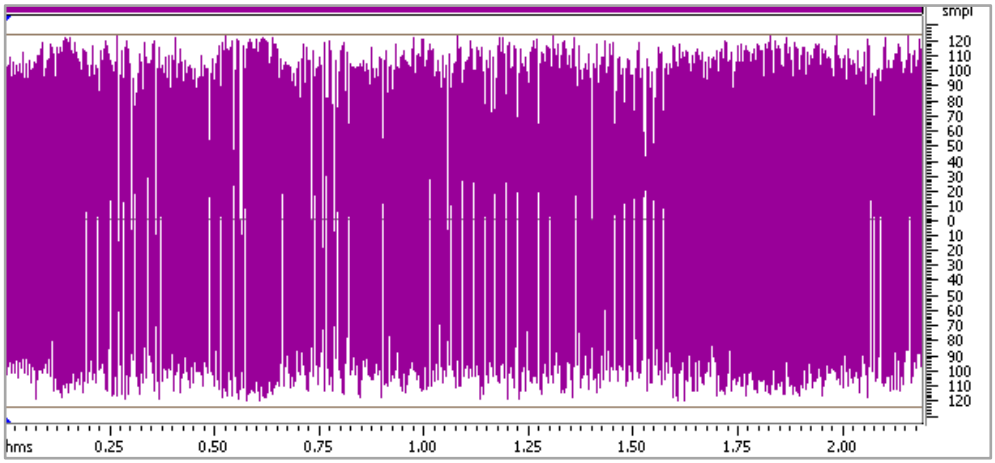

Figure 7.51 Samples of SYM40_out.wav from 0 to 2.20s

To make the above figure simpler and see it detailed, it is zoomed into the first 7ms and shown in Figure 7.52.

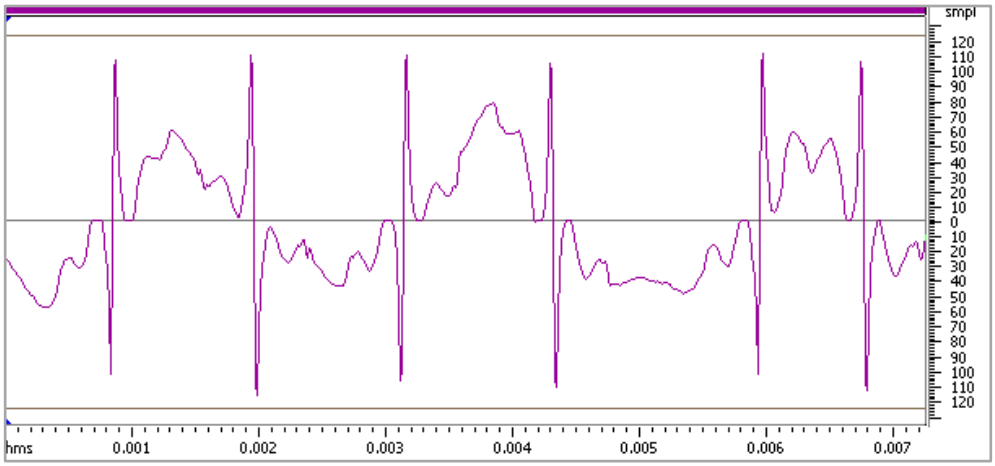

Figure 7.52 Samples of SYM40_out.wav from 0 to 7ms

Frequency analysis of the SYM40_out.wav file with selected options is shown in Figure 7.53.

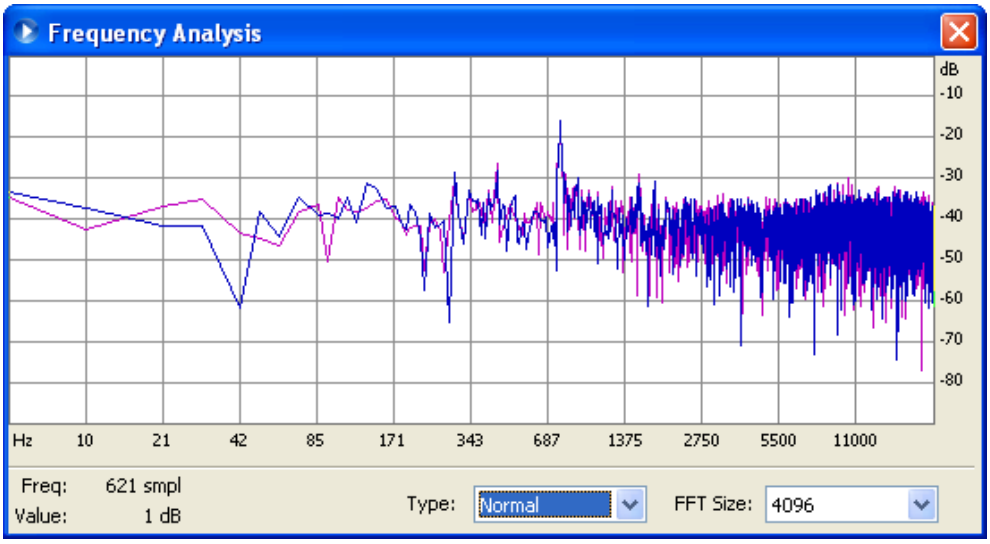

Figure 7.53 Frequency analysis of the SYM40_out.wav

From the results of the filtering operations on audio files with different filters which has different orders or architectures it can be said that sharp jumps and falls have occurred on the output signals.

Original audio file has been filtered with 4kHz cutoff frequency low pass filter by Power Sound Editor and frequency response of the filtered data is shown in Figure 7.54. Filtered file I_44100_out_4khz.wav is stored in the relevant directory of the attachment CD.

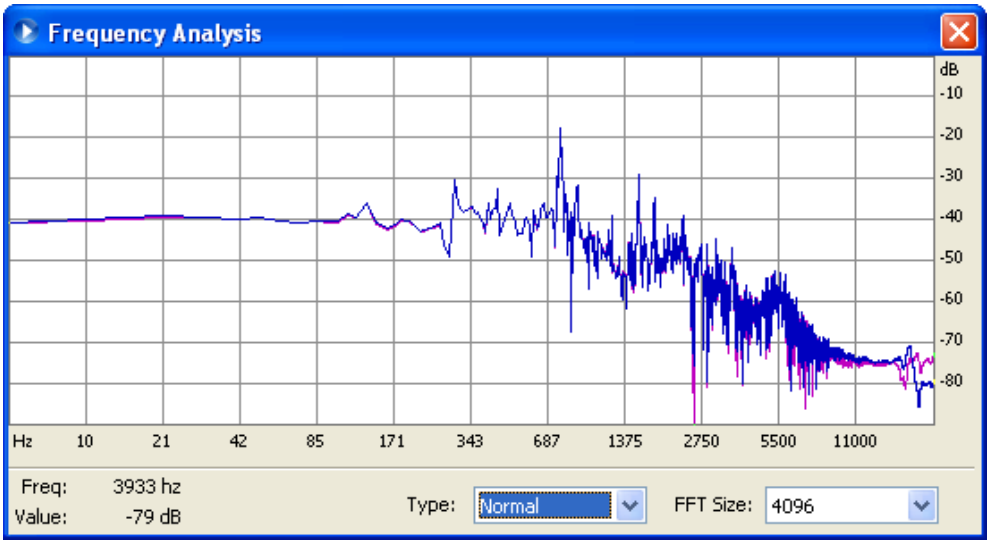

Figure 7.54 Frequency analysis of filtered file I_44100_out_4khz.wav

Expected result does not fit with the received results because of the sharp jumps and fall downs in the samples. To test if there is a problem in the filter modules or other modules, an audio file has been constructed using MatLab and tested on all implementations. For constructing this file, “wavread” and “zeros” commands were used In MatLab. For more information about these commands, please refer to Matlab/Help.

This audio file I_44100_zeros.wav which includes only zero samples is sent through the configured system and received data is saved as I_44100_zeros_out.wav. This file is stored in the relevant directory of the attachment CD. Received data has only zeros as original one including no jumps or no sharp sample changes. This shows there is no external module problem inside the project.

Every single option in FDATool except 8-bit input and 8-bit output has been tried for generating new VHDL files of filters to test the audio data but meaningless sharp jumps and fall downs between samples occurred in every received output.

All command lines of the VHDL modules has been analyzed and mathematical algorithms has been understood.

As the filter designs were constructed according to the top design with 8-bit input and 8-bit output, expression of audio samples between +127 and -128 were processed through mathematical operations.

Numbers multiplied with coefficients and stored in temporary registers. For full precision in the output, an FIR filter which has 8-bit coefficient lengths and 10^{th} order, minimum accumulator length, minimum product length and output width length are shown in next two figures.

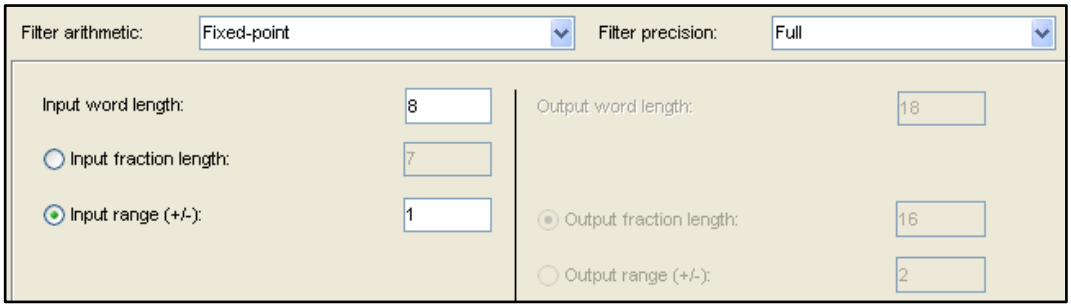

Figure 7.55 Proper specifications chosen automatically by MatLab for full precision in the output of the filter

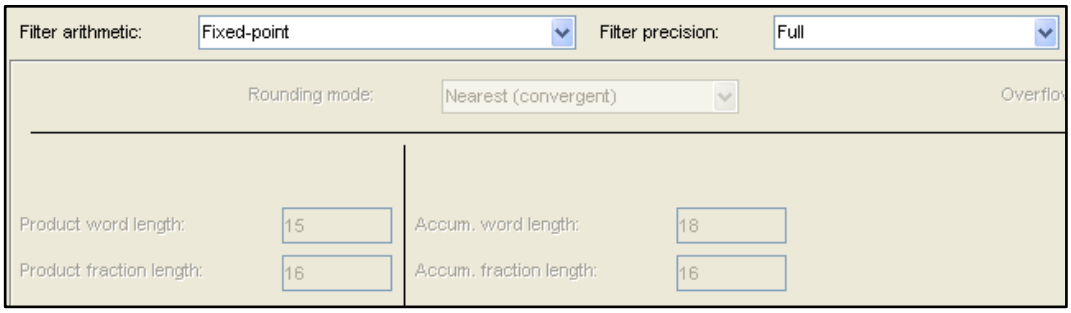

Figure 7.56 Proper specifications chosen automatically by MatLab for full precision in the output of the filter

In the figures above, for a full precision of filter according to the specifications, output word length should have 18-bits length and. Also product and accumulator word lengths should be as seen in figures above.

Because result of multiplication operation on coefficients and samples; outputs a number which has to be wide enough to represent result. As it is chosen Specify All option is selected according to the top design, resizing operation is processed inside FPGA to fit a wide number to slammer bit length; so information loss occurs in the output data. That causes the unexpected noise in the output audio files.

In this example resizing operation is processed between temporary sum registers and product registers inside FPGA, last operation which resizes the final result into 8-bit length causes the loss of information in the data. Resizing codes of final result to 8-bit length is shown in next lines.

```
-------------------------------------------------------------------
output_typeconvert <= (7 => '0', OTHERS => '1')
  WHEN (finalsum(17) = '0' AND finalsum(16) /= '0') OR
(finalsum(17) = '0' AND finalsum(16 DOWNTO 9) = "01111111") special
case
ELSE (7 => '1', OTHERS => '0')
  WHEN finalsum(17) = '1' AND finalsum(16) /= '1'
ELSE (resize(shift_right(finalsum(17) & finalsum(16 DOWNTO 0) + (
"0" & (finalsum(9) & NOT finalsum(9) & NOT finalsum(9) & NOT
finalsum(9) & NOT finalsum(9) & NOT finalsum(9) & NOT finalsum(9) &
NOT finalsum(9) & NOT finalsum(9))), 9), 8));
-------------------------------------------------------------------
```

It is clearly shown that scaling of an 18-bit result to 8-bit number is operated in the VHDL codes for a few special cases and a generalization is operated except these few cases. This causes the loss of information for different which are generalized in the codes.

Assuming that audio signal has amplitude is between +1 and -1 and represented with signed 8-bit samples. In this case +1 is relevant to +127 and -1 is relevant to -128. After the mathematical operations inside the filter algorithm with full precision, sample results represented with signed 18-bit numbers for above example. In this case +1 is represented with 131071 and -1 is represented with -131072. But the resize command can’t provide such scaling operation between 8-bit to 18-bit numbers. Dividing integer value of MSB (Most Significant Bit) in the 18-bit result to the integer value on MSB in the 8-bit result 1024 is the correct step correspondence between input and output. VHDL does not support such scaling between input and output with these length specifications, except for a few special cases.

CHAPTER EIGHT
CONCLUSION

Throughout this thesis work, a system to filter digital signals through FPGA is designed. Digital filter applications with FPGA's have attracted attention with the technologic growth of FPGA's in the past two decades. However digital filtering applications include large numbers of mathematical operations and need MAC units according to the order of filter. This requirement size differs with the types and architectures of filter algorithms and therefore choosing optimum order and proper architecture can reduce the area and units used in FPGA.

Hardware descriptive language is a complicated software language and requires hardware background to create stable and working designs. VHDL is used to construct design parts separately to improve project step by step. Zero values are given to the initial conditions of registers and pipelines used in project, to get better results and prevent crashes.

Transmitting and receiving algorithms are designed and implemented for 8-bit samples according to the serial port RS-232 which supports maximum 8 data bits between start and stop bits. Serial port is used for communication between FPGA and computer to send and receive digital data. However using different communication technique could turn better result out

In the filter design numerator word length affects the product lengths which are retrieved from multiplication of samples and coefficients. Choosing wider numerator length results loss of information in the received data because of resizing. Thus numerator lengths reduced closer to the input word length to decrease loss of information in the received data. Also synchronous reset type in filter specifications is chosen to prevent system crushes because of synchronization errors on FPGA.

Finite state machine algorithm is used both in receiving and transmitting parts to make data flow precise without any errors.

For a filter with any order specification, symmetric structure uses about 50% less MAC units than direct form architecture and about 40% less LUTs than other architectures. On the other way this architecture could perform at about 50% speed of sampling rate compared to the direct form architecture.

In conclusion, specifications of the design and software to be used should be carefully analyzed before implementation and some tradeoffs between speed and area has to be considered by designer according to these results.

8.1 Futureworks

Even though the results of the system are in accordance with theoretical expectations they are evaluated using serial port which will not give best results. Using Ethernet communication for data flow with higher bit depth and sampling rates will return more reliable results.

System is designed for processing stored data and it can be enhanced by improving software and hardware for real time application by connecting input and output devices to the PC.

REFERENCES

Al Mahdi Eshtawie, M. & Bin Othman, M. (2008). An Algorithm Proposed for FIR Filter Coefficients Representation. *International Journal of Applied Mathematics and Computer Sciences 4 (1)*.

Bicakci, S., Çetinkaya, B. & Karaboğa, N. (2005). *Using FPGA in the Digital Filter Design*. Retrieved March 9, 2008, from http://www.emo.org.tr/ekler/7b58836dc941cc4_ek.pdf.

Chou, C., Mohanakrishnan, S. & Evans, J. B. (1993). *FPGA implementation of digital filters*. Retrieved Feruary 11, 2009, from http://citeseerx.ist.psu.edu/

Chu, P. P. (2008). *FPGA prototyping by VHDL examples: Xilinx Spartan – 3 version*. Hoboken, NJ: Wiley Interscience.

Dikmeşe, Ş. (2007). *Kablosuz haberleşme sistemlerinde FPGA uygulaması*. M.Sc. Thesis, Kocaeli Üniversitesi.

Dueck, R. (2004). *Digital design with CPLD applications and VHDL* (2nd ed.). New York: Delmar Cengage Learning.

Elhossini, A., Shawki A. & Dony, R. (2006). *An FPGA implementation of LMS adaptive filter for audio processing*.

Jesman, R, Vallina, F. M., Saniie, J. (2007) *Creating a simple embedded system and adding custom peripherals using Xilinx EDK software tools*. Retrieved February 22, 2009, from http://ecasp.ece.iit.edu/mb**tutorial**.pdf.

Pedroni, V. A. (2004). *Circuit design with VHDL*. Massachusetts: The MIT Press.

Perry, D. L. (2002). *VHDL: Programming by example* (4th ed.). New York: McGraw-Hill Professional.

Petrone, J. (2004). *Adaptive filter architectures for FPGA implementation*. M.Sc. Thesis, The Florida State University.

Tamer, Ö. (2007). FPGA based smart antenna implementation. Ph.D. Thesis, Dokuz Eylül University.

Vaidyanathan, P. P. (2001). *Filter banks in digital communications*. Retrieved February 8, 2008, from http://citeseerx.ist.psu.edu/

Wang, Y. (2005). *Implementation of digital filter by using FPGA*. B.Sc. Thsis, Curtin University of Technology.

White, S. (2000). *Digital signal processing: Filtering approach*. New York: Delmar Cengage Learning

Woods, R., McAllister, J., Turner, R., Yi, Y. & Lightbody, G. (Eds.). (2008). *FPGA-based implementation of signal processing systems*. Hoboken, NJ: Wiley.

Zack, S., Dhanami, S. (March 18, 2004). *DSP Co-Processing in FPGAs: Embedding high performance, low-cost DSP functions*. Retrieved November 12, 2008, from http://www.xilinx.com/support/documentation/white_papers/wp212.pdf

APPENDIX

CODES

9.1 VHDL Codes

9.1.1 VHDL Codes for Transmitter

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Transmitter is
    Port ( clock : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           input : in  STD_LOGIC_VECTOR (7 downto 0);
           start : in  STD_LOGIC;
           output: out  STD_LOGIC);
end Transmitter;

architecture Behavioral of Transmitter is
           type state_type is (st1_a, st2_b, st3_c, st4_d);
           signal state, next_state: state_type;
           signal reg_n            : std_logic_vector (0 to
9):="0000000000";
           signal reg              : std_logic_vector (7 downto 0);
           signal load_enable: std_logic;
           constant bolme          : integer:=2171; --57600 baud
           signal sayac            : integer range 0 to bolme:=0;
           signal sayac2           : integer range 0 to 10:=0;
           signal s_reg_en   : std_logic;
           signal start_ctrl: std_logic:='0';
begin
           reg(7) <= input(0);
           reg(6) <= input(1);
           reg(5) <= input(2);
           reg(4) <= input(3);
           reg(3) <= input(4);
           reg(2) <= input(5);
           reg(1) <= input(6);
           reg(0) <= input(7);

      SYN_PROC: process (clock)
           begin
           if (clock'event and clock = '1') then
                 if (reset = '1') then
                       state <= st1_a;
                 else
                       state <= next_state;
                 end if;
           end if;
           end process;

           process (start,clock)
           begin
                 if clock'event and clock = '1' then
                       if(start = '1') then
```

```
                        start_ctrl <= '1';
                  end if;
                  if sayac2 = 9 then
                        start_ctrl <= '0';
                  end if;
            end if;
      end process;


      process (clock, start_ctrl, load_enable, s_reg_en)
      begin
      if clock'event and clock = '1' then
            if (start_ctrl = '1') then -- degisti
                  if load_enable = '1' then
                        reg_n(1 to 8) <= reg;
                        reg_n(9) <= '1';
                        reg_n(0) <= '0';
                  end if;
                  if s_reg_en = '1' then
                        reg_n <= reg_n(1 to 9) & '0';
                  end if;
            else
                  reg_n(0) <= '1';
            end if;
      end if;
      end process;

      process (clock)
      begin
      if (clock'event and clock = '1') then
            output <= reg_n(0);
      end if;
      end process;

OUTPUT_DECODE: process (state,clock)
      begin
      if clock'event and clock='1' then
            case (state) is
                  when st1_a =>
                        load_enable <= '0';
                        sayac             <=  0 ;
                        sayac2            <=  0 ;
                        s_reg_en          <= '0';
                  when st2_b =>
                        load_enable <= '1';
                        sayac <= sayac+1;
                  when st3_c =>
                        load_enable <= '0';
                        sayac <= sayac+1;
                        s_reg_en          <= '0';
                  when st4_d =>
                        sayac2 <= sayac2+1;
                        sayac             <= 0;
                        s_reg_en          <= '1';
            end case;
      end if;
      end process;
```

```
      NEXT_STATE_DECODE: process (state, start_ctrl, sayac, sayac2,
clock)
            begin
            next_state <= state;
            case (state) is
                  when st1_a  =>
                        if start_ctrl = '1' then -- changed
                              next_state <= st2_b;
                        end if;
                  when st2_b =>
                        next_state <= st3_c;
                  when st3_c =>
                        if sayac = (bolme-1) then
                              next_state <= st4_d;
                        end if;
                  when st4_d =>
                        if sayac2 = 9 then
                              next_state <= st1_a;
                        else
                              next_state <= st3_c;
                        end if;
            end case;
            end process;

end Behavioral;
```

9.1.2 VHDL Codes for Receiver

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Receiver is
Port ( input            : in  STD_LOGIC;
       reset            : in  STD_LOGIC;
       clock            : in  STD_LOGIC;
       output     : out STD_LOGIC_VECTOR (7 downto 0):="00000000";
       start            : out STD_LOGIC:='0' );
end Receiver;


architecture Behavioral of Receiver is

type state_type is (st1_idle, st2_sayac, st3_reg, st4_reg1,
st5_sayac2);

      signal state, next_state: state_type;
      signal s_reg_en         :std_logic:='0';
      signal reg_n      :std_logic_vector (9 downto
0):="0000000000";
      signal output1          :std_logic_vector (9 downto 0);
      constant bolme          :integer:=2171; --57600 baud
      signal sayac            :integer range 0 to bolme:=0;
      signal sayac2           :integer range 0 to bolme:=0;
      signal sayac_stop       :integer range 0 to bolme:=0;
```

```
	signal counter		:integer range 0 to bolme*100:=0;
	signal enable		:std_logic:='0';
	signal c		:std_logic_vector(7 downto 0):="11111110";
	signal m				:std_logic:='0';
	signal stop_en		:std_logic:='0';


begin
	SYNC_PROC:  process (clock, reset)
	begin
	if clock'event and clock ='1' then
		if reset='1' then
			state <= st1_idle;
			else
			state <= next_state;
		end if;
	end if;
	end process;

	process (clock, s_reg_en)
	begin
	if clock'event and clock ='1' and reset = '0' then
		if s_reg_en ='1' then
			reg_n <= reg_n (8 downto 0) & input;
		end if;
	end if;
	end process;

	start <= stop_en;
	process(clock,stop_en,sayac_stop)
	begin
		if clock'event and clock='1' then
			if(stop_en='1') then
				sayac_stop <= sayac_stop + 1;
			else
				sayac_stop <= 0;
			end if;
		end if;
	end process;

	process (state, clock)
	begin
	if clock'event and clock='1' then
		if (state = st1_idle) then
			c(0) <= output1(1);
			c(1) <= output1(2);
			c(2) <= output1(3);
			c(3) <= output1(4);
			c(4) <= output1(5);
			c(5) <= output1(6);
			c(6) <= output1(7);
			c(7) <= output1(8);
		end if;
	end if;
	end process;

	output <= c;
	output1(0) <= reg_n(9);
	output1(1) <= reg_n(8);
```

```
	output1(2) <= reg_n(7);
	output1(3) <= reg_n(6);
	output1(4) <= reg_n(5);
	output1(5) <= reg_n(4);
	output1(6) <= reg_n(3);
	output1(7) <= reg_n(2);
	output1(8) <= reg_n(1);
	output1(9) <= reg_n(0);

	process (clock, state)
	begin
	if (clock'event and clock='1') then
		case (state) is
			when  st1_idle =>
				sayac       <= 0;
				sayac2      <= 0;
				s_reg_en    <= '0';
			when st2_sayac =>
				sayac <= sayac+1;
			when st3_reg      =>
				s_reg_en    <= '1';
				sayac2 <= sayac2+1;
			when st4_reg1     =>
				s_reg_en    <= '0';
				sayac <= 0;
			when st5_sayac2 =>
				sayac <= sayac+1;
		end case;
	end if;
	end process;

	NEXT_STATE_DECODE: process (state, input, sayac, sayac2,
clock)
	begin
	next_state <= state;
	case (state) is
		when st1_idle     =>

			if(sayac_stop > (bolme-1)/2 ) then
				stop_en <= '0';
			end if;

			if input = '0'    then
				next_state  <=    st2_sayac;
			else
				next_state  <=    st1_idle;
			end if;
		when st2_sayac =>
			if(sayac_stop > (bolme-1)/2 ) then
				stop_en <= '0';
			end if;
			if sayac    =     ((bolme-1)/2) then
				next_state <= st3_reg;
			end if;
		when st3_reg      =>
			if(sayac_stop > (bolme-1)/2 ) then
				stop_en <= '0';
			end if;
			next_state <= st4_reg1;
```

```
            when st4_reg1     =>
                  if(sayac_stop > (bolme-1)/2 ) then
                        stop_en <= '0';
                  end if;

                  if (sayac2 = 10) then
                        next_state <= st1_idle;
                        m <= '1';
                        stop_en <='1';
                  else
                        next_state  <= st5_sayac2;
                  end if;
            when st5_sayac2 =>
                  if (sayac = (bolme-1)) then
                        next_state <= st3_reg;
                  end if;
      end case;
      end process;

      process (sayac2, clock)
      begin
      if clock'event and clock = '1' then
            if (sayac2<8) then
                  enable <= '1';
            else
                  enable <= '0';
            end if;
      end if;
      end process;

end Behavioral;
```

9.1.3 VHDL Codes for Delayer

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity Delay is
    Port ( clock       : in  STD_LOGIC;
           rec_out     : in  STD_LOGIC:='0';
           tra_in      : out  STD_LOGIC);
end Delay;



architecture Behavioral of Delay is
                  type state_type is (st1_gor, st2_bekle, st3_ver);
                  signal state, next_state: state_type;
                  constant CD             : integer:=2171; --57600
baud
                  signal countbekle       : integer range 0 to
2*CD:=0;
                  signal counttut         : integer range 0 to
2*CD:=0;
```

```
      signal      tra_in_ctrl : std_logic:='0';
      signal      rec_out_ctrl: std_logic:='0';
      signal      countbekle_en: std_logic:='0';
      signal      counttut_en : std_logic:='0';

begin

      SYNC: process (clock)
      begin
      if (clock'event and clock = '1') then
            state <= next_state;
      end if;
      end process;


      process (clock)
      begin
      if clock'event and clock = '1' then
            if countbekle_en = '1' then
                  countbekle <= countbekle+1;
            else
                  countbekle <= 0;
            end if;

            if counttut_en = '1' then
                  counttut <= counttut+1;
            else
                  counttut <= 0;
            end if;
      end if;
      end process;


      OUTPUT: process (state,clock)
      begin
      if clock'event and clock='1' then
            case (state) is
                  when st1_gor =>
                        counttut_en       <= '0';
                        countbekle_en     <= '0';
                  when st2_bekle =>
                        countbekle_en     <= '1';
                  when st3_ver =>
                        countbekle_en     <= '0';
                        counttut_en       <= '1';
            end case;
      end if;
      end process;



      STATE_DECODE: process (rec_out, countbekle, counttut)
      begin
      next_state <= state;
      case (state) is
            when st1_gor      =>
                  if rec_out = '1' then
                        next_state <= st2_bekle;
                  end if;
```

```
                when st2_bekle =>
                            if countbekle = CD*2 then
                                  next_state <= st3_ver;
                            end if;
                when st3_ver =>
                      if counttut = CD*2 then
                            next_state <= st1_gor;
                      end if;
            end case;
            end process;

tra_in_ctrl <=  counttut_en;

tra_in <= tra_in_ctrl;
rec_out_ctrl <= rec_out;

end Behavioral;
```

9.2 C# Codes

9.2.1 Write Read Serial Port Codes

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO;
using System.IO.Ports;
using System.Threading;

namespace write_read_comport
{
    public partial class Form1 : Form
    {
        SerialPort port;

        bool send_all = false;      // if true sends all bytes, if
false sends a byte with evey click on filter
        bool taken = true;         // when it takes filtered byte
from port, this'll be true, so we can send a new byte
        bool cont = true;          // controling continuosly sending
data

        bool port_listen = false;
        int counter;            // counts the index of sample byte
of file.
        int counter_listen;     // counts listening bytes.
        int length_all;
        byte[] wave;
        byte[] filtered_wav;

        public Form1()
```

```
        {

            InitializeComponent();
            boudsCBox.SelectedIndex = 5;
            port = new SerialPort();

            counter = 0;
            counter_listen = 0;
            length_all = 0;
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            CheckForIllegalCrossThreadCalls = false;

            // adding all com ports to combo box
            try
            {
                portsCBox.Items.AddRange(SerialPort.GetPortNames());
            }
            catch (Exception ex) { MessageBox.Show(ex.Message); }

            // disable buttons
            connectBtn.Enabled = false;
            disconnectBtn.Enabled = false;
            filterBtn.Enabled = false;
            openFileBtn.Enabled = false;
        }

        private void portsCBox_SelectedIndexChanged(object sender,
EventArgs e)
        {
            connectBtn.Enabled = true;

        }

        private void connectBtn_Click(object sender, EventArgs e)
        {

            port.PortName = (string)portsCBox.SelectedItem;
            port.BaudRate =
Int32.Parse(boudsCBox.SelectedItem.ToString());
            port.DataBits = 8;
            port.Parity = (Parity)Enum.Parse(typeof(Parity),
"None");
            port.StopBits = (StopBits)Enum.Parse(typeof(StopBits),
"1");

            try {
                port.Open();
                port_listen = true;
                portsCBox.Enabled = false;
                connectBtn.Enabled = false;
                disconnectBtn.Enabled = true;
                openFileBtn.Enabled = true;
            }
            catch (Exception ex) { MessageBox.Show(ex.Message); }

        }
```

```
        private void button1_Click(object sender, EventArgs e)
        {
            BinaryReader reader = new
BinaryReader(File.Open(filenameTxt.Text, FileMode.Open));
            int pos = 0;
            int length = (int)reader.BaseStream.Length;
            length_all = length;

            byte sample = 0;
            wave = new byte[length];
            filtered_wav = new byte[length];
            // first 44 byte (header)
            while ((pos < length) && (pos < 44)) {
                sample = reader.ReadByte();
                filtered_wav[pos] = sample;
                wave[pos] = sample;
                pos += sizeof(byte);
            }

            // whole file to memory (array)
            while (pos < length)
            {
                wave[pos] = reader.ReadByte();
                pos += sizeof(byte);
            }

            reader.Close();
            counter = 44;
            counter_listen = 44;
            //

            // progress bar
            progressBar1.Minimum = 0;
            progressBar1.Maximum = length;
            progressBar1.Visible = true;
            progressBar1.Step = 1;
            progressBar1.Style = ProgressBarStyle.Blocks;
            //

            // start writer thread to write wave data to filter com.
            backgroundWorker2.RunWorkerAsync();
            backgroundWorker3.RunWorkerAsync();
        }

        private void disconnectBtn_Click(object sender, EventArgs e)
        {
            try
            {
                port_listen = false;
                portsCBox.Enabled = true;
                filterBtn.Enabled = false;
                openFileBtn.Enabled = false;
                port.Close();
                counter = 0;
            }
            catch (Exception ex) { MessageBox.Show(ex.Message); }
```

```
        }

        private void checkBox1_CheckedChanged(object sender,
EventArgs e)
        {
            if (checkBox1.Checked)
            {
                send_all = true;
            }
            else { send_all = false; }
        }

        private void takeHeaderBtn_Click(object sender, EventArgs e)
        {
            openFileDialog1.Filter = "Wave files (*.wav)|*.wav|All
files (*.*)|*.*";
            openFileDialog1.FileName = "";
            if (openFileDialog1.ShowDialog() == DialogResult.OK) {
                filterBtn.Enabled = true;
                filenameTxt.Text = openFileDialog1.FileName;
            }

        }

        private void backgroundWorker2_DoWork(object sender,
DoWorkEventArgs e)
        {
            filterBtn.Enabled = false;
            byte incoming = 0;

            while ((counter < wave.Length) && (cont))
            {
                if (!send_all)
                {
                    cont = false;
                }
                else { cont = true; }

                // write byte to port
                //if (taken) // if we did not take the data, we
should wait the filter to writes.
                //{
                    if (port.IsOpen)
                    {
                        port.Write(wave,counter,1);
                        taken = false;
                        counter +=sizeof(byte);
                    }


                    while (true)
                    {
                        if (port.BytesToRead > 0)
                        {
                            incoming = (byte)port.ReadByte();
                            port.DiscardInBuffer();
                            if (counter_listen <
filtered_wav.Length)
                            {
```

```
                                filtered_wav[counter_listen] =
incoming;
                            }
                            counter_listen++;
                            break;
                        }
                        if (!send_all)
                        {
                            Thread.Sleep(100);
                        }
                    }


            }

            // close control
            if (counter >= wave.Length) {
                write_filetered_file();
            }

            cont = true;
            filterBtn.Enabled = true;
        }

        // progress bar refresh
        private void backgroundWorker3_DoWork(object sender,
DoWorkEventArgs e)
        {
            while (counter < length_all)
            {
                progressBar1.Value = counter;
                progressBar1.Refresh();
                progressBar1.Update();

                int number = (int)(((float)(counter+1) /
(float)length_all) * 100);
                progressLbl.Text = number.ToString() + " %";
                Thread.Sleep(500);
            }
            progressBar1.Value = length_all;
            progressLbl.Text = "100 %";
        }

        private void progressBar1_Click(object sender, EventArgs e)
        {

        }

        private void write_filetered_file() {
            BinaryWriter writer = new
BinaryWriter(File.Open("filtered.wav", FileMode.Create));

            int pos = 0;

            while (pos < filtered_wav.Length) {
                writer.Write(filtered_wav[pos]);
                pos++;
            }
```

```
                writer.Close();
            }
        }
    }
```

9.3 Filter Equation Tables

Following tables include bacground knowledge about filter design.

Table 9.1 Mathematical expressions of FIR filters according to architecture and order

Type	Symmetry of $h(n)$	N	α	β	Form of $H_M(e^{j\omega})$	Constraints on $H_M(e^{j\omega}) \cdot e^{j\beta}$
I	$h(n) = h(N-1-n)$ symmetric	odd	$\frac{N-1}{2}$	0 π	$\sum_{n=0}^{\frac{N-1}{2}} a(n) \cdot \cos \omega n$	real
II	$h(n) = h(N-1-n)$ symmetric	even	$\frac{N-1}{2}$	0 π	$\sum_{n=1}^{\frac{N}{2}} b(n) \cdot \cos \omega (n - \frac{1}{2})$	real $H_M(e^{j\pi}) = 0$
III	$h(n) = -h(N-1-n)$ anti-symmetric	odd	$\frac{N-1}{2}$	$\frac{\pi}{2}$ $3\pi/2$	$\sum_{n=1}^{\frac{N-1}{2}} c(n) \cdot \sin \omega n$	Pure Imaginary $H_M(e^{j0}) = H_M(e^{j\pi}) = 0$
IV	$h(n) = -h(N-1-n)$ anti-symmetric	even	$\frac{N-1}{2}$	$\frac{\pi}{2}$ $3\pi/2$	$\sum_{n=1}^{\frac{N}{2}} d(n) \cdot \sin \omega (n - \frac{1}{2})$	Pure Imaginary $H_M(e^{j0}) = 0$

Table 9.2 Suitability of FIR for filter responses

Type	Low pass	High pass	Band pass	Band stop
I				
II		Not suitable		Not suitable
III	Not suitable	Not suitable		Not suitable
IV	Not suitable			Not suitable