



**Design and Implementation
of a Hardware Accelerator for one-Dimensional
signal Filtering Operations**

Konstantinos Koutropoulos

June 2014

**Dissertation submitted in partial fulfilment for the degree of
Master of Science in *Communication & Information Systems***

**Department of Informatics & Communications
TEI of Central Macedonia**

Abstract

The objective of this thesis is the design of a low-pass Finite Impulse Response filter using hardware description language for FPGA implementation. The window design method was followed and the filter was described in VHDL. The design tool used for the synthesis of the filter is Quartus II v. 9.1 by Altera. Modelsim by Mentor Graphics was used for simulation, in order to verify the filter operation and the accuracy of the results. The comparison with a software-based implementation of the same filter demonstrates that the filter meets the requirements.

Bottom-up hierarchical design was used. The various components were first described in VHDL and then they were instantiated in order to produce the top design entity of the filter. Such filter components that need description are the shift register for the creation of the convolution window, the ROM stage where the filter coefficients are stored, the computationally demanding parallel multiplication stage and finally, the accumulation and normalization stage, where the output sample is computed. Specifications for the necessary data types were defined and alternative implementations of specific stages were tested, as a means to establish best design methodology.

We find that a filter with 101 coefficients can reproduce the original double precision filter specifications using just 14% of the resources of a Cyclone II EP2C35 low cost FPGA device. Also, it can achieve a maximum clock frequency of 50 MHz.

Attestation

I understand the nature of plagiarism, and I am aware of the University's policy on this.

I certify that this dissertation reports original work by me.

Signature

Date

Acknowledgements

I would like to thank my supervisor Dr. John Kalomiros for his guidance and assistance during the elaboration of this thesis. I would also like to thank my family for their patience and support.

Table of Contents

Abstract	i
Attestation.....	ii
Acknowledgements	iii
List of Figures.....	viii
List of Tables	xii
1 Introduction	1
1.1 Background and Context	1
1.2 Scope and Objectives.....	1
1.3 Achievements	2
1.4 Overview of Dissertation.....	2
2 State-of-The-Art	3
3 Digital Filters.....	4
3.1 Introduction	4
3.2 Describing Systems	4
3.2.1 Time Domain Response.....	5
3.2.2 Frequency Domain Response $H(j\Omega)$	5
3.2.3 System Function $H(s)$	6
3.2.3.1 Poles and Zeros.....	7
3.3 Filter specifications.....	8
3.3.1 Types.....	8
3.3.1.1 Low-pass Filter	8
3.3.1.2 High-pass Filter.....	8
3.3.1.3 Band-pass Filter	8
3.3.1.4 Band-stop or Band-reject Filter	8
3.3.2 Specifying Magnitude Response	9
3.3.2.1 Pass-band Cut-off Frequency F_p	9
3.3.2.2 Pass-band Ripple.....	11
3.3.2.3 Stop-band Ripple	13
3.3.2.4 Stop-band Cut-Off Frequency.....	13
3.3.3 Specifying phase response.....	13
3.4 Describing Discrete Time Signals and Systems	14
3.4.1 Sampling Frequency	14
3.4.2 Discrete Time signals in Time and Frequency domain.....	15
3.4.3 Discrete Time systems in Time and Frequency domain	15

3.4.4	The System Function $H(z)$	15
3.4.5	The Difference Equations and $H(z)$	16
3.5	FIR Filters.....	19
3.5.1	Convolution	19
3.5.2	An elementary form of an FIR filter.....	21
3.5.3	Symmetry in FIR Filters.....	24
3.5.4	Windowing	25
3.5.5	Structures for FIR Filters.....	31
3.5.5.1	Direct-form filter structure.....	31
3.5.5.2	Symmetric direct-form filter structure.....	32
3.5.5.3	Transposed direct-form structure.....	33
3.6	IIR Filters.....	34
3.6.1	IIR filter design process.....	35
3.6.2	Indirect Design Method.....	35
3.6.2.1	Analog Filter Prototypes.....	36
3.6.2.2	Mapping from s to z	37
3.7	FIR versus IIR Filters	38
4	VHDL.....	39
4.1	Introduction	39
4.2	The process of implementing logic circuits – EDA tools	39
4.3	Code Structure-Fundamental VHDL Units.....	41
4.3.1	Generic declarations	42
4.4	VHDL objects.....	43
4.4.1	SIGNALS	43
4.4.2	VARIABLES	43
4.4.3	CONSTANTS.....	44
4.5	Data Types	44
4.5.1	Predefined data types.....	44
4.5.1.1	Basic predefined data types (package standard).....	44
4.5.1.2	Standard-Logic Data Types (Package <code>std_logic_1164</code>).....	45
4.5.1.3	Unsigned and Signed Data Types	46
4.5.2	User-Defined Data Types	47
4.5.2.1	Arrays	47
4.5.2.2	Records	47
4.5.2.3	Port Array.....	47
4.5.2.4	Data Conversion	48
4.6	Operators and Attributes.....	48

4.6.1	Assignment operators	48
4.6.2	Logical Operators	49
4.6.3	Arithmetic operators	49
4.6.4	Relational operators	50
4.6.5	Shift operators	50
4.6.6	Concatenation operators	51
4.7	Attributes	51
4.8	Concurrent and Sequential codes.....	52
4.8.1	Concurrent Code.....	52
4.8.1.1	SELECT statement	52
4.8.1.2	WHEN...ELSE statement.....	52
4.8.1.3	GENERATE statement	53
4.8.2	Sequential code.....	53
4.8.2.1	PROCESS	53
4.8.2.2	IF statement	54
4.8.2.3	WAIT statement	54
4.8.2.4	CASE statement	55
4.8.2.5	LOOP statement.....	55
4.9	Components	55
4.10	Simulation (VHDL Test benches).....	57
4.10.1	Simulation Types	57
5	Generating the Filter Coefficients	60
5.1	Introduction	60
5.2	FDATool	60
5.3	Designing the FIR filter.....	61
6	Implementation in Hardware Description Language.....	65
6.1	Diagram of the filter	65
6.2	Internal parts of the implemented filter	65
6.3	Entities Hierarchy Diagram	68
7	Simulation - Results	71
7.1	Simulation.....	71
7.2	Results	71
8	Conclusions	89
8.1	Summary.....	89
8.2	Evaluation.....	89
8.3	Future Work	93
	REFERENCES	94

Appendix A VHDL Code.....	95
Appendix B ModelSim Tutorial	104
B.1 Introduction	104
B.2 Creating a Project	104
B.3 Compiling Project Files	109
B.4 Running a Functional Simulation	110
Appendix C Quartus II Tutorial.....	113
C.1 Introduction	113
C.2 Creating a Project	114
C.3 Synthesizing the Design	115
C.4 Simulating the Circuit.....	118
Appendix D Read and Store files for simulation.....	121
D.1 Read and Store files for simulation	121
Appendix E Design and Test Files for functional simulation.....	124
E.1 Design File and Test File.....	124
Appendix F Matlab.....	127
F.1 Simulation using Matlab.....	127
F.2 Convert coefficients to integer.....	128

List of Figures

Figure 3.1	Basic idea of a filter	4
Figure 3.2	The impulse signal $\delta(t)$	5
Figure 3.3	A system in the time domain	6
Figure 3.4	Frequency and phase response of a low-pass filter	6
Figure 3.5	The s – plane	7
Figure 3.6	Magnitude response of low-pass filter	9
Figure 3.7	Magnitude response of High-pass Filter	10
Figure 3.8	Magnitude response of Band-pass Filter	10
Figure 3.9	Magnitude response of Band-stop Filter	11
Figure 3.10	Low-pass analog filter parameters	12
Figure 3.11	Low-pass digital filter parameters	12
Figure 3.12	Linear phase response of a low-pass filter	13
Figure 3.13	Analog and Discrete time signal.....	14
Figure 3.14	The z – plane	16
Figure 3.15	IIR filter structure.....	18
Figure 3.16	FIR structure.....	19
Figure 3.17	Relationship between impulse response and coefficients of an FIR filter	20
Figure 3.18	A four tapped filter	22
Figure 3.19	The output $y(n)$	22
Figure 3.20	Magnitude response using equal coefficients.....	23
Figure 3.21	Magnitude response using individual weights of coefficients.....	23
Figure 3.22	Improvement of sharpness of the roll-off in the transition area	24
Figure 3.23	Symmetry in FIR Filters.....	24
Figure 3.24	Impulse response of an ideal low-pass filter	26
Figure 3.25	Truncated ideal low-pass filter impulse response.....	27
Figure 3.26	Magnitude response using 21 coefficients	27
Figure 3.27	Magnitude response using twice as many coefficients as in figure 3.26.....	28
Figure 3.28	Smoothing the truncated impulse response.	28
Figure 3.29	Magnitude response using windowed coefficients.....	29
Figure 3.30	Time domain and magnitude response of some common windows	31
Figure 3.31	A 4-tap filter implemented in direct-form	32
Figure 3.32	A 4-tap FIR filter using the symmetric direct-form.....	33
Figure 3.33	A 4-tap FIR Filter using the transposed direct-form.....	33
Figure 3.34	Direct I form of an IIR filter.....	34

Figure 3.35	Regions of stable poles in the s-plane and z-plane.....	36
Figure 4.1	Design Flow	40
Figure 4.2	Fundamental sections of a VHDL code.....	41
Figure 4.3	Type I test bench.....	57
Figure 4.4	Type II test bench	57
Figure 4.5	Type III test bench.....	58
Figure 4.6	Type IV test bench (full bench).....	58
Figure 5.1	Graphical User Interface	61
Figure 5.2	Frequency response of the filter	63
Figure 5.3	Impulse response of the filter	63
Figure 5.4	Linear phase in the pass-band area.....	64
Figure 6.1	Diagram of the FIR	65
Figure 5.2	Internal parts of the implemented filter.....	67
Figure 6.3	Entities Hierarchy Diagram.....	68
Figure 7.1	Procedure for simulation	71
Figure 7.2	Input signal 500 Hz	72
Figure 7.3	Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim).....	72
Figure 7.4	Input signal 1KHz	73
Figure 7.5	Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim).....	73
Figure 7.6	Input signal 1KHz	74
Figure 7.7	Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim).....	74
Figure 7.8	Input signal 10 KHz	75
Figure 7.9	Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim).....	75
Figure 7.10	Input signal 15 KHz	76
Figure 7.11	Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim).....	76
Figure 7.12	Input signal 20 KHz	77
Figure 7.13	Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim).....	77
Figure 7.14	Input signal 25 KHz	78
Figure 7.15	Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim).....	78
Figure 7.16	Input signal 30 KHz	79

Figure 7.17 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)	79
Figure 7.18 Input signal 36 KHz	80
Figure 7.19 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)	80
Figure 7.20 Input signal 42.6 KHz	81
Figure 7.21 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)	81
Figure 7.22 Input signal 50 KHz	82
Figure 7.23 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)	82
Figure 7.24 Input signal 60 KHz	83
Figure 7.25 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)	83
Figure 7.26 Input signal 65 KHz	84
Figure 7.27 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)	84
Figure 7.28 Input signal 75.2 KHz	85
Figure 7.29 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)	85
Figure 7.30 Input signal 100 KHz	86
Figure 7.31 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)	86
Figure 7.32 Diagram of the filter frequency response	88
Figure 8.1 A 4-tap filter implemented in direct-form	90
Figure 8.2 parallel accumulator	92
Figure 8.1 Linear phase filter with reduced number of multipliers	93
Figure B.1	104
Figure B.2	105
Figure B.3	105
Figure B.4	105
Figure B.5	106
Figure B.6	106
Figure B.7	106
Figure B.8	108
Figure B.9	109
Figure B.10	109

Figure B.11	110
Figure B.12	110
Figure B.13	111
Figure B.14	111
Figure B.15	112
Figure B.16	112
Figure B.17	112
Figure C.1	113
Figure C.2	114
Figure C.3	114
Figure C.4	115
Figure C.5	115
Figure C.6	116
Figure C.7	117
Figure C.8	117
Figure C.9	117
Figure C.10	118
Figure C.11	118
Figure C.12	119
Figure C.13	119
Figure C.14	120
Figure C.15	120
Figure C.16	120

List of Tables

Table 3.1	Ideal impulse responses for common filter types.....	25
Table 3.2	Key properties of windows.....	30
Table 4.1	Main type conversion options.....	49
Table 4.2	Shift operators and their functions.....	50
Table 4.3	Pre-defined data attributes.....	51
Table 4.4	Signal attributes.....	51
Table 5.1	Filter coefficients as integers.....	64
Table 7.1	Frequency response using the Matlab software Model and the hardware model, simulated in Modelsim.....	87
Table 8.1	Recourse Requirements.....	89

1 Introduction

1.1 Background and Context

Hardware accelerators are specific-processor systems designed to implement computationally intensive operations at high processing rates. Instead of executing sequential commands they perform parallel data processing, unfolding the sequential loop into a pipelined data path.

In this thesis a digital filter was designed by implementing the basic Multiply-Accumulate operation as a hardware component on-a-chip. The convolution process was parallelized using shift registers. The filter coefficients were hardwired in on-chip ROM memory, in the form of a look-up table. Also the specifications of the filter were set. A low-pass filter with $F_C = 35$ KHz (-3dB) was designed (see details in paragraph 5.3).

First, a software model of the specific filter was implemented using the Matlab programming environment in order to verify the results which were produced by the hardware.

Second, the hardware system was designed in VHDL hardware description language, using software tools like Quartus II by Altera for synthesis and Modelsim by Mentor Graphics for simulation.

1.2 Scope and Objectives

The objective of this work is to establish best design rules for the hardware implementation of a FIR filter and create a detailed description of the filter design, using the Hardware Description Language VHDL. The design is assessed in terms of the required hardware resources and the maximum operating frequency. Also, it is assessed in terms of accuracy, by comparing the results of a software implementation of the same filter, using double precision variables and floating point computations.

As a methodology, bottom-up hierarchical design is used, in which the various components are first described in VHDL and then they are instantiated in order to produce the top design entity of the filter. Such filter components that need description are the shift register for the creation of the convolution window, the ROM stage which keeps the filter coefficients, the computationally demanding parallel multiplication stage, and finally the accumulation and normalization stage, where the output sample is computed. Specifications for the necessary data types were defined. Alternative implementations of specific stages were tested, using different bit-widths and accumulation techniques, as a means to establish best design methodology.

Test-benches were designed for the simulation of the filter. For this purpose we use VHDL test code that reads signals stored in text files. The simulation is performed with ModelSim and the output is stored in a new text file. Then, the results can be compared with the expected waveforms.

1.3 Achievements

The design is assessed using as a measure the resource requirements for the overall implementation of the filter in a medium FPGA device, like the Cyclone II EP2C35. In addition, the timing analysis results provide a measure of the performance of the design, when it is implemented in the above FPGA device. We find that a filter with 101 coefficients can reproduce adequately the original double precision filter specifications using just 4677 logic elements including logic registers. This represents 14% of the resources of a low cost FPGA device. Also, it can achieve a maximum clock frequency of 50 MHz.

1.4 Overview of Dissertation

In chapter 3 there is a presentation of digital filters. In particular, the Finite Impulse Response Filters and the window design method are described. In Chapter 4, the Hardware Description Language VHDL is presented. At the end of the chapter, the simulation with VHDL test benches is described, which is one of the most crucial steps in VHDL circuit design. In Chapter 5, the procedure followed for the generation of the filter coefficients is analyzed, using the Graphical User Interface Filter Design and Analysis Tool (FDATool). In Chapter 6, the description of the hierarchical design of the filter is given. In Chapter 7, the results that were derived by the programming environment Matlab and the simulator Modelsim are presented and compared. Finally, in Chapter 8, the achievements and the conclusions of this thesis are presented. Also, a possible future development of this work is proposed.

2 State-of-The-Art

Signal processing has been used to transform or manipulate analog or digital signals. Digital signal processing has found many applications like filtering and convolution, Fourier transform, audio processing, image processing, information systems and i.e.

Digital signal processing is a mature technology and has replaced analog signal processing systems in many applications. Compared to analog systems, digital signal processing systems has several advantages, like the insensitivity to change in temperature, aging or component tolerance. Two events have accelerated DSP development [4]. The first was the disclosure by Cooley and Tuckey of an efficient algorithm to compute the discrete Fourier Transform and the second was the introduction of programmable digital signal processor (PDSP).

Programmable digital signal processors have enjoyed tremendous success for the last two decades [4]. They are based on a reduced instruction set computer (RISC) with an architecture consisting of at least one fast array multiplier with an extended word width accumulator. The PDSP advantage comes from the fact that most signal processing algorithms are multiply and accumulate intensive.

Now, Field-programmable gate arrays (FPGAs) are on the verge of revolutionizing digital signal processing in the manner that PDSP did two decades ago [4]. Many digital signal processing algorithms such as FFTs, FIR or IIR filters which built by PDSPs are now replaced by FPGAs. Many high-bandwidth signal processing applications such as wireless, multimedia or satellite transmission can be found today, and FPGA technology can provide more bandwidth through multiple MAC on one chip. Also compared to PDSPs, FPGA design exploits parallelism e.g., implementing multiple multiply-accumulate calls efficiency, e.g., zero product-terms are removed, and pipelining, i.e., each LE has a register, therefore pipelining requires no additional resources [4]. It is assumed that in the future PDSPs will dominate applications that require complicated algorithms, while FPGAs will dominate more frond-end applications like FIR filters, CORDIC algorithms, or FFTs.

Another trend in the DSP hardware design is the migration from graphical design entries to hardware description language. It has been found that code reuse is much higher with hardware description language than with graphical design entries. Two hardware description languages are popular today. The US west cost and Asia prefer Verilog, while US east cost and Europe prefer VHDL. For DSP using FPGA both languages seem to be well suited, although VHDL examples are a little easier to read because of the supported signed arithmetic and multiply/divide operations in the IEEE VHDL 1076-1987 and 1076-1993 standards [4].

3 Digital Filters

3.1 Introduction

Digital filtering is one of the most powerful tools of DSP. Digital filters are capable of performance specifications that it would be difficult or impossible to achieve with an analog filter. Moreover the characteristics of a digital filter can be easily changed under software control. Therefore they are widely used in communications and modems [6].

Designing a digital filter requires a procedure that has the same fundamental elements as that for analog filters. First the desired filter response is characterized and the filter parameters are then calculated [6]. That means that characteristics such as the amplitude and phase response are derived in the same way. The fundamental difference between analog and digital filter is instead of calculating capacitor, resistor and inductor values in analog filter, coefficient values are calculated for a digital filter [6]. As a result in digital filter, numbers replace the resistors and the capacitors components of the analog filter. This numbers are stored in a memory as filter coefficients and are used with the input signal (sampled data values) from the ADC (Analog to Digital Converter) to perform the filter calculations.

The function of filtering is the same for analog or digital signal. In signal processing, the function of a filter is to remove unwanted parts of the signal such as random noise, or to extract useful parts of the signal, such as the components lying within a certain frequency range. Figure 3.1 illustrates the basic idea of a filter.



Figure 3.1 Basic idea of a filter

3.2 Describing Systems

In paragraph 3.1 we mentioned that filters are a system that takes an input signal, make some modifications, and produces an output signal [3]. Usually a system has one or more inputs and one or more outputs. The filter as a system has one input signal and one output signal. We know that if we want to describe a signal we use the time domain and the frequency domain. Just like signals, there are the time domain and the frequency domain to describe systems. But a system is not a signal, a system modifies signals. Therefore the only we can do is to describe what a system does to signals which is called system response. Hence we must describe the Time Domain Response and the Frequency Domain Response of a system.

3.2.1 Time Domain Response

The most common way to characterize a system in the time domain is its impulse response. This means that we observe the response of the system (output) to an impulse input.

An impulse is like a spike. The signal is at 0, the impulse width is very small and the amplitude is very large. Mathematically the width is infinitesimally small and the height infinite (see figure 3.2). However if we integrate the signal which is the area under the pulse we will get a total area of 1. The notation for the impulse signal is $\delta(t)$ and the notation $h(t)$ is used for the impulse response of a system (see Figure 3.3) The most important aspect of the impulse is that excites a system equally at all frequencies [3].

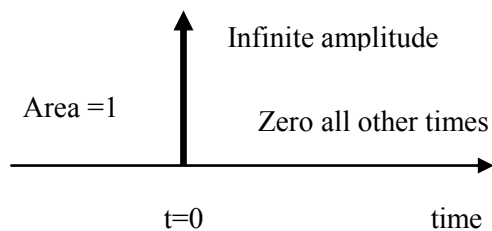


Figure 3.2 The impulse signal $\delta(t)$

3.2.2 Frequency Domain Response $H(j\Omega)$

For a system (Linear, Time Invariant) $H(j\Omega)$ describes both the gain and phase shift that a signal of frequency Ω experiences when going through a system [3]. The frequency Ω is expressed in radians/second. $H(j\Omega)$ is also called the *transfer function* of the system. As we can observe from the j , the transfer function is complex which means that if we evaluate $H(j\Omega)$ at some frequencies the result is a complex number which shows the magnitude $|H(j\Omega)|$ and the phase $\angle H(j\Omega)$. The magnitude is the ratio of the magnitude of the output to the magnitude of an input at that frequency and the phase is the difference in phases between the output and input [3]. Example:

We assume that $H(j\Omega) = 142120/142120 + 533j\Omega - \Omega^2$ is the transfer function of a system. If the input of the system is a sinusoid with magnitude 4 and frequency 60Hz, what is this system output?

First we need to convert from Hertz to radians/sec which is $60 \text{ Hz} = 60 \cdot 2\pi \text{ rad/sec} = 377 \text{ rad/sec}$. This is Ω . For the calculations we use complex arithmetic because there is a j in the denominator. Finally $H(j\Omega) = -0.000008 - 0.70729j$, or in polar form $H(j\Omega) = 0.70729 \angle -1.57$ (angle is in radians). It is convenient to express the angle in terms of π which is $-1.57 \text{ rad} = -\pi/2$. Therefore the output of this system has a magnitude which is $4 \times 0.70729 = 2.83$ and a phase which is different by -90° from the input signal.

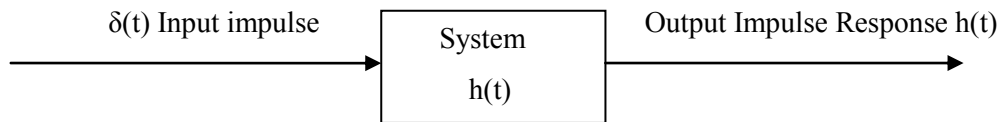


Figure 3.3 A system in the time domain

Now if we do the above calculations for different values of frequency and plot the frequency and phase response we will understand the overall behavior of the system. Figure 3.4 [3]) shows the frequency and phase response of the system. We can see from the plots that $H(j\Omega)$ describes a system that passes lower frequencies with no attenuation, but start attenuating signals above 200Hz. The phase goes through a transition as well approaching $-\pi$. Hence the specific $H(j\Omega)$ describes a low-pass filter.

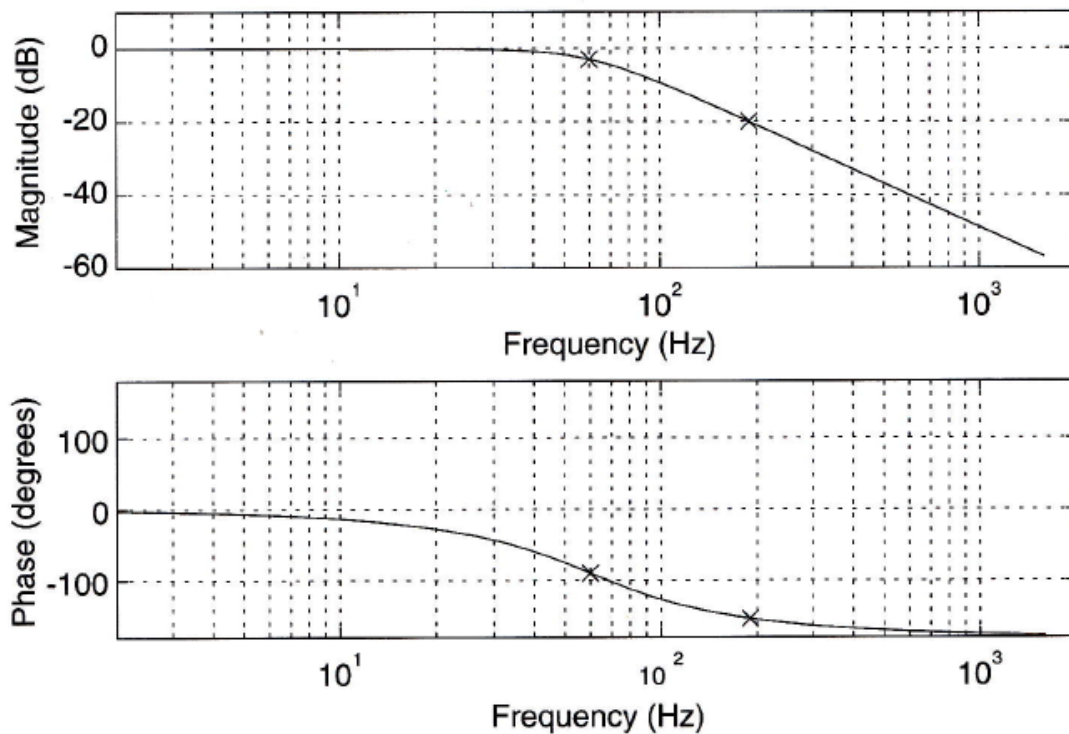


Figure 3.4 Frequency and phase response of a low-pass filter

3.2.3 System Function $H(s)$

$H(j\Omega)$ provides us with the description of the system and plot of the magnitude response. However, we use a more general description which is called the system function $H(s)$. But which is the difference between $H(j\Omega)$ and $H(s)$? The first function uses the simple frequency Ω while the latter uses the complex frequency s . Signals with complex frequency are sinusoids with exponential growth or decay of the amplitude [3]. Therefore we multiply the sinusoid by the exponential term $e^{\sigma t}$ to get $x(t) = A e^{\sigma t} \cos(\Omega t + \theta)$ which is an example of complex frequency. The σ term affects how quickly and in what direction amplitude growth occurs [3].

$H(s)$ can bundle both frequency and the exponential factor into a complex number which is called s . Therefore the real part is the exponential growth factor σ while the imaginary part is the frequency Ω . Therefore any value of $s = \sigma + j\Omega$. Complex frequency is used to represent signals and systems in the s -plane. The horizontal axis of the s -plane is the real part of s the σ , and the vertical axis of the s -plane is the imaginary part of s the $j\Omega$. The s -plane is very important because we can describe a system by locating the poles and zeros of the system.

3.2.3.1 Poles and Zeros

$H(s)$ is always of the form

$$H(s) = \frac{\text{Numerator}}{\text{Denominator}}$$

There are two interesting sets of values for s in this equation. First there are values of s such that the numerator of $H(s)$ is equal to 0. These values of s are the roots of the numerator of $H(s)$ which are the zeros of $H(s)$. Poles are those values of s such that the denominator equals to zero.

Stable filters may have zeros located anywhere in the s -plane. Zeros located on the $j\Omega$ axis completely block signals at that frequency.

Stable filters may have only poles in the half of the s -plane with $\sigma < 0$. Poles on the $j\Omega$ axis correspond to constant oscillation. Poles with $\sigma > 0$ lead to exponentially increasing outputs and are associated with unstable filters. Figure 3.5 shows poles and zeros in the s -plane.

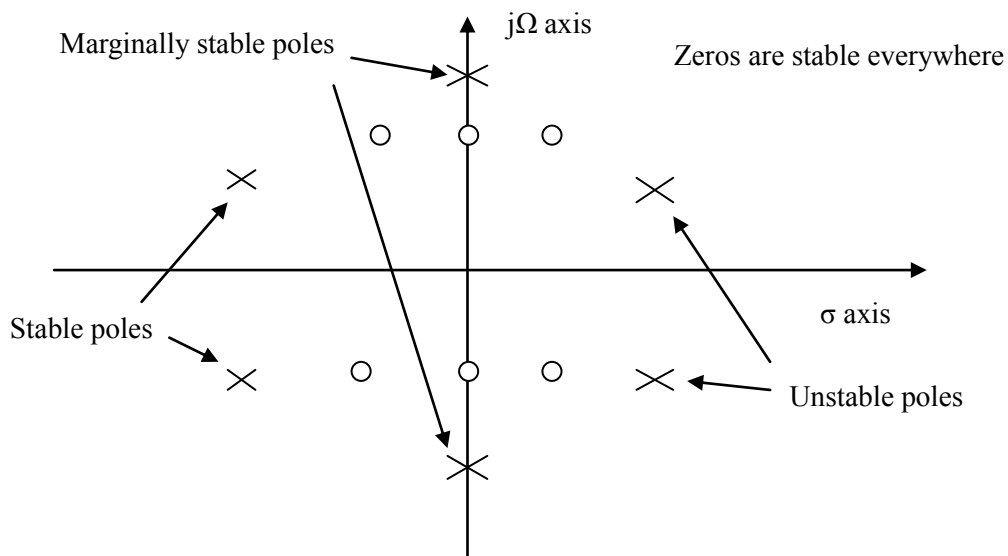


Figure 3.5 The s – plane

Also the order of a filter corresponds to the maximum number of poles or zeros in a filter whether analog or digital. By adding poles and zeros in a filter we can get smoother pass-band,

lower stop-band gain or steeper transition. In an analog filter we can increase the number of op-amp or the number of capacitors in order to increase the poles and zeros. In a digital filter higher order filter requires more coefficients which mean more multiplications and more memory. The trade off is between how good the filter characteristics are and how much processing time such a filter requires [3]. So filter design is the ability of using a minimum filter order to meet filter design criteria.

3.3 Filter specifications

3.3.1 Types

One way of organizing filters is by their frequency magnitude characteristics. The types of filters are low-pass, high-pass, band-pass and band-stop. [3].

3.3.1.1 Low-pass Filter

The following diagrams show in figures 3.6 – 3.12 are taken from citation [3].

Figure 3.6 shows the magnitude response of a low-pass filter. Frequencies from zero to a specified frequency which called “cut-off frequency” or -3dB frequency are passed with relatively equal magnitude, while higher frequencies are greatly attenuated. Low-pass filter are generally used when the signal of interest is in the range of DC (0Hz) to some frequency, but where other higher frequency signals (like noise) are present [3].

3.3.1.2 High-pass Filter

Figure 3.7 shows a high-pass filter magnitude response. This type of filter has ideally infinite attenuation at DC. In reality we must mention that there is a limit on the upper frequency that can be passed. For example the sampling rate of a DSP hardware will set the upper frequency limit in digital filter implementation. For high-pass filters the cut-off frequency is the lower limit of the pass-band.

3.3.1.3 Band-pass Filter

Figure 3.8 shows a band-pass filter magnitude response. As we can see there are two cut-off frequencies, one on the low side and one on the high side. We can get this response by cascading a low-pass and a high-pass filter but it is common to design a single filter with this response.

3.3.1.4 Band-stop or Band-reject Filter

Figure 3.9 shows a band-stop filter magnitude response. Observe that the band-stop filter is the opposite of the band-pass filter. In a band-stop filter a specific range of frequencies is attenu-

ated. The common application of this filter is to reject a specific single frequency rather than a range of frequencies.

3.3.2 Specifying Magnitude Response

Figure 3.10 shows the magnitude response of a low-pass filter with their important characteristics. Also figure 3.11 shows a low-pass filter magnitude response which used to describe digital low-pass filter. The difference between the two magnitude responses is how the allowed variation in the pass-band is described. The first shows a peak to peak variation while the second shows a peak deviation.

The gain of the magnitude frequency response is normalized to 1 (0dB) for a particular frequency depending on the filter type. Also for specifying the magnitude response of a filter we will describe the magnitude response of a low-pass filter.

3.3.2.1 Pass-band Cut-off Frequency F_p

The *pass-band cut-off frequency* F_p marks the end of the pass-band. Frequencies lower than the cut-off frequency is in the pass-band, and frequencies higher than this are in the transition

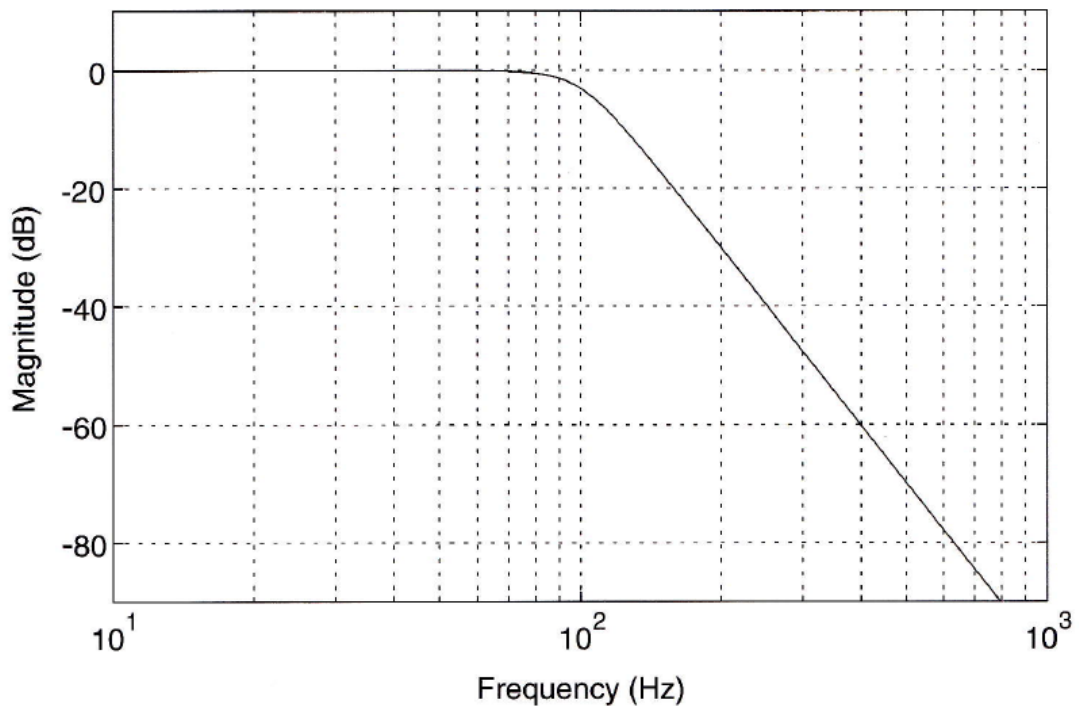


Figure 3.6 Magnitude response of low-pass filter

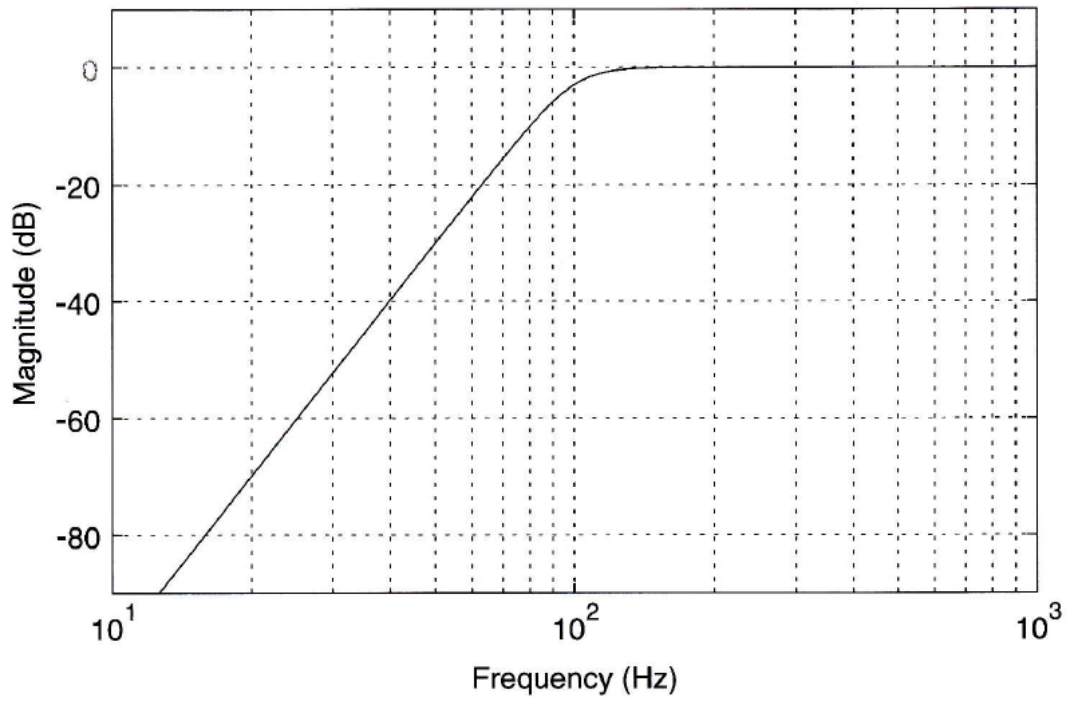


Figure 3.7 Magnitude response of High-pass Filter

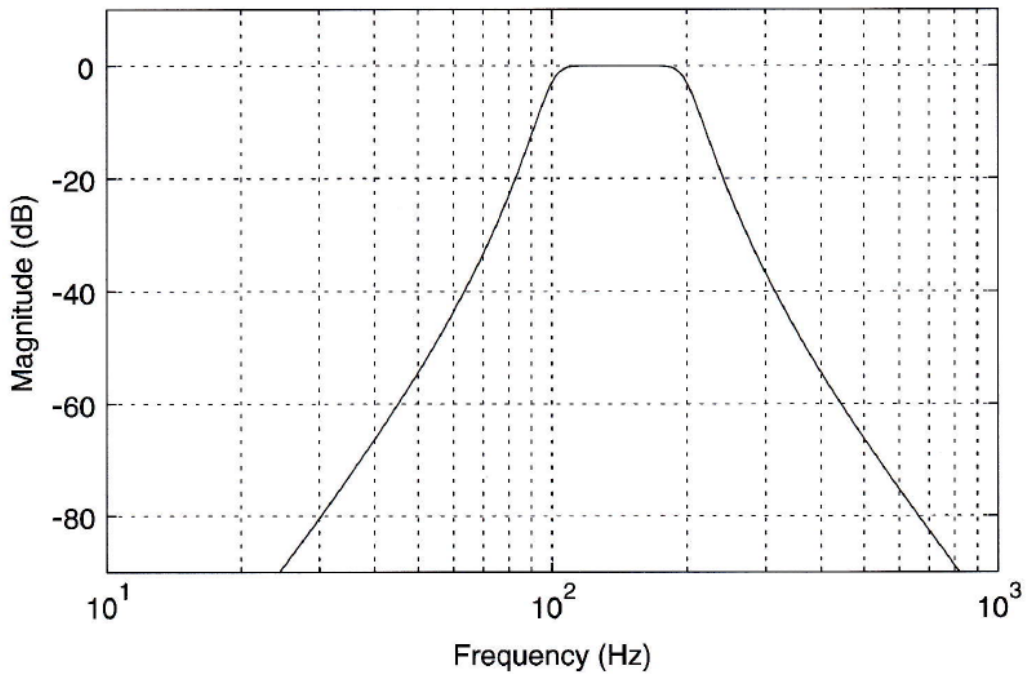


Figure 3.8 Magnitude response of Band-pass Filter

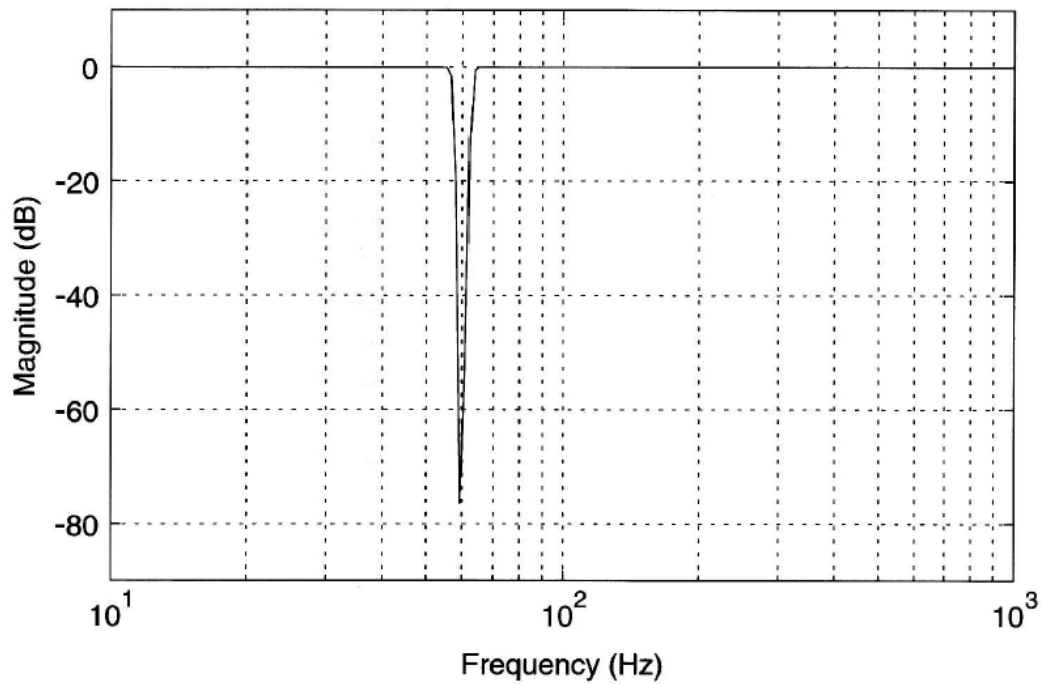


Figure 3.9 Magnitude response of Band-stop Filter

band. Note that the cut-off frequency F_C or -3dB point is in the transition band as shown in figure 3.10.

3.3.2.2 Pass-band Ripple

Pass-band ripple is a measure of the allowed variation in magnitude response in the pass-band of the filter [3]. Is often specified in terms of δ_p also known as the *pass-band deviation*. As we mentioned in paragraph 3.3.2 there are two different ways of define ripple. Figure 3.10 shows the first which is used for analog filter and figure 3.11 shows the second which is used in digital filter. The first method specifies the maximum deviation measured from a gain of one and is associated with analog filter design [3]. The second method measures the deviation from the ideal pass-band magnitude to the minima and maxima.

The units as shown in figure 3.10 may be in decibels (A_p) or linear deviation (δ_p). The relationship between A_p and δ_p depends on the style (analog or digital). The relationship is shown in equations 3.1 and 3.2 for analog and equations 3.3 and 3.4 for digital.

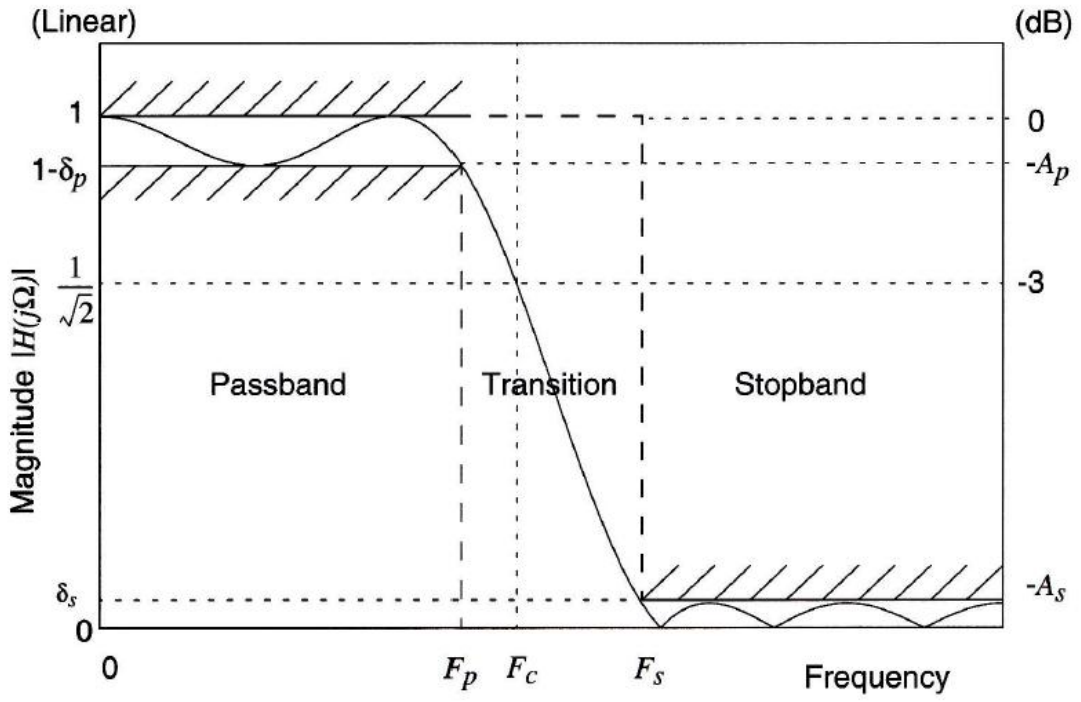


Figure 3.10 Low-pass analog filter parameters

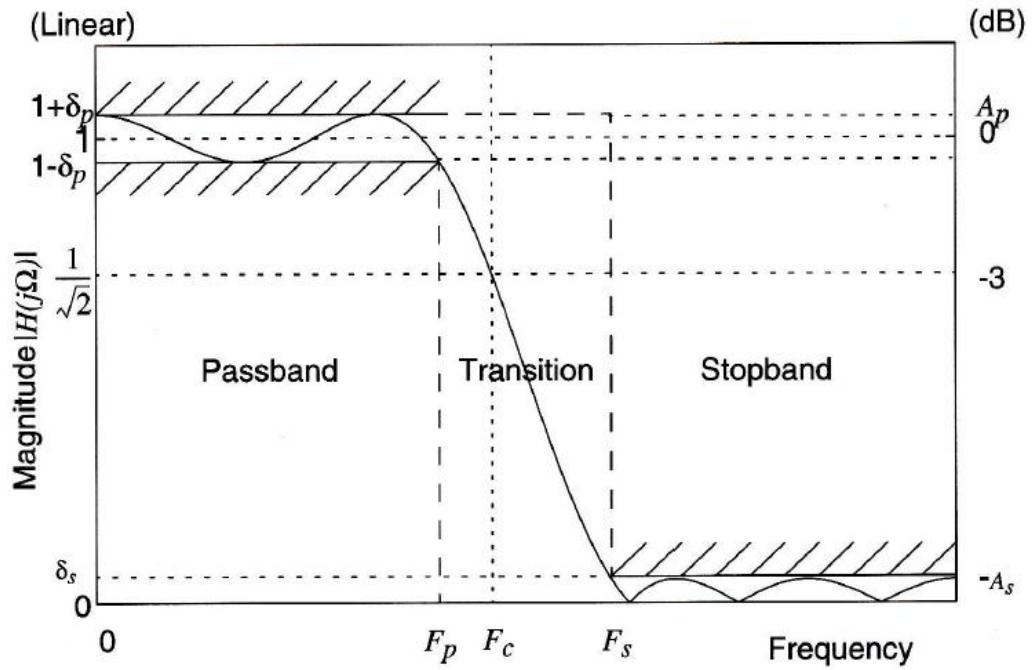


Figure 3.11 Low-pass digital filter parameters

3.3.2.3 Stop-band Ripple

Stop-band ripple describes the maximum gain (or minimum attenuation) we want for a signal above the stop-band cut-off frequency. It can be described as either a linear or decibel value. A positive value in decibels like 25 dB should be read as minimum stop-band attenuation, while a negative value should be maximum gain.

3.3.2.4 Stop-band Cut-Off Frequency

As with the pass-band cut-off frequency all the frequencies above this frequency will meet the stop-band ripple tolerance [3]. *Stop-band cut-off frequency* usually abbreviated as F_{stop} or F_s (it is not the sampling frequency).

3.3.3 Specifying phase response

If every frequency experiences the same time delay the result will be a linear change in phase. The equation 3.5 describes the relationship between phase (θ), time delay (t_d) and frequency (Ω radians/sec).

Figure 3.12 shows a linear phase response for a filter. As we can see in the pass-band the phase wraps a few times (jumps of 2π) while in the stop-band has discontinuities in the phase (jumps of π).

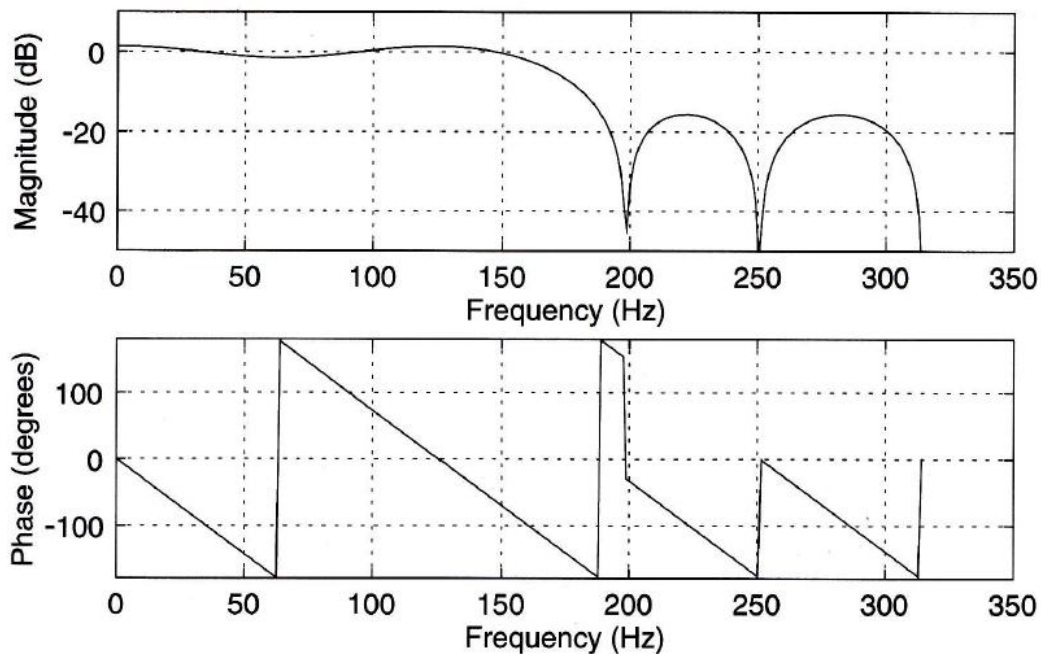


Figure 3.12 Linear phase response of a low-pass filter

Filters with linear phase pass signals without phase distortion. This property is very important in communications, data transmission and other applications where the temporal relationships between different frequency components are important.

3.4 Describing Discrete Time Signals and Systems

As with the continuous time signals there is a frequency domain representation of discrete time signals. Also we can describe discrete time systems in the time domain and frequency domain for which we need the discrete time equivalents of the impulse response and the system function. Before we begin the description of a discrete system in time and frequency domain we need to describe the sampling frequency or sampling rate.

3.4.1 Sampling Frequency

One of the most critical decisions in DSP is choosing the rate at which an analog signal is sampled. This is the *sampling frequency or frequency rate* and usually denoted by F_s (Hz). The resulting signal which is discontinuous in time is a *Discrete Time Signal*. Figure 3.13 shows an analog signal and the DT signal. From the sampling frequency F_s we can calculate the *sampling period* T_s which is $T_s = 1/F_s$.

The choice for sampling frequency is depend on the frequency content of the signal to be sampled. Sampling a signal at sampling signal F_s folds any frequencies higher than $F_s/2$ back into the frequency range of $0 - F_s/2$. This is known as *aliasing*. Moreover signal components with frequencies higher than $F_s/2$ will produce the same samples as a signal with some frequency in the range $0 - F_s/2$ (aliased signal). $F_s/2$ is known as the *Nyquist frequency or folding frequency*. As an example assume a signal at 600Hz and a sampling frequency at 1000 Hz. The signal that produced after the sampling appears to have 400 Hz instead of 600 Hz due to the aliasing.

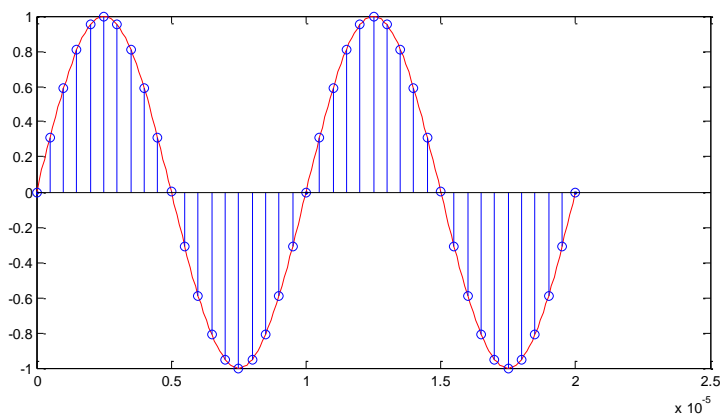


Figure 3.13 Analog and Discrete time signal

3.4.2 Discrete Time signals in Time and Frequency domain

If $x(t)$ is the continuous signal $x(nT_s)$ is a discrete time representation where $n=0,1,2,\dots T_s$ is the sampling period. It is common practice to drop the T_s which is equivalent to setting the sampling period to 1 sec. Therefore the discrete time sequence is denoted as $x(0), x(1), x(2), \dots$

Having set T_s to 1sec all the calculations for discrete time signals and systems can be carried out in terms of *normalized frequency* ω . The relationship between normalized frequency f and the frequency of the signal F is

The spectrum of a discrete time signal is restricted to a limited range (0 to $F_s/2$) due to the aliasing effects. In terms of normalized frequency ω this is the frequency range 0 to π or in terms of f from 0 to 1.

3.4.3 Discrete Time systems in Time and Frequency domain

In paragraph 3.2.1 we discussed that the impulse signal can be used to excite systems at all frequencies. In discrete time systems the impulse response at sample $n=0$ or the sample at time $0T_s$ is equal to 1. All other samples have value 0.

Discrete time signals as continuous time signals can be characterized in the frequency domain. The notation that will use is $H(e^{j\omega})$, the magnitude $|H(e^{j\omega})|$ is the gain a signal of frequency ω will experience and the angle $\angle H(e^{j\omega})$ is the phase shift.

3.4.4 The System Function $H(z)$

The system function of a discrete time system is $H(z)$. In paragraph 3.2.3 we discussed the system function $H(s)$, the s – plane and the mapping of poles and zeros for a continuous time system. Now in the discrete time systems instead of having s – plane we have the z – *plane* for mapping the poles and zeros of the system. We will use $H(z)$ to express digital filters because it easily translates to difference equations. Difference equations are used to write the code to perform a digital filter.

The complex variable z is defined in a similar way as the complex variable s of continuous time systems. The difference is that z is defined in terms of polar coordinates rather than rectangular coordinates. Therefore in z – plane we associate angle with normalized frequency ω and radical distance with growth/decay.

z is defined as:

Where r is the growth/decay factor and ω is the angle (in radians). The r is associated with decay if $r < 1$, with growth if $r > 1$ and no change if $r = 1$.

Now we can define the *system function* $H(z)$ for a discrete time system which relates the output $Y(z)$ to the input $X(z)$ of the system. As with $H(s)$, $H(z)$ can be expressed in terms of the ratio of two polynomials:

Figure 3.14 shows the z – plane. In the z – plane the unit circle is the crucial dividing line between stable and unstable poles. Poles inside the unit circle are associated with stable systems while poles outside the unit circle are associated with unstable systems. Poles right on the line are associated with marginally stable systems. Growth and decay is the distance r from the pole. The unit circle has a radius =1

3.4.5 The Difference Equations and $H(z)$

In paragraph 3.2.3 we determined the $H(s)$ as the Laplace Transform of a system’s impulse response $h(t)$. With Laplace transform the differential equations are transformed in $H(s)$ as polynomials which are far easier to use. We have a similar situation with $H(z)$ which is the Discrete Time system function. $H(z)$ is the z – transform of the discrete time impulse response

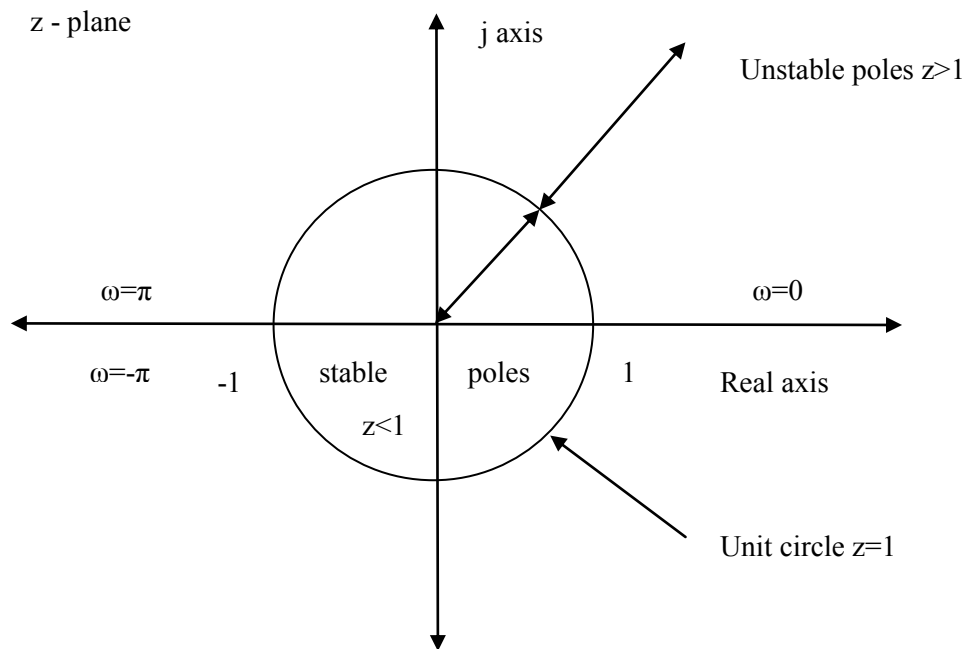


Figure 3.14 The z – plane

$h(n)$ [5]. However discrete time systems are not described in terms of differential equations because of their discrete nature they are described by using *difference equations*. The differ-

ence equations express the current output of a system as a linear combination of current input samples, past input samples and past output samples [3]. The most important think is that we can translate these equations into computer programs which used to implement digital filters.

The difference equations have the form:

Where

$x(0), x(1), x(2), \dots$ is the sequence of input samples

$y(0), y(1), y(2), \dots$ is the sequence of output samples

n is the sample index which has values $0, 1, 2, \dots$

$x(n-k)$ is the k th sample prior to the current sample

$y(n-k)$ is the k th sample prior to the current sample

b_0, b_1, \dots, b_{nz} and a_0, a_1, \dots, a_{np} are constant coefficients. Each coefficient may be positive, negative or zero.

n_z is the number of zeros in the system.

n_p is the number of poles in the system.

We must mention that the prior samples are current samples that have been delayed. For example $x(-2)$ was the current value two sample time ago and has been delayed a total of two sample times [3].

The mapping between difference equations and descriptions in the z – domain is equally straightforward. Delays in the difference equation map to multiplication by powers of z^{-1} in the z -domain. For example $x(n)$ which has no delay is mapped to $X(z)$, but $x(n-1)$ which has a delay of 1 is mapped to $z^{-1}X(z)$. In general $x(n-k)$ is mapped to $z^{-k}X(z)$.

For example we take a difference equation:

Now we replace each delay by an appropriate power of z^{-1} , $y(n)$ with $Y(z)$ and $x(n)$ with $X(z)$ so we have the following equation:

Remember that $H(z)$ relates the output to the input (see equation 3.9) and therefore:

Now we can take the z transform of the equation 1.10 as we did above which gives the $H(z)$

Also from the equation 3.10 we can design the data flow graphically which is shown in figure 3.15 [3].

This type of diagram is used for expressing most digital filter structures and represents the most general class of digital filters.

When at least one of the a_i values in equation 3.10 is nonzero this class of digital filter is called an *infinite impulse response filter* or *IIR filter*. Figure 3.15 shows the structure of an IIR filter. The name is taken by its impulse response which is infinitely long due to the feedback of its output. Hence IIR filters have both poles and zeros and may be unstable if the poles are not properly located (see figure 3.14). Also they are called recursive filters due to the feedback.

Now a second class of digital filter is possible if the a_i values are all equal to zero. That means that there is no feedback and the system has only zeros without poles. This class of filter is called *finite impulse response filter* or *FIR filter*. Figure 3.16 [3] shows the structure of an FIR filter which is an IIR filter without feedback. Also they are called non recursive filters.

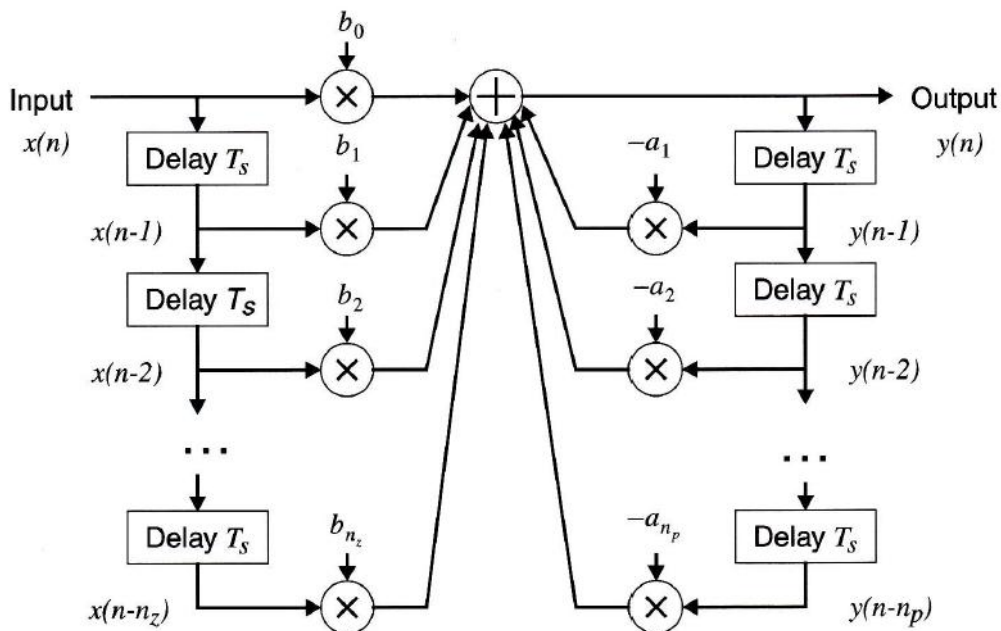


Figure 3.15 IIR filter structure

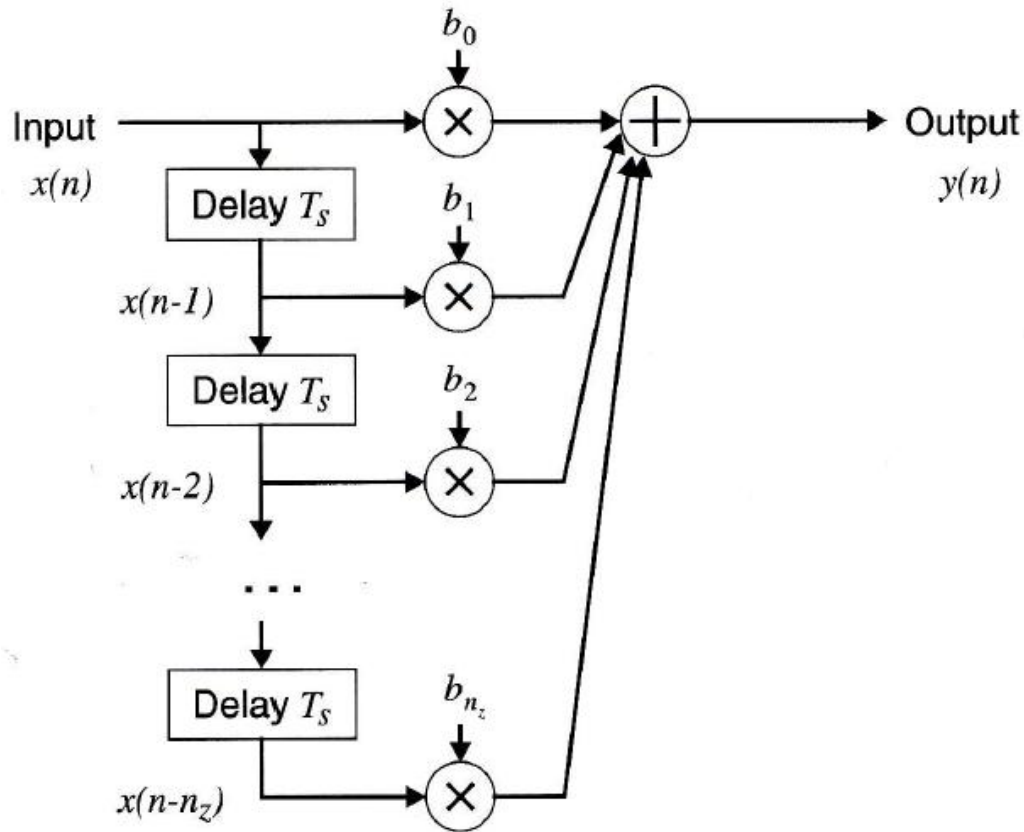


Figure 3.16 FIR structure

3.5 FIR Filters

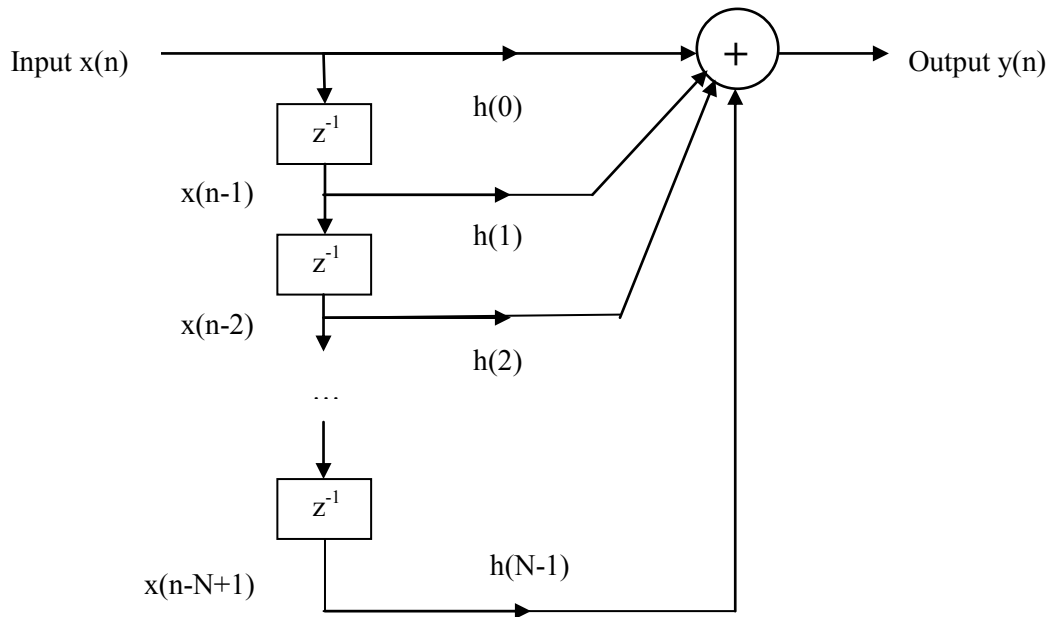
As the terminology suggests these filters refer to the filter's impulse response. By varying the weight of coefficients and the number of filter taps any frequency response characteristic can be realized with an FIR filter. FIR filters can achieve performance levels which are not possible with analog filters [5]. Generally FIR filters require a large number of multiply-accumulates and therefore require fast and efficient DSPs.

3.5.1 Convolution

In order to understand how FIR filters work we need to mention that FIR filters are Linear and Time-Invariant systems LTI.

Linear systems by definition respond to individual frequency components of signals independently of which other frequencies are present. Scaling (multiplying) the input by some factor scales (multiplying) the output by the same factor. Time-invariant systems respond the same to a signal regardless of when it is present.

As we can see from figure 3.17 the impulse response of an FIR filter is just the sequence of coefficient values one right after another. As the "1" of the impulse travels through the filter



Impulse response (i.e., $x(n)=1,0,0, \dots$ for $n=0, 1, 2, \dots$)

	$x(n)$	$x(n-1)$	$x(n-2)$...	$x(n-N-2)$	$x(n-N+1)$	$y(n)$
0	1	0	0	0	0	0	$h(0)$
1	0	1	0	0	0	0	$h(1)$
2	0	0	1	0	0	0	$h(2)$
...	0	0	0	...	0	0	$h(\dots)$
$N-2$	0	0	0	0	1	0	$h(N-2)$
$N-1$	0	0	0	0	0	1	$h(N-1)$
N	0	0	0	0	0	0	0
$N+1$	0	0	0	0	0	0	0

Figure 3.17 Relationship between impulse response and coefficients of an FIR filter

only one coefficient at a time is output. Since FIR filters are linear, if we scale the impulse we input to the system then the output will be a scaled version of the impulse response. Now we can see the input signal (sequence of samples) as just a sequence of scaled impulses each of which produces a scaled and delayed impulse response [3]. Therefore the total output of the filter, should be the impulse response (no delay) scaled by the first sample, added to the impulse response delayed by one sample period and scaled by the second sample and so on. Hence the output of the FIR filter at any given time is the sum of the scaled and delayed impulse responses caused by the current and prior input samples. If $h(k)$ is the impulse response

of the filter and N the number of coefficients which is also the length of the filter we can write the following equation:

Mathematically we get the output $y(n)$ by convolving the impulse response with the input signal.

Also we can write that the convolved output y of the linear system is $y=h*x$ where h is the impulse response of the filter and x is the input signal (sequence of samples) [5].

For linear time-invariant systems it is sometimes more convenient to express (3.12) in the z -domain with

where $H(z)$ is the FIR's transfer function defined in the z -domain by

3.5.2 An elementary form of an FIR filter

An elementary form of an FIR filter is shown in figure 3.18 [6]. This elementary form is identical with figure 3.15 which shows the general structure of an FIR filter. The filter has a four tapped delay line (four coefficients) denoted by N . The input samples $x(n)$ are passed through a series of delays which are registers. The registers are labelled as z^{-1} corresponding to the z -transform representation of a delay element (see paragraph 3.4.5). In the example all the coefficients have the same value which is 0.25. Each sample is multiplied by 0.25 and these results are added to give the output $y(n)$.

Therefore the output $y(n)$ is:



For this filter the output $y(n)$ is:

—

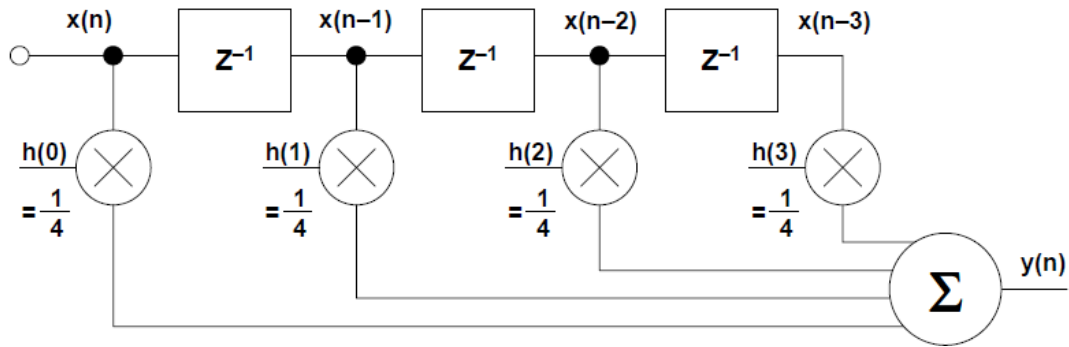


Figure 3.18 A four tapped filter

Since the coefficients are equal there is an easier way to perform the output $y(n)$ as shown in figure 3.19 [6]. The first step is to store the first four samples $x(0)$, $x(1)$, $x(2)$ and $x(3)$ in a register. Then each sample is multiplied by a coefficient, then added together to form the output. Note that the initial outputs $y(0)$, $y(1)$ and $y(2)$ are not valid because all registers are not full until sample $x(3)$ is received. When $x(4)$ sample is received the sample $x(0)$ is subtracted from the result.

$$\begin{aligned}
 y(3) &= 0.25 \left[\begin{array}{c} x(3) + x(2) + x(1) + x(0) \end{array} \right] \\
 y(4) &= 0.25 \left[\begin{array}{c} x(4) + x(3) + x(2) + x(1) \end{array} \right] \\
 y(5) &= 0.25 \left[\begin{array}{c} x(5) + x(4) + x(3) + x(2) \end{array} \right] \\
 y(6) &= 0.25 \left[\begin{array}{c} x(6) + x(5) + x(4) + x(3) \end{array} \right] \\
 y(7) &= 0.25 \left[\begin{array}{c} x(7) + x(6) + x(5) + x(4) \end{array} \right]
 \end{aligned}$$

Figure 3.19 The output $y(n)$

It is possible to improve the performance of this simple FIR filter by properly selecting the individual weights of coefficients rather than giving them equal weight (see figures 3.20 and 3.21). Also the sharpness roll-off of the transition area of the filter can be improved by adding more stages (taps) (see figure 3.22) [6] and the stop-band attenuation characteristics can be improved by properly selecting the filter coefficients. The essence of FIR filter design is the appropriate selection of the filter coefficients and the number of taps to realize the desired transfer function $H(f)$ [6].

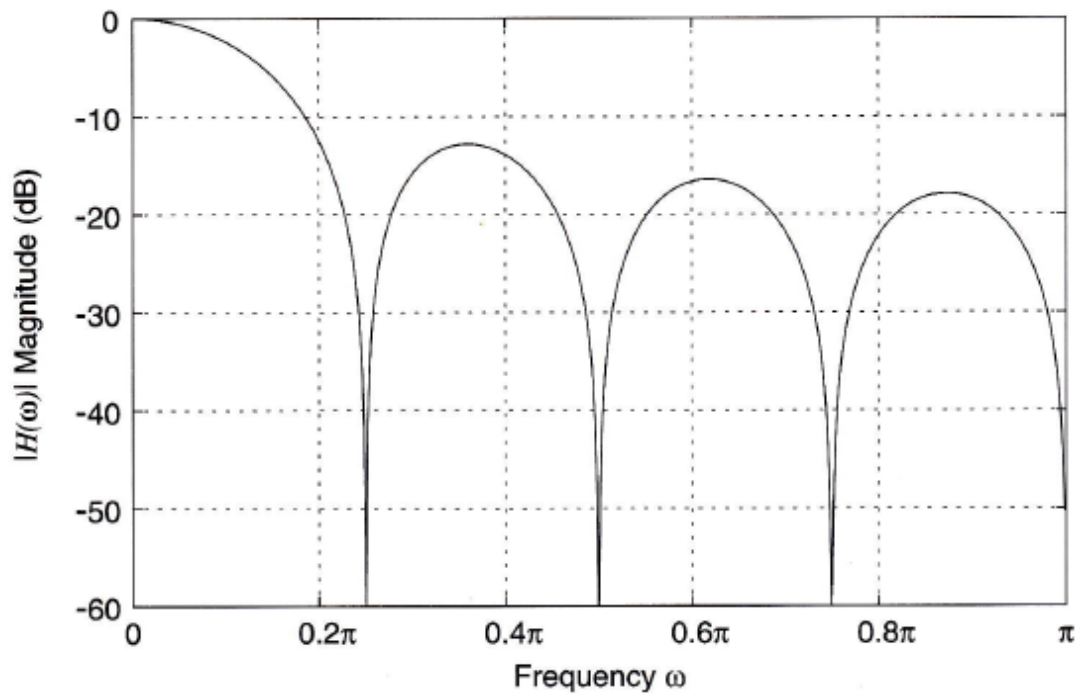


Figure 3.20 Magnitude response using equal coefficients

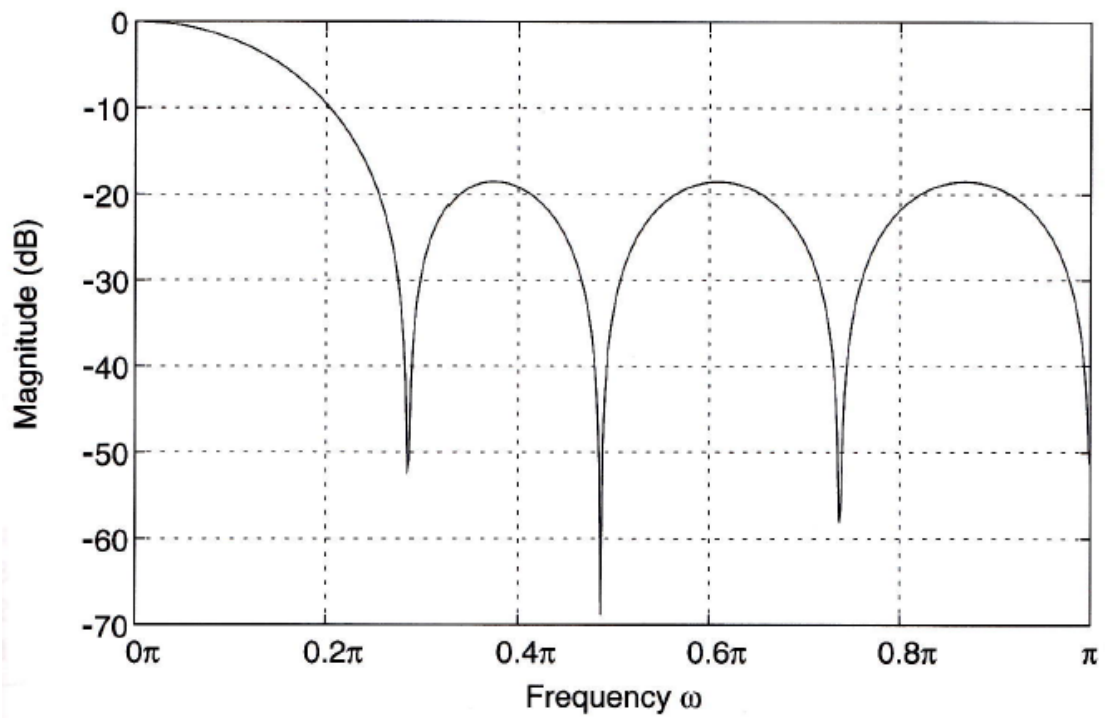


Figure 3.21 Magnitude response using individual weights of coefficients

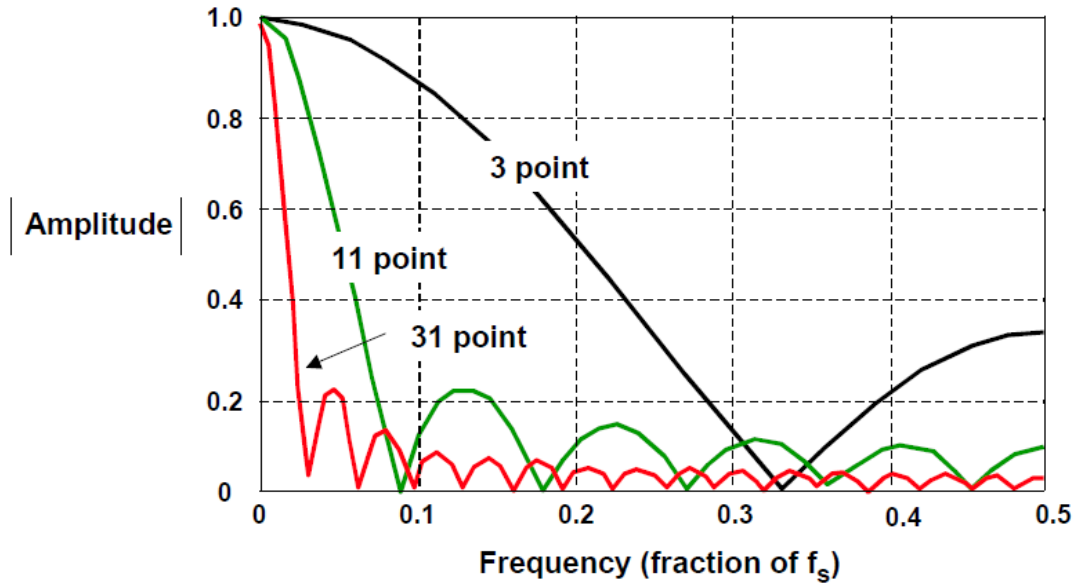


Figure 3.22 Improvement of sharpness of the roll-off in the transition area

3.5.3 Symmetry in FIR Filters

The center of an FIR's impulse response is an important point of symmetry. It is sometimes convenient to define this point as the 0th sample instant. Such filter descriptions are a-causal [4]. For an odd-length FIR the a-causal filter model is given by:

There are four possible linear phase FIR filters which are shown in figure 3.23

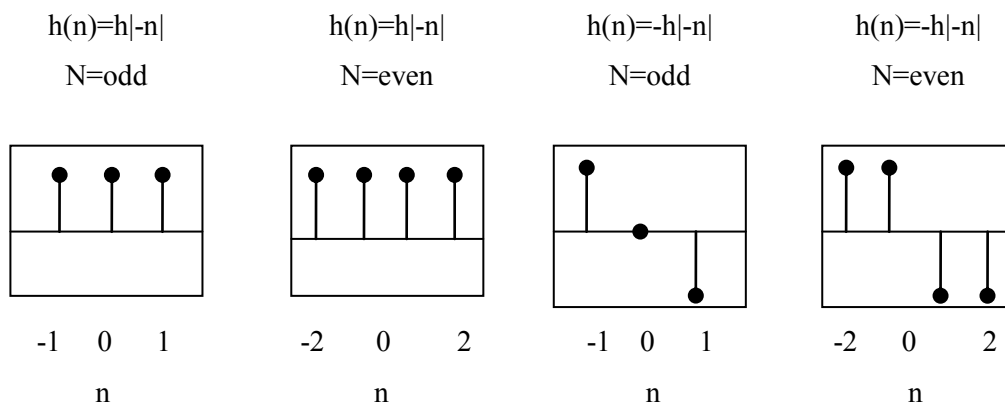


Figure 3.23 Symmetry in FIR Filters

3.5.4 Windowing

The idea is to take the ideal frequency response and calculate its impulse response which is the filter coefficients. But a problem arises since the impulse response for a filter with any sharpness to its frequency response is infinitely long. By definition an FIR filter has a limited number of coefficients hence we need to overcome the previous problem. *Windowing* is one way of getting around this problem, hence the name of this technique [6].

In order to understand the windowing first we need to calculate the impulse response for an ideal filter response. The *inverse discrete time Fourier transform IDTFT* is the useful mathematical transform that we can use for converting between the continuous frequency response and the discrete time impulse response. The mathematical definition is:

—

where $H(\omega)$ is the discrete-time Fourier transform and $h(n)$ is the impulse response. Using the equation 3.16 we can compute $h(n)$ for the ideal low-pass, high-pass, band-pass and band-stop filter types as shown in table 3.1[3].

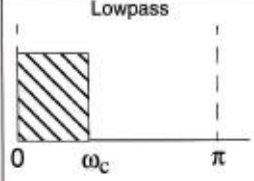
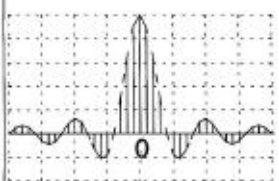
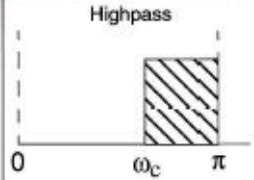
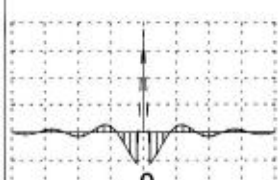
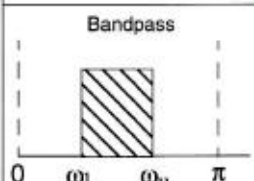
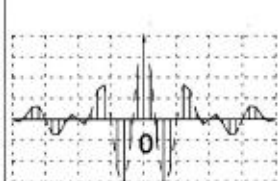
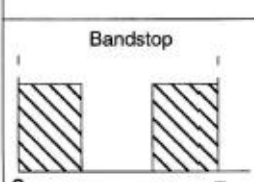
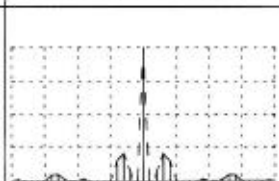
Type	$h_d(n), n \neq 0$	$h_d(0)$	$h_d(n)$
<p>Lowpass</p> 	$\frac{\sin(n\omega_c)}{\pi n}$	$\frac{\omega_c}{\pi}$	
<p>Highpass</p> 	$\frac{-\sin(n\omega_c)}{\pi n}$	$1 - \frac{\omega_c}{\pi}$	
<p>Bandpass</p> 	$\frac{\sin(n\omega_u)}{\pi n} - \frac{\sin(n\omega_l)}{\pi n}$	$\frac{\omega_u}{\pi} - \frac{\omega_l}{\pi}$	
<p>Bandstop</p> 	$\frac{\sin(n\omega_l)}{\pi n} - \frac{\sin(n\omega_u)}{\pi n}$	$1 - \left(\frac{\omega_u}{\pi} - \frac{\omega_l}{\pi}\right)$	

Table 3.1 Ideal impulse responses for common filter types

The infinitely sharp cutoffs of the filters imply a non causal response hence the impulse response extends into positive and negative time. Our filters can deal only with $n \geq 0$. Also due the mathematics we need to evaluate separately the cases for $n=0$.

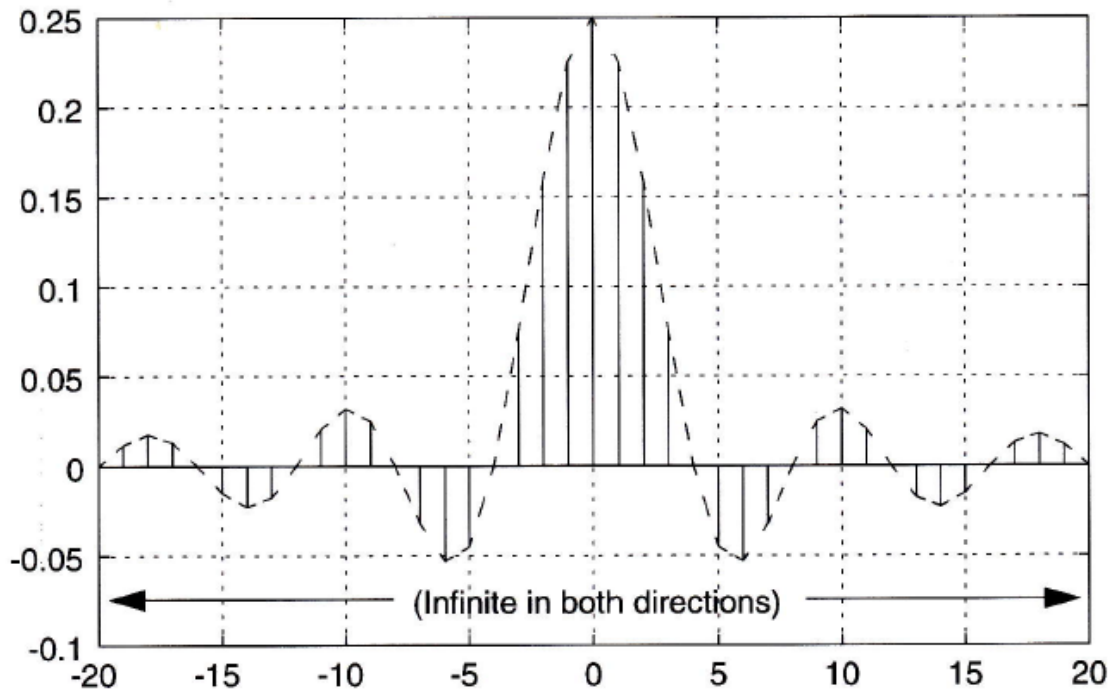


Figure 3.24 Impulse response of an ideal low-pass filter

The following diagrams show in figures 3.24 – 3.30 are taken from citation [3]. Figure 3.24 shows the impulse response of an ideal low-pass filter ($\omega_c = \pi/4$). Because we need a finite number of coefficients we truncate the impulse response after its get fairly small. Figure 3.25 shows the impulse response after truncating it to 21 points. The corresponding frequency response is shown in figure 3.26 [3]. Now we must observe the “ringing” near the transition frequencies. Can we get rid of that “ringing” by taking more points? Figure 3.27 shows the frequency response using twice as many coefficients as in figure 3.26. Comparing the two figures we can see that although the filter coefficients are essentially increased the ringing at the edge still has about the same quantity. The observed ringing is due to the Gibbs phenomenon which relates to the inability of a finite Fourier spectrum to reproduce sharp edges [7]. The effects of ringing can only suppressed with the use of a data “window” that tapers smoothly to zero on both sides [7]. In figure 3.28 we have taken a special window function and smoothly attenuated our 21 coefficients to zero at both ends. In figure 3.29 we can see that data windowing overlay the FIR’s impulse response, resulting in a smoother magnitude frequency response

with an attendant widening of the transition band.

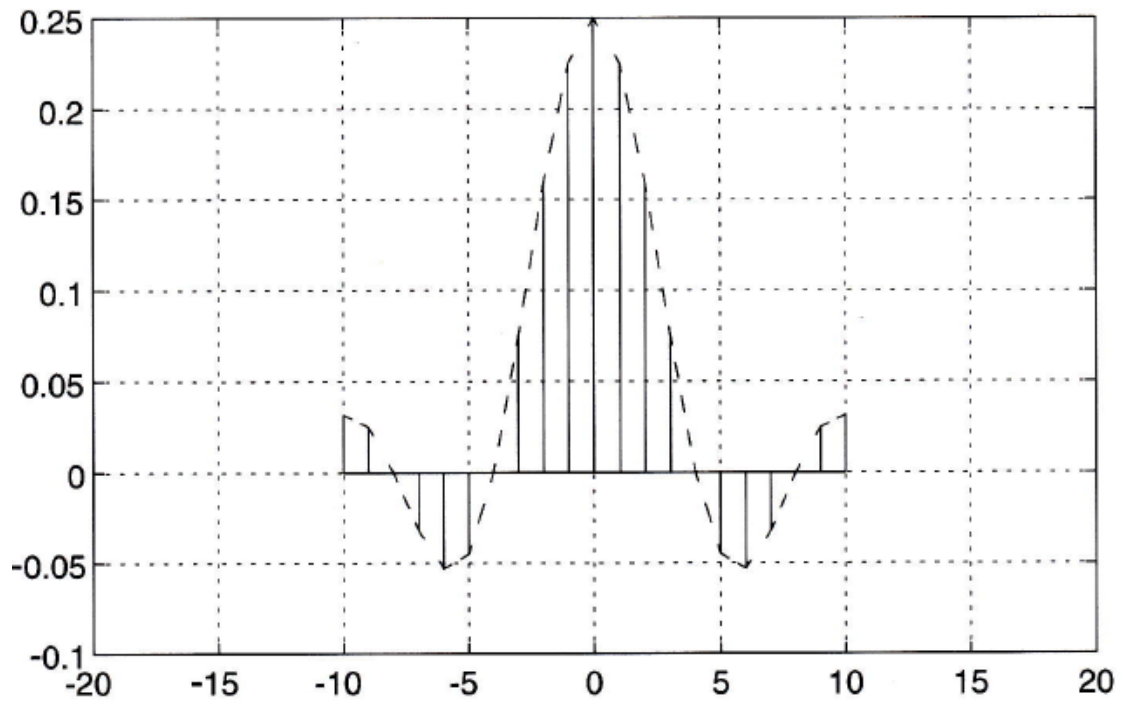


Figure 3.25 Truncated ideal low-pass filter impulse response

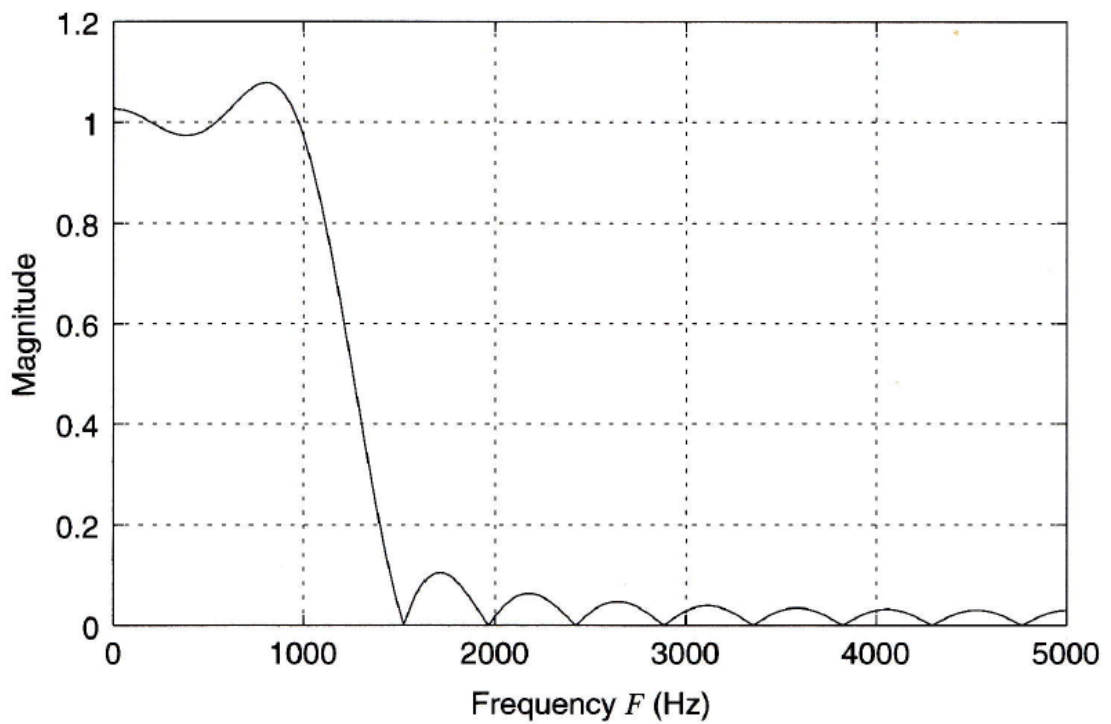


Figure 3.26 Magnitude response using 21 coefficients

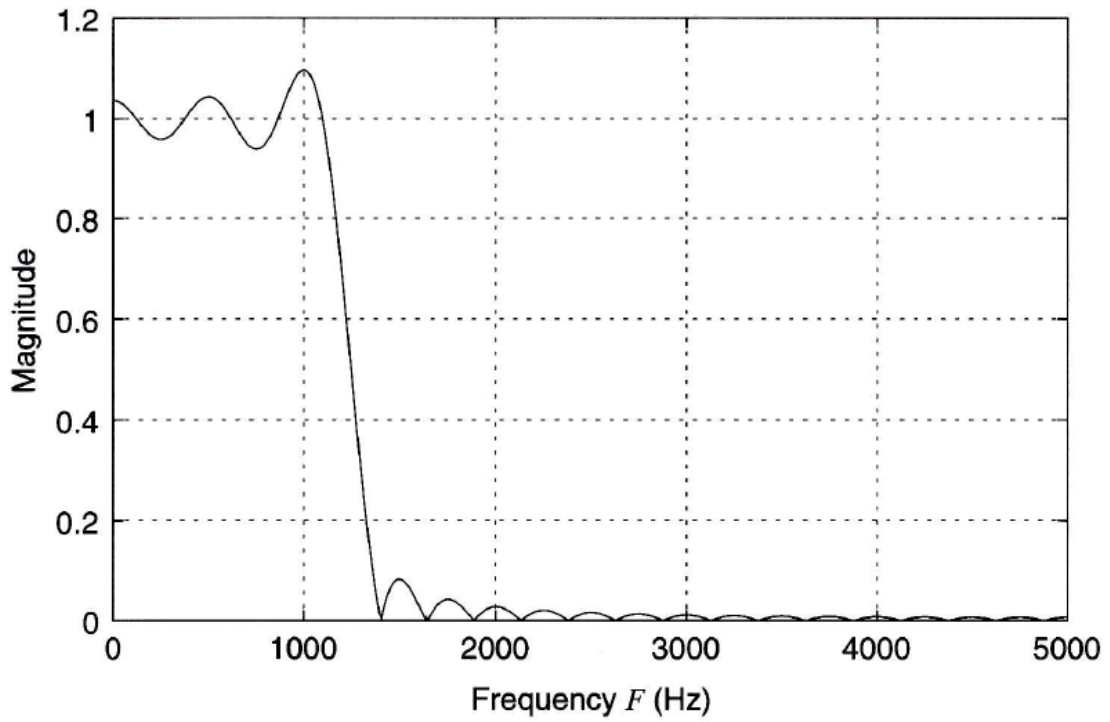


Figure 3.27 Magnitude response using twice as many coefficients as in figure 3.26

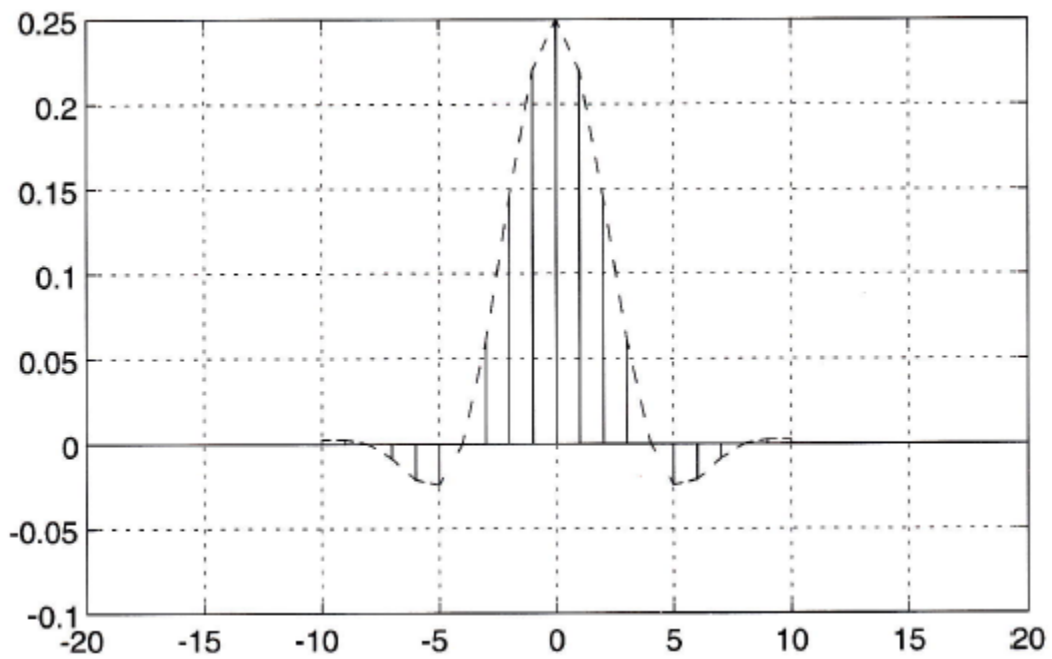


Figure 3.28 Smoothing the truncated impulse response.

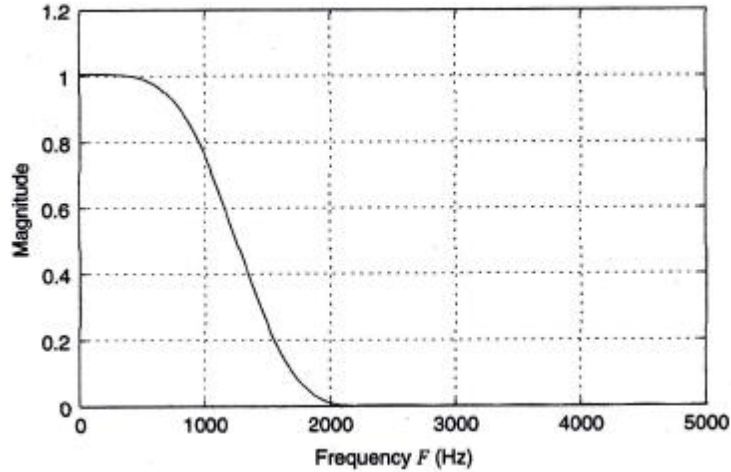


Figure 3.29 Magnitude response using windowed coefficients

Other classic window functions are summarized in table 3.2 [3]. They differ in terms of their ability to make tradeoffs between “ringing” and transition bandwidth extension. The number of recognized and published window functions is large. The most common windows denoted $w(n)$, are:

- Rectangular: $w(n) = 1$. This is effectively “no window”. Offers the sharpest transition in the frequency domain, but at the expense of lessened attenuation in the stop-bands.
- Hanning: $w(n) = 0.5(1 - \cos(2\pi n/N))$. Much wider transition but with a stop-band attenuation of 30 dB.
- Hamming: $w(n) = 0.54 - 0.46\cos(2\pi n/N)$. A bit wider transition area than Hanning, but an additional 10 dB of stop-band attenuation.
- Blackman: $w(n) = 0.42 - 0.5\cos(2\pi n/N) + 0.08\cos(4\pi n/N)$. Continuing the trade-off of transition width for stop-band attenuation it delivers 74 dB attenuation in the stop-band, but with a transition width is six times bigger than the rectangular window.
- Kaiser: $w(n) = \frac{I_0(\beta \sqrt{1 - (n/N)^2})}{I_0(\beta)}$. Based on the first order Bessel function I_0 is special in two respects. It is nearly optimal in terms of the relationship between ringing suppression and transition width and second it can be tuned by β which determines the ringing of the filter. This can be seen from the following equation credited to Kaiser.

Where $A = 20 \log_e r$ is both stop-band attenuation and the pass-band ripple in dB. The Kaiser window length to achieve a desired level of suppression can be estimated:

Window	Passband ripple (dB)	Stopband attenuation (dB)	First side lobe (dB)	Transition width Δf (norm. Hz)
Rectangular	0.7416	21	-13	$0.9/N$
Kaiser, $A=30$ $\beta=2.12$	0.270	30	19	$1.5/N$
Hanning	0.0546	44	-31	$3.1/N$
Kaiser, $A=50$ $\beta=4.55$	0.0274	50	-34	$2.9/N$
Hamming	0.0194	53	-41	$3.3/N$
Kaiser, $A=70$ $\beta=6.76$	0.00275	70	-49	$4.3/N$
Blackman	0.0017	74	-57	$5.5/N$
Kaiser, $A=90$ $\beta=8.96$	0.000275	90	-66	$5.7/N$

Table 3.2 Key properties of windows

Also the time domain and magnitude response of some common windows is shown in figure 3.30.

The key theorem of FIR filter design is that the coefficients $h(n)$ of the FIR filter are the quantized values of the impulse response of the frequency transfer function $H(f)$. Conversely the discrete Fourier transform can be used to transform a digital system's impulse response into its frequency response. Therefore the coefficients of an FIR filter are equivalent to the impulse response of the filter and the output of the filter can be imagined to be the sum of a number of scaled and delayed impulse responses where the scaling is taken from the input signal samples.

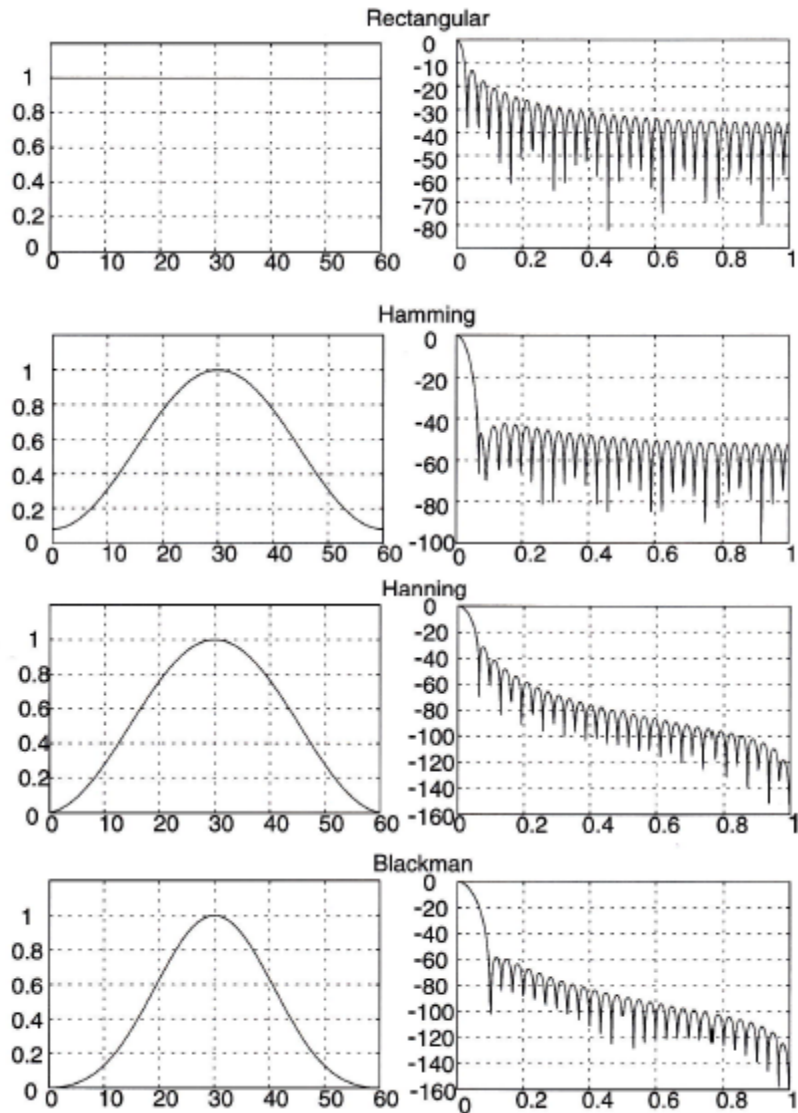


Figure 3.30 Time domain and magnitude response of some common windows

3.5.5 Structures for FIR Filters

Once the filters coefficients have been determined the next step is to decide on the structure of the filter. From a design perspective linear phase FIR filters have symmetric and antisymmetric coefficients. Depending on the target hardware it may be possible to implement a linear phase FIR filter using less multipliers by taking advantage of the symmetry.

3.5.5.1 Direct-form filter structure

The following diagrams show in figures 3.31 – 3.33 was taken from citation [7]. Figure 3.31 shows a 4-tap FIR filter implemented in direct-form. The number of delays is equal to the filter order and the number of coefficients (taps) which is one more than the number of delays determines the filter length [7].

The structure has some regularity in that a sample is read (a delay is a register), multiplied with the filter coefficient, and accumulated to form the output. DSP processors have historically been built with this multiply-accumulate (MAC) instruction in mind [7]. The structure requires a shift of the input data throughout all delays for each sample.

A downside of the direct-form structure is that does not take advantage of the symmetry of the coefficients so the cost is maximum in terms of the number of multipliers required.

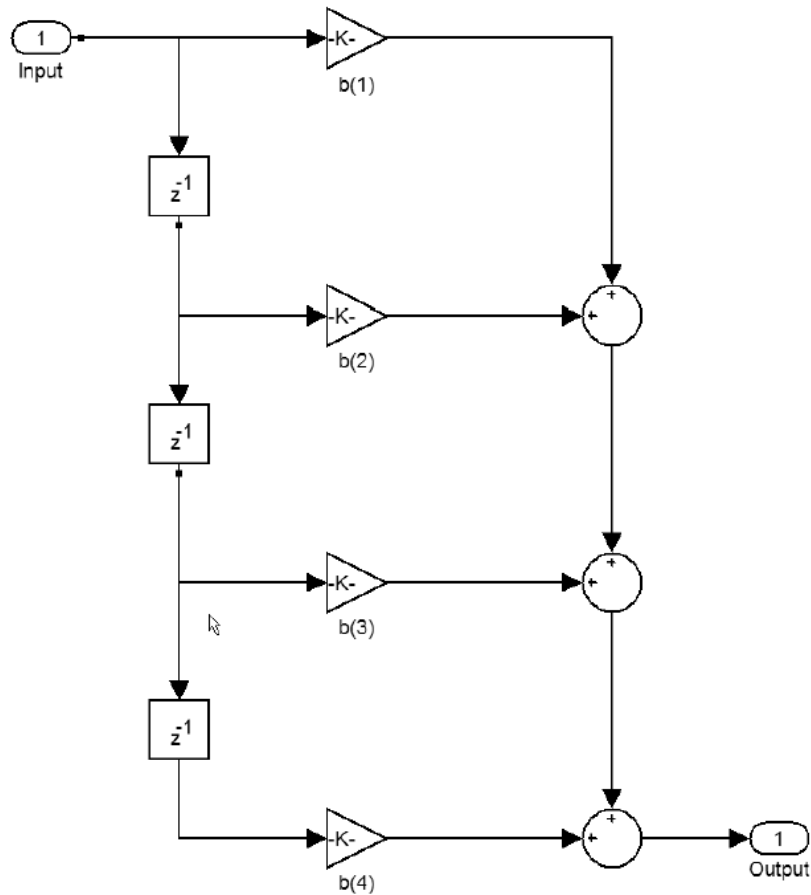


Figure 3.31 A 4-tap filter implemented in direct-form

3.5.5.2 Symmetric direct-form filter structure

The symmetric direct-form filter structure take advantage of symmetry of the coefficients hence the filter can be implemented with the half number of multipliers than the direct-form filter structure. Figure 3.32 shows a 4-tap symmetric FIR filter using the symmetric direct-form. We must observe that even though there are only two multipliers there are still three delays required (same as the direct-form) since the number of delays corresponds to the filter order.

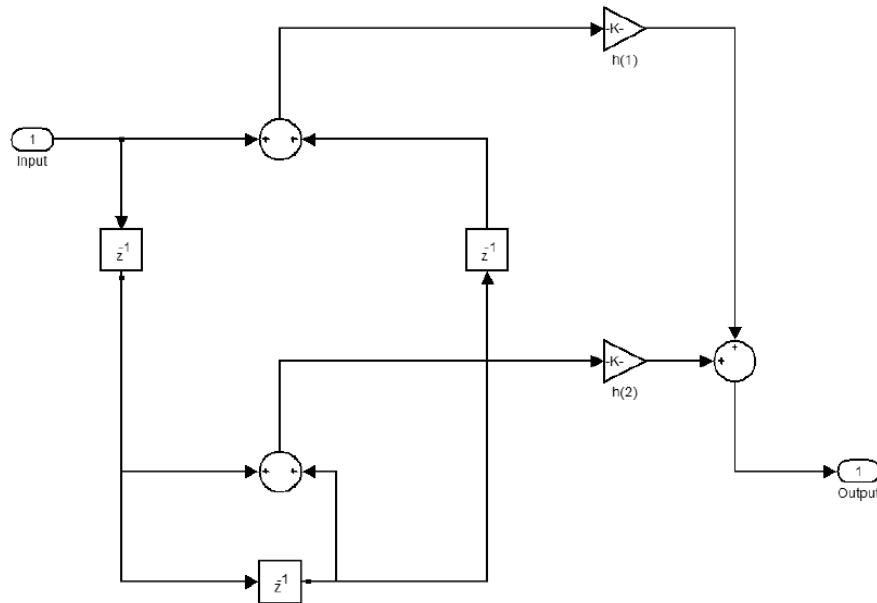


Figure 3.32 A 4-tap FIR filter using the symmetric direct-form

3.5.5.3 Transposed direct-form structure

The direct-form structure has the disadvantage that each adder has to wait for the previous adder to finish before it can compute its result [7]. A solution to this is to use transposed direct-form structure instead. The benefit of this filter is that we do not need an extra shift register for the input samples, and there is no need for an extra pipeline stage for the adder of the products to achieve high throughput. Figure 3.33 shows a 4-tap FIR filter implemented in transposed-form.

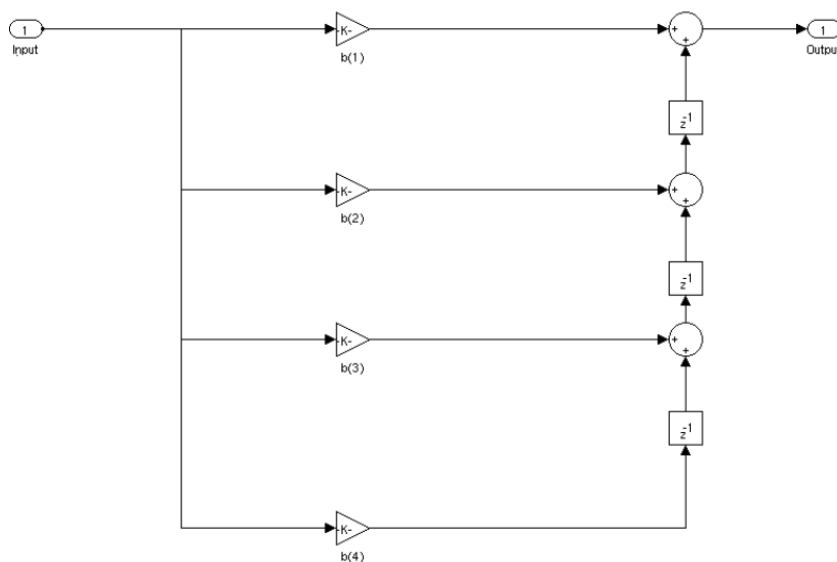


Figure 3.33 A 4-tap FIR Filter using the transposed direct-form

3.6 IIR Filters

In the previous paragraphs we discussed FIR filters, that they have no feedback and can have only zeros. Also FIR filters have no real analog counterpart. On the other hand IIR filters can have both poles and zeros and have traditional analog counterparts (Butterworth, Chebyshev, Elliptic, and Bessel). Therefore they can be analyzed and synthesized using more familiar traditional design techniques.

Infinite Impulse response Filters get their name because their impulse response extends for an infinite period of time. Although they can be implemented with fewer computations than FIR filters, IIR filters do not reach the performance achievable with FIR filters, and do not have linear phase. Figure 3.34 [3] shows a Direct I form of an IIR filter, and writing out the difference equation explicitly shows the feedback of the output into the filter.

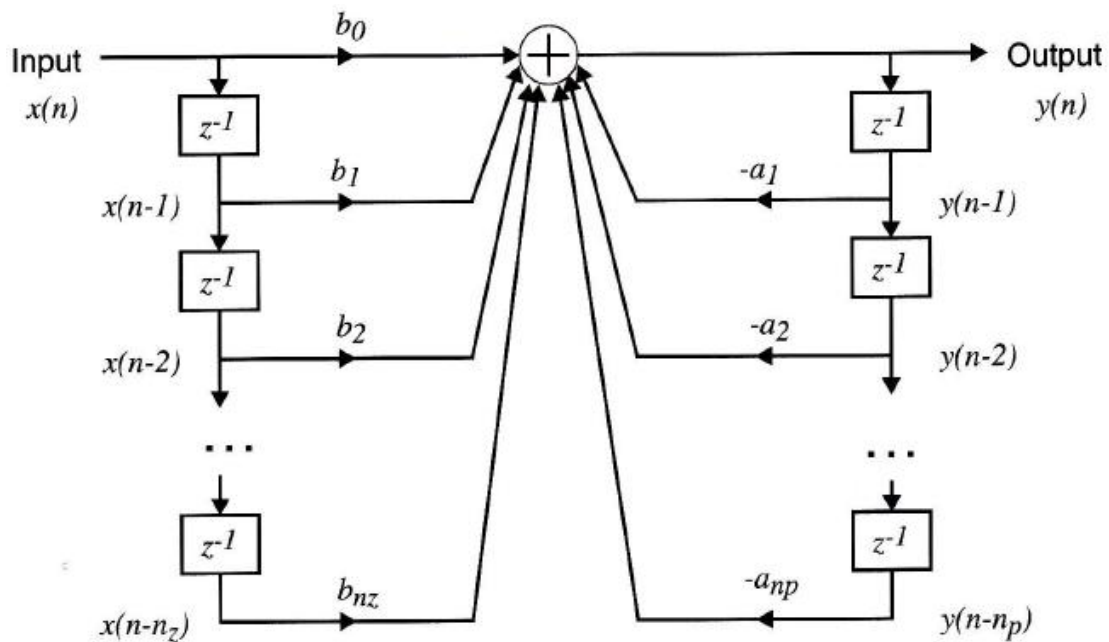


Figure 3.34 Direct I form of an IIR filter

The current value of y is based on a weighted sum of current and past values of the input x and of weighted values of the past n_p values of y . Here n_p is the numbers of poles and n_z is the number of zeros [3]. Adding feedback to a digital filter has some effects. For example, for a given magnitude response it is usually for an IIR filter to be 5-10 times shorter (less coefficients) than the equivalent FIR filter. On the other hand the design and the implementation of IIR filters is much less straightforward than for FIR filters. Also IIR filters can never have ab-

solutely linear phase which is a major problem for systems where linear phase is major selling point. In addition there is no computational advantage achieved when the output of an IIR filter is decimated because each output value must always be calculated [3]. The basic characteristics of an IIR filters are:

- Uses feedback
- Impulse response has a infinite duration
- Potentially unstable
- Non linear phase
- More efficient than FIR filters
- No computational advantage when decimating output

3.6.1 IIR filter design process

The design process for an IIR filter begins with the same information as that required for FIR filters. The desired behavior of the filter in the frequency domain, usually emphasizing the magnitude response rather than the phase response. Also the presence of feedback means that the filter coefficients and the impulse response are no longer related. Therefore we can't use the same techniques which are used for FIR filters. So a popular method design for IIR filter is to first design the analog equivalent filter and then mathematically transform the transfer function $H(s)$ into the z -domain $H(z)$. This method is called *indirect design method*.

3.6.2 Indirect Design Method

Indirect design method produces very efficient IIR filters. As we mentioned above first we must design the analog equivalent filter. The most popular analog filters are the Butterworth, Chebyshev, Elliptical and Bessel and there are many CAD programs available to generate the Laplace transform $H(s)$ for these filters. But the analog filter is only the half of what we need. The key to the indirect method is finding a useful mapping between the analog filters and digital filters, the s to z mapping.

Figure 3.35 shows the regions of stable poles in the s -plane and the z -plane. Remember that the region of the s -plane representing stable poles is the left half plane ($\sigma < 0$) and the region for stable poles in the z -plane is inside the unit circle. The most popular method which is used for mapping from the s -plane to z -plane is the *bilinear z-transform* (BZT).

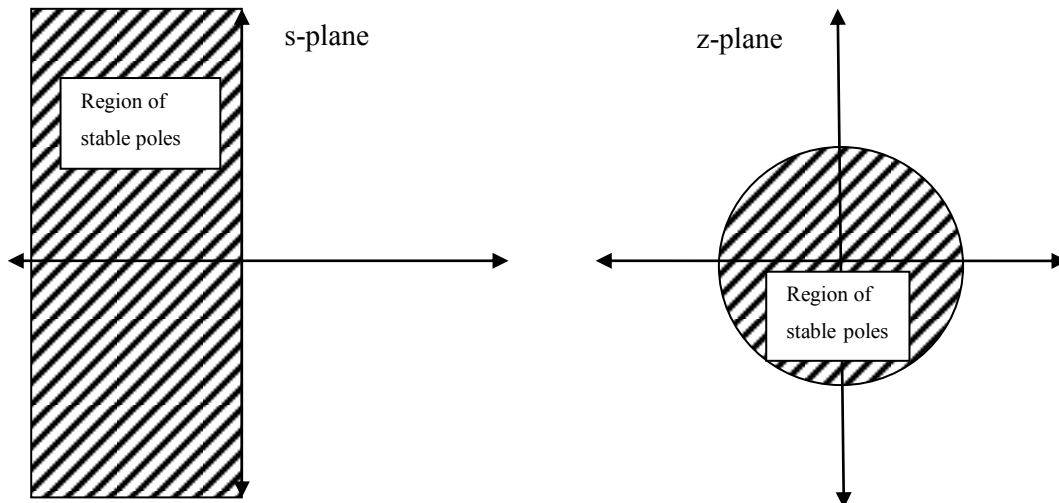


Figure 3.35 Regions of stable poles in the s-plane and z-plane

3.6.2.1 Analog Filter Prototypes

In this paragraph we will summarize the characteristics of the most popular analog filter prototypes, that is analog filters used as models for the eventual digital filters.

The basic characteristics of a Butterworth filter are:

- The magnitude response in the pass-band is maximally-flat (also the Butterworth filter is called maximally-flat filter).
- The magnitude response in both the pass-band and stop-band is monotonic which is that the magnitude response only decreases or stay the same, as frequency increases.
- Has only poles and no zeros. The poles are evenly spaced along a half circle in the left half s-plane.

The basic characteristics of a Chebyshev I filter are:

- It has ripples which are equal sized in the pass-band
- Monotonic stop-band
- Relatively sharp transition between pass-band and stop-band compared with a Butterworth filter with the same order.
- Have only poles and no zeros. Poles in the s-plane lie on an ellipse and the angles of these poles are the same as those for a Butterworth filter.
- Worse phase shift than Butterworth (non linear near pass-band edge frequency).

The basic characteristics of a Chebyshev II filter are:

- It has ripples in the stop-band which are equal sized.
- Monotonic pass-band.
- Relatively sharp transition between pass-band and stop-band.
- Has poles and zeros. The angles of these poles are the same as those for an equivalent order Butterworth filter but are not at the same locations as for Chebyshev I.
- Worse phase shift than Butterworth (non linear near pass-band edge frequency).

The basic characteristics of the Elliptical filter are:

- It has ripples both in the pass-band and the stop-band.
- Has pole and zeros.
- Sharper transition between pass-band and stop-band than the Chebyshev filter for the same order.
- Even worse phase than Chebyshev I and II.

The basic characteristics of the Bessel filter are:

- No ripples in pass-band and stop-band.
- Is an all pole filter.
- Has maximally linear phase for an IIR filter.
- Has the longest transition region than all the others analog filters for the same number of poles or order.

3.6.2.2 Mapping from s to z.

There are three methods used to convert the Laplace transform into the z transform: impulse invariant transformation, bilinear transformation and the matched z-transform. Any method we choose for mapping the s-plane to the z-plane should preserve the crucial elements of the s-plane, including [3]:

- The $j\Omega$ axis of the s-plane must map to the unit circle in the z-plane.
- The left half plane of the s-plane must be mapped to the inside of the unit circle of the z-plane.
- Small negative values of σ should map to locations near (but inside) the unit circle (these poles and zeros have the biggest effects on frequency response).

- Large negative values of σ should map locations near the origin of the z-plane (less effect).

3.7 FIR versus IIR Filters

Typically IIR filters are more efficient than FIR filters because they require less memory and fewer multiply-accumulate are needed. IIR filters can be designed based upon previous experience with analog filter designs.

On the other hand FIR filters require more taps and multiply-accumulates for a given cutoff frequency response but have linear phase characteristics which is necessary in communication systems and they are always stable.

If the processing time is at a premium and a sharp cutoff filter is needed then IIR filters should be chosen. If the number of multiply-accumulates is not prohibitive but a linear phase is a requirement then the FIR filter should be chosen.

4 VHDL

4.1 Introduction

VHDL is a hardware description language. It describes the behavior or structure of an electronic circuit or system from which the physical circuit or system can then be implemented.

The term VHDL is the abbreviation of the words VHSIC Hardware Description Language where VHSIC means Very High Speed Integrated Circuits. The language was initiated and funded by the U.S Department of Defense in the 1980s. The first version was VHDL 87, later upgraded by VHDL 93, then VHDL 2002, and finally VHDL 2008 [1]. It was the first hardware description language which was standardized by IEEE, first through the 1076 standard and later from an additional standard the 1164. The main applications of the language include the description and synthesis of digital circuits in CPLD chips (Complex Programmable Logic Device), FPGA chips (Field Programmable Gate Array) and in the field of ASIC (Application-Specific Integrated Circuit). VHDL is a standard technology/vendor independent language and is therefore portable and reusable.

VHDL is intended for circuit synthesis as well as circuit simulation. Synthesis is the translation of a source code into a hardware structure which implements the desired functionality. Simulation is a testing procedure which ensures that such functionality is indeed achieved by the synthesized circuit.

Contrary to other regular computer programs which are sequential, VHDL statements are inherently concurrent. That's the reason why VHDL is usually referred to us as code rather than a program. Later we will see that statements that are placed inside a PROCESS, FUNCTION and PROCEDURE are executed sequentially.

Another language except from VHDL which has wide use and acceptance is the hardware description language Verilog.

4.2 The process of implementing logic circuits – EDA tools

A simplified view of the design flow that the designer of logical circuits must follow when working with VHDL code is summarized in figure 4.1 [2]. The designer has a set of specifications for which a compliant circuit should be generated. Then we write a VHDL code that fulfills the specifications. The code must be saved in a text file with the extension .vhd and the same name must be used in the entity. For small designs the description consists of one design unit while bigger descriptions are implemented from many design units. The upper hierarchical design unit is called top-level entity. The code next is compiled using a synthesis tool. Several files are generated during the compilation process. The first step of the compilation is called

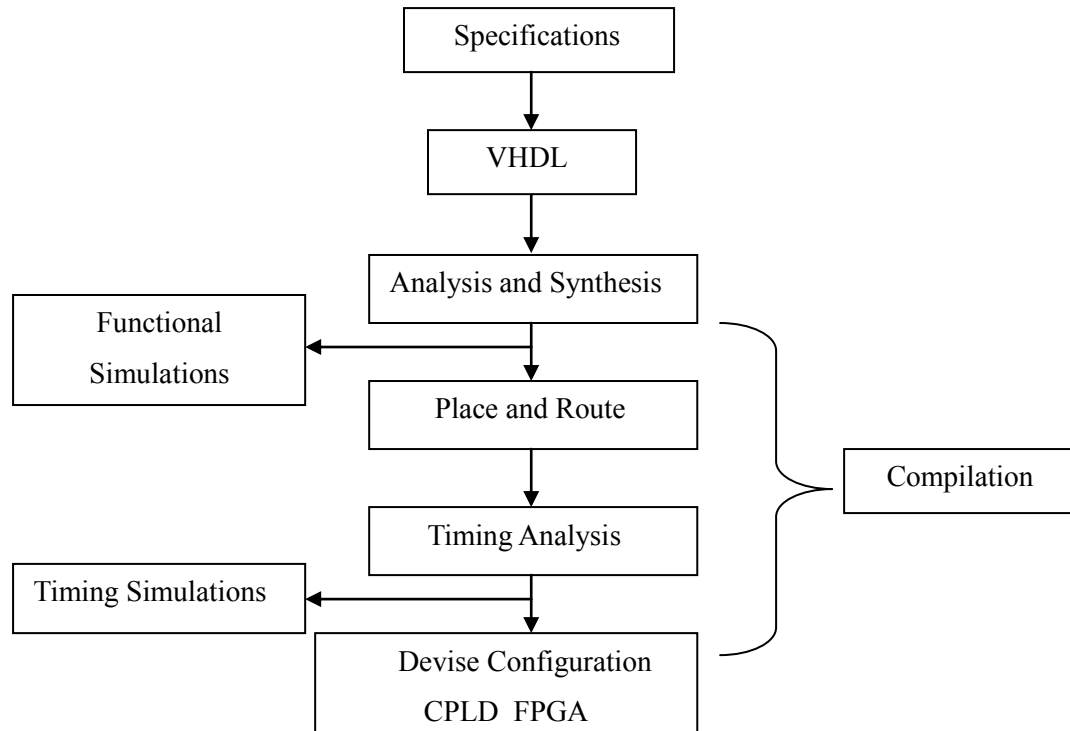


Figure 4.1 Design Flow

analysis. The analyzer is a tool that processes the code for syntax errors and return comments to guide the user to correct the errors. Synthesis is the most important procedure in the design flow. During the synthesis the compiler design the circuit that describes code according to the rules of the device that is intended to shape our plan. That means that the result is different for a CPLD and a FPGA. As a result of synthesis the EDA tool can perform the first stage of functional simulation. During fitting (place and route) each structure inferred by the synthesizer is assigned a specific place inside the device. This positional information is very important because it influences the timing behavior of the circuit. Therefore after the fitting, the timing simulation is possible to follow. With the timing information that generated by the fitting process, the software allows the circuit to be fully simulated. Once the specifications have been met the designer is able to proceed to the final step which is the configuration of the device. During the configuration a programming file for the device is generated. For CPLD and FPGA the design flow is concluded by downloading the programming file from the computer to the target device [2].

There are several EDA (Electronic Design Automation) tools available for synthesis and simulation using VHDL. Some EDA tools are listed below.

- Quartus II from Altera for synthesis and graphical simulation. Appendix 1 provides a briefly description of Quartus II sp2 Web Edition.

- ModelSim from Mentor Graphics for simulation. Appendix 2 provides an introduction to ModelSim 6.4 starter edition for functional simulation.

4.3 Code Structure-Fundamental VHDL Units

The fundamental sections that comprise a standalone piece of regular VHDL code are: *LIBRARY* declarations, *ENTITY* and *ARCHITECTURE*. Figure 4.2 shows the three fundamental sections of a VHDL code.

LIBRARY and *PACKAGE* declarations: Contains a list of libraries and respective packages to be used in the design. For example commonly used libraries are ieeecore, std and work. Placing such pieces inside a library allows the code to be reused and shared by other designs.

The libraries declared with the keyword *LIBRARY* while the package declared with the keyword *USE*. Corresponding syntax is:

```
LIBRARY library_name;
```

```
USE library_name.package_name.all;
```

Note here that in the declarations above the semi-colon (;) indicates the end of a declaration or statement, while a double dash (--) indicates a comment.

ENTITY: Specifies the Input and Output I/O ports (pins) of the circuit. In the part of an entity the name of the entity and all input and output ports (pins) of the circuit must be declared explicitly. A simplified syntax is shown below.

```
ENTITY entity_name IS
```

```
    PORT (
```

```
        port_name: port_mode signal type;
```

```
        port_name: port_mode signal type;
```

```
        ... );
```

```
END entity_name;
```

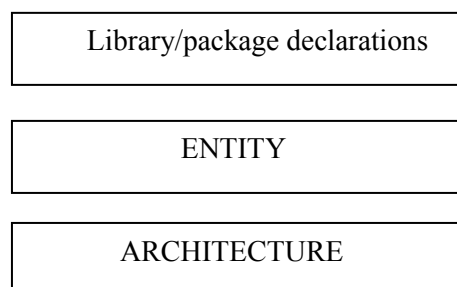


Figure 4.2 Fundamental sections of a VHDL code.

As we can see the code word *ENTITY* is followed by the name of the entity which can be any word (except VHDL reserved words) and the code word *IS*. In the *PORT* field all items are *SIGNALS* that is, wires that go in and out of the circuit. The mode can be:

- *IN*: Signals which are the inputs of the circuit (unidirectional).
- *OUT*: Signals which are the outputs of the circuit (unidirectional)
- *INOUT*: Bidirectional signals. The use of *INOUT* is important when implementing memories which often employ the same data bus for writing and reading
- *BUFFER*: Signals that are sent out but they must also be used internally

Finally the type can be *BIT*, *INTEGER*, *STD_LOGIC* and other data types (see also paragraph 2.5).

The declaration of the entity closes with the word *END* and the name of the entity.

ARCHITECTURE: Contains the proper VHDL code, which describes how the circuit should function, from which a compliant hardware is inferred. The syntax is shown below.

```
ARCHITECTURE architecture_name OF entity_name IS
```

```
    [architecture_declarative_part]
```

```
BEGIN
```

```
    Architecture_statements_part
```

```
END [ARCHITECTURE] [architecture_name];
```

The *architecture_name* can be any word except VHDL reserved words. The *entity_name* is the same name which has already declared in the *ENTITY*. The description of the architecture always begin with the reserved word *BEGIN*. As shown in the syntax above, there are two parts in the architecture. The first part is the declaration where signals, constants, variables are declared except from the signals that had already declared in the *ENTITY* (I/O signals) and the second part is the code that is from *BEGIN* and down. The declaration of the architecture closes with the word *END* and the name of the architecture.

4.3.1 Generic declarations

Generic declarations allow the specification of generic parameters. The VHDL reserved word for generic declarations is *GENERIC* and declared in the *ENTITY* before the *PORT* clause. Is the only declaration which placed before the *PORT*. The purpose of a generic declaration is to parameterize a design that gives to the code more flexibility and reusability [1]. The syntax is shown below.

```
GENERIC (constant_name: constant_type := constant_value;
        constant_name: constant_type := constant_value;
        ... );
```

4.4 VHDL objects

VHDL language manages three data objects that can carry information in a system and convey values in different points. In each object we give a name and we define it with a specific data type. The objects that used in VHDL are: *SIGNALS*, *VARIABLES* and *CONSTANTS* [1].

For a data object, a declaration, a type and a value are necessary.

4.4.1 SIGNALS

SIGNALS are very important while passes values in and out in the circuit as well as its internal units. Eventually a signal represents circuit interconnects (wires). In the ENTITY all ports are signals by default.

Signal declarations can be made in the declarative part of ENTITY, ARCHITECTURE, PACKAGE, BLOCK and GENERATE. Signals can be used in concurrent code and sequential code but they are not allowed signal declarations in sequential code. A simplified syntax for signal declaration is shown below.

```
SIGNAL signal_name: signal_type [range] [:=default_value];
```

To assign a value to a SIGNAL the proper operator is "<=" and for default value is ":="

4.4.2 VARIABLES

VARIABLE represents local information and they are used for the temporary storage of values that created from arithmetic operations. A variable declared and used only in parts of sequential code (inside a PROCESS). Other parts of code that describes sequential commands are the subprograms (FUNCTIONS and PROCEDURES). Hence a variable is used and declared only inside a PROCESS or in a subprogram. Its update is immediate, so the new value can be used to the next line of code. Since the update is immediate multiple assignments to the same variable are fine [1], [2]. A simplified syntax for signal declaration is shown below.

```
VARIABLE variable_name: variable_type [range] [:=default_value];
```

To assign a value to a VARIABLE the proper operator is ":="

4.4.3 CONSTANTS

CONSTANTS are objects whose value cannot be changed. A simplified syntax for constant declaration is shown below.

```
CONSTANT constant_name: constant_type :=constant_value];
```

A *CONSTANT* can be declared in the declarative part of *ENTITY*, *ARCHITECTURE*, *PACKAGE*, *PACKAGE BODY*, *BLOCK*, *GENERATE*, *PROCESS*, *FUNCTION* and *PROCEDURE*. When a constant declared in a package is global because a package can be used by any design file. When declared in an entity it is only global to the architectures that follow the entity. When declared in the architecture it is global only to the architecture [2].

4.5 Data Types

Each data type in VHDL should have a type. The type of a signal determines the values that can receive the signal as and the operations it supports. Each data type supports the use of some operators (see paragraph 1.7) while does not support the use of some others [2].

In VHDL we distinguish the predefined data types and those created by the users (user defined data types). VHDL contains a series of predefined data types specified through the IEEE 1076 and IEEE 1164 standards. Hence their use can be done with a simply reference in the packages which describes such types [2].

4.5.1 Predefined data types

4.5.1.1 Basic predefined data types (package standard)

Such data types definition from the package standard are:

BIT: Signal of this type can take values '0' and '1' (two level logic). The type supports logical and relational operators

BIT_VECTOR: Signal of this type can take values '0' and '1'. This kind of type supports logical, relational, shift and concatenation operators. This type defined as a one dimension array with elements bit. The sequence of the bits "10001110" is a vector with eight elements [2].

INTEGER: A signal of this type carries integer values. The default range of *INTEGER* consists of a 32 bit representation from $-(2^{31} - 1)$ to $(2^{31} - 1)$. It supports arithmetic and comparison operations [2].

NATURAL: A subtype of *INTEGER* which consist of non – negative integers. Has the same dimensionality and supports the same operators as the *INTEGER* type [2].

BOOLEAN: Signal of this type can take value TRUE or FALSE. It is scalar and supports logical and concatenation operators [2].

CHARACTER: Signal of this type can take values from a total of 256-symbol of 8-bit. This type supports only comparison operations. The symbols are from the ISO 8859-1 character set which the first 128 symbols comprising the regular ASCII code. Since each symbol is represented by 8-bits this type falls in the 1D array [2].

TIME: Signal of this type intended only for simulation. It represented by integers which have the same range as INTEGER. It supports arithmetic and concatenation operators [2].

4.5.1.2 Standard-Logic Data Types (Package std_logic_1164)

The types *STD_LOGIC* and *STD_LOGIC_VECTOR* are the industry standards. They are defined in the std_logic_1164 package which introduced along with the VHDL 93. Several features added in VHDL 2008. The declaration of the library ieee and the package std_logic_1164 made in the declarative part of the libraries as:

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;
```

According to the package which determines the *STD_LOGIC* type, a signal can take the values '0' and '1' and in addition might take another six values.

The values that can take a signal of *STD_LOGIC* are:

'0' Forcing low

'1' Forcing high

'X' Forcing unknown

'Z' High impedance

'-' Don't care

'W' Weak unknown

'L' Weak low

'H' Weak high

The basic feature of the *STD_LOGIC* type compared to the *BIT* type is the inclusion of the ('Z') high impedance and ('-') don't care values, that allow the construction of tri-state buffers and a better hardware optimization for lookup tables, respectively [2].

Signals of type `STD_LOGIC_VECTOR` can take the same values as the `STD_LOGIC` type. The difference is that since the type is vector will represent an array of values above and not just one of them ("10001110").

The package `std_logic_1164` defines only logical and concatenation operators for the `STD_LOGIC` and `STD_LOGIC_VECTOR`. However if the package `STD_LOGIC_UNSIGNED` or `STD_LOGIC_SIGNED` are also declared in the code then arithmetic, comparison and shift operations will also be allowed.

In fact the type defined in the `std_logic_1164` package is `STD_ULOGIC` where the 'U' stands for unresolved, of which `STD_LOGIC` is a resolved subtype. The `STD_LOGIC` is a resolved subtype because if more than one source drives a common node the logic level in the node determined by a predefined resolution function [1].

As mentioned in the previous paragraph the data types `STD_LOGIC` and `STD_LOGIC_VECTOR` support only logical and concatenation operators. What about with the description of a circuit which requires unsigned or signed addition? How can we use the arithmetic operators?

In some cases such circuits may be implemented with an integer form, hence the type of the signal declared as `INTEGER`. On the other hand `STD_LOGIC` and `STD_LOGIC_VECTOR` are the industry standards. The feature of using arithmetic operators with signals `STD_LOGIC` and `STD_LOGIC_VECTOR` achieved by using one of the packages `STD_LOGIC_UNSIGNED` and `STD_LOGIC_SIGNED`. Then the compiler considers the signal `STD_LOGIC` and `STD_LOGIC_VECTOR` as unsigned or signed and synthesizes the respectively arithmetic circuits.

4.5.1.3 Unsigned and Signed Data Types

`UNSIGNED` and `SIGNED` data types are defined in two different package of the library `ieee`. The two packages called `numeric_std` and `std_logic_arith`, and they are competitive. Therefore to make use of the `UNSIGNED` or `SIGNED` data types one of the packages above must be declared in the code.

The package `numeric_std` defines logical, arithmetic, comparison and shift operators. The package `std_logic_arith` does not include logical operators but has a wider set of data-conversion functions. Hence the two packages are only partially equivalent. The packages cannot be used together and because `numeric_std` is a standardized package by the IEEE it should be preferred [1].

4.5.2 User-Defined Data Types

Except from the predefined data types, VHDL allows the user to define his own data types (user-defined data types). The new data types can be scalar (like type BIT) or composite (like the predefined type BIT_VECTOR). Another composite type is *ARRAYS* which can be one-dimensional, two-dimensional (2D) or one-dimensional by one-dimensional (1D x 1D). Also a composite type can be a *RECORD* which includes a set of different data types [2].

The most common place for TYPE declarations are in the declarative part of the ARCHITECTURE or a PACKAGE which is more convenient for large designs. The declaration of a user-defined data type is:

```
TYPE type_name IS description of type;
```

4.5.2.1 Arrays

An *ARRAY* can be one-dimensional (1D), two-dimensional (2D) or one-dimensional by one-dimensional (1D x 1D) and consist of the same data type. They can be of higher dimensions but they aren't synthesizable. The pre-defined VHDL data types include only scalar (single bit) and vector (one-dimensional array of bits) categories. Therefore two-dimensional (2D) and one-dimensional by one-dimensional (1D x 1D) arrays must be defined by the user. To do so, first we must define the new data TYPE then the new SIGNAL, CONSTANT or VARIABLE must be declared using that data type [1]. Generally an array is declared as follows.

```
TYPE type_name IS ARRAY (specification) OF data_type;
```

To make use of the new array type:

```
SIGNAL signal_name: type_name [:=initial_value];
```

4.5.2.2 Records

Records are similar to arrays with the only difference that they build from objects of different data types.

4.5.2.3 Port Array

As we mentioned above they are not pre-defined data types of more than one dimension. In a design maybe we need to specify the ports as arrays of vectors. Also we know that TYPE declarations are not allowed in an ENTITY. The solution is to declare user-defined data types in a PACKAGE which will then be visible to the whole design.

In the declarative part of the libraries an additional *USE* clause included to make user-defined package visible to the design.

4.5.2.4 Data Conversion

VHDL distinguishes strictly data types, which means that does not allow direct operations (arithmetic, logical, etc) between data of different types. So if we try to assign a value of type INTEGER in a signal of STD_LOGIC_VEVTOR then the compiler produces an error message [2]. An example is shown below.

```
SIGNAL x : INTEGER RANGE 0 TO 127;  
SIGNAL y : STD_LOGIC_VECTOR ( 7 DOWNT0 0);  
  
y<=x; --type mismatch
```

Hence it is necessary to convert data from one type to another. This can be done by using a FUNCTION from a pre-defined PACKAGE which is capable of doing it for us or we write a piece of VHDL code. In the example above the programmer must check if the proper packages, has declared with the right way. The packages enable to convert data from one type to another. So the type mismatch in the example can be corrected if we convert the signal INTEGER to a STD_LOGIC_VECTOR signal [2]. The function that must be used is:

```
conv_std_logic_vector (<signal>, <number_of_bits>);
```

Hence

```
y<=conv_std_logic_vector (x,8);
```

The previous function included in the package st_logic_arith. Therefore the package declared in the declarative part of the library by a USE clause.

Type-Conversions: The packets that describe the data types support direct type conversions with Type-conversion functions. The main cases are listed in table 4.1 [1].

4.6 Operators and Attributes

In VHDL we can do arithmetic, logical and concatenation operations using the proper operators. The operators that supported by VHDL are: Assignment, Arithmetic, Logical, Relational, Shift and Concatenation operators [2].

4.6.1 Assignment operators

They are used to assign values to SIGNALS, VARIABLES and CONSTANTS.

<= Used to assign a value to a SIGNAL.

:= Used to assign a value to VARIABLE, CONSTANT and GENERIC. They used also to declare initial values.

=> Used to assign values to individual vectors or with the OTHERS.

From	To	Type conversion function	Package of origin
INTEGER	STD_LOGIC_VECTOR	conv_std_logic_vector(a, cs)	std_logic_arith
	UNSIGNED	to_unsigned(a, cs) conv_unsigned(a, cs)	numeric_std std_logic_arith
	SIGNED	to_signed(a, cs) conv_signed(a, cs)	numeric_std std_logic_arith
	UFIXED	to_ufixed(a, cs)	fixed_generic_pkg
	SFIXED	to_sfixed(a, cs)	fixed_generic_pkg
	FLOAT	to_float(a, cs)	float_generic_pkg
BIT_VECTOR	STD_LOGIC_VECTOR	to_stdlogicvector(a, cs)	std_logic_1164
STD_LOGIC_VECTOR	INTEGER	conv_integer(a, cs) conv_integer(a, cs) to_integer(a, cs)	std_logic_signed std_logic_unsigned numeric_std_unsigned
	BIT_VECTOR	to_bitvector(a, cs)	std_logic_1164
	UNSIGNED	unsigned(a) (*) unsigned(a) (*)	numeric_std std_logic_arith
	SIGNED	signed(a) (*) signed(a) (*)	numeric_std std_logic_arith
	UFIXED	to_ufixed(a, cs)	fixed_generic_pkg
	SFIXED	to_sfixed(a, cs)	fixed_generic_pkg
	FLOAT	to_float(a, cs)	float_generic_pkg
UNSIGNED and SIGNED	INTEGER	to_integer(a, cs) conv_integer(a, cs)	numeric_std std_logic_arith
	STD_LOGIC_VECTOR	std_logic_vector(a) (*) std_logic_vector(a) (*) conv_std_logic_vector(a, cs)	numeric_std std_logic_arith std_logic_arith
	UNSIGNED	conv_unsigned(a, cs)	std_logic_arith
	SIGNED	conv_signed(a, cs)	std_logic_arith
	UFIXED (unsigned only)	to_ufixed(a, cs)	fixed_generic_pkg
	SFIXED (signed only)	to_sfixed(a, cs)	fixed_generic_pkg
	FLOAT	to_float(a, cs)	float_generic_pkg
UFIXED and SFIXED	INTEGER	to_integer(a, cs)	fixed_generic_pkg
	STD_LOGIC_VECTOR	to_slv(a, cs)	fixed_generic_pkg
	UNSIGNED (ufixed only)	to_unsigned(a, cs)	fixed_generic_pkg
	SIGNED (sfixed only)	to_signed(a, cs)	fixed_generic_pkg
	SFIXED (ufixed only)	to_sfixed(a, cs)	fixed_generic_pkg
	FLOAT	to_float(a, cs)	float_generic_pkg
FLOAT	INTEGER	to_integer(a, cs)	float_generic_pkg
	STD_LOGIC_VECTOR	to_slv(a, cs)	float_generic_pkg
	UNSIGNED	to_unsigned(a, cs)	float_generic_pkg
	SIGNED	to_signed(a, cs)	float_generic_pkg
	UFIXED	to_ufixed(a, cs)	float_generic_pkg
	SFIXED	to_sfixed(a, cs)	float_generic_pkg

(a, cs) = (argument, conversion specifications)
cs may include vector size, left/right range constants, overflow and rounding specs, etc. (consult package)
(*) = type casting

Table 4.1 Main type conversion options

4.6.2 Logical Operators

They used to perform logical operations. The data can be of type BIT, STD_LOGIC or STD_ULOGIC, BIT_VECTOR, STD_LOGIC_VECTOR or STD_ULOGIC_VECTOR [1]. The logical operators are: NOT, AND, NAND, OR, NOR, XOR and XNOR.

4.6.3 Arithmetic operators

They used to perform arithmetic operations. The data can be of type INTEGER, SIGNED, UNSIGNED or REAL. Also we can use arithmetic operators with type STD_LOGIC_VECTOR if the package *std_logic_signed* or *std_logic_unsigned* of the library *ieee* is declared. The arithmetic operators are: + Addition, - Subtraction, * Multiplication, / Division, ** Exponentiation, MOD (y mod x returns the remainder of y/x with the signal of

x), REM (y rem x returns the remainder of y/x with the signal of y), and ABS (returns the absolute value) [1].

4.6.4 Relational operators

The relational operators are: = Equal to, /= Not equal to, < Less than, > Greater than, < = Less than equal to, and > = Greater than or equal to.

The synthesizable predefined data types that support relational operators are BIT, BIT_VECTOR, BOOLEAN, INTEGER, NATURAL, POSITIVE, CHARACTER and STRING. If one of the package *numeric_std* or *std_logic_arith* is declared then UNSIGNED and SIGNED data type can also be used. Also if the package *std_logic_unsigned*, *std_logic_signed* or *numeric_std_unsigned* is declared then STD_LOGIC_VECTOR can be used [1].

4.6.5 Shift operators

The shift operators used by VHDL for shifting the bits of data type vector. They support the predefined type BIT_VECTOR. If the package *numeric_std* is declared then UNSIGNED and SIGNED data type can also be used [2]. The operators and their functions are shown in the table 4.2 [2].

SIGNAL x, y : BIT_VECTOR (3 DOWNTO 0);

x <= "1101"

Operator	Syntax	Function	Result (n=2)
SLL	y< = x SLL n	Shift left logic. Positions on the right are filled with '0'	y=0100
SRL	y< = x SRL n	Shift right logic. Positions on the left are filled with '0'	y=0011
SLA	y< = x SLA n	Rightmost bit is replicated on the right	y=0111
SRA	y< = x SRA n	Leftmost bit is replicated on the left	y=1111
ROL	y< = x ROL n	Circular shift to the left	y=0111
ROR	y< = x ROR n	Circular shift to the right	y=0111

Table 4.2 Shift operators and their Functions.

4.6.6 Concatenation operators

The concatenation operator is used for grouping objects and values. The representation of concatenation operator is &. It supports the synthesizable predefined data types BIT_VECTOR, BOOLEAN_VECTOR (VHDL 2008), INTEGER_VECTOR, STD_(U)LOGIC_VECTOR, (UN)SIGNED and SRTING [1].

4.7 Attributes

VHDL gives on data types and data objects certain attributes. The language distinguishes data attributes and signal attributes.

Data Attributes: Returns value about a data vector. The pre-defined synthesizable data attributes are shown in the table 4.3.

Attribute	Function
Signal_name'LOW	Returns lower array index
Signal_name'HIGH	Returns upper array index
Signal_name'LEFT	Returns leftmost array index
Signal_name'RIGHT	Returns rightmost array index
Signal_name'LENGTH	Returns vector size
Signal_name'RANGE	Returns vector range
Signal_name'REVERSE_RANGE	Returns vector range in reverse order

Table 4.3 Pre-defined data attributes

Signal Attributes: They are used to monitor a signal (returns the value TRUE or FALSE). The signal attributes of a signal x are shown in table 2.4.

Attribute	Function
x'EVENT	Returns true when an event occurs to x
x'STABLE	Returns true if no event has occurred on x
x'ACTIVE	Returns true if x='1'
x'QUIET<time>	Returns true if no event has occurred during the time specified
x'LAST_EVENT	Returns the time elapsed since last event
x'LAST_ACTIVE	Returns the time elapsed since last x='1'
x'LAST_VALUE	Returns the value of x before the last event

Table 4.4 Signal attributes

4.8 Concurrent and Sequential codes.

In paragraph 2.1 we mentioned that contrary to other regular computer programs which are sequential, VHDL statements are inherently concurrent. Only the statements that are placed inside a PROCESS, FUNCTION and PROCEDURE are executed sequentially. Despite the fact that this block of code is executed sequentially, the block as a whole is concurrent with any external statements. In order to describe sequential logic circuits, sequential code must be employed. On the other hand combinational logic circuits are built with concurrent code. Also with sequential code we can implement both sequential as well as combinational circuits.

4.8.1 Concurrent Code

Code which is outside from PROCESS, FUNCTION and PROCEDURE considered concurrent and executed directly regardless of the position command inside the code. In each cycle of simulation the simulator returns all lines of code and transmits the changes by updating the results until the signals are stabled. During synthesis of concurrent code the compiler synthesize combinational circuits which refresh immediately the outputs as a result of combinational of the inputs [1].

The statements *SELECT*, *WHEN* and *GENERATE* are concurrent statements and are placed outside from a PROCESS and subprograms (FUNCTION and PROCEDURE).

4.8.1.1 SELECT statement

Its syntax is shown below.

```
WITH identifier SELECT  
assignment WHEN value,  
assignment WHEN value,  
...;
```

The identifier it could be the name of a signal. Since the SELECT requires to covering all the possible values taken by the identifier code structure above, often ends with the expression WITH OTHERS; [1].

4.8.1.2 WHEN...ELSE statement

The concurrent WHEN...ELSE statement is the simplest conditional statement. It is approximately equivalent to the sequential statement IF [2]. Its syntax is shown below.

```
Assignment_expression WHEN conditions ELSE  
Assignment_value WHEN conditions ELSE
```

...;

4.8.1.3 GENERATE statement

GENERATE is another concurrent statement and the form FOR...GENERATE (unconditional GENERATE) is equivalent to the sequential statement LOOP. This statement creates a section of code which is repeated a number of times. The number of times is declared by an index [1]. The other form is the IF...GENERATE (conditional GENERATE). Its syntax is shown below (for the popular form unconditional GENERATE)

label: FOR identifier IN range GENERATE

[declarative_part

BEGIN]

Concurrent_statements_part

END GENERATE [label];

Notice that the word BEGIN is only needed when declarations are made.

4.8.2 Sequential code

As we mentioned in paragraph 4.8 the sequential statements are inside PROCESS or inside subprograms (FUNCTIONS and PROCEDURES). The sequential statements are *IF*, *WAIT*, *LOOP* and *CASE*. One important aspect of sequential code is that it is not limited to sequential logic. We can build sequential circuits as well as combinational circuits. The sequential code is also called *behavioral* code [1].

VARIABLES can be used only in sequential code which means that are inside a PROCESS, FUNCTION or PROCEDURE. Contrary to a SIGNAL a VARIABLE can never be global, so its value can't be passed out directly. If it is necessary then the value must be assigned to a SIGNAL. The update of a VARIABLE is immediate which means that the new value is used to the next line of the code. That does not apply when a SIGNAL used inside a PROCESS. The new value is available after the conclusion of the present run of the PROCESS [2].

4.8.2.1 PROCESS

The most common section of a sequential code is PROCESS. Inside a PROCESS there are the sequential statements *IF*, *WAIT*, *LOOP* and *CASE* and also there is a sensitivity list. When the WAIT statement is used there is not a sensitivity list. The PROCESS is executed every time a signal in the sensitivity list changes. In case of WAIT statement the PROCESS is executed when the condition related to WAIT is fulfilled. The sequential statements inside a PROCESS

are executed sequential one after the other [1]. The whole PROCESS from the perspective of the user execute like a concurrent statement. Its syntax is shown below .

```
[label:] PROCESS (sensitivity list)
    [VARIABLE name type [range] [:= initial_value;]]
BEGIN
    (sequential code)
END PROCESS [label];
```

Inside a PROCESS the use of VARIABLE is optional. If used then it must be declared in the declarative part of the PROCESS which is before the word BEGIN. The label can be any word except VHDL reserved words.

4.8.2.2 IF statement

IF is a sequential statement which can be used only inside a PROCESS, FUNCTION or PROCEDURE. IF is a conditional branch statement. Its syntax is shown below.

```
IF conditions THEN assignments;
ELSIF conditions THEN assignments;
...
ELSE assignments;
END IF;
```

4.8.2.3 WAIT statement

It is another sequential statement which can be used only inside a PROCESS, FUNCTION or PROCEDURE. As we mentioned in section 4.8.2.1 when the WAIT statement is used there is not a sensitivity list in the PROCESS. Also the operation of WAIT is sometimes similar to that of IF. There are three forms of WAIT statement which are shown below.

```
[label:] WAIT UNTIL signal_condition;
[label:] WAIT ON sensitivity_list;
[label:] WAIT FOR time;
```

The WAIT UNTIL statement accepts only one signal and since the PROCESS does not have a sensitivity list the WAIT UNTIL statement should be the first statement in the PROCESS.

The WAIT ON statement accepts multiple signals. The PROCESS is put on hold until any of the signals listed changes.

The WAIT FOR statement is used only for simulation.

4.8.2.4 CASE statement

CASE is another sequential statement which is used exclusively for sequential code. Its syntax is shown below.

```
CASE identifier IS
  WHEN value => assignments;
  WHEN value => assignments;
  ...
END CASE;
```

The CASE statement which used in sequential code is similar to WAIT statement of concurrent code. In CASE all permutations must be tested hence the keyword OTHERS is helpful [1].

4.8.2.5 LOOP statement

The LOOP statement is useful when a piece of code must be instantiated several times. There are several ways of using which are shown below.

- FOR/LOOP: The loop is repeated a fixed number of times.
[label:] FOR identifier IN range LOOP
 (sequential statements)
END LOOP [label];
- WHILE/LOOP: The loop is repeated until a condition no longer holds.
[label:] WHILE condition LOOP
 (sequential statements)
END LOOP [label];
- EXIT: Using exit the loop.
[label:] EXIT [label] [WHEN condition];
- NEXT: Used for skipping loop steps.
[label:] NEXT [loop_label] [WHEN condition];

4.9 Components

The *COMPONENT* is a piece of code which includes LIBRARY declarations, ENTITY and ARCHITECTURE. VHDL is a language which allows the construction of hierarchical designs [2]. So a complex circuit can be described by sub-circuits which are included in a top-level entity and called components. The components are smaller circuits that have already described using VHDL. Also a COMPONENT is another way of partitioning a code and providing code sharing and code reuse [1].

To use (instantiate) a COMPONENT first must be declared. The syntax is shown below.

Declaration:

```
COMPONENT component_name IS  
  
PORT(  
  
    port_name : signal_mode signal_type;  
  
    port_name : signal_mode signal_type;  
  
...);  
  
END COMPONENT;
```

Instantiate label: component_name PORT MAP (port_list);

From the syntax above we can see that a label is required to instantiate a COMPONENT. Also a PORT MAP declaration is needed. The port_list is a list that relates the ports of the actual circuit to the ports of the pre-designed COMPONENT which is being instantiated [1]. There are two ways to map the PORTS of a COMPONENT. The first is *positional mapping* and the second is *nominal mapping*. Positional mapping is easy to write but nominal mapping is less error-prone [1].

Example:

Positional mapping.

```
COMPONENT file_read IS  
  
PORT ( clk: IN STD_LOGIC;  
  
    rst: IN STD_LOGIC;  
  
    x_out: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));  
  
END COMPONENT;
```

...

```
U1: file_read PORT MAP (clk, rst, z);
```

Nominal mapping

```
U1: file_read PORT MAP ( clk=>clk, rst=>rst, x_out=>z);
```

There are two ways to declare a COMPONENT. Once we have designed and placed it in the destination LIBRARY, we can declare it in the main code or we can declare it using a PACKAGE.

4.10 Simulation (VHDL Test benches)

In this paragraph we will describe simulation with VHDL Test benches which is one of the most crucial steps in VHDL language.

Figure 4.1 in paragraph 4.1 shows a simplified view of the design flow using VHDL. After synthesis a functional simulation is executed which verify that the functionalities hold after the process of synthesis. We must mention here that there is no timing information yet. After fitting is the final simulation which includes internal sell and routing delays. After this procedure the actual size of the devise is represented. Since timing information is included, it is a timing simulation. With the timing information generated by the fitting process the software allows the circuit to be fully simulated [2]. Time information is annotated in a SDF (Standard Delay Format) file. Such information allows simulation under best and worst case operating conditions.

4.10.1 Simulation Types

There are four simulation types that are summarized below.

Type I test bench (manual functional simulation): The internal delays of the DUT (Design Under Test) are not considered and the output is manually verified. Figure 4.3 shows that the stimuli are produced by a VHDL code while the output is graphical.



Figure 4.3 Type I test bench

Type II test bench (manual timing simulation): The internal delays of the DUT are taken into account, and the output is still manually verified. Figure 4.4 shows Type II test bench in which the stimuli are produced by a VHDL code while the output is graphical.

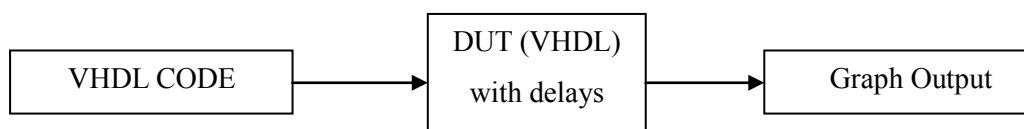


Figure 4.4 Type II test bench

Type III test bench (automated functional simulation): The internal delays of the DUT are not considered but the output is automatically verified by the simulator. Figure 4.5 shows that both input and output are treated using VHDL.



Figure 4.5 Type III test bench

Type IV test bench (automated timing simulation): The internal delays of the DUT are taken into account, and the output is automatically verified by the simulator. Figure 4.6 shows Type IV test bench in which both input and output are treated using VHDL.

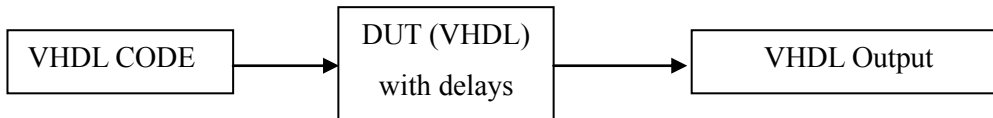


Figure 4.6 Type IV test bench (full bench)

It is also necessary to specify which files must be provided to run simulation. When using VHDL test benches different files are needed to test the circuit. Two files are needed to be prepared by the user and the others produced automatically by the synthesizer. For functional simulation (test benches I and III) only two files must be prepared by the user, a Design File and a Test file (see appendix E). On the other hand timing simulation (test benches II and IV) needs more files which are: a Design File (prepared by the user), a Test File (prepared by the user), a Postsynthesis File (generated by the synthesizer) and SDF File (generated by the synthesizer) [1].

A generalization for a Test File (fir_tb) is shown below. As we can see, a Test File is similar to a regular VHDL code which has LIBRARY declarations, ENTITY and ARCHITECTURE. The particularity in the ENTITY is that it is empty except from GENERIC which can be used optional. The particularity in the ARCHITECTURE is that it is for simulation and not for hardware inference [1].

In the declarative part of the ARCHITECTURE (lines 10-19) the DUT is declared along with the signals needed to test. The signals are clk_tb, rst_tb and b_tb. In the ARCHITECTURE body (lines 20-33) first the DUT is instantiated (lines 22-23), then the stimuli are generated (lines 25-26) and finally an optional code section is shown (lines 28-32). The optional code section is needed when we want the simulator to automatically compare the values obtained for b_tb against expected values for b.

```

1  -----fir_tb-----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
  
```

```

5  ENTITY fir_tb IS
6      GENERIC (...);
7  END fir_tb;
8  -----
9  ARCHITECTURE behavior OF fir_tb IS
10 -----DUT_declaration-----
11 COMPONENT digital_filter IS
12     PORT (clk: IN STD_LOGIC;
13           rst: IN STD_LOGIC;
14           b: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
15 END COMPONENT;
16 -----signal_declarations-----
17 SIGNAL clk_tb: STD_LOGIC:= '0';
18 SIGNAL rst_tb: STD_LOGIC:= '0';
19 SIGNAL b_tb: STD_LOGIC_VECTOR (7 DOWNTO 0);
20 BEGIN
21 -----DUT_instantiation-----
22 dut: digital_filter
23 PORT MAP (clk=>clk_tb, rst=>rst_tb, b=>b_tb);
24 -----Stimuli_generation-----
25 clk_tb<= NOT clk_tb AFTER 10ns;
26 rst_tb<= '1' AFTER 5ns, '0' AFTER 20ns;
27 -----Output_verification_optional-----
28 PROCESS
29 BEGIN
30     WAIT FOR...
31     ASSERT (b_tb=b)....
32 END PROCESS;
33 END behavior;
34 -----
35

```


5 Generating the Filter Coefficients

5.1 Introduction

The filter that we are going to design is an FIR low-pass filter using the window method design. The procedure is to create the filter coefficients from the filter specifications using the proper window. As we have already mentioned in paragraph 3.5.4 there are several popular windows from which to choose. Also table 3.2 summarizes their key points.

The steps for using windows are:

- Determine the window type that will satisfy the stop-band attenuation requirements.
- Determine the minimum size of the window (N) using the transition width.
- Calculate the filter coefficients.
- Generate the impulse response of the low-pass filter.

For the last two steps we will use the Filter Design and Analysis Tool (FDATool) which is a powerful graphical user interface in the Signal Processing Toolbox for designing and analysing filters. Therefore before designing the filter a brief presentation of FDATool is necessary [8].

5.2 FDATool

FDATool enable us to quickly design digital FIR or IIR filters by setting the filter specifications.

We start the FDATool from the Matlab writing in the command window `>>fdatool`

The Graphical User Interface (GUI) displays with a default filter (see figure 5.1). The GUI has three main regions:

- The Current Filter Information region.
- The Filter Display region.
- The Design panel

The upper half of the GUI displays information on filter specifications and responses for the current filter. The Current Filter Information region in the upper left, displays filter properties, namely filter structure, order, number of sections used and whether the filter is stable or not.

The Filter Display region, in the upper right, displays various filter responses, such as, magnitude response, group delay and filter coefficients.

The lower half of the GUI is the interactive portion of FDATool. The Design Panel in the lower half is where we define the filter specifications. It controls what is displayed in the other two upper regions.

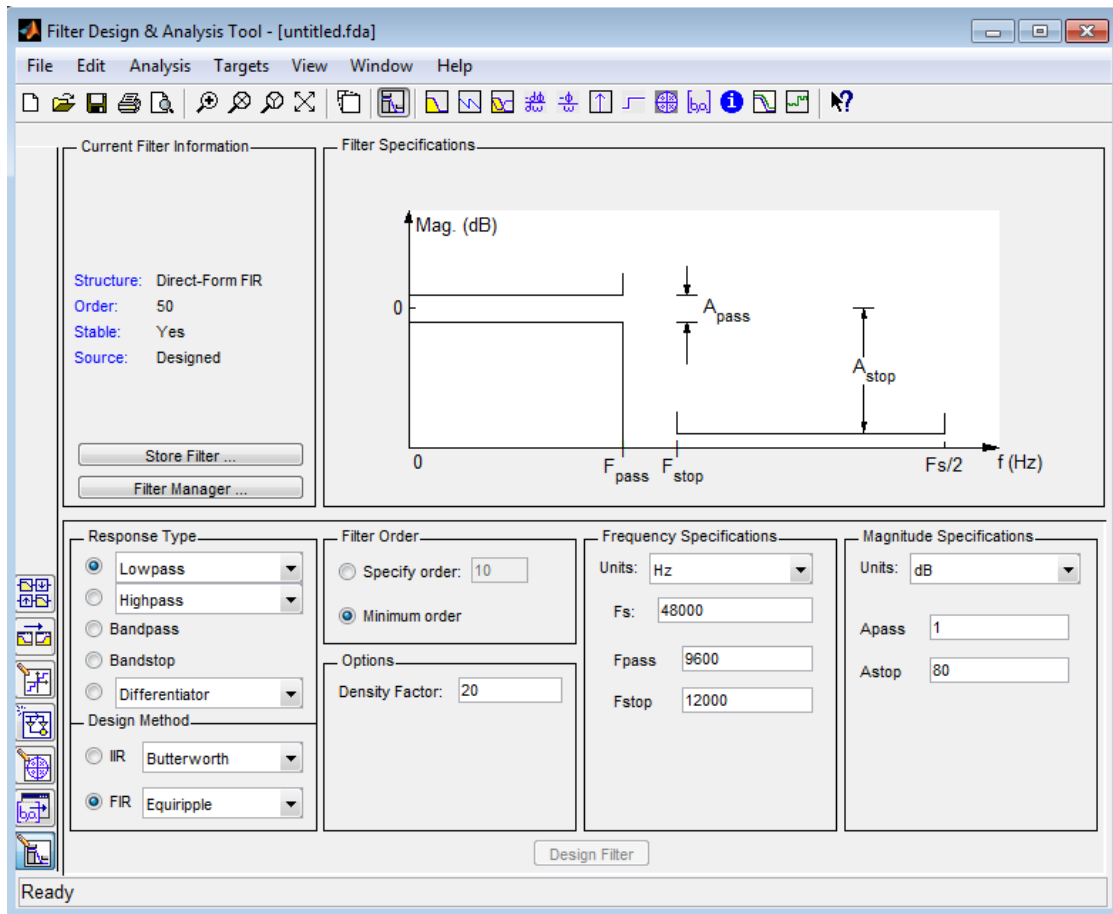


Figure 5.1 Graphical User Interface

5.3 Designing the FIR filter

We will design a low-pass FIR filter using the window method (see paragraph 3.5.4). The filter specifications are:

Sampling Frequency $F_s = 2$ MHz.

Pass-band Frequency $F_p = 10$ KHz.

Stop-band Frequency $F_{st} = 75$ KHz – 77 KHz, 50 dB minimum attenuation.

The first step is to determine the window type. From table 3.2 we can observe that Hamming, Blackman and Kaiser ($\beta \geq 4.55$) windows satisfy the stop-band attenuation requirements. To minimize the transition width and therefore minimize the number of calculations we choose the Hamming window.

The next step is to calculate the size of the window (N) which is the number of coefficients of the filter. The transition width Δf (normalized) is:

$$\Delta f = \frac{f_2 - f_1}{f_s}$$

Now using the relationship from the table 3.2 for the Hamming window we calculate the number of coefficients (N).

$$N = \frac{3.1}{\Delta f}$$

Hence the number of coefficients is 101.

The next step is to calculate the filter coefficients. For this purpose we will use the FDATool. The frequency F_c in FDATool is in the middle of the transition band.

Therefore

$$F_c = \frac{f_1 + f_2}{2}$$

Hence

We Select Low-pass from the dropdown menu under Response Type and Window under FIR Design Method.

We Select Specify order in the Filter order area and we enter the number 100. The order of the filter is N-1 hence $101-1=100$. Also in the Options area we select Hamming window. Now in the Frequency specifications we select $F_s = 2000000\text{Hz}$ and $F_c = 42600\text{Hz}$.

After setting the design specifications we click the Design Filter button of the GUI to design the filter. The magnitude response of the filter is displayed in the Filter Analysis area after the coefficients are computed. Figure 5.2 shows the frequency response of the filter. Also figure 5.3 shows the impulse response of the filter and we can observe the symmetry in the coefficients. The symmetry in the coefficients ensures that the phase is linear in the pass-band which can be shown in figure 5.4. Finally, we multiply the coefficients by 2^{15} and rounding them as integer values (see Appendix F.2). Table 5.1 shows the filter coefficients as integer values.

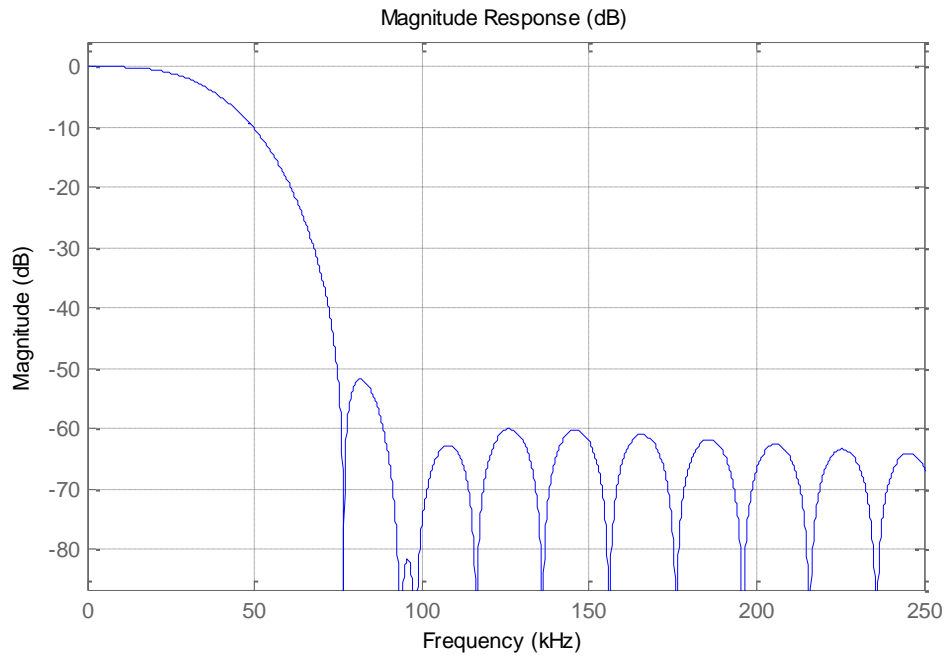


Figure 5.2 Frequency response of the filter

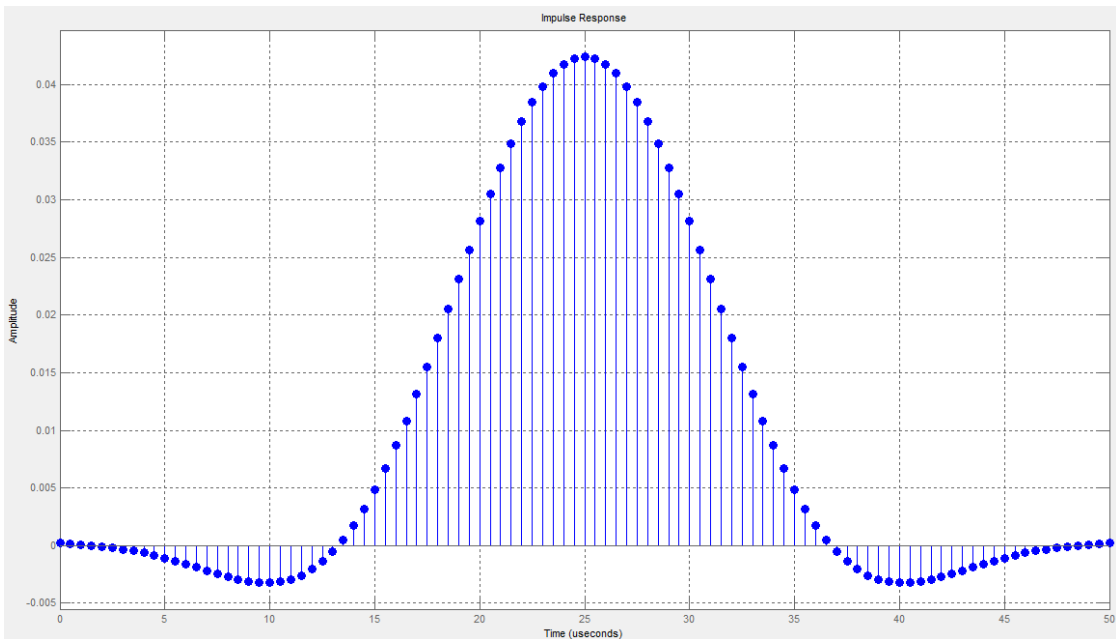


Figure 5.3 Impulse response of the filter

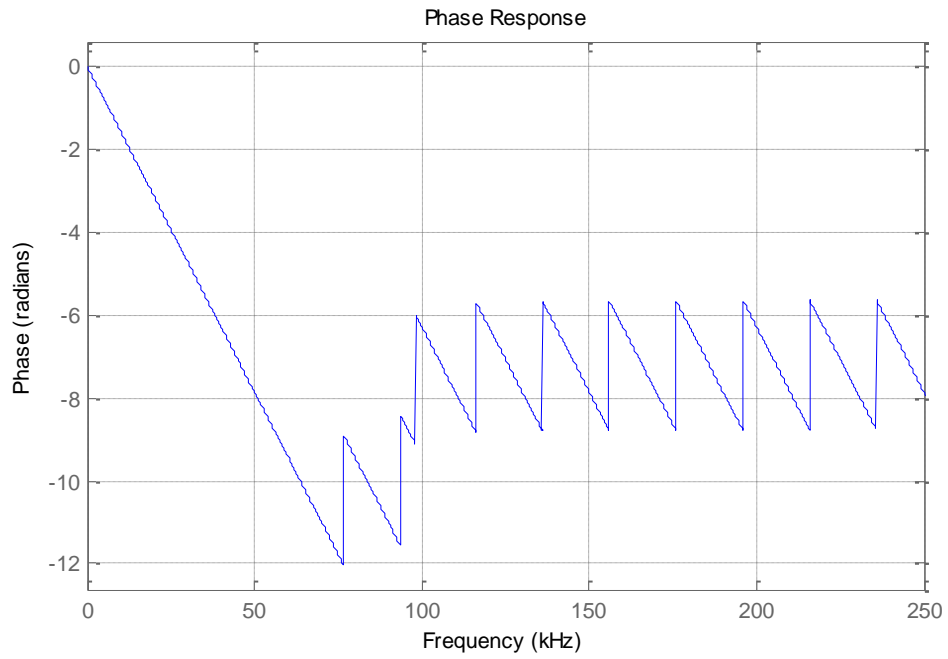


Figure 5.4 Linear phase in the pass-band area

1.	7	12.	-43	23.	-96	34.	355	45.	1206	56.	1260	67.	430	78.	-84	89.	-52	100.	5
2.	5	13.	-52	24.	-84	35.	430	46.	1260	57.	1206	68.	355	79.	-96	90.	-43	101.	7
3.	3	14.	-61	25.	-68	36.	509	47.	1306	58.	1144	69.	284	80.	-103	91.	-35		
4.	0	15.	-71	26.	-46	37.	590	48.	1343	59.	1075	70.	19	81.	-106	92.	-27		
5.	-3	16.	-80	27.	-17	38.	674	49.	1369	60.	1001	71.	159	82.	-105	93.	-21		
6.	-6	17.	-89	28.	17	39.	758	50.	1385	61.	922	72.	105	83.	-102	94.	-15		
7.	-10	18.	-96	29.	58	40.	841	51.	1391	62.	841	73.	58	84.	-96	95.	-10		
8.	-15	19.	-102	30.	105	41.	922	52.	1385	63.	758	74.	17	85.	-89	96.	-6		
9.	-21	20.	-105	31.	159	42.	1001	53.	1369	64.	674	75.	-17	86.	-80	97.	-3		
10.	-27	21.	-106	32.	219	43.	1075	54.	1343	65.	590	76.	-46	87.	-71	98.	0		
11.	-35	22.	-103	33.	284	44.	1144	55.	1306	66.	509	77.	-68	88.	-61	99.	3		

Table 5.1 Filter coefficients as integer values

6 Implementation in Hardware Description Language

6.1 Diagram of the filter

In figure 6.1 we observe the filter diagram implemented in hardware description language. This filter consists of three components, namely *rom* memory, a *shift register* and a *multiplier-accumulator (mac)*. The input x and the output y of the filter are 8 bits `std_logic_vector` data type, which is the industrial standard. Also the `clk` and `rst` are signals of `std_logic`.

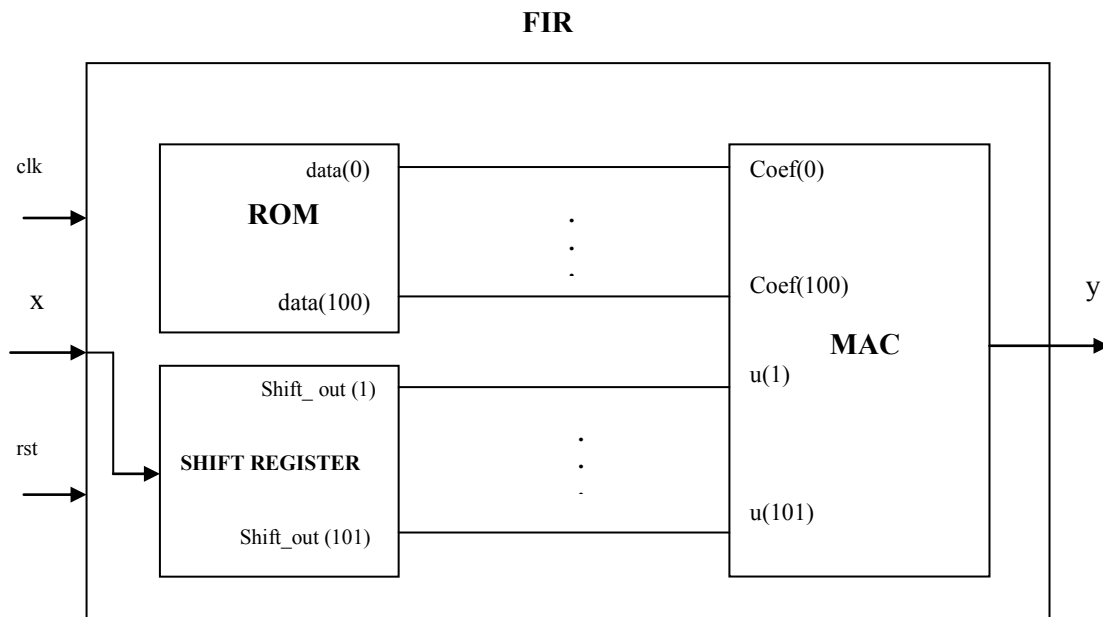


Figure 6.1 Diagram of the FIR

6.2 Internal parts of the implemented filter

Moreover, figure 6.2 portrays the internal parts of the implemented filter. The 101 filters coefficients, that have been generated as described in section 5.3, are stored in a rom memory as integer values. The memory output is an one dimensional array comprising 101 signed elements of 15 bits each. The input signal namely the samples created through the signals sampling, is driven in the filter input. The samples are then driven in the shift register input in 8 bits signed form. This particular register is a serial input – parallel output register, which means that at each positive edge of the clock we have a data insertion with their concurrent right shift at each time instant. The shift register output is a one dimension array comprising 101 signed elements of 15 bits each. The outputs of the memory and the shift register, both one dimension arrays with equal number of elements, are driven to the multiplier-accumulate unit so as for the samples to be multiplied with the coefficients. The multiplier output is also a 23

bits one dimension array. The number 23 comes of the addition of the 15 bits of the coefficients plus the 8 bits of the shift register. Next, the accumulator contacts the addition of the results that come from the output of the multiplier. The accumulator is a ten places array. The nine first places include 10 elements of the convolve array while the tenth includes 11 elements of the convolve array, thereby succeeding to parallel add the first 5 array elements with the following 5 array elements. This way we manage to parallel add the 10 convolve elements situated in each place of the accum array. Next we perform the normalization of the output dividing by 2^{15} , which essentially is 15 places right sift. In the end the filter output is the convolution of the filter coefficients with the input samples. However, it is of the importance to point out that the first and the last 101 results are invalid (see paragraph 3.5.2).

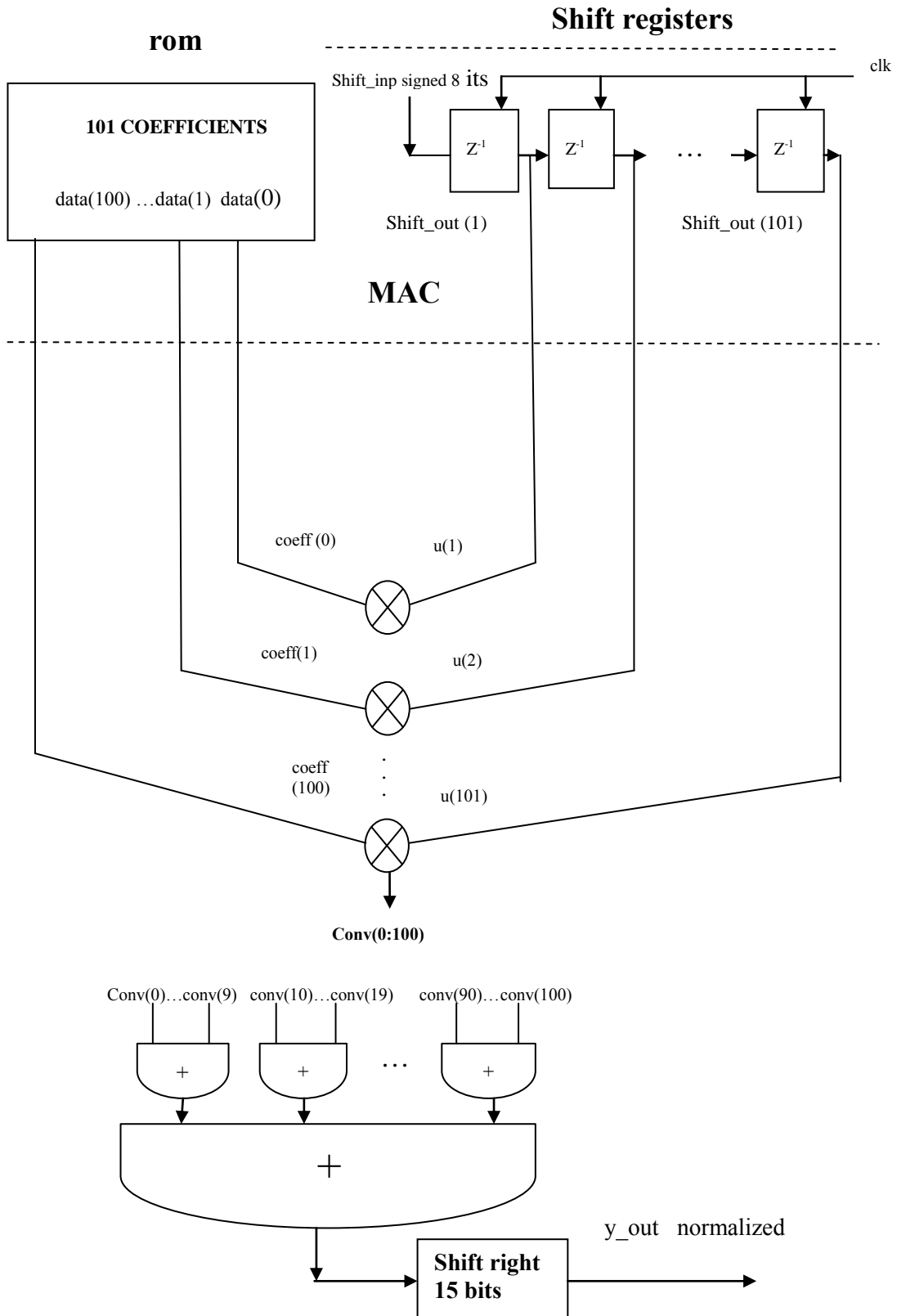


Figure 5.2 Internal parts of the implemented filter

6.3 Entities Hierarchy Diagram

The filter has been implemented employing components denoted in a particular package (see paragraph 4.9). As a methodology, bottom-up hierarchical design is used, in which the various components are first described in VHDL and then instantiated in order to produce the top design entity of the filter (see Appendix A). The entities hierarchy diagram is shown in figure 6.3. It consists of the three following components *mac*, *shift_reg* and *rom*, the main code *fir* (top level entity), which encompasses the instance of the components. Also there exist “*components*” and “*my_declarations*” packages.

In addition, we use in the declarative part of the libraries the *numeric_std* package (see paragraph 4.5.1.3) because is standardized by ieee.

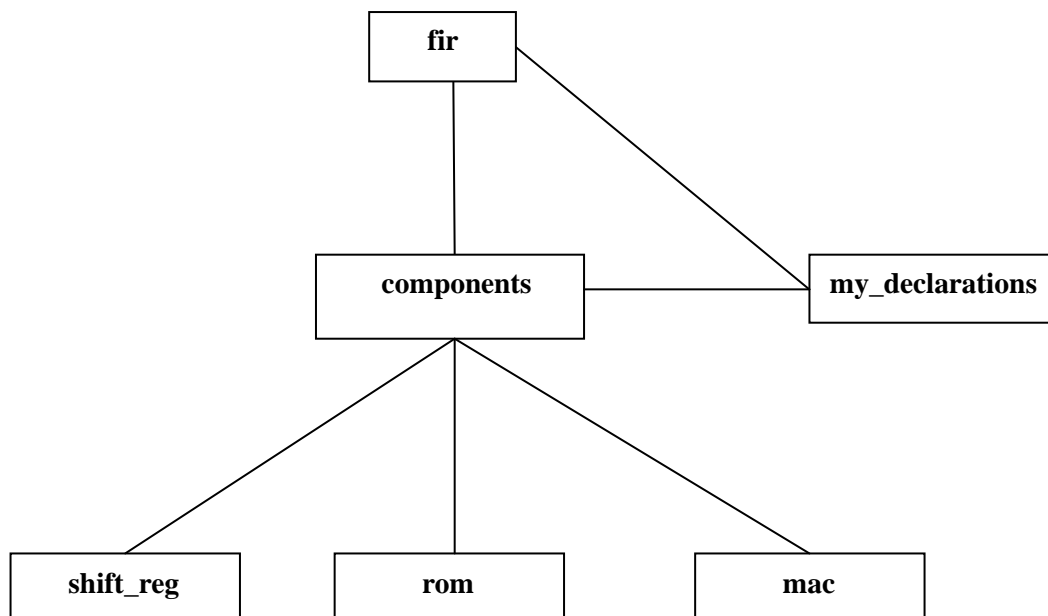


Figure 6.3 Entities Hierarchy Diagram

Main code (see Appendix A):

Main code *fir* includes the declarations of the libraries as well as the declarations of the packages “*components*” and “*my_declarations*”. Input *x* and output *y* of the filter are *std_logic_vector* of 8 bits. Moreover the signals *clk* and *rst* are declared. In the code architecture there exist the instance of the *shift_reg*, *mac* and *rom* components. Also the signals *c* and *w* are declared with the aim of correctly corresponding the gates between the components and the main code. This mapping is nominal. Furthermore in the main code architecture the input signal is converted from *std_logic_vector* to 8 bits signed.

Package *my_declarations* (see Appendix A):

This package contains the basic data type declarations (see paragraph 4.5.2.3) included in the code. These data types are:

memory_coeffs: an one dimension array which include the 101 filters coefficients. The data type is integer.

convolve: an one dimension array which includes the multiplications. The data type is signed of 23 bits.

maccum: an one dimension array which includes the additions. The data type is signed of 23 bits.

vector_coeffs: an one dimension array which includes the filter coefficients. The data type is signed of 15 bits.

window: an one dimension array which includes the sampling input. The data type is signed of 8 bits.

Package Component (see Appendix A):

Component shift_reg.

In the shift register the data are serially loaded in the shift_inp input while the output is an one dimension array type window of 101 spaces. This data type window has been defined in my_declaration package which is declared in the declarative part of the shift_reg library. The structure gen0 nullifies the outputs of the registers when the rst signal takes the logical value 1. The structure gen1 guiding the compiler to create a number of registers (one after another) where each register receives input from the output of the previous register.

Component rom:

In this rom the 101 filter coefficients are stored. In the rom architecture the coefficients are converted from integer to 15 bits signed. The output is an one dimension array type vector_coeffs of 101 spaces.

Component mac:

Mac has two inputs, coeff signal and u signal. Coeff signal is one dimension array type vector_coeffs while u signal is also an one dimension array of type window. The mac output is a signal std_logic_vector of 8 bits. In the mac architecture there exist two processes. The former is dedicated to the multiplication of the memory coefficients to the input signals while the latter is utilized for the addition. The sensitivity list for both processes is the clk signal. In the first occasion the multiplications take place at each positive edge of the clock where as in the second occasion the variables accumulate, accumulate1 and accumulate2 are nullified. The nullification of the variables is of paramount significant because we must not let the previous

result to be added to each new addition. The variable accum holds the result from the addition of ten conv elements. The structure gen2 is used for the addition of the first 5 maccum array elements and structure gen3 is used for the addition of the rest 5 maccum array elements. The normalization of the output is performed by dividing by 2^{15} using the shift operator SRL. At the end we convert the output from signed to std_logic_vector of 8 bits.

7 Simulation - Results

7.1 Simulation

The design software that used for the synthesis of the filter is the Quartus II by Altera while Modelsim by Mentor Graphics used for simulation. The procedure for simulation is to read from a txt file the input data and store the results in another txt file. Figure 7.1 shows the simulation procedure. For this purpose we create in Matlab a txt file for each frequency inserted in the filter (see Appendix F.1). This txt file encompasses integer values coming from the input signals sampling. Afterwards, through a file that has been created with the assistance of VHDL description language (see appendix D file_read) the data that are inserted in the shift register are read. Then follow the convolution of the data with the filter coefficients. The convolution outcomes are available at the filter output. In order to store the convolution results we have generated a txt file using the aforementioned hardware description language (see appendix D file_store). The values in the txt file are also integers. Also for the simulation another two files have to be prepared by the user (see paragraph 4.10.1) when using VHDL test benches. The files are a Design File and a Test File (see Appendix E).



Figure 7.1 Procedure for simulation

7.2 Results

In this paragraph we will verify the results we derive at the output of the filter through the simulation. These results shall be compared to the Matlab outcomes (see Appendix F.1). This comparison will demonstrate if the filter that we implemented meets the demands we set in paragraph 5.3 and that the filter behavior is in accordance with magnitude response depicted in figure 5.4.

The following figures (7.2 – 7.31) show the signals at the output of the filter as calculated in Matlab programming environment using the original double precision coefficients (red line) and the simulator Modelsim using fixed point arithmetic (blue line). Table 7.1 contains the corresponding values of the output signals as calculated in Matlab and the simulator Modelsim (columns 2 and 3 respectively). Also, in columns 4 and 5 of the table there are the values in columns 2 and 3 in dB.

Signal 500 Hz

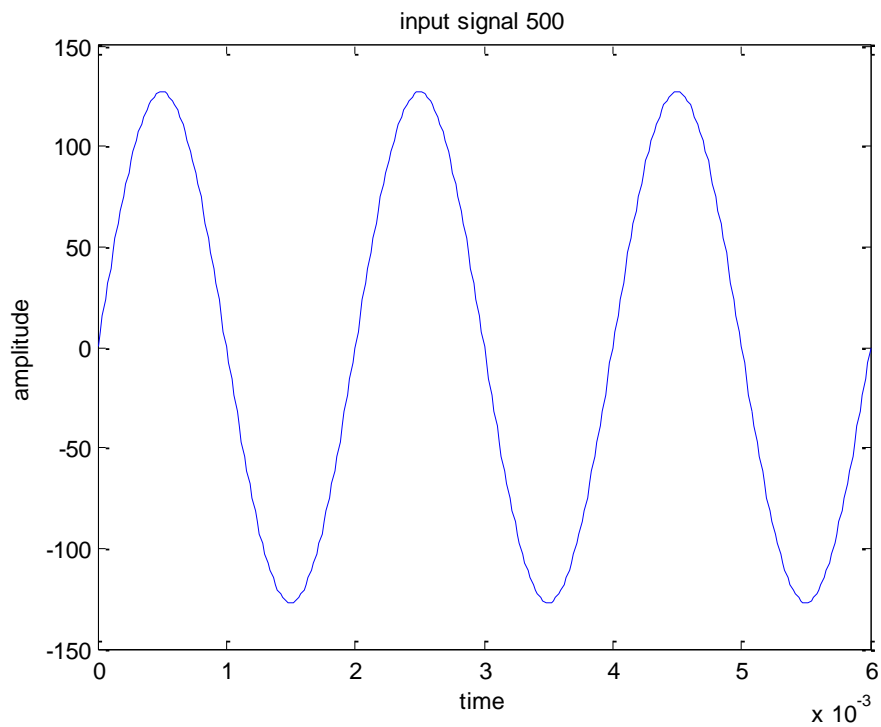


Figure 7.2 Input signal 500 Hz

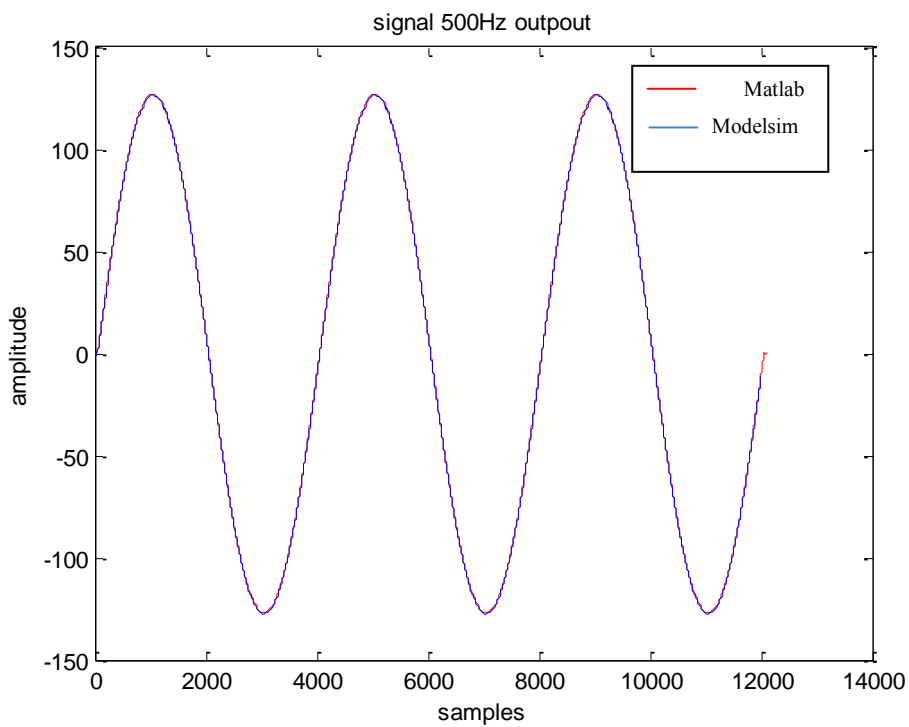


Figure 7.3 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)

Signal 1 KHz

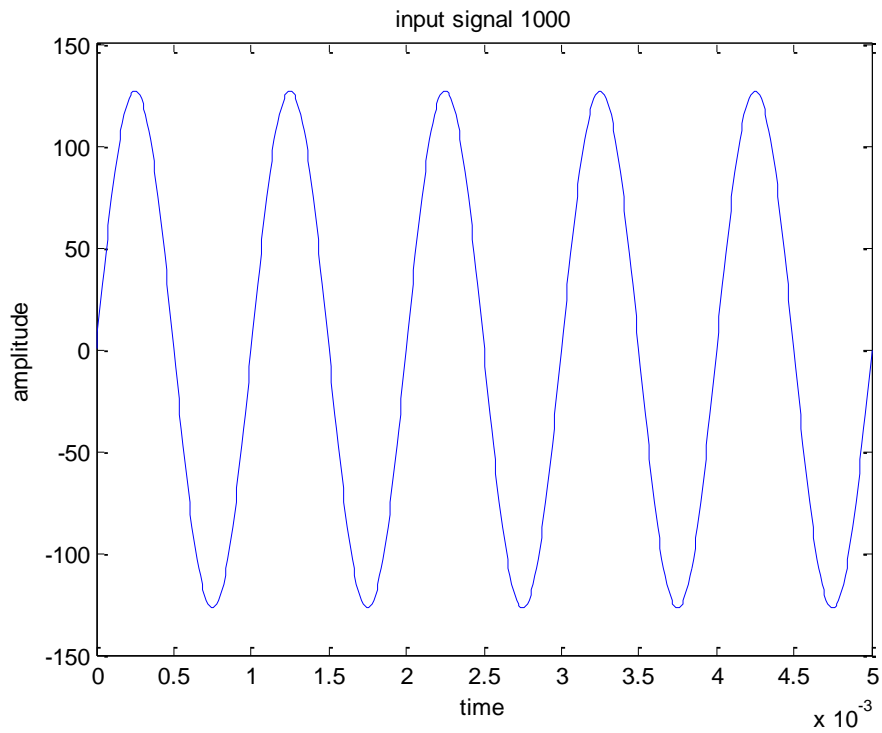


Figure 7.4 Input signal 1KHz

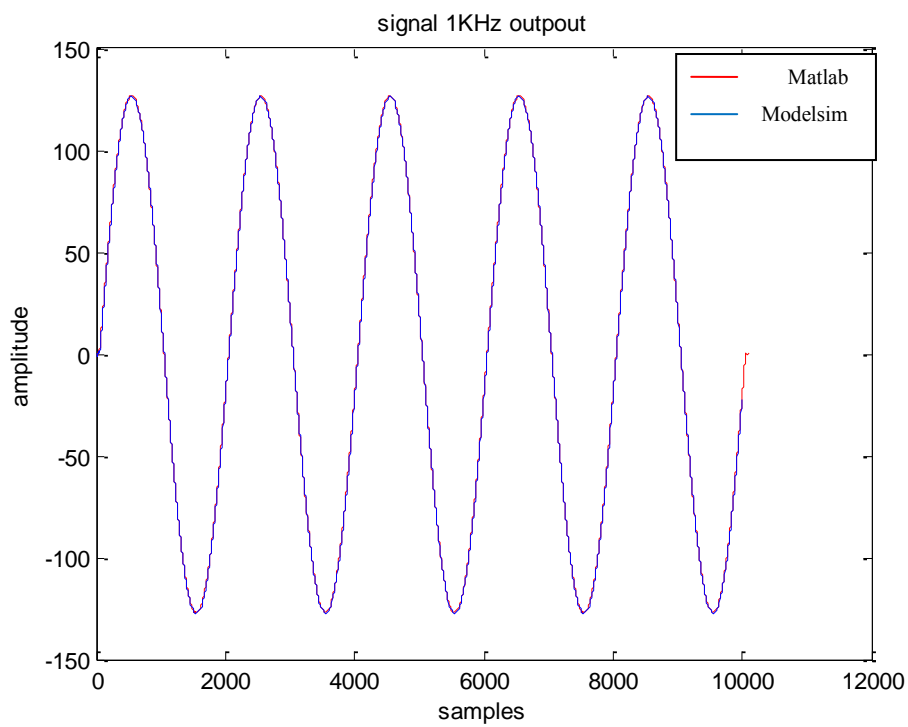


Figure 7.5 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)

Signal 5 KHz

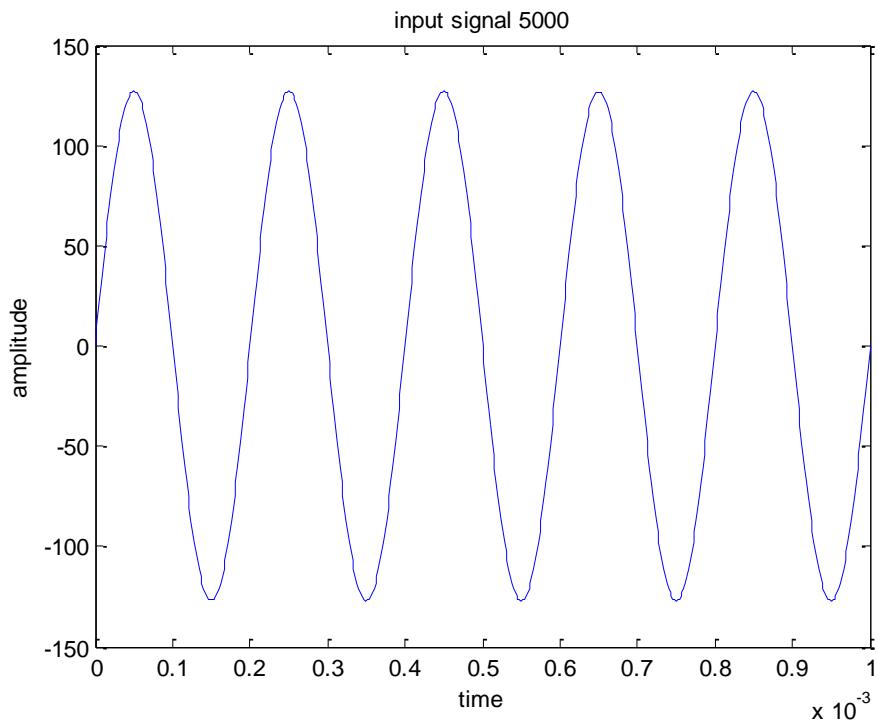


Figure 7.6 Input signal 1KHz

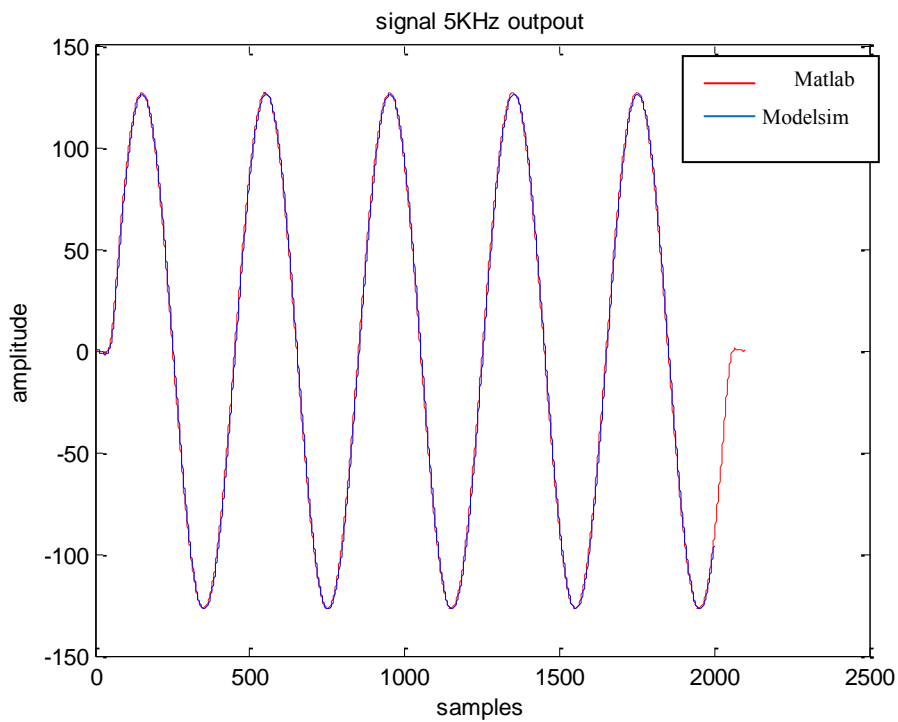


Figure 7.7 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)

Signal 10 KHz

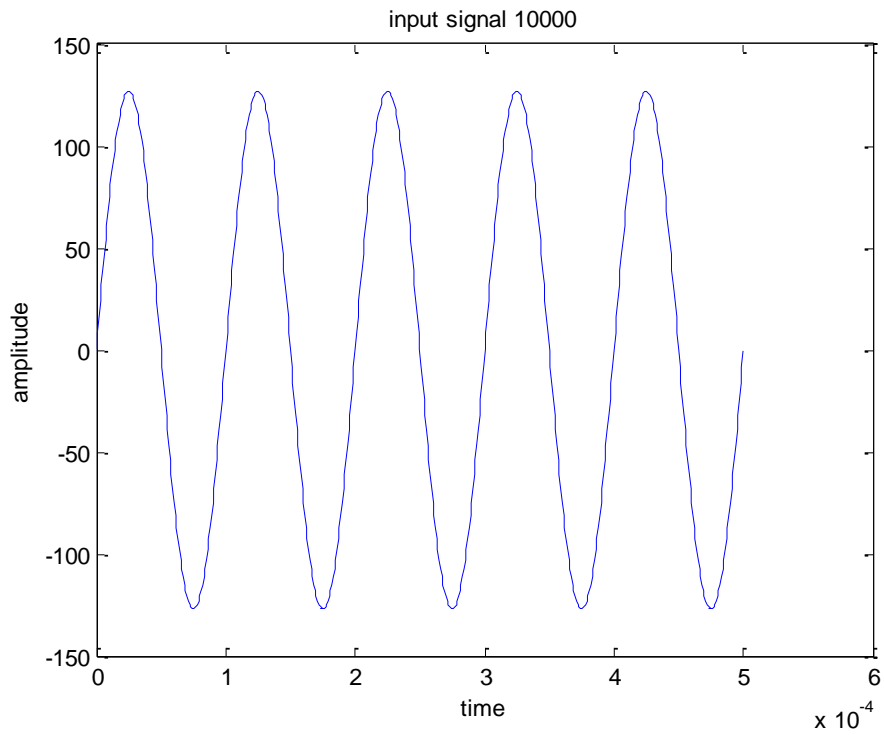


Figure 7.8 Input signal 10 KHz

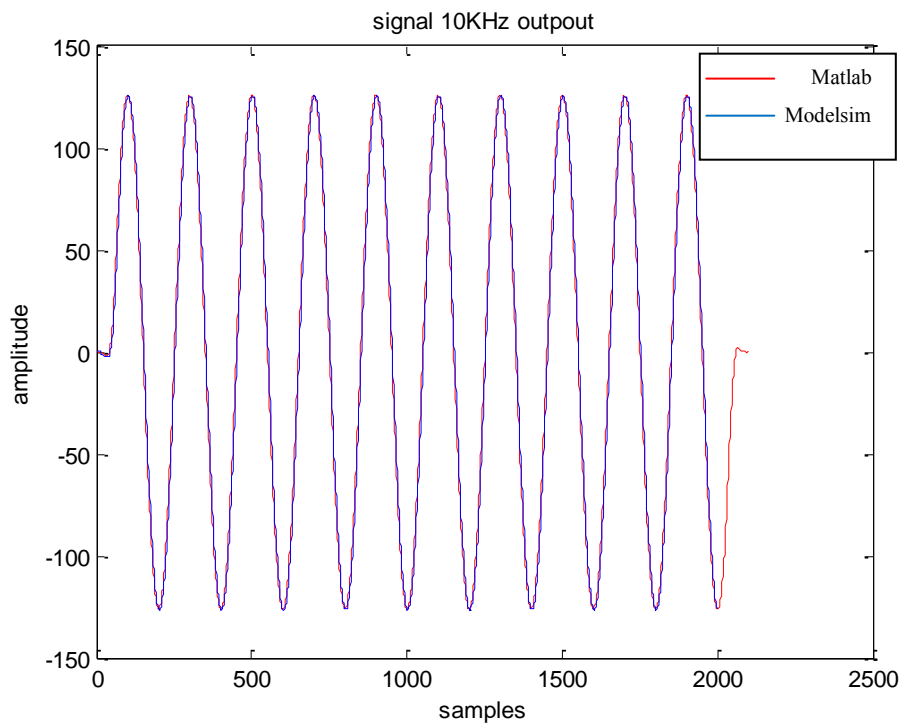


Figure 7.9 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)

Signal 15 KHz

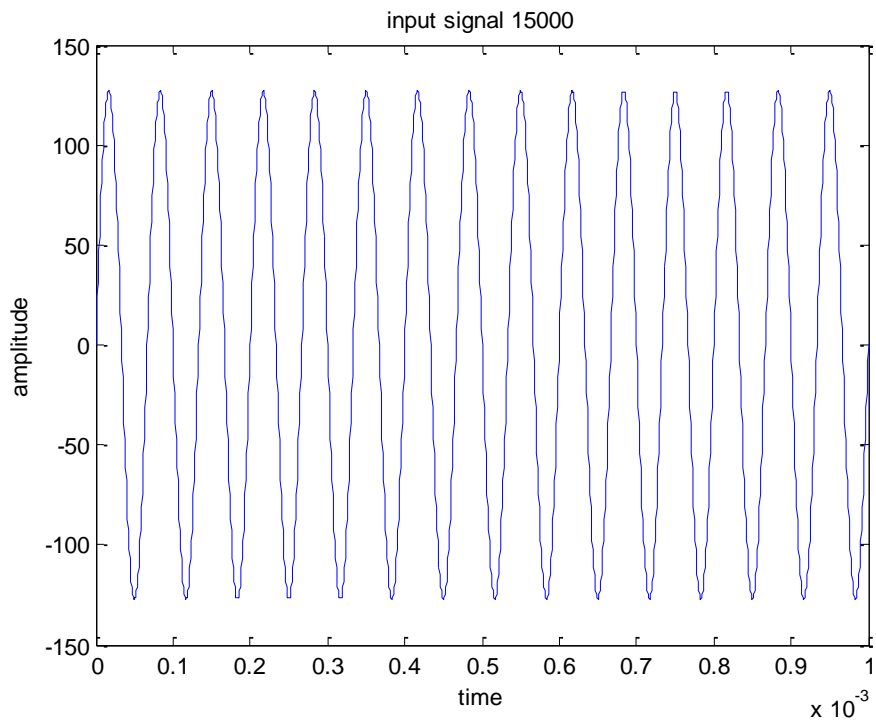


Figure 7.10 Input signal 15 KHz

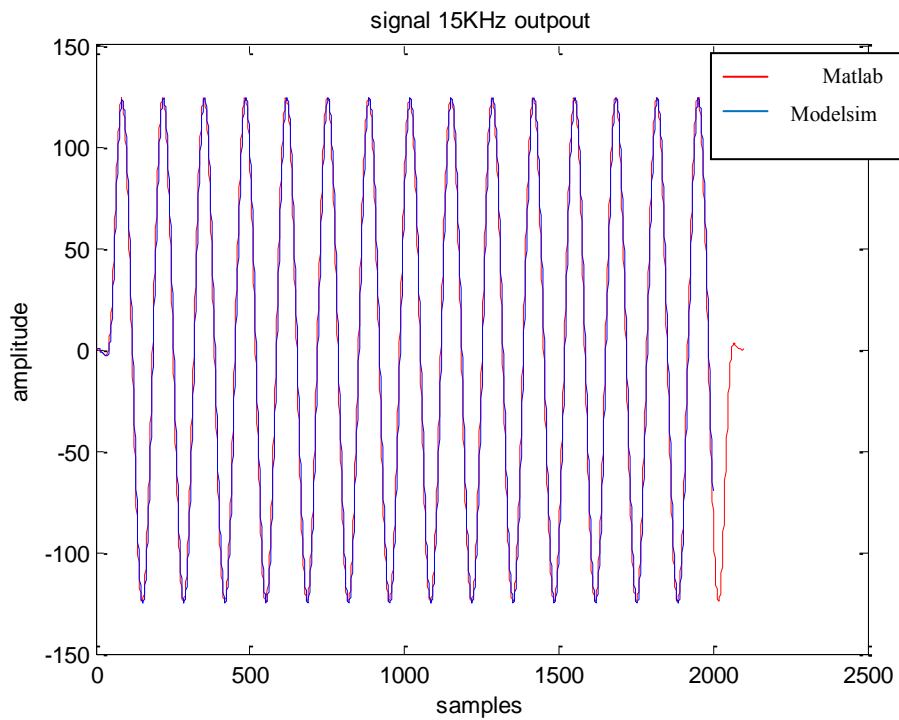


Figure 7.11 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)

Signal 20 KHz

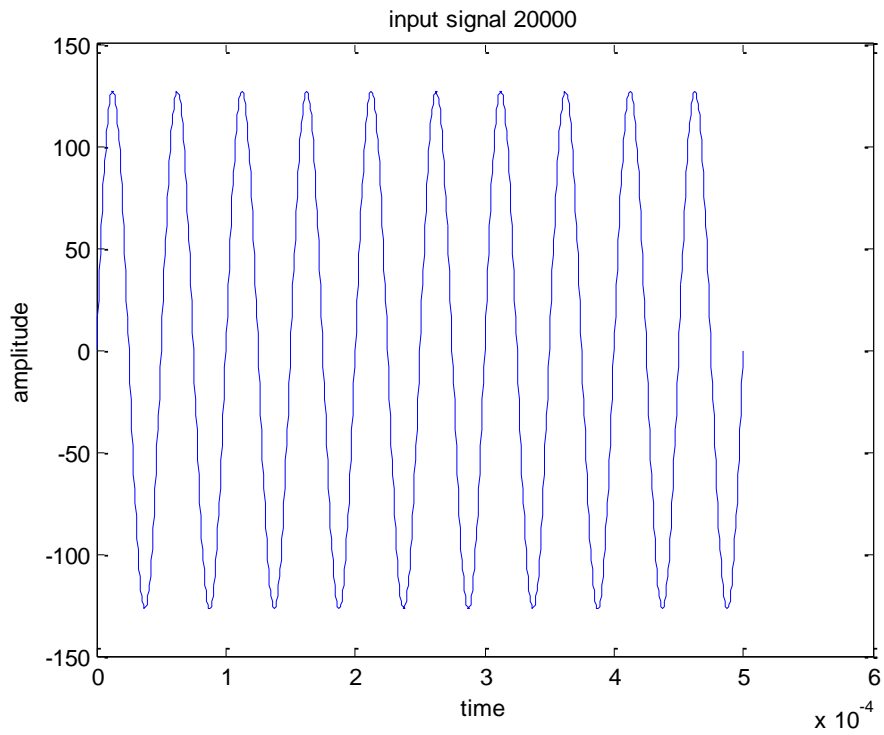


Figure 7.12 Input signal 20 KHz

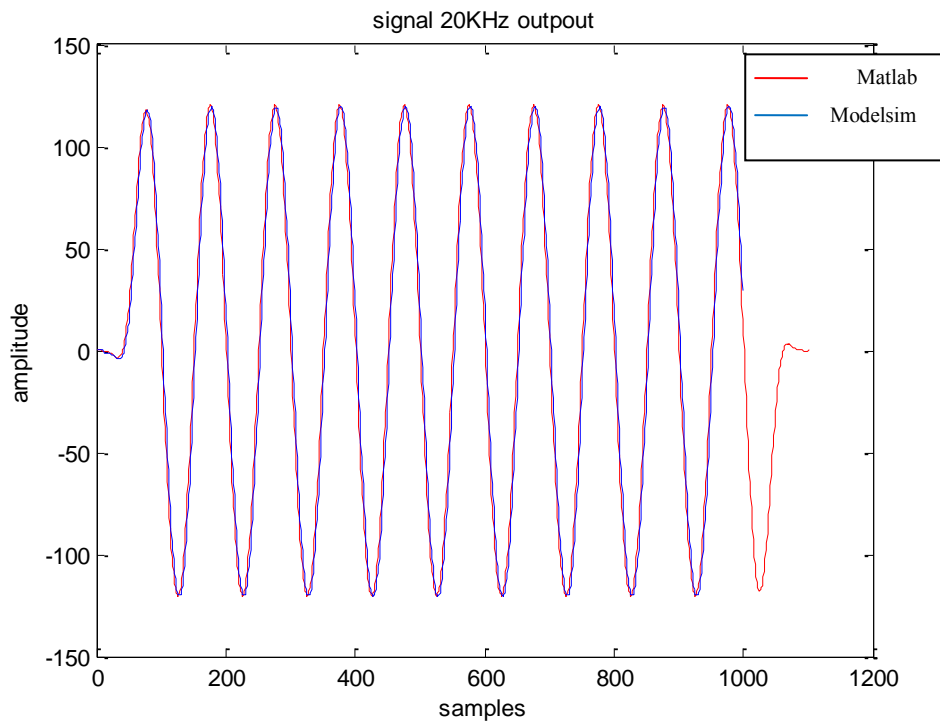


Figure 7.13 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)

Signal 25 KHz

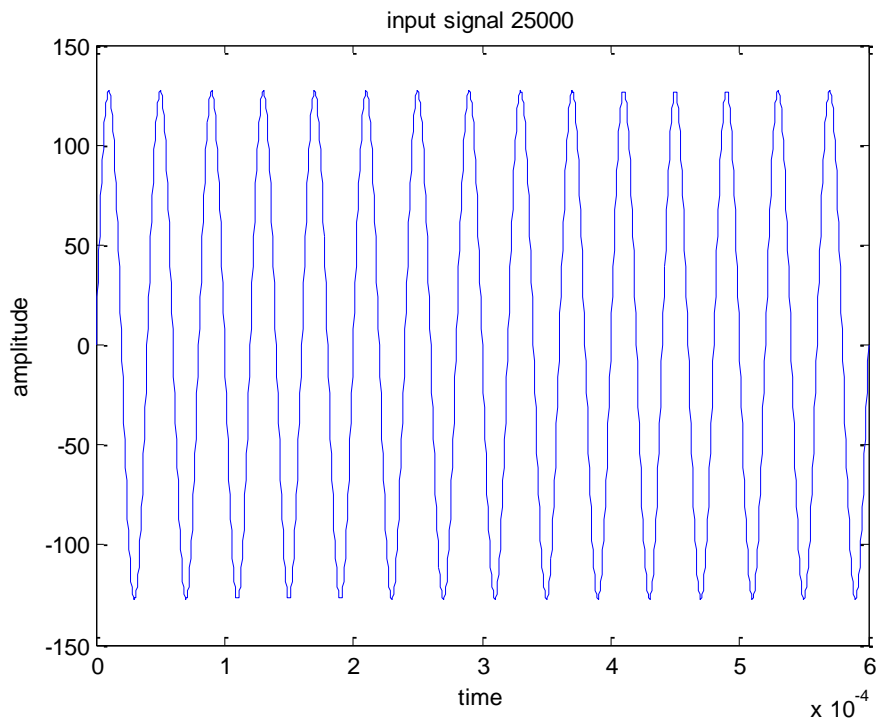


Figure 7.14 Input signal 25 KHz

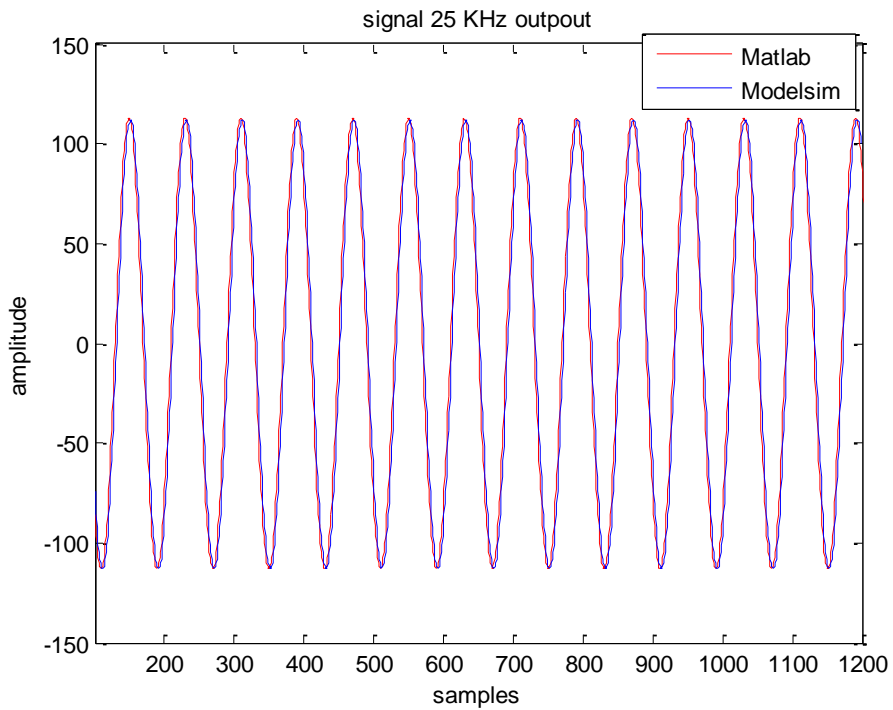


Figure 7.15 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)

Signal 30 KHz

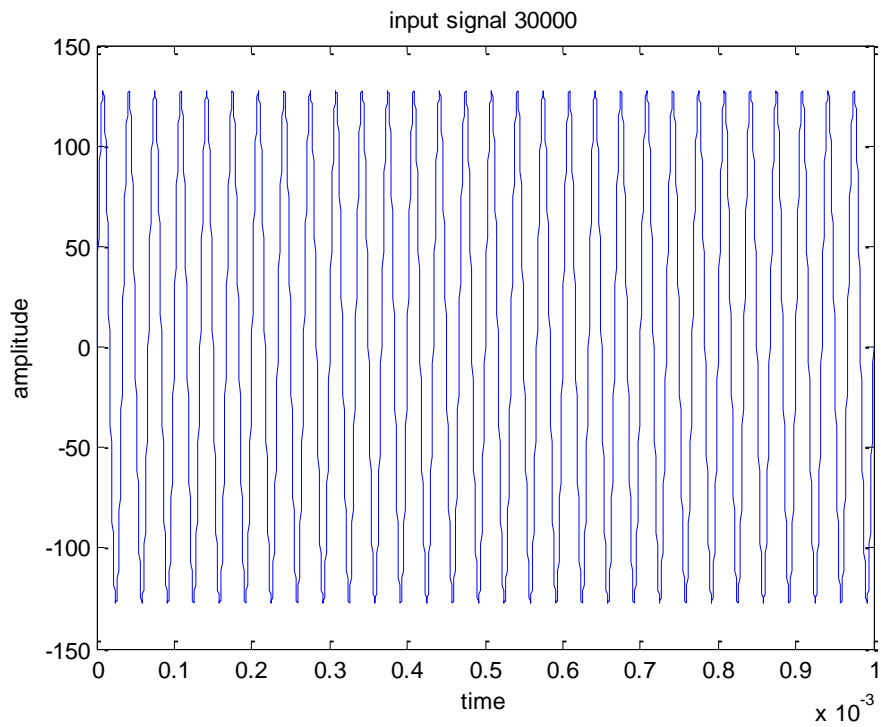


Figure 7.16 Input signal 30 KHz

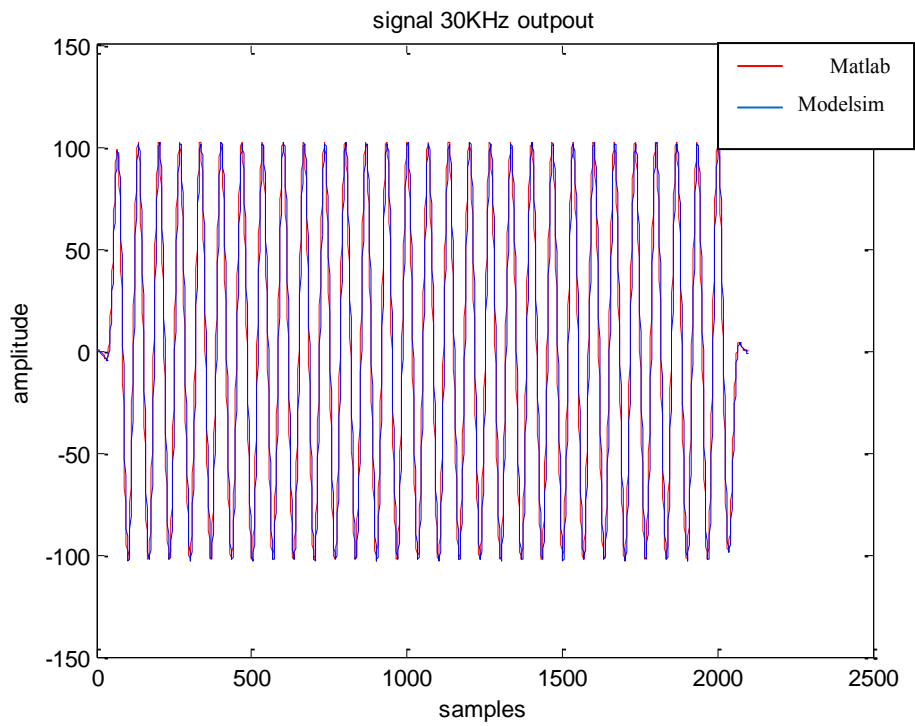


Figure 7.17 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)

Signal 36 KHz

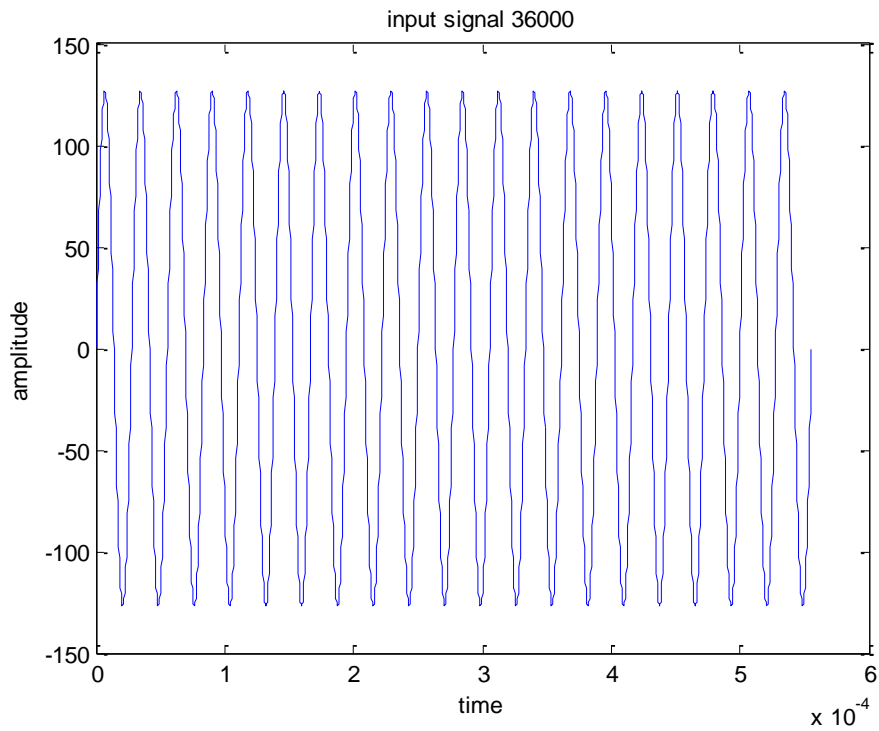


Figure 7.18 Input signal 36 KHz

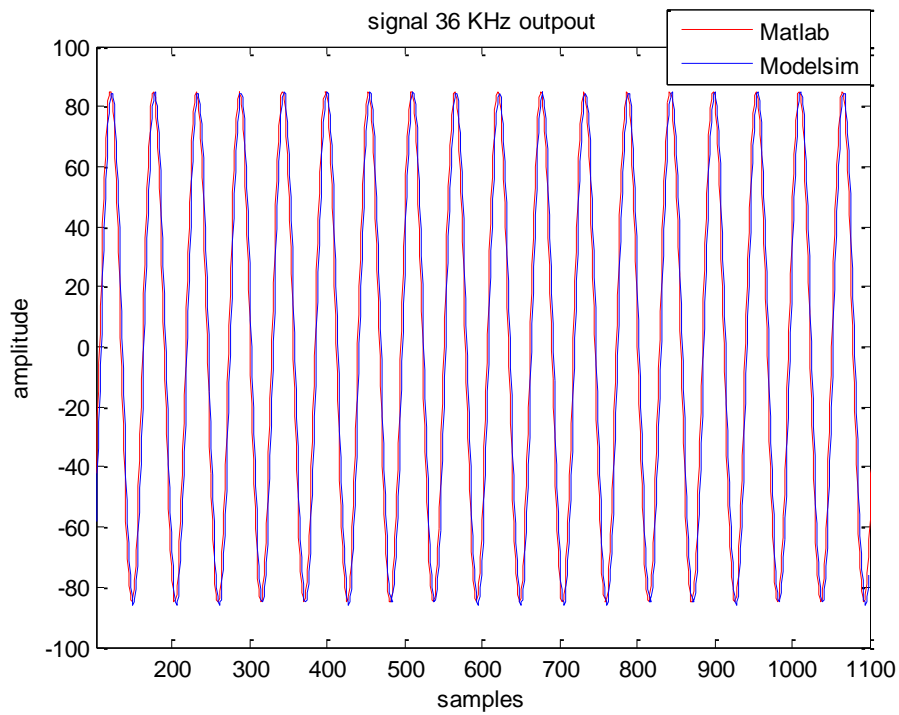


Figure 7.19 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)

Signal 42.6 KHz

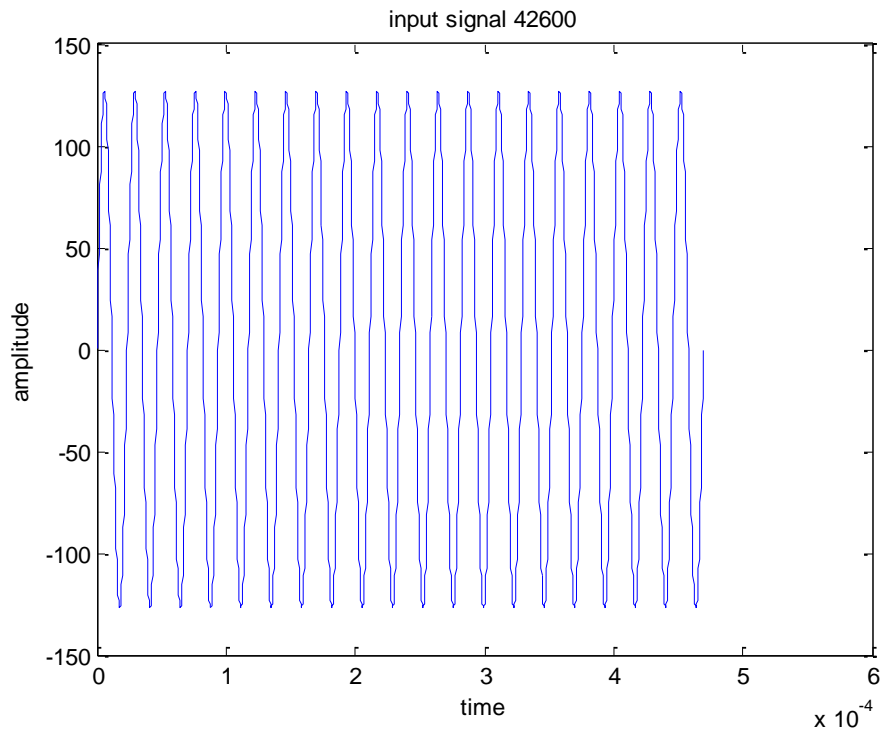


Figure 7.20 Input signal 42.6 KHz

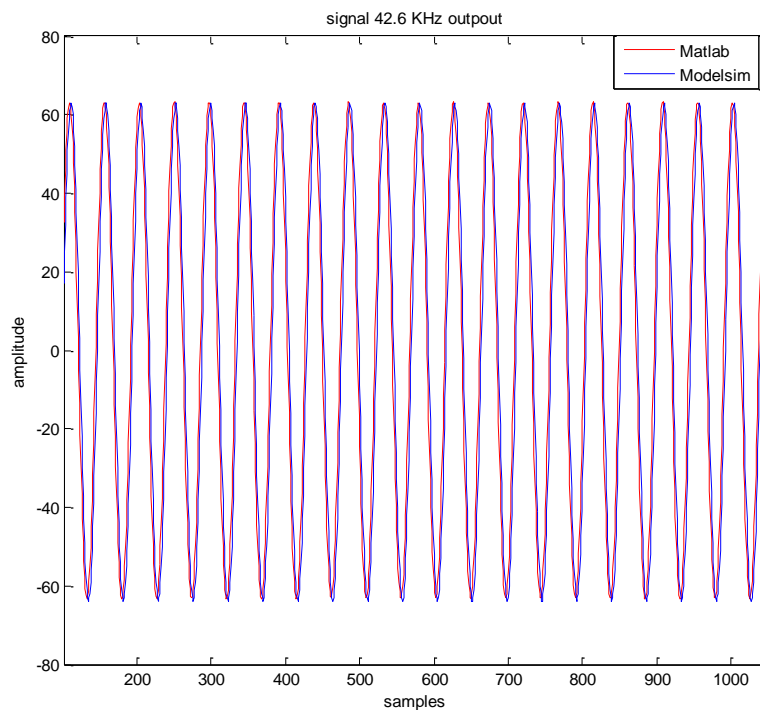


Figure 7.21 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)

Signal 50 KHz

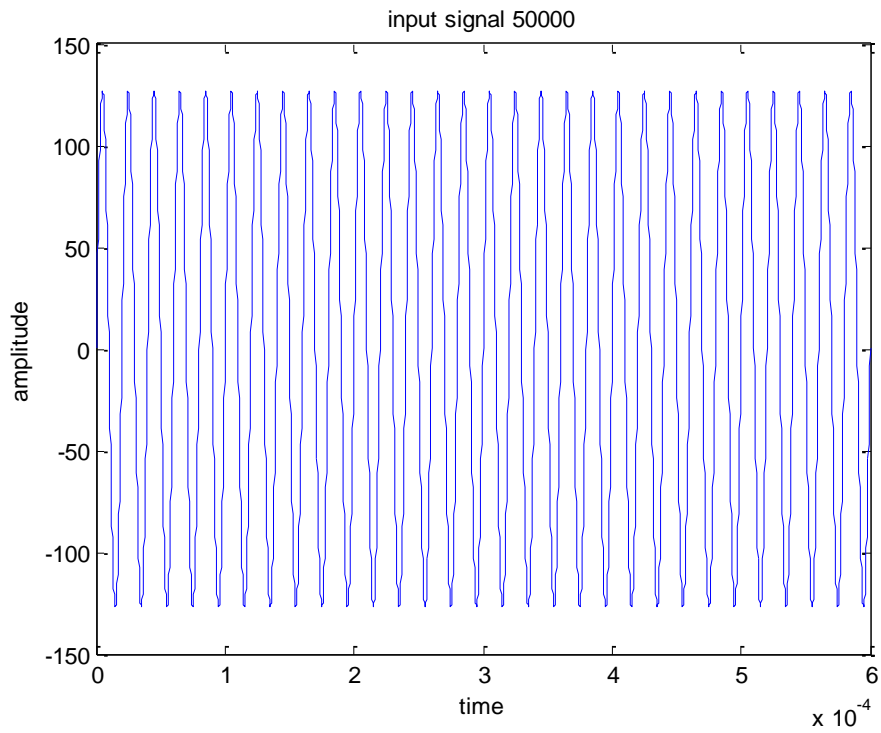


Figure 7.22 Input signal 50 KHz

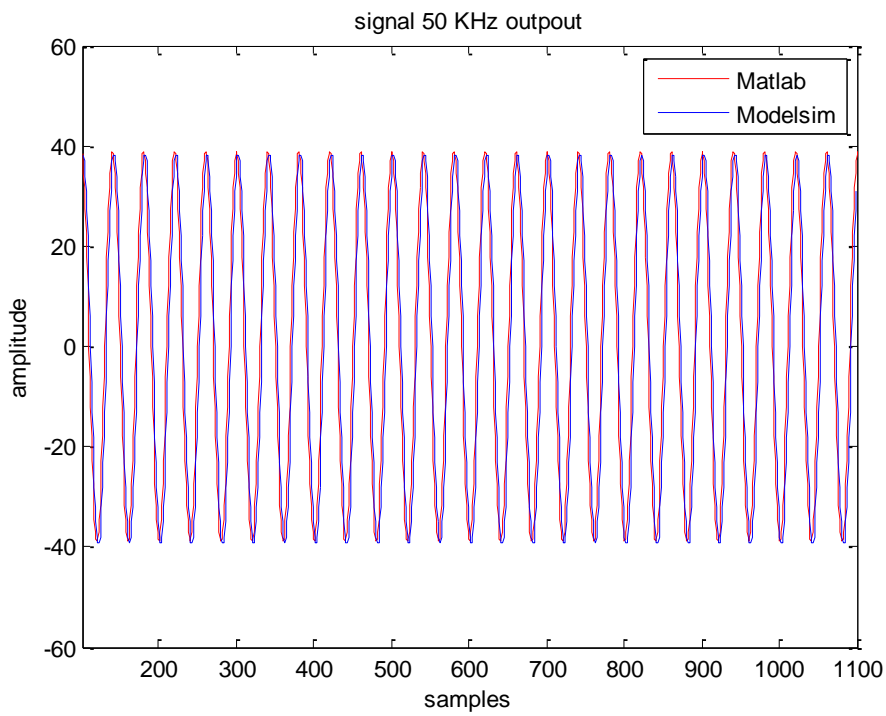


Figure 7.23 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)

Signal 60 KHz

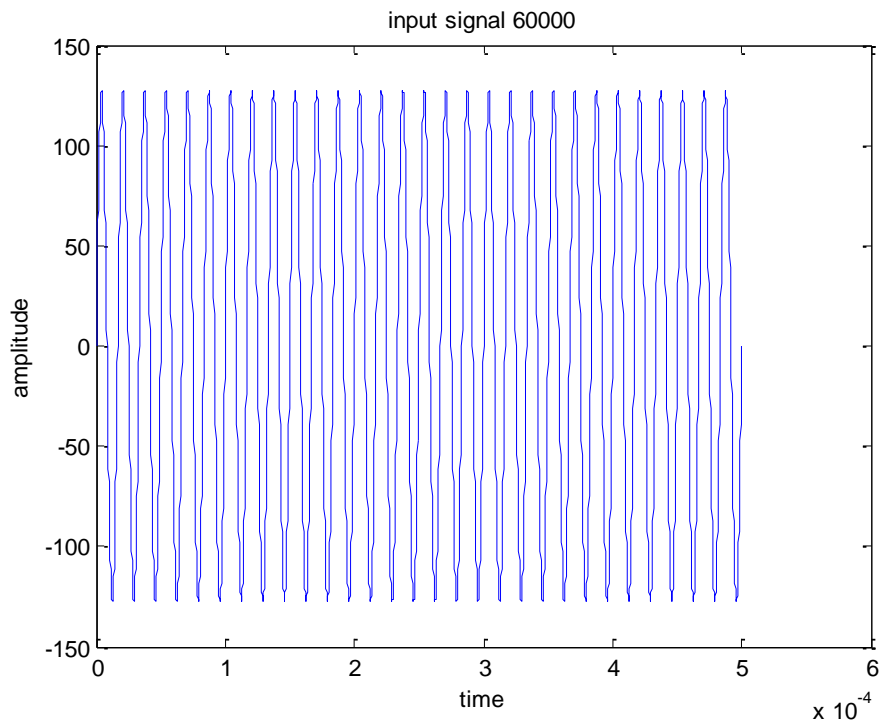


Figure 7.24 Input signal 60 KHz

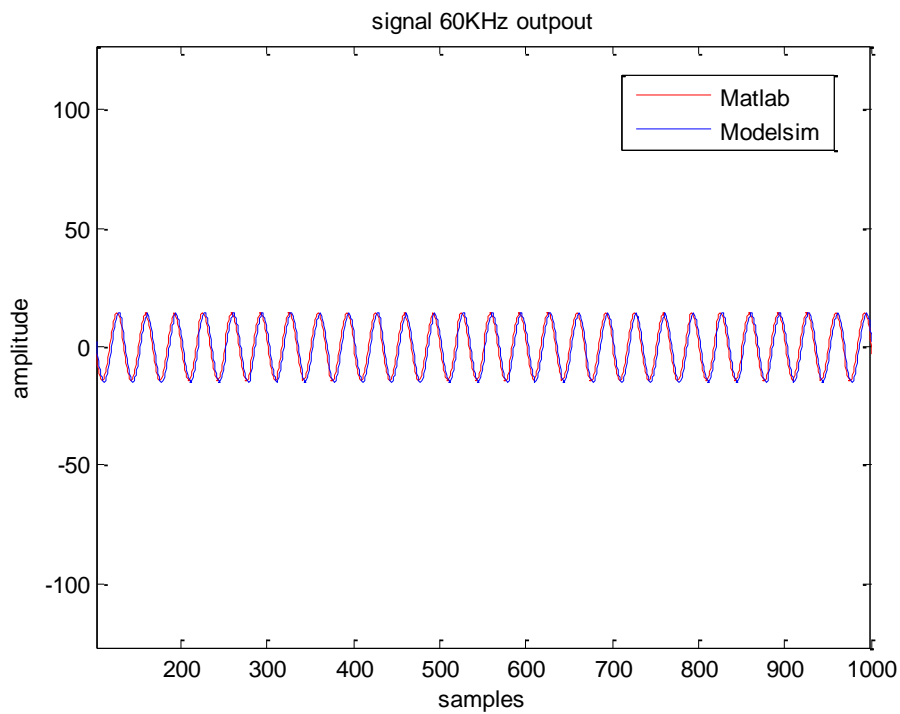


Figure 7.25 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)

Signal 65 KHz

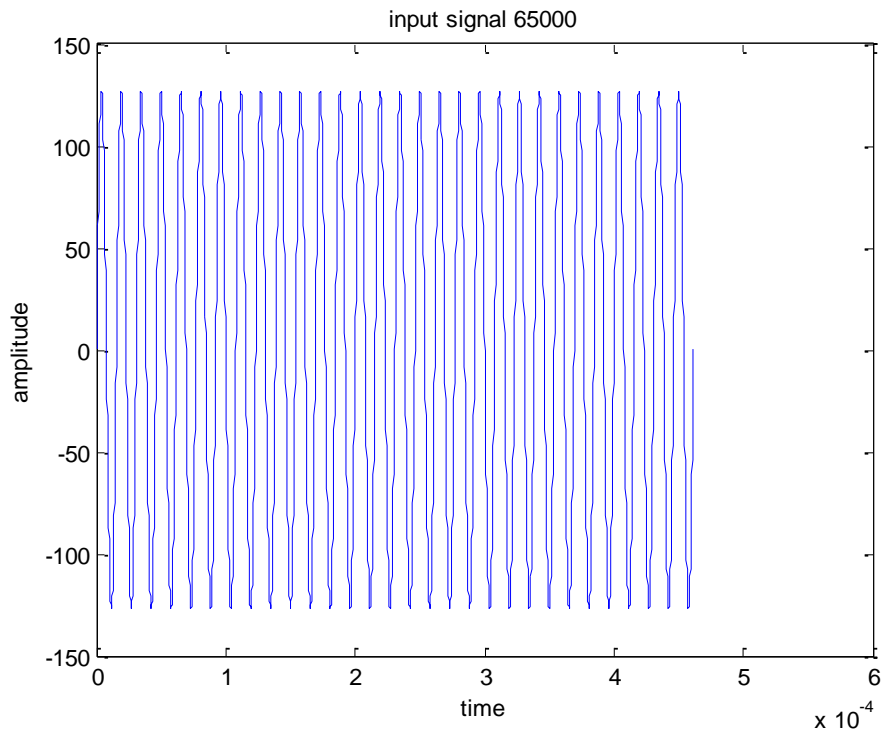


Figure 7.26 Input signal 65 KHz

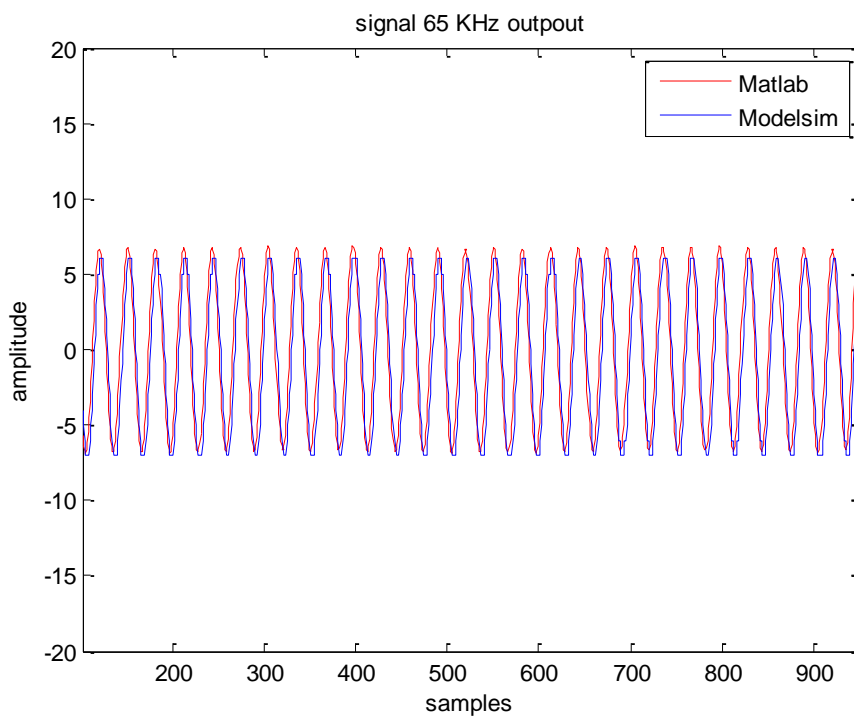


Figure 7.27 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)

Signal 75.2 KHz

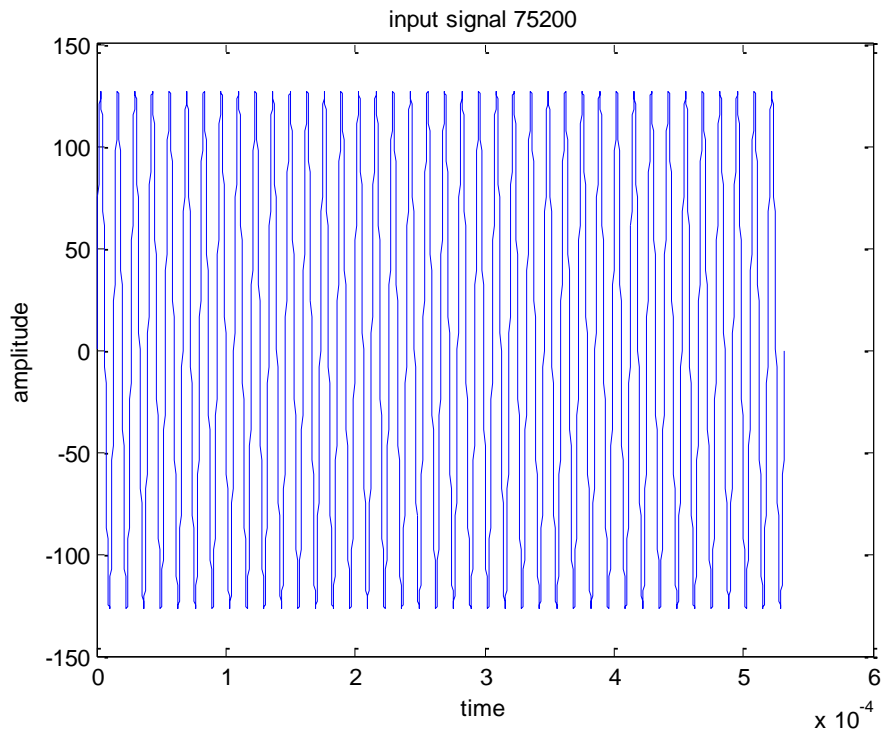


Figure 7.28 Input signal 75.2 KHz

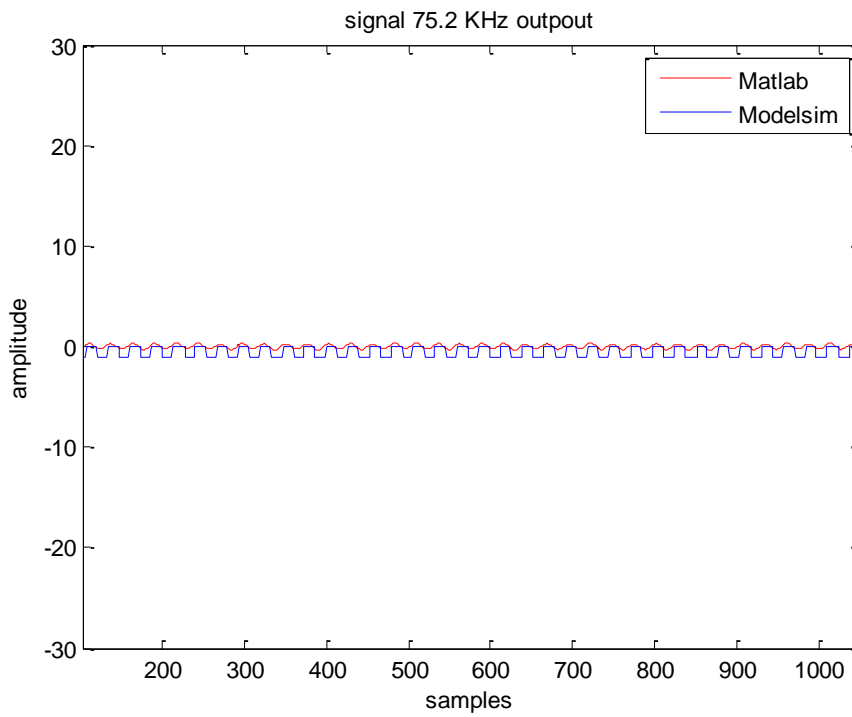


Figure 7.29 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)

Signal 100 KHz

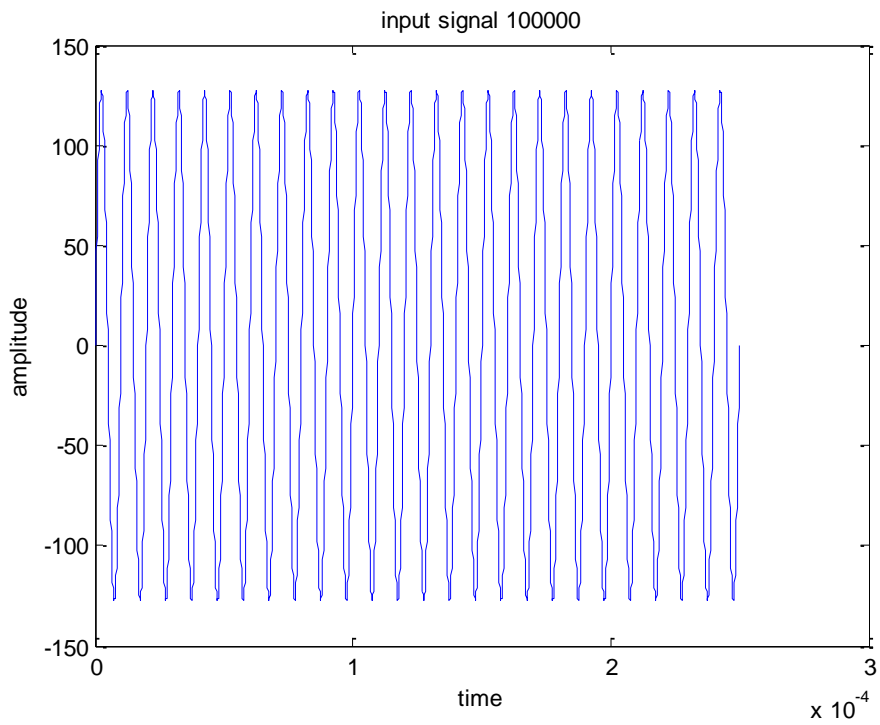


Figure 7.30 Input signal 100 KHz

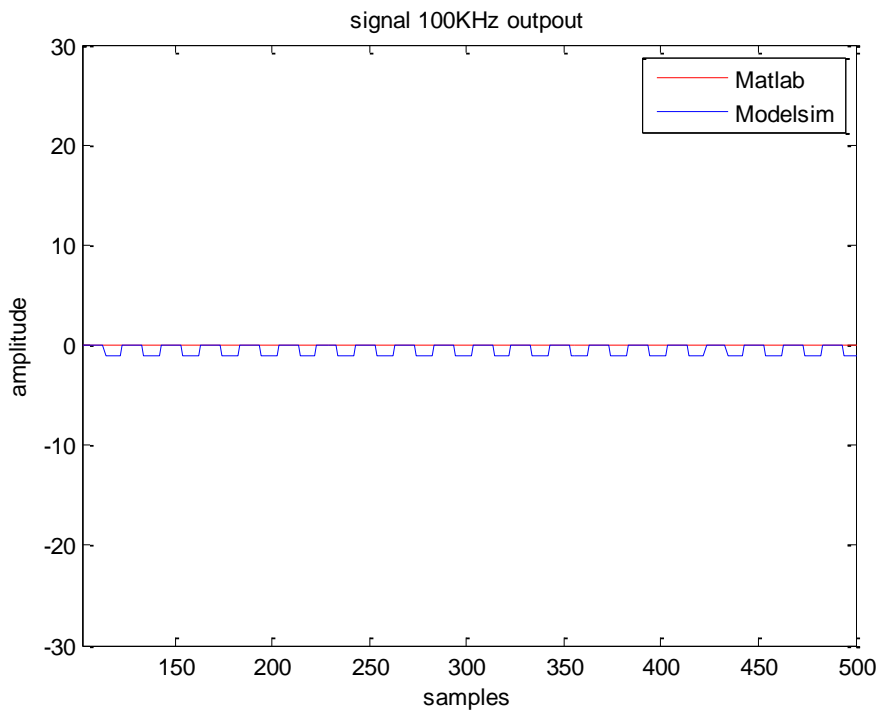


Figure 7.31 Comparison between the filter output based on Matlab implementation and the output from the VHDL hardware model (Modelsim)

1	2	3	Output dB (Matlab)	Output dB (Modelsim)
0.5	127.04	127	0	0
1	127.06	127	0	0
5	126.86	126	-0.009	-0.068
10	126.19	126	-0.055	0.068
15	124.25	124	-0.478	-0.207
20	120.19	120	-0.478	-0.492
25	112.79	112	-1.03	-1.09
30	102.09	102	-1.896	-1.904
36	85.10	85	-3.47	-3.48
42.6	63.05	63	-6.081	-6.089
50	38.95	38	-10.26	-10.48
60	14.16	14	-19.054	-19.15
65	6.79	6	-25.43	-26.51
75.2	0.32	0	-51.862	> -51.8
100	0.02	0	-75.84	>-75.84

Table 7.1 Frequency response using the Matlab software Model and the hardware model, simulated in Modelsim

1. The amplitude of the input signal is kept constant throughout the simulation. It corresponds to the signed 8-bit binary code $A = 127$
2. Amplitude of the output signal, using Matlab and double precision computations.
3. Amplitude of the output signal, using the hardware model simulated in Modelsim.

From table 7.1 we observe the values of the filter output in Matlab and we draw a comparison with the Modelsim results.

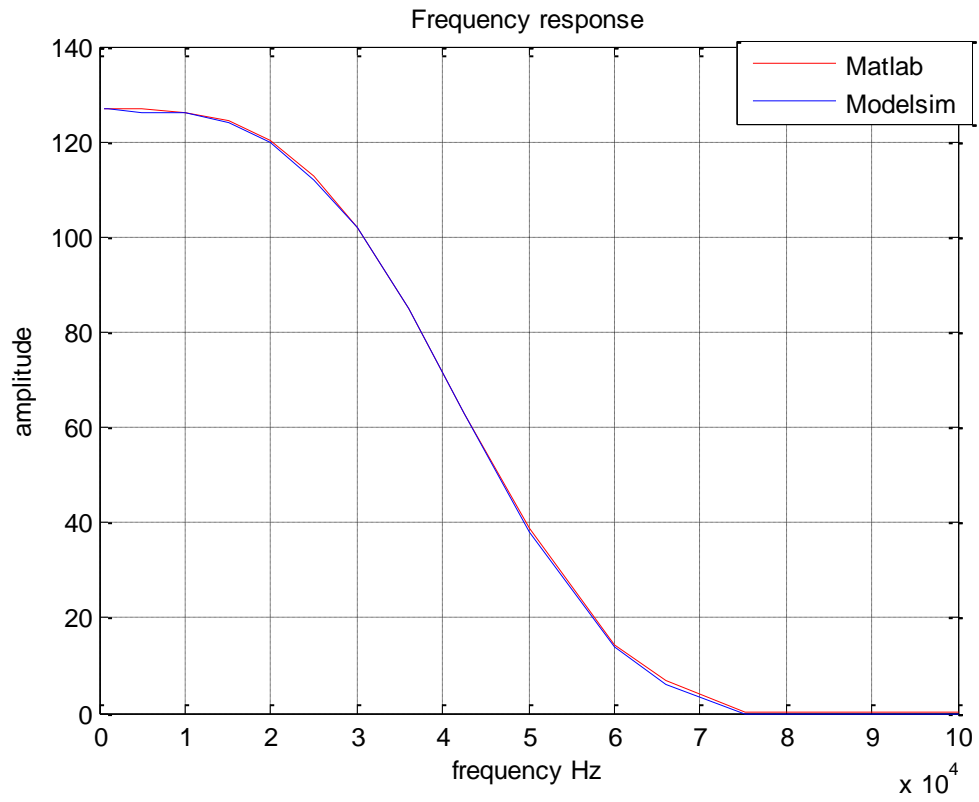


Figure 7.32 Diagram of the filter frequency response

8 Conclusions

8.1 Summary

In this thesis a digital filter was designed by implementing the basic Multiply-Accumulate operation as a hardware component on a chip.

The Graphical User Interface Filter Design and Analysis Tool (FDATool) was used in order to generate the filter coefficients according the specifications.

A software model of the specific filter was implemented using the Matlab programming environment in order to verify the results which were produced by the hardware.

The hardware system was designed in VHDL hardware description language, using the Quartus II by Altera for synthesis and Modelsim by Mentor Graphics for simulation.

8.2 Evaluation

We observe from table 7.1 that the values were calculated in Matlab is slightly larger than those we got at the output of the filter with simulation in Modelsim. This small discrepancy is due to the fixed point computations inherent in the FPGA datapath architecture.

From the figure 7.17 we observe that the filter that we implemented meets the demands we set in paragraph 5.3 and that the filter behavior is in accordance with frequency response depicted in figure 5.4.

The assessment of the design is achieved using as a measure the resource requirements for the overall implementation of the filter in a medium FPGA device, like the Cyclone II EP2C35. Table 8.1 summarizes the recourse requirements for the overall implementation.

Family	Cyclone II
Device	EP2C35F484C6
Total Logic Elements	4,677 / 33,216 (14 %)
Total Registers	816
Total pins	18 / 322 (6 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Table 8.1 Recourse requirements

All resources are from Logic Elements. We do not use embedded multipliers or embedded memory in our design. As depicted in table 8.1 a filter with 101 coefficients can reproduce the original double precision filter specifications using just 14% of the resources of a low FPGA device.

In addition, the timing analysis performed during compilation provides a measure of the performance of the design when it is implemented in the above FPGA device.

The filter that we implemented is an improved direct FIR filter structure compared with the direct form structure which is depicted in figure 8.1.

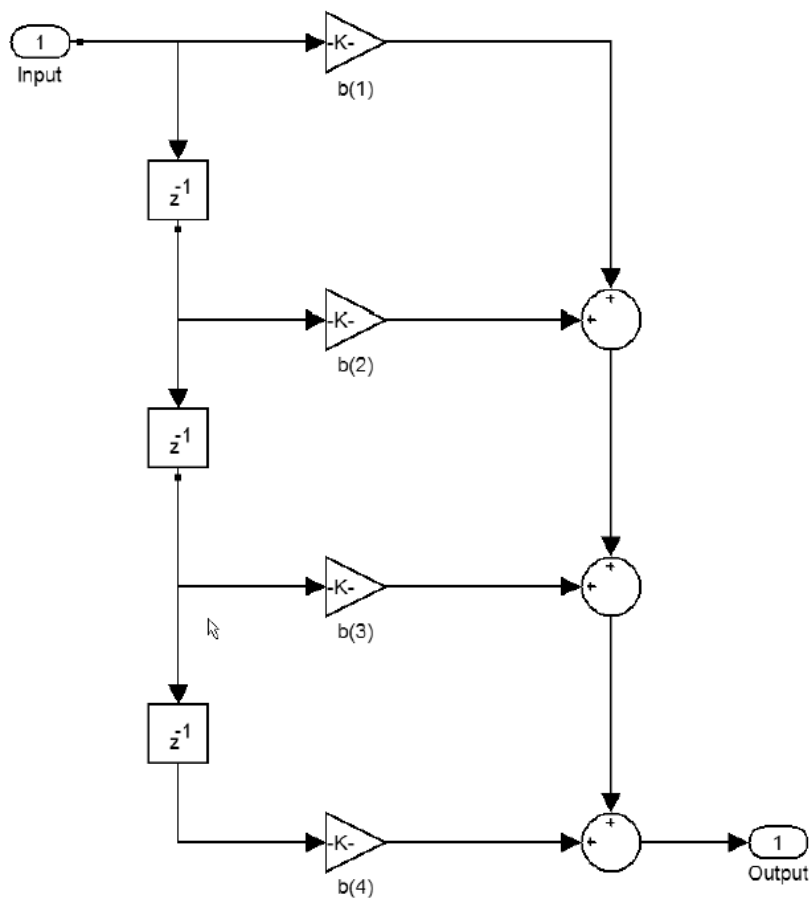


Figure 8.1 A 4-tap filter implemented in direct-form

In order to compare the timing analysis performed during the compilation we wrote code that implements the accumulator as depicted in figure 8.1. The code that was written is:

```
acc:=coef(0)*reg(0);  
FOR i IN 1 TO 101 LOOP  
prod:=coef(i)*reg(i);  
acc:=acc+prod;  
END LOOP;
```

As we can see from this piece of code the implementation of the accumulator has the disadvantage that each adder has to wait for the previous adder to finish before it can compute its result. This structure can achieve a maximum clock frequency of 19 MHz.

The improvement is associated with the design of the accumulator as depicted in figure 8.2. The improved Direct FIR structure now has a ten places array accumulator. The nine first places include 10 elements of the convolution array while the tenth includes 11 elements of the convolution array, thereby succeeding to parallel add the first 5 array elements with the following 5 array elements. This way we manage to parallel add the 10 convolve elements situated in each place of the accumulation array. This structure of the filter can achieve a maximum clock frequency of 50 MHz.

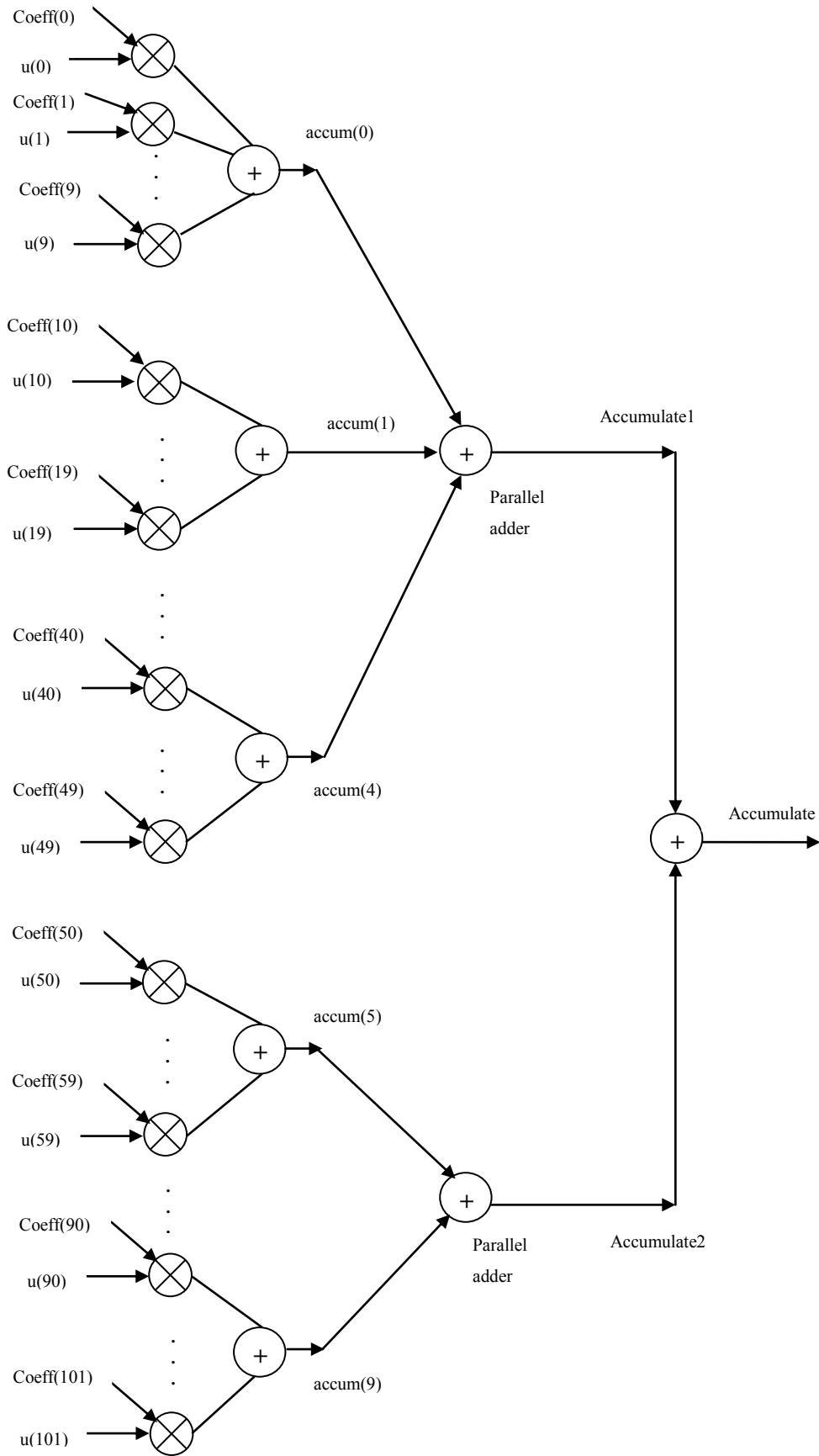


Figure 8.2 parallel accumulator

8.3 Future Work

The symmetry properties intrinsic to a Linear-phase FIR can also be used to reduce the necessary numbers of multipliers. The basic idea for the specific filter is shown in figure 8.1. We must observe that this symmetric architecture has a multiplier budget per filter cycle which is equal to 51. For an even symmetry structure the multiplier budget per filter cycle is exactly the half.

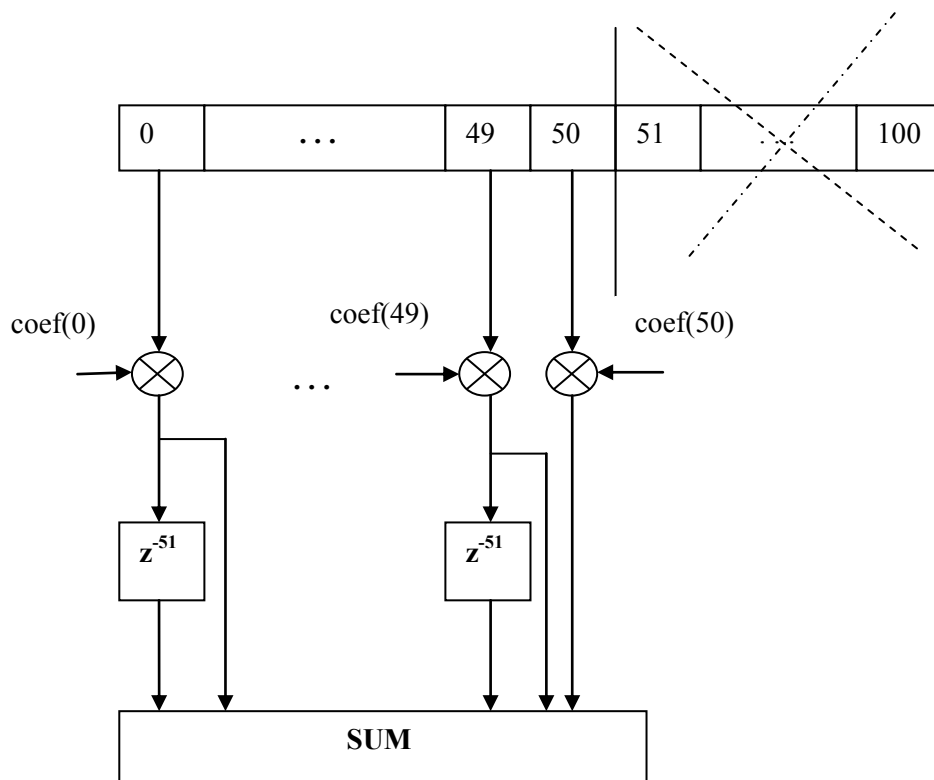


Figure 8.1 Linear phase filter with reduced number of multipliers

REFERENCES

- [1] Volnei A. Pedroni, *Circuit Design and Simulation with VHDL*, Second Edition, The MIT Press London, England, 2010.
- [2] John A. Kalomiros, *Introduction to VHDL Language*, TEI of Central Macedonia, 2012
- [3] Dale Grover and John R. Deller, *Digital Signal Processing and the Microcontroller*, Prentice Hall PTR, 1999.
- [4] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*, Third Edition, Springer, Verlag Berlin Heidelberg, 2007.
- [5] Bernard Sklar, *Digital Communications, Fundamentals and Applications*, Second Edition, Prentice Hall PTR, 2001.
- [6] Analog Devices.
http://www.analog.com/static/imported-files/seminars_webcasts/MixedSignal_Sect6.pdf
- [7] Ricardo A. Losada. http://www.mathworks.com/tagteam/55876_digfilt.pdf
- [8] <http://www.mathworks.com/products/signal/examples.html?file=/products>
- [9] *Introduction to Simulation of VHDL Designs using Modelsim Graphical Waveform Editor*, <http://www.altera.com>

Appendix A VHDL Code

The code that implemented in Hardware Description Language

--MAIN CODE

-----fir-----

LIBRARY ieee;

USE ieee.std_logic_1164.all;

USE ieee.numeric_std.all;

USE work.my_declarations.all;

USE work.components.all;

ENTITY fir IS

 GENERIC(N:INTEGER:=101);

 PORT (clk, rst: IN STD_LOGIC;

 x: IN STD_LOGIC_VECTOR (7 DOWNTO 0);

 y: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));

END fir;

ARCHITECTURE structural OF fir IS

SIGNAL u_1: SIGNED (7 DOWNTO 0);

SIGNAL Z: std_logic_vector (7 DOWNTO 0);

SIGNAL w: vector_coeffs (0 TO 100);

SIGNAL c: window (1 TO N);

BEGIN

u_1<=signed(x); --convert x from std_logic-vector to signed of 8 bits

--nominal mapping

U1: shift_reg PORT MAP (clk=>clk,rst=>rst,shift_inp=>u_1,shift_out=>c);

U2: rom PORT MAP (data=>w);

```
U3: mac PORT MAP (clk=>clk, coeff=>w, u=>c, y_out=>y);  
END structural;
```

--PACKAGES

-----my_declarations-----

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;
```

```
USE ieee.numeric_std.all;
```

PACKAGE my_declarations IS

```
TYPE memory_coeffs IS ARRAY (NATURAL RANGE <>) OF INTEGER RANGE -16384  
TO 16383;--coefficients are signed integers
```

```
TYPE convolve IS ARRAY (NATURAL RANGE <>) OF SIGNED (22 DOWNT0 0);  
--multiplication output is 23 bits
```

```
TYPE maccum IS ARRAY (NATURAL RANGE <>) OF SIGNED (22 DOWNT0 0);  
--accumulation elements array
```

```
TYPE window IS ARRAY (NATURAL RANGE <>) OF SIGNED (7 DOWNT0 0);  
-array of samples as type signed
```

```
TYPE vector_coeffs IS ARRAY (NATURAL RANGE <>) OF SIGNED (14 DOWNT0 0);--  
array of coefficients as type signed
```

```
END my_declarations;
```

-----components-----

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
USE work.my_declarations.all;
```

```
-----
PACKAGE components IS
```

```
-----rom-----
```

```
COMPONENT rom IS
```

```
    PORT (data: OUT vector_coeffs (0 TO 100));
```

```
END COMPONENT;
```

```
-----shift_reg-----
```

```
COMPONENT shift_reg IS
```

```
    GENERIC(N: INTEGER:=101);
```

```
    PORT (clk,rst:IN STD_LOGIC;
```

```
          shift_inp:IN SIGNED (7 DOWNT0 0);
```

```
          shift_out:OUT window (1 TO N));
```

```
END COMPONENT;
```

```
-----mac-----
```

```
COMPONENT mac IS
```

```
    GENERIC(N: INTEGER:=101);
```

```
    PORT(clk:IN STD_LOGIC;
```

```
          coeff:IN vector_coeffs (0 TO 100);
```

```
          u:IN window (1 TO N);
```

```
          y_out: OUT STD_LOGIC_VECTOR (7 DOWNT0 0));
```

```
END COMPONENT;
```

```
-----
END components;
-----
```

--COMPONENTS

-----shift_reg-----

LIBRARY ieee;

USE ieee.std_logic_1164.all;

USE ieee.numeric_std.all;

USE work.my_declarations.all;

ENTITY shift_reg IS

 GENERIC(N: INTEGER:=101);

 PORT (clk,rst:IN STD_LOGIC;

 shift_inp:IN SIGNED (7 DOWNTO 0);

 shift_out:OUT window (1 TO 101));

 END shift_reg;

ARCHITECTURE behavior OF shift_reg IS

BEGIN

 PROCESS(clk,rst)-- at each positive edge of the clock we have a data insertion with their
 concurrent right shift at each time instant

 VARIABLE q>window (0 TO N); --Array q holds the outputs of FFs

 BEGIN

 IF rst='1' THEN

 gen0: FOR i IN 1 TO N LOOP

 q(i):=(OTHERS=>'0');

 END LOOP;

 ELSIF clk'EVENT AND clk='1' THEN

 q(0):=shift_inp;--serial input

```

gen1: FOR i IN N DOWNTO 1 LOOP
    q(i):=q(i-1);
END LOOP;
END IF;
shift_out<=q(1 TO N);--parallel output
END PROCESS;
END behavior;

```

-----mac-----

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
USE work.my_declarations.all;

```

```

ENTITY mac IS
    GENERIC(N: INTEGER:=101);
    PORT(clk:IN STD_LOGIC;
        coeff:IN vector_coeffs (0 TO 100);
        u:IN window (1 TO N);
        y_out:OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END mac;

```

```

ARCHITECTURE behavior OF mac IS
    SIGNAL conv:convolve (0 TO 100);--An ARRAY of 101 elements for convolution
    SIGNAL normalized: SIGNED (22 DOWNTO 0);--23 bits for accumulation
    SIGNAL y_out1: SIGNED (7 DOWNTO 0);
BEGIN

```



```

PROCESS(clk)--Process for multiplications

VARIABLE temp:convolve (0 to 100);

BEGIN

gen1: FOR i IN 1 TO N LOOP
    temp(i-1):=coeff(i-1)*u(i);
END LOOP;

conv<=temp;

END PROCESS;

PROCESS (clk)--Process for accumulation

VARIABLE accum:maccum(0 TO 9);--Holds the result from the addition 10 conv ele-
ments

VARIABLE accumulate, accumulate1, accumulate2:signed (22 DOWNT0 0);

BEGIN

IF clk'EVENT AND clk='1' THEN

-- nullify variables accumulate at each positive edge of the clock

accumulate:=(OTHERS=>'0');

accumulate1:=(OTHERS=>'0');

accumulate2:=(OTHERS=>'0');

accum(0):=conv(0)+conv(1)+conv(2)+conv(3)+conv(4)+conv(5)+conv(6)+conv(7)+conv(8)
+conv(9);

accum(1):=conv(10)+conv(11)+conv(12)+conv(13)+conv(14)+conv(15)+conv(16)+conv(17)
+conv(18)+conv(19);

accum(2):=conv(20)+conv(21)+conv(22)+conv(23)+conv(24)+conv(25)+conv(26)+conv(27)
+conv(28)+conv(29);

accum(3):=conv(30)+conv(31)+conv(32)+conv(33)+conv(34)+conv(35)+conv(36)+conv(37)

```

```

+conv(38)+conv(39);

accum(4):=conv(40)+conv(41)+conv(42)+conv(43)+conv(44)+conv(45)+conv(46)+conv(47)
+conv(48)+conv(49);

accum(5):=conv(50)+conv(51)+conv(52)+conv(53)+conv(54)+conv(55)+conv(56)+conv(57)
+conv(58)+conv(59);

accum(6):=conv(60)+conv(61)+conv(62)+conv(63)+conv(64)+conv(65)+conv(66)+conv(67)
+conv(68)+conv(69);

accum(7):=conv(70)+conv(71)+conv(72)+conv(73)+conv(74)+conv(75)+conv(76)+conv(77)
+conv(78)+conv(79);

accum(8):=conv(80)+conv(81)+conv(82)+conv(83)+conv(84)+conv(85)+conv(86)+conv(87)
+conv(88)+conv(89);

accum(9):=conv(90)+conv(91)+conv(92)+conv(93)+conv(94)+conv(95)+conv(96)+conv(97)
+conv(98)+conv(99)+conv(100);

gen2: FOR i IN 0 TO 4 LOOP--add 5 accum
    accumulate1:=accumulate1+accum(i);
END LOOP;

gen3: FOR i IN 5 TO 9 LOOP--add 5 accum
    accumulate2:=accumulate2+accum(i);
END LOOP;

accumulate:=accumulate1+accumulate2; --add all

END IF;

normalized<=accumulate SRL 15; -- Normalize by dividing 2^15

END PROCESS;

```

```

y_out1<=normalized (7 DOWNT0 0);
y_out<=std_logic_vector (y_out1);--Convert signed y_out1 to type std_logic_vector
END behavior;

```

-----rom-----

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
USE work.my_declarations.all;

```

```

ENTITY rom IS
    PORT (data: OUT vector_coeffs (0 TO 100));
END rom;

```

```

ARCHITECTURE behavior OF rom IS
--define 101 coefficients in the range -16384 TO 16383 15 bits signed
CONSTANT memory: memory_coeffs :=(7,5,3,0,-3,-6,
-10,-15,-21,-27,-35,-43,-52,-61,-71,-80,-89,-96,
-102,-105,-106,-103,-96,-84,-68,-46,-17,17,58,105,
159,219,284,355,430,509,590,674,758,841,922,1001,
1075,1144,1206,1260,1306,1343,1369,1385,1391,1385,
1369,1343,1306,1260,1206,1144,1075,1001,922,841,758,
674,590,509,430,355,284,219,159,105,58,17,-17,-46,-68,
-84,-96,-103,-106,-105,-102,-96,-89,-80,-71,-61,-52,-43,
-35,-27,-21,-15,-10,-6,-3,0,3,5,7);

```

```

BEGIN

```

```
gen1: FOR i IN 0 TO 100 GENERATE
```

```
  data(i) <= to_signed(memory(i), 15); --Convert Integer coefficients to type signed
```

```
END GENERATE gen1;
```

```
END behavior;
```

Appendix B ModelSim Tutorial

This tutorial provides an introduction to ModelSim from Mentor Graphics, which is a simulator for VHDL (and other) designs. It shows how the simulator can be used to perform functional simulation of a circuit using VHDL. This tutorial is based on ModelSim 6.4.a starter edition for Altera devices which is available free of charge at www.altera.com.

The tutorial is divided in four parts.

- B.1 Introduction
- B.2 Creating a Project
- B.3 Compiling Project Files
- B.4 Running a Functional Simulation

B.1 Introduction

The circuit that it will be used in this tutorial is a four stage shift register which is shown in figure B.1. In a shift register the output y is four positive clock edges behind the input bit d (is a single bit shift register). To perform a functional simulation two files must be created by the user: a *design file* (here called `shift_register.vhd`) and a *Test file* (here called `shift_register_tb.vhd`) which contains the testbench.

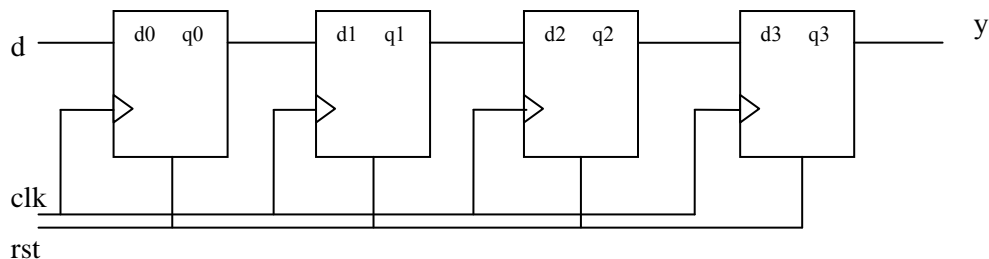


Figure B.1

B.2 Creating a Project

- a. Start ModelSim.
- b. After starting ModelSim we should see the *Welcome to version 6.4.a* window (see Figure B.2). If the window does not show up we can display it by selecting *Help>Welcome Menu* from the main window.

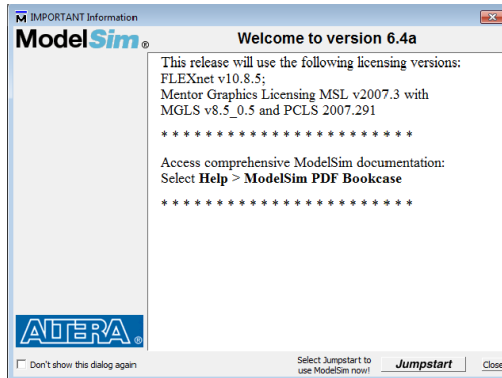


Figure B.2

- c. Click on Jumpstart on the Welcome to ModelSim window and then Create a Project (see Figure B.3).

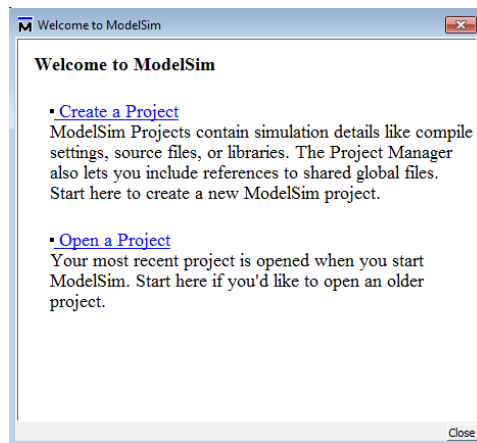


Figure B.3

- d. A Create Project pop up box will appear (see Figure B.4). Select an appropriate name for your project (shift_register); set the Project location as shown and leave the Default Library Name to work. Click OK.

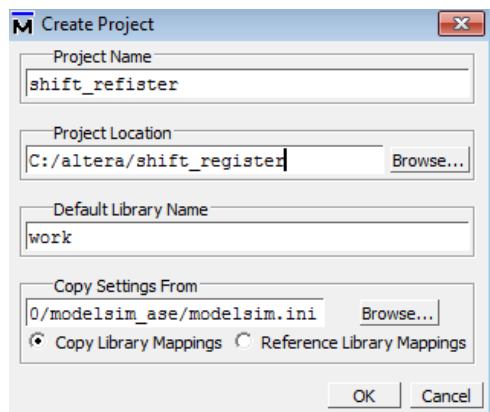


Figure B.4

- e. An ADD items to the Project dialog pops up (see Figure B.5). Click on Create New File.

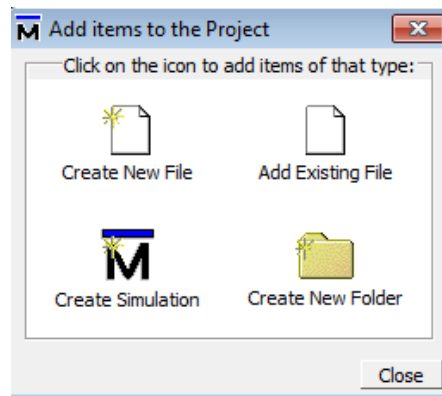


Figure B.5

- f. A Create Project File dialog pops out (see Figure B.6). Select an appropriate File Name (shift_register) for the file; choose the VHDL as the add file as type and Top Level as Folder option.

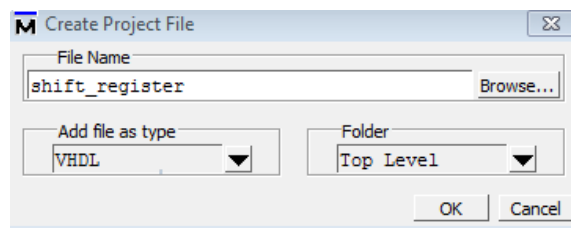


Figure B.6

- g. On the workspace section of the main window, double click on the file that has just created (see Figure B.7).

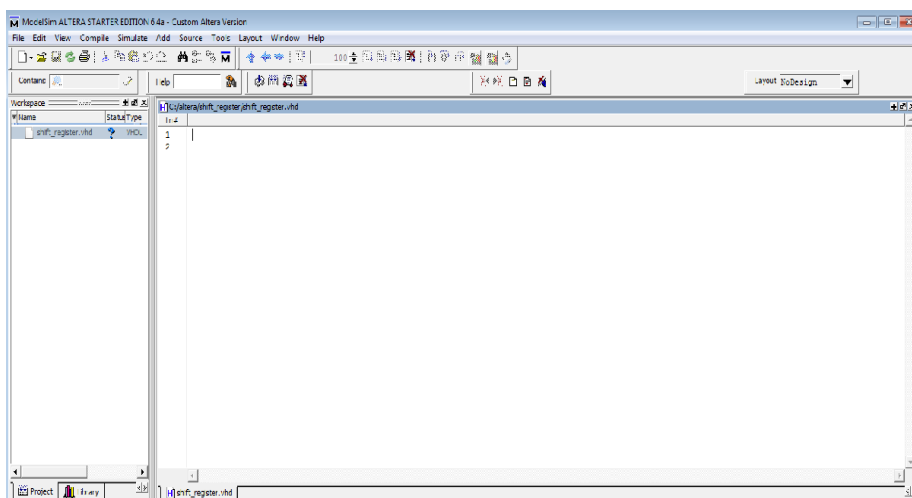


Figure B.7

h. Type in the code below in the new Window.

```
1 -----shift_register-----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY shift_register IS
6     PORT ( clk, rst : IN STD_LOGIC;
7           d: IN STD_LOGIC;
8           y: OUT STD_LOGIC);
9 END shift_register;
10 -----
11 ARCHITECTURE behavior OF shift_register IS
12     SIGNAL q : STD_LOGIC_VECTOR ( 0 TO 3);
13 BEGIN
14     PROCESS (clk, rst)
15     BEGIN
16         IF (rst='1') THEN
17             q <= (OTHERS=>'0');
18         ELSIF (clk'EVENT AND clk='1') THEN
19             q<=d & q(0 TO 2);
20         END IF;
21     END PROCESS;
22     y<=q(3);
23 END behavior;
```

i. From the main window select File>Save to save the file.

- j. Now we will add to the Project a new file. This is a Test File which contains the testbench. From the main window select the File in the workspace right click>Add to Project>New File.
- k. A Create Project File dialog pops up (see Figure B.8). Select an appropriate File Name (shift_register_tb) for the file; choose the VHDL as the add file as type and Top Level as Folder option. Click OK

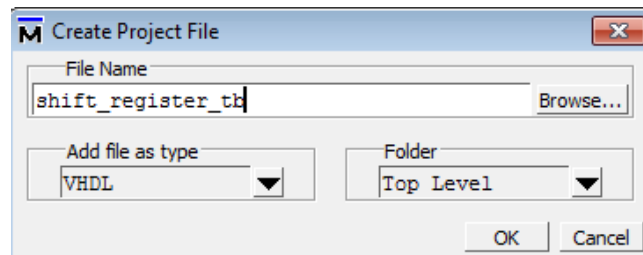


Figure B.8

- l. Repeat steps g-h and leave out the last semicolon of code below. We will see later that the compiler will illustrate an error. After writing the code save the file (step i).

```

1 -----shift_register_tb-----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY shift_register_tb IS
6 END shift_register_tb;
7 -----
8 ARCHITECTURE behavior OF shift_register_tb IS
9     COMPONENT shift_register IS
10     PORT (clk: IN STD_LOGIC;
11           rst: IN STD_LOGIC;
12           d : IN STD_LOGIC;
13           y: OUT STD_LOGIC);
14     END COMPONENT;
15     SIGNAL clk_tb: STD_LOGIC:= '0';

```

```

16     SIGNAL rst_tb: STD_LOGIC:='1';
17     SIGNAL d_tb:  STD_LOGIC:='0';
18     SIGNAL y_tb:  STD_LOGIC;
19 BEGIN
20     dut: shift_register
21     PORT MAP (clk=>clk_tb, rst=>rst_tb, d=>d_tb, y=>y_tb);
22     clk_tb<= NOT clk_tb AFTER 30ns;
23     rst_tb<='0' AFTER 5ns;
24     d_tb <= '1' AFTER 150ns, '0' AFTER 220ns, '1' AFTER 300ns;
25 END behavior;
26 -----

```

B.3 Compiling Project Files

At this point the main ModelSim window will include the files as indicated in Figure B.9. Observe that there is a question mark in the Status column of the workspace.

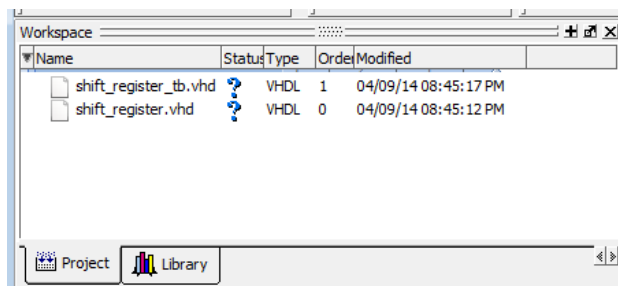


Figure B.9

- From the main window select **Compile>Compile all**.
- The compilation results are shown on the main window. A red message indicates that there is an error in our code (see Figure B.10).

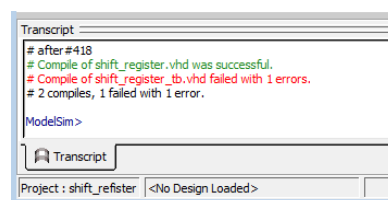


Figure B.10

- c. Double click on the error on the main window. This will open a new window which describes the nature of the error (see Figure B.11).

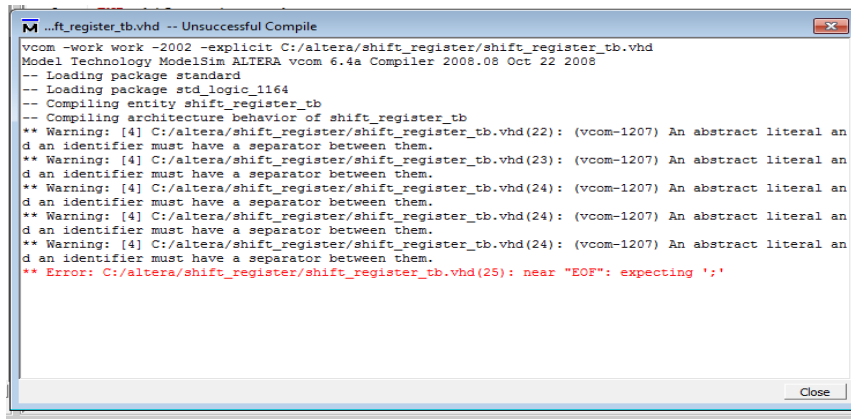


Figure B.11


- d. Double click on the Error message and the error is highlighted in the source window (see Figure B.12).

```

18     SIGNAL y_tb:    STD_LOGIC;
19     BEGIN
20         dut: shift_register
21         PORT MAP (clk=>clk_tb, rst=>rst_tb, d=>d_tb, y=>y_tb);
22         clk_tb<= NOT clk_tb AFTER 30ns;
23         rst_tb<='0' AFTER 5ns;
24         d_tb <= '1' AFTER 150ns, '0' AFTER 220ns, '1' AFTER 300ns;|
25     END behavior

```



Figure B.12

- e. Correct the above error by adding a semicolon after the END behavior statement. Click  and recompile the file. Repeat the steps above until the code compiled with no errors. Observe that in the Transcript window (at the bottom) the files have compiled successfully. Note that this is also indicated by a check mark in the Status column of the workspace.

B.4 Running a Functional Simulation

To perform simulation of the designed circuit it is necessary to enter the simulation mode.

- a. From the main window Select **Simulate>Start Simulation**. This leads to the window in Figure B.13.
- b. Expand the work directory and select the design called shift_register_tb as shown in the Figure B.13. Then click OK.

- c. When the process ends Figure B.14 is displayed in the *sim* tab of the workspace. Right Click *shift_register_tb* and select *Add>Add to Wave>All items in region*.
- d. The wave pane of Figure B.15 will then be exhibited but as you can see without the waveforms.
- e. Set the simulation time interval by selecting *Simulate>Runtime Options*. Enter the value 700ns (see Figure B.16).
- f. To run the simulation select *Simulate>Run>Run 100*. Alternative click . The waveforms of Figure B.17 will then be exhibited.
- g. Click the Restart icon  to clean the waveforms window.

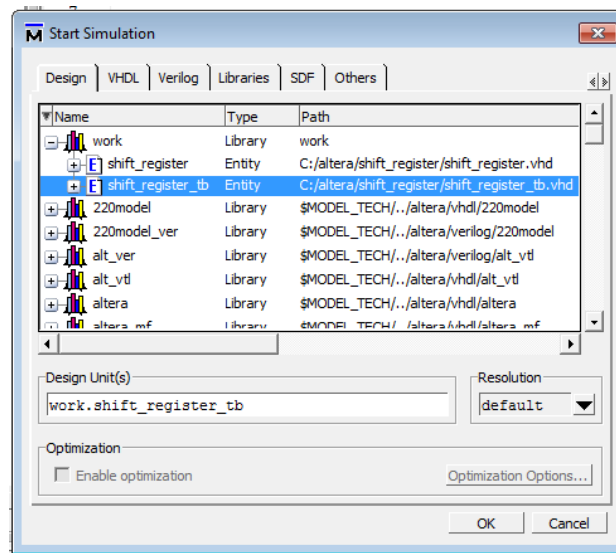


Figure B.13

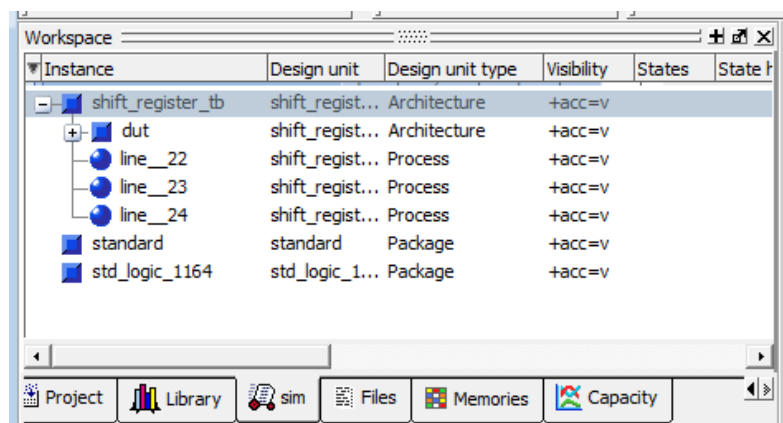


Figure B.14

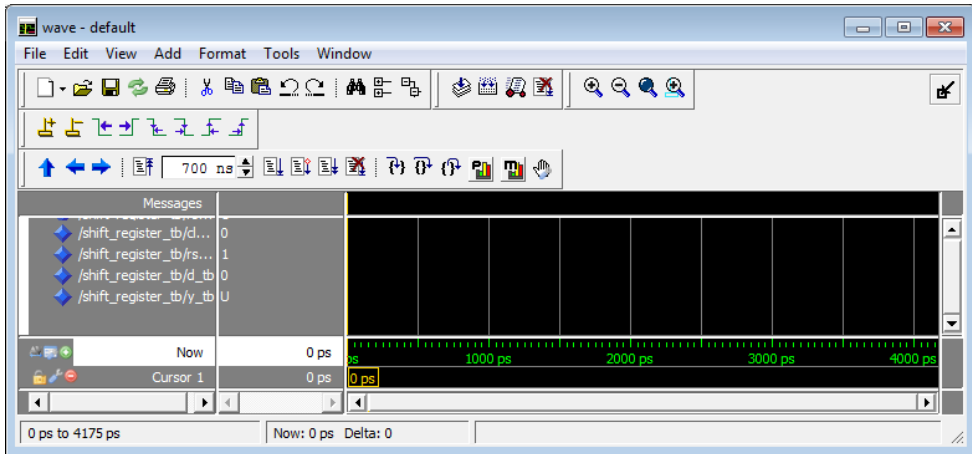


Figure B.15

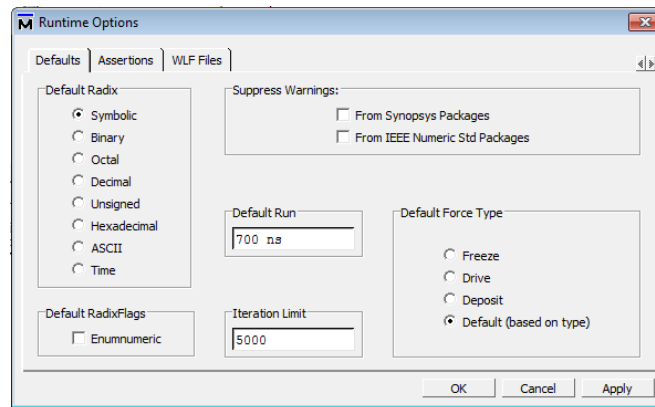


Figure B.16

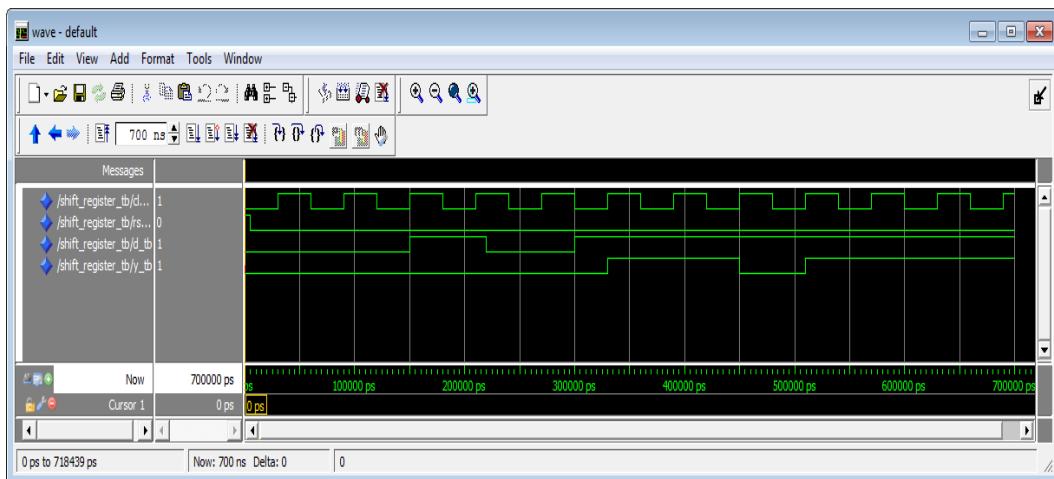


Figure B.17

As can be seen from the Figure B.17 the output y of the shift register is indeed four positive edges behind the input d .

Appendix C Quartus II Tutorial

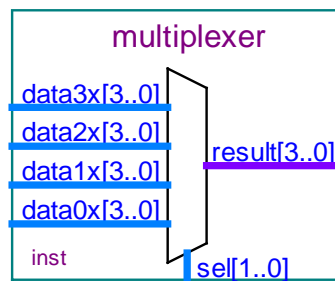
This tutorial provides a briefly description of Quartus II from Altera. It is a design software for circuits which are based on CPLD and FPGA. This tutorial is based on Quartus II 9.0 sp2 Web Edition which is available free of charge at www.altera.com.

The tutorial is divided in four parts.

- C.1 Introduction
- C.2 Creating a Project
- C.3 Synthesizing the Design
- C.4 Simulating the Circuit

C.1 Introduction

The circuit that will be used in this tutorial is a multiplexer which is shown in Figure C.1.a. The output in a multiplexer is equal to the input selected by the selection. The corresponding design file (*mux.vhd*) is shown in Figure C.1.b. The Quartus II 9.0 sp2 allows synthesis with the Quartus II synthesizer and manual graphical simulation with the integrated Quartus II simulator. For simulation with VHDL testbenches use ModelSim.



a

```
1  -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
5  ENTITY mux IS
6  PORT (a,b,c,d: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
7        sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
8        y: OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
9  END mux;
10 -----
11 ARCHITECTURE behavior OF mux IS
12 BEGIN
13     y<=a WHEN sel="00" ELSE
14       b WHEN sel="01" ELSE
15       c WHEN sel="10" ELSE
16       d;
17 END behavior;
18 -----
```

b

Figure C.1

C.2 Creating a Project

- a. Start Quartus II.
- b. Select **File>New Project Wizard** to create a new project. A New Project Wizard pop up box will appear (Figure C.2).

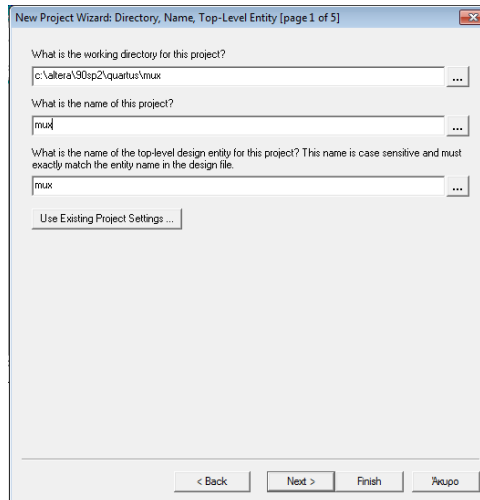


Figure C.2

- c. In the working directory field select the directory where all projects files should be located. In the project name field enter the desired name (mux) and observe that the entity name field is automatically filled with the same name. It is preferable to use the same name for the three fields, directory, project and entity. Also never store different projects in the same directory. Click Finish until the Project Navigator is displayed (Figure C.3).

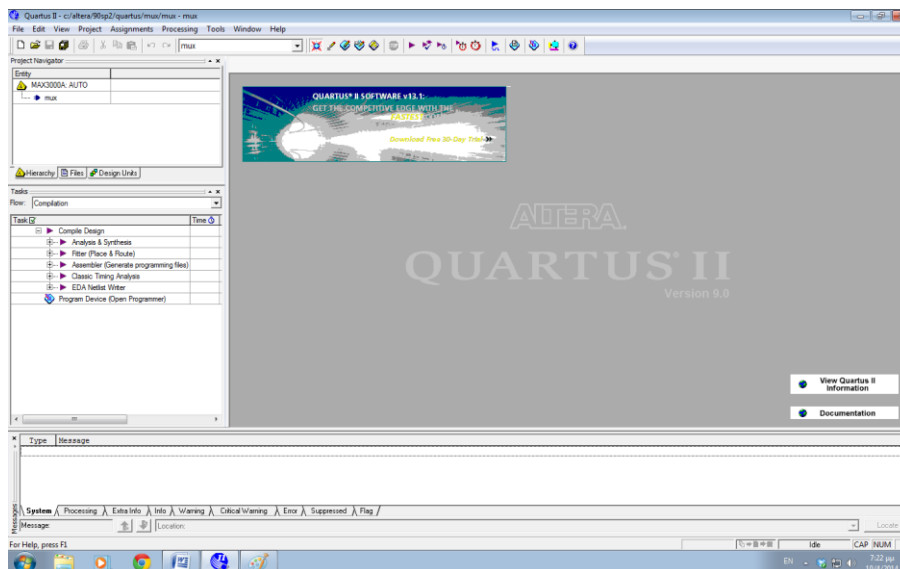



Figure C.3

- d. Select File>New which calls the dialog box of figure C.4. Select VHDL File and then click OK. Alternative open the VHDL editor by clicking . A blank page will be presented. Type the VHDL code (Figure C.1.b).

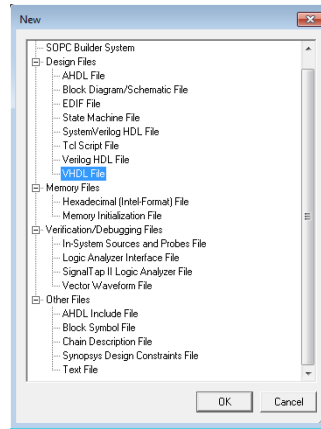


Figure C.4

- e. Select File>Save and save the VHDL code as mux.vhd.

C.3 Synthesizing the Design

- a. Select the device in which the circuit should be implementing. Select Assignments>Devise. Select FLEX10K EPF 10K10TC144-4 as shown in figure C.5

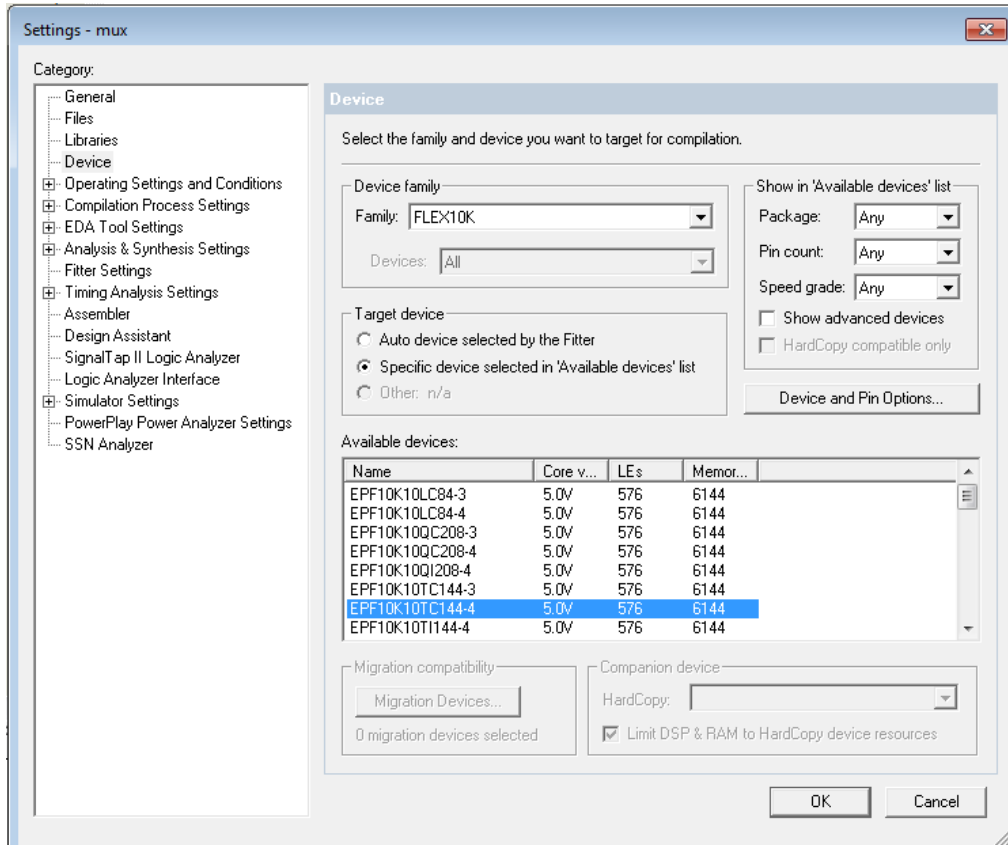



Figure C.5

- b. From the main menu select Processing>Start>Start Analysis & Synthesis. Alternative click .
- c. When the compilation ends the report of figure C.6 is exhibited. The compilation report contains several pieces of valuable information. Some of them are:
 - a. Devise type and number of pins: In figure C.6 the devise type is EPF 10K10TC144-4 and the total pins are 22.
 - b. Number of logic elements: Figure C.6 shows the amount of logic elements needed to implement the circuit. In this case 8 logic elements are needed.
 - c. RTL View: This tool shows how the code was interpreted by the compiler. Select Tools>Netlist Viewers>RTL Viewer which exhibits the circuit of figure C.7.
 - d. Pin assignments: In the compilation report select Fitter>Resource section>Input Pins. This leads to the table which is shown in figure C.8. Next select Fitter >Resource section>Output Pins which leads to the table which is shown in figure C.9.

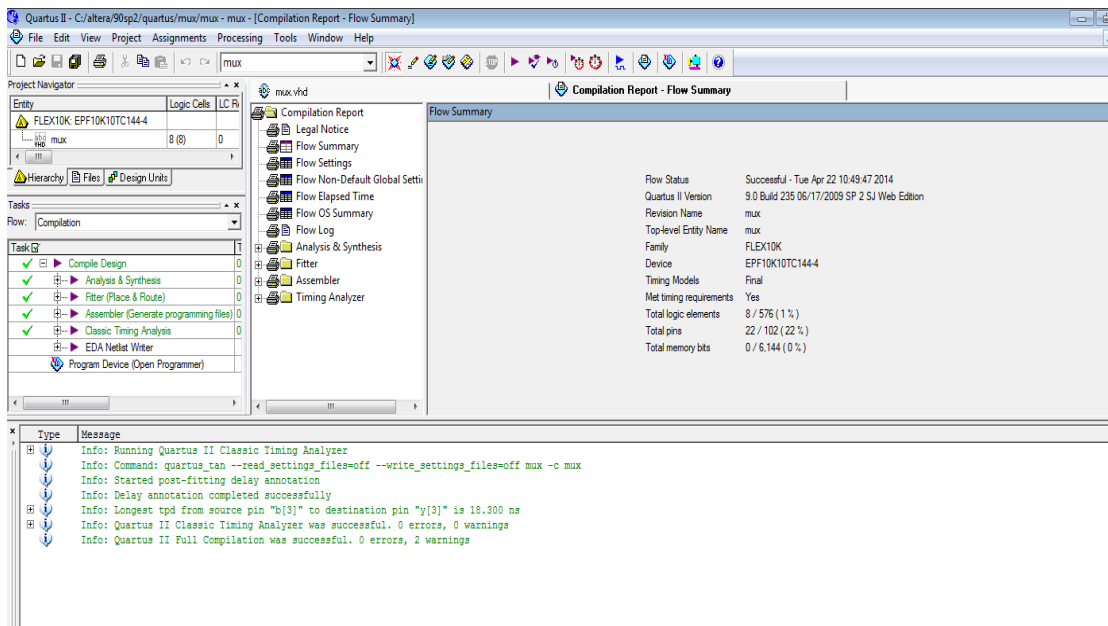


Figure C.6

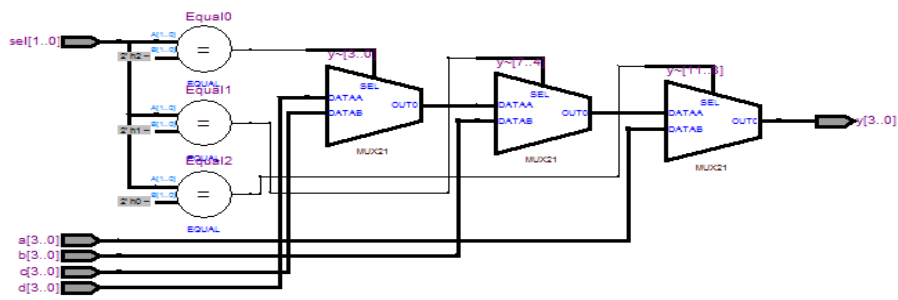


Figure C.7

mux.vhd

Compilation Report - Input Pins

RTL Viewer

	Name	Pin #	Row	Col.	Fan-Out	Global	I/O Register	Use Local Routing	Power Up High	Single-Pin CE	I/O Standard
1	b[0]	54	--	--	1	no	no	no	no	no	TTL
2	sel[0]	126	--	--	6	no	no	no	no	no	TTL
3	c[0]	56	--	--	1	no	no	no	no	no	TTL
4	sel[1]	124	--	--	6	no	no	no	no	no	TTL
5	a[0]	125	--	--	1	no	no	no	no	no	TTL
6	d[0]	55	--	--	1	no	no	no	no	no	TTL
7	c[1]	100	A	--	1	no	no	no	no	no	TTL
8	b[1]	97	A	--	1	no	no	no	no	no	TTL
9	a[1]	95	A	--	1	no	no	no	no	no	TTL
10	d[1]	98	A	--	1	no	no	no	no	no	TTL
11	b[2]	101	A	--	1	no	no	no	no	no	TTL
12	c[2]	10	A	--	1	no	no	no	no	no	TTL
13	a[2]	99	A	--	1	no	no	no	no	no	TTL
14	d[2]	96	A	--	1	no	no	no	no	no	TTL
15	c[3]	8	A	--	1	no	no	no	no	no	TTL
16	b[3]	102	A	--	1	no	no	no	no	no	TTL
17	a[3]	13	A	--	1	no	no	no	no	no	TTL
18	d[3]	9	A	--	1	no	no	no	no	no	TTL

Figure C.8

mux.vhd

Compilation Report - Output Pins

RTL Viewer


Technology Map Viewer - Post-Fitting

	Name	Pin #	Row	Col.	I/O Register	Use Local Routing	Power Up High	Slow Slew Rate	Single-Pin OE	Single-Pin CE	Open Drain	TRI Primitive	I/O Standard
1	y[0]	14	A	--	no	no	no	no	no	no	no	no	TTL
2	y[1]	12	A	--	no	no	no	no	no	no	no	no	TTL
3	y[2]	7	A	--	no	no	no	no	no	no	no	no	TTL
4	y[3]	11	A	--	no	no	no	no	no	no	no	no	TTL

Figure C.9

C.4 Simulating the Circuit

The procedure is done through the Waveform Editor which introduces the appropriate input waveforms for the simulation. Therefore to perform manual graphical simulation we need first to create the input waveforms.

- a. Click  or select **File>New** which will open the dialog of figure C.4. Select **Vector Waveform File** and click OK.
- b. Select **View>Fit in Window** to have the complete plot exhibited in the waveforms window (Figure C.10).

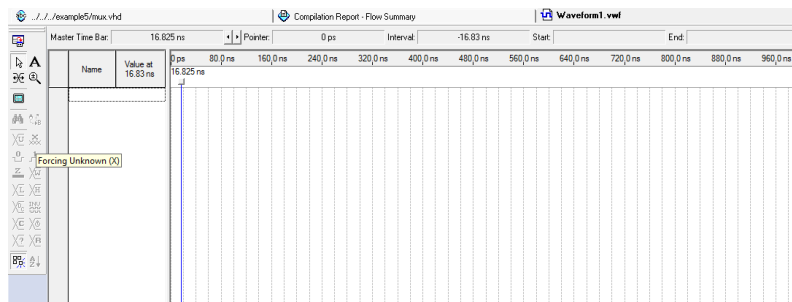


Figure C.10

- c. The time axis in Figure C.10 goes from 1 to 1 μ s. Select **Edit>End Time** if a different time is needed.
- d. Select **Edit>Grid Size** and enter 50ns.
- e. Right click in the white area under Name and select **Insert>Insert Node or Bus** (Figure C.11).

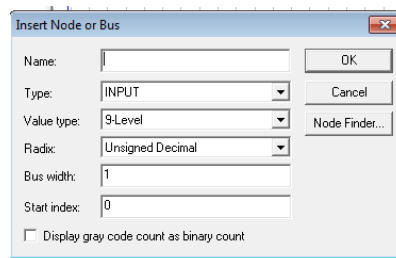




Figure C.11

- f. Click **Node Finder** and a **Node Finder** dialog pops up as shown in (Figure C.12.a).
- g. In the **Filter Field** select **Pins: all** and then click **List**.
- h. The left column of **Node Finder** window now is filled with the signals. Select the desired signals individually  or all  which are copied to the column on the left (Figure C.12.b). Click **OK** twice which will fill the **Waveform Window** with the signals as shown in Figure C.13.

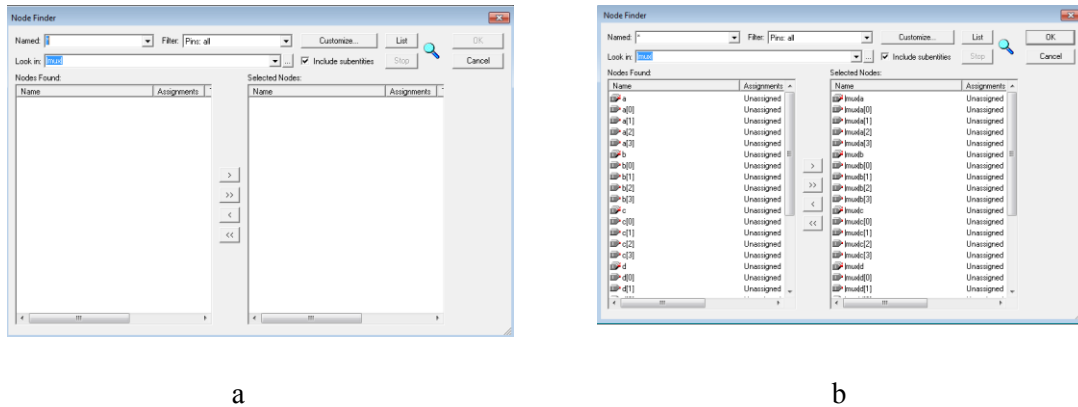


Figure C.12

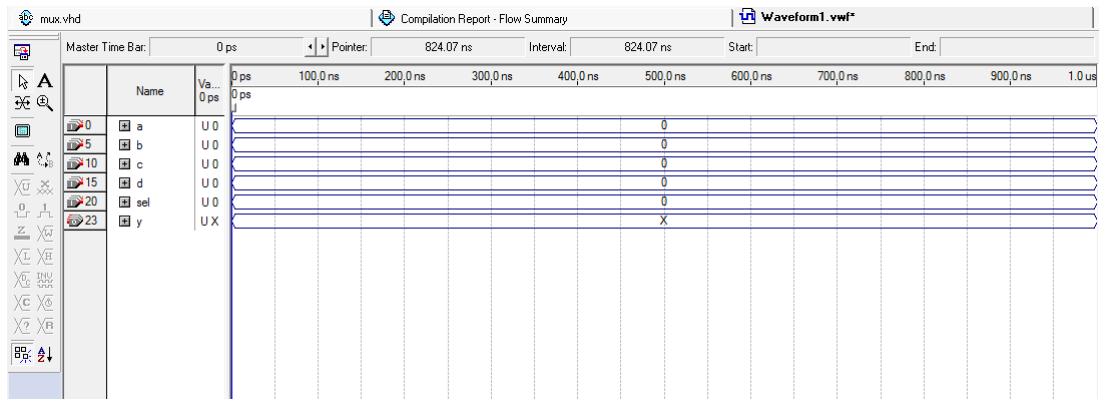





Figure C.13

- i. Now we need to draw the waveforms for the signals a, b, c, d and sel after which the simulator will draw the output waveform y.
- j. First we will draw the waveform for signal a. Highlight line a from 0 to 100ns, click the arbitrary icon  enter 2 and click OK. Repeat the procedure entering 4 for the interval 0 to 1μs.
- k. Repeat the process above for the signals b, c and d by entering the values shown in Figure C.15.
- l. Now we will draw the waveform for sel. Select line sel and click the counter icon . A Count Value dialog pop up as shown in figure C.14.a. Enter start value = 0, increment=1. Click on Timing tab and enter the value 50ns in the Count every field as shown in figure C.14.b.
- m. Save the file with the same name (mux) with the extension .vwf.
- n. Since we want to check the design functionalities we choose functional simulation by select Processing>Generate Functional Simulation Netlist. After the procedure has fin-

ished select Assignments>Settings>Simulator Settings. In the simulation mode list choose Functional. Click OK.

- o. Select Run>Start Simulation or click the icon .
- p. The simulator will draw the waveform for y as shown in figure C.16. Examine the results.

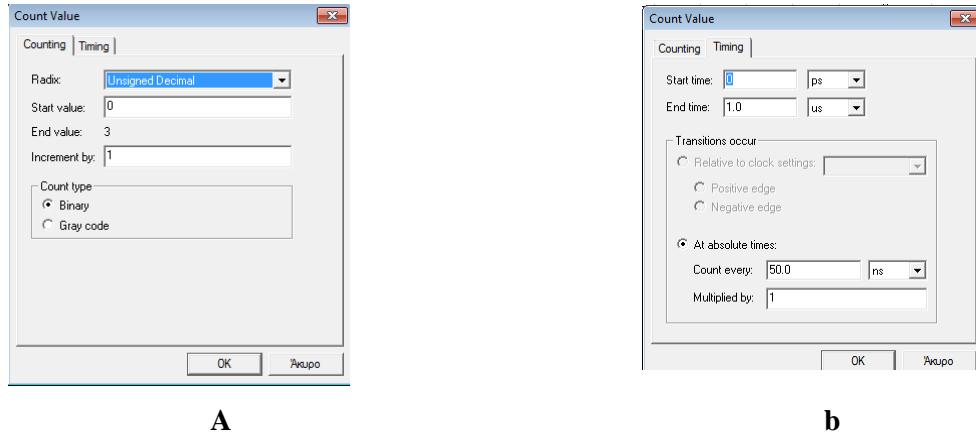


Figure C.14

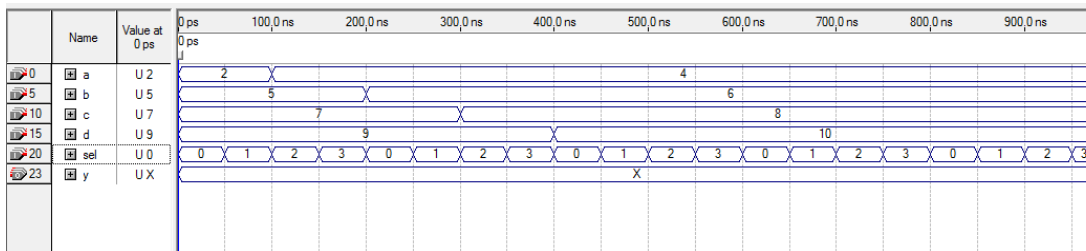


Figure C.15

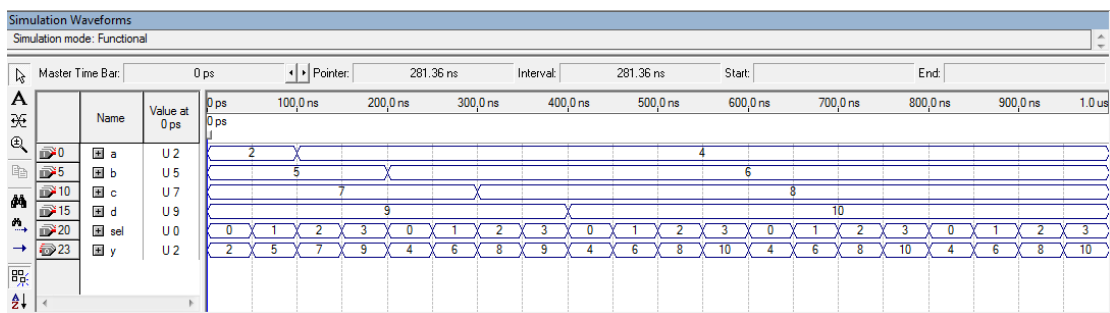


Figure C.16

Simulation results, confirming the functionality of the circuit as shown in figure C.16. The output y of the circuit is equal to the input selected by the sel.

Appendix D Read and Store files for simulation

D.1 Read and Store files for simulation

Files are very helpful for storing and reading data used in simulations. Despite the fact that VHDL does not allow data from files to be directly loaded into the synthesis environment, such action is possible for simulation. The main VHDL procedures extracted from the package *textio* and *std*.

Reading data from a txt file:

```
-----file_read-----  
  
LIBRARY ieee;  
  
USE ieee.std_logic_1164.all;  
  
USE std.textio.all;  
  
USE ieee.std_logic_arith.all;  
  
-----  
  
ENTITY file_read IS  
  
    PORT (clk: IN STD_LOGIC;  
          rst: IN STD_LOGIC;  
          x_out: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));  
  
    END file_read;  
  
ARCHITECTURE file_read OF file_read IS  
  
    FILE f: TEXT OPEN READ_MODE IS "y_75200.txt"; --open a file in "read" mode where f -  
--is the identifier  
  
    BEGIN  
  
        PROCESS  
  
            VARIABLE l:LINE;          --write a value to a variable l of type Line. The data type can -  
-be BOOLEAN, BIT, BIT_VECTOR, INTEGER, REAL, TYPE, CHARACTER or STRING  
  
            VARIABLE s: INTEGER;  
  
        BEGIN  
  
            wait until rst='1';
```

```

wait until rst='0';

WHILE NOT ENDFILE (f) LOOP

    READLINE(f,l);  --read a value from the file identified by f and assign it to the variable
--l

    READ(l,s);--read a value from the line l and assign it to the variable s

    x_out<=CONV_STD_LOGIC_VECTOR (s, 8);  --  convert  the  variable  s  to
std_logic_vector of 8 bits

    wait until clk='1';

    END LOOP;

END PROCESS;

END file_read;

```

Storing data to a txt file:

```

-----file_store-----

LIBRARY ieee;

USE ieee.std_logic_1164.all;

USE std.textio.all;

-----

ENTITY file_store IS

    PORT (clk: IN STD_LOGIC;

          rst: IN STD_LOGIC;

          s_in: IN INTEGER RANGE -128 TO 127 ;

          s_out: OUT INTEGER RANGE -128 TO 127 );

    END file_store;

ARCHITECTURE file_store OF file_store IS

    FILE f: TEXT OPEN WRITE_MODE IS "y_75200_1.txt  --open a file in “write” mode
--where f is the identifier

    BEGIN

```

```
PROCESS (clk,rst)
  VARIABLE l:LINE;
BEGIN
  IF (rst='1') THEN
    s_out<=0;
  ELSIF(clk'EVENT AND clk='1') THEN
    WRITE(l,s_in);  --to write a value s_in to a variable l of type line
    WRITELINE(f,l); --to write the line l to the identifier by f
    s_out<=s_in;
  END IF;
END PROCESS;
END file_store;
```

Appendix E Design and Test Files for functional simulation

E.1 Design File and Test File.

For functional simulation two files are needed to be prepared by the user which, are a Design File and a Test File.

Test file:

```
-----testbench-----  
  
LIBRARY ieee;  
  
USE ieee.std_logic_1164.all;  
  
USE ieee.numeric_std.all;  
  
USE work.my_declarations.all;  
  
-----  
  
ENTITY testbench IS  
  
END testbench;  
  
-----  
  
ARCHITECTURE behavior OF testbench IS  
  
COMPONENT fir IS  
  
PORT (clk: IN STD_LOGIC;  
      rst: IN STD_LOGIC;  
      y: OUT INTEGER RANGE -128 TO 128);  
  
END COMPONENT;  
  
SIGNAL clk_tb: STD_LOGIC:= '0';  
  
SIGNAL rst_tb: STD_LOGIC:= '0';  
  
SIGNAL y_tb: INTEGER RANGE -128 TO 127;  
  
BEGIN  
  
    dut: fir  
  
    PORT MAP (clk=>clk_tb, rst=>rst_tb, y=>y_tb);  
  
    clk_tb<= NOT clk_tb AFTER 10ns;
```

```
rst_tb<='1' AFTER 5ns, '0' AFTER 20ns;
```

```
END behavior;
```

Design File:

-----fir-----

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;
```

```
USE ieee.numeric_std.all;
```

```
USE work.my_declarations.all;
```

```
USE work.components.all;
```

```
ENTITY fir IS
```

```
    PORT ( clk, rst: IN STD_LOGIC;
```

```
           y: OUT INTEGER RANGE -128 TO 127);
```

```
END fir;
```

```
ARCHITECTURE structural OF fir IS
```

```
SIGNAL u_1: SIGNED (7 DOWNTO 0);
```

```
SIGNAL Z: std_logic_vector (7 DOWNTO 0);
```

```
SIGNAL w: vector_coeffs (0 TO 100);
```

```
SIGNAL c: window (1 TO 101);
```

```
SIGNAL p: INTEGER RANGE -128 TO 127;
```

```
BEGIN
```

```
u_1<=signed(z);
```

```
U1: shift_reg PORT MAP (clk=>clk,rst=>rst,shift_inp=>u_1 ,shift_out=>c);
```

```
U2: file_read PORT MAP (clk=>clk,rst=>rst,x_out=>z);
```

```
U3: rom PORT MAP (data=>w);
```

```
U4: mac PORT MAP (clk=>clk, coeff=>w, u=>c, y_out=>p);
```

```
U5: file_store PORT MAP (clk=>clk, rst=>rst, s_in=>p, s_out=>y);
```

```
END structural;
```

Appendix F Matlab

F.1 Simulation using Matlab

```
a=load('coefficients.txt'); % Load the coefficients of the filter

f=500; % frequency of input signal change to 500, 1000, 5000, 10000, 15000, 20000, 25000
%30000, 36000, 42600, 50000, 60000, 65000, 75200, 100000

T=1/f;

tmin=0;

tmax=60*T;

dt=T/100;

t=tmin:dt:tmax;

y=127*sin(2*pi*f*t); % input signal

>> figure(1);

>> plot(t,y);

xlabel('time');

ylabel('amplitude');

title('input signal 500');

dts=1/2000000; %sampling period

ts=tmin:dts:tmax;

s=127*sin(2*pi*f*ts); %sampling the input signal

stem(ts,s);

fileID=fopen('y_500.txt','w'); % open a txt file. This txt file shall be read from File_read for
%functional simulation using the simulator Modelsim

fprintf(fileID, '%3.0f\r\n',s); % Convert to integer

fclose(fileID);

q=load('y_500.txt'); % Load the txt file

k=conv(a,q); %convolving the coefficients with the samples

e=load('y_500_1.txt'); % Load the txt file which contains the results from the simulation
```

```
% using Modelsim  
figure(2);  
plot(k,'r'); % output signal using Matlab  
hold on  
plot(e); % output signal using Modelsim  
>> xlabel('samples');  
ylabel('amplitude');  
title('signal 500Hz output');
```

F.2 Convert coefficients to integer

```
a=load('coefficients.txt'); % Load the txt file which contains the filter coefficients in their  
%original form  
w=a*2^15;  
h=round(w);  
save('integer coefficients.txt','h', '-ascii') % save the integer coefficients in a txt file
```