FPGA Signal Preprocessing for Digital Wireless Receivers

Bjarne Petersen

Kongens Lyngby 2012
IMM-M.Sc-2012-102

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Summary

This thesis deals with the task of exchanging analog filters with digital filters. These analog filters, used in a base station receiver for wireless communication, have the job of filtering incoming TETRA and TEDS signals for unwanted channels and blockers. Analysis performed in this thesis based on a set of requirements for the filter process, have concluded that the best filter type for the digital filers is FIR filters of a symmetric structure.

In order to apply FIR filters, a flexible filter architecture has been designed and implemented as an RTL hardware model with VHDL. Digital filtering can be broken down to a sum of additions and multiplications. Since embedded multipliers are limited in FPGAs the designed architecture is based on a resizeable parallel and sequential part which allows it to make the best use of the multipliers taken the desired clock frequency into account. The architecture supports symmetric FIR filters of an odd order number. The order can vary from 7 to 575 in predefined steps. A suitable FPGA is necessary to implement filters of high orders.

This architecture has been used to implement a set of single and dual carrier systems based on filters of order 383 on a Spartan 3 FPGA.

In order to test the system, the architecture has been surrounded by an environment consisting of a set of interfaces enabling the system to receive incoming data from an ADC and send filtered data to a pc for further analysis. Through this, the filter architecture was verified and the implemented filters tested successfully.

Resumé

Dette speciale beskæftiger sig med at udskifte analog filtre med digitale filtre. Disse analog filtre sidder i modtageren på en base station brugt til trådløs kommunikation og har til opgave at filtrere TETRA og TEDS signaler for uønskede kanaler samt støj. Der er udført analyser baseret på fastlagte krav, som konkluderer, at den bedst egnede digitale filter type er FIR filter med en symmetrisk struktur.

For at anvende FIR filtre er der blevet udviklet en fleksibel filter arkitektur, som er implementeret som RTL hardware model i VHDL. Digital filtrering kan nedbrydes til en sum af multiplikationer og additioner. Eftersom mængden af integrerede *multipliers* er begrænset på FPGAer, så er den forslåede arkitektur baseret på en parallel og sekvensiel del, hvilket muliggør at udnytte *multpliers* bedst muligt i forhold til den drivende klok frekvens. Denne arkitektur understøtter symmetriske FIR filtre med en ulig orden. Denne orden kan gå fra 7 og op til 575 i prædefinerede trin. En passende FPGA er nødvendig for at implementere filtre med en høj orden.

Arkitekturen er blevet brugt til at implementere en håndfuld *single-* og *dual-carrier* systemer, baseret på filtre med en orden på 383, i en Spartan 3 FPGA.

For at teste systemet er arkitekturen implementeret med et sæt brugerflader, som gør det muligt at modtage data fra en ADC og sende filtreret data til en pc til videre data behandling. Gennem dette er filterarkitekturen blevet verificeret og de implementerede testfiltre testet med success.

Preface

This thesis was prepared for the Department of Informatics and Mathematical Modelling at the Technical University of Denmark in partial fulfilment of the requirements for acquiring the Master of Science degree in engineering.

The thesis deals with additional digitalization of the receiver in a base station used for wireless communication designed by Motorola Solutions. The task, defined by Motorola Solutions, consists of applying digital signal processing techniques on a field-programmable-gate array thereby replacing analog filtering with digital filtering.

Lyngby, August 2012

Bjarne Petersen

Acknowledgements

I would like to thank my supervisor at the Technical University of Denmark, Prof. Dr. Alberto Nannarelli, for having given me the opportunity to do a project in coorporation with Motorola Solution. Furthermore, I would like to thank him for his support and guidance during the project.

In addition, I would like to thank Motorola Solutions for having offered me the project, and for having given me a workplace and equipment at their site in Glostrup. A special thanks goes to Hans Erik Gram, deployed at Motorola Solutions, for defining the project, providing the hardware, and for guiding me throughout the entire project.

Moreover, I would like to thank the entire Tetra Basestation Hardware Department at Motorola Solutions for making me feel welcome and always being helpful.

Last but not least, I would like to thank everyone who has helped proof reading this thesis.

Contents

CHAPTER 1

Introduction

Digital signal processing (DSP) is a primarily technology driven field which started from around mid 1960s when digital computers and digital circuitry became fast enough to process large amounts of data efficiently. Today DSP is used for anything from speech recognition to seismology.

The still ongoing increase in field programmable gate arrays' (FPGA) size and performance have made them great hardware accelerators. Furthermore, they are very cheap when only a small amount of chips are needed (compared to IC production). These two features, combined with the fact that FPGAs are reprogrammable, have made them very popular for rapid prototyping.

This project is about using an FPGA for digital signal processing, thereby improving the receiver in a base station used for wireless communication.

1.1 Project Description

The job of a base station is to handle wireless communication. The term *communication* can be broken down to two parts, *sending* and *receiving*. This project focuses the receiver part.

The receiver has to filter the incoming signals (picked up by an antenna) down to the desired parts. This filtering consists of a series of steps. During one of these steps, a transition from the analog domain to the digital domain is performed. The reason for a digital system is manifold, for one it makes it possible to transmit not just audio but data as well. Furthermore this data can be encrypted.

This project's objective is to move the digital transition one step closer to the antenna by replacing the analog channel-filters with digital filters. The design proposed by Motorola Solution is shown in figure 1.1.

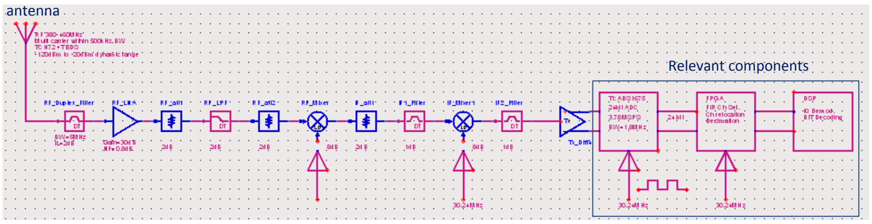


Figure 1.1: Receiver Chain

This project will only be concerned with the last few components of this chain as depicted in figure 1.2.

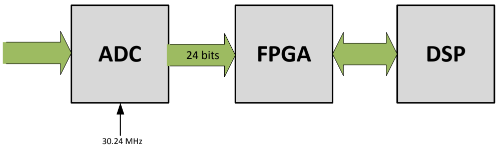


Figure 1.2: Relevant Components

The base station can communicate under TETRA[1] and TEDS[2] specifications. The current base station design contains a set of analog filters used to suppress everything but the carrier signal. Since these analog filters cannot be modified,

[1]TETRA: Terrestrial Trunked Radio. Two-way transceiver specification designed for government agencies, emergency services, transport services and military

[2]TEDS: Tetra enhanced data service. Supports wider channels, thereby increasing the bandwidth and enabling data transmission

the position of the carrier signal has to be predetermined within the frequency band. Digital filtering with tune-able filters would relax this requirement and give more freedom for the placement of the carrier signal. Furthermore, additional carrier signal types with different channel widths can be supported. TEDS comes with four different classes (U25, U50, U100, and U150). The number indicates the bandwidth of the signal in kHz. So far only two of these are supported by Motorola's analog based filter design. Supporting all classes would be too expensive when designed with analog components.

Summing up, digitalizing the filters gives the following advantages: *Flexible reprogrammable filters* and *Support for more channels and channel types* resulting in a very flexible multi-carrier receiver.

1.1.1 Delimitation

This is a prototype development based on a *proof of concept* mentality. Only the neighbouring components of the FPGA are of concern, these being the analog-to-digital-converter (ADC) and the DSP. The interface to the ADC is well defined while the interface to the DSP is not defined at all. Therefore, no interface for the DSP will be designed on the FPGA. After incoming data has been filtered on the FPGA, the next step would be IQ demodulation. The IQ demodulation is not part of this project. The coefficients defining the filters are thought to be generated on the DSP and transmitted to the FPGA. The ADC provided does not have the desired dynamic range wanted for a final product, hence, the output of the filters cannot have the desired dynamic range. As shown in figure 1.2 an external clock is driving the ADC/FPGA. In this project the clock will be generated on the FPGA board and driving the ADC.

1.2 Approach

The first step of this thesis was to **study filters**, both analog and digital. By doing this, a general knowledge was achieved which would function as a foundation through the entire project.
The next step was to do **analysis on filters** and find the right filter type for this project based on predefined requirements. Since the available hardware resources limit the filter size, different filter structures and algorithms were analysed to reduce the required hardware resources. For this **MATLAB** was used since it is very strong in creating and analysing filters. Time was spent learning the program and its *filterbuilder* tool.

The next step was to **create an environment** on the FPGA in which the filter architecture could be implemented. First, a control unit was designed to interface with the ADC, thereafter a control unit to interface with a PC through an USB connection. At that point the environment was ready to be connected to a **basic test filter**. In order to analyse the data from the filter the basic environment needed to be **extended with an SDRAM controller** to store data samples. The USB interface was extended as well in order to load the data stored in the SDRAM to a pc and analyse them with MATLAB.

1.3 Equipment

This section presents a brief description of the equipment used in this project.

ADS1675REF Hardware Kit
This kit consists of two hardware boards, the ADS1675REF (ADC by TI) and the XEM3010-1500P (FPGA by Opal Kelly). The FGPA board can be mounted on the ADC board as shown in figure 1.3.

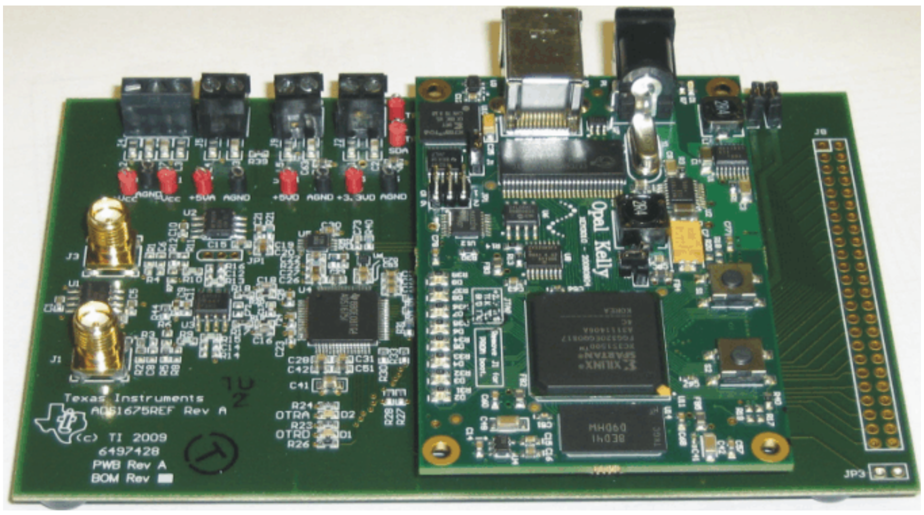


Figure 1.3: ADS1675REF Hardware Kit

ADS1675REF
This board holds the ADS1675 chip, a 24-bit, ΔΣ analog-to-digital converter that has the following key features:

- AC Performance:
  103dB of Dynamic Range at 4MSPS
  111dB of Dynamic Range at 125kSPS
  -107dB THD

- DC Accuracy:
  3ppm INL
  4mV/°C Offset Drift
  4ppm/°C Gain Drift

- Programmable Digital Filter with User-Selectable Path:
  Low-Latency: Completely settles in 2.65ms
  Wide-Bandwidth: 1.7MHz BW with flat passband

- Flexible Read-Only Serial Interface:
  Standard CMOS
  Serialized LVDS

- Easy Conversion Control with START Pin
- Out-of-Range Detection
- Power: 575mW

A full description of the chip can be found here [3].

XEM3010-1500P

The XEM3010 board holds the FPGA chip and has the following key features:

- Xilinx Spartan-3 FPGA
  1.5 M gates - 30k logic cells
  32 *18x18*-multipliers
  32MB SDRAM
- Cypress PLL clock generator
- USB microcontroller
- 8 LEDS, 2 pushbuttons
- Two 80 pin expansion connectors
- FrontPanel support (see next section)

Power Supplies
A set of controllable power supplies were used to supply the ADC and the FPGA board.

Signal Generators
Signal generators were used (sine, TETRA) to generate input signals for the ADC.

Digital Oscilloscopes
Digital Oscilloscopes were used for onboard sanity measurements.

PC
A computer was used to run the software presented in the next section.

1.4 Software Tools

ISE Project Navigator [Xilinx]
Version 13.4, O.87xd
Used for HDL development, synthesis, implementation and bit-file generation.

CoreGenerator [Xilinx]
Version 13.4, O.87xd
Used for generation of hardware components such as storage elements.

Modelsim [Altera]
Version 10.0, starter edition
Used for HDL simulations.

FrontPanel [Opal Kelly]
Version 4.0.8
Used to program the FPGA by loading bit-files generated by Xilinx ISE. Also used to set the PLL clocks which, among others, drive the ADC. Last but not least it is used to execute small programs that can communicate with the FPGA through a USB connection.

MATLAB
VERSION R2010A
Used for filter analysis and creation, data analysis, and creating the graphs used in this report.

ADCPro [Texas Instruments]
VERSION 1.2.2 BUILD 5
This tool enables data sampling with the ADS1675REF kit and has a few analysis features such as FFT. Data samples are viewable as plots. This tool has served as a reference when analysing data in MATLAB and for verifying the hardware setup.

XVI32
VERSION 2.54
This hex editor was used to convert data, extracted from the FPGAs SDRAM though a FrontPanel script, from HEX to ASCII.

Notepad++ (with HEX-Editor Plugin)
VERSION 6.1.2 (VERSION 0.9.5)
Used for quick-view of data extracted from the FPGA SDRAM and for code editing.

Texmaker (with MiKTeX)
VERSION 3.3.3 (VERSION 2.9.4407)
Used to create this document.

Inkscape
VERSION 0.48
Used for manipulation of PDF images.

Visio 2010 [Microsoft]
Used to create most of the diagrams in this report.

1.5 Setup

This section describes how the ADC and the FPGA board were set up and connected with power supplies and their settings.

The XEM3010 and the ADS1675 board have a set of jumpers onboard which must be set in correlation with the use of the board as stated in [4]. A silkscreen drawing of the ADS1675 is shown in figure 1.4.

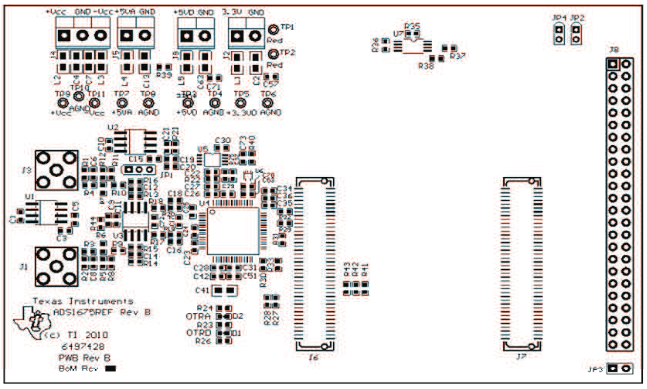


Figure 1.4: ADS1675REF Silkscreen Drawing

The pins and jumpers were supplied and set as stated in the following table.

Pin	Applied	Description
J5-1	+5V	Supplies analog parts
J5-2	GND	
J4-1	+9V	Differential amplifier supply
J4-2	GND	
J4-3	GND	
J4-4	-4V	
J9-1	J2-1	Connected to J2
J9-2	J2-2	
J2-1	+3.0V	Supplies digital parts
J2-2	GND	
P1-1	+5V	XEM3010 supply
P1-2	GND	
JP1	OFF	Located on FPGA board
JP2	OFF	
JP3	ON	
JP4	OFF	
J1	ON	Located on FPGA board
J2	OFF	Located on FPGA board

A differential input signal, generated by a signal generator, was applied at J1 and J3.

In figure 1.5 a block diagram of the ADC is shown. On the right side all its I/O pins are shown, those will be explained at a later point.

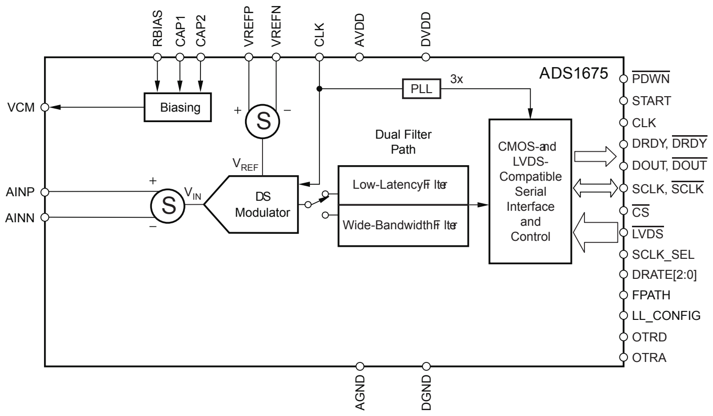


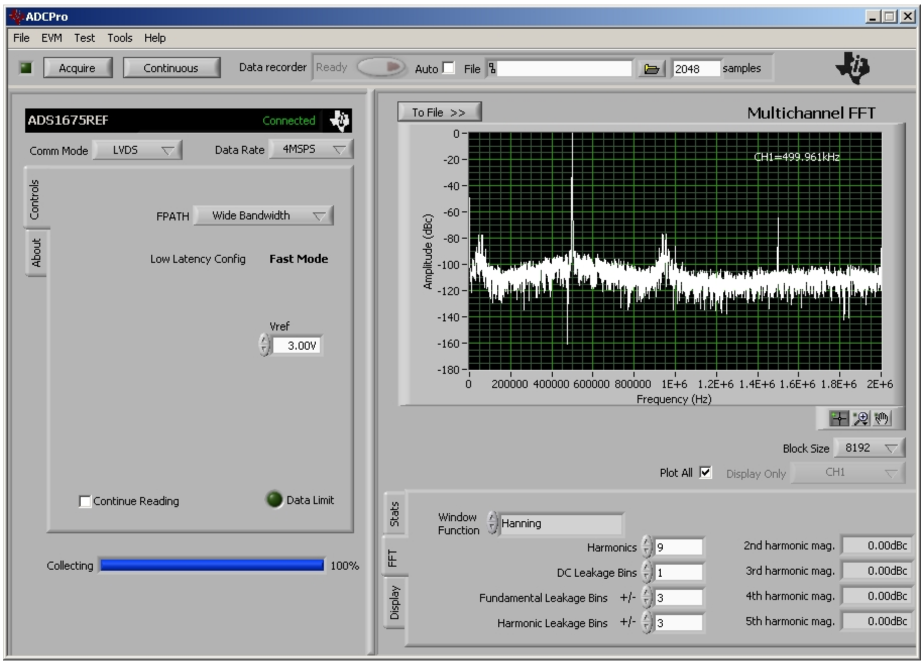
Figure 1.5: ADS1675REF Block Diagram

The DS modulator compares the input signal with the reference signal. Afterwards the signal goes through a filter. In this project the wide-bandwidth filter and the fastest high-speed mode (4MSPS) is used. Finally the sample is outputted as a 24bit signal, transmitted bit by bit (MSB first) at a clock rate 3 times the input clock. This value is of type 2's complement and calculated as follows:

Input	**Ideal output code**
$V_{in} \geq V_{REF}$	$\texttt{7FFFFF}_{hex}$
$V_{in} = \frac{-V_{REF}}{2^{23}-1}$	$\texttt{000001}_{hex}$
$V_{in} = 0$	$\texttt{000000}_{hex}$
$V_{in} = \frac{-V_{REF}}{2^{23}-1}$	$\texttt{FFFFFF}_{hex}$
$V_{in} \leq -V_{REF}\left(\frac{2^{23}}{2^{23}-1}\right)$	$\texttt{800000}_{hex}$

In high-speed mode (used in this project) the 24th bit (LSB) is held low, hence, only a 23 bit resolution is provided.

For initial test purposes the XEM3010 was connected to a PC through its USB port. The ADCPro software was installed[3] on that machine and used to collect samples. A signal generator generated a 500kHz 2VPP sine wave. A frequency plot is shown in figure 1.6. The plot shows a peak close to 500kHz corresponding to the frequency of the generated signal. At 1MHz and 1.5MHz the first and second harmonics are visible. This shows that the test setup is working.



Data recorder Ready Auto File 2048 samples

ADS1675REF Connected

Comm Mode LVDS Data Rate 4MSPS

FPATH Wide Bandwidth

Low Latency Config **Fast Mode**

Vref 3.00V

Continue Reading Data Limit

Collecting 100%

CH1=499.961kHz

Block Size 8192

Plot All ✓ Display Only CH1

Window Function Hanning

Harmonics 9

DC Leakage Bins 1

Fundamental Leakage Bins +/- 3

Harmonic Leakage Bins +/- 3

2nd harmonic mag. 0.00dBc

3rd harmonic mag. 0.00dBc

4th harmonic mag. 0.00dBc

5th harmonic mag. 0.00dBc

Figure 1.6: ADCPro FFT on 2VPP 500kHz sine wave

Sanity measurements have shown a significant difference in the impedance of the signal generator and the ADC resulting in a voltage reduced incoming signal. This is of no concern for this project, since it has been verified that the voltage across the input terminals corresponds to the voltage measured by the ADC.

The ADC samples can sample at a rate of up to 4MSPS (samples generated at 4MHz). This means, as given by the Nyquist sample theorem, that the maximum frequency that can be captured is:

$$f_s > 2B \Leftrightarrow B < \frac{f_s}{2} = \frac{4MHz}{2} = 2MHz$$

[3]Windows XP required

Every signal at this frequency or higher will drown due to aliasing and cannot be reconstructed. This behaviour is depicted in the plot as well since the frequency band goes from 0 to 2 MHz. The wide-bandwidth filter applied starts suppressing at a frequency of 0.425 times the data rate which at 4MSPS corresponds to 1.7 MHz. Hence, the influence of high frequency signals is minimized.

1.6 Thesis Structure

The thesis is divided into chapters as follows:

Chapter 2 introduces digital filtering and performs filter analysis in order to find the best suited filter type for this project.

Chapter 3 describes how an environment is created as a hardware model consisting of interfaces and a filter architecture capable of implementing filters of the desired filter type.

In chapter 4 the hardware model is tested and verified.

Chapter 5 presents a set of results obtained with the hardware model.

In chapter 6 future work is presented while chapter 7 concludes the thesis.

Chapter 2

Digital Filters

Digital filtering is a type of signal processing and the task of a filter is to suppress unwanted components (such as noise) while letting everything else pass. The filters studied in this thesis are applied on time domain signals but do alter the signals based on their frequency, which is the most common method[1]. Hence, the filter analysis presented here is based on the frequency spectrum.

2.1 Filter Types

There exists a big variety of digital filters, classified by many different aspects. This chapter will provide are brief overview and outline which filter types are relevant for this project.

The magnitude response (in terms of frequency) of a filter can basically be broken down to four classes, categorised by which part of the frequency-band is affected by the filter. They are called low-pass, high-pass, band-pass and band-reject (or band-stop) filters. Figure 2.1 shows examples of these four filter classes.

[1]There exists filters that do not act in the frequency domain (e.g. image processing) but they are of no interest for this project

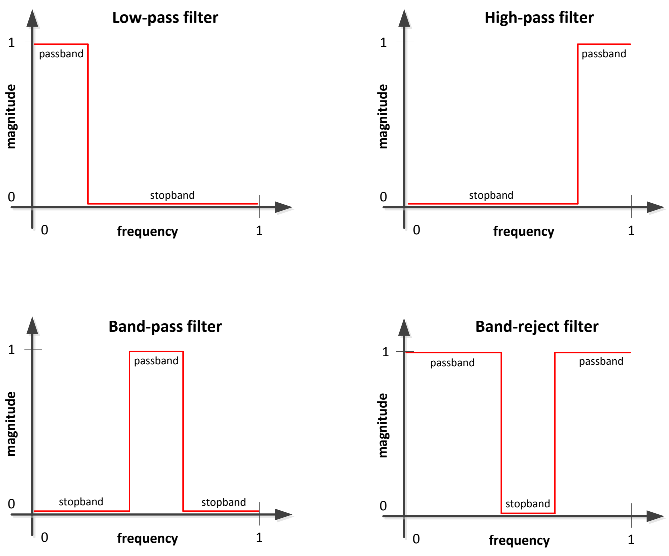


Figure 2.1: Filter classes in terms of magnitude response (normalized plots)

Since this project is about receiving data transmitted by a carrier signal of a certain bandwidth (channel width) located around a certain center/base frequency and suppressing everything else, the filters of interest are band-pass filters. Figure 2.2 shows such a channel with a filter around it and the resulting signal, which is gained by multiplying the filter and the signal graph.

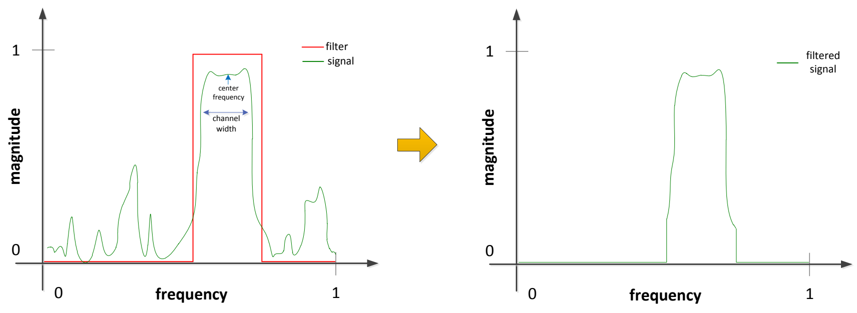


Figure 2.2: Band-pass filter example

Digital filters are most commonly of the type *linear time-invariant system* (LTI system). An LTI system can be characterized entirely by a single function called the *impulse response*.

Such a filter acts on its input signals through linear convolution, denoted $y = f * x$ where f is the filter's impulse response, x is the input signal, and y is the convolved output. The formally definition of the linear convolution process is as follows:

$$y[n] = x[n] * f[n] = \sum_k x[k]f[n-k] = \sum_k f[k]x[n-k] \tag{2.1}$$

This definition is also called for the time-domain point of view. In the frequency domain an LTI system is described by its transfer function, which for discrete-time systems is the Z-transform of the impulse response[2]. Convolution in the time-domain corresponds to multiplication in the frequency-domain. This is depicted in figure 2.3.

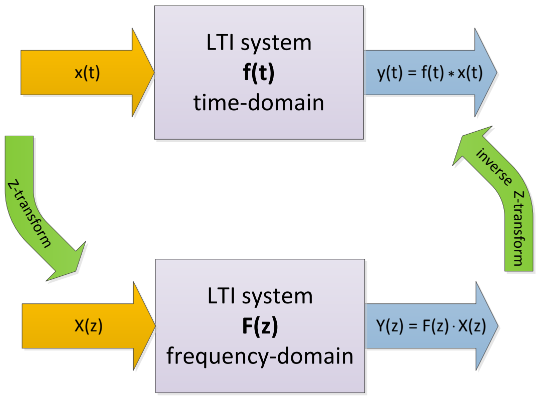


Figure 2.3: LTI system

[2]For continues-time systems the Laplace transform is used which would be the case for analog filters.

The direct transfer to the Z-plane for discrete-time signals is defined as:

$$X(z) = Z\{x[n]\} = \sum_{n=-\infty}^{\infty} x[n]z^{-n} \tag{2.2}$$

where z is a complex variable.

LTI system filters can be divided into two categories: *finite impulse response* (FIR) filters and *infinite impulse response* (IIR) filters. As the name implies, a FIR filter consists of a finite number of sample values, reducing the convolution sum (equation 2.1) to a finite sum per output sample instant. An IIR filter, however, requires that an infinite sum is performed.

The infinite response is produced through a feedback. Thus IIR filters are also called *recursive* filters and FIR filters *non-recursive.* The following section about structures will elaborate on this. More on this topic can be found here [9]. The feedback plays a crucial role for the behavior of a filter. The filter analysis in section 2.4will go deeper into this.

2.2 Filter Structures

Figure 2.4 shows the direct form structure of an L^{th} order FIR filter. This structure corresponds to a graphical representation of the transfer function defined as:

$$F(z) = \sum_{k=0}^{L-1} f[k]z^{-k} \tag{2.3}$$

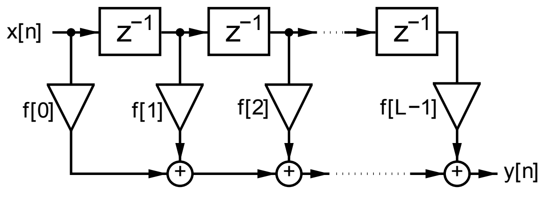


Figure 2.4: Direct form FIR filter [9]

The structure consists of a delay pipeline (tapped delay), adders, and multipliers. A delay in the signal is transformed into a multiplication by z^{-1} in the Z-transform. The operands for the multipliers are the delayed input values and the coefficients defining the transfer function and, consequently, the filter. The output of the filter, given by the finite convolution sum, is:

$$y[n] \;=\; x[n] * f[n] \;=\; \sum_{k=0}^{L-1} f[k]x[n-k] \tag{2.4}$$

where f holds the coefficients.

Figure 2.5 shows the same filter with a different structure called the transposed-form. This structure is achieved by taking the direct-form structure and **1)** exchanging the input and output **2)** inverting the signal flow direction and **3)** substituting the adders with forks and vice versa. The transposed structure shows benefits in the number of required shift registers.

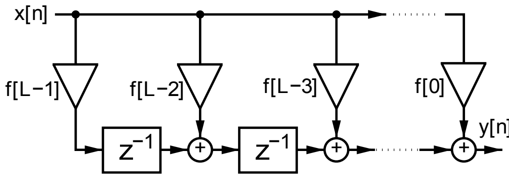


Figure 2.5: Transposed form FIR filter [9]

A very interesting structure is shown in figure 2.6. This structure is achieved by designing the impulse response in such a way that its coefficients are mirrored around the center. By using this symmetry the number of multiplications can be halved by folding the delay pipe line. As will be shown later, the critical resource on the FPGA is the amount of multipliers. The symmetric structure is very interesting because it reduces the amount of required multipliers.

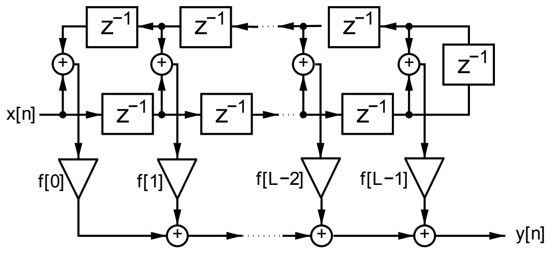


Figure 2.6: Symmetric form FIR filter [9]

As mentioned, the IIR filters do have a feedback loop and do, consequently, consist of a recursive part and a non-recursive part compared to FIR filters which only have the latter. This is depicted in figure 2.7. The transfer function for such a filter consists of two summations, one for the recursive part and one for the non-recursive part. IIR filters can also be of a transposed form as shown for the FIR filters.

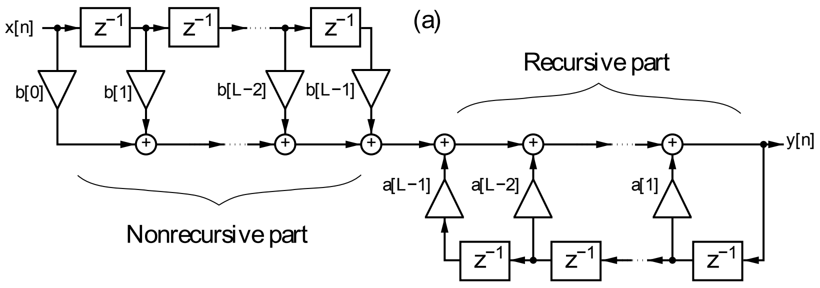


Figure 2.7: IIR [9]

The recursive part in the figure visualizes why the IIR filters have an infinite impulse response compared to FIR filters. The FIR filter's output will eventually, depending on its length (order), stabilize given a single input while the IIR's output will not due to the recursive part.

The structures presented here are only a few of many.

2.3 Requirements

Further analysis is required to determine whether a filter of type IIR or FIR is the correct choice for this project. In order to continue the filter analysis a set of requirements is presented which will guide the filter design in the right direction.

There are five different signal types to be handled with the following specifications:

1. TETRA - 25 kHz bandwidth
2. TEDS-U25 - 25 kHz bandwidth
3. TEDS-U50 - 50 kHz bandwidth
4. TEDS-U100 - 100 kHz bandwidth
5. TEDS-U150 - 150 kHz bandwidth

Those channels could be placed anyway within the frequency band. Since it is a multi-carrier system, two or more of those channels could actually be present with a proper distance between the two channels.

In order to design suitable filters additional requirements are necessary which define the relations between the stop-band and the pass-band. These requirements are described by the following blocker criteria:

1. +/- 500kHz and more, blocker level = -105 dBc
2. +/- 200kHz, blocker level = -100 dBc
3. +/- 100kHz, blocker level = -95 dBc
4. +/- 50kHz, blocker level = -90 dBc

The requirements shall be read as follows: +/- [distance in terms of frequency from the channels center frequency], [required suppression at that frequency compared to a carrier signal]. These blocker requirements are only true for the TETRA specification. The requirements for the TEDS specifications are less strict. Hence, a filter design able to hold filters that obey the strict TETRA requirements will automatically be able to hold filters with less strict requirements.

Furthermore the following requirements have to be met:

- Passband ripple within 1 dB
- Good linearity
- Stability
- Filter must be moveable in the frequency band at a resolution of 250 Hz

Last but not least, the filter must be implementable on an FPGA.

2.4 Filter Analysis

In this chapter the difference between the IIR and FIR filter class in respect of this project will be analysed.

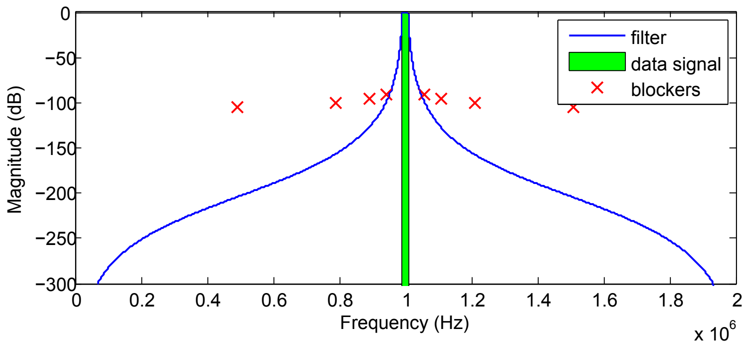
To begin with, the blocker requirements' influence will be analysed. Figure 2.8 shows a 25 kHz wide TETRA signal centered at 1 MHz (green) surrounded by matched filters (blue). The red crosses mark the blocker requirements.

The two plots show an IIR and a FIR sample filter, respectively. These magnitude plots point out one of the main differences between the two filter types. The IIR filters attenuation keeps decreasing and the slope decays more rapidly than the blocker requirements, hence, the closest blocker with respect to the signal determines the requirements for the IIR filter.
The FIR filters attenuation, however, reaches a limit[3]. This limit corresponds to the strongest blocker, which is the blocker furthest away from the channel. Due to the nature of the FIR filter this requirement must already have been met at the closest blocker location.

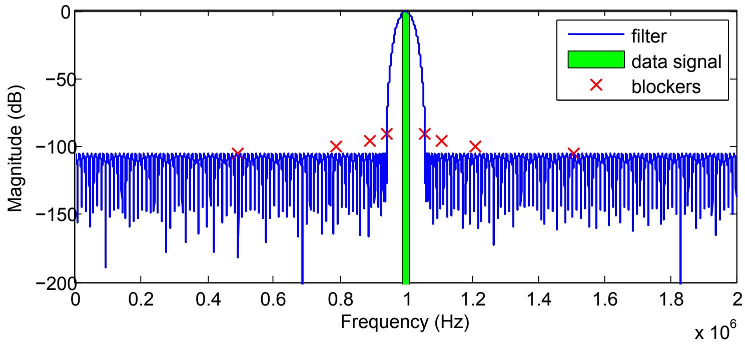
Using the blocker requirements, the following filter specific requirements can be determined:

1. For an IIR filter the stopband attenuation has to be -90dB at +/- 50kHz away from the signal center
2. For a FIR filter the stopband attenuation has to be -105dB at +/- 50kHz away from the signal center

[3]This is not the case for all FIR algorithms but will serve as a good guideline



(a) IIR filter



(b) FIR filter

Figure 2.8: The two different filter types meeting the blocker requirements

By looking at the order of the two filters used to generated these example plots another difference comes clear. The IIR filter is of an order around 10 while the FIR filter's order is above 300. The order size has a direct influence on the hardware required for an implementation. Based on this, the IIR filter seems to be the right choice.

However, the magnitude response is only one side of a filter, its counterpart is the phase response. The IIR and FIR filter show different qualities here as well. One of the requirements has defined the phase to have good linearity within the passband. Non-linearity in the passband can distort the signal and make it impossible to reconstruct the transmitted data. IIR filters do not guaranty linearity. FIR filters on the other side can very easily guarantee linearity. This linearity can be achieved by making the filter symmetric or anti-symmetric, which is no drawback at all. This means that a symmetric filter structure can be used (as presented in 2.2) without further consideration, which halves the amount of multiplications required.

Non-linearity can be corrected in an IIR filter by applying correction functions, but those are heavy and require a lot of operations. Furthermore, IIR filters have some negative sides when quantizing the filter coefficients. This quantization will introduce oscillation effects that cannot be avoided, furthermore the quantization has negative effects on the phase. The filter might even become unstable. FIR filters maintain good linearity even if their coefficients becomes quantized. Due to their oscillation effects and stability issues the IIR filter are not an option for this project. So, even though the FIR filter requires a much larger filter order to meet the magnitude requirements it has advantages in all other aspects. Due to these reasons the filter type used for this project will be FIR filters. More on this topic can be found in [10], [7], [8], and [5].

The difference in the channel width of the five TETRA/TEDS specifications plays a minor role. The TETRA has the narrowest channel (25kHZ) while the TEDS go up to 150kHz. The order of a filter grows inversely proportional to the passband width. Hence, in order to meet the TEDS requirements a filter of a lower or equal order is required than for the TETRA. Therefore, the analysis will from hear on be based on the TETRA specifications. Furthermore, the position of the channel in the frequency band does not influence the filter order.

2.5 FIR Filters

In this section different FIR bandpass filter algorithms will be analysed. This analysis will be based on MATLAB's filter tool *filterbuilder* and is, consequently, restricted to the filter types supported by it. The goal is to find a suited filter with sufficient stop-band attenuation that can be implemented on the FPGA. The filter order is of great concern because it is restricted to the amount of multiplications and additions the FPGA can perform each cycle.

MATLAB's filterbuilder supports four different filter types:

1. Single-rate
2. Decimation
3. Interpolation
4. Sample-rate converter

Since the incoming sample rate is defined to be the same as the outgoing the type of interest is *single-rate.*

Furthermore a set of different algorithms for FIR filters are supported. The algorithms consists of a method and a structure. The structures being:

1. Direct form FIR
2. Direct form FIR transposed
3. Direct form symmetric FIR
4. Overlap-add FIR

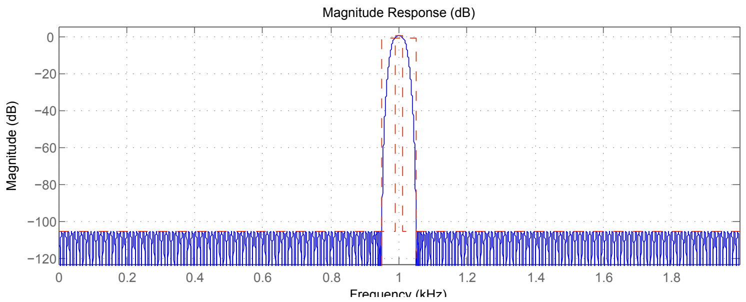
As mentioned multiplication is a critical aspect in the filter implementation, hence, the number of multiplication should be as small as possible. This would be achieved by using the *direct-form symmetric FIR* structure which halves the number of needed multipliers, as established earlier. However, the filter structure does not influence the filter order so for this analysis the chosen filter structure does not matter.

Last but not least the design methods:

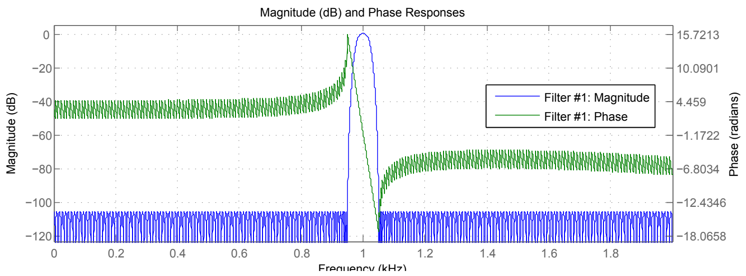
1. Equiripple
2. FIR least squares
3. Kaiser Window

The following is an analysis of these three methods. The analysis will determine which filter method suits the project best in terms of usage of hardware resources which is directly related to the filter order. Therefore, the parameter of concern is the filter order only.

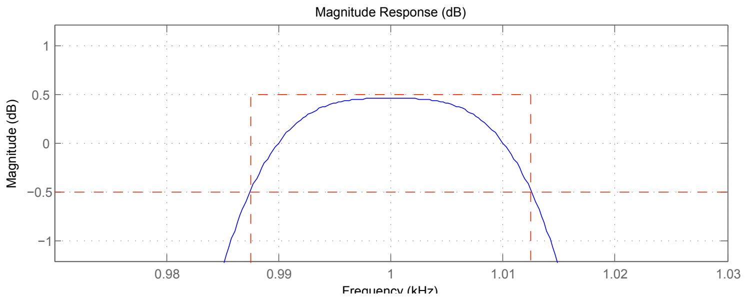
Figure 2.9 shows a FIR filter generated with the Equiripple algorithm that meets all requirements. Figure 2.10 shows a close up of the filter in the passband and highlights the linearity of the phase. This behavior is consistent for all all FIR filters. MATLAB has generated this filter with the minimum order necessary to meet all requirements which is 417.



(a) Magnitude response and requirements



(b) Magnitude and phase response



(c) Passband - ripple within 1 dB

Figure 2.9: Equiripple FIR filter design, minimum order = 417

Figure 2.11 shows a minimum order filter designed with the Kaiser window method. The close-up of the passband reveals an extremely smooth band with no visible oscillation, contrary to the Equiripple. But this behaviour comes with a price because the minimum order for this filter is 724. It can be seen that the attenuation of the stopband slowly increases (as it does for the IIR filter). By taking advantage of this the filter order can be reduced to 691 without violating the requirements. The yield is not that high since the order has decreased by 33 which corresponds to 4.5%. This trick can also be applied on the Equiripple

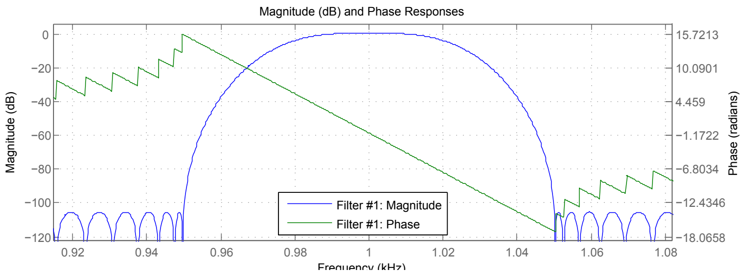
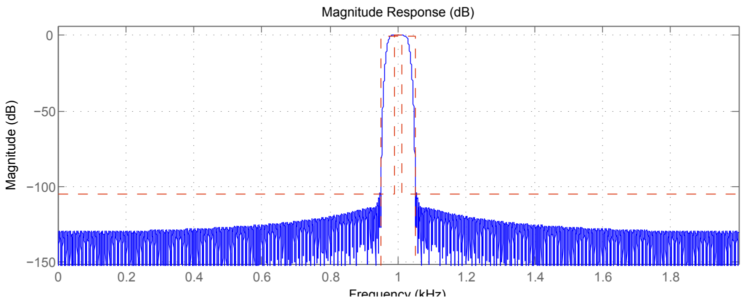
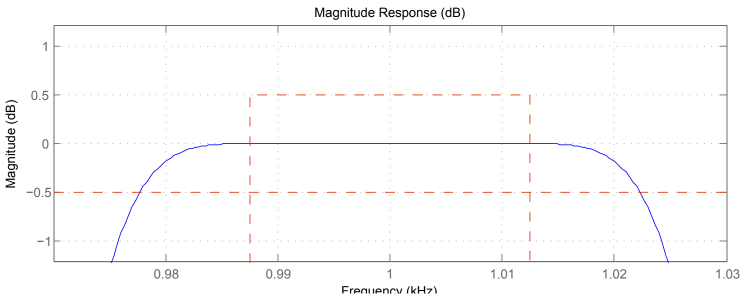


Figure 2.10: FIR filter linearity

algorithm but since the transition from the passband to the stopband is quite sharp only a reduction of few orders can be achieved.



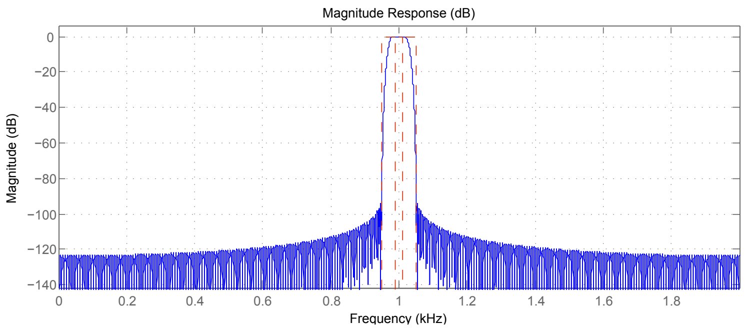
(a) Magnitude response and requirements



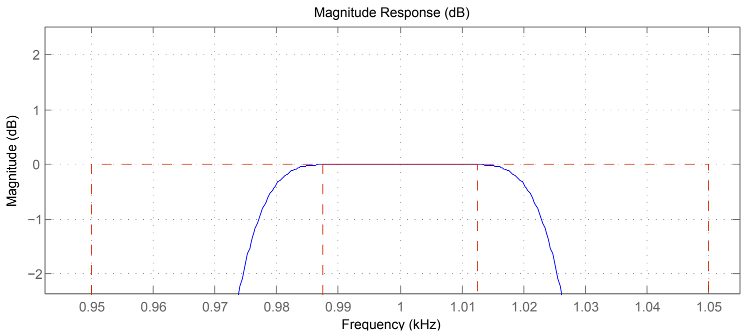
(b) Passband - no visible ripples

Figure 2.11: Kaiser-Window FIR filter design, minimum order = 724

The last method is the least-square method as depicted in figure 2.12. A minimum order options is not supported. The plots show a figure of order 600 which is in between the order number for the Equiripple and Kaiser-Window. All requirements are met. By looking at the passband it comes clear that it is just as smooth as for the Kaiser-Windows, but comes for a lower price in terms of order number.



(a) Magnitude response and requirements



(b) Passband - no visible ripples

Figure 2.12: Least-squares FIR filter design, order = 600

Summing up, the best filter method, in terms of lowest order, that meets the requirements is the Equiripple. The least-squares method offers a much better pass band behaviour, but for a high price. The Kaiser-Window is too costly. The figures in appendix A.5, A.6, A.7, A.8, A.9, and A.10 show the settings and information of the filters used in this chapter.

2.6 Chapter Concluding Remarks

In this chapter an overview of the different digital filter types was given. Based on the requirements (as stated in 2.3) the most suitable filter type for this project is the band-pass FIR filter. A variety of different algorithms (as presented in 2.5) for designing FIR filter exists. Since the analysis was restricted to MATLABs filterbuilder only the algorithms supported by it have been analysed. It was concluded that the best algorithm for this project is the Equiripple with a symmetric structure. A suitable filter architecture will have to support filters of an order up to 417 which is necessary to meet the TETRA requirements. Since the requirements for the TEDS are not as strict those filter's minimum order will, consequently, be lower.

CHAPTER 3

Implementation

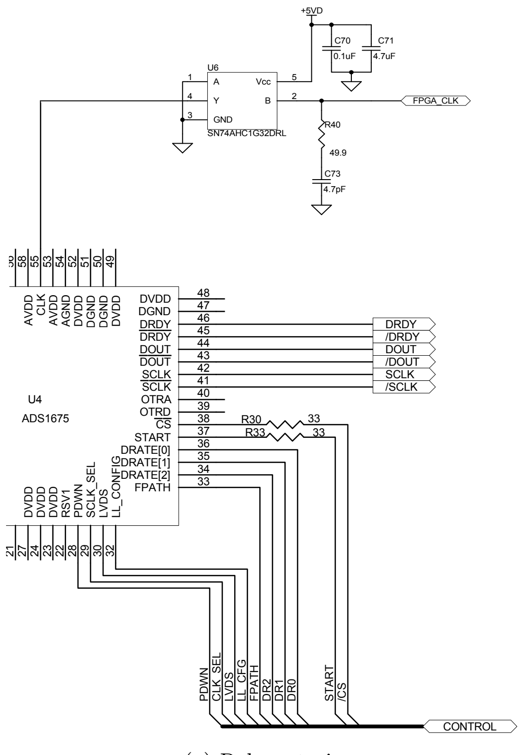
The first part of the implementation is to create an environment on the FPGA that is capable of communicating with the ADC and receive its samples. Furthermore, it should be possible to get hold of these samples, which requires another interface. The XEM3010 board comes with a USB port and predefined libraries to implement it with an hardware description language (HDL). By connecting the board to a PC via the USB port communication can be achieved by writing a program in XML and executing it with FrontPanel. Other programming languages are supported as well, such as C and JAVA, but since this interface is a small part of the environment a simple XML program will suffice. The chosen HDL is VHDL and implementation is done on the register-transfer level (RTL).

3.1 Creating an Environment

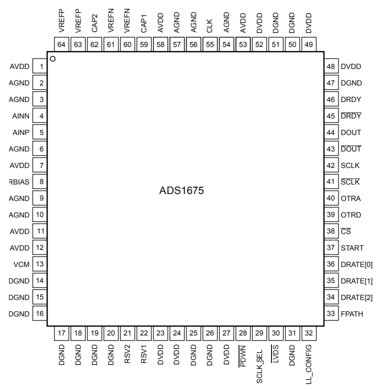
This chapter describes the HDL-implementation of an interface on the FPGA able to communicate with the ADC and a PC connected with an USB cable, in that order. The final VHDL source code can be found in appendix B.2.

3.1.1 ADC Interface

The layout of the ADC chip (ADS1675), shown in figure 3.1b, reveals the chip's pins. Of all the pins only some are relevant for the interface (pin 28 to 46 and 55), a schematic of these pins is shown in figure 3.1a.



(a) Relevant pins



(b) ADS1675 chip overview

Figure 3.1

The pins' specific purpose and function will be explained later in this chapter, for now it is enough to know that we have a set of control pins, a set of output pins and a clock driving the chip.

The first part of the interface is to assign the necessary pins in a UCF file. In order to create this file, it is necessary to know how the pins of the ADC are connected to the FPGA. As mentioned, the FPGA is part of the XEM3010 board which is mounted on the ADC board. They are connected through two 80-pin connectors. One of them serves as a passthrough and enables connection of other devices. The other connector enables connection between the FPGA and the ADC chip. A schematic of how the pins linked to the connector can be found in figure A.4 in appendix A. The pin enumeration of the connector (J6) corresponds to the pin enumeration of the FPGA chip.

In the UCF file physical pins have to be assigned by using their symbolic pin names, figure A.3 in appendix A shows table used to assign the pins. For correct instantiation of the pins, their function and behavior must be known. Table 3.1 shows the relevant pins together with a brief description. The resulting UCF-file defining the pin assignments is attached in appendix B.1.

No.	Name	Type	Description	Setting
28	PDWN	CMOS	Power down mode	Always low
29	CLK SEL	CMOS	SCLK generation	Set to '0' (internal generation)
30	LVDS	CMOS	Selects CMOS or LVDS behavior	Set to '1' (LVDS)
32	LL CFG	CMOS	Low latency filter behaviour (not used)	Set to '0'
33	FPATH	CMOS	Select wide bandwith / low latency	Set to '0' (WB)
34,35,36	DR2, DR1, DR0	CMOS	Select data rate	Set to "101" (fast rate)
37	START	CMOS	start sampling	set to $\overline{reset}$ (continuous sampling)
38	/CS	CMOS	Chip select	set to '0' (normal mode)
41, 42	/SCLK, SCLK	LVDS	DOUT clock rate (3 times the FPGA clock)	connected to LVDS buffer
43, 44	/DOUT, DOUT	LVDS	Data bit	connected to LVDS buffer
45, 46	/DRDY, DRDY	LVDS	Start of new data sample transmission	connected to LVDS buffer
55	FPGA_CLK	CMOS	chip driving clock	30 MHz

Table 3.1: Pin description

The control signals of the ADC define the mode of the chip. For this project the fastest high-speed mode (DRATE = 101) with the wide-bandwidth filter is used (see section 1.5). Since these control pins are directly connected to the FPGA they just need to be assigned in the VHDL implementation. The next step is to receive data from the ADC. For this, six pins are set aside. Since the mode is set to high-speed these six pins work as three differential LVDS pins. A

LVDS buffer has been implemented and outputs the signals: DOUT - one bit of data, SCLK - the clock at which the data bit is updated, DRDY - indicates the beginning of a new data sample. A data sample consists of 24 bits, hence, the DRDY signal will go high every 24th clock cycle when the ADC is active. The datasheet in [3] states that the very first sample after a reset should be ignored since it might be invalid. To satisfy this and to control the collection of the data samples a state machine has been implemented as shown in figure 3.2.

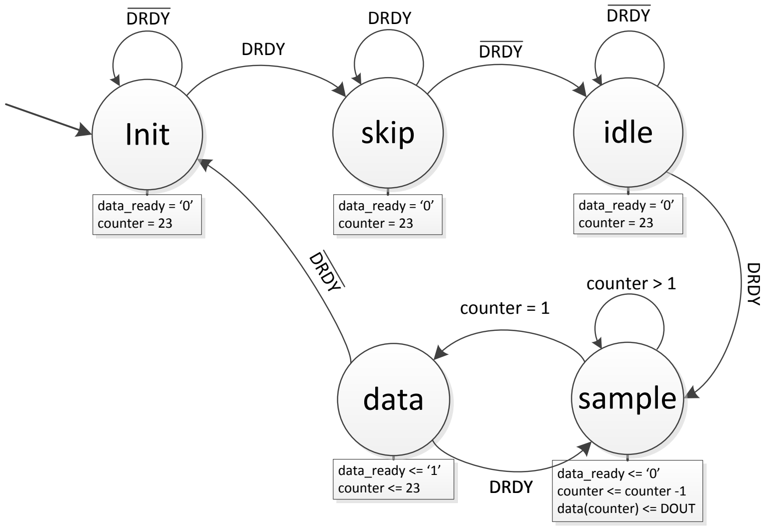


Figure 3.2: ADC sample receiving FSM

The following is a description of the functionality of the FSM: A system reset brings the FSM in the *init* state. Here it remains until a ready signal from the ADC is received (DRDY = '1'). When this happens state *skip* is entered in order to ignore the first sample (as mentioned above). When the ready signal goes low again (according to specifications it remains high for 2-4 clock cycles) the machine enters the *idle* state and is now ready to receive data as soon as DRDY goes high again by entering the *sample* state. The ADC sends the 24bit data sample bit by bit, a 24bit wide register is ready to receive them. For this purpose a counter counting from 23 and down is used which assigns the current data bit to the correct place in the 24 bit data register (this register is only enabled in the *sample* state). When 23 clock cycles have passed and 23 bits have been received the FSM enters the *data*-state and sets a ready signal high, indicating that a new data sample has been received, furthermore the sample is written to a register allowing the sample to be stable for the next 24 cycles. At this point the DRDY signal should go high again, and a new transmission should start (the machine would then go back to the *sample* state). If, for any reason, this should not be the case, the FSM goes to the *init* state instead, waiting for the ADC to send samples again. Recall, that the ADC is operating in high-speed mode. Hence, the LSB is not generated by the ADC and can be ignored. For this reason the *data* state is entered when the counter becomes 1 and not 0.

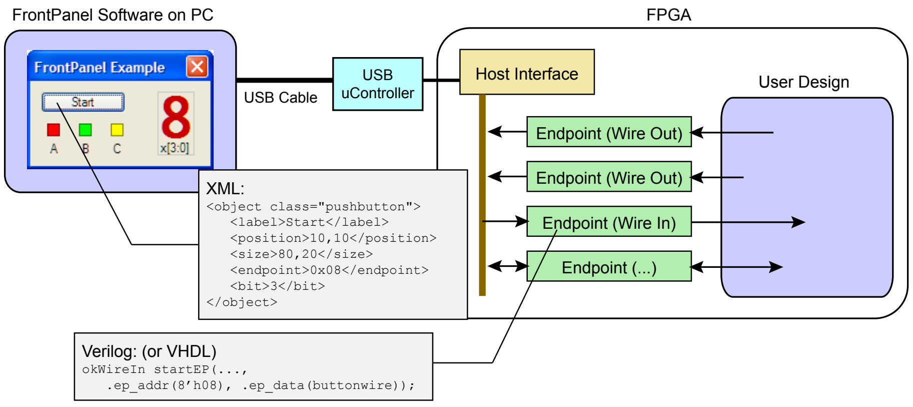
This state machine completes the ADC interface.

3.1.2 USB Interface

The next step is to implement a USB interface which enables the FPGA to communicate with a PC. Opal Kelly has mounted a USB connector on the XEM3010 board. For utilization of the USB port a set of libraries and components are available, that are ready for implementation. Opal Kelly calls it for the okHost interface as described in [6]. At the PC side Opal Kelly has provided the tool FrontPanel which can execute programs that can communicate with the oKhost module on the FPGA.

Figure 3.3 shows the overall structure. The left side holds the FrontPanel Software which has executed an example program written in XML. On the right side is the FPGA which contains the host interface elements and a user specific design. At this point the user specific design is the ADC interface as documented in the previous section. This user design is now being extended with the okHost elements to create the USB interface.

The okHost interface supports bi-directional communication enabling it to receive and send signals or data to and from the FPGA. The okHost component is the main component which can be connected to different kinds of endpoints. The okWireOr component is used to control the communication between the different endpoints and the okHost. Five different kind of endpoints are supported. The okPipes are used to send a series of data. They are perfect for unloading storage components on the FPGA to the PC or to fill them. There are two pipe components, one for sending and one for receiving. The okWires are used to continuously send or receive the current value of a signal (vector of 16 bits). The last type is the TriggerIn. It can be used to send up to 16 single-bit trigger signals which can be synced to any local clock. The other components



```
<object class="pushbutton">
   <label>Start</label>
   <position>10,10</position>
   <size>80,20</size>
   <endpoint>0x08</endpoint>
   <bit>3</bit>
</object>
```

```
okWireIn startEP(...,
   .ep_addr(8'h08), .ep_data(buttonwire));
```

Figure 3.3: USB interface overview [6]

are driven by the USB clock which is 48MHz. An overview of the components is shown in figure 3.4.

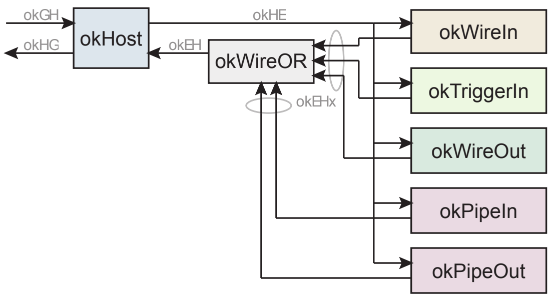


Figure 3.4: okHost components [6]

The first part of this interface is to send the value of a signal from the FPGA to the PC. This signal could be the register holding the samples received from the ADC. Since this sample is 24 bits wide, two okWireOut blocks are necessary (each block can only submit 16 bit wide signals). Furthermore, an okHost and an okWireOr block is necessary. This interface is part of the main VHDL file which can be found in appendix B.2. To receive the data a XML program has been written which can show the signal as a hex-value and as a binary value as shown in figure 3.5. At a later point the interface was extended with triggers and pipes, see section 3.2. The final XML code can be found in appendix B.4.

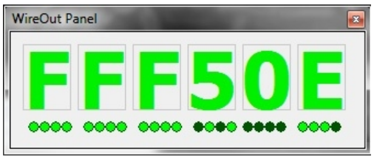
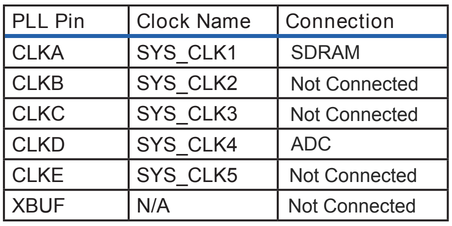


Figure 3.5: FrontPanel wireout example

3.1.3 Setting the Clocks

The XEM3010 comes with a PLL which is capable of generating five different clocks. These clocks can be set with the FrontPanel software. Figure 3.6 shows an overview of the clocks. CLKA is set 100 MHz and used to drive the SDRAM (see section 3.2). CLKD is connected to the ADC and set to be 30 MHz. The projects description defined the clock to be 30.24 MHz but the PLL can not generate such precise clock frequencies. Even though the dividers actually do support this frequency, measurements have shown that the resulting frequency is not the desired one when using large dividers, hence, 30 MHz will be used. The timing constraints created for the clock nets are based on the desired clock frequency (30.24 MHz).
The ADC generates an output clock, SCLK, which is 3 times its driving clock (CLKD). This clock is used to drive the ADC FSM. Last but not least there is the USB clock accessable through the okHost. All components interacting with the USB controller are driven by this clock.



PLL Pin	Clock Name	Connection
CLKA	SYS_CLK1	SDRAM
CLKB	SYS_CLK2	Not Connected
CLKC	SYS_CLK3	Not Connected
CLKD	SYS_CLK4	ADC
CLKE	SYS_CLK5	Not Connected
XBUF	N/A	Not Connected

Figure 3.6: XEM3010 PLL clocks [2]

3.1.4 Peripherals

The XEM3010 has 2 buttons and 8 LEDs onboard ready for use. These have been used heavily for debugging. Figure A.1 in appendix A shows a table describing their symbolic pin locations. Holding the two buttons at the same time will activated the reset signal, while holding the buttons individually has been used for transmitting the current value of a register. The LEDs have been used to show the clocks, which makes it easy to verify whether the board is working. Furthermore, the LEDs have been used to show different system statuses.

3.2 SDRAM

In order to verify that the implementation on the FPGA board works properly, sampled data has to be analysed. Since the system receives data in real time and the data cannot be analyzed in real time it has to be stored. This enables performing multiple analysis on the same data set. The board comes with 32MByte of SDRAM and the idea is to synthesize the SDRAM block and use it to store processed data. When enough samples are collected the contents of the memory will be read to a PC through the FrontPanel software. From here the data can be analysed with various tools, in this case MATLAB.

3.2.1 Implementing the Memory

In order to use the memory block a controller is necessary, this controller is not trivial to implement. Xilinx's CoreGenerator is able to generate memory controllers, but unfortunately it does not support SDRAM. The FrontPanel software comes with a few sample projects and one of them[1] actually implements the SDRAM, hence a controller. As it turns out, this project is designed with Verilog. Only the part containing the controller is of interest which needs to be extracted. Due to the structure of this project the easiest way of doing it is by extracting the relevant code of the top module and translating it to VHDL. The components used by the top-module written in Verilog can be used as they are.

The memory controller uses page-writes of 512x16 bits blocks. A FIFO is applied on the read and on the write part of the memory block. The controller is able to switch between read and write mode. Reading and writing at the same time is not supported. The SDRAM system is shown in figure 3.7. The top component of the SDRAM controller consists of the FIFO FAULT block, the FIFOs and a minor part of the SDRAM controller itself. The minor part consists of a state machine controlling the data flow between the controller and the FIFOs and some signal synchronisations. These parts have been rewritten in VHDL in order to integrate it in this project. Two changes have been made as well. The incoming data signal (DIN) of the FIFO on the write side (input) of the SDRAM has been changed from 16 bits to 64 bit. The second change is applied

[1]The project is named RAMteser

on the data pointer of the SDRAM. Instead of using a single pointer it now has two, one for reading and one for writing, which makes it more dynamic. The reason for the 64 bit wide input signal is the following: The output signal of the filter will be over 40 bit wide, depending on the length of the filter (this will come clear in section 3.3.4). A 48 bit wide signal would have been sufficient but since the CoreGenerator does not support 48/16 bit FIFOs a 64/16 bit FIFO has been chosen instead. The FIFOs size has not changed.

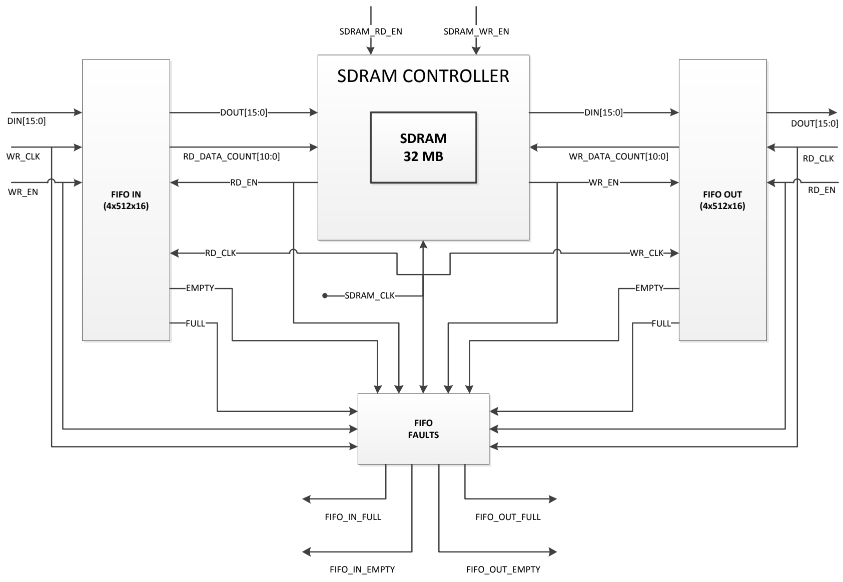


Figure 3.7: Memory configuration

Each FIFO can contain 4 blocks, which is sufficient since the SDRAM_CLK (set to 100 MHz) is much faster than the other clocks applied to the FIFOs. The output FIFO interacts with the USB communication unit (okHost) and runs at 48 MHz. The INPUT FIFO stores sampled data or data from the filter. These a produced at a clock rate of $30MHz \times 3/24 = 3.75MHz$. Since the width of the input vector has been widened to four times its original size, the bandwidth is four times as much, resulting in a relative frequency of 15 MHz. This is the rate at which data is send to the FIFO, the actual clock to which it is connected is $3 \times 30MHz = 90MHz$.
SDRAM uses refresh cycles to keep its contents stored. These refresh cycles reduce the effective bandwidth of the SDRAM by a negligible factor, hence over/underrun of the FIFOs will not be a problem.

The controller is supposed to read one block of data of the input FIFO as soon as it contains at least one block of data, and write one block of data if the output FIFO has room for at least on more block of data. If, for any reason, this rule is not obeyed the FIFO FAULT block sets a corresponding alarm register high until next reset. There are four of these registers:

1. FIFO_IN_FULL goes and stays high if WR_EN of the input FIFO is triggered even though the FIFO is full.
2. FIFO_IN_EMPTY goes and stays high if RD_EN of the input FIFO is triggered even though the FIFO is empty.
3. FIFO_OUT_FULL goes and stays high if WR_EN of the output FIFO is triggered even though the FIFO is full.
4. FIFO_OUT_EMPTY goes and stays high if RD_EN of the output FIFO is triggered even though the FIFO is empty.

For debugging purposes these four values have been connected to a LED on the board.

The use of the memory block will be as follows:

1. Set the controller to write mode and begin to sample
2. When a sample is received from the ADC write it to the input FIFO[2].
3. If the FIFO contains at least one block[3] write it to the Memory.
4. When enough samples are taken change the controller to read mode.
5. The controller reads data from the memory and writes a block to the output FIFO when it has room for at least one more block.
6. A FrontPanel script reads the data at the output FIFO through the USB interface and stores it in a file on the PC.

A counter is used to keep track of the amount of samples written to the SDRAM. When a predefined number is reached it will stop writing data. In the final design this number has been set to a value corresponding to 32 MB of data contents in the SDRAM. When reading data with the FrontPanel 128 kB data is read.

[2]The input FIFO is the FIFO containing the data that will be written to the memory
[3]512*16bits

This number can be changed to any desired value in the source code. Figure 3.8 shows the final design of the graphical USB interface script.

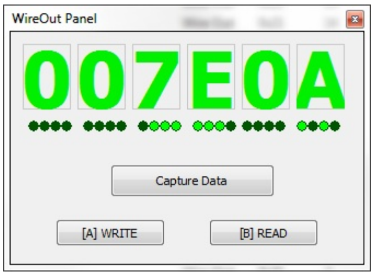


Figure 3.8: Graphical interface designed with FrontPanel

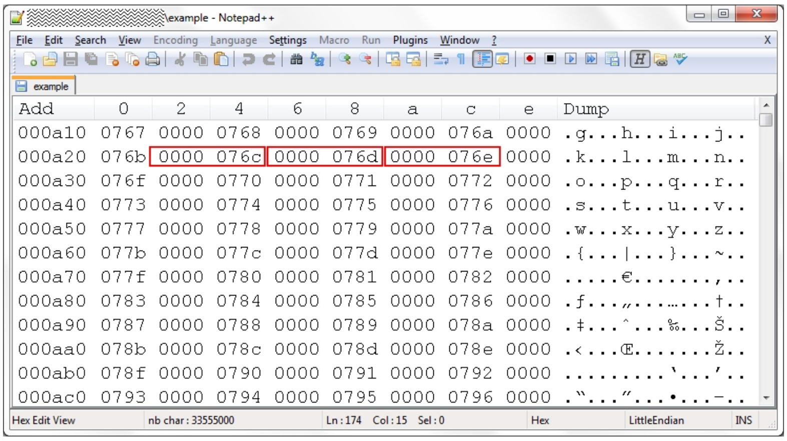
The last two digits of the hex value correspond to the state of the memory. In state [A] the memory is in write mode and in state [B] it is in read mode. The first four digits correspond to the value of the address write-pointer. In this specific case $7E_{hex}$ pages haven been written to the memory. By pressing the button in the lower right corner labelled "[B] READ" the controller will switch to read mode. The hex value will now show the write-pointer address. At this point it is possible read data from the memory by pressing the "Capture Data" button. The write pointer will increase accordingly. Furthermore a file is created containing the memory dump.

3.2.2 Timing Constraints

The timing adjustmens between the SDRAM and the rest of the sample project (from which the controller was extracted) consist of a clock buffer applied on the SDRAM clock and a timing constraint specified in the UCF file. When applying the SDRAM controller to this project the data received from the SDRAM block became invalid. After some testing this could be lead back to timing issues. By defining the set and hold times specified by the memory block vendor [1] along with I/O optimizations valid timings were achieved. Xilinx's Timing Constraints User Guide [11] was used as a reference for this. A lot of tests and time was necessary to achieve this. It should be noted than when the clocks used in this project get replaced by others, valid timing can not be guaranteed.

3.2.3 The Memory Dump

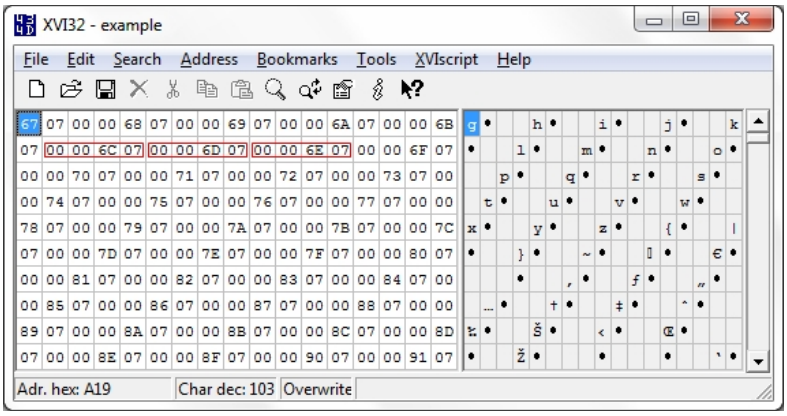
As mentioned, the FrontPanel script is able to create a dump file of the memory's contents. Such a file consists of raw unformatted data. To access the data two different tools were used. Notepad++ with a hex-plugin, used to get quick access to the data, shows the binary data as hex-values. As it turns out it is not possible to save the data as an ASCII formatted text file. For this purpose the tool XVI comes in play. After opening the file it is possible to export the data into an ASCII formatted file by using the print function. Unfortunately, XVI interprets the hex data in a different way than Notepad++ (the correct way), the order of the data is mixed up. Since the exported ASCII file is going to be used in MATLAB for further analysis, the data can be brought in the right order by MATLAB. In figure 3.9 and 3.10 the tools' data representation is shown.


\example - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
example

```
Add     0    2    4    6    8    a    c    e    Dump
000a10  0767 0000 0768 0000 0769 0000 076a 0000 .g...h...i...j..
000a20  076b 0000 076c 0000 076d 0000 076e 0000 .k...l...m...n..
000a30  076f 0000 0770 0000 0771 0000 0772 0000 .o...p...q...r..
000a40  0773 0000 0774 0000 0775 0000 0776 0000 .s...t...u...v..
000a50  0777 0000 0778 0000 0779 0000 077a 0000 .w...x...y...z..
000a60  077b 0000 077c 0000 077d 0000 077e 0000 .{...|...}...~..
000a70  077f 0000 0780 0000 0781 0000 0782 0000 .....€.......,..
000a80  0783 0000 0784 0000 0785 0000 0786 0000 .f...„...…...†..
000a90  0787 0000 0788 0000 0789 0000 078a 0000 .‡...ˆ...‰...Š..
000aa0  078b 0000 078c 0000 078d 0000 078e 0000 .‹...Œ.......Ž..
000ab0  078f 0000 0790 0000 0791 0000 0792 0000 .........‘...’..
000ac0  0793 0000 0794 0000 0795 0000 0796 0000 .“...”...•...–..
```

Hex Edit View | nb char : 33555000 | Ln : 174 Col : 15 Sel : 0 | Hex | LittleEndian | INS

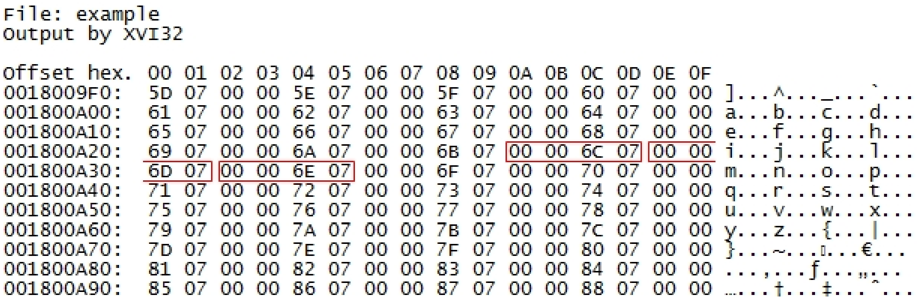
Figure 3.9: Memory dump shown in Notepad++ with hex-plugin



```
67 07 00 00 68 07 00 00 69 07 00 00 6A 07 00 00 6B
07 00 00 6C 07 00 00 6D 07 00 00 6E 07 00 00 6F 07
00 00 70 07 00 00 71 07 00 00 72 07 00 00 73 07 00
00 74 07 00 00 75 07 00 00 76 07 00 00 77 07 00 00
78 07 00 00 79 07 00 00 7A 07 00 00 7B 07 00 00 7C
07 00 00 7D 07 00 00 7E 07 00 00 7F 07 00 00 80 07
00 00 81 07 00 00 82 07 00 00 83 07 00 00 84 07 00
00 85 07 00 00 86 07 00 00 87 07 00 00 88 07 00 00
89 07 00 00 8A 07 00 00 8B 07 00 00 8C 07 00 00 8D
07 00 00 8E 07 00 00 8F 07 00 00 90 07 00 00 91 07
Adr. hex: A19   Char dec: 103   Overwrite
```

Figure 3.10: Memory dump shown in XVI32

The data consists of a 32 bit wide counter value. In the first picture three values are marked. The same three values are marked in the other picture as well, but the order of the numbers is messed up. Each values consists of 8 hex numbers $X = x_1x_2x_3x_4x_5x_6x_7x_8$. In XVI32 the order has become the following $x_3x_4x_1x_2x_7x_8x_5x_6$. This can be seen in the next figure 3.11 as well. Here a piece of the exported ASCII file is shown. This file will be the one loaded into MATLAB and after correcting the order of the digits the data can be analyzed.



```
File: example
Output by XVI32

Offset hex.  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0018009F0:   5D 07 00 00 5E 07 00 00 5F 07 00 00 60 07 00 00 ]...^..._...`...
001800A00:   61 07 00 00 62 07 00 00 63 07 00 00 64 07 00 00 a...b...c...d...
001800A10:   65 07 00 00 66 07 00 00 67 07 00 00 68 07 00 00 e...f...g...h...
001800A20:   69 07 00 00 6A 07 00 00 6B 07 00 00 6C 07 00 00 i...j...k...l...
001800A30:   6D 07 00 00 6E 07 00 00 6F 07 00 00 70 07 00 00 m...n...o...p...
001800A40:   71 07 00 00 72 07 00 00 73 07 00 00 74 07 00 00 q...r...s...t...
001800A50:   75 07 00 00 76 07 00 00 77 07 00 00 78 07 00 00 u...v...w...x...
001800A60:   79 07 00 00 7A 07 00 00 7B 07 00 00 7C 07 00 00 y...z...{...|...
001800A70:   7D 07 00 00 7E 07 00 00 7F 07 00 00 80 07 00 00 }...~...▯...€...
001800A80:   81 07 00 00 82 07 00 00 83 07 00 00 84 07 00 00 ....‚...ƒ...„...
001800A90:   85 07 00 00 86 07 00 00 87 07 00 00 88 07 00 00 …...†...‡...ˆ...
```

Figure 3.11: Memory dump exported to ASCII with XVI32

3.2.4 Precision

According to the ADS1675 reference guide [3] 14.33 noise-free bits are guaranteed when using the high speed mode. By using the implemented environment this could be verified in the following way: A simple state machine was added to the design which keeps track of the lowest and biggest sample value received. These extrema are stored in registers and can be send to the FrontPanel software by using the buttons of the XEM3010 board. Figure 3.12 shows extrema sampled over a few seconds until the values became stable, no input signal applied. The hex-values are of type 2's complement.

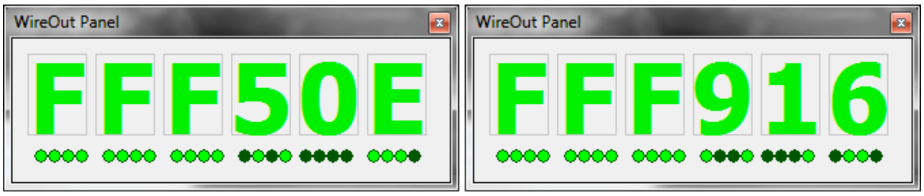


Figure 3.12: Extreme values: minimum (left), maximum (right)

The first observation that comes to mind when looking at the two values is that both are negative. This indicates some offset (DC-noise), otherwise the maximum value would be positive. By assuming that the non-DC noise influences the signal equally in both directions (positive and negative) the noise swing becomes:

$$V_{noise} = \pm\frac{V_{max} - V_{min}}{2} = \pm\frac{\mathrm{FFF916}_{hex} - \mathrm{FFF50E}_{hex}}{2} = \pm 516_{dec}$$

The number of bits necessary to represent this value is:

$$N_{bits} = log_2(516) bits = 9.01 bits$$

Therefore, the number of noise-free bits comes to:

$$N_{noise-free} = (24 - 9.01) bits = 14.99 bits$$

Which is within the range guaranteed by the reference guide and corresponds to a dynamic range of 90 dB.

3.2.5 Data Quick Test

The state machine used for storage of the extreme values as mentioned in the previous section has been used for debugging purposes as well. When sampling a predefined number of values and storing them in the SDRAM these extreme values should be present in the SDRAM. After loading the contents of the SDRAM to the PC and investigating the data with a hex-editor a quick sanity check could be performed by verifying that the two extreme values are present in the data. This quick test came in handy when resolving the timing issues which occurred when implementing the SDRAM controller.

3.3 Implementation of Filters

The realisation of a filter consists of a series of multiplications and additions. The operands for the multiplications are the incoming data samples generated by the ADC and the coefficient constants defining the filter. Multiplication with a constant can with advantage be done by a chain of shifters and adders. Since the filters in this projects are not fixed their coefficients are not constant. Therefore, hardcoded multiplication is not an option. Instead the 32 multipliers embedded on the FPGA are going to be used.

3.3.1 Data types

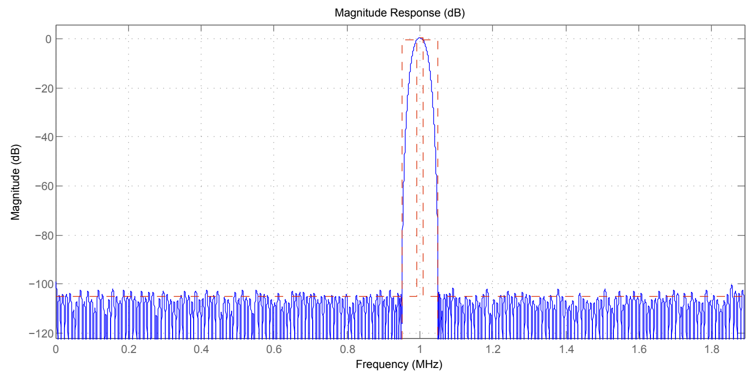
A decision has to be made regarding what number representation system (NRS) is going to be used. The classic types are floating point and fixed point (such as 2's complement). Since the data from the ADC is of 2's complement the obvious choice for this project is fixed point. The filter coefficients will, consequently, also be of a fixed point form. Fixed point systems do have the advantages of higher speed and reduced complexity. Furthermore, the multipliers on the FPGA are designed for fixed point multiplication.

3.3.2 Quantization

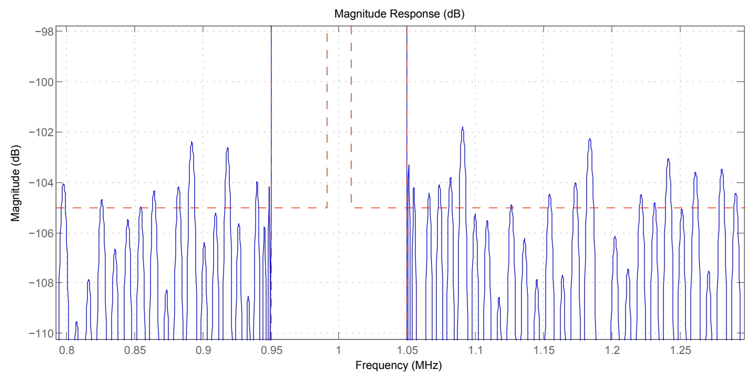
The 32 multipliers on the FPGA can multiply two 18-bit operands and produce a 36-bit product, hence full precision. The intuitive choice for data and coefficient width is, consequently, 18 bit. The data transmitted by the ADC is 24 bit wide (23 in high speed mode) but a precision analysis has shown that only 15 bits are noise-free. Therefore, the signal will be trimmed down to 17 bits. It might seem odd to chose 17 and not 18 bits, but since the filter will be of a symmetric type, the values fed to the multipliers will be a sum of two samples. To avoid overflow problems, 17 bit is the safe choice.

Setting the signal-width for the filter coefficients to 18 bits will influence the filter due to quantization effects. These effects have been analyzed in a graphical way by using MATLAB. The same filter specifications as declared in section 2.5 are used, but the data type has been changed from real to fixed point.

Figure 3.13 shows the result of changing the data type to fixed point.



(a) Magnitude response and requirements



(b) Close-up

Figure 3.13: Quantized Equiripple FIR filter

As can be seen, a 18 bit resolution is not enough to maintain the requirements since the -105dB attenuation is not withhold. Choosing a wider vector for the filter coefficients will solve this issue. Tests have shown that 24 bits seem to sufficient for all 5 channel types. A more mathematical approach as described in [5] can be applied to assure the quantization effects are within limits. For this project, which is primarily about proving a concept, 18 bit wide coefficients will suffice.

The fixed point representation used by MATLAB is of a fractional nature. The radix point, separating the fractional part from the integer part, is placed outside of the 18 bit vector (the coefficients are less than 1). How far outside changes with the channel width of the signal (TETRA / TEDS). In a multi carrier system this has to be accounted for, otherwise the filter outputs are not comparable.

3.3.3 Filter Resolution

A resolution of 250 Hz was required for positioning the passband in the frequency band. MATLAB tests have shown that the filter coefficients change even when moving the passband 10 Hz and the precision was found to be < 50 Hz. Hence, a resolution of 250 Hz is granted.

3.3.4 Filter Architecture

There are different ways in implementing a filter's structure. Basically one can divide the implementation methods into three main categories: parallel, sequential/serial or a combination of both. A parallel implementation is the fastest in terms of latency but requires a lot of hardware resources. A serial implementation reuses hardware but has a bigger latency (assuming the same clock frequency is used). The third method is a combination of the first two and has the benefits and drawbacks of both depending on the ratio between the serial and the parallel part.

The FPGA has a limited amount of embedded multipliers but a basically unlimited amount of adders[4]. Hence, the amount of multiplications is much more critical than the amount of additions. Since there is no requirement about minimizing the hardware utilization of the FPGA the best performance is achieved by using all multipliers in parallel. Since this is not enough the design has to be serialized as well. For this purpose the behaviour of the ADC's data transmission becomes a great advantage. The ADC transmits one bit at a time and it takes 24 cycles to complete a sample transmission. The data-bit is synchronised to a clock generated by the ADC itself. By using this clock signal for the serialization part the multipliers can be reused up two 24 times for each data sample. This of course requires the hardware being able to run at that clock frequency.

Since the critical factor is the multiplication, a filter structure should be chosen which does the best use of the multipliers. This type is called the symmetric

[4]The number of adders is limited to the amount of logic cells available on the FPGA

structure which reduces the amount of multipliers to half as described in section 2.2.

The minimum filter order required was found to be 417, hence a lot more multipliers than 16 are needed, 209 to be precise. Since the limit of parallel multiplications is defined by the multipliers embedded, sequential reuse of the multipliers is necessary. By using all 24 cycles and the 32 multipliers up to $32 \times 24 = 768$ multiplications can be performed per data sample. This would be enough to implement three filters.

The coefficients defining the filters need to be stored somewhere. To do this, block memory designed with the CoreGenerator is used. For this project read-only memory blocks will be used to hold predefined coefficients generated with MATLAB. Since several coefficients are going to be used each cycle (1 for each multiplication) the output of this block memory will be a multiple of 18 bits. For a filter with 209 coefficients $\frac{209}{24} = 8.67 \simeq 9$ multiplications each cycle are necessary. This would require a memory output width of $9 \times 18 = 162 bits$. As it turns out, the CoreGenerator does not support a 162 bit wide output when using 18 bit vectors. For this reason, the filters used for the FPGA implementation will be of a slightly lower order, namely 383. In a final product the coefficients will not be stored hardcoded as ROMs on the FPGA, and, consequently, this design choice has no influence on the final design, as long as the architecture is flexible enough to support filters of a bigger order than 383. Setting the filter order to 383 results in $\frac{192}{24} = 8$ multiplications per cycle. Hence, the output vector of the ROM is defined to be $8 \times 18 = 144$ bits.

The type of the order number (even or odd) does influence the symmetric filter structure. Since the minimum order was found to be odd the filter structure used will only support filters of an odd order. The number of multiplications (tapsums) is defined by $T = \frac{L+1}{2} = \frac{383+1}{2} = 192$ where L is denotes the order. The output of the filter is described by the following equation:

$$y(n) = \sum_{k=0}^{T-1} f_k \left(x(n-k) + x(n-L+k) \right) \tag{3.1}$$

By dividing this equation into a parallel and serial part it can be rewritten as follows:

$$y(n) = \sum_{c=0}^{C-1} \left(\sum_{m=0}^{M-1} f_{Mc+m} \left(x(n-Mc+m) + x(n-L+Mc-m) \right) \right) \tag{3.2}$$

where $C = 24$ denotes the number of cycles (serial part) and $M = 8$ the number of multipliers (parallel) part.

Figure 3.14 shows the suggested architecture based on equation 3.2 applied on this project, the VHDL implementation can be found in appendix B.3.

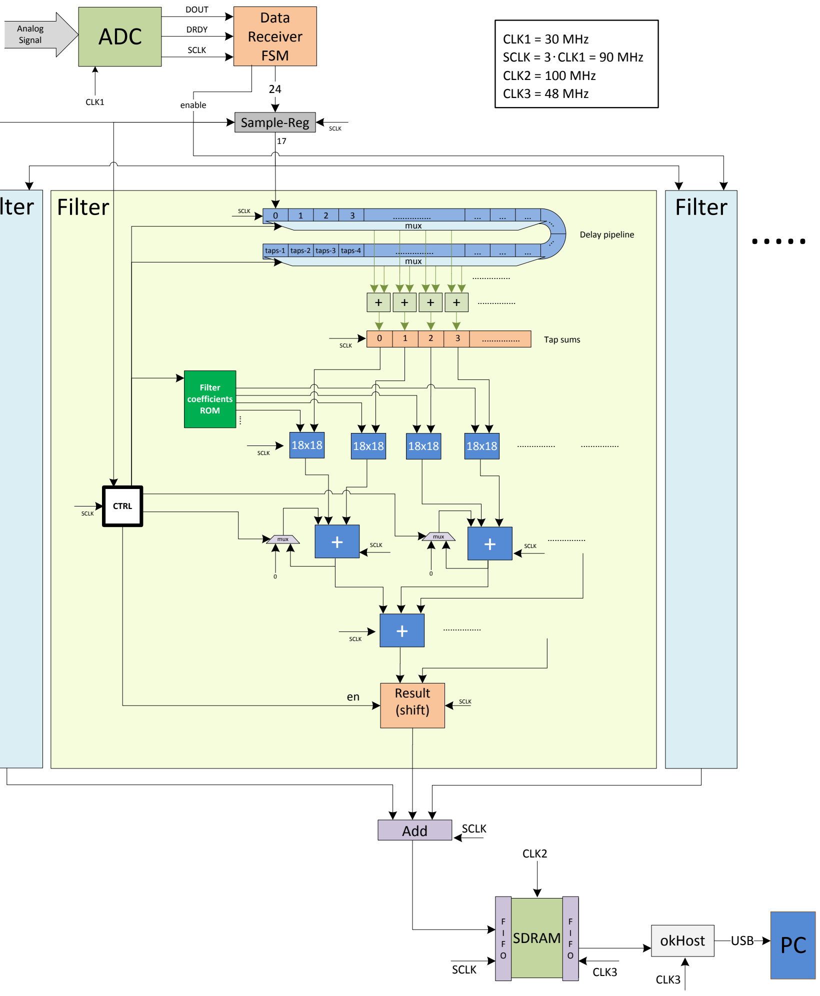


Figure 3.14: Filter Architecture

The design allows the number of multipliers to be $\{2, 4, 6, 8, 10, 12\}$ which allows filters of an order up to $L_{max} = 2(M_{max} \cdot C) - 1 = 2(12 \cdot 24) - 1 = 575$. This is based on the number of cycles being fixed at 24. A clock divider/multiplier can easily be added in the design making it possible to vary the filter order even more. Since this is a change in the serial part the number of cycles could basically be unlimited, it just depends on the restrictions set by the FPGA, which is based on the required throughput. The control unit requires the number of cycles not to be less than 4.

The VHDL hardware design is extremely flexible and is formed by a set of generics. The following can be defined through the generics:

- The width of the input signal (set to 17)
- The width of the input signals of the multipliers which should be equal the input signal width +1, or wider (set to 18)
- The width of the filter coefficients which should be less or equal the width of the multiplier inputs (set to 18)
- The amount of left-shift operations performed on the output (only relevant for multi-filter systems)
- The number of multipliers used in parallel (set to 8)

The width of the filter inputs must of course fit the multiplier type available on the specific FPGA. All internal signals are generated with full precision which results in a quite wide output signal depending on the filter order. The output is also of full precision, no truncation or rounding is applied. The architecture is based on a pipelined design of six steps. The last step creates the final result and applies a left-shift operation, if required. This shift operation is necessary when implementing filters with different radix point placements since the results are added before they are written to the SDRAM.

3.4 FPGA Resources

Implementation-tests have shown that the FPGA used for this project can accommodate two filters, even though the FPGA basically has enough block memory for the coefficients and multipliers to hold three filters. The problem lies in the sharing of the resources. The following error message occurs when trying to synthesizing three filters:

ERROR:Place:665 *The design has 19 block-RAM components of which 17 block-RAM components require the adjacent multiplier site to remain empty. This is because certain input pins of adjacent block-RAM and multiplier sites share routing resources. In addition, the design has 24 multiplier components. Therefore, the design would require a total of 41 multiplier sites on the device. The current device has only 32 multiplier sites.*

As a result of this issue the design will be limited to a multi-carrier system of two carriers. When selecting another board for a final product, this issue should be kept in mind. However, this might not even be an issue in newer FPGAs. The Spartan 3 chip in this FPGA is of a rather old family. Xilinx's datasheet for this chip even states that the Spartan 3 should not be used for never designs.

3.5 Chapter Concluding Remarks

In this chapter a basic environment was created on the FPGA. This environment consists of an interface site capable to communicate with the ADC, to store processed data in the SDRAM, and to unload the data to a PC through a USB interface. Furthermore, a filter architecture was designed an implemented allowing flexibility in the size of the filter. Implementations based on up 12 multipliers working in parallel are supported. An implementation of 8 multipliers is used in this project connected to a driving clock of 90 MHz corresponding to an effective clock of $8 \cdot 90MHz = 720MHz$. These 8 multipliers are used in a serial loop of 24 cycles which corresponds to 192 multiplications per data cycle. Given a symmetric structure this enables implementation of filters of order 383.

Chapter 4

Testing

This chapter is all about verifying the implemented architecture through tests. These tests are based on simulations performed in Modelsim and MATLAB. Furthermore, measurements are verified. The first section deals with the state machine receiving the data samples from the ADC. Afterwards the filter architecture will be tested.

4.1 ADC FSM

This section verifies that the state machine used to receive the samples from the ADC has been implemented correctly by simulating its behaviour in Modelsim. For clarity reasons the FSM diagram is shown again in figure 4.1.

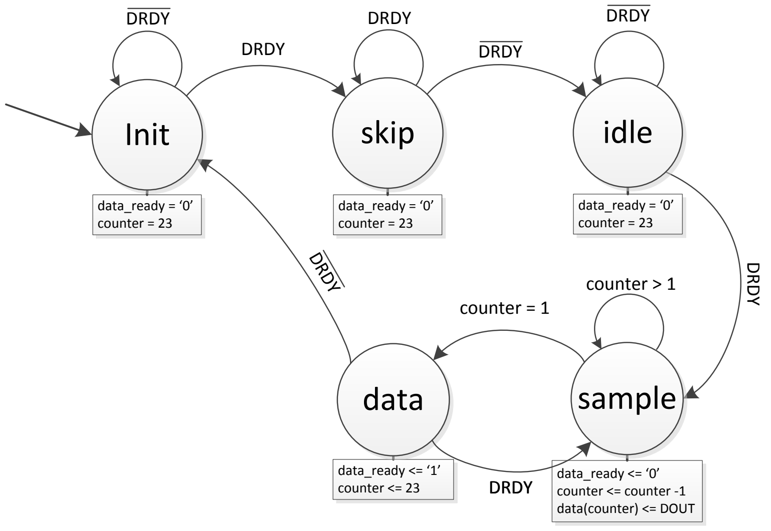


Figure 4.1: ADC sample receiving FSM

In order to verify correct behaviour, two parts must be tested. The first part is to verify that the state machine goes through the correct states. The second part is to see if the correct output is generated.

Figure 4.2 shows a Modelsim screenshot. The system starts with a reset and the *init* state is entered. Afterwards the DRDY signal goes high, thereby indicating the beginning of a data sample. The very first sample should be ignored. For this reason, the machine enters the *idle* state as a response to DRDY going low. The machine remains in this state until DRDY goes high again. As can be seen in the Modelsim snapshot, there exists a DOUT_buffer. This signal is necessary because the counter defining where to store the incoming bit is delayed one cycle compared to the incoming data bit. To synchronise them, the incoming data bit is delayed together with DOUT.

When DRDY goes high the second time, the *sample* state is entered. The first bit transmitted is DOUT = 1, which is the MSB. One clock cycle later the data signal changes from its reset value (00FF00) to 80FF00. The counter now keeps decreasing each cycle while DOUT is 0. The data signal changes accordingly.

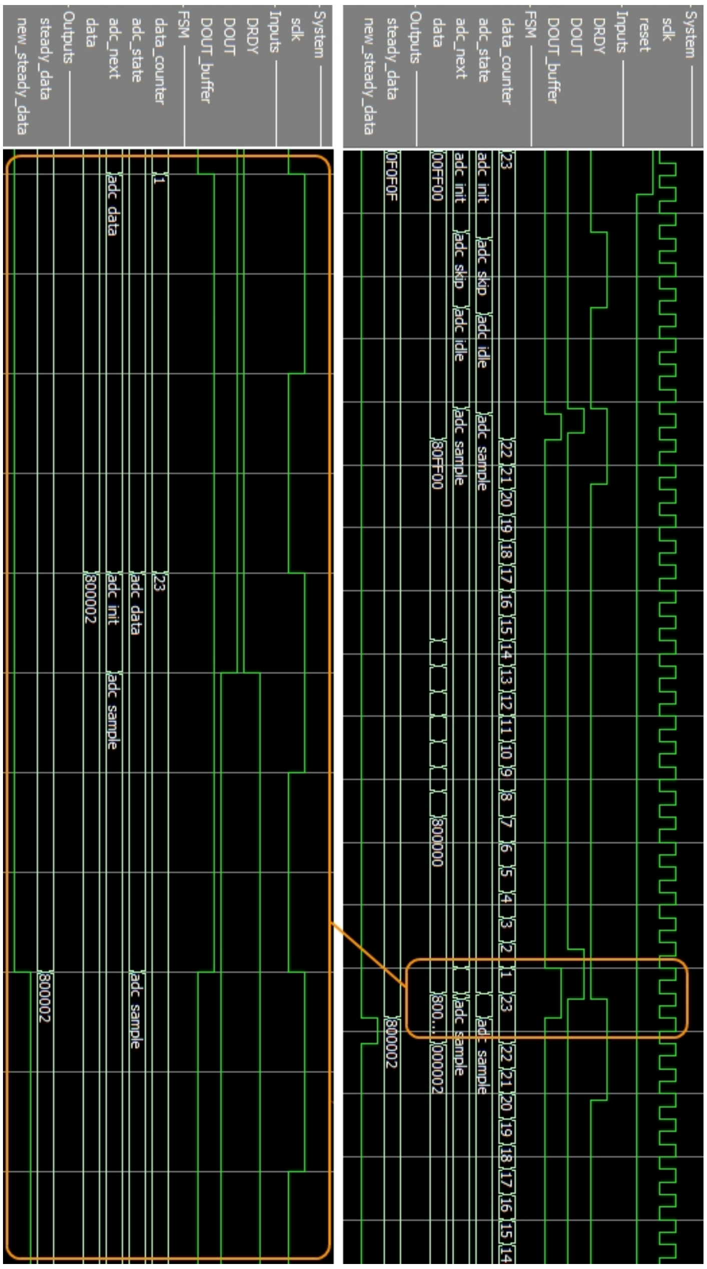


Figure 4.2: Modelsim screenshot showing the behaviour of the ADC FSM

Things become interesting when the counter reaches 1. Data(1) has been set to DOUT=1. The next state is determined to be the *data* state and DOUT is still high. The LSB should not be captured since it is not set by the ADC. To verify, that the FSM does not sample the LSB, the incoming data is set high in this simulation. The Modelsim screenshot shows, that the data's LSB is not being sampled. This verifies correct behaviour. During the *data* state the next state is determined to be the *init* state, but as DRDY goes high it changes to the *idle* state instead. Furthermore, the new_steady_data and the steady_data signal is updated. Correct behaviour has therefore been verified.

4.2 Filter Architecture

An easy and extremely efficient way to verify a correctly implemented filter architecture is to apply a unit impulse function and analyse the filter output. The filter output corresponds to the impulse response which should be equal to the filter's coefficients.

To do this, a test filter has been designed. Its first and last coefficients are shown below:

```
15, 8, -15, -14, 8, 30, 1, -37, -25, 39, 52, -21, -80,
-15, 94, 71, -81, -131, 31, 179, 56, -188, -168, 140,
275, -24, -340, -148, 325, 343, -204, -505, -23, 573,
320.......................................................................
......................... -61995, 31272, 78981, 6064, -78190,
-44016, 58849, 73453, -24707, -86896, -16684, 80456,
55695, -54946, -82931, 15843, 91560, 27920, -79080,
-66071, 48028, 89489, -5458, -92464, -38719, 74130
```

Figure 4.3 shows the output of a simulated impulse function. The first few cycles and the center of the response is shown. It can be seen, that the data signal has the value 1 for a single cycle and 0 for all others. By comparing the result values to the coefficients above it can be shown that the filter works as expected. The first four values are 15,8,-15,-14 in both cases.

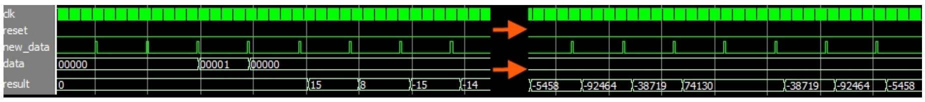


Figure 4.3: Impulse response simulated with Modelsim

The plots then skips a lot of cycles. The value 74130 is received for two clock cycles indicating the center of the impulse response which corresponds to the last filter coefficient. Afterwards the coefficients are received in reverse order.

Even though this test confirms correct behaviour, it has not tested the entire architecture. The delay pipeline is filled with a lot of zeros and a single 1 rippling through it. Hence, only one multiplier and some of the adders are used each cycle. To utilize the entire architecture, another test is performed. This test sets the input to 0 and changes it to 1 at some point, and corresponds to a unit step function. The filter will respond to this change and after a while it will assume a stable value. To find this value, a MATLAB script has been written which can be found in [10]. The outcome of the script is shown in figure 4.4. The first plot shows the entire step response while the second one is a close up of the end of the response.

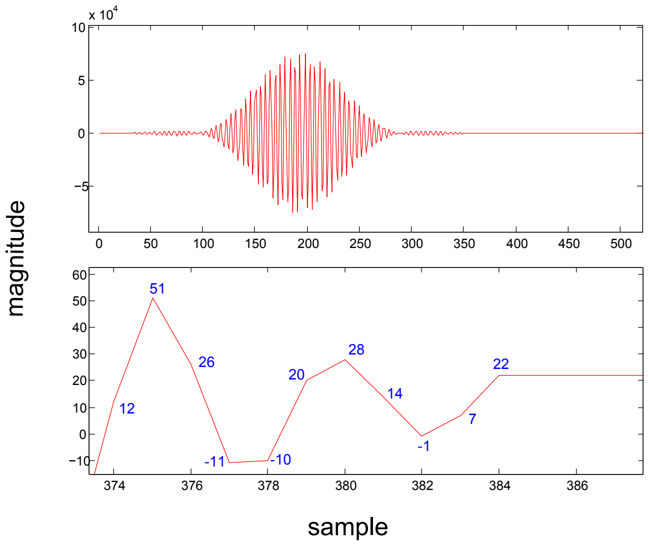


Figure 4.4: MATLAB simulated response of a test filter.

It can be seen that the filter settles at the value 22. Figure 4.5 shows a Modelsim simulation of the filter. Clearly the implemented architecture settles at the same value.

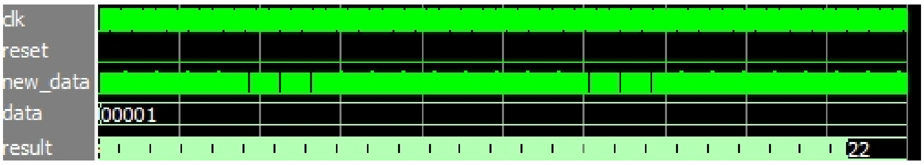


Figure 4.5: Test filter response in Modelsim

Figure 4.6 shows a close-up of the last cycles. The values match the values generated with MATLAB. Correct filter implementation has therefore been verified.

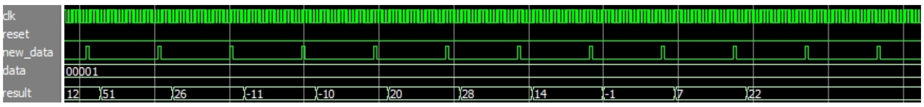


Figure 4.6: Test filter response in Modelsim - close-up

Last but not least the same tests have been performed on the FPGA. By analyzing the contents of the SDRAM with MATLAB it could be verified that the contents hold the same values as the Modelsim simulations.

Chapter 5

Results

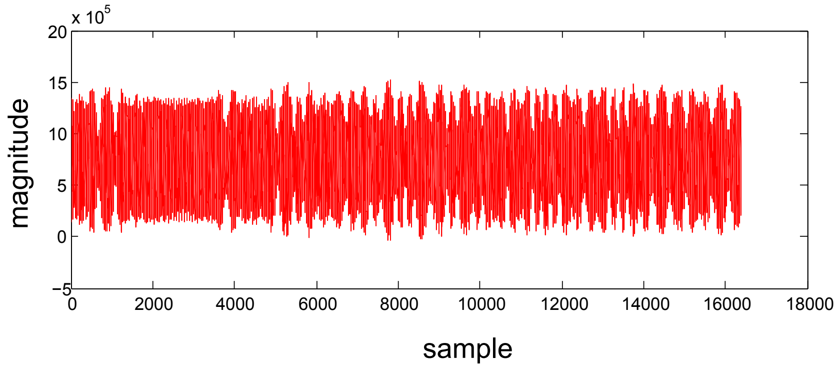
This chapter presents the results obtained with a set of filters implemented with the architecture presented in 3.3.4. The signal applied on the input terminals of the ADC is generated by two signal generators, one for TETRA/TEDS signals and the other for blocker signals represented by a sine wave. The filters used are designed with MATLAB's *filterbuilder* tool and the filter coefficients are implemented as read-only block memory generated with the Core-Generator. Due to the nature of the memory, as explained in section 3.3.4, the filters' order is set to 383. As stated in section 2.5, a filter order slightly bigger than this is necessary to meet all requirements. Furthermore, quantization effects due to the 18-bit wide coefficients, will influence the results as well. Since these alterations only affect the stop-band, all requirements, except for -105 dB stop-band attenuation, should be met.
In order to use different filters, a set of bit-files was created, each containing one or two filters. Furthermore, a bit-file without filters was used to sample unfiltered data for comparison. By loading the different bit files to the FPGA, the filters applied could be changed. The measurements have been taken as described in section 3.2, which can be broken down to the following:

1. Store sampled data (filtered or unfiltered) in the SDRAM.
2. Read SDRAM content and export it as a file to a PC
3. Analyse data file with MATLAB

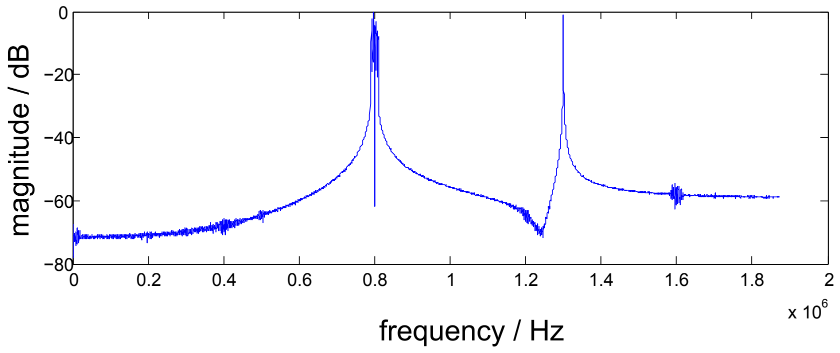
Consequently all the plots shown in this chapter are based on data sampled with the FPGA, unless otherwise stated. Furthermore, all plots have been normalized, hence the maximum value occurring in a set of sampled data corresponds to 0 dB.

5.1 Single Carrier

This section shows results gained with a single filter applied. The first plots (see figure 5.1) show sampled data (no filter applied) of a TETRA channel located at 800 kHz and a blocker at 1300 kHz. The peak-to-peak voltage for both signals is set to 200 mV, this voltage is used for all signals.



(a) Sampled data - time domain

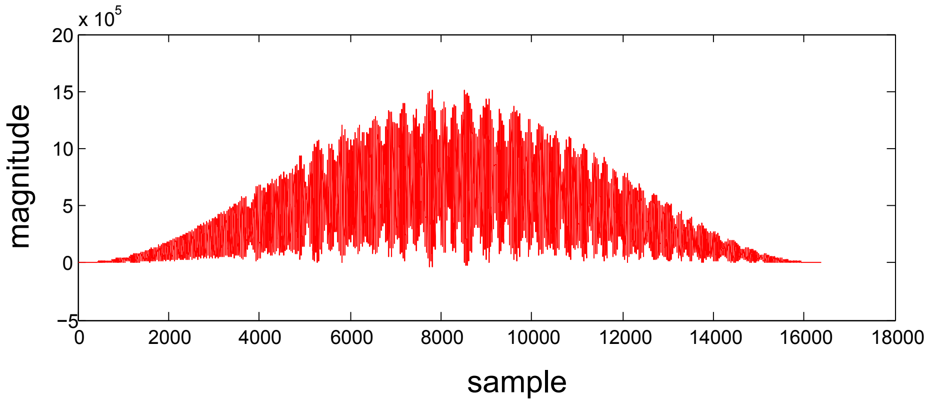


(b) Sampled data - frequency domain

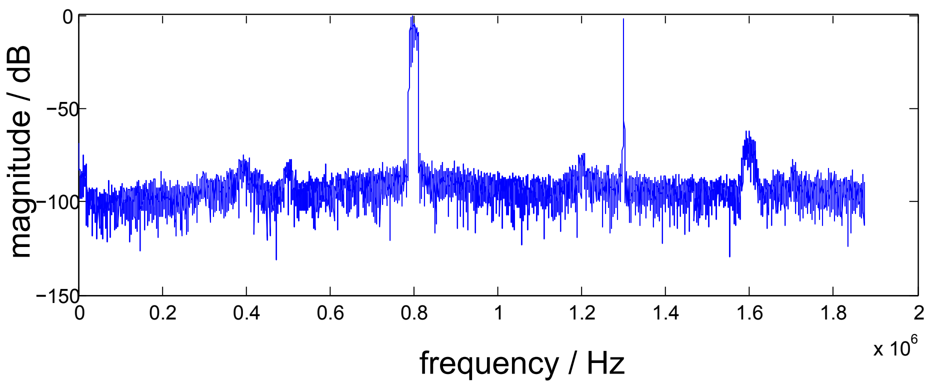
Figure 5.1: Sampled values in the time- and frequency-domain for a TETRA signal at 800kHz and a sine blocker at 1300kHz

The data plot shown consists of 16384 samples corresponding to 128 kB sampled data gained by continuously sampling for approximately 4.3 ms. The driving clock at the ADC was set to 30 MHz resulting in a Nyquist frequency of 1.875 MHz. The system allows to process up to 32 MB of data, which corresponds to a sample time of approximately 1 second. This amount of data is extremely heavy to analyse, hence 128kB of data was used as it proved to be an appropriate size.

A window function of type Hanning is applied to expose the noise-floor and to avoid artefacts such as peak smearing. This function was applied during the analysis in MATLAB and is not part of the implementation. The result is shown in figure 5.2, which shows a much clearer picture of the sampled values. At 1.6 MHz the first harmonic of the incoming TETRA signal is visible.



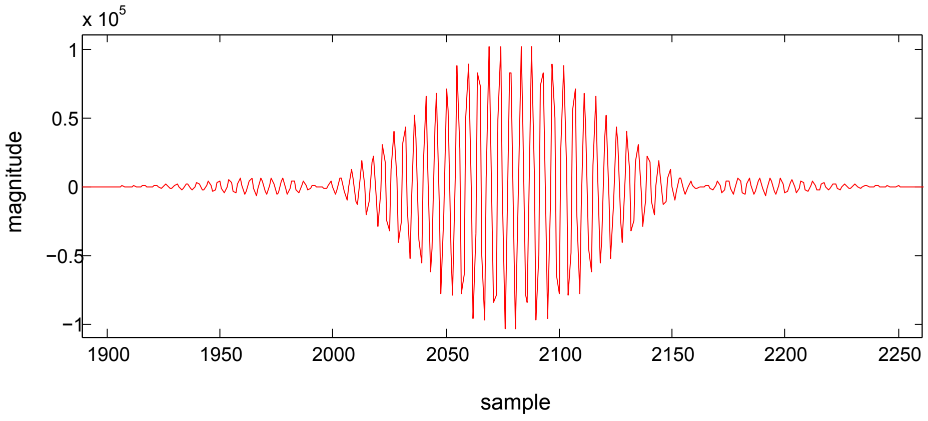
(a) Sampled data - time domain



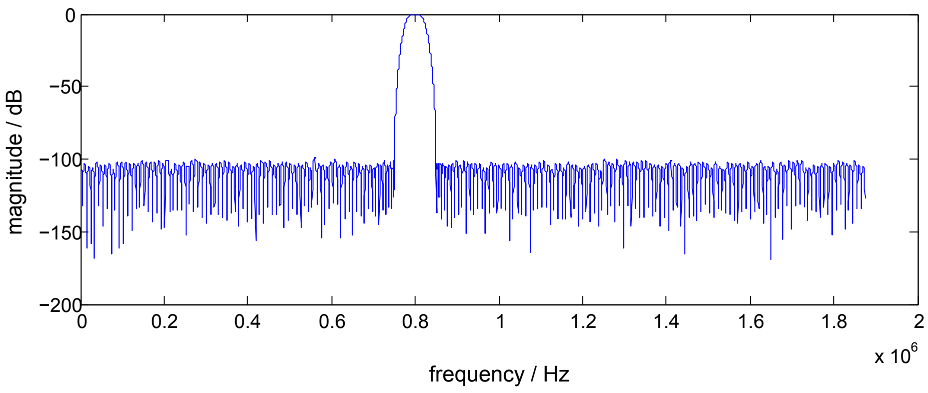
(b) Sampled data - frequency domain

Figure 5.2: Sampled values in time- and frequency-domain for a TETRA signal at 800kHz and a sine blocker at 1300kHz with a Hanning window applied.

The next configuration loaded to the FPGA contained a filter designed for a TETRA signal at 800 kHz. The data fed to the filter were not the incoming samples from the ADC, but a step function generated on-board. The result is the filter's impulse response as shown in figure 5.3.



(a) Impulse response - time domain



(b) Impulse response - frequency domain

Figure 5.3: Filter for a TETRA signal at 800kHz, order = 383

The impulse response shows that the stop-band settles at around -100dB and not at -105dB. As discussed, this is due to the quantization effects and the slightly lower order than the minimum order required.

The next step is to apply this filter on incoming data. The expected result is the data plot from figure 5.2 with the blocker suppressed by -100dB. Figure 5.4 shows the expected result, a Hanning window has been applied.

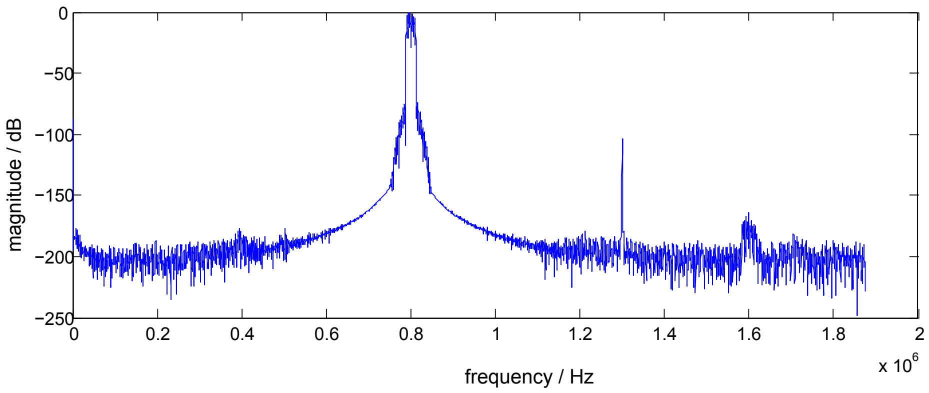


Figure 5.4: Filtered TETRA signal with blocker at 1300 kHz

As can be seen, the blocker has been suppressed down to -100dB as has the noise-floor, except for the part in the transition from the pass-band to the stop-band. The reason for the TETRA signal looking slightly different in this plot compared to the first plot is that the samples are taken at two different time instances. Figure 5.5 shows the incoming signal, filtered signal and blocker requirements in one plot, zoomed close to the channel and blocker.

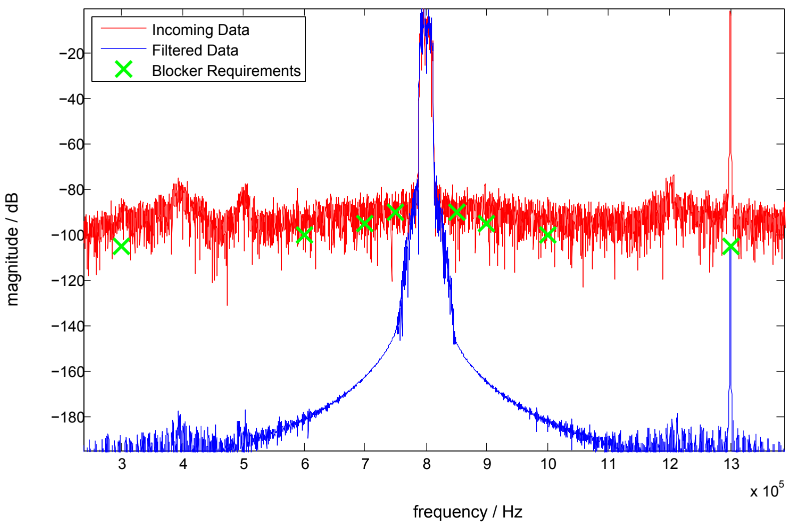
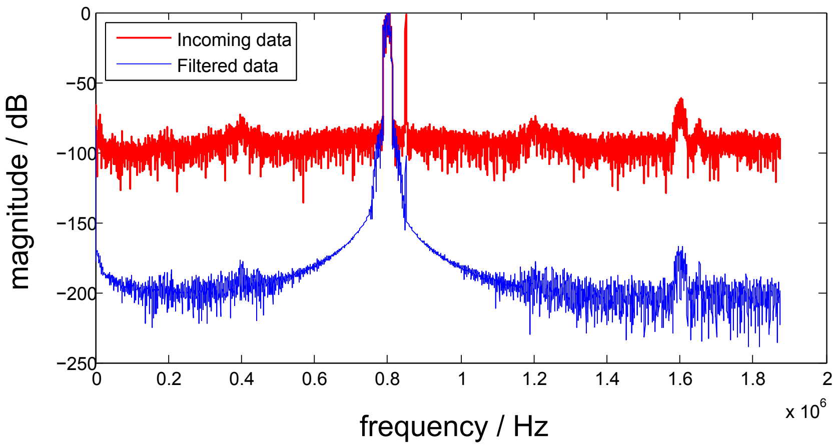


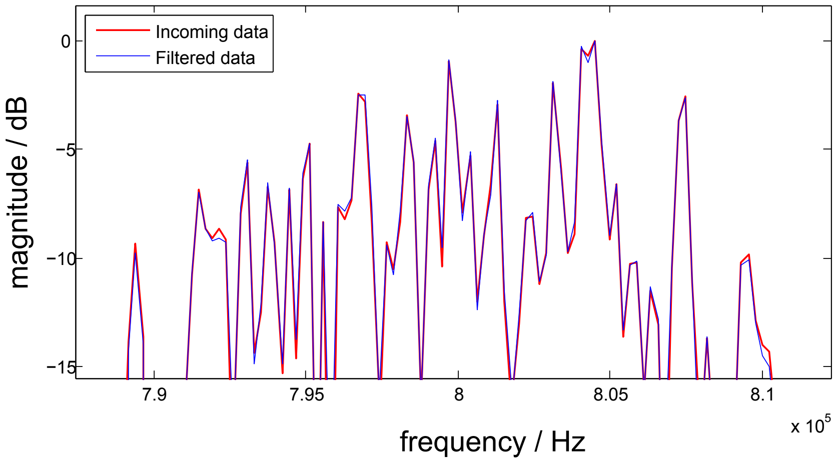
Figure 5.5: Combined plot showing the incoming signal, filtered signal, and blocker requirements

It seems like the blocker has been suppressed down to the required level, but zooming in closer would reveal that this, as expected, is not the case, since this would require a filter of a slightly bigger filter order.

As mentioned, the incoming data and the filtered data are sampled at two different time instances. To visualize the pass-band's impact on the incoming signal the filter's architecture has been simulated in MATLAB making it possible to compare unfiltered and filtered data of the same data set. This is shown in figure 5.6.



(a) Full view



(b) Close-up

Figure 5.6: Filter influence in the passband

The first plot shows the full view. In the passband the unfiltered data (red) is hidden behind the filtered data (blue) indicating that the signal has not been modified significantly. The requirements state that a ripple of 1 dB in the passband is acceptable, the filter applied has been designed with respect to this requirement. The second plot shows a magnification of the passband. Small differences are visible but they are within requirements and thereby confirming correct behaviour of the filter.

The next two plots show the result of moving the blocker closer to the TETRA channel. Figure 5.7 shows the blocker at 900 kHZ and figure 5.8 at 850 kHz.

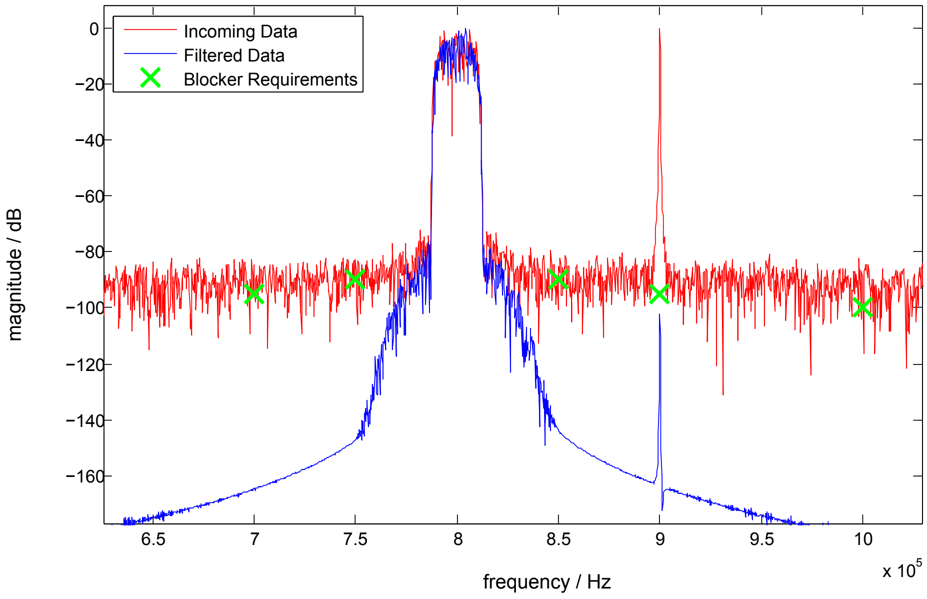


Figure 5.7: TETRA signal at 800kHz and blocker at 900kHz

As before, the measurements are taken at different time instances. The plots, show that the tight blocker requirements close to the channel are, due to the nature of the FIR filter, easily met. The gap between the blocker's peak and the blocker requirement can be exploited to improve the filter, thereby reducing the minimum order slightly.

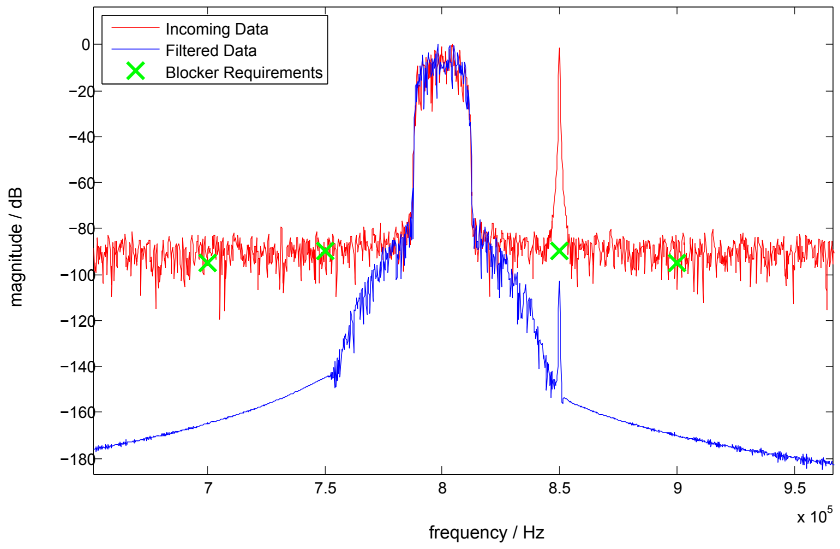


Figure 5.8: TETRA signal at 800kHz and blocker at 850kHz

In a TETRA network it may happen that two channels are located directly next to each other. It is therefore relevant to know how the filter affects a neighbour channel. This is shown in figure 5.9. The filter cuts through the neighbour channel, letting one side almost untouched and suppressing the other side up to -30 dB. The center frequency is suppressed by -10 dB.

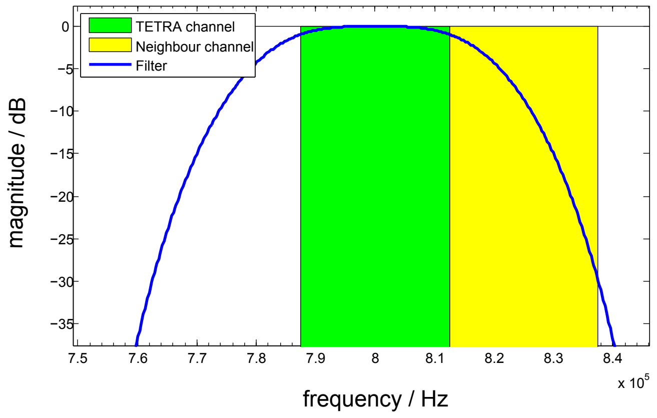
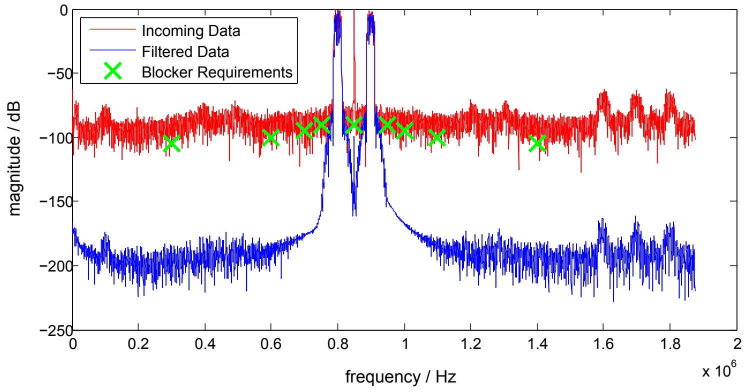


Figure 5.9: Two TETRA channels placed next to each other. The wanted channel's filter affects a neighbour channel

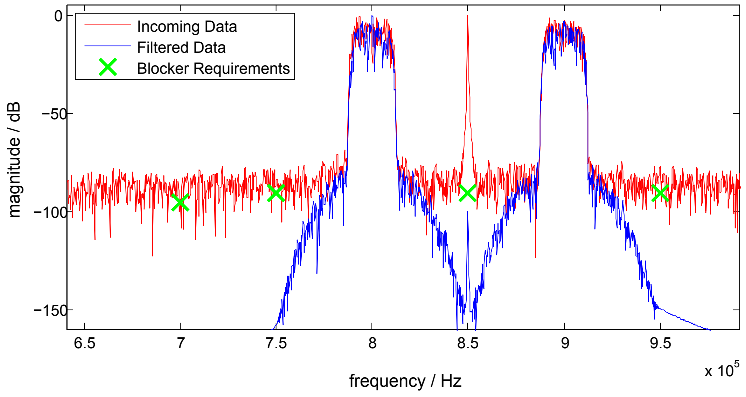
5.2 Dual Carrier

When using two filters, the output of the individual filters has been added, which makes it easy to analyse the data. But this also means, that frequencies suppressed equally by both filters will appear to be twice as strong (compared to a single filter system) which corresponds to +3 dB. In a final system, the filter outputs will be transmitted individually and not added. Therefore, when comparing the filtered blockers to the requirements, 3 dB should be subtracted. Plots that point out this behaviour will be presented.

The plot in figure 5.10 shows a system with two TETRA carriers and a blocker in between them.



(a) Dual TETRA system - full view

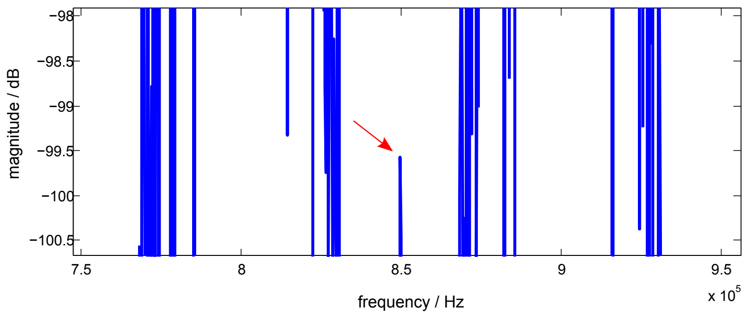


(b) Dual TETRA system - close-up

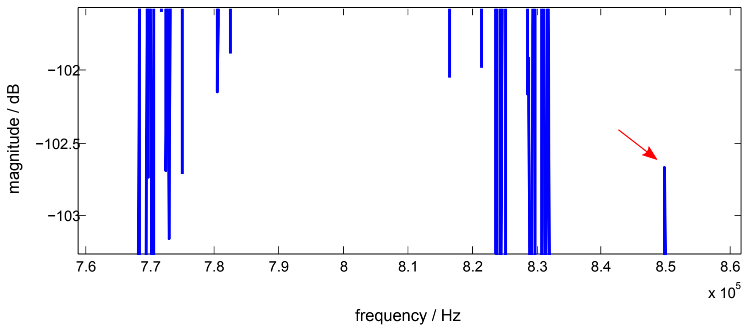
Figure 5.10: Results gained with two TETRA channels and a blocker placed at 850kHz

Clearly, a dual carrier system works just as well as a single carrier.

As mentioned, adding the filter outputs can influence the measured output. Magnifying the blocker's peak for the single and dual carrier system reveals this issue. This is shown in figure 5.11. As expected, the blocker level is about 3 dB stronger for the dual carrier system (-99.5 dB versus -102.5 dB).



(a) As indicated by the arrow, the blocker is located at 850 kHz. To the left and the right are the two TETRA channels.



(b) Blocker located at 850kHz with a TETRA channel to the left

Figure 5.11: Blocker peaks for single and dual carrier TETRA system

Care must be taken when implementing a dual carrier system based on two different channel types. Figure 5.12 shows the impulse response of a system based on a TETRA channel located at 800 kHz and a TEDS U50 channel located at 1 MHz.

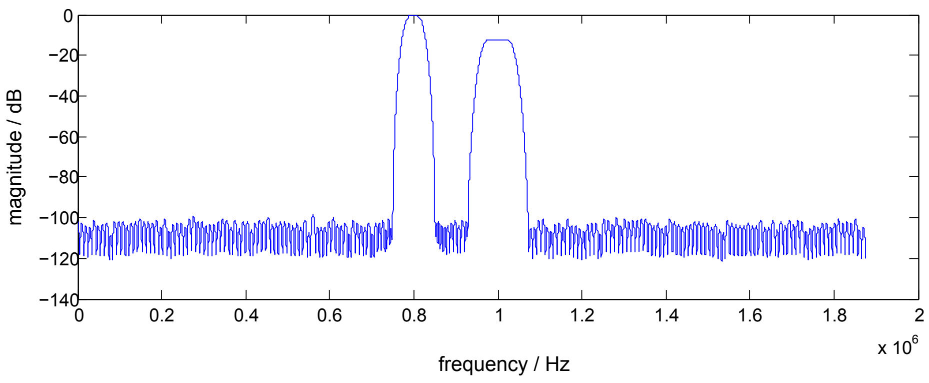


Figure 5.12: Impulse response for dualcarrier system with a TETRA channel at 800kHz and a U50 channel at 1MHz

Clearly their magnitudes have peaks of different strengths. This behaviour was discussed in section 3.3.4 and grounds in the position of the radix point of the filter coefficients. This can be fixed by shifting the output of one filter accordingly before adding the filter outputs. The result of doing so is shown in figure 5.13.

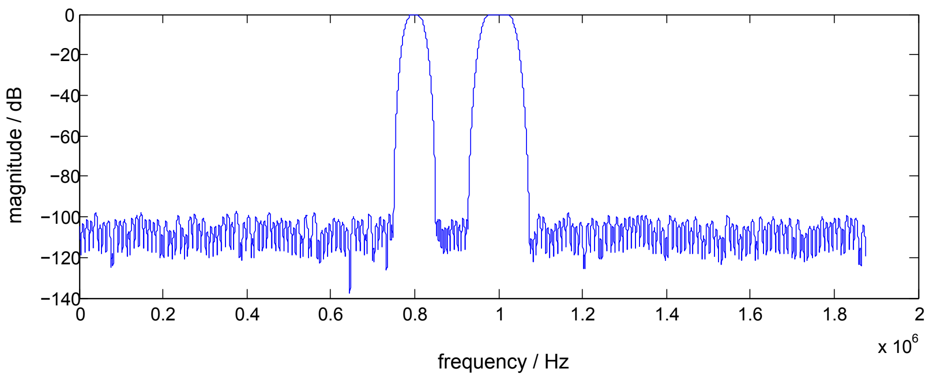


Figure 5.13: Impulse response with matched radix points

Applying this dual filter on incoming data with a blocker placed in between is shown in figure 5.15:

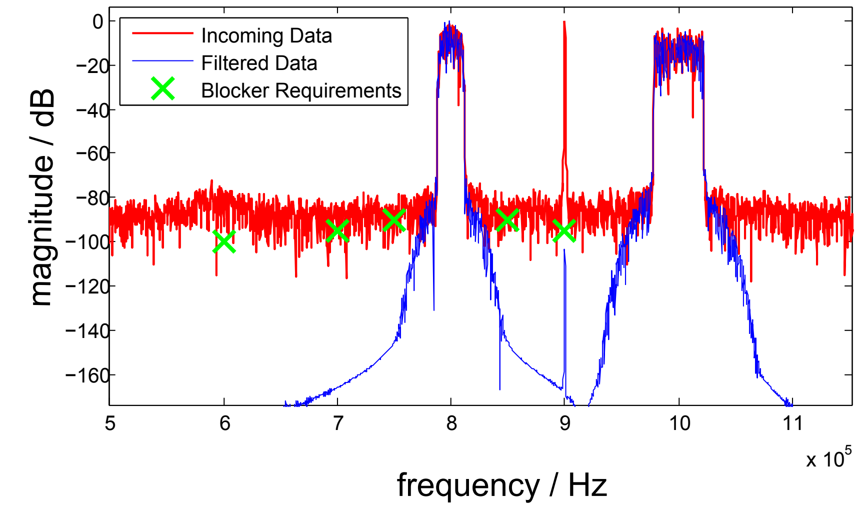


Figure 5.14: Dual carrier sustem for TETRA and a TEDS-U50 channel located at 800 kHz and 1 MHz, respectively. A blocker is located a 900 kHz.

What happens when the distance between the channels is increased is shown in figure 5.14.

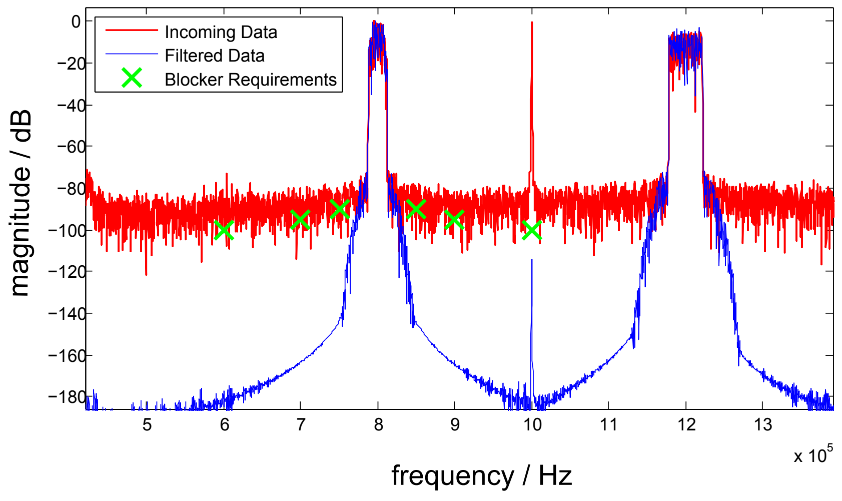


Figure 5.15: Dual carrier sustem for TETRA and a TEDS-U50 channel located at 800 kHz and 1.2 MHz, respectively. A blocker is located a 1 MHz.

Yet again the system behaves as requested and the blocker requirements have been met. In figure 5.14 the blocker is almost suppressed down to -120dB. This can be led back to the heavy ripples in the stop-band as depicted in figure 5.13. Apparently the blocker is located at a low point in the stop band resulting in additional suppression.

The final plot in figure 5.16 shows the impulse response for a TEDS U100 and U150 filter system. Since no masks were available to create input signals for these specifications, no data samples are available. The filter response shows that quantization effects are a little stronger for these wide-channel filters.

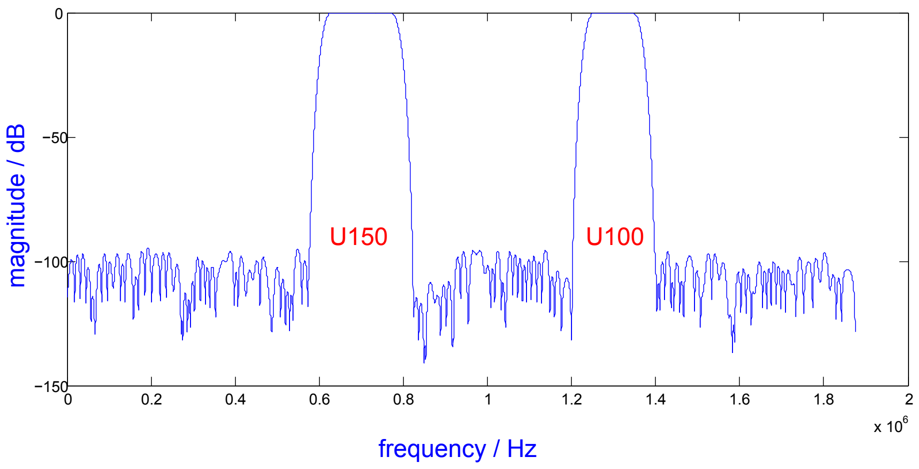
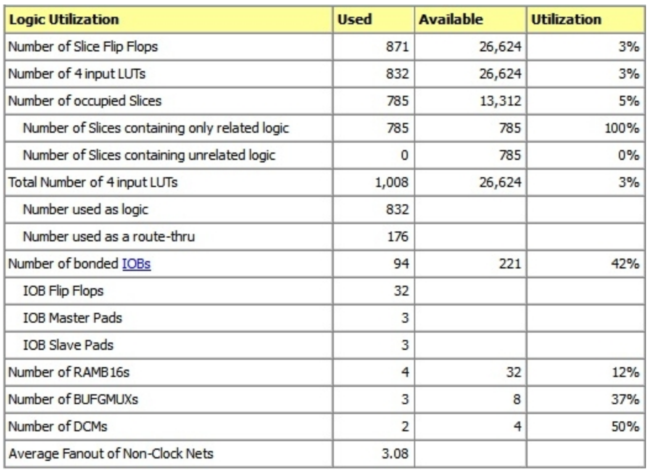


Figure 5.16: U150 and U100 impulse response

5.3 FPGA Utilization

This section contains screendumps showing the utilization of the FPGA in Xilinx for different designs.

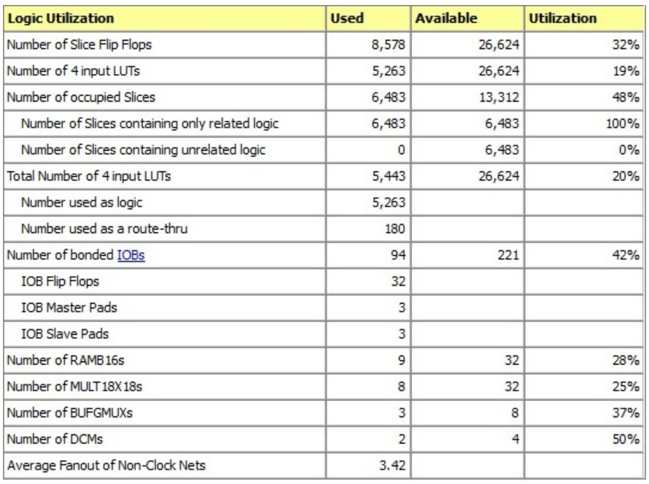
The picture in figure 5.17 shows the utilization of a design with no filters implemented. This design was used to sample unprocessed data from the ADC. The required resources are used for the interface to the ADC and the USB port.



Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	871	26,624	3%
Number of 4 input LUTs	832	26,624	3%
Number of occupied Slices	785	13,312	5%
Number of Slices containing only related logic	785	785	100%
Number of Slices containing unrelated logic	0	785	0%
Total Number of 4 input LUTs	1,008	26,624	3%
Number used as logic	832		
Number used as a route-thru	176		
Number of bonded IOBs	94	221	42%
IOB Flip Flops	32		
IOB Master Pads	3		
IOB Slave Pads	3		
Number of RAMB16s	4	32	12%
Number of BUFGMUXs	3	8	37%
Number of DCMs	2	4	50%
Average Fanout of Non-Clock Nets	3.08		

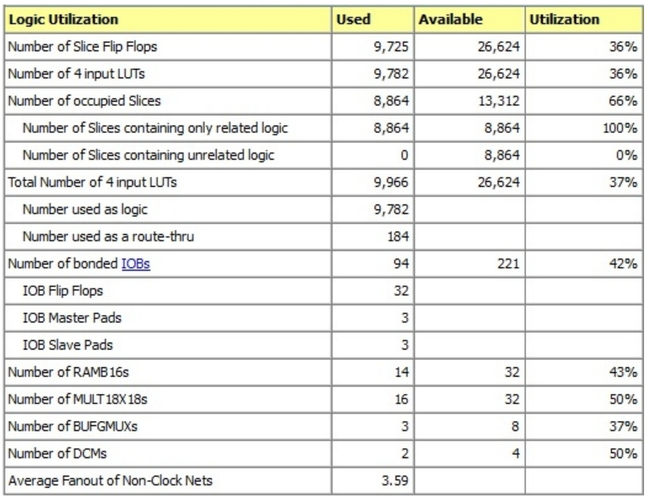
Figure 5.17: Utilization for a no-filter system (interfaces)

Figure 5.18 shows the utilization of a system with a single filter while figure 5.19 is obtained for a system with two filters.



Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	8,578	26,624	32%
Number of 4 input LUTs	5,263	26,624	19%
Number of occupied Slices	6,483	13,312	48%
Number of Slices containing only related logic	6,483	6,483	100%
Number of Slices containing unrelated logic	0	6,483	0%
Total Number of 4 input LUTs	5,443	26,624	20%
Number used as logic	5,263		
Number used as a route-thru	180		
Number of bonded IOBs	94	221	42%
IOB Flip Flops	32		
IOB Master Pads	3		
IOB Slave Pads	3		
Number of RAMB16s	9	32	28%
Number of MULT18X18s	8	32	25%
Number of BUFGMUXs	3	8	37%
Number of DCMs	2	4	50%
Average Fanout of Non-Clock Nets	3.42		

Figure 5.18: Utilization for a single-filter system (order=383)



Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	9,725	26,624	36%
Number of 4 input LUTs	9,782	26,624	36%
Number of occupied Slices	8,864	13,312	66%
Number of Slices containing only related logic	8,864	8,864	100%
Number of Slices containing unrelated logic	0	8,864	0%
Total Number of 4 input LUTs	9,966	26,624	37%
Number used as logic	9,782		
Number used as a route-thru	184		
Number of bonded IOBs	94	221	42%
IOB Flip Flops	32		
IOB Master Pads	3		
IOB Slave Pads	3		
Number of RAMB16s	14	32	43%
Number of MULT18X18s	16	32	50%
Number of BUFGMUXs	3	8	37%
Number of DCMs	2	4	50%
Average Fanout of Non-Clock Nets	3.59		

Figure 5.19: Utilization for a dual-filter system (orders=383)

5.4 Latency

The latency of the filter architecture corresponds to one data cycle of the ADC. Running the ADC with a clock frequency of 30.24 MHz results in a data rate of $\frac{3\times30.24}{24} = 3.78$ MSPS. Hence, the latency comes to $\frac{1}{3.78M}s = 265ns$. A few clocks are added due to the pipelined addition which increases the latency to somewhere between $300ns$ and $350ns$.

Chapter 6

Future Work

In order to take this project to the next level a more suitable FPGA has to be selected. The following contains useful observations and necessary changes for doing so.

The dynamic range in the output signal of the ADC used in this project does not have the required dynamic range for a final design neither does the implemented hardware model. An increase in the dynamic range of the incoming data requires bigger multipliers, hence a bigger FPGA. A dynamic range of 120 dB requires a resolution of 20 bits. Since the data in the delay pipeline is added before entering the multipliers (only true for a symmetric structures as used in this project) at least 21 bit multipliers are necessary. Bigger multipliers would also deal with the quantization effects and a safe choice would be 24x24 bits multipliers. This would allow a dynamic range of up to 138 dB in the input signal and reduce the quantization effects to a minimum. Furthermore, a bigger FPGA would allow the implementation of more than two filters.
Since 24x24 bit multipliers are hard to find in an FPGA an alternative solution is to combine up to 4 18x18 bit multipliers which would result in a 36x36 bit multiplier. However, this requires a lot of multipliers. The total number of multipliers, based on the 24 iterations, required for a suitable TETRA system filter, is 9. Hence, 36 multipliers would be needed for such a filter when combining the multipliers.

The figures in the *FPGA utilization* section of chapter 5 can be used to determine a proper sized FPGA. The critical numbers seem to be the number of slices, the amount of block memory (RAMB16s) and of course the multipliers. An issue regarding resource allocation between the block memory and the multipliers was discovered for the FPGA used in this project. This should be kept in mind when selecting a bigger FPGA.

In order to use the suggested architecture in a design where the filter coefficients are delivered by an external source the block memory holding the coefficients needs to be modified. In this project read-only memory (ROM) with hardcoded filter coefficients was used. This has to be replaced by rewritable memory blocks (RAM). Furthermore, a control unit needs to be added to manage the sweep of the filter coefficients.

IQ demodulation was not within the scope of this project. If IQ modulation is going to be added to the design, the FPGA should have additional resources, accordingly.

Last but not least, the FPGA should have sufficient IO resources in order to transmit multiple filter outputs separately. This, of course, depends on the number of filters and the desired dynamic range of the output values.

Chapter 7

Conclusion

The purpose of this project was to implement digital filtering in an FPGA. This filtering is part of the receiver in a base station which handles wireless TETRA and TEDS communication. In order to design a suitable hardware model, an analysis on filter types based on predefined requirements is performed in chapter 2. This chapter concludes that band-pass FIR filters are the right choice due to their stability and linearity. Moreover, it is concluded that the filter structure should be of a symmetric type since this reduces the amount of multiplications which is a critical resource in an FPGA. The filter order must be kept as low as possible in order to make the best use of the multipliers. A set of filter design algorithms were analyzed and the Equiripple method gave the best results in terms of the order number, which was found to be 417.

An important note is that the filters in this receiver model are not going to be constant, it must be possible to change them. Based on this, an architecture was developed in chapter 3 which allows the implementation of symmetric FIR filters of different sizes. The filter order can vary from 7 to 575 in predefined steps and several filters can be implemented at the same time depending on the resources available on the FPGA. This is achieved by a resizeable, partly parallel, and partly serial architecture. The serial part can easily be modified in order to support even bigger filters. In order to let the filter coefficients be replaceable they are stored in block memory. In this prototype design the block memory has no write access, hence, the coefficients are not changeable. Exchanging this read-only block memory with writeable memory will grant changeable filters.

The filter architecture is embedded in a system capable of communicating with an ADC[1] and receive data from it. Furthermore, data processed on the FPGA is stored in the on-board SDRAM. The contents of this memory can be accessed through an interface designed for this purpose. This interface consists of a module implemented in the FPGA and a script executed on a PC. The board and the PC are communicating through a USB connection.

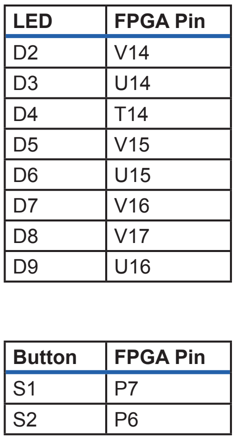
Correct implementation of the architecture and the interfaces has been verified and documented in chapter 4.

Finally, a couple of single and dual carrier filter systems were created which were applied with TETRA/TEDS signals and sine blockers. The results are documented in chapter 5. The filters used were of order 383 which is slightly lower than the minimum order necessary to meet requirements. The results obtained are indeed satisfying except for filter coefficient quantization effects. These effects can be minimized by increasing the width of the filter coefficients' vectors as mentioned in chapter 6. However, this would have exceeded the resources available on the FGPA made available for the scope of the project.

[1]not any ADC but the ADC used in this project

Appendix A

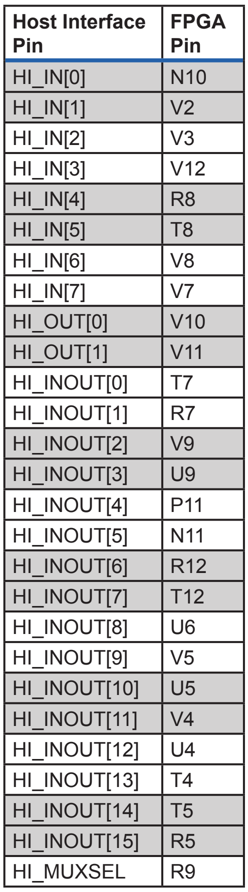
Additional Tables And Figures



LED	FPGA Pin
D2	V14
D3	U14
D4	T14
D5	V15
D6	U15
D7	V16
D8	V17
D9	U16

Button	FPGA Pin
S1	P7
S2	P6

Figure A.1: XEM3010 LED and button pins



Host Interface Pin	FPGA Pin
HI_IN[0]	N10
HI_IN[1]	V2
HI_IN[2]	V3
HI_IN[3]	V12
HI_IN[4]	R8
HI_IN[5]	T8
HI_IN[6]	V8
HI_IN[7]	V7
HI_OUT[0]	V10
HI_OUT[1]	V11
HI_INOUT[0]	T7
HI_INOUT[1]	R7
HI_INOUT[2]	V9
HI_INOUT[3]	U9
HI_INOUT[4]	P11
HI_INOUT[5]	N11
HI_INOUT[6]	R12
HI_INOUT[7]	T12
HI_INOUT[8]	U6
HI_INOUT[9]	V5
HI_INOUT[10]	U5
HI_INOUT[11]	V4
HI_INOUT[12]	U4
HI_INOUT[13]	T4
HI_INOUT[14]	T5
HI_INOUT[15]	R5
HI_MUXSEL	R9

Figure A.2: XEM3010 80 pin connector

JP2 Pin	Connection	FPGA Pin	LVDS	Length (mm)	JP2 Pin	Connection	FPGA Pin	LVDS	Length (mm)
1	DGND				2	+3.3VDD			
3	+2.5VDD				4	+3.3VDD			
5	JTAG_TCK				6	+3.3VDD			
7	JTAG_TMS				8	X_TDO			
9	X_TDI				10	-			
11	SYS_CLK4				12	-			
13	DGND				14	DGND			
15	XBUS-1	T16 (*)	L01P_3	17.196	16	XBUS-0	U18	L16P_3	17.089
17	XBUS-3	T17 (*)	L01N_3	12.588	18	XBUS-2	T18	L16N_3	17.172
19	XBUS-5	R16	L17P_3	13.817	20	XBUS-4	R17	L19N_3	18.255
21	XBUS-7	P15	L21N_3	14.607	22	XBUS-6	R18	L19P_3	16.923
23	XBUS-9	P16	L17N_3	13.276	24	XBUS-8	P17	L20P_3	18.006
25	XBUS-11	N15	L21P_3	14.359	26	XBUS-10	P18	L20N_3	16.675
27	XBUS-13	M15	L23N_3	14.442	28	XBUS-12	N17	L24P_3	17.757
29	XBUS-15	M16	L23P_3	13.110	30	XBUS-14	M18	L24N_3	17.426
31	XBUS-17	L15	L34N_3	16.068	32	XBUS-16	L17	L35P_3	18.450
33	XBUS-19	L16	L34P_3	14.736	34	XBUS-18	L18	L35N_3	16.885
35	+VCCO3				36	DGND			
37	XBUS-21	N14 (†)	L22P_3	13.950	38	XBUS-20	K17	L40N_3	17.509
39	XBUS-23	M14 (†)	L22N_3	14.282	40	XBUS-22	K18	L40P_3	16.509
41	XBUS-25	K13 (†)	L39N_3	14.951	42	XBUS-24	L14 (†)	L27N_3	19.266
43	XBUS-27	K14 (†)	L39P_3	13.619	44	XBUS-26	L13 (†)	L27P_3	19.935
45	XBUS-29	K15	-	12.288	46	XBUS-28	J13 (†)	-	21.426
47	XBUS-31	J14 (†)	L40P_2	14.316	48	XBUS-30	H13 (†)	L27N_2	22.119
49	XBUS-33	J15	L40N_2	12.868	50	XBUS-32	H14 (†)	L27P_2	20.539
51	XBUS-35	G14 (†)	L22P_2	16.068	52	XBUS-34	J18	L39N_2	15.680
53	XBUS-37	F14 (†)	L22N_2	15.736	54	XBUS-36	J17	L39P_2	16.929
55	+VCCO2				56	DGND			
57	XBUS-39	H16	L34N_2	10.791	58	XBUS-38	H18	L35P_2	15.515
59	XBUS-41	H15	L34P_2	12.163	60	XBUS-40	H17	L35N_2	16.680
61	XBUS-43	G16	L24P_2	11.205	62	XBUS-42	G18	L23N_2	15.515
63	XBUS-45	G15	L24N_2	12.205	64	XBUS-44	F17	L23P_2	15.686
65	XBUS-47	F15	L21N_2	12.039	66	XBUS-46	E18	L20P_2	14.355
67	XBUS-49	E16	L19P_2	10.956	68	XBUS-48	E17	L20N_2	15.686
69	XBUS-51	E15	L21P_2	12.288	70	XBUS-50	D18	L17P_2	15.018
71	XBUS-53	D16	L19N_2	11.205	72	XBUS-52	D17	L17N_2	16.101
73	XBUS-55	C17 (*)	L01P_2	10.785	74	XBUS-54	C18	L16P_2	14.852
75	XBUS-57	C16 (*)	L01N_2	11.453	76	XBUS-56	B18	L16N_2	16.601
77	XCLK1	F10	L32P_1	21.992	78	DGND			
79	XCLK2	E10	L32N_1	21.756	80	DGND			

Notes: * - Pin is a DCI pin with optionally-installed resistors.
† - Some routing on inner layer is not necessarily 50Ω.

Figure A.3: OkHost interface pins

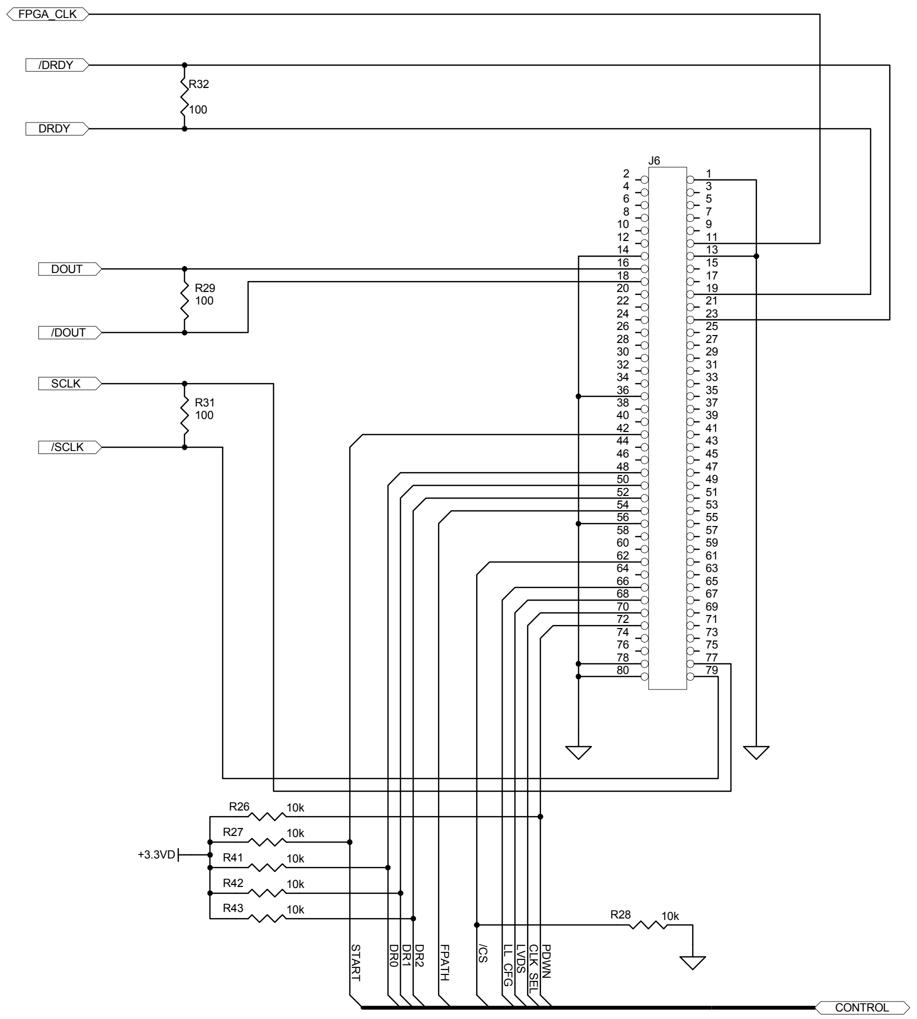
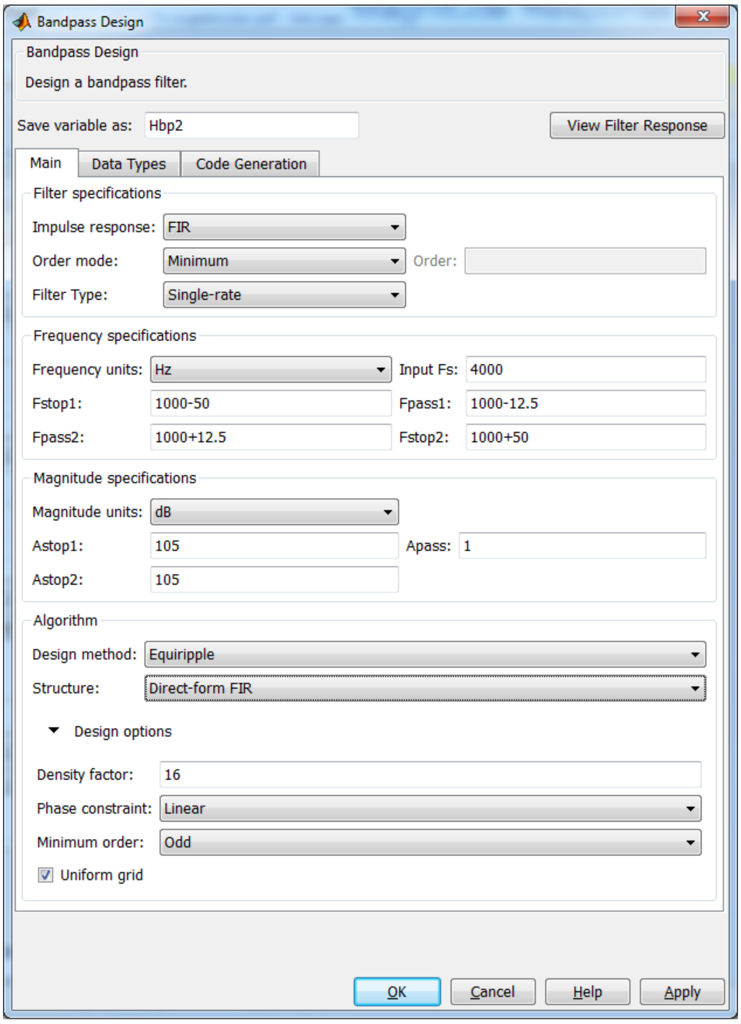


Figure A.4: Physical connection of ADC - FPGA



Bandpass Design

Bandpass Design

Design a bandpass filter.

Save variable as: Hbp2

Filter specifications

Impulse response: FIR

Order mode: Minimum

Order:

Filter Type: Single-rate

Frequency specifications

Frequency units: Hz

Input Fs: 4000

Fstop1: 1000-50

Fpass1: 1000-12.5

Fpass2: 1000+12.5

Fstop2: 1000+50

Magnitude specifications

Magnitude units: dB

Astop1: 105

Apass: 1

Astop2: 105

Algorithm

Design method: Equiripple

Structure: Direct-form FIR

Design options

Density factor: 16

Phase constraint: Linear

Minimum order: Odd

☑ Uniform grid

Figure A.5: filterbuilder screenshot of Equiripple design for TETRA

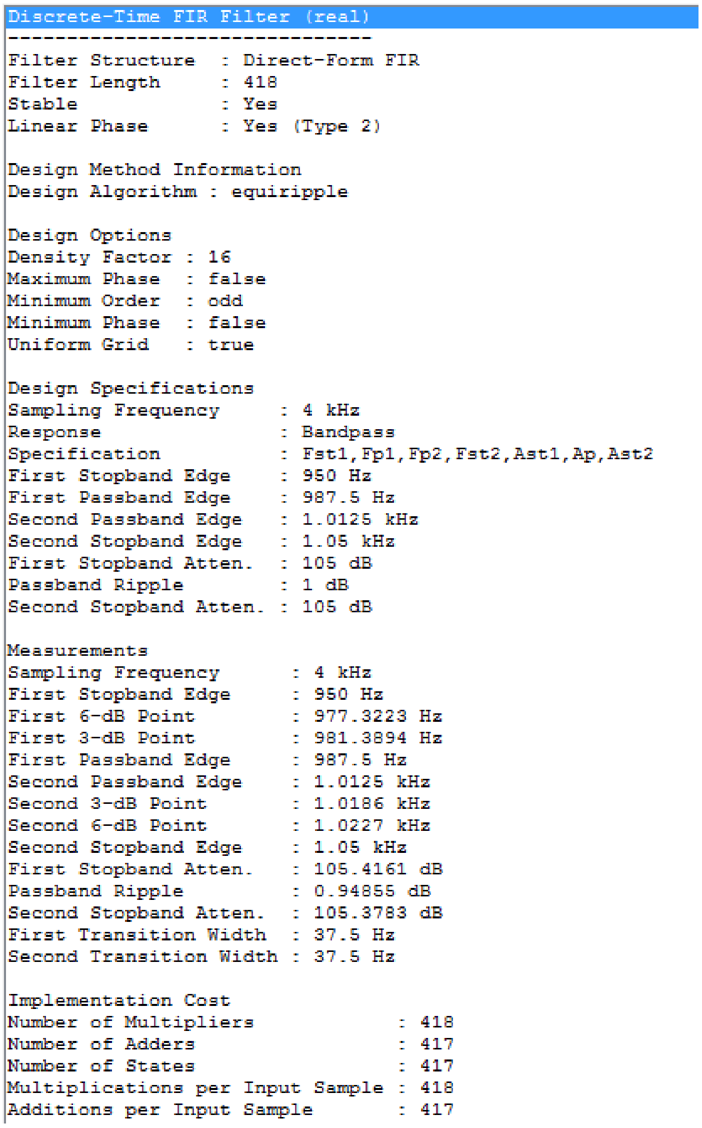
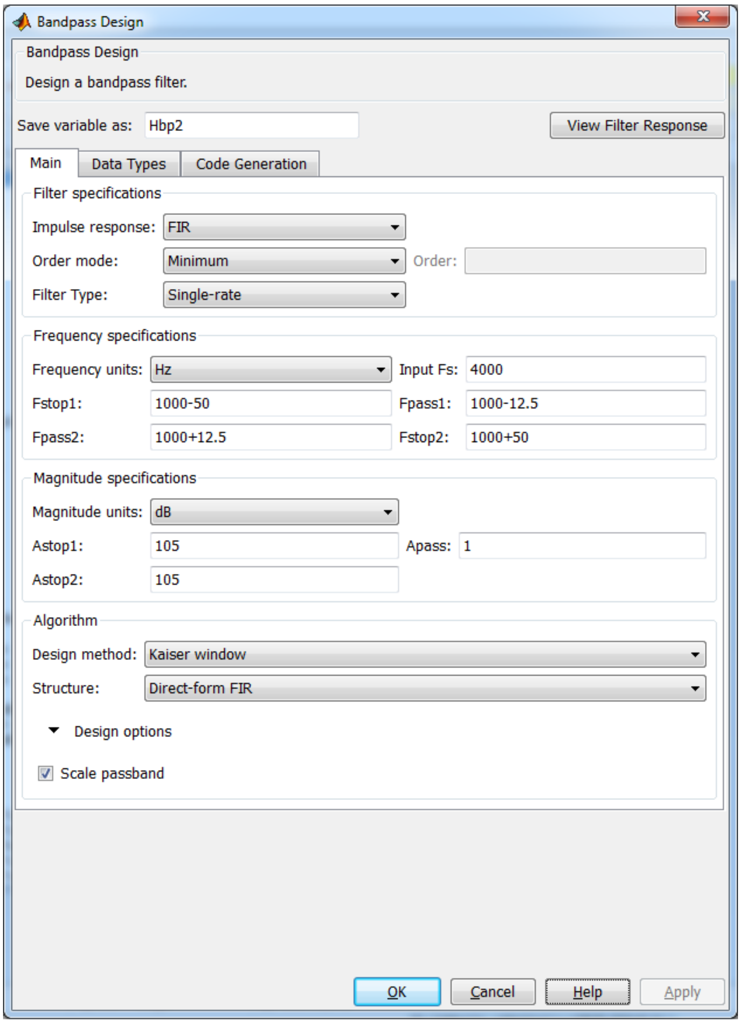

```
Discrete-Time FIR Filter (real)
-------------------------------
Filter Structure  : Direct-Form FIR
Filter Length     : 418
Stable            : Yes
Linear Phase      : Yes (Type 2)

Design Method Information
Design Algorithm : equiripple

Design Options
Density Factor : 16
Maximum Phase  : false
Minimum Order  : odd
Minimum Phase  : false
Uniform Grid   : true

Design Specifications
Sampling Frequency      : 4 kHz
Response                : Bandpass
Specification           : Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2
First Stopband Edge     : 950 Hz
First Passband Edge     : 987.5 Hz
Second Passband Edge    : 1.0125 kHz
Second Stopband Edge    : 1.05 kHz
First Stopband Atten.   : 105 dB
Passband Ripple         : 1 dB
Second Stopband Atten.  : 105 dB

Measurements
Sampling Frequency      : 4 kHz
First Stopband Edge     : 950 Hz
First 6-dB Point        : 977.3223 Hz
First 3-dB Point        : 981.3894 Hz
First Passband Edge     : 987.5 Hz
Second Passband Edge    : 1.0125 kHz
Second 3-dB Point       : 1.0186 kHz
Second 6-dB Point       : 1.0227 kHz
Second Stopband Edge    : 1.05 kHz
First Stopband Atten.   : 105.4161 dB
Passband Ripple         : 0.94855 dB
Second Stopband Atten.  : 105.3783 dB
First Transition Width  : 37.5 Hz
Second Transition Width : 37.5 Hz

Implementation Cost
Number of Multipliers            : 418
Number of Adders                 : 417
Number of States                 : 417
Multiplications per Input Sample : 418
Additions per Input Sample       : 417
```

Figure A.6: info screenshot of Equiripple design for TETRA



Bandpass Design

Bandpass Design

Design a bandpass filter.

Save variable as: Hbp2

Filter specifications

Impulse response: FIR

Order mode: Minimum

Order:

Filter Type: Single-rate

Frequency specifications

Frequency units: Hz

Input Fs: 4000

Fstop1: 1000-50

Fpass1: 1000-12.5

Fpass2: 1000+12.5

Fstop2: 1000+50

Magnitude specifications

Magnitude units: dB

Astop1: 105

Apass: 1

Astop2: 105

Algorithm

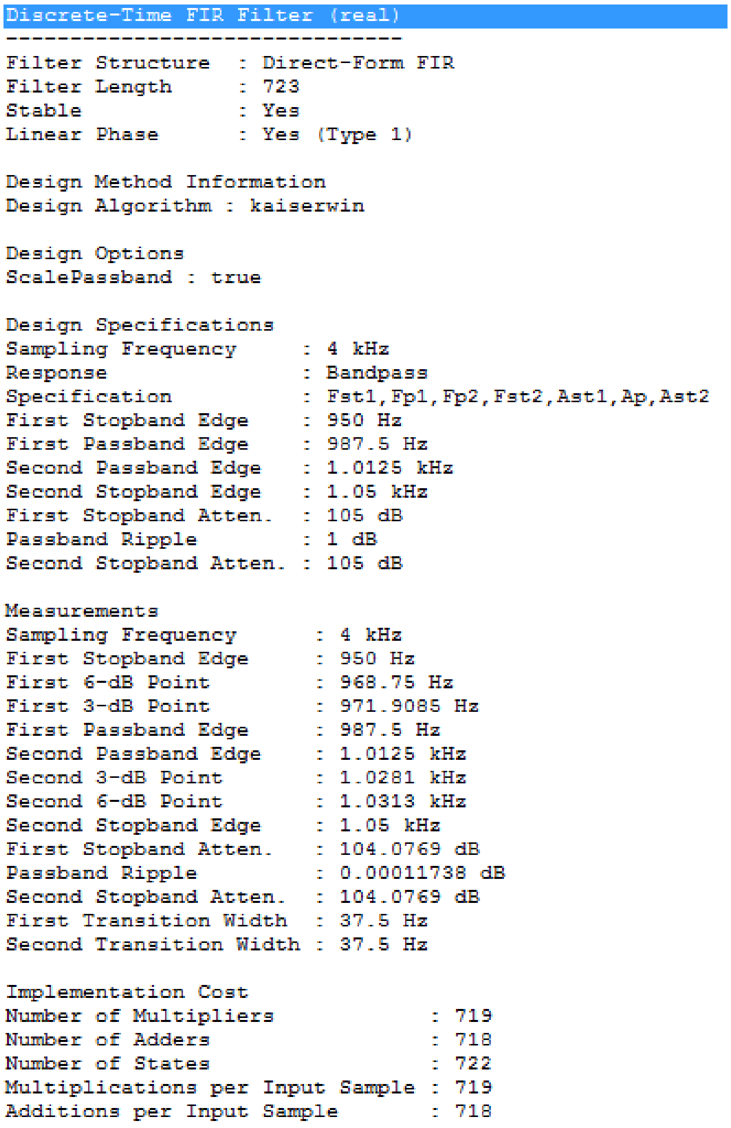
Design method: Kaiser window

Structure: Direct-form FIR

Design options

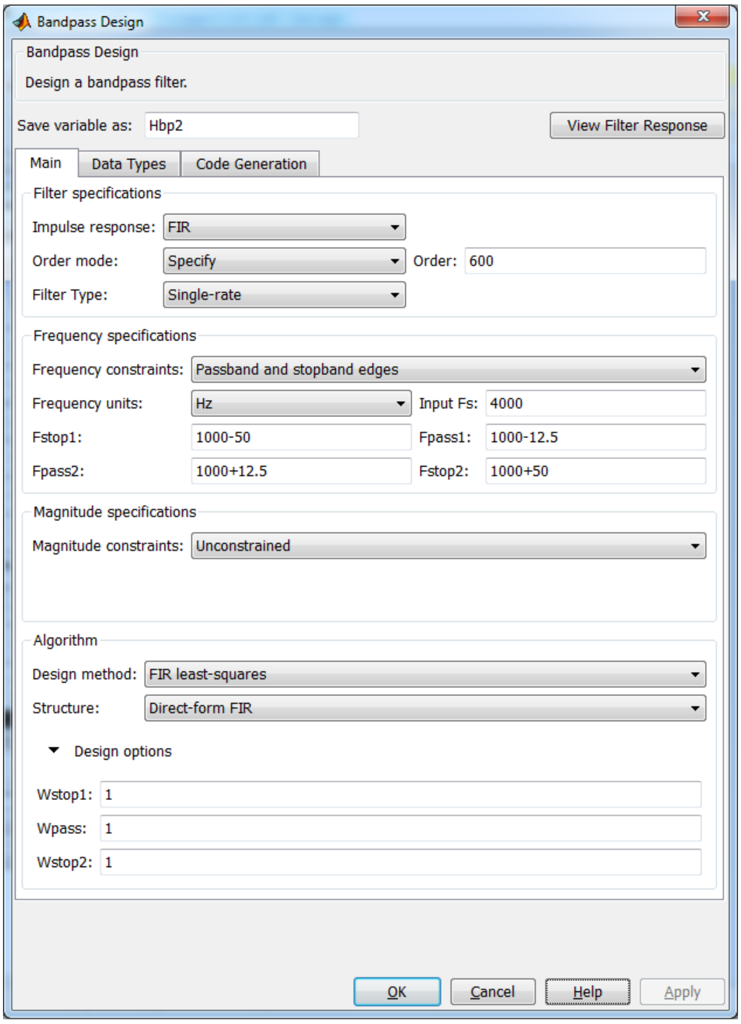
[✓] Scale passband

Figure A.7: filterbuilder screenshot of Kaiser-window design for TETRA



```
Discrete-Time FIR Filter (real)
-------------------------------
Filter Structure  : Direct-Form FIR
Filter Length     : 723
Stable            : Yes
Linear Phase      : Yes (Type 1)

Design Method Information
Design Algorithm : kaiserwin

Design Options
ScalePassband : true

Design Specifications
Sampling Frequency     : 4 kHz
Response               : Bandpass
Specification          : Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2
First Stopband Edge    : 950 Hz
First Passband Edge    : 987.5 Hz
Second Passband Edge   : 1.0125 kHz
Second Stopband Edge   : 1.05 kHz
First Stopband Atten.  : 105 dB
Passband Ripple        : 1 dB
Second Stopband Atten. : 105 dB

Measurements
Sampling Frequency      : 4 kHz
First Stopband Edge     : 950 Hz
First 6-dB Point        : 968.75 Hz
First 3-dB Point        : 971.9085 Hz
First Passband Edge     : 987.5 Hz
Second Passband Edge    : 1.0125 kHz
Second 3-dB Point       : 1.0281 kHz
Second 6-dB Point       : 1.0313 kHz
Second Stopband Edge    : 1.05 kHz
First Stopband Atten.   : 104.0769 dB
Passband Ripple         : 0.00011738 dB
Second Stopband Atten.  : 104.0769 dB
First Transition Width  : 37.5 Hz
Second Transition Width : 37.5 Hz

Implementation Cost
Number of Multipliers            : 719
Number of Adders                 : 718
Number of States                 : 722
Multiplications per Input Sample : 719
Additions per Input Sample       : 718
```

Figure A.8: info screenshot of Kaiser-window design for TETRA



Bandpass Design

Bandpass Design

Design a bandpass filter.

Save variable as: Hbp2

Filter specifications

Impulse response: FIR

Order mode: Specify Order: 600

Filter Type: Single-rate

Frequency specifications

Frequency constraints: Passband and stopband edges

Frequency units: Hz Input Fs: 4000

Fstop1: 1000-50 Fpass1: 1000-12.5

Fpass2: 1000+12.5 Fstop2: 1000+50

Magnitude specifications

Magnitude constraints: Unconstrained

Algorithm

Design method: FIR least-squares

Structure: Direct-form FIR

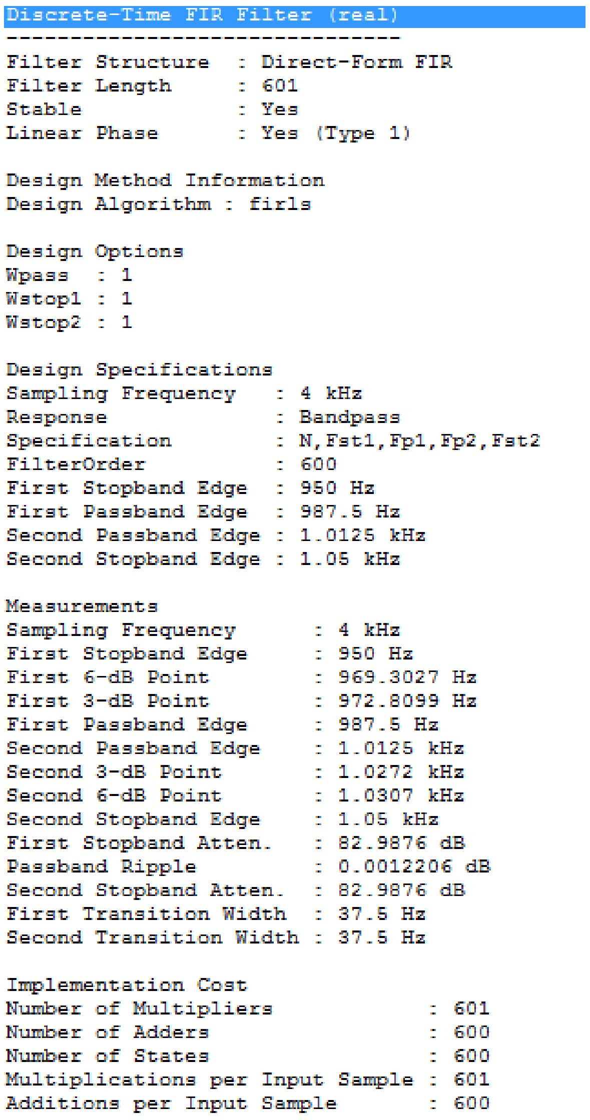
Design options

Wstop1: 1

Wpass: 1

Wstop2: 1

Figure A.9: filterbuilder screenshot of least-squares design for TETRA



```
Discrete-Time FIR Filter (real)
-------------------------------
Filter Structure  : Direct-Form FIR
Filter Length     : 601
Stable            : Yes
Linear Phase      : Yes (Type 1)

Design Method Information
Design Algorithm : firls

Design Options
Wpass  : 1
Wstop1 : 1
Wstop2 : 1

Design Specifications
Sampling Frequency    : 4 kHz
Response              : Bandpass
Specification         : N,Fst1,Fp1,Fp2,Fst2
FilterOrder           : 600
First Stopband Edge   : 950 Hz
First Passband Edge   : 987.5 Hz
Second Passband Edge  : 1.0125 kHz
Second Stopband Edge  : 1.05 kHz

Measurements
Sampling Frequency      : 4 kHz
First Stopband Edge     : 950 Hz
First 6-dB Point        : 969.3027 Hz
First 3-dB Point        : 972.8099 Hz
First Passband Edge     : 987.5 Hz
Second Passband Edge    : 1.0125 kHz
Second 3-dB Point       : 1.0272 kHz
Second 6-dB Point       : 1.0307 kHz
Second Stopband Edge    : 1.05 kHz
First Stopband Atten.   : 82.9876 dB
Passband Ripple         : 0.0012206 dB
Second Stopband Atten.  : 82.9876 dB
First Transition Width  : 37.5 Hz
Second Transition Width : 37.5 Hz

Implementation Cost
Number of Multipliers            : 601
Number of Adders                 : 600
Number of States                 : 600
Multiplications per Input Sample : 601
Additions per Input Sample       : 600
```

Figure A.10: info screenshot of least-squares design for TETRA

Appendix B

Source Code

B.1 UCF file

The following shows the contents of the XEM3010.ucf file which holds pin declarations and timing constraints.

```
#-------------------------------------------------------------------
# FrontPanel Host Interface pins
#-------------------------------------------------------------------
NET "hi_in<0>"     LOC = "N10";
NET "hi_in<1>"     LOC = "V2";
NET "hi_in<2>"     LOC = "V3";
NET "hi_in<3>"     LOC = "V12";
NET "hi_in<4>"     LOC = "R8";
NET "hi_in<5>"     LOC = "T8";
NET "hi_in<6>"     LOC = "V8";
NET "hi_in<7>"     LOC = "V7";

NET "hi_out<0>"  LOC = "V10";
NET "hi_out<1>"  LOC = "V11";

NET "hi_inout<0>"  LOC = "T7";
NET "hi_inout<1>"  LOC = "R7";
NET "hi_inout<2>"  LOC = "V9";
NET "hi_inout<3>"  LOC = "U9";
NET "hi_inout<4>"  LOC = "P11";
NET "hi_inout<5>"  LOC = "N11";
NET "hi_inout<6>"  LOC = "R12";
NET "hi_inout<7>"  LOC = "T12";
NET "hi_inout<8>" LOC = "U6";
NET "hi_inout<9>" LOC = "V5";
NET "hi_inout<10>"  LOC = "U5";
NET "hi_inout<11>"  LOC = "V4";
NET "hi_inout<12>"  LOC = "U4";
NET "hi_inout<13>"  LOC = "T4";
NET "hi_inout<14>"  LOC = "T5";
NET "hi_inout<15>"  LOC = "R5";

NET "hi_muxsel"       LOC = "R9";
NET "i2c_sda"      LOC = "R13"  |PULLUP;
NET "i2c_scl"      LOC = "U13"  |PULLUP;

#NET "jtag_tck"   LOC = "P14"
#NET "jtag_tms"   LOC = "R14"
#NET "jtag_tdi"   LOC = "R10"
#NET "jtag_tdo"   LOC = "P12"
#----------------
# PLL Clock pins
#----------------
NET "clk1"     LOC = "N9"; # SDRAM
#NET "clk2"    LOC = "P9";
#NET "clk3"    LOC = "P10";


#-------------
# SDRAM
#-------------
#The min setup (TSU) of the SDRAM-8 is 2ns, plus 500ps of board delay
#we need to add this OFFSET to all outputs to SDRAM
#
NET sdram_addr[*] OFFSET = OUT : 2.5 : BEFORE : clk1 ;
NET sdram_data[*] OFFSET = OUT : 2.5 : BEFORE : clk1 ;
NET sdram_ras_n OFFSET = OUT : 2.5 : BEFORE : clk1 ;
NET sdram_cas_n OFFSET = OUT : 2.5 : BEFORE : clk1 ;
NET sdram_cs_n OFFSET = OUT : 2.5 : BEFORE : clk1 ;
NET sdram_we_n OFFSET = OUT : 2.5 : BEFORE : clk1 ;
NET sdram_bank[*] OFFSET = OUT : 2.5 : BEFORE : clk1 ;
#
#The max clock-to-out (Tac) of the SDRAM-8 is 6ns, plus 300ps of board↩
     delay
#we need to add this OFFSET to all inputs from SDRAM
NET sdram_data[*] OFFSET = IN : 6.5 : VALID : 0.8 : AFTER : Clk1 ; #↩
    6.3

#Set NODELAY mode for inputs from SDRAM.
#By default, the IBUF has a DELAY element to guarantee 0 hold time
#By turning off the DELAY element, we save ~500ps in IBUF delay
NET sdram_data[*] NODELAY  | SLEW = "FAST" ;
NET sdram_addr[*] SLEW = "FAST" | IOSTANDARD = SSTL2_I;
NET sdram_bank[*] SLEW = "FAST" | IOSTANDARD = SSTL2_I;

NET "sdram_cke"    LOC = "F8"  | SLEW = "FAST" | IOSTANDARD = SSTL2_I;
NET "sdram_cas_n"    LOC = "E11" | SLEW = "FAST" | IOSTANDARD = SSTL2_I↩
    ;
NET "sdram_ras_n"    LOC = "D12" | SLEW = "FAST" | IOSTANDARD = SSTL2_I↩
    ;
NET "sdram_we_n"     LOC = "E7"  | SLEW = "FAST" | IOSTANDARD = SSTL2_I↩
    ;
NET "sdram_cs_n"     LOC = "E8"  | SLEW = "FAST" | IOSTANDARD = SSTL2_I↩
    ;
NET "sdram_ldqm"     LOC = "D9"  | SLEW = "FAST" | IOSTANDARD = SSTL2_I↩
    ;
NET "sdram_udqm"     LOC = "A9"  | SLEW = "FAST" | IOSTANDARD = SSTL2_I↩
    ;
```

```

NET "sdram_addr<0>"    LOC = "A15";
NET "sdram_addr<1>"    LOC = "A16";
NET "sdram_addr<2>"    LOC = "B15";
NET "sdram_addr<3>"    LOC = "B14";
NET "sdram_addr<4>"    LOC = "D11";
NET "sdram_addr<5>"    LOC = "B13";
NET "sdram_addr<6>"    LOC = "C11";
NET "sdram_addr<7>"    LOC = "A12";
NET "sdram_addr<8>"    LOC = "A11";
NET "sdram_addr<9>"    LOC = "D10";
NET "sdram_addr<10>"     LOC = "A17";
NET "sdram_addr<11>"     LOC = "B10";
NET "sdram_addr<12>"     LOC = "A10";

NET "sdram_bank<0>"    LOC = "C12";
NET "sdram_bank<1>"    LOC = "A14";

NET "sdram_data<0>"    LOC = "C4";
NET "sdram_data<1>"    LOC = "D5";
NET "sdram_data<2>"    LOC = "C5";
NET "sdram_data<3>"    LOC = "D6";
NET "sdram_data<4>"    LOC = "D7";
NET "sdram_data<5>"    LOC = "C7";
NET "sdram_data<6>"    LOC = "C8";
NET "sdram_data<7>"    LOC = "D8";
NET "sdram_data<8>"    LOC = "B9";
NET "sdram_data<9>"    LOC = "A8";
NET "sdram_data<10>"     LOC = "A7";
NET "sdram_data<11>"     LOC = "B6";
NET "sdram_data<12>"     LOC = "A5";
NET "sdram_data<13>"     LOC = "B5";
NET "sdram_data<14>"     LOC = "A4";
NET "sdram_data<15>"     LOC = "B4";


#-------------
# LEDS
#-------------
NET "led<0>"       LOC = "V14";
NET "led<1>"       LOC = "U14";
NET "led<2>"       LOC = "T14";
NET "led<3>"       LOC = "V15";
NET "led<4>"       LOC = "U15";
NET "led<5>"       LOC = "V16";
NET "led<6>"       LOC = "V17";
NET "led<7>"       LOC = "U16";

#-------------
# Buttons
#-------------
NET "BTN_right"  LOC = "P7";
NET "BTN_left"     LOC = "P6";

#-------------
# Control
#-------------
NET "START"      LOC = "L14";   # PIN 42 - START
NET "DRATE<0>"     LOC = "H13";   # PIN 48 - DR0
NET "DRATE<1>"     LOC = "H14";   # PIN 50 - DR1
NET "DRATE<2>"     LOC = "J18";   # PIN 52 - DR2
NET "FPATH"      LOC = "J17";   # PIN 54 - FPATH
NET "CS_inv"     LOC = "G18";   # PIN 62 - /CS
NET "LL_CONFIG"  LOC = "E18";   # PIN 66 - LL CFG
NET "LVDS"       LOC = "E17";   # PIN 68 - LVDS
NET "SCLK_SEL"     LOC = "D18";  # PIN 70 - CLK SEL
NET "PDWN"       LOC = "D17";  # PIN 72 - PDWN
#--------------
# DATA
#--------------
#NET "FPGA_CLK"  SYS_CLK4 (CKLD); # PIN 11 - FPGA_CLK
NET "DRDY_P"  LOC = "R16" |IOSTANDARD = LVDS_25 ; # PIN 19 - DRDY
NET "DRDY_N"  LOC = "P16" |IOSTANDARD = LVDS_25 ; # PIN 23 - /DRDY
NET "SCLK_P"  LOC = "F10";  # PIN 77 - SCLK
NET "SCLK_N"  LOC = "E10";  # PIN 79 - /SCLK
NET "DOUT_P"  LOC = "U18" |IOSTANDARD = LVDS_25 ; # PIN 16 - DOUT
NET "DOUT_N"  LOC = "T18" |IOSTANDARD = LVDS_25 ; # PIN 18 - /DOUT

NET "clk1" TNM_NET = clk1;
TIMESPEC TS_clk1 = PERIOD "clk1" 10 ns HIGH 50%; #100 MHz Memory clock

NET "hi_in<0>" TNM_NET = hi_in<0>;
TIMESPEC TS_hi_in_0_ = PERIOD "hi_in<0>" 20 ns HIGH 50%; #48MHz USB ↩
    clock

NET "SCLK_P" TNM_NET = SCLK_P;
TIMESPEC TS_SCLK_P = PERIOD "SCLK_P" 10.8 ns HIGH 50%; # 30,24*4 MHz ↩
    ADC clock (32MHz)
```

B.2 Main.vhd

This is the top file of the VHDL implementation, all interfaces are part of this code.

```vhdl

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

Library UNISIM; -- Xilinx primitives
use UNISIM.vcomponents.all;

use FRONTPANEL.all;

entity Main is
    Port ( clk1    : in  STD_LOGIC; -- SDRAM clk
        BTN_left : in  STD_LOGIC;
        BTN_right : in STD_LOGIC;
        led    : out STD_LOGIC_VECTOR(7 downto 0);

        -- USB Host
        hi_muxsel, i2c_scl, i2c_sda : out STD_LOGIC;
        hi_in     : in    std_logic_vector(7 downto 0); --Host interface input signals.
        hi_out    : out   std_logic_vector(1 downto 0); --Host interface output signals.
        hi_inout  : inout std_logic_vector(15 downto 0);--Host interface bidirectional signals.

        -- SDRAM
        sdram_data   : inout std_logic_vector(15 downto 0);
        sdram_cke    : out std_logic;                    -- clock enable
        sdram_bank   : out std_logic_vector(1 downto 0);   -- bank selection
        sdram_addr   : out std_logic_vector(12 downto 0);  -- address

        sdram_cs_n   : out std_logic;
        sdram_ras_n  : out std_logic;
        sdram_cas_n  : out std_logic;
        sdram_we_n   : out std_logic;

        sdram_ldqm   : out std_logic;
        sdram_udqm   : out std_logic;


         -- ADC
        START    : out STD_LOGIC;    -- pulse=single-sample, constant=continously-sampling
        PDWN    : out STD_LOGIC;    -- power down, active low [inverted??]
        SCLK_SEL : out STD_LOGIC; -- (CLK SEL) shift-clock source select: 0=internally, 1=externally
        LVDS    : out STD_LOGIC;    -- LVDS(0) / CMOS(1) compatible voltage level [inverted??]
        LL_CONFIG : out STD_LOGIC;  -- low latency digital filter: 0=single-cycle, 1=fast-response
        FPATH : out STD_LOGIC;     -- digital filter path selection: 0=wide-bandwidth, 1=low-latency
        DRATE : out STD_LOGIC_VECTOR(2 downto 0); -- data rate selection
        CS_inv : out STD_LOGIC;    -- chip select, active low
        SCLK_N : in STD_LOGIC;     -- negative shift clock
        SCLK_P  : inout STD_LOGIC;  -- positive shift clock, 0=input, ↩
            1=output
        DOUT_N : in STD_LOGIC;     -- negative LVDS serial data
        DOUT_P  : in STD_LOGIC;    -- positive LVDS serial data
        DRDY_N : in STD_LOGIC;     -- negative data ready
        DRDY_P  : in STD_LOGIC     -- positive data ready

        );
end Main;



architecture Behavioral of Main is

  COMPONENT sdramctrl
    PORT(
      clk : IN std_logic;
      clk_read : IN std_logic;
      reset : IN std_logic;
      cmd_pagewrite : IN std_logic;
      cmd_pageread : IN std_logic;
      rowaddr_in : IN std_logic_vector(14 downto 0);
      fifo_din : IN std_logic_vector(15 downto 0);
      sdram_d : INOUT std_logic_vector(15 downto 0);
      cmd_ack : OUT std_logic;
      cmd_done : OUT std_logic;
      fifo_dout : OUT std_logic_vector(15 downto 0);
      fifo_write : OUT std_logic;
      fifo_read : OUT std_logic;
      sdram_cmd : OUT std_logic_vector(3 downto 0);
      sdram_ba : OUT std_logic_vector(1 downto 0);
      sdram_a : OUT std_logic_vector(12 downto 0)
    );
  END COMPONENT;

    COMPONENT dcm_sys
    PORT (
      CLKIN_IN : IN std_logic;
      RST_IN : IN std_logic;
      CLKIN_IBUFG_OUT : OUT std_logic;
      CLK0_OUT : OUT std_logic;
      LOCKED_OUT : OUT std_logic
    );
  END COMPONENT;

    COMPONENT fifo_gen
    PORT(
      rst : in std_logic;
      wr_clk : IN std_logic;
      rd_clk : IN std_logic;
      din : IN std_logic_vector(15 downto 0);
      wr_en : IN std_logic;
      rd_en : IN std_logic;
      dout  : OUT std_logic_vector(15 downto 0);
      full : OUT std_logic;
      empty  : OUT std_logic;
      rd_data_count : OUT std_logic_vector(10 downto 0);
      wr_data_count : OUT std_logic_vector(10 downto 0)
    );
  END COMPONENT;


  COMPONENT fifo_64_16
```

```
    PORT(
      rst : in std_logic;
      wr_clk : IN std_logic;
      rd_clk : IN std_logic;
      din : IN std_logic_vector(63 downto 0);
      wr_en : IN std_logic;
      rd_en : IN std_logic;
      dout  : OUT std_logic_vector(15 downto 0);
      full : OUT std_logic;
      empty  : OUT std_logic;
      rd_data_count : OUT std_logic_vector(10 downto 0);
      wr_data_count : OUT std_logic_vector(8 downto 0)
    );
  END COMPONENT;

  signal reset: std_logic;

  -- clock dividers
  constant C : integer := 50000000; -- number used for clock division
  signal counter1, counter2, counter3 : integer range 0 to C;
  signal leds1, leds2, leds3 : std_logic;

  -- OkHost
  constant W : integer :=3; --number of output wires connected usb ↩
      host interface (HI)
  signal hi_data: std_logic_vector(23 downto 0);
  signal ti_clk: std_logic; -- Output Buffered copy of the host ↩
      interface clock (48 MHz).  This signal does not need to be ↩
      connected to the target end-points because it is replicated ↩
      within OK1.
  signal ok1 : std_logic_vector(30 downto 0); --(HI to target) Control↩
        signals to the target endpoints.
  signal ok2, ok2_21, ok2_20, ok2_pipe_out : std_logic_vector(16 ↩
      downto 0); --(Target to HI) Input Control signals from the ↩
      target endpoints.
  signal ok2s : std_logic_vector(W*17-1 downto 0);
  signal triggers : std_logic_vector(15 downto 0);
  signal pipe_out_data  : std_logic_vector(15 downto 0);
  signal pipe_out_rdy : std_logic;

  -- SDRAM
  signal rd_wr_switch : std_logic;
  signal sdram_clk : std_logic;
  signal sdram_cmd : std_logic_vector(3 downto 0); --cs_n, ras_n, ↩
      cas_n, we_n
  signal cmd_pagewrite : std_logic;
  signal cmd_pageread : std_logic;
  signal cmd_ack : std_logic;
  signal cmd_done : std_logic;
  signal sdram_rowaddr_read, sdram_rowaddr_write,sdram_rowaddr_in : ↩
      unsigned( 14 downto 0);
  signal sdram_reset : std_logic;

  signal ram_rd_en, ram_wr_en, sdram_rd_en, sdram_wr_en : std_logic;
  type ram_states is (idle, r_ack_wait, w_ack_wait, busy);
  signal ram_state : ram_states;
  signal ram_data : std_logic_vector(23 downto 0);
  signal cont_counter : unsigned(15 downto 0);
  signal ramcounter : unsigned(22 downto 0); --22 =32mB --14 = 128 kb
    -- (7 downto 0) = 1 page = 1kByte data

  type transfer_states is (state_A, state_B);
  signal transfer_state : transfer_states;


```

```vhdl
    signal init : std_logic;


  -- FIFOs
  signal  fifo_out_din :  std_logic_vector(15 downto 0);
  signal  fifo_out_full : std_logic;
  signal  fifo_out_empty : std_logic;
  signal  fifo_out_wr_en : std_logic;
  signal  fifo_out_rd_en : std_logic;
  signal  fifo_out_dout : std_logic_vector(15 downto 0);
  signal  fifo_out_status : std_logic_vector(10 downto 0);

  signal  fifo_in_dout : std_logic_vector( 15 downto 0);
  signal  fifo_in_wr_en : std_logic;
  signal  fifo_in_rd_en : std_logic;
  signal  fifo_in_empty : std_logic;
  signal  fifo_in_full : std_logic;
  signal  fifo_in_din : std_logic_vector(63 downto 0);
  signal  fifo_in_status : std_logic_vector(10 downto 0);

  signal fault_fifo_in_empty, fault_fifo_out_empty, fault_fifo_in_full↩
      , fault_fifo_out_full : std_logic;


  -- FILTER
  signal result_ready : std_logic;
  signal filter_out1, filter_out2: signed(63 downto 0);

  -- ADC
  signal data : std_logic_vector (23 downto 0);
  signal DOUT, DRDY, SCLK : std_logic;
  signal min, max, steady_data : signed(23 downto 0);
  signal smaller, bigger, new_steady_data : std_logic;

  type adc_states is (adc_init, adc_skip, adc_idle, adc_sample, ↩
      adc_data);
  signal adc_state, adc_next : adc_states;
  signal data_counter : integer range 0 to 23;
  signal steady_data_en, data_counter_en, data_counter_reset, ↩
      DOUT_buffer, data_counter_low : std_logic;

  -- unit step function
  signal step : signed(23 downto 0);
  signal step_counter : unsigned(12 downto 0);


begin


--*************************************************************
--                        OkHost
--*************************************************************
  okHI : entity okHost
    port map (hi_in => hi_in,
          hi_out => hi_out,
          hi_inout => hi_inout,
          ti_clk => ti_clk, --out
          ok1 => ok1, --out
          ok2 => ok2); --in

  ok2s <= ok2_pipe_out & ok2_21 & ok2_20;

  wireOR :   okWireOR
    generic map (N => 3)
    port map (ok2 => ok2,
             ok2s => ok2s);

   wire20 :  okWireOut
     port map (ok1 => ok1,
           ok2 => ok2_20,
           ep_addr => x"20",
           ep_datain => hi_data (15 downto 0)); -- 16 bit

   wire21 :  okWireOut
     port map (ok1 => ok1,
           ok2 => ok2_21,
           ep_addr => x"21",
           ep_datain => hi_data (23 downto 8)); -- 16 bit

   trigIn1 : okTriggerIn
     port map (ok1 => ok1,
           ep_addr => x"53",
           ep_clk => not sdram_clk,
           ep_trigger => triggers);

   pipeOutA3 : okPipeOut
     port map (ok1 => ok1,
           ok2 => ok2_pipe_out,
           ep_addr => x"a3",
           ep_datain => pipe_out_data,
           ep_read => pipe_out_rdy);


--*******************************************************************
--                     LVDS BUFFERS
--*******************************************************************
   IBUFDS_1 : IBUFDS
     port map (
       O => DOUT,
       I => DOUT_P,
       IB => DOUT_N);

   IBUFDS_2 : IBUFDS
     port map (
       O => DRDY,
       I => DRDY_P,
       IB => DRDY_N);

   IBUFDS_3 : IBUFDS
       port map (
         O => sclk,
         I => SCLK_P,
         IB => SCLK_N);


--*******************************************************************
--                     FILTER(s)
--*******************************************************************

   FILTERA : ENTITY work.myfilter2
     generic map (filterblock => 1)
       port map (
         clk => SCLK,
         reset => reset,
         new_data => new_steady_data,
         data => signed(steady_data(23 downto 7)), -- ADC data
--         data => step(16 downto 0), -- impulse response
--         data => '0' & x"0001", -- constant data
         result_ready => result_ready,
         result => filter_out1
```

```
        );

  FILTER1 : ENTITY work.myfilter2
    generic map (filterblock => 4, FRAC_SHIFT => 2)
      port map (
        clk => SCLK,
        reset => reset,
        new_data => new_steady_data,
        data =>  signed(steady_data(23 downto 7)), -- ADCdata
--          data => step(16 downto 0), -- impulse response
--          result_ready => result_ready,
        result => filter_out2
        );


--*********************************************************************
--                         STEP FUNCTION
--*********************************************************************

  step_response: process(sclk, reset) begin
    if (reset = '1') then
      step <= (others => '0');
      step_counter <= (others => '0');
    elsif (sclk'event and sclk = '1') then
      if(new_steady_data = '1') then
        step <= x"000000";
        if (step_counter(step_counter'left) = '0') then
          step_counter <= step_counter +1;
          if (step_counter(step_counter'left-1) = '1') then
            step_counter <= (others => '1');
            step <= x"000001";
          end if;
        end if;
      end if;
    end if;
  end process;


--*********************************************************************
--                         SDRAM / FIFOs
--*********************************************************************

  sdram_cke <= '1';
  sdram_cs_n  <= sdram_cmd(3);
  sdram_ras_n <= sdram_cmd(2);
  sdram_cas_n <= sdram_cmd(1);
  sdram_we_n  <= sdram_cmd(0);
  sdram_ldqm <= '0';
  sdram_udqm <= '0';



  SDRAM: sdramctrl
    PORT MAP(
          clk => not sdram_clk,
          clk_read => not sdram_clk,
          reset => sdram_reset,

          cmd_pagewrite => cmd_pagewrite,
          cmd_pageread => cmd_pageread,
          cmd_ack =>    cmd_ack,
          cmd_done => cmd_done,
          rowaddr_in => std_logic_vector(sdram_rowaddr_in),

          fifo_din => fifo_in_dout,
          fifo_dout => fifo_out_din,
          fifo_write => fifo_out_wr_en,
          fifo_read => fifo_in_rd_en,

          sdram_cmd => sdram_cmd, -- {cs_n, ras_n, cas_n, we_n}
          sdram_ba => sdram_bank,
          sdram_a => sdram_addr,
          sdram_d => sdram_data
        );

  sdramDCM : dcm_sys
    PORT MAP(
      CLKIN_IN => clk1,
      RST_IN => '0',
--      .CLKIN_IBUFG_OUT(),
      CLK0_OUT => sdram_clk
--      .LOCKED_OUT()
    );

  fifo_in: fifo_64_16
    PORT MAP(
      rst => reset,
      wr_clk => Sclk,
      rd_clk => not sdram_clk,
      din => fifo_in_din,
      wr_en => fifo_in_wr_en,
      rd_en => fifo_in_rd_en,
      dout => fifo_in_dout,
      full => fifo_in_full,
      empty => fifo_in_empty ,
      rd_data_count => fifo_in_status
    );

  fifo_out: fifo_gen --(16/16)
    PORT MAP(
      din => fifo_out_din,
      rd_clk => ti_clk,
      rd_en => fifo_out_rd_en,
      rst => reset,
      wr_clk => not sdram_clk,
      wr_en => fifo_out_wr_en,
      dout => fifo_out_dout,
      empty => fifo_out_empty ,
      full => fifo_out_full,
      wr_data_count => fifo_out_status
    );

--*******************************************************************
--                  I/O CONTROLS
--*******************************************************************

  hi_muxsel <= '0'; -- force active usb interfacing
  i2c_scl <= 'Z'; -- tied to high impedance
  i2c_sda <= 'Z'; -- tied to high impedance
  reset <= not(BTN_left) and not(BTN_right); -- BTNs are active low

  START <= not reset;
  PDWN <= '1';    -- allways power on, active low
  LVDS <= '0';    -- high speed requires LVDS (setting is ignored ↩
      under highspeed)
  SCLK_SEL <= '0';  -- must be generated internally under highspeed, ↩
      therefore this setting is ignored under highspeed
  LL_CONFIG <= '1'; -- must be high when using wide-bandwidth (WB)
  FPATH <= '0';    -- wide-bandwidth
  DRATE <= "101";   -- high speed mode:
```

```vhdl
  CS_inv <= '0';    -- 0=normal, 1=high impedance on DOUT_P: used when↩
       DOUT_P is communicated through a shared bus


  led(0) <= leds1; -- divided ti clock
  led(1) <= leds2; -- divided sdram clock
  led(2) <= leds3; -- divided sclk clock
  led(3) <= not(fault_fifo_in_full or fault_fifo_out_full or ↩
       fault_fifo_in_empty or fault_fifo_out_empty);

--  led(0) <= not fault_fifo_in_empty;
--  led(1) <= not fault_fifo_in_full;
--  led(2) <= not fault_fifo_out_empty;
--  led(3) <= not fault_fifo_out_full;

  led(4) <= not(fifo_in_empty);
  led(5) <= not(fifo_in_full);
  led(6) <= not(fifo_out_empty);
  led(7) <= not(fifo_out_full);


  process(BTN_left, BTN_right, min, max, steady_data, ram_data)
  begin
    if (BTN_left = '0' and BTN_right = '1') then
      hi_data <= std_logic_vector(min);
    elsif (BTN_left = '1' and BTN_right = '0') then
      hi_data <= std_logic_vector(max);
    else
      hi_data <= std_logic_vector(ram_data);
    end if;
  end process;

  pipe_out_data <= fifo_out_dout;


  ram_data <= '0' & std_logic_vector(sdram_rowaddr_in) & x"0A" WHEN ↩
       transfer_state = state_A ELSE '0' & std_logic_vector(↩
       sdram_rowaddr_in) & x"0B";

  -- WRITE DATA TO FIFO-IN IF SDRAM IS IN WRITE MODE
  process(reset, Sclk) begin
    if (reset = '1') then
      init <= '1';
      fifo_in_wr_en <= '0';
      fifo_in_din <= ( others => '0');
      ramcounter <= ( others => '0');
      cont_counter <= ( others => '0');
      rd_wr_switch <= '0';
    elsif (Sclk = '1' and Sclk'event) then
      fifo_in_din <= std_logic_vector(shift_left(filter_out1,1)+↩
           filter_out2);
      if ((ramcounter(ramcounter'left) = '1') )  then
        rd_wr_switch <= '1';
      else
        rd_wr_switch <= '0';
      end if;

      fifo_in_wr_en <= '0';
      cont_counter <= cont_counter +1;
      if (cont_counter > 4000) then -- counter used to wait for the ↩
           SDRAM compliting its initiation phase
        init <= '0';
      end if;

      if (rd_wr_switch = '0' and init = '0') then
```

```
        if (result_ready = '1') then
          ramcounter <= ramcounter +1; -- counting transmitted samples
          fifo_in_wr_en <= '1';
        end if;
      end if;
    end if;
  end process;

  sync : process(sdram_clk) begin
    if (sdram_clk'event and sdram_clk = '0') then
      sdram_wr_en <= ram_wr_en;
      sdram_rd_en <= ram_rd_en;
      sdram_reset <= reset;
    end if;
  end process;


  transferFSM : process(reset, ti_clk) begin
    if reset = '1' then
      ram_wr_en <= '0';
      ram_rd_en <= '0';
      transfer_state <= state_A;
      fifo_out_rd_en <= '0';

    elsif (ti_clk'event and ti_clk = '1') then
      ram_wr_en <= '0';
      ram_rd_en <= '0';
      fifo_out_rd_en <= pipe_out_rdy;

      case transfer_state is
        when state_A => -- WRITE TO SDRAM
          ram_wr_en <= '1';
          transfer_state <= state_A;
          if (triggers(1) = '1') then
            transfer_state <= state_B;
          end if;

        when state_B => -- READ FROM SDRAM
          ram_rd_en <= '1';
          transfer_state <= state_B;
          if (triggers(0) = '1') then
            transfer_state <= state_A;
          end if;

      end case;
    end if;
  end process;

--********************************************************************
--                    SDRAM NEGOTIATOR
--********************************************************************
--// SDRAM transfer negotiator
--//    This block handles communication between the SDRAM controller ↩
    and
--//    the FIFOs.  The FIFOs act as a simplified cache, holding at ↩
    least
--//    a full page on-chip while the PC reads the FIFO.  This ↩
    dramatically
--//    increases DRAM access performance since full pages can be read ↩
    very
--//    quickly.  Since the PC transfers are slower than the DRAM, ↩
    there is
--//    no fear of underrun.


```

```
process(sdram_clk, sdram_reset) begin
  if (sdram_reset = '1') then
    ram_state <= idle;
    cmd_pagewrite <= '0';
    cmd_pageread <= '0';
    sdram_rowaddr_in <= (others => '0');
    sdram_rowaddr_read <= (others => '0');
    sdram_rowaddr_write <= (others => '0');

  elsif (sdram_clk='0' and sdram_clk'event) then
    cmd_pagewrite <= '0';
    cmd_pageread <= '0';

    case (ram_state) is
      when idle =>
        ram_state <= idle;

        -- If SDRAM WRITEs are enabled, trigger a block write whenever
        -- the Pipe In buffer is at least 1/4 full (1 page, 512 words)↩
          .
        if ((sdram_wr_en = '1') and (unsigned(fifo_in_status(10 downto↩
             7)) >= "0100")) then --0100
          ram_state <= w_ack_wait;
          sdram_rowaddr_in <= sdram_rowaddr_write;


        -- If SDRAM READs are enabled, trigger a block read whenever
        -- the Pipe Out buffer has room for at least 1 page (512 words↩
          ).
        elsif ((sdram_rd_en = '1') and (unsigned(fifo_out_status(10 ↩
            downto 7)) <= "1000")) then
          ram_state <= r_ack_wait;
          sdram_rowaddr_in <= sdram_rowaddr_read;
        end if;



      when w_ack_wait =>
        cmd_pagewrite <= '1';
        ram_state <= w_ack_wait;
        if (cmd_ack = '1') then
          sdram_rowaddr_write <= sdram_rowaddr_write +1;
          ram_state <= busy;
        end if;

      when r_ack_wait =>
        cmd_pageread <= '1';
        ram_state <= r_ack_wait;
        if (cmd_ack = '1') then
          sdram_rowaddr_read <= sdram_rowaddr_read + 1;
          ram_state <= busy;
        end if;

      when busy =>
        ram_state <= busy;
        if (cmd_done = '1') then
          ram_state <= idle;
        end if;

    end case;
  end if;
end process;


--*********************************************************************
```

```
--                         FIFO FAULTS
--***************************************************************

  process(ti_clk, reset) begin
    if (reset = '1') then
      fault_fifo_out_empty <= '0';
    elsif (ti_clk = '1' and ti_clk'event) then
      if ((fifo_out_rd_en = '1') and (fifo_out_empty = '1')) then
        fault_fifo_out_empty <= '1';
      end if;
    end if;
  end process;

  process(sclk, reset) begin
    if (reset = '1') then
      fault_fifo_in_full <= '0';
      FC<=(others => '0');
    elsif (sclk = '1' and sclk'event) then
      if ((fifo_in_wr_en = '1') and (fifo_in_full = '1')) then
        fault_fifo_in_full <= '1';
        FC <= FC+1;
      end if;
    end if;
  end process;

  process(reset, sdram_clk) begin
    if (reset = '1') then
      fault_fifo_out_full <= '0';
      fault_fifo_in_empty <= '0';
    elsif (sdram_clk = '0' and sdram_clk'event) then

      if ((fifo_out_wr_en = '1') and (fifo_out_full = '1')) then
        fault_fifo_out_full <= '1';
      end if;
      if ((fifo_in_rd_en = '1') and (fifo_in_empty = '1')) then
        fault_fifo_in_empty <= '1';
      end if;
    end if;
  end process;

--***************************************************************
--                         CLOCK DIVIDERS
--***************************************************************

  led_clock1: process(ti_clk, reset)
  begin
    if reset = '1' then
     leds1 <= '0';
     counter1 <= 0;
    elsif (ti_clk = '0' and ti_clk'event) then
     if counter1 = C then
      leds1 <= not leds1; -- toggle divided clock
      counter1 <= 0;
     else
      counter1 <= counter1 +1;
     end if;
    end if;
  end process;


  led_clock2: process(sdram_clk, reset)
  begin
    if reset = '1' then
     leds2 <= '0';
     counter2 <= 0;
    elsif (sdram_clk = '0' and sdram_clk'event) then
     if counter2 = C then
      leds2 <= not leds2; -- toggle divided clock
      counter2 <= 0;
     else
      counter2 <= counter2 +1;
     end if;
    end if;
  end process;


  led_clock3: process(sclk, reset)
  begin
    if reset = '1' then
     leds3 <= '0';
     counter3 <= 0;
    elsif (sclk = '0' and sclk'event) then
     if counter3 = C then
      leds3 <= not leds3; -- toggle divided clock
      counter3 <= 0;
     else
      counter3 <= counter3 +1;
     end if;
    end if;
  end process;


--*************************************************************
--                    STORE MIN/MAX SAMPLES
--*************************************************************

  smaller <= '1' WHEN steady_data < min ELSE '0';
  bigger <= '1' WHEN steady_data > max ELSE '0';

  min_max: process(reset, sclk) -- storing the min. and max values
  begin
    if reset = '1' then
      min <= X"7FFFFF"; --set to largest value possible
      max <= X"800000"; --set to smallest value possible
    elsif( sclk'event and sclk='1') then
      if (fifo_in_wr_en = '1') then
        if (smaller = '1') then
          min <= steady_data;
        elsif (bigger = '1') then
          max <= steady_data;
        end if;

      end if;
    end if;
  end process;



--*************************************************************
--                    ADC DATA SAMPLES FSM
-- the LSB of the data signal is always 0 (the ADC doesn't sample this↩
     bit either when drate = 101)
--*************************************************************
dataCounter : process(sclk, data_counter_en, data_counter_reset)
  begin
  if data_counter_reset = '1' then
    data_counter <= 23;
  elsif (sclk'event and sclk='1') then
    if data_counter_en = '1' then
     data_counter <= data_counter -1;
```

```vhdl
    end if;
  end if;
end process;

process(sclk, reset)
begin
  if (reset = '1') then
     data<= x"00ff00";
    DOUT_buffer <= '0';
    adc_state <= adc_init;
    steady_data <= x"0f0f0f";
    new_steady_data <= '0';
  elsif (sclk'event and sclk = '1') then
    DOUT_buffer <= DOUT;
    adc_state <= adc_next;
    new_steady_data <= '0'; --
    data(data_counter) <= DOUT_buffer;

    if steady_data_en = '1' then
      new_steady_data <= '1';
      steady_data <= signed(data);
    end if;
  end if;
end process;

adc_output_logic : process(adc_state, data)
begin
  data_counter_reset <= '0';
  data_counter_en <= '0';
  steady_data_en <= '0';

  case adc_state is
    when adc_init =>
      data_counter_reset <= '1';
    when adc_skip =>
    when adc_idle =>
    when adc_sample =>
      data_counter_en <= '1';
    when adc_data =>
      steady_data_en <= '1';
      data_counter_reset <= '1';
  end case;
end process;

data_counter_low <= '1' WHEN data_counter = 1 ELSE '0';

adc_next_state_logic : process(adc_state, DRDY, data_counter_low)
begin
  adc_next <= adc_state;

  case adc_state is
    when adc_init =>
      if(DRDY = '1') then
        adc_next <= adc_skip;
      end if;

    when adc_skip =>
      if(DRDY = '0') then
        adc_next <= adc_idle;
      end if;

    when adc_idle =>
      if(DRDY = '1') then
        adc_next <= adc_sample;
      end if;

    when adc_sample =>
      if(data_counter_low = '1') then
        adc_next <= adc_data;
      end if;

    when adc_data =>
      if(DRDY = '1') then
        adc_next <= adc_sample;
      else
        adc_next <= adc_init;
      end if;
  end case;
end process;
--*******************************************************************



end Behavioral;
```

B.3 myfilter2.vhd

This is the VHDL implementation of the filter architecture.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;


entity myfilter2 is
  generic (INPUT_WIDTH : integer := 17;
         MULT_WIDTH : integer := 18;
         COEF_WIDTH : integer := 18;
         FRAC_SHIFT : integer := 0; -- used to sync the radix point when multiple filters are used
         CYCLES_DIVIDER : integer := 1; -- must not be changed since no clock divider/multiplier has been implemented
         MULTS : integer := 8; -- number of multipliers used in each cycle (must be even number, the final amount of multiplications must be <= 576
         filterblock : integer := 1); -- selects which filter to implemented (the filters are hardcoded)
    Port (
         data : in  signed(INPUT_WIDTH-1 downto 0);
            result : out  SIGNED(63 downto 0);
            new_data : in std_logic;
         result_ready : out std_logic;
         clk : in  STD_LOGIC;
            reset : in  STD_LOGIC);
end myfilter2;

architecture Behavioral of myfilter2 is

  COMPONENT MULT18X18
    port (
      P : out signed (2*MULT_WIDTH-1 downto 0);
      A : in signed (MULT_WIDTH-1 downto 0);
      B : in signed (MULT_WIDTH-1 downto 0));
  END COMPONENT;

-- order must odd!! up to 575

constant CYCLES : integer := 24/CYCLES_DIVIDER;-- number of multplier reuse within one data cycle (2,4,6,8,10,12)
constant COEFFICIENTS : integer := MULTS*CYCLES; -- Filter ½length, must be even
constant TAPS : integer := 2*COEFFICIENTS; --  Filter length, must be even (odd order)

TYPE mult_out_type IS ARRAY (NATURAL range <>) OF signed(2*MULT_WIDTH-1 DOWNTO 0);
  SIGNAL mult_out                     : mult_out_type(0 TO MULTS-1);

TYPE mult_in IS ARRAY (NATURAL range <>) OF signed(MULT_WIDTH-1 DOWNTO 0);
  SIGNAL mult_in_A                      : mult_in(0 TO MULTS-1);
  SIGNAL mult_in_B                      : mult_in(0 TO MULTS-1);

TYPE coeff_type IS ARRAY (NATURAL range <>) OF signed(COEF_WIDTH-1 downto 0);
  signal coeff : coeff_type(0 to MULTS-1);
```

```vhdl

TYPE delay_pipeline_type IS ARRAY (NATURAL range <>) OF signed(INPUT_WIDTH-1 DOWNTO 0);
  SIGNAL delay_pipeline                   : delay_pipeline_type(0 TO TAPS-1);

TYPE tapsum_type IS ARRAY (NATURAL range <>) OF signed(INPUT_WIDTH DOWNTO 0);
  SIGNAL tapsum                  : tapsum_type(0 TO MULTS-1);

TYPE products_type IS ARRAY (NATURAL range <>) OF signed(2*MULT_WIDTH-1 DOWNTO 0);
  SIGNAL products               : products_type(0 to (MULTS-1));

TYPE sums_type IS ARRAY (NATURAL range <>) OF signed(43 DOWNTO 0);
  SIGNAL sums               : sums_type(0 to 5);
  SIGNAL sums_buffer         : sums_type(0 to 5);

  signal resultA, resultB :  SIGNED(63 downto 0);

   -- Filter Coefficients
   signal filter_coef_addr_small : STD_LOGIC_VECTOR(7 DOWNTO 0);
  signal filter_coef_dout_small :STD_LOGIC_VECTOR (17 DOWNTO 0);

  signal filter_coef_addr_big : STD_LOGIC_VECTOR(4 DOWNTO 0);
  signal filter_coef_dout_big :STD_LOGIC_VECTOR (18*MULTS-1 DOWNTO 0);

  signal coef_counter, coef_counter2 : integer range 0 to CYCLES-1;
  signal indexA, indexB : integer range 0 to (CYCLES-1)*MULTS;

------------------------ HARDCODED FILTERS ---------------------------------------
  COMPONENT tetra800
    PORT (
      clka : IN STD_LOGIC;
      addra : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
      douta : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
      clkb : IN STD_LOGIC;
      addrb : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
      doutb : OUT STD_LOGIC_VECTOR(18*MULTS-1 DOWNTO 0)
    );
  END COMPONENT;

  COMPONENT tetra900
    PORT (
      clka : IN STD_LOGIC;
      addra : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
      douta : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
      clkb : IN STD_LOGIC;
      addrb : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
      doutb : OUT STD_LOGIC_VECTOR(18*MULTS-1 DOWNTO 0)
    );
  END COMPONENT;

  COMPONENT U50w1M
    PORT (
      clka : IN STD_LOGIC;
      addra : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
      douta : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
      clkb : IN STD_LOGIC;
      addrb : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
      doutb : OUT STD_LOGIC_VECTOR(18*MULTS-1 DOWNTO 0)
    );
  END COMPONENT;

  COMPONENT U50w12M
    PORT (
      clka : IN STD_LOGIC;
      addra : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
      douta : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
      clkb : IN STD_LOGIC;
      addrb : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
      doutb : OUT STD_LOGIC_VECTOR(18*MULTS-1 DOWNTO 0)
    );
  END COMPONENT;

  COMPONENT U150w700
    PORT (
      clka : IN STD_LOGIC;
      addra : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
      douta : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
      clkb : IN STD_LOGIC;
      addrb : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
      doutb : OUT STD_LOGIC_VECTOR(18*MULTS-1 DOWNTO 0)
    );
  END COMPONENT;

  COMPONENT U100w13M
    PORT (
      clka : IN STD_LOGIC;
      addra : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
      douta : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
      clkb : IN STD_LOGIC;
      addrb : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
      doutb : OUT STD_LOGIC_VECTOR(18*MULTS-1 DOWNTO 0)
    );
  END COMPONENT;

  COMPONENT tetra800khz2 --testfilter
    PORT (
      clka : IN STD_LOGIC;
      addra : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
      douta : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
      clkb : IN STD_LOGIC;
      addrb : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
      doutb : OUT STD_LOGIC_VECTOR(143 DOWNTO 0)
    );
  END COMPONENT;

-----------------------------------------------------------------

begin


B1 : if filterblock = 1 generate
    filter_coef: tetra800
    port map(
      clka => clk,
      addra => filter_coef_addr_small,
      douta => filter_coef_dout_small,
      clkb => clk,
      addrb => filter_coef_addr_big,
      doutb => filter_coef_dout_big
    );
end generate B1;

B2 : if filterblock = 2 generate
    filter_coef: tetra900
    port map(
      clka => clk,
      addra => filter_coef_addr_small ,
      douta => filter_coef_dout_small ,
      clkb => clk ,
      addrb => filter_coef_addr_big ,
      doutb => filter_coef_dout_big
    );
end generate B2;

B3 : if filterblock = 3 generate
    filter_coef: U50w1M
     -- (RADIX POINT @21)
    port map(
      clka => clk ,
      addra => filter_coef_addr_small ,
      douta => filter_coef_dout_small ,
      clkb => clk ,
      addrb => filter_coef_addr_big ,
      doutb => filter_coef_dout_big
    );
end generate B3;

B4 : if filterblock = 4 generate
    filter_coef: U50w12M
     -- (RADIX POINT @21)
    port map(
      clka => clk ,
      addra => filter_coef_addr_small ,
      douta => filter_coef_dout_small ,
      clkb => clk ,
      addrb => filter_coef_addr_big ,
      doutb => filter_coef_dout_big
    );
end generate B4;

B5 : if filterblock = 5 generate
    filter_coef: U150w700
     -- (RADIX POINT @20)
    port map(
      clka => clk ,
      addra => filter_coef_addr_small ,
      douta => filter_coef_dout_small ,
      clkb => clk ,
      addrb => filter_coef_addr_big ,
      doutb => filter_coef_dout_big
    );
end generate B5;

B6 : if filterblock = 6 generate
    filter_coef: U100w13M
     -- (RADIX POINT @20)
    port map(
      clka => clk ,
      addra => filter_coef_addr_small ,
      douta => filter_coef_dout_small ,
      clkb => clk ,
      addrb => filter_coef_addr_big ,
      doutb => filter_coef_dout_big
    );
end generate B6;

B0 : if filterblock = 0 generate
    filter_coef: tetra800khz2
     -- testfilter
    port map(
      clka => clk ,
```

```vhdl
      addra => filter_coef_addr_small,
      douta => filter_coef_dout_small,
      clkb => clk,
      addrb => filter_coef_addr_big,
      doutb => filter_coef_dout_big
    );
end generate B0;


  filter_coef_addr_small <= std_logic_vector(to_unsigned(coef_counter↩
      ,7)) & '1'; -- dummy value
  filter_coef_addr_big <= (others => '0') WHEN coef_counter = CYCLES-2↩
       ELSE
               std_logic_vector(to_unsigned(1, 5))WHEN coef_counter = ↩
                   CYCLES-1 ELSE
               std_logic_vector(to_unsigned(coef_counter+2,5));

  counter_control : process(clk, reset) begin
    if (reset = '1') then
      coef_counter <= 0;
      coef_counter2 <= 0; --buffered counter, used for timing ↩
          optimisation
    elsif (clk = '1' and clk'event) then
      coef_counter <= 0;
      coef_counter2 <= 0;
      if (coef_counter = CYCLES-1) then
        coef_counter <= 0;
        coef_counter2 <= 0;
      elsif (coef_counter /= 0) then
        coef_counter <= coef_counter +1;
        coef_counter2 <= coef_counter2 +1;
      elsif (new_data = '1') then
        coef_counter <= coef_counter +1;
        coef_counter2 <= coef_counter2 +1;
      end if;
    end if;
  end process;

  -- loading the coefficients for one cycle
  gen_c: for N in 0 to (MULTS-1) generate
    coeff(N) <= signed(filter_coef_dout_big(COEF_WIDTH*N+(COEF_WIDTH↩
        -1) downto COEF_WIDTH*N));
  end generate;

  -- delay pipeline
  delay_pipe : process (clk, reset) begin
    if reset = '1' then
      delay_pipeline(0 to (TAPS-1)) <= (others => (others => '0'));
    elsif clk'event and clk = '1' then
      if new_data= '1' then
        delay_pipeline(0) <= data;
        delay_pipeline(1 to  TAPS-1) <= delay_pipeline(0 to TAPS-2);
      end if;
    end if;
  end process;



-- calculationg the tapsums
indexA <= coef_counter*MULTS;
indexB <= coef_counter2*MULTS;

process(clk, reset)
    variable index1: integer ;
    variable index2: integer ;
```

```
begin
  if (reset = '1') then
    tapsum(0 TO  (MULTS-1)) <= (OTHERS =>(OTHERS => '0'));
  elsif (clk'event and clk='1') then
    for N in 1 to (MULTS) loop
      index1 := N-1+ indexA;
      index2 := TAPS-N-indexB;
      tapsum(N-1) <= resize(delay_pipeline(index1), INPUT_WIDTH+1) + ↩
          resize(delay_pipeline(index2), INPUT_WIDTH+1);
    end loop;
  end if;
end process;

-- instantiating the multipliers
gen_1: for I in 1 to MULTS generate
  MULT_A : MULT18X18
        port map (
            P => mult_out(I-1),
        A => mult_in_A(I-1),
        B => mult_in_B(I-1)
      );
end generate gen_1;


-- connecting multiplier inputs:
gen_in: for I in 1 to MULTS generate
  mult_in_A(I-1) <= resize(coeff(I-1),MULT_WIDTH);
  mult_in_B(I-1) <= resize(tapsum(I-1), MULT_WIDTH);
end generate gen_in;


-- creating muxs to control adder inputs
genning : for N in 1 to (MULTS/2) generate
  sums_buffer(N-1) <= sums(N-1) WHEN coef_counter /= 3 ELSE (others ↩
      =>'0'); -- hardcoded
end generate genning;

-- adder tree
addition : process(clk, reset) begin
  if (reset = '1') then
    result <= (others => '0');
    result_ready <= '0';
      sums(0 TO sums'right) <= (OTHERS => (OTHERS => '0'));
  elsif (clk'event and clk='1') then
    result_ready <= '0';

    for N in 1 to (MULTS) loop
      products(N-1) <= (mult_out(N-1));
    end loop;

    for N in 1 to (MULTS/2) loop
      sums(N-1) <= sums_buffer(N-1) + products((N-1))+ products(MULTS↩
          /2+(N-1));
    end loop;

      resultA <= resize(sums(0),64) + resize(sums(1),64) + resize(sums↩
          (2),64);
      resultB <= resize(sums(3),64) + resize(sums(4),64) + resize(sums↩
          (5),64);

    if (coef_counter = 4) then   -- hardcoded
      result_ready <= '1';
      result <= shift_left(resultA+resultB, FRAC_SHIFT);
    end if;

  end if;
end process;

end Behavioral;
```

B.4 ADC.xfp

This is the source code for USB GUI loaded with FrontPanel used to communicate with the FPGA.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--
ADC-FPGA Data Collector

-->


<!-- ==============================
     Panel 2
     ============================== -->
<object class="okPanel" name="panel1">
  <title>WireOut Panel</title>
  <size>320,210</size>

  <object class="okLED">
    <position>15,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x21</endpoint>
    <bit>15</bit>
  </object>

  <object class="okLED">
    <position>25,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x21</endpoint>
    <bit>14</bit>
  </object>

  <object class="okLED">
    <position>35,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x21</endpoint>
    <bit>13</bit>
  </object>

  <object class="okLED">
    <position>45,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x21</endpoint>
    <bit>12</bit>
  </object>

  <object class="okLED">
    <position>65,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x21</endpoint>
    <bit>11</bit>
  </object>

  <object class="okLED">
    <position>75,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x21</endpoint>
    <bit>10</bit>
  </object>

  <object class="okLED">
    <position>85,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x21</endpoint>
    <bit>9</bit>
  </object>

  <object class="okLED">
    <position>95,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x21</endpoint>
    <bit>8</bit>
  </object>

  <object class="okLED">
    <position>115,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x21</endpoint>
    <bit>7</bit>
  </object>

  <object class="okLED">
    <position>125,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x21</endpoint>
    <bit>6</bit>
  </object>

  <object class="okLED">
    <position>135,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x21</endpoint>
    <bit>5</bit>
  </object>

  <object class="okLED">
    <position>145,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x21</endpoint>
    <bit>4</bit>
  </object>

  <object class="okLED">
    <position>165,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x21</endpoint>
    <bit>3</bit>
  </object>

  <object class="okLED">
    <position>175,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x21</endpoint>
    <bit>2</bit>
  </object>

  <object class="okLED">
    <position>185,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x21</endpoint>
    <bit>1</bit>
  </object>

  <object class="okLED">
    <position>195,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x21</endpoint>
    <bit>0</bit>
  </object>

  <object class="okLED">
    <position>210,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x20</endpoint>
    <bit>7</bit>
  </object>

  <object class="okLED">
    <position>220,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x20</endpoint>
    <bit>6</bit>
  </object>

  <object class="okLED">
    <position>230,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x20</endpoint>
    <bit>5</bit>
  </object>

  <object class="okLED">
```

```
    <position>240,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x20</endpoint>
    <bit>4</bit>
  </object>

  <object class="okLED">
    <position>260,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x20</endpoint>
    <bit>3</bit>
  </object>

  <object class="okLED">
    <position>270,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x20</endpoint>
    <bit>2</bit>
  </object>

  <object class="okLED">
    <position>280,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x20</endpoint>
    <bit>1</bit>
  </object>

  <object class="okLED">
    <position>290,80</position>
    <size>10,10</size>
    <style>CRICLE</style>
    <color>#00ff00</color>
    <endpoint>0x20</endpoint>
    <bit>0</bit>
  </object>


  <object class="okHex">
    <color>#00f000</color>
    <position>10,10</position>
    <size>44,60</size>
    <tooltip>WireOut21[23:20]</tooltip>
    <endpoint>0x21</endpoint>
    <bit>12</bit>
  </object>
  <object class="okHex">
    <color>#00f000</color>
    <position>60,10</position>
    <size>44,60</size>
    <tooltip>WireOut21[19:16]</tooltip>
    <endpoint>0x21</endpoint>
    <bit>8</bit>
  </object>
  <object class="okHex">
    <color>#00f000</color>
    <position>110,10</position>
    <size>44,60</size>
    <tooltip>WireOut21[15:12]</tooltip>
    <endpoint>0x21</endpoint>
    <bit>4</bit>
  </object>
  <object class="okHex">
    <color>#00f000</color>
    <position>160,10</position>
    <size>44,60</size>
    <tooltip>WireOut21[11:8]</tooltip>
    <endpoint>0x21</endpoint>
    <bit>0</bit>
  </object>
  <object class="okHex">
    <color>#00f000</color>
    <position>210,10</position>
    <size>44,60</size>
    <tooltip>WireOut21[7:4]</tooltip>
    <endpoint>0x20</endpoint>
    <bit>4</bit>
  </object>
  <object class="okHex">
    <color>#00f000</color>
    <position>260,10</position>
    <size>44,60</size>
    <tooltip>WireOut21[3:0]</tooltip>
    <endpoint>0x20</endpoint>
    <bit>0</bit>
  </object>

  <object class="okFilePipe">
    <label>Capture Data</label>
    <position>90,120</position>
    <size>150,30</size>
    <endpoint>0xa3</endpoint>
    <!-- <length>33554432</length>  -->
     <length>131072</length>
    <tooltip>Read a file from Pipe 0xA3</tooltip>
    <!-- <append />
    <starttrigger><endpoint>0x40</endpoint><bit>0</bit></starttrigger>
    <donetrigger><endpoint>0x40</endpoint><bit>1</bit></donetrigger>
    -->
  </object>

  <object class="okTriggerButton">
    <label>[A] WRITE</label>
    <position>40,170</position>
    <size>100,25</size>
    <endpoint>0x53</endpoint>
    <bit>0</bit>
    <tooltip>Load 1 Page </tooltip>
  </object>

  <object class="okTriggerButton">
    <label>[B] READ</label>
    <position>180,170</position>
    <size>100,25</size>
    <endpoint>0x53</endpoint>
    <bit>1</bit>
    <tooltip>Store 1 Page </tooltip>
  </object>


</object>

<object class="okPanel" name="panel2">
```

```
      <title>ADC−FPGA Data Collector</title>
      <size>200,20</size>


      <!−− PLL22393 settings
         These will only be visible when the attached device has
         a 22393 PLL (XEM3010).
       −−>
      <object class="okPLL22393">
        <label>PLL1</label>
        <position>10,0</position>
        <size>40,15</size>
        <pll0 p="400" q="48"/>
        <output0 source="pll0_0" divider="8">on</output0>
        <output1 source="ref" divider="1">on</output1>
      </object>
      <object class="okPLL22393">
        <label>PLL2</label>
        <position>130,0</position>
        <size>40,15</size>
        <pll0 p="400" q="48"/>
        <output0 source="pll0_0" divider="16">on</output0>
        <output1 source="ref" divider="2">on</output1>
      </object>

      <object class="okCFrontPanel">
        <object class="IsFrontPanelEnabled">
        </object>
      </object>

    </object>

</resource>
```

B.5 myfilter2.m

This is the code used to simulate the filter architecture in MATLAB.

```
clc
close all
clear all

% load coefficients
fid=fopen('tetra.txt','rt');
a=fscanf(fid,'%c');
fclose(fid);
coef1 = regexp(a,',','split');

C = (size(coef1,2)); %number of rows

coefs1 = zeros(1,C);

for i = 1:C
   coefs1(i) = str2double(cell2mat(coef1(i)));
end

TAPS = 2*C;
MULTS = 8;
cycles = 24;
S = 1000; %samples
data = ones(1,S);


d_pipe = zeros(1,TAPS);
results = zeros(1,TAPS-100);
tapsums = zeros(1,MULTS);
coeff1 = zeros(1, MULTS);
products1 = zeros(1, MULTS);

for L = 1:S;
    result_temp1 = 0;
    d_pipe(2:TAPS) = d_pipe(1:TAPS-1);
    d_pipe(1) =  data(L);
    for M = 1:cycles; %multiplier cycles
        index = (M-1)*MULTS;
        for N = 1:MULTS; %multiplier data
            index1 = N + index;
            index2 = TAPS+1 - (N + index) ;
            tapsums(N) = d_pipe(index1)  + d_pipe(index2);
        end
        for N = 1:MULTS;
            coeff1(N) = coefs1(N+(M-1)*MULTS);
        end
        for N = 1:MULTS;
            result_temp1 = result_temp1 + tapsums(N) * coeff1(N);
        end
    end
       results(L) = result_temp1 ;
end


Fs = 378000;                    % Sampling frequency
T = 1/Fs;                        % Sample time
L = length(results);             % Length of signal
t = (0:L-1)*T;                   % Time vector
```

```
NFFT = 2^nextpow2(L);          % Next power of 2 from length of y

sample = zeros(1,L);
for k = 1:L;
   sample(k) = k;
end


figure(1);
subplot(211);
    plot(sample,results,'r-');
subplot(212);
        Y = fft(results,NFFT)/L;
        f = Fs/2*linspace(0,1,NFFT/2+1);
        YY = 2*abs(Y(1:NFFT/2+1));
        YYdb = mag2db(YY/max(YY));
        plot(f,YYdb,'r')
```

Bibliography

[1] Micron Technology Inc. Sdr sdram mt48lc64m4a2, mt48lc32m8a2, mt48lc16m16a2 specifications. PDF: 09005aef8091e6d1, 256Mb_sdr.pdf - Rev. N 1/10 EN, 1999.

[2] Opal Kelly Incorporated. Xem3010 user's manual. 20091113.

[3] Texas Instruments. Ads1675. 4MSPS 24-Bit Analog-to-Digital Converter, SBAS416D-DECEMBER 2008-REVISED AUGUST 2010.

[4] Texas Instruments. Ads1675ref. User's Guide, SBAU162A-December 2009-Revised September 2010.

[5] Dimitris G. Manolakis John G Proakis. *Digital Signal Processing - Principles, Algorithms, and Applications 4. edition.* Pearson, 2007.

[6] Opal Kelly. Frontpanel user manual. 2012-01-03.

[7] Edmund Lai. *Practical Digital Signal Processing for Engineers and Technicians.* Elsevier, Newnes, 2004.

[8] Ricardo A. Losada. *Digital Filters - Principals and Applications with MATLAB.* John Wiley and Sons Inc, 2011.

[9] Uwe Meyer-Baese. *Digital Signal Processing with Field Programmable Gate Arrays 3. edtion.* Springer, 2007.

[10] Fred J. Taylor. *Digital Filters with MATLAB.* The MathWorks Inc., 2008.

[11] Xilinx. Timing constraints user guide. UG612 (v 13.1), March 1, 2011.