Arbitrary Decimation for High Sample Rates, Algorithm Design and FPGA implementation

FREDRIK PETERSON
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Arbitrary Decimation for High Sample Rates, Algorithm Design and FPGA implementation.

Fredrik Peterson
elt14fpe@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Assoc. prof. Liang Liu

Examiner: Prof. Erik Larsson

August 21, 2019

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

This master thesis investigates how to perform irrational decimation, the process of reducing the sample rate of a signal, for high throughput systems. The thesis work consists both an algorithmic part and an implementation part. In the algorithmic part, an algorithm that could cope with the requirements is found, and investigated due to different aspects. A group of filters that are investigated further are different version of the so called farrow structure. The farrow structure can be used in a lot of different applications but the interesting thing here is that decimation can be made, quite arbitrary, while all parameters of the structure are kept static, aside from one control parameter that must be recalculated for every sample. The result of the algorithm investigation is an algorithm that consists of a transposed farrow filter in series with a fixed halfband filter. The series connection of the two different filters were found to be necessary to cope with the given requirements of the project. Especially the requirements on the stopband attenuation (80 dB) and passband ripple (0.02 db) in combination with the bandwidth (80%), mad it impossible to use just a farrow filter without using too high filter orders. Both the transposed farrow structure and the halfband filter can be implemented, using high levels of parallelization, which was necessary in the implementation phase, to be able to match the high throughput demands.

During the implementation part, the algorithm is implemented onto a FPGA, a Xilinx Virtex Ultrascale. The implementation aims towards proving that the algorithm could be used for high throughput applications and is implemented in 8 parallel at a clock speed of 312.5 MHz. The parallelization in combination with the clock speed gives the implementation a capacity of decimating an incoming signal at a sample rate of 2.5 GSa/s. To solve the problem of the multiple data rates within the implementation, a shift register is used to only execute the main parts of the implementation when all parallelization branches have valid data at their inputs. This transforms almost all the problem with the multirate system to a much simpler data driven implementation. The decimation factor can be selected from 2 and upwards with a certain resolution. The resolution is higher for smaller decimation factors than for larger ones.

Acknowledgements

I would like to thank Teledyne SP Devices for the opportunity of doing my master thesis work in collaboration with the company and all employees at he company for their help, kindness and all interesting discussions. Especially, I would like to thank Mikael Gustavsson for always taking time to answering my questions and discuss difficult problems or suggestions. I would also like to thank Jens Månsson and Martin Olsson for sharing their knowledge within the area of FPGA implementations.

From Lund University, I would like to thank the supervisor of the master thesis Liang Liu as well as the examiner Erik Larsson.

Finally, I would like to express my gratitude towards all friends and family who have been supportive throughout the whole process and helped me to keep the spirit up when the work has been demanding.

Fredrik Peterson

Popular Science Summary

Today more and more electronic things work of digital signals instead of analog signals. Digital signals have a lot of favorable properties over analog signals and the development, with faster electronics and computers makes it possible to perform more and more advanced stuff with the signals. In this master thesis, a way that might make the digital signals easier to work with, is presented.

There are two types of signals, analog and digital. An analog signal is continuous, which means that it has a value at any given time. A digital signal is not continuous but instead discrete, which means that it only have values at specific time points. One can convert an analog signal to a digital signal by saving values of the analog signal at specific times. The process of saving the values of the analog signal, making it digital, is called sampling. How often the signal is sampled is called the sample rate of the signal. The time period between the times, where the data is saved, are always equal. As an example one can save the value of the analog signal at every second, giving the signal a sample rate of 1 sample per second (Sa/s). Depending on what one wants to make with the digital signal one might want to have signal, that is not sampled every second but where the time points are separated by a larger time, as an example instead every minute. To obtain the values of a digital signal that is samples every minute instead of every second, the signal must be processed through a so called decimation algorithm. A decimation algorithm can calculate what the values should have been if the analog signal, already from the beginning, would have been sampled at these longer times instead of the old short ones. This thesis work is about how to perform this change, between a higher sample rate and a lower one.

There exist many ways to perform decimation, but the special thing about the work of this master thesis is that we want to be able to change the sample rate between one static value (a fast one) and any lower one. For example if we have the signal that had a sample rate of 1 sample per second (Sa/s), we want to be able to convert it to 1/60 Sa/s, or 1/1000 Sa/s or even 1/2.5472 Sa/s. This makes the whole thing a lot more difficult. In addition to this the actual sample rate of the signal is not 1 Sa/s but for example 5 GSa/s, that's 5 billion samples per second. This is very fast, and the method must then be able to calculate 5 billion values every second, this is very fast and it is not easy!

Table of Contents

List of Figures

List of Tables

Terminology

ADC - Analog-to-Digital Converter

DAC - Digital-to-Analog Converter

ENOB - Effective Number Of Bits

Fir - Finite impulse response

FPGA - Field Programmable Gate Array

$\boldsymbol{F_s}$ - Sample frequency

H(f) - Frequency response

h(t) - Impulse response

Lsb - Least Significant Bit

M - Number of branches of a farrow filter

Msb - Most Significant Bit

N - Length of each branch in a farrow filter

R - Decimation factor

S - Scale Factor

Sa - Samples

SNR - Signal to Noise Ratio

SFDR - Spurious-Free Dynamic Range

$\boldsymbol{T_{in}}$ - Sampling time of the input signal

$\boldsymbol{T_{out}}$ - Sampling time of the output signal

VLSI - Very Large Scale Integration

$\boldsymbol{\mu}$ - Control parameter

Chapter 1
Introduction

This master thesis has been carried out as a project at Teledyne SP Devices, a company that develops digitizers. The problem investigated was sample rate conversion and more specific decimation. In a digitizer the analog to digital converter (ADC) generally works at a fixed sample rate but the system would be more flexible if the output sample rate could be varied (decreased) and this master thesis investigates how to achieve this in a manner that could be efficiently implemented.

1.1 Background

Because of easier configuration and a larger flexibility more and more functionality of modern electronics systems and signal processing are moved from hardware to software or programmable hardware. This migration is made possible of the development of faster and more energy efficient digital components, which includes DSP processors and FPGAs. A trend within communication systems as well as other high data rate systems in need of signal processing is to move the converter (analog to digital or digital to analog) closer to the antenna and make as much as the signal processing as possible in the digital domain using digital components instead of processing the analog signal using analog components.

The digitizers developed by Teledyne SP Devices include firmware which parts of is running on a field programmable gate array (FPGA) board included at the digitizer's printed circuit board (PCB). The analog to digital converter (ADC) generally have a fixed sample rate but there are applications which would gain of a more flexible sample rate, in this case lower than the sample rate of the ADC. The intended applications of the implementation will be mainly within the RF field, where a signal often has a high carrier frequency but the signal content is in comparison to that frequency narrow band.

1.1.1 Decimation

Decimation is the process of reducing the sample rate of a signal. It could be said that there are two categories with several subcategories of decimators. The two main categories are programmable and fixed, and the category refers to the flexibility of the decimator. A fixed decimator always decimate by the same factor where a fully programmable decimator could be set to different factors. Both of

these categories could also be divided further into integer, rational and irrational converters. An integer decimator could decimate by an integer factor. A rational decimator can decimate by a factor D/I where I and D are integers and D are greater than I for decimation. Almost all numbers could be described as rationals but the limitations of a rational decimator lays in the fact of how the rational decimation is achieved. The most common way of achieving rational decimation is to first interpolate by I and then decimate by D. This leads to a really high data rate if I is large. An irrational decimator could potentially decimate by any factor. The scope of this thesis is within the last case, an irrational or arbitrary decimator.

There are several reasons for decimating a signal but a key to able to do this is that the signal is oversampled or that the interesting content of the signal can be fitted into the maximum bandwidth of the new sample rate, otherwise some of the content of the signal will be lost or damaged. Two reasons for decimation are to save resources, by either the data reduction or complexity reduction of the algorithm, or to match the sample rate of a connected system or algorithm.

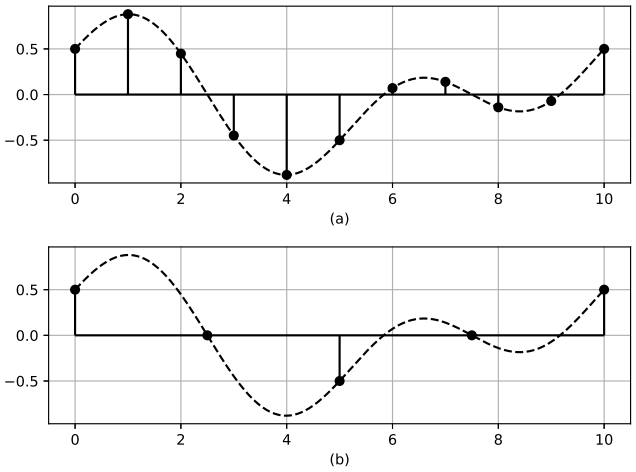


Figure 1.1: (a) a sampled signal. (b) the same signal decimated by a factor $R = 2.5$.

Resources can be saved because of that operations on fewer samples generally requires less computations than more samples. Resources could in this case be memory size, energy consumed when an algorithm is ran or area consumed by the implementation of the algorithm on a chip or FPGA. Energy could be saved both because that less computations are needed but might be even further reduced if the clock speed or supply voltage could be reduced.

One example algorithm is fast Fourier transform (FFT) which uses N/2log(N) complex multiplications [1]. If N is halved in this case the reduction in computations is more than halved.

A reduction of sample rate makes the design of components working on the signal with lower sample rates easier and less complex. As examples the degree of

parallelisation may be reduced or filters could use fewer coefficients due to lower demands.

In the case of sample rate matching it could be that an algorithm is implemented for a certain sample rate and to be able to connect several systems the sample rate need to be converted between them. In this case the sample rate conversion could be both downwards (decimation) or upwards (interpolation).

1.2 Issue and Requirements

Decimation is common in modern electronics systems and the absolute majority of these implementations are of the fixed decimation type. The scope of this master thesis is to investigate ways to make an as arbitrary decimator as possible. The decimator should be possible to implement in an efficient way at a FPGA and an implementation, as a proof of concept, is also to be made. There are some specified requirements, seen in table 1.1, that the implementation should try to cope with. In general it could be said that all of these requirements are rather strict and that high precision algorithms will have to be used to match the requirements. The bandwidth, stopband attenuation, and passband ripple demands will lead to high order filters. The sample rate demands are really high and high levels of parallelization of the algorithm will be needed for the implementation.

Requirement	Value
Bandwidth	80 % of $F_s/2$
Stopband attenuation	- 80 dB
Passband Ripple	$\pm$ 0.008 dB
Maximum input sample rate	1 x 5 GSa/s or 2 x 2.5 GSa/s

Table 1.1: Requirements of the decimator.

Throughout the project several difficulties will have to be addressed. The first one is to find a good balance between performance and FPGA resource usage. One of the keys here is to succeed to give a good estimate of the resource usage, prior to the implementation to be able to choose the best algorithm. Another difficulty is that 2.5 GSa/s or 5 GSa/s are a really high sample rates which make the implementation more difficult. A high degree of parallelization will be needed to cope with this demand and the algorithm must be implementable with this high degree of parallelization. One last difficulty is that algorithm's purpose is to change the sample rate. This means that somewhere within the steps of the algorithm the sample rate will change and the FPGA implementation must be able to handle this rate change, which in this case not will be known beforehand. The main difficulties that will need to be resolved during the project are as follows:

- Find a good balance between performance and resource usage.
- Make an implementation that can handle really high sample rates.
- Find a way to cope with the sample rate change at the FPGA.

1.3 Outline

In the first chapter of the report the topic of the master thesis was introduced and the task of the work presented. The second chapter gives some of the theory needed to understand the topic, such as further details about decimation and different structures to use for decimation. In the third chapter the methodology of the work is presented. The fourth chapter present the result of the work. In this chapter the suggested algorithm are presented as well as a proof of concept FPGA implementation. The last chapter summarizes the work and gives suggestion for future work on the topic.

Chapter 2
Theory

2.1 Decimation

Decimation is the process of reducing the sample rate of a digital signal. Depending on the signal and the requirements of the output signal, this process might be more or less complicated. A trivial case is decimation by an integer factor, R, in cases when neither aliasing nor imaging occurs. In these cases it just a matter of keeping every I:th sample and disregard the others. In non trivial cases a lowpass filter is needed to attenuate parts of the signal [2].

2.1.1 Spectrum

When a signal is decimated the spectrum is changed. To avoid aliasing the signal must be bandlimited to $F_s/2R$, where R is the decimation factor. This also means that the signal content could only be fully preserved, in the case of decimation, if the signal is bandlimited so that aliasing does not occur, otherwise information will be lost. The spectrum of a decimated signal is seen in fig. 2.1.

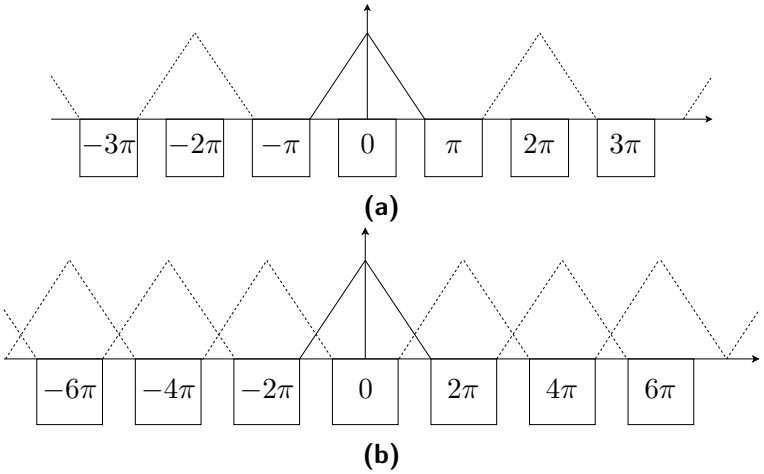


Figure 2.1: (a) The spectrum of a digital signal. (b) The same spectrum but the signal is decimated by a factor of 2.

2.2 Hybrid analog model

When working with decimation, or interpolation, structures one convenient why to model the system is the "hybrid analog model" [4], [5]. This model allows the lowpass filter to be designed in the frequency domain which is necessary to handle the behavior of the filter for frequencies above F_s. The model transforms the problem of designing the filter from the time domain to the frequency domain [7]. The model can be seen in Fig. 2.2 and it consists of an ideal digital-to-analog converter (DAC) followed by the lowpass filter and a resampling step that converts the signal back to digital with the new sampling frequency.

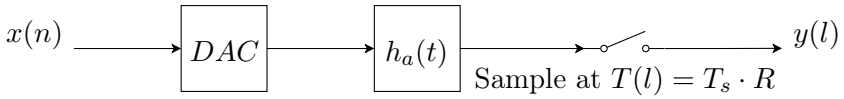


Figure 2.2: The hybrid analog model.

2.3 Farrow Structure

The farrow structure, fig. 2.3, was originally suggested as a continuously variable digital delay filter which was primary targeted as a part of a system for echo cancellation [3]. The structure consists of M parallel finite impulse response (FIR) filters of order N. A single control parameter, μ, sets the delay of the filter while all other parameters remain static. This fact makes it suitable for a Very-large-scale integration (VLSI) implementation. Depending on the choice of the control parameter, together with the coefficients, the structure can perform different tasks. The farrow structure is also mentioned as a polynomial filter since the filter, in a mathematical sense, can be seen as piece-wise defined polynomials where the control parameter change the variable part of the polynomial and the filter coefficients set the coefficients.

A small note about the notation, m denotes one of the branches $0, \ldots, M-1$ and n denotes one coefficient number $0, \ldots, N$ in one of the branches.

2.3.1 Modified Farrow Structure

A small modification of the farrow structure has to be made to be able to design the filters in the frequency domain, which is a necessity to be able to reach the specified requirements, section 2.3.5. The modification is that instead of multiplying the input with μ^m, $(2\mu-1)^m$ is used [6]. This modification is from this point assumed in all cases if not otherwise noted.

This modification leads to a change of the basis functions which are described in section 2.3.2 and are simple, yet crucial for the ability to design the filters.

2.3.2 Basis functions

The farrow structure, as earlier, mentioned consists of piece-wise defined polynomials. These polynomial can be described by a coefficient together with a basis

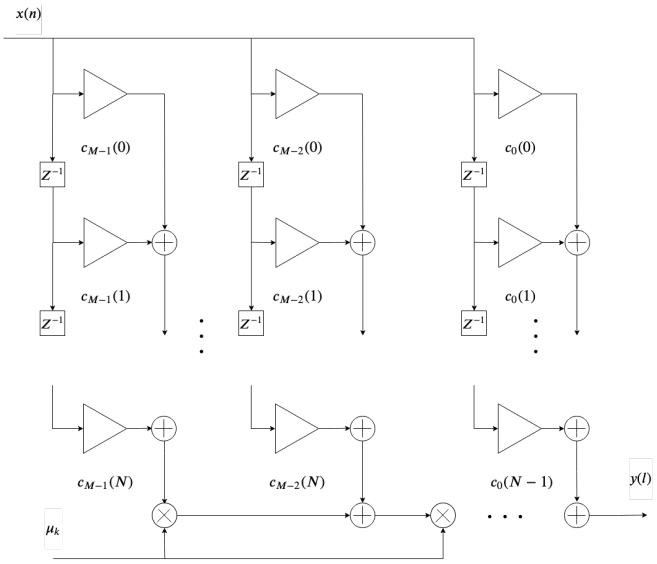


Figure 2.3: The farrow structure.

function [6]. The equation for the basis function can be seen in eq. 2.1. It can be seen that for example $m = 0$ gives a zero order function and that $m = 1$ gives a linear function, and so on. The four first orders of the basis function are plotted in Fig. 2.4.

$$f_m(t) = \begin{cases} \left(\frac{2t}{T_s} - 1\right)^m & 0 \leq t < T_s \\ 0 & otherwise \end{cases} \tag{2.1}$$

2.3.3 Impulse response

Together with coefficients the basis functions describes the polynomials of the farrow structures and these builds the impulse response of the whole filter. The impulse response is a summation over all the coefficients and corresponding basis functions as seen in eq. 2.2.

$$h(t) = \sum_{n=-N/2}^{N/2-1} \sum_{m=0}^{M} c_m(n) f_m(t - nT_s) \tag{2.2}$$

A symmetric FIR filter has linear phase which is a desirable property of the farrow structure when used for decimation. If all the sub filters are symmetric (or anti-symmetric) the whole structure will have a symmetric impulse response and gain the benefits from a linear phase behavior. A symmetric filter also requires just half the number of unique coefficients which is a good point when it comes to implementation of the filter. Using the symmetric (and anti-symmetric) property the coefficients can be described by $c_m(-n) = c_m(n-1)$ for m even and $c_m(-n) =$

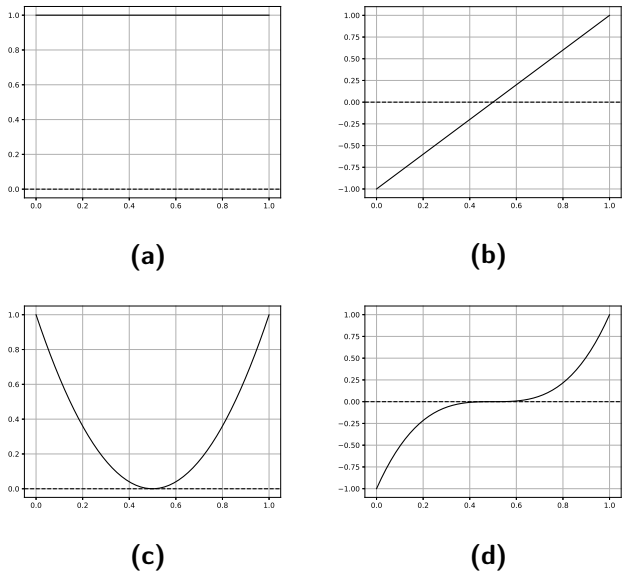


Figure 2.4: The basis functions up to order 3, for m = 0 to 3 (a to d).

$-c_m(n-1)$ for m odd [6]. The summation in the impulse response, eq. 2.3 can then be summed just over half the number of the coefficients using a modification of the basis function equation 2.4.

$$h(t) = \sum_{n=0}^{N/2-1} \sum_{m=0}^{M} c_m(n) g_m(n, T_s, t) \tag{2.3}$$

$$\begin{aligned} g_m(n, T_s, t) &= f_m(t - nT_s) + (-1)^m f(t + (n+1)T_s) \\ &= \begin{cases} \left(\frac{2(t-nT_s)}{T_s} - 1\right)^m & nT_s \leq t < (n+1)T_s \\ (-1)^m \left(\frac{2(t+(n+1)T_s)}{T_s} - 1\right)^m & -(n+1)T_s \leq t < -nT_s \\ 0 & otherwise \end{cases} \end{aligned} \tag{2.4}$$

2.3.4 Frequency response

The frequency response of the farrow structure can be obtained as the Fourier transform of the impulse response from eq. 2.3 [5].

$$H_a(f) = \sum_{n=0}^{N/2-1} \sum_{m=0}^{M} c_m(n) G(n, m, T_s, f) \tag{2.5}$$

The function G, eq. 2.6, is again the Fourier transform of the modified basis function from eq. 2.4 and eq. 2.7 is just used to make the equation easier to read.

$$G_m(n,T,f) = \begin{cases} 2cos(2\pi f(n+\frac{1}{2})T)((-1)^{m/2}m!\Phi(m,T,f) + sinc(\pi fT)) & m\ even \\ 2(-1)^{\frac{m+1}{2}}Tm!sin(2\pi f(n+\frac{1}{2})T)\Phi(m,T,f) & m\ odd \end{cases} \tag{2.6}$$

$$\Phi(m,T,f) = \sum_{k=0}^{\lfloor (m-1)/2 \rfloor} (\pi fT)^{2k-1}\frac{(-1)^k}{(2k)!}\left(sinc(\pi fT) - \frac{cos(\pi fT)}{2k+1}\right) \tag{2.7}$$

2.3.5 Filter Design

Time domain design

When designing farrow filters in the time domain, classical interpolation techniques are used. Two examples are Lagrange interpolation and B-Splines [7]. The time domain design is easy but suffers from problem with the performance of the filters. This fact limits the usage of the time domain design approach to systems which not require sharp filters or filters with high attenuation in the stopband.

Frequency domain design

When the farrow filters are designed in the frequency domain the principle is to minimize the difference between the frequency response, eq. 2.5, and the desired frequency response $D(f)$ [6]. The desired response is 1 in the passband and 0 in the stopband. The region between the passband and the stopband, called the transition band, has no requirements. A weighting function, $W(f)$ can be used to add different weighting for different parts of the frequency response for example the passband versus the stopband. One can choose different functions to minimize for example summed least squares or minmax. In the case of summed least squares, the function to minimize is the sum of all errors (the error at each frequency point) squared. The square is used to get rid of the problem with negative errors. In the minmax case, which is used throughout this project, one want to minimize the maximum error shown in eq. 2.8. The minimization can be made using different minimization algorithms.

$$\epsilon = \max_f \{W(f)(H(f) - D(f)\} \tag{2.8}$$

2.4 Transposed Farrow Structure

Both the original farrow structure and the modified version perform poorly when it comes to decimation, especially regarding their anti-aliasing ability [8]. To solve this problem the whole structure is transposed and the length of the polynomials, e.g the basis functions, are selected to be T_{out}. Effectively this results in that the farrow filter can be designed in relation to the output signal's sample rate and requirements. An overview of the transposed farrow structure can be seen in Fig. 2.5.

In contrary to the non-transposed farrow structure the transposed farrow structure does not conserve the energy of the signal. To conserve the energy the output from the structure has to be divided by the decimation factor, R.

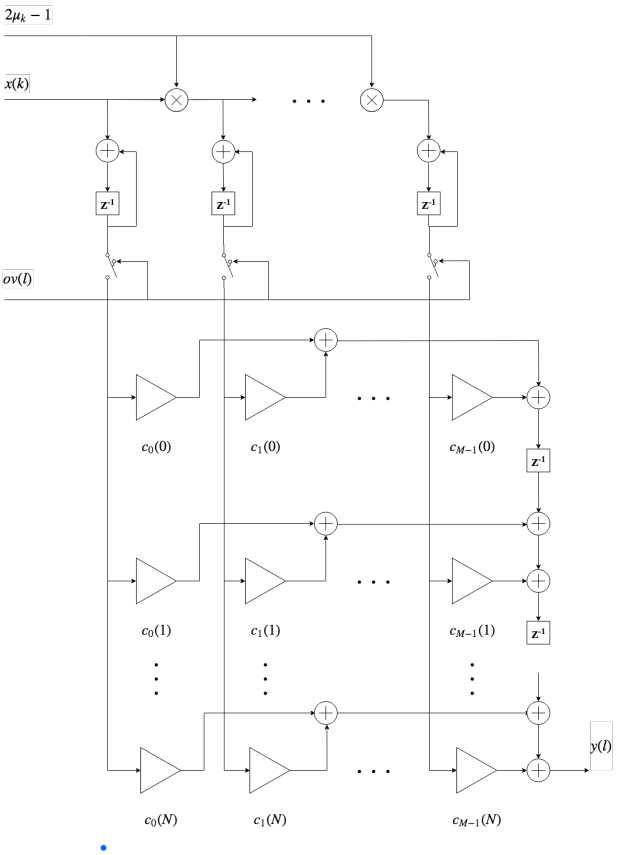


Figure 2.5: The transposed modified farrow structure.

In the case of the transposed farrow structure μ is the relative distance between the input sample and the latest output sample, which is described by fig. 2.6. The distance is normalized by the output sample interval, T_{out}. Because of this, the following condition must always be satisfied, $0 \leq \mu < 0$. μ can be calculated iteratively as described by eq. 2.9.

$$\mu_{k+1} = \mu_k + 1/R - \lfloor \mu_k + 1/R \rfloor \tag{2.9}$$

In the transposed farrow structure a number of input samples might be accumulated after multiplication by $2\mu - 1$. The number of accumulated samples is easily understood as the number of input samples in the interval between two output samples. In the case of no decimation, i.e R = 1, this number is always equals to one. But in other cases the number will vary between $\lfloor R \rfloor$ and $\lfloor R \rfloor + 1$, if R is not an integer. In the integer case the number will not vary but always be equal to R. The number can also be calculated using eq. 2.9, every time $\mu_k + 1/R$

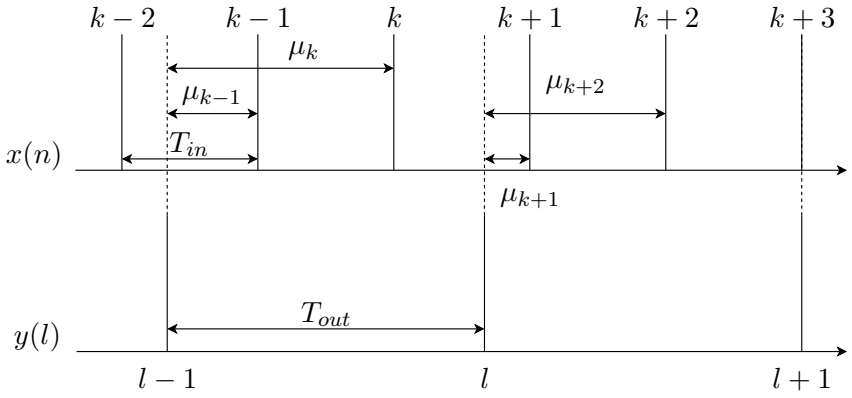


Figure 2.6: Block schematic of the accumulator_top module, showing it's submodules and their connections.

is bigger than one, the current accumulator value should be dumped and a new accumulation should begin from zero.

2.5 Halfband filters

A halfband filter is a L:th band FIR filter where $L = 2$ and they are widely used for decimation [10]. The halfband filter is a filter that decimates by a factor of 2 and it's frequency response is "anti-symmetric" around $f_s/4$. The symmetry means that the ripple in the passband is of the same order as the attenuation in the stopband. When the number of taps are odd the filter has the special property that every other filter coefficient is 0 and that the filter is symmetric. These facts can be used to make efficient implementations.

2.6 Cascaded structures

A cascaded structure is a structure where one of the farrow structures are used in series with some other structure for decimation or interpolation, an example is seen in Fig. 2.7. The purpose of using a cascaded structure is to relax the requirement on the filters. Relaxed requirements results in that fewer coefficients are needed and that the overall complexity of the system might be reduced [9]. There are a lot of different cascaded structures but in this thesis work a combination of the farrow structure and a halfband filter will be used. If the halfband filter is put after the transposed farrow structure the requirement on the bandwidth of the farrow structure will be halved. The frequency responses of the two structures can also be co-optimized to further reduce the complexity.

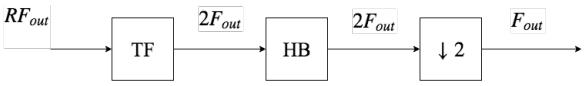


Figure 2.7: An example of a cascaded decimation structure.

2.7 Polyphase Representation

When working with decimation or interpolation some tricks can be applied to always use the filters at the lower sample rate which reduce the number of computations needed, in this section some of these are presented.

To start with, two so called identities are presented, the first one can be seen in Fig. 2.8 [2]. This identity describes the relationship between branching, multiplication and decimation. The decimators can be placed at all the branches or just at the sum of the branches. The decimators can also be placed either before or after multiplication without affecting the result.

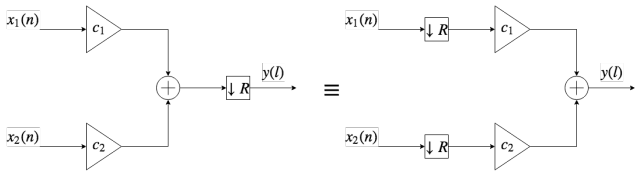


Figure 2.8: Multiplication and decimation can be in any order and are independent of branching.

The second identity presented can be seen in Fig. 2.9 [2]. From the figure it is seen that decimation can be either before or after filtering, if the filter are adjusted accordingly.

These to identities lay the foundation for the following so called polyphase representation of a filter [2], [1]. A normal single branch FIR filter can be split into multiple branches according to eq. 2.10. In the example seen in Fig. 2.10 the filter (H) is split into two branches denoted H_0 and H_1. In the next step, the decimation is moved to in front of each filter, which reduces the sample rate through the filters by a factor of 2. In the last step the decimators are replaced by a switch that alters the sample between the two branches. This has effectively changed a filter running at F_s to two half the length filters running at $F_s/2$.

$$H(z) = \sum_{k=0}^{N-1} z^{-k} H_k(z^N) \tag{2.10}$$

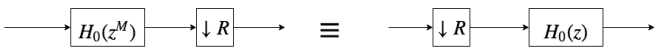


Figure 2.9: Decimation and filtering can switch places if taps of the filter are adjusted.

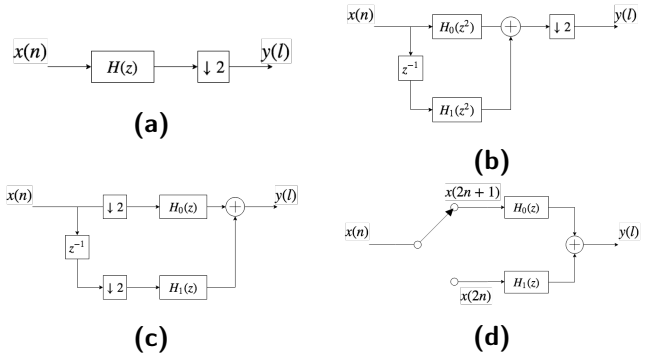


Figure 2.10: Decimation and filtering can switch places if taps of the filter are adjusted.

Chapter 3
Methodology

This master thesis consist of two major parts, namely a signal processing or algorithm part and an implementation part. The steps within these two parts are described in further detail in the following sections.

3.1 Algorithm development

The first step of the work is to try to find an algorithm that could fulfill the requirements. The algorithm must both be able to give good enough performance but also be implementable in an efficient way. When an algorithm has been chosen it must be tested, through simulations, before implementation. Two different types of simulations were used. The two version were a floating point python simulation and a fixed point simulation also built using python in combination with LTSpice.

3.1.1 Literature studies

To find algorithms that could be used to pass the requirements of the project literature studies are made. These studies aims toward finding different algorithms for decimation that could suit the requirements of the project. Important parameters to look for, when the studies are being made, are the arbitrariness of the decimation and that efficient implementations are possible.

3.1.2 Floating point simulations

The floating point model implements the equation describing the filter structure. The first step was to find the number of coefficients needed. This was an iterative process where different numbers where tried out and the resulting frequency response was compared to the requirements of the structure. The optimal values of the coefficients were found using minimization algorithms.

When a filter structure that fulfilled the requirements was found the behaviour of the structure was investigated by passing signals with different frequencies through the filters and setting different decimation factors.

3.1.3 Fixed point simulation

The problem with taking a floating point model and implementing it on a FPGA using fewer bits is that the behavior will change, the question is just how much. The fixed point simulation aims towards answering this question and to fine tune the parameters for best suiting the implementation. Fewer bits will lead to larger rounding of the coefficients and the products and sums in the filter. This rounding both change the frequency response of the filter and generates some rounding noise.

Teledyne SP Devices has developed python packages that converts the normal floating point behavior of python into fixed point numbers using a specified number of bits and location of the decimal point. Using this tool a fixed point model was generated. By rerunning the earlier simulations the behavior could be compared and analyzed.

3.2 Implementation

The biggest change from the previous models and the implementation is that parallel processing will be needed to be able to run the algorithm in real time at the FPGA. The implementation could be split into two parts, filters and a parallel processing part which aims to get the samples to the correct location in the filters. The HDL development, simulation and synthesis is carried out using the Xilinx Vivado tools.

3.2.1 Filter implementation

The filters are implemented by a python script that generates the verilog code of the filter. The purpose of doing it this way is to keep the flexibility to easily change the numbers of filters or the the orders of the filters if later needed.

3.2.2 Parallel processing

The clock of the FPGA is throughout the scope of this master thesis work static and set to 312.5 MHz. The fact that the implemented algorithm should ran at 5 GSa/s or 2x2.5 GSa/s make it necessary to implement it in parallel. Parallelization means that several input samples will be handled in each clock cycle and that several output samples might be produced (depends on the decimation factor). The parallelization of the algorithm complicates the implementation since it is not only to implement several instances of the algorithm. The difficulty lays in that all instances are dependent on each other and that the samples need to be directed to the correct instance. One additional thing that complicates the parallelization in this case is that the data rate in the structure not will be fixed but depend on the decimation factor.

In the case of an input sample rate of 5 GSa/s, 16 parallel instances will be needed and in the case of 2.5 GSa/s, 8 instances are needed.

3.2.3 Binary Multiplication

Another area that potentially can cause trouble is in all the multiplication throughout the algorithm. In general, multiplying two binary numbers with N and M number of bits will result in a product of N + M bits. Throughout this thesis work the input data is a two's complement number of 16 bits in total, 1 sign bit and 15 fractional bits, and the output data should have the same format. Within the steps of the algorithm the number of bits are limited to 27, where multiplications are made because this is the maximum input size that one DSP-element can handle at the larger of the two ports. To solve this problem, either truncation or rounding must be made at several locations within the algorithm. In this thesis work, three slightly different implementations were made with slightly different approaches to the rounding/truncation. The problems with truncation an/or roudning are explained in the following sections.

Truncation

The simplest way to deal with the excess bits is to use truncation. Truncation means that the excess bits are disregarded and only the number of wanted bits are considered. The problem with using this method is that it will introduce truncation error which, depending on the application, might or might not be acceptable. When the excess bits are disregarded the remaining bits will always be interpreted as a number that is smaller than or equal to the full bit number. This fact will introduce a bias resulting in an offset [1]. Depending on how many truncations that are made and how many bits that are truncated the overall truncation error can be large. In fig. 3.1 the process of truncation, in the case where the binary numbers are integers, are showed. The truncation process works in the same way when fractional binary numbers are used, but instead of truncation to the closest integer, the value are truncated to the closest value that can be represented by the current number of bits.

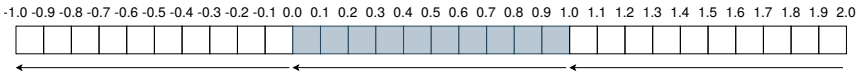


Figure 3.1: Truncation always make the resulting value smaller.

Rounding

If truncation is not giving a small enough error some other method to deal with he excess bits must be used. The problem with he truncation was mainly the introduction of the bias when the resulting value always was smaller than or equal to the real value. The introduced bias can be reduced by using a rounding scheme instead [1]. There exists a lot of different methods for rounding [13], but the method used in this thesis is the simplest possible, where a number equal to or greater than 0.5 are rounded upwards values smaller than 0.5 are rounded downwards. This method still introduce a bias, because of that the range of numbers rounded upwards are slightly larger than the range of number rounded

downwards. However this bias is much smaller than in the case of truncation. In fig. 3.2 the process of rounding, in the case where the binary numbers are integers, are showed. The principle of the rounding process is the same even for fractional binary numbers, but instead of rounding to the closest integer the number is rounded to the closest number that could be represented by the current number of bits.

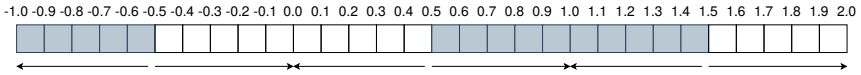


Figure 3.2: Rounding, rounds the number to the closest integer with 0.5 rounded upwards.

This rounding method can easily be implemented by adding a 1 to the binary number, with the wanted number of bits, plus one excess bit and then using the sum without the excess lsb in the sum as the rounded value [13].

3.2.4 Behavioral simulations

To investigate if the behaviour of the implemented filters are correct simulations are needed. Testvectors were generated using the floating point model. These vectors were then compared to the output from the different implementations. This way, the function of the implementations could be verified.

Chapter 4
Results

4.1 Filter design

The requirements of the decimation structure is seen in table 1.1. The filter must be designed to cope with the requirements when it comes to passband ripple, stopband attenuation and relative bandwidth. The filters are designed in the frequency domain, by minimizing eq. 2.8, according to the hybrid analog model, section 2.2. The result of the minimization, together with different parameter values are presented below. The parameters that could be changed, of a farrow structure, are the number of subfilters, the order of the subfilters, the weight function and the frequency vector over which the minimization is made. In addition to these parameters the farrow filter can also be combined in series with other filters. In the following sections the transposed farrow structure is used, either alone or in series with a halfband filter.

Frequency Vector

The frequency vector used in the following minimizations has been iteratively found and the same vector is used for all the following results. The distance between the frequency points in the vector must be small enough to avoid peaks in the frequency response between them, but more points than needed make the optimization unnecessarily slow. In addition to the distance between the frequency point the start and endpoint must also be found. It has showed that starting from frequency 0 is not a good choice since the equation for the frequency response, eq. 2.5, is not accurate close to 0. A good starting frequency has been found to be $0.02F_s$. The trend of the frequency response is that the attenuation increases with higher frequencies, but the end frequency must be set high enough to make the peaks after this point smaller than the attenuation in the optimized frequency band. A higher frequency than needed, again, leads to either more points to optimize or a less dense vector. A good end frequency has been found to be $6.5F_s$. The frequency vector is linearly spaced with 100 points in the passband $0.02F_s$ to $(0.5relbw)F_s$, and 451 point in the stopband $(1-(0.5relbw))F_s$ to $6.5F_s$. $relbw$ is 0.8 in the case of just the transposed farrow structure and 0.4 in the case of the cascaded structure where the transposed farrow structure is in series with a halfband filter. The frequency vector has no frequency points in the transition

band, since folding into this band is allowed. The parameters of the frequency vector are seen in table 4.1.

Frequency vector	Start (F_s)	Stop F_s	Points	Weight
Passband frequencies	0.02	0.5relbw	100	0.1
Stopband frequencies	1-(0.5relbw)	6.5	451	1

Table 4.1: The frequency vector used in the filter simulations.

Weight Vector

According to the requirements of the design, the attenuation should be one order higher than the maximal allowed passband ripple. This makes the choice of weight vector quite obvious. The weight vector used throughout the following filter optimizations is 0.1 for the frequency points within the passband and 1 for the points in the stopband. This weight vector means that the maximum error in the passband can be 10 times larger than the maximum error in the stopband.

4.1.1 Transposed Farrow Structure

In this section results from minimization attempts, when only the transposed farrow structure was considered, are presented. In table 4.2 the required orders of the subfilters, to fulfill the 80 dB attenuation, for different numbers of parallel filters are presented. The resulting number of coefficients and the maximum error according to eq. 2.8 are included as well.

Nfilt	ford	max error	# coefficients (unique)
4	> 100	-	-
6	21	$5.21 \cdot 10^{-5}$	66
8	21	$5.09 \cdot 10^{-5}$	88
10	21	$5.04 \cdot 10^{-5}$	110
12	19	$9.95 \cdot 10^{-5}$	114

Table 4.2: Filter orders needed to fulfill the project requirements for different numbers of subfilters, in the case of only the transposed farrow filter.

The best alternative, of the ones presented in table 4.2, would be the one with the fewest unique coefficients. In this case, this is the alternative using 6 subfilters of order 21. The frequency response of this filter is seen in fig. 4.1.

4.1.2 Halfband filter

The coefficients of the halfband filter was found using the Scipy implementation of the Remez algorithm [15]. The frequency response of the halfband filter must,

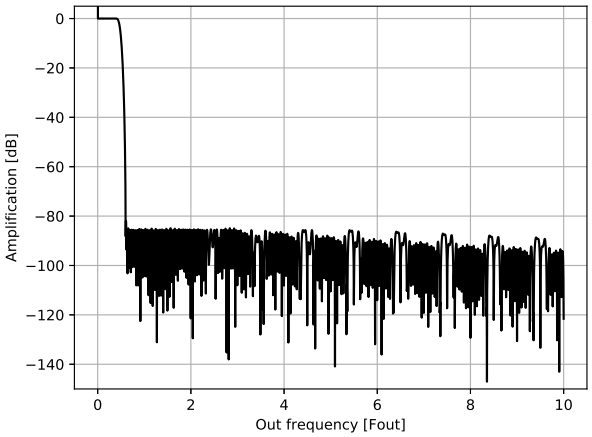


Figure 4.1: The frequency response of the transposed farrow filter with 6 subfilters of order 21, and designed with relbw = 0.8.

when summed with the frequency response of the transposed farrow filter, cope with the requirements of the project. Since the passband ripple and attenuation of the halfband filter directly corresponds, the one with the highest demands will set the required length of the filter. The toughest requirement should in this case be the total attenuation but since the attenuation of both the filters will be summed the ripple in the passband will instead be limiting factor. The coefficients of the resulting filter is seen in table 4.6.

4.1.3 Cascaded structure

The cascaded structure consists of the transposed farrow structure in series with a halfband filter, as seen in fig 2.7. By using the filters in series, the bandwidth requirement on the transposed farrow structure are relaxed from $0.8F_out$ to $0.4F_out$ which means that fewer coefficients are needed to fulfill the requirements. In additional to the bandwidth relaxation the filters could also be co-optimized which might further reduce the number of coefficients needed or give a smaller maximum error of the overall design. The main drawback of the cascaded structure is that the minimal decimation is increased from 1, which is the case when decimation is made only the transposed farrow structure, to 2. This is because the halfband filter always performs a decimation by 2.

Without co-optimization

When the transposed farrow filter and the halfband filter are designed without co-optimization they are designed separately. The overall frequency response of the structure is then obtained as the sum (in log-scale) of both the frequency responses. The lowest order of the subfilters, still fulfilling the requirements for

different number of subfilters are presented in table 4.3.

Nfilt	ford	max error	# coefficients (unique)
4	> 100	-	-
6	7	$3.53 \cdot 10^{-5}$	$24 + 13$
8	7	$3.36 \cdot 10^{-5}$	$32 + 13$
10	7	$3.31 \cdot 10^{-5}$	$40 + 13$
12	7	$3.26 \cdot 10^{-5}$	$48 + 13$

Table 4.3: Filter orders needed to fulfill the project requirements for different numbers of subfilters in the case of cascaded structure without co-optimization.

According to table 4.3 the best alternative of the suggested structures would be the one where the transposed farrow structure consists of 6 subfilters of order 7. The halfband filter is designed such that the minimum number of coefficients still fulfilling the requirements are used. The frequency response of this transposed farrow filter, together with the halfband filter and the sum of both the frequency responses are seen in fig. 4.2.

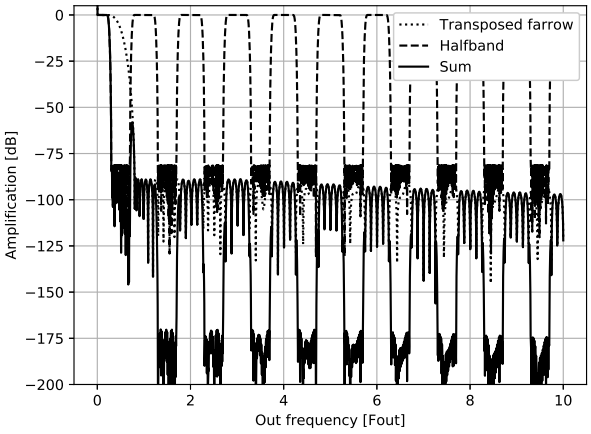


Figure 4.2: The frequency response of the transposed farrow filter with 6 subfilters of order 7, together with the halfband filter and the sum of both. No co-optimization.

With co-optimization

When utilizing co-optimization the halfband filter is first generated. The frequency response of the halfband filter is then used as a weight in the optimization of the transposed farrow filter. The results for the minimal order filters, still fulfilling

the requirements, are seen in table 4.4. Comparing these results, with the ones obtained from the filter design without co-optimization, seen in table 4.3, it is seen that the order cannot be reduced but the maximal error is significantly reduced.

The alternative with fewest coefficients, still fulfilling the requirements is the one with 6 subfilters of order 7. The, overall, resulting frequency response of this alternative can be seen in fig. 4.3. As seen in the figure the co-optimization allows the frequency response of the transposed farrow filter to peak at points where the halfband filter has a high attenuation. This lead to a smaller maximum error than in the design without the co-optimization.

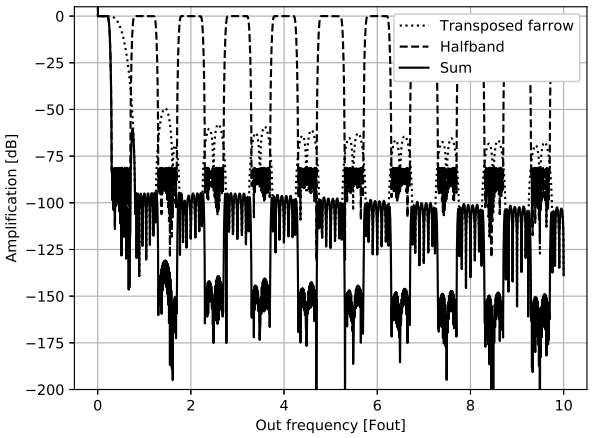


Figure 4.3: The frequency response of the transposed farrow filter with 6 subfilters of order 7, together with the halfband filter and the sum of both. With co-optimization.

subfilter	ford	max error	# coefficients (unique)
4	> 100	-	-
6	7	$1.76 \cdot 10^{-5}$	$24 + 13$
8	7	$1.68 \cdot 10^{-5}$	$32 + 13$
10	7	$1.65 \cdot 10^{-5}$	$40 + 13$
12	7	$1.60 \cdot 10^{-5}$	$48 + 13$

Table 4.4: Filter orders needed to fulfilled the requirements for different numbers of subfilters in the case of cascaded structure with co-optimization.

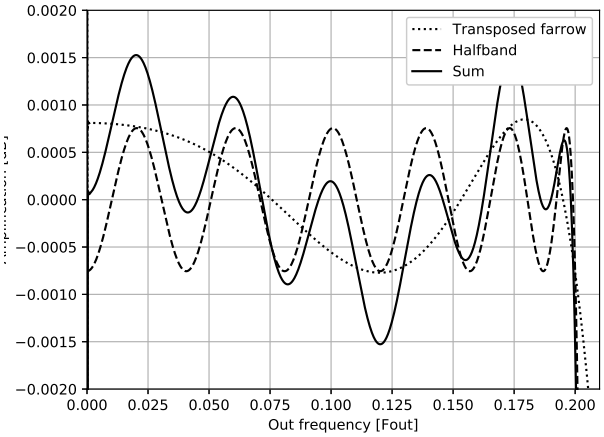


Figure 4.4: The passband of the frequency response of the transposed farrow filter with 6 subfilters of order 7, together with the halfband filter and the sum of both. With co-optimization.

4.2 Simulations

Two different models were used for the simulations, a floating point model and a fixed point model. The floating point model is used to estimate the performance of the algorithm and to get a more accurate estimate the fixed point model is used. The following sections include results from the simulations, describing the performance for different cases.

4.2.1 Python model simulations

First an example where a decimation factor of 2.3 is presented. Sinusoidals with an amplitude of 1 and varying frequency has been passed through the filter structure. The frequency was swept from a point close to DC to 0.4 F_out which is the upper bandwidth requirement from table 1.1.

In fig. 4.5, the signal to noise ratio (SNR) is plotted. To fulfill the requirements the attenuation should be at least 80 dB, and from the plot minimal SNR around 88 dB is seen. According to the figure the attenuation seams to be high enough for the specific case to fulfill the requirement.

Spurious-free dynamic range (SFDR) is the difference, in dB, between the signal tone and the strongest non-wanted noise tone. In addition to fulfilling the requirements the SFDR should reflect the attenuation from the frequency response to verify the function of the system. The minimal attenuation of the filters used is about 95 dB and inspecting fig. 4.6 the worst SFDR is about 93 dB which in this case is rather close to the expected result.

The signal, out from the system, should conserve the energy of the input signal within the variation allowed by the requirement on the passband ripple, which is

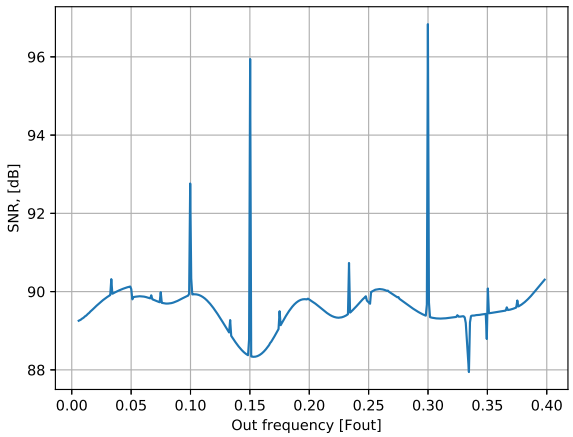


Figure 4.5: SNR of floating point model simulation. R = 2.3.

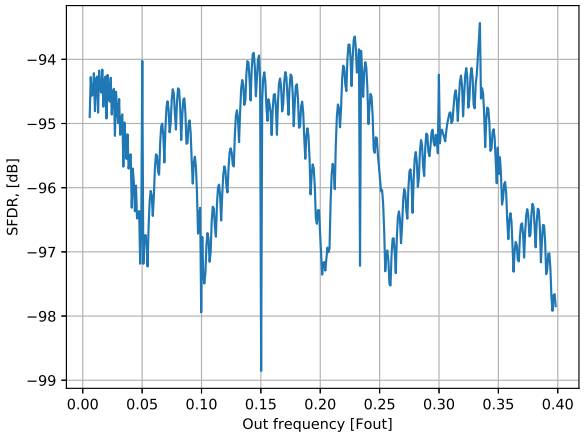


Figure 4.6: SFDR of floating point model simulation. R = 2.3.

n \ m	0	1	2	3	4	5
0	-0.00454	-0.00139	0.00602	0.00287	-0.00072	0.00073
1	0.03280	0.00974	-0.03946	-0.01770	0.00436	-0.00102
2	-0.13387	-0.05168	0.1461	0.07166	-0.00919	-0.00042
3	0.60564	0.61577	-0.11260	-0.14634	0.00550	0.00252
4	0.60564	-0.61577	-0.11260	0.14634	0.00550	-0.00252
5	-0.13387	0.05168	0.1461	-0.07166	-0.00919	0.00042
6	0.03280	-0.00974	-0.03946	0.01770	0.00436	0.00102
7	-0.00454	0.00139	0.00602	-0.00287	-0.00072	-0.00073

Table 4.5: Coefficients of the transposed farrow filter co-optimized with the halfband filter.

0.0	-0.000198	0.0	0.0005768	0.0	-0.0013522
0.0	0.0027292	0.0	-0.0049882	0.0	0.0084995
0.0	-0.0137886	0.0	0.0217132	0.0	-0.03398
0.0	0.0549449	0.0	-0.1006574	0.0	0.3164574
0.5	0.3164574	0.0	-0.1006574	0.0	0.0549449
0.0	-0.03398	0.0	0.0217132	0.0	-0.0137886
0.0	0.0084995	0.0	-0.0049882	0.0	0.0027292
0.0	-0.0013522	0.0	0.0005768	0.0	-0.000198
0.0					

Table 4.6: Coefficients of the halfband filter co-optimized with the transposed farrow filter.

0.02 dB. In fig. 4.7, the amplitude change compared to the input amplitude is plotted. The result from this example is within the 0.02 dB requirement. The amplitude of the signal also seams to follow the pattern of the frequency response plot of the passband, seen in fig. 4.4, which is an expected result.

The model most be verified for a lot of different decimator factors and a lot of different signal frequencies. This is a time consuming task and in fig. 4.8 and fig. 4.9 the result from an attempt of verification is plotted. From the plots two general trends can be seen. The first one is that both the SNR and the SFDR is better with higher decimation factors where the big change is within the span of factors up to 50. The second trend seen is that almost all the values are within the requirements and those which don't all have one thing in common. The values that shows a really bad SFDR and or SNR is at $1/3$ or $1/4$ of F_{out}. The result is unwanted but hard to fix, the cause is that at these frequencies a lot of signals are folded on top of each other. This is a common problem within the area of signal processing.

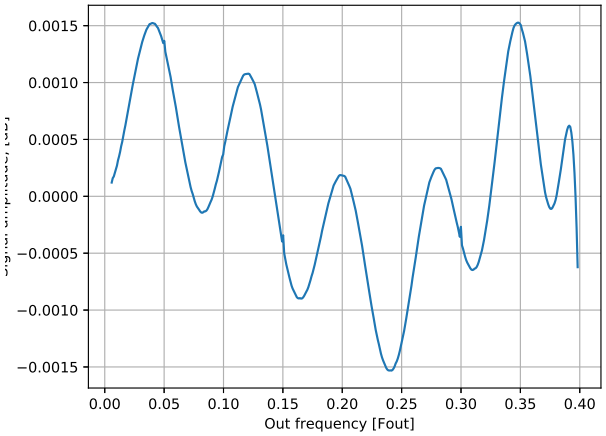


Figure 4.7: Amplitude of floating point model simulation. R = 2.3.

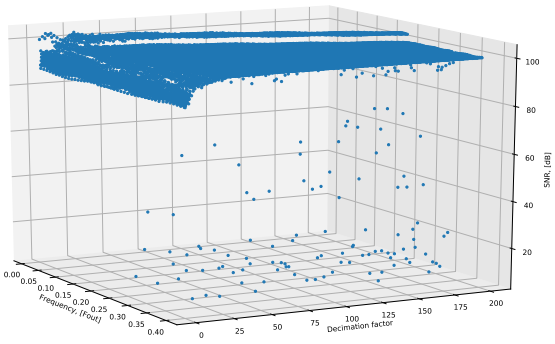


Figure 4.8: SNR of floating point model simulation.

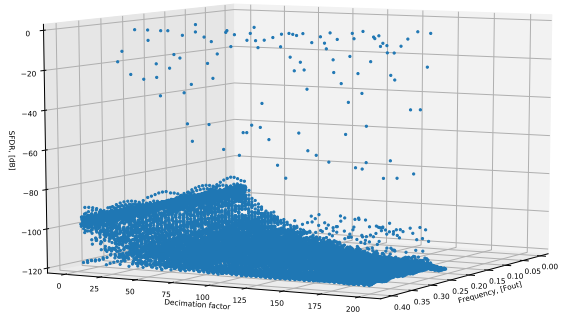


Figure 4.9: SFDR of floating point model simulation.

4.2.2 Fixed point model simulations

The main intention of the fixed point model simulations was to use it to verify that a fixed point implementation should work at all. After this the model was also intended to be used to investigate the number of bits needed in each part of the design. It turned out that this was much more difficult to achieve than expected. The main source of difficulty was to handle truncation and/or rounding correctly throughout the design when a fractional binary representation was used. The model worked quite well for signals without the fractional representation (all signals and coefficients were considered to be integers, by moving the decimal point to "infinity").

The results from a working model could also has been used as ground truth for the implementation. But since the produced results from the simulations did not achieve the expectations they could not be used for this. The only functions of the results was that they have been used during certain parts of the implementations phase to verify the data flow in the implementation. This works when both the models are fed with integer values as describe above.

The last purpose of the fixed point model was to show that the filter responses did not change to much when the filter coefficients used are rounded, due to a limited number of bits. The floating point coefficients, seen in tables 4.5 and 4.6, were rounded to a 16 bit binary number (1 sign, 15 fractional), the resulting frequency response are seen in fig. 4.10 and fig. 4.11. The results show that the frequency response of the fixed point coefficients still fulfilled the requirements.

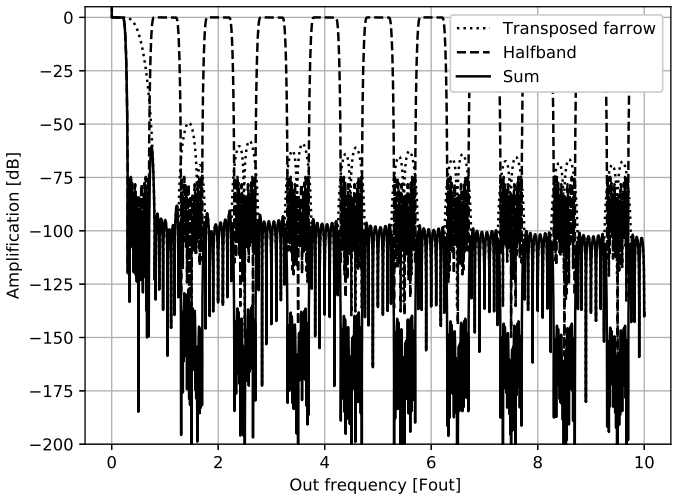


Figure 4.10: Frequency response of the transposed farrow filter, the halfband filter and the sum of those. All filters use the fixed point coefficients.

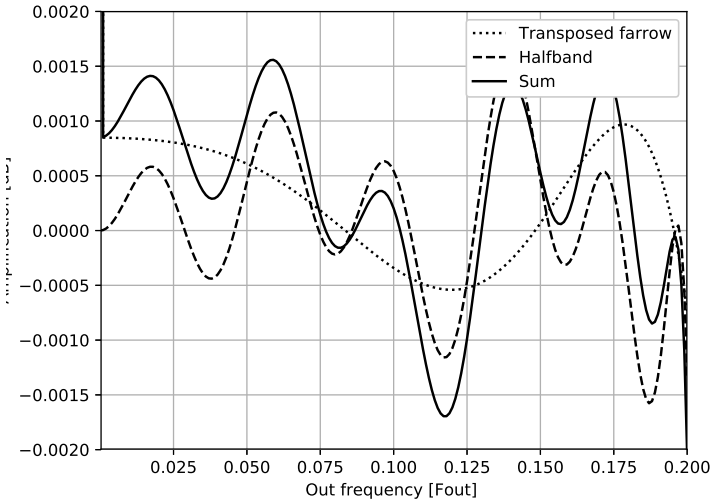


Figure 4.11: The passband of the frequency response of the transposed farrow filter, the halfband filter and the sum of those. All filters use the fixed point coefficients.

4.3 Implementation

In the following sections the FPGA implementation of the decimation algorithm will be described. The implementation is made in a 8-parallel manner and is, at the current FPGA chip, running with a clock speed set to 312.5 MHz. This means that the implementation can decimate a signal with an input sample rate of 2.5 GSa/s. The implemented algorithm consists of a transposed farrow structure with 6 subfilters of order 7 in series with a halfband filter with 48 coefficients. The decimation factor can, in the implementation, be selected from 2 and upwards.

One advantage of the transposed farrow structure is that the sample rate is lowered already after integrate and dump step. This means that if the minimum decimation factor would be increased, the degree of parallelization after this step can be reduced. This could be used to lower the resource usage. However, this is not used in the current implementation since it should be able to decimate as arbitrary as possible. In the description of the implementation, the decimation factor, R, is used for the decimation in the transposed farrow structure. The total decimation will be 2R, after the decimation by 2 in the halfband filter.

The implementation is split into four different Verilog modules which are, accumulate_top, farrow_top, scale_data, and halfband_top, as seen in fig. 4.12. The different parts directly corresponds to the different parts of the algorithm.

In all the following figures, the thicker double lines arrows corresponds to 8-parallel data paths. The thinner, single lined, arrows corresponds to control signals or data paths that not are in parallel.

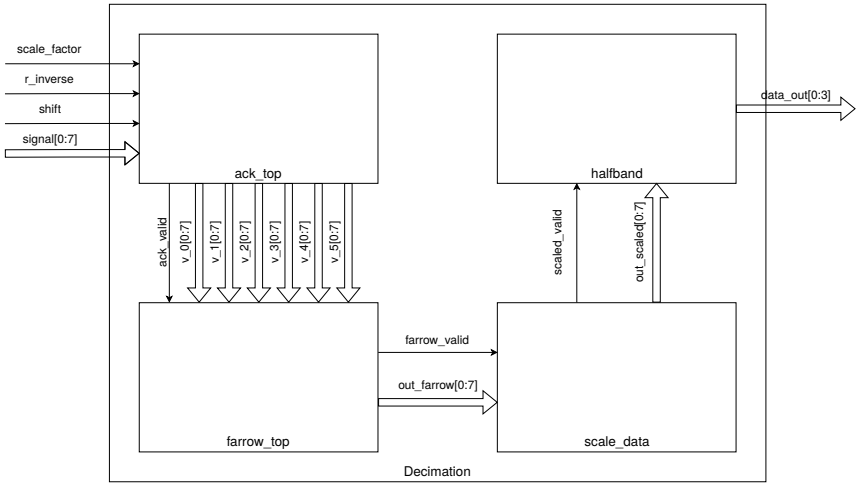


Figure 4.12: Block schematic of the top modules of the implemented design.

4.3.1 accumulate_top

An overview of the accumulate_top module is seen in fig. 4.13. The functions of this module are to accumulate the incoming samples a variable number of times,

after multiplication with the factor $2\mu - 1$, as well as iteratively calculating the μ. It is also in this module that the first data rate change is performed, by the factor R. It should also be said that this part is the only part of the whole design that must be able to handle a variable data rate. This is true because after this step all parallelization branches always will be filled but not executed at every clock cycle. The problem of the variable sample rate are transformed to a data driven design.

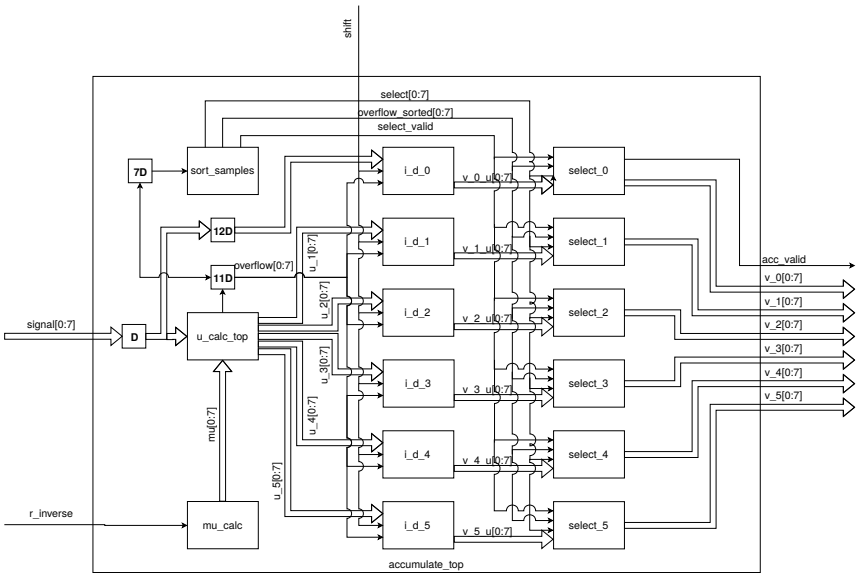


Figure 4.13: Block schematic of the accumulator_top module, showing it's submodules and their connections.

All the submodules are described in more detail in the following sections but in summary the modules function in the following way. r_inverse, which is 2/R, is passed to the mu_calc module where it is iteratively added to find the μ corresponding to each input sample. The μs are then multiplied by 2 and the product is then subtracted by one, generating the control parameter $2\mu - 1$. Every time the addition of r_inverse accumulates to a value greater than, or equal to, one an overflow is generated. The $2\mu - 1$ values are forwarded to the u_calc module where it is multiplied with the incoming signal, multiple times to generate the u-signal. The u-signal is split into 6 parts, one for each branch of the farrow filter, and is passed to the different integrate_dump modules. In these modules the u-samples are accumulated, the number of accumulated samples are controlled by the earlier mentioned overflow signal. The output signals from integrate_dump modules are unsorted and just some of the samples should be used. These samples are sorted and chosen by the select_samples modules controlled by the signals from the sort_samples module. When 8 valid samples are available at each of the 6 output branches a acc_valid is set high, indicating to the farrow_top module that all it's inputs are valid.

mu_calc

The tasks of the mu_calc component are mainly to produce the $2\mu - 1$ factor, which need to be calculated for every input sample, and to indicate how many samples that should be accumulated in the integrate and dump step later. The main difficulty of this module is in the nature of iterative calculation of μ, eq. 2.9. The iterative calculation means that the value before the wanted one must be known, eg. μ_{k-1} must be known to calculate μ_k. This fact would result in that a chain of additions, of the length of the number of parallelization branches, must be performed at each clock cycle to achieve the required throughput of 8 new μs every clock cycle. Such a chain can not be implemented, since it is too long to meet the timing requirements of the design. Instead some sort of trick must be used to decrease the length of chain of consecutive logic elements. In this implementation the solution to this issue is to use multiples of 1/R for the addition. The multiples can be generated through simple shift operations, which are both fast, and lean. To further increase the performance these multiples and the results of some of the additions can be performed in the previous clock cycle, the con of doing this is that it will slightly increase usage of logic elements. However, compared to the overall design, this small increase is not noticeable.

The second task of this module is, as previously mentioned, to indicate how many samples that should be added in the integrate and dump modules. This feature is implemented in the way of letting the iterative additions overflow and use the carry. As seen from eq. 2.9, only the decimal part of μ is interesting for the later calculations. The number of bits for the sums are chosen such that there are only fractional bits and an overflow bit. The overflow bit is then indicating that the sum would have been greater than one, and this is the condition that indicates that the integrate and dump module should dump it's current value and reset the accumulation to zero.

4.3.2 u_calc_top

The u_calc_top module includes 8 instances of a module called u_calculator. Each of these submodules has a throughput of one sample per clock cycle, which means that they all together achieve the necessary throughput of 8 samples per clock cycle. The implemented farrow filter has 6 branches and each of this branches are connected to a separate accumulator. Each of these accumulators have different inputs according to $x(2\mu - 1)^m$, where x is the input sample and m is the branch number, reaching from 0 to 5. This means that each input sample should be multiplied with the corresponding mu-value 5 times, where the product after each multiplication is the input to one of the accumulators. The input to the first accumulator corresponds to $x(2\mu - 1)^0 = x$, which is simply the input sample.

Inside each of the 8 instances of u_calculator are 5 instances of a manually instantiated DSP48 element [11]. The reason for this manual instantiation is that the timing requirements not were met when the tool generated the implementation. In addition there are several register to handle the pipelining of the module and to achieve a synchronized output. By synchronized it is meant that all the outputs that corresponds to the same input sample, but have been multiplied different number of times by $2\mu - 1$, should be at the output simultaneously.

integrate_dump

The 6 instances of the integrate_dump module implement the accumulators, located before each branch of the transposed farrow filter. Each integrate_dump module always output 8 samples, but depending on the decimation factor, between 0 and 8 of these are valid samples that should be passed to the transposed farrow filter. The non valid samples are just intermediate products that later should be disregarded. In short, each module instance sums the values from the u_calc module until a 1, indicating overflow in the calculation of mu occurs. Then a new accumulation from zero is started.

The accumulations of large numbers of samples could potentially lead to overflow in the accumulators if these do not have a sufficient number of bits. The implementation aims toward being able to decimate the signal by the factors up to several thousands. This would mean that the accumulators would need up to somewhere around 15 extra bits to handle this accumulation. In addition to this, it is not only the accumulators that would need an increased number of bits but this would affect all the multipliers, registers, and adders in the farrow structure as well. To solve this issue, without increasing the number of bits, all the values that should be accumulated are divided by the closest two factor number greater than the factor R. This ensures that the accumulated value never are greater than 1, at the cost of accuracy of the accumulations. The loss of accuracy will lead to increased noise, but this noise will only correspond to the loss of one bit or 6 dB. The division is implemented as a simple shift operation, and this operation will lead to a division by factor D, eq. 4.1. An alternative approach to this problem would have been to to implement an exact division by R, i.e multiplication by 1/R. But to use that method would lead to a much higher resource usage, compared to the chosen solution in combination with performing the exact scaling in a later stage. The reason for this is that in this stage the data is split into 6 branches compared to just one in the stage where the scaling is made in the current implementation. One third alliterative is to put the scaling before the data is split into 6 branches. But since this would mean that it must be performed before the multiplication by $2\mu - 1$, but in this case the error will be amplified throughout the multiplication stages and the whole swing wouldn't in most cases be used since the multiplication always make the product smaller than or equal to the input.

$$D = \lceil log2(R) \rceil \tag{4.1}$$

sort_samples

The task of the sort_samples module is to generate signals for the control of the instances of the select_samples modules. As earlier mentioned in the integrate_dump module description, all eight samples out from every integrate_dump module are not valid, in most cases. The number of valid samples can be calculated by counting the number of ones in the overflow signal generated by u_calc. The module generates three control signals with slightly different purposes. The first one is a signal called select_valid, this signal is triggered when eight ones

has been counted. When the signal goes high it indicates that the output from the select_samples module will be valid. The second one is a signal called overflow_sorted and this signal is a shifted version of the input overflow signal. All ones are shifted to the lowest position not already occupied by a one, beginning from a counter value. The last control signal is simply called select and this signal indicates, for the select_samples modules, at which position there are valid samples. The position is referenced as a three bit number 0-7. The only positions cared about in the select signal is the ones that corresponds to the ones in the sorted overflow signal, and the values at the other positions are irrelevant.

An example will be given to further illustrate the behavior of the control signals. The example describes the control signals at three consecutive clock cycles. overflow is the input overflow signal, counter is an internal signal to keep track of the position to shift the values to and overflow_sorted is the output signal. Lastly select is the select signal and in the example the values of this signal is written using decimal numbers, corresponding to the indexes of the valid samples, and x are don't care. In the first clock cycle, 5 valid samples exist, 5 ones are shifted to the five most lsb bits of overflow_sorted, and the counter counts up to 5. Since less then 8 valid samples has occurred, select_valid will be 0. select is set to the indexes of the valid samples at the positions of the ones in the overflow_sorted signal. In the second clock cycle, 5 additional valid samples is indicated. 5 ones are again shifted into overflow_sorted, but since the counter is at 5 at the beginning of this clock cycle they are shifted to position starting at 5 (lsb position is indexed 0), and starting from zero again after the msb at position 7 have been given a one. After this clock cycle more than 8 samples have passed and select_valid is set to 1. In the last clock cycle of the example, four new valid samples are indicated. Four ones are shifted into overflow_sorted from position 2 and select valid are set to zero, since 8 additional valid samples not have occurred.

counter = 0, overflow = [10101011] ⇒ overflow_sorted = [00011111], select_valid = 0, select = [xxx75310]

counter = 5, overflow = [11010101] ⇒ overflow_sorted = [11100011], select_valid = 1, select = [420xxx76]

counter = 2, overflow = [10101010] ⇒ overflow_sorted = [00111100], select_valid = 0, select = [xx7531xx]

select_samples

In the select_samples instances the valid samples from the integrate_dump instances are sorted depending on the input control signals. The control signals are generated by the sort_samples module and are described in the section describing that module. The outputs from the select_samples module are a vector with a width of 8 samples, called v_sorted and a valid_out signal. The v signal is the input to the transposed farrow filter and valid_out indicates that the v signal is valid. The samples in the v signal are sorted such that the first valid sample is at position 0 and the preceding samples at the preceding indices. Each instance of the select_samples module generates the samples for one of the six branches of the farrow structure.

The behavior is further illustrated using an example. In the example the input samples, from an integrate and dump module, are denoted $u_{cc,idx}$, where cc is the clock cycle of the sample and idx the position of the samples in the 8 samples wide input signal u.

overflow_sorted = [00011111], select_valid = 0, select = [xxx75310] ⇒
filled = [00011111], filled_next = [00000000], valid_out = 0,
v_sorted = [$000u_{0,7}u_{0,5}u_{0,3}u_{0,1}u_{0,0}$], v_sorted_next = [00000000]

overflow_sorted = [11100011], select_valid = 1, select = [420xxx76] ⇒
filled = [11111111], filled_next = [00000011], valid_out = 1,
v_sorted = [$u_{1,4}u_{1,2}u_{1,0}u_{0,7}u_{0,5}u_{0,3}u_{0,1}u_{0,0}$],
v_sorted_next = [$000000u_{1,7}u_{1,6}$]

overflow_sorted = [00111100], select_valid = 0, select = [xx7531xx] ⇒
filled = [00111111], filled_next = [00000000], valid_out = 0,
v_sorted = [$u_{1,4}u_{1,2}u_{1,0}u_{0,7}u_{0,5}u_{0,3}u_{0,1}u_{0,0}$], v_sorted_next = [00000000]

4.3.3 farrow_top

An overview of the farrow_top module is seen in fig. 4.14. The filtering itself is made within the 8 instances of the farrow module, and the main function of the farrow_top module is to connect the outputs of these in the correct way. The outputs from the filters are summed in an adder tree of total 7 additions in 3 steps with registers in between to avoid too long chains of logic. The registers are data driven in the sense that they only are updated when acc_valid is set to high. That acc_valid is set to high indicates that the input from the accumulator stage is correct and that one cycle of the filtering should be executed. It should be noted that if the decimation factor R is greater than one, filtering will not occur in every clock cycle. For larger decimation factors the filtering operation will be executed less frequent. This part of the module is thus working on the decimated sample rate instead of, as the steps before, the input sample rate. But since the worst case is when the decimation factor is 1, the implementation must be designed to be able to handle this case.

The set_valid_out module indicates when the output of the farrow_top is valid. The two submodules are described in more detail in the following sections.

farrow_filter

One instance of the farrow_filter module performs the filtering through one transposed farrow filter. The 6 inputs to every branch is multiplied with the corresponding coefficients and then the sum over the branches at every level is calculated in two steps. This sum is then passed to the farrow_top module which combines the outputs from the different instances of the farrow filters to a valid total output. Just as in all components within the farrow_top the action is just performed when a valid input is present, indicated by acc_valid.

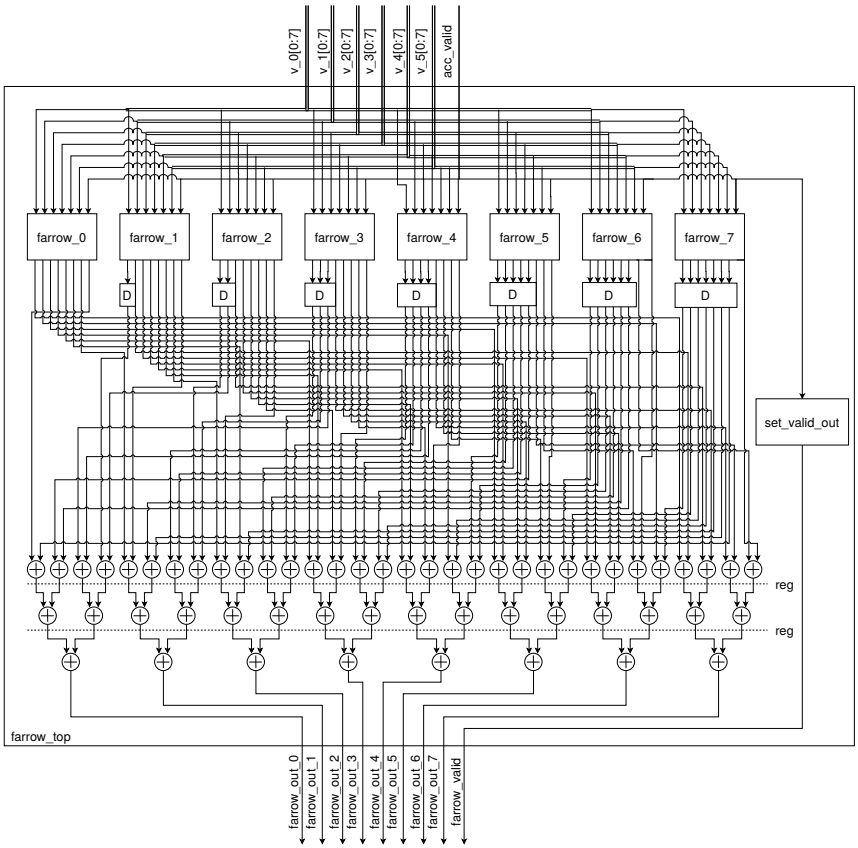


Figure 4.14: Block schematic of the farrow_top module, showing it's submodules and their connections.

set_valid_out

The function of the set_valid_out module is to indicate when the output of the farrow_top module is valid. The output is valid every clock cycle after the input is valid, after some initial latency of 6 valid input samples.

4.3.4 scale_data

As briefly mentioned in section 2.4 and further described in description of the integrate_dump module, the transposed farrow structure does not conserve the energy of the signal. The amplitude of the output signal is R times larger than the amplitude of the input signal. To achieve an energy conservation the signal must be scaled. This is accomplished by the scale_data module in combination with the shifting in the integrate and dump stage. The shifting in the integrate_dump stage performs a division by a factor D, according to eq. 4.1. The division by D in combination with multiplication by the scale factor, S, should result in a division by R. This is achieved by choosing the scale factor according to eq. 4.2

$$\frac{1}{R} = \frac{S}{D} = \frac{S}{2^{\lceil log2(R) \rceil}} \Rightarrow S = \frac{2^{\lceil log2(R) \rceil}}{R}, \in [1, 2) \tag{4.2}$$

The function of the scale_data module, seen in fig. 4.15, is simply implemented by 8 parallel multiplications by the scale factor. The scale factor is in the current version calculated outside the FPGA.

4.3.5 halfband_top

The halfband_top module implements the function of the halfband. In this module the second stage decimation, from R to 2R, is performed. The inputs to the module are the 8 scaled data paths from the scale_data module and a valid signal. Since the signal is decimated by two within this stage the output sample rate is just half the input sample rate and thus only 4 output data paths are needed.

The design corresponds to a fir halfband filter that is polyphase decomposed to reduce the complexity and the speed requirements of the implementation. One of the polyphase branches corresponds to the filtering part of the decomposition and the other by a delay implemented using a FIFO. The fir halfband branch corresponds to the non zero taps (center tap excluded) of the halfband filter and the delay branch corresponds to all the zeros and the center tap. These two branches should later be added, after the delay branch is divided by 2 (corresponds to the center tap of the filter) and delayed by the latency (19 clock cycles) of the fir halfband implementation plus 3 clock cycles which corresponds to 12 zero taps before the center tap of the halfband filter.

Fir halfband

The fir halfband module is generated using Xilinx Fir Compiler [12]. The instance has four parallel data input and four parallel outputs. In addition to the data paths the instance has scaled_valid as input to only make it active when the input data

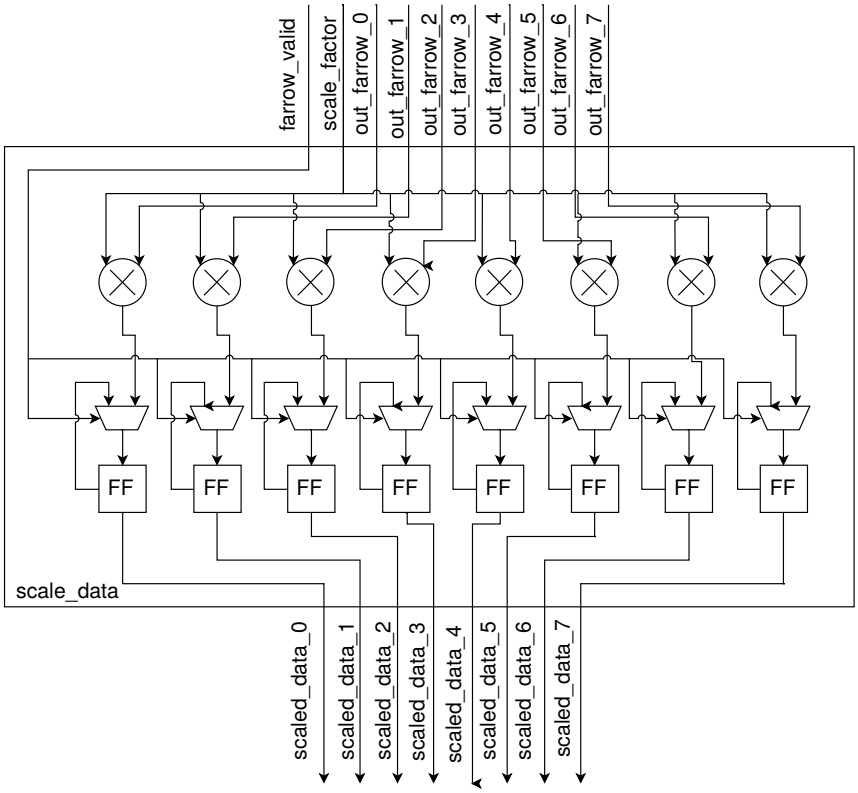


Figure 4.15: Block schematic of the scale_data module.

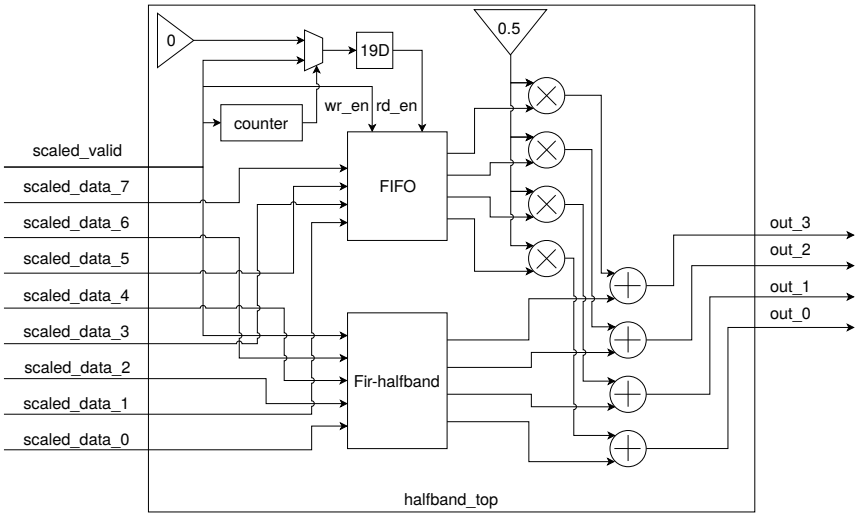


Figure 4.16: Block schematic of the halfband_top module, showing it's submodule and their connections.

is valid. The coefficients for the fir filter are the non zero ones presented in table 4.6.

Counter

In fig. 4.16 a module called counter is seen. The name is a bit misleading but the task of this module is to generate a control signal for a mux, which decides if the write enable signal of the FIFO should be 0 or data_valid. The write enable port should be tied to the data_valid signal after the three first data_valids have occurred. This is the case since the 3 first outputs of the fir halfband module should not be summed with the delayed signal, which is delayed 3 additional clock cycles. The implementation is simply a counter which activates a valid signal after 3 data_valids.

FIFO

The fifo-memory is generated using Xilinx Fifo generator [14]. The memory is configured to be a shift register with common clock for read and write. The data width is 96 bits (4 times one input sample to the halfband filter) and the depth of the fifo is 32. The task of the fifo is to match the delay branch with the latency of the fir halfband filter after the initial 3 data_valids. This is achieved by data_valid as both the read- and write enable signal but the read enable is delayed by the latency of the fir halfband implementation. The maximum number of samples that should be stored in the fifo are 23, the smallest two factor number greater than 23 is 32. The depth of the fifo is because of this set to 32.

4.3.6 Performance measure

To verify the function of the implementation, the results from the implementation are compared to the results from the floating point python model. To do the comparison, all filter coefficients, the input signal as well as the decimation factor must be equal between both the models. This is achieved by rounding these floating points signals to their fixed point equivalent and using these for both the floating point model and the implementation. Some rounding and truncation in the implementation will make the result differ a bit but the error should not be larger than the value of half the lsb, in this case 2^{-15}.

The verification was made using a few different decimation factors, seen in table 4.7. Since all values cannot be exactly expressed using the binary representation some of the decimation factors are rounded to the closest possible decimation factor.

Three slightly different implementations were used during the verification, one where truncation was used throughout the whole design where necessary and a second one using truncation everywhere but in the very last step, rounding was used instead. The results, seen in table 4.8, of these two models are slightly different, where in general the version using rounding performed better, which was as expected. A third version was also verified where rounding replaced all occurrences of truncation. This model produced almost exactly the same result as the model, only using rounding in the last step. The reason for the this is

R	Rounded R
2.0	2.0
3.45	3.4499847
4.0	4.0
50	49.98932113
107.89	107.96705107
1035.02	1040.25396825

Table 4.7: Decimation factors used for the test cases.

probably that the last truncation/rounding reduces the number of bits a lot more than any of the others. Simply, the rounding in the previous steps did not result in changing any of the upper bits in the final output. The final output consists, in both cases, of 16 bits and out of those 15 are fractional.

In table 4.9, the max errors between the floating point model and the different implementations are shown. It can be seen that the max error in the truncation case is double the error in the rounding cases

Dec.	Truncation model		Rounding last		Rounding all	
Factor	SNR (dB)	ENOB	SNR (dB)	ENOB	SNR (dB)	ENOB
2.0	92.41	15.06	92.24	15.03	92.25	15.03
3.45	90.23	14.7	90.07	14.67	90.06	14.67
4.0	92.81	15.13	92.46	15.07	92.46	15.07
49.99	92.61	15.09	92.75	15.11	92.75	15.11
107.97	92.65	15.1	92.77	15.12	92.77	15.12
1040.25	92.69	15.11	92.79	15.12	92.79	15.12

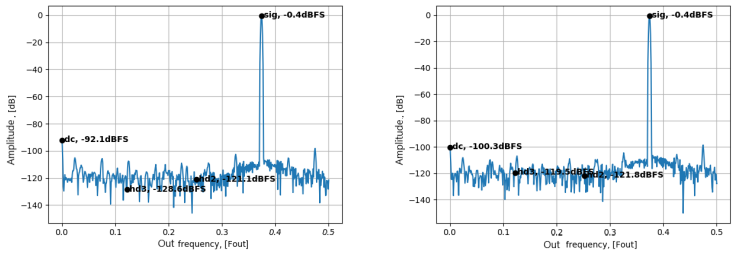
Table 4.8: Resulting SNR and ENOB of the verified implementations.

Dec.	Truncation model		Rounding last		Rounding all	
Factor	mean err.	max err.	mean err.	max err.	mean err.	max err.
2.0	-1.53e-05	3.05e-05	-7.51e-08	1.53e-05	-1.52e-08	1.52e-05
3.45	-1.50e-05	3.05e-05	-1.69e-07	1.53e-05	-1.09e-07	1.52e-05
4.0	-1.53e-05	3.05e-05	-3.16e-08	1.53e-05	2.83e-08	1.53e-05
49.99	-1.53e-05	3.06e-05	-1.98e-08	1.53e-05	-1.98e-08	1.53e-05
107.97	-1.53e-05	3.06e-05	-3.29e-07	1.53e-05	-2.69e-07	1.53e-05
1040.25	-1.52e-05	3.05e-05	-3.14e-08	1.52e-05	-6.17e-08	1.53e-05

Table 4.9: Resulting mean and max error of the implementations.

It is easily seen that the results, from table 4.8, are not much different between the two models. The main difference is that the truncation introduces an small offset error, this is easily seen when comparing the FFTs of the output signals of

the two models. As seen in fig. 4.17, the DC component is significantly lower in the Rounding case compared to the truncation case.



(a) **(b)**

Figure 4.17: (a) Truncation, (b) Rounding.

4.3.7 Resource usage

This section will present the FPGA resource usage of the implementation. Firstly, the implementation has been aimed towards to function as a proof of concept for the algorithm and is in no way optimally implemented. The only resource that have been put in some effort to not overuse is the DSP-elements. The resource usage of the implementation is seen in table 4.10. As seen from the resource usage details, the implementation consumes quite a lot of resources, at this rather large FPGA chip. Through optimization the resource usage should quite easily be reduced to some extent, but the design will never be really small. By only considering the amount of DSP-elements used, it can be seen that the use is minimized and that the DSP-elements are only used where intended.

Module:	acc_top	farrow_top	halfband	scale_d.	Total
DSP48E2	40 (1.5)	184 (6.7)	48 (1.7)	8 (0.3)	280 (10.1)
CLB LUT	13165 (4.9)	18344 (5.5)	806 (0.2)	2 (0.0)	32310 (9.7)
CLB Reg.	12689 (1.9)	43980 (6.6)	2977 (0.5)	194 (0.0)	59840 (9.0)
Carry 8	209 (0.5)	3448 (8.3)	24 (0.1)	0 (0.0)	3681 (8.9)
F7 Muxes	1296 (0.8)	0 (0.0)	0 (0.0)	0 (0.0)	1296 (0.8)
LUT Logic	11935 (3.6)	16808 (5.1)	180 (0.1)	2 (0.0)	28918 (8.7)
LUT Mem.	1230 (0.8)	1536 (1.1)	626 (0.4)	0 (0.0)	3392 (2.3)
LUT Pairs	7275 (2.2)	16465 (5.0)	71 (0.0)	1 (0.0)	23902 (7.2)

Table 4.10: Resource usage of the FPGA implementation.

As described earlier, the whole design does not work at the same actual speed which potentially can be used to reduce the resource usage. The most resources, at least when it comes to DSP-elements are used in the filters and both of the filter structures are on the slower side of the design. The current implementation is designed for the worst case scenario, which is decimation by 2. If the minimal

decimation factor are increased the level of parallelization can be decreased. As an example if the minimal decimation factor were to be 16, the level of parallelization in all steps after the accumulators could be reduced to 1.

Chapter 5
Conclusion

In this section a summary of the findings of the master thesis will be given as well as some suggestions for future work on the same topic.

5.1 Summary

The result of this master thesis work is an algorithm for arbitrary decimation for high throughput applications. The suggested algorithm consists of a transposed farrow structure in series with a halfband filter. Together, the two steps can decimate from a factor 2 and upwards. The transposed farrow structure works as a variable decimation filter and can perform decimation from a factor 1 (no decimation) and upwards. In the suggested design the farrow filter consists of 6 branches with transposed fir filters of order 7. This gives 48 coefficients in total, but only 24 of those are unique. The halfband filter works as a fixed decimator by a factor 2 (halves the sample rate) and consist of 49 taps. Out of these taps, 25 are non zero and out of this subset 13 are unique. The reason for having both the filters working in series is that the transposed farrow structure alone, couldn't fulfill the attenuation requirements of the project without using an unreasonable high number of coefficients.

The algorithm is also implemented onto a Xilinx Virtex Ultrascale FPGA. The implementation of the algorithm works as a proof of concept rather than an optimal implementation. To achieve a high throughput the algorithm is implemented in 8-parallel, but in a way that should be quite easy to extend to higher levels of parallelization. The parallelization together with a clock speed of 312.5 MHz gives a total throughput of the input signal of 2.5 GSa/s, corresponding to maximum 1.25 GSa/s at the output. The implementation is also verified to produce results that corresponds to the result of the floating point model simulations for some examples that is intended to test various functions of the implementation. The optimization of the implementation is very limited, just a few things have been made. Firstly, the symmetry of both the filters are utilized to reduce the number of DSP-elements used and the halfband filter is also implemented in it's polyphase version to reduce the resource usage.

5.2 Future Work

In this section different areas that are not very well investigated in this thesis are presented and improvement within these areas are suggested to be investigated in the future.

5.2.1 Arbitrariness of the decimation

The aim of this thesis work was to suggest an algorithm for arbitrary decimation. The level of arbitrariness is thus limited by some factors, which of the most important is the resolution of the inverse of 2/R, or the resolution of μ. Throughout the work different ways to improve these limits have been thought about. The first and most easy method is to just increase the numbers of bits where necessary, and this would increase the arbitrariness to some extent. Another suggestion is to use some other method to represent for example fractional numbers like 1/3 or to swap between slightly different decimation factors to get the wanted one as the mean factor. Another method to increase the arbitrariness would be to put another decimator in front of the suggested one, that for example can decimate by factors of 2^x. It would be interesting to further investigate this area of possible improvement.

5.2.2 Optimization

The implementation, suggested in this master thesis, works as a proof of concept but the design is in no way optimal. To reduce the resource usage of the implementation optimization is needed. Optimization can further be used already in the algorithm design phase to for example reduce the number of unique coefficients in the transposed farrow structure. In the implementation part optimization can be made in several ways, both regarding the resource usage and the performance. The resource usage can be minimized using existing optimization methods to try to minimize the use of for example DSP-blocks and registers. The implementation in this thesis work was made for a fixed clock speed of 312.5 MHz and the design was adopted to this demand. If the clock speed could be higher, or maybe lower, the performance can be optimized in different ways. A lower clock speed could lead to less pipelining which would lower the resource usage. A higher clock speed would probably require some more pipelining but instead the level of parallelization might be reduced.

5.2.3 Alternatives to the halfband filter

The need for a fixed decimator in series with the transposed farrow structure was obvious to not get a very large number of coefficients. But the choice of the halfband filter are not very well investigated. A halfband filter is probably the most common way of achieving a decimation by 2 and is a easily implementable. In literature other, more complex methods, that aims towards reducing the resource usage exists. The use of another, less common, method might result in a less resource demanding implementation and this is an area where improvements might

be made. With that said, the halfband filter occupies quite few resources in comparison with the transposed farrow structure.

5.2.4 Connection of the implementation

This thesis work has been focused on the algorithm itself and the implementation of it. No efforts have been made to put the algorithm or the implementation into context. The current function of the algorithm is to calculate the new sample values and not to give a signal that works at the output sample rate. Depending on the application it might be interesting to investigate how to actually recreate the output signal, using also the correct time axis and not only the correct sample values. Investigation of which connections to use to connect the algorithm to other algorithms running at the same FPGA or interfaces to use for connection with other components outside the FPGA could be interesting to look deeper into.

References

[1] J.G. Proakis, D.K. Manolakis, *Digital Signal Processing 4e*, Pearson 2014, ISBN: 1-292-02573-5

[2] L. Wanhammar, H. Johansson, *Digital Filters*, Department of Electrical Engineering Linköping University 2002

[3] C. W. Farrow, *A continuously variable digital delay element*, IEEE International Symposium on Circuits and Systems, Espoo, Finland, 1988, pp. 2641-2645 vol.3. doi: 10.1109/ISCAS.1988.15483

[4] T. Ramstad, *Digital methods for conversion between arbitrary sampling frequencies*, IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 32, no. 3, pp. 577-591, June 1984. doi: 10.1109/TASSP.1984.1164362

[5] J. Vesma, *Optimization and application of polynomial-based interpolation filters*, Doctoral Thesis, Tampere University of Technology, Publications 254, 1999.

[6] J. Vesma and T. Saramaki, *Interpolation filters with arbitrary frequency response for all-digital receivers*, 1996 IEEE International Symposium on Circuits and Systems. Circuits and Systems Connecting the World. ISCAS 96, Atlanta, GA, USA, 1996, pp. 568-571 vol.2. doi: 10.1109/ISCAS.1996.541788

[7] D. Babic, J. Vesma, T. Saramäki, V. Lehtinen, M. Renfors, *Polynomial-based interpolation filters for DSP applications, Design, Implementations, and applications*, Tampere University of Technology, Finland, `http://www.cs.tut.fi/kurssit/TLT-5806/Interpol.pdf`.

[8] D. Babic, J. Vesma, T. Saramaki and M. Renfors, *Implementation of the transposed Farrow structure*, 2002 IEEE International Symposium on Circuits and Systems. Proceedings (Cat. No.02CH37353), Phoenix-Scottsdale, AZ, USA, 2002, pp. IV-IV. doi: 10.1109/ISCAS.2002.1010374

[9] D. Babic, T. Saramäki, and M. Renfors, *Sampling rate conversion between arbitrary sampling rates using polynomail-based interpolation filter*, The Second International Workshop on Spectral Methods and Multirate Signal Processing SMMSP 2002, Toulouse, France, September 2002, pp. 57-64.

[10] R. G. Lyons, *Understanding Digital Signal Processing 3e*, Pearson Education International, USA, 2011. ISBN: 0-13-211937-4.

[11] *UltraScale Architecture DSP Slice, User Guide v 1.8*, Xilinx, May 14 2019. https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf.

[12] *FIR Compiler v7.2, LogiCORE IP Product Guide*, Xilinx, November 18 2015. https://www.xilinx.com/support/documentation/ip_documentation/fir_compiler/v7_2/pg149-fir-compiler.pdf

[13] D. Gisselquist *Rounding Numbers without Adding a Bias*, Gisselquist Technology LLC, viewed June 5 2019, published July 7 2017. https://zipcpu.com/dsp/2017/07/22/rounding.html

[14] *FIFO Generator v13.1, LogiCORE IP Product Guide*, Xilinx, April 5 2017. https://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v13_1/pg057-fifo-generator.pdf

[15] *Scipy remez, User Guide*. Viewed June 5 2019, Scipy version 1.2.1. https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.remez.html

Printed by Tryckeriet i E-huset, Lund 2019

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2019-719
http://www.eit.lth.se