Low-Energy FIR Filter Realisations on Hardware and Software Programmable Platforms

A Project in Applied Signal Processing and Implementation

Autumn 2011/Spring 2012

Master Thesis

Conducted by group 1041

Stine Martine Gullaksen

Morten Danmark Nielsen

AALBORG UNIVERSITY

Department of Electronic Systems
Fredrik Bajers Vej 7
9220 Aalborg Ø
Tlf. 99 40 86 00
http://www.sict.aau.dk

Title:
Low-Energy FIR Filter Realisations
on Hardware and Software
Programmable Platforms

Project period:
3. and 4. semester of the ASPI master program
Autumn 2011/Spring 2012

Project group:
Group 1041

Group members:
Stine Martine Gullaksen
Morten Danmark Nielsen

Supervisor:
Peter Koch

Copies: 4

Pages: 178

Attachments: 1 x CD-ROM

Finished: 31-05-2011

Abstract:

Low-energy filters are needed in our everyday life and even further optimisation of these are needed. This project explores the question on how energy dissipation can be minimised for digital FIR filters on software and hardware programmable platforms.

The answer to this is found by analysing, implementing and evaluating several methods for this purpose. Realisations of the methods were utilised, as to make a practical comparison between the optimisation methods.

Two different software approaches was conducted: multirate filtering and Hamming distance optimisation. The multirate filter gave an optimisation, in relation to the reference point, of up to 20 % and the Hamming distance optimisation method gave up to 2 %. Three different multiplierless approaches were applied on the hardware programmable platform, in addition to two reference filters. Of these realisations the shift based filter gave the best performance. Next to this a parallel direct form filter, and third a multirate filter. The least energy efficient methods was a generic direct form filter and a filter utilising distributed arithmetic.

The evaluation of the realisation showed that several methods could be used as energy efficient solutions compared to a generic reference point. Based on the methods investigated in this project a set of guidelines are conducted. These are aimed at designers with the objective of designing low-energy filters.

Preface

This project report documents the work done in the master thesis conducted by group 1041 on the 3. and 4. semester of the Applied Signal Processing and Implementation (ASPI) master program at Aalborg University. The project period has been the fall semester of 2011 and spring semester of 2012. The project period has been twice the normal length, as the project has been conducted as a long master.

As a result of this project a set of guidelines is conducted. These can be of use for engineers/engineering students in the situation of designing low-energy filters. The guidelines are intended to be read as a prolongation of the report and can be found in chapter 20 on page 139.

The report consist of five parts: Introduction, Software Optimisation Methods, Hardware Optimisation Methods, Conclusion and Appendix. The report is further divided into chapters, sections and subsection. Citations will be done in the Harvard style, which gives a source reference as: [authors lastname/company, year], and will in some cases be noted by a page reference. A full list of bibliography can be found on page 153. Complete lists of abbreviations, figures, tables and code listings can be found in the following pages.

The CD attached in the back of this report, contains the source code of the Python, Assembly and VHDL code developed in the project. The project has used two IDE's for development of systems for the software and hardware platform. For the software platform Microchip's MPLAB® X IDE has been used. For the hardware platform Altera's Quartus® II and ModelSim® has been used. A copy of this report is also enclosed on the CD.

The report is composed by:

Stine Martine Gullaksen

Morten Danmark Nielsen

Abbreviations

ASIC	Application Specific Integrated Circuit
AST	Adjacent Signal Toggles
BSE	Binary Subexpression Elimination
CMOS	Complementary Metal-Oxide-Semiconductor
COP	Combinatorial Optimisation Problem
CSAC	Common Subexpression Across Coefficients
CSD	Canonical Signed Digit
CSE	Common Subexpression Elimination
CSWC	Common Subexpression Within a Coefficient
DA	Distributed Arithmetic
DFG	Data-Flow Graph
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Array
GA	Genetic Algorithms
HD	Hamming Distance
HDL	Hardware Description Language
LAB	Logic Array Block
LE	Logic Element
LFSR	Linear Feedback Shift Register
LO	Logic Operator
LSB	Least Significant Bit
LUT	Look-Up-Table
MAC	Multiply-Accumulate
PG	Precedence Graph
PLL	Phase-Locked Loop
PRNG	Pseudo Random Number Generator
RTL	Register Transfer Level
SA	Simulated Annealing
SNR	Signal-to-Noise Ratio

Contents

List of Figures

List of Tables

Listings

Part I

Introduction

1 Introduction

This chapter will make a general introduction to the project and describe the motivation for this. The following sections will describe the parameters of power dissipation and the terms power and energy are seen in context. It also gives a short introduction to optimisation levels, platforms and the outset of this project.

Mobility is a key feature in nowadays technology. This is clearly seen as the amount of sold portable devices grew over 60 percent from 2010 to 2011*. Mobility introduces new problems for designers and developers as battery lifetime and an ever increasing need for computer performance has to go hand in hand. The need for energy efficient devices are as relevant as ever before.

The popular interpretation of Moore's law is that transistor density doubles every other year. In contrast; battery manufactures use 5 to 10 years to achieve the same increase in energy density. Since an increase in computing performance is most likely to cause an increase in energy consumption, methods to optimise on energy, while maintaining the performance has to be found. This in such a way that the end consumer will not feel frustration about battery lifetime issues.

Further development on Moore's law is the so called Koomey's law [Koomey et al., 2011]. Koomey's law introduces the energy consumption aspect, to show another very important trend in technology evolution. Koomey's law predicts that the number of computations per kilowatt-hour doubles every 1.5 year. This is illustrated on figure 1.1.

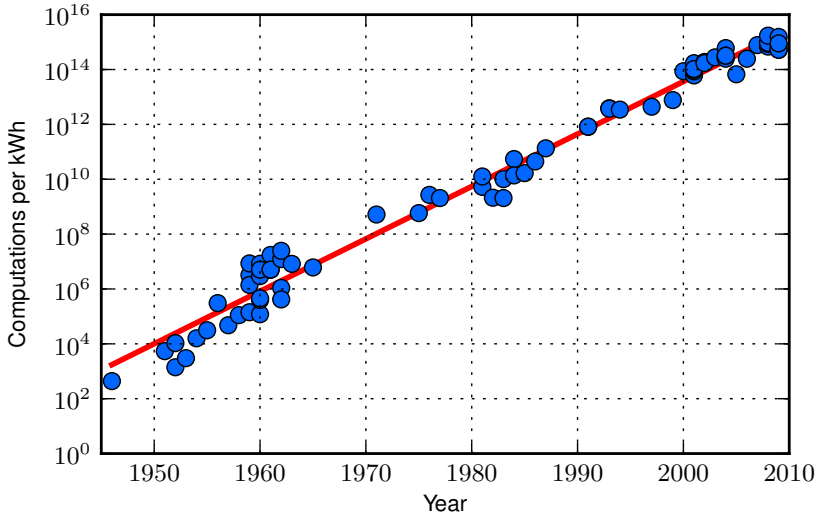

Figure 1.1: The trend known as Koomey's law [Koomey et al., 2011].

From the figure, it is seen that Koomey's law builds on more than 60 years of technology evolution. From the first house-sized computers using vacuum tubes to nowadays portable devices using nano-scaled transistors. The reason why this is possible is among other things an extensive effort in developing methods for optimising energy consumption. This means developing low-energy features on either hardware or software parts of computer technology.

*IDC Press Release, 6. february 2012, `http://www.idc.com/getdoc.jsp?containerId=prUS23299912`

Research has brought forward advanced methods for reducing the energy dissipation in embedded electronics, but methods and research that aims at further reduction of energy dissipation are still needed. By exploring numerous optimisation techniques, this project will show how these can be used to gain a lower energy consumption. This by using practical implementations which will back up the theoretical basis and give a concrete measure of efficiency.

1.1 Power dissipation in CMOS

In this section sources of power dissipation in CMOS are presented. The CMOS (Complementary Metal-Oxide-Semiconductor) technology, which is used in most integrated circuits, has two main components of dissipation, static and dynamic. The total power dissipation is expressed as:

$$P_{Total} = P_{Static} + P_{Dynamic} \tag{1.1}$$

The static dissipation comes mainly from what is called sub-threshold leakage current. This leakage current is the current that flows between the source and the drain of a transistor when it is in the sub-threshold region, which is when it is off. The leakage current depends on the size of the transistor and as this decreases, the size of this current can become as big as 50 % of the total power dissipation of the transistor [Narendra et al., 2004].

The dynamic dissipation comes from two sources of dissipation, switching and short-circuit. The switching dissipation occurs as gates and wires in the integrated circuit in practice function as capacitances. As a logic gate is flipped from low to high and from high to low, a capacitance, C, is charged from V_{DD} and discharged to ground. This is illustrated on figure 1.2.

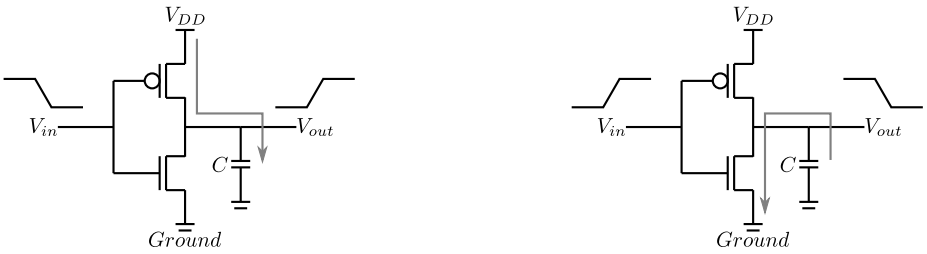

Figure 1.2: Example of power dissipation due to switching.

On figure 1.2 a CMOS inverter is seen. As V_{in} switches to low, energy is drawn from the supply, V_{DD}. Half the energy is stored in the capacitor and the rest is dissipated in the P-transistor. As the capacitor is charged the energy $E = \frac{1}{2} \cdot C \cdot V_{DD}^2$ is stored in the capacitor. As V_{in} switches to high, the energy from the capacitor is discharged to ground trough the N-transistor. This gives a dissipation of energy in the size of:

$$E = \frac{1}{2} \cdot C \cdot V_{DD}^2 \tag{1.2}$$

and the power dissipation can therefore be expressed as:

$$P_{Switching} = E \cdot F_{Clock} \cdot \alpha \tag{1.3}$$

$$= \frac{1}{2} \cdot C \cdot V_{DD}^2 \cdot F_{Clock} \cdot \alpha \tag{1.4}$$

where F_{Clock} is the clock frequency of the circuit and α is the activity factor. An activity factor is introduced as most gates do not switch every clock cycle. The activity factor of a gate is therefore the ratio between the gate switching and the clock frequency. As the clock in a system switches every cycle it has an activity factor of 1.

The short-circuit dissipation is introduced as every transition between high and low has a rise or fall time. As CMOS technology consist of a stacked pair of N and P transistors, a transition means that at a point both transistors will be in a transition mode, thereby leading a current directly from V_{DD} to ground.

The expression of the total power dissipation can therefore be expanded as:

$$P_{Total} = P_{Static} + P_{Dynamic} \tag{1.5}$$

$$= I_{Leakage} \cdot V_{DD} + \frac{1}{2} \cdot C \cdot V_{DD}^2 \cdot F_{Clock} \cdot \alpha + I_{Short-Circuit} \cdot V_{DD} \tag{1.6}$$

Although the power dissipation due to leakage contribution has become significant as the technology has scaled, this has been considered and accepted as one of the problems or drawbacks of using this technology. Despite this fact of increased significance of leakage power, the switching power is still considered as the factor where the designer has the most influence of the total power dissipation. Power optimisation is done by reducing one or more of the factors in eq. (1.6), while still maintaining the specifications of the application.

1.2 Power vs. energy

Energy and power are two terms that are closely related, their connection will be described in this section. Sources of power dissipation in CMOS are described in the previous section and how this related to energy consumption is important to know for the further understanding of this report.

To start with, the term energy has to be introduces in connection with power. Power is defined as the rate at which energy is consumed, therefore the energy consumption is given by:

$$E = P \cdot T \tag{1.7}$$

where power can be formulated as:

$$P = I \cdot V = \frac{V^2}{R} \tag{1.8}$$

The time, T, can be expressed as:

$$T = N \cdot \tau \tag{1.9}$$

where N is the number of clock cycles, a program, instruction or architecture takes to execute, and τ is the clock period. The execution time depends on the clock frequency and the number clock cycles the filter takes to execute. For a program implemented in software, for instance on a microcontroller, the instruction set of the current microcontroller usually gives information about how many clock cycles each instruction takes. For implementation in hardware, on a FPGA, a simulation of the filter can give information about the number of cycles needed to complete the program.

All the above equations formulate the final expression:

$$E = \frac{V^2}{R} \cdot N \cdot \tau \tag{1.10}$$

As different filter implementations may use a different amount of clock cycles or run with a different clock period, energy has to be the comparison base. This assures that when comparing different filter implementations, the energy will be a measure of the power dissipated for a filter computing a single output from a single input. The use of energy, will in other words, make the different filter implementations comparable independent of the number of cycles and throughput of the filter.

Most of the literature used in this project, use power as a measure for evaluating the low-energy methods. This is not seen as the optimal way of making proper analysis of the optimisation methods. The reason for this, is that the reader does not have anything to relate the power dissipation to. By using energy in stead of power, the reader can at least relate to the fact, that this is the energy dissipated by one filter computation. If this again is seen in perspective to other optimisation methods, the energy measure will make a solid foundation for comparing these.

If a comparison is performed in power, and the optimisation methods introduce different amount of clock cycles, then it will not be possible to make a correct assessment.

As this project will contain a number of optimisation methods which will be implemented and compared, the term energy is preferred. It is not possible to know the number of cycles of the implementations in advance, which is why energy is a more suitable measure than power. Due to this, all results will be shown in terms of energy.

1.3 Optimisation levels

For the reason of optimising an electrical system in terms of energy, or any other sort of optimisation, the abstraction levels are considered. The different levels are addressed in different ways in order to optimise the overall system. The level at which optimisation is conducted can in general be one of the following:

- System level
- Algorithm level
- Architecture level (Register transfer level)
- Circuit level (Functional unit level)
- Gate level
- Transistor level

The list is ordered by abstraction level and starts with the system level as the highest. In the system level the designer can typically choose whether the system is on or off in order to optimise the energy consumption. The next level is the algorithm level, where the designer has the possibility to design an energy aware algorithm. This might be done by reducing the computational complexity of the algorithm. An algorithm with parallel operations are also a possibility in order to meet any energy constraints, as this uses less cycles to complete than a sequential algorithm.

The architectural level gives the designer great possibilities for optimising with concern on energy. On the architecture level which is also called register transfer level (RTL), the energy can be optimised by e.g. introducing parallel calculations and lowering the size of the buses needed. Many techniques exists for power optimising the architecture by reducing the switching capacitance, such as the choice of number representation and exploration of signal correlation [Chandrakasan and Brodersen, 1995]. In some cases the algorithmic level and the architecture level might be seen in context to each other, as the designer can design an algorithm which exploits certain architectural structures.

On the circuit level it is the functional units that are optimised. All functional units are build up by logic gates which also can be optimised. On the lowest level the transistors can be optimised in order to use as less power as possible.

The trend in later years is that optimisation is conducted on the three highest abstraction level. As the transistor size is decreasing to nano scale and power optimal gates and functional units are already discovered, it is left to the designer to optimise electrical systems on the higher abstraction levels.

1.4 Optimisation platform

Choosing a platform when working with digital signal processing either software programmable or hardware programmable can be chosen. Both having different pros and cons depending on the specific application and specifications such as power or time constraints.

When working with signal processing, two major platforms of integrated circuits comes to mind, the digital signal processor (DSP) and the field-programmable gate array (FPGA). The DSP is used in many signal processing applications e.g. Fourier transforms and filters, as one of the key features is the multiply-accumulate (MAC) function,

which is used greatly in these applications. The DSP has a fixed hardware architecture and it is therefore only possible to change the software. The FPGA is a newer technology which has had a great deal of interest in the last 10 years, within the field of digital signal processing. This growing interest comes with the ongoing growing interest in parallel processing and optimisation on the architectural level, which can be explored as the FPGA has programmable hardware. The FPGA can be used as a way of testing the design of application specific integrated circuits (ASIC) before going into production, as the FPGA is programmable in both software and hardware where the ASIC is not. The FPGA is programmed using hardware description language (HDL), which comes in different variations such as VHDL and Verilog.

The ASIC is many times chosen in commercial products when it is not necessary to reprogram the unit in any means. In this way the company can optimise the product all the way from a system level and down to transistor level in order to get the most efficient and cheapest product possible.

This project will turn to use both DSP and FPGA technology, as energy consumption can be minimised on both fixed and programmable hardware. This means on either the algorithmic level, e.g. minimising the number of arithmetical functions and data management, or on the architectural level, e.g. making multiplier-less circuits. The two technologies will be used in the project to explore optimisation techniques in software on the DSP and in hardware on the FPGA. This is also connected to the previous mentioned abstraction levels which are commonly focused on the algorithm and architecture levels.

1.5 Project outset

The arguments presented in previous sections of this chapter forms the motivation of why low-energy design is important and how energy consumption can be reduced. This section elaborate on the outset of the project, by briefly looking into one paper on the topic of low-power design. Despite that this article aims at low-power design, this relates directly to low-energy as discussed in section 1.2.

This project will take an outset in already known methods for designing low-power digital filters. One main article within this field is [Mehendale et al., 1998], an article proposing seven methods for designing low-power FIR filter for use on programmable DSP's. In this article the authors start out with a general presentation of FIR filters, sources of power dissipation and previous work made on the topic.

Regarding the sources of power dissipation they mention the hardware components that experience the most signal activity. As the multiply-accumulate is a main component in a FIR filter, this and the data and address buses to it experience the most signal activity. They also comment on the way of updating the input signals for the filter computations. Storing the input data in a circular buffer can reduce the power dissipation compared to actual moving the data. In a circular buffer only the pointer to the current data needs to be changed. Further on they discuss switching activity in the buses as a significant source of power dissipation.

The switching activity is one of the main issues they try to reduce the effects of in the methods they present. Some of the methods aims at reducing the switching activity in the data and address buses, while other in the multiplier. The article also includes a method which focuses on reducing computation complexity of the FIR filter, by implementing a multirate structure instead of the traditional direct form realisation.

A general perception of this article is that it presents methods for low-power filter design. It does however not present the results of actual power measurements of all the methods presented. Only power measurements of the multirate filters are presented. The methods focusing on signal switching activity are not evaluated by an actual implementation. Actual measurements are indeed necessary in order to evaluate and compare the effects of the methods.

The methods presented in [Mehendale et al., 1998] are suggestions of methods that could be looked more into in this project. In the paper, DSP's are the only intended implementation platform. As mentioned previously, FPGA's are a popular platform for signal processing applications, and methods for designing low-energy digital filters on such will also be investigated. As both platforms are interesting seen in context of signal processing, methods for both platforms will be looked into. As one platform is software programmable and the other is hardware programmable, the same methods might not be used on both platforms. This however depends on the methods, and can not be concluded at the moment.

1.6 Design strategy

In this section the design methodology used in the project is presented. The design methodology takes an offset from the A^3 model. This section presents the model in general and describes in more detail how it will be applied in this project.

The A^3 model is a visualisation of the development domains which the designer has to go through, to come from an application specification to a finished product. The A^3 name, comes from the fact that it divides the development into three domains: application, algorithm and architecture. The model does not specify the development process, but the designer can use it to visualise the development process. The three domains in the model, will shortly be described here.

Application: This is the starting point of most projects. At this stage the specifications and attributes of the application are described, which normally sets the basis for a project. A specification requirement is usually the final product in this domain.

Algorithm: The algorithmic domain specifies the possible methods available for implementing the given application. Usually several algorithms can satisfy the specification from the application domain. The methods are analysed and evaluated in order to find the most appropriate method for the application. This might be in terms of computational complexity, memory usage or robustness.

Architecture: The architectural domain is focused on the various implementations suitable for the chosen algorithm. Usually several different technologies can be used to implement the algorithm. These could be e.g. DSP, FPGA or microcontroller. The most suitable architecture has to be found and is in many cases finally evaluated in terms of the specification found in the application domain.

Within the A^3 some iterations between the domains are allowed. This might be done if the solution found in one domain makes it necessary to make changes in the previous domain or domains. The overall picture of the A^3 concept will also be used in this project in order to structure the work. However; some modifications are made to make it more suitable for the current project. As the objective of this project is not to find one final solution, but more to evaluate and compare several solutions, the A^3 concept has to be adjusted. The interpretation of the A^3 model used in this project is as follows:

Application: As this project does not have a specific application in mind, the application domain will describe the overall objective of the project. As the project is about design of low-energy filters, this is a very general description of the application. In addition, the application domain will also contain a general evaluation set and a reference point specification. The final implementations will be evaluated in terms of the evaluation set. As there are no specifications available, the evaluation set should include various filters that are representative for comparison. The reference point is specified, as to have a common starting point design. In this project, the reference point is an unoptimised direct form filter.

Algorithm: When going from the application domain to the algorithmic domain, several optimisation methods will be used as to generate a number of low-energy algorithms. In addition, the reference point will also be mapped into an algorithm, as to have a reference filter for comparison. The optimised filters are simulated in order to verify that they work in the same way as the reference filter. The main difference from the basic A^3 is that several of the algorithms will be chosen for implementation, instead of one.

Architecture: As of the architectural domain, this does not consist of every possible implementation platform. Only DSP and FPGA will be considered due to the previous discussion in section 1.4, as these two are the most popular for signal processing applications. The chosen algorithms will be implemented on one of the platforms or both. As the FPGA can be used to make any architecture, this is used to generate several implementations.

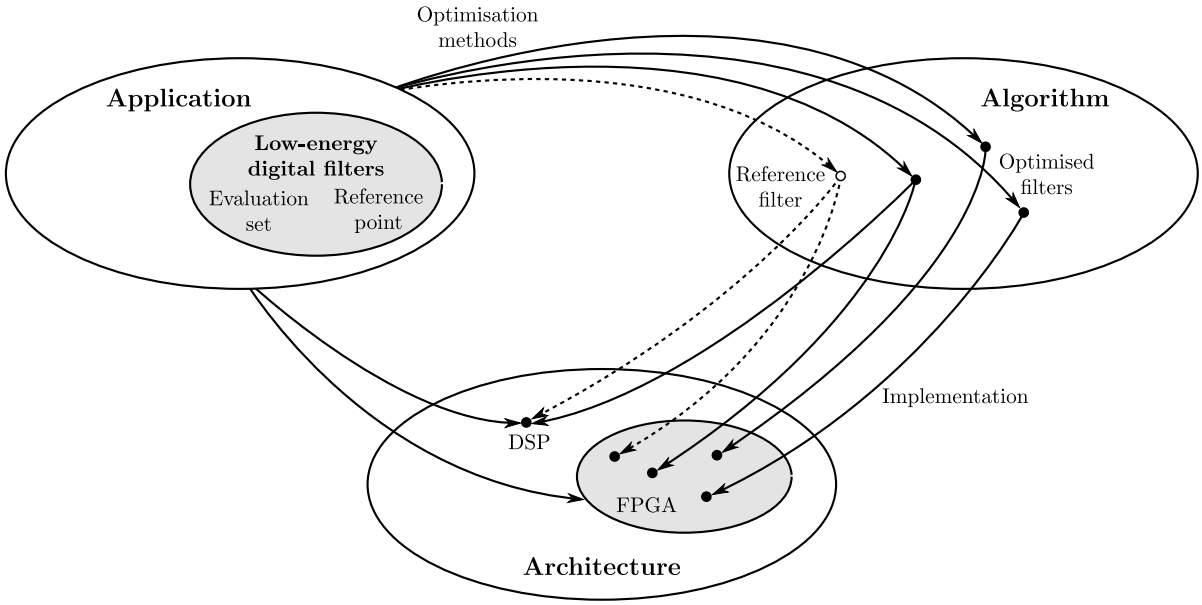

Figure 1.3: The modified A^3 concept used in this project.

The customised version of the A^3 leads to figure 1.3. In this figure it is seen that the application domain contains the main topic of the project. Under this topic, an evaluation set and reference point are specified. The algorithmic and architectural domains are the dominant domains of this project. In the algorithmic domain several filters are analysed and simulated. All of them are implemented, unless the simulation proves that the method is not suitable for low-energy design. Among the filter designs, some may be more appropriate for software implementation and some for hardware implementation. Some designs might be used on both implementation platforms, if this is possible. The reference filter will be implemented on both platforms, as this should be used for comparison of the used filter optimisation methods.

In figure 1.3 a connection from the application domain to the architecture domain is seen. This illustrates the evaluation of the implementation by using the evaluation set specified in the application domain. The different implementations are to be evaluated in terms of energy consumption compared to the reference filters.

When going from the algorithmic domain to the architectural domain it is important to do this systematic. A DSP has a fixed architecture, so in case of implementation on this, the implementation is based on the algorithms designed in the algorithmic domain. For implementations on a FPGA, which is hardware programmable, the implementation is also performed with an outset in the theory and algorithm describing the method, but where the DSP needs an algorithm written in C or assembly, the FPGA need a specification on how the architecture of the filter is to be assembled. Implementation on FPGA's are usually done using a hardware description languages e.g. VHDL or Verilog. When implementing a method onto hardware terms as scheduling and allocation has to be taken into account. These topics will not be discussed here, but considered when appropriate.

The project will try to make a practical view point on how to develop and evaluate digital filters. This meaning, that the found methods in the algorithmic domain will be analysed, simulated and implemented, as would be the approach in a real-life engineering task.

The modified A^3 concept is designed and chosen as design strategy due to the straightforward structure which is very intuitive for a designer in this situation. It clearly differs between the phases in a project and helps the designer to structure the whole process of developing an actual implementation.

2 Project Specification

This chapter will describe the general scope and specification of the project documented in this report. The scope will reflect on the content of the project and a problem statement will introduce overall questions, which will be answered throughout the report.

The purpose of the project is to investigate, implement and evaluate various methods in the context of designing low-energy digital filters. Digital filters are one of the most used techniques in digital consumer products today. Digital filters can in terms of energy dissipation in general be optimised in two domains, software or hardware. In means of software optimisation, this project will address the DSP, while for hardware optimisation the focus will be on the FPGA. Some of the methods might even be used to optimise both in hardware and in software.

The project has initially no particular application in mind, which means that no design specifications of the digital filters exist. Some specifications are however necessary to specify, as an evaluation of the different methods has to have a common reference point. The project will be limited to energy optimisation for FIR filters, as these are expected to be the most common filters in signal processing. Therefore; the reference point of the evaluation will be a reference pool consisting of FIR filters of various lengths. When choosing the reference point, it has to be representative for comparing the different methods. This means that the reference point is in its most general and basic form, making it suitable for comparing any optimisation method to it. A general evaluation set will be determined and used throughout the project, as the main focus in this project is *methods* for energy optimisation in general, not concerning any specific applications. The filter specifications will be chosen and presented in chapter 3. There will be conducted tests throughout the report in order to verify that the evaluation set is representative for comparison.

With no predefined design specifications available the main objective of the overall cost function is energy. Although the project is about energy optimisation, it is not assumed that other cost function parameters such as area and time are unlimited. The cost function parameters will be discussed where it is found necessary.

Both the reference design and the optimised designs will be simulated in order to clarify that the functionality of this is correct.

Procedures for measuring energy dissipation of both hardware and software implementations will be specified and analysed, in order to make a valid evaluation of the optimisation techniques. The evaluation will contain actual measurements of the energy consumption of the individual implementations.

The result of the project is going to be a set of guidelines which gives an overview of different energy optimisation methods. The guidelines will also include a basic description of how to implement the methods and a comparison of how they perform. The results of the optimisation in software and hardware will not be directly comparable, since they are optimised on different levels, but an overall evaluation will be made.

The above described scope leads to the following problem statement:

> ***How can the energy dissipation in FIR filters implemented on both software as well as hardware-programmable platforms be minimised?***

This problem statement is followed by a number of sub-questions. These will clarify some of the challenges in the project and be of great help to structure the project work.

- *How can a reference filter be designed in a way that it will be representative for comparison of different optimisation methods?*

- *Which algorithmic and architectural methods exists for minimising the energy dissipation in digital filters and is it possible to improve them further?*
- *How can the methods be simulated, implemented and tested so they are easily comparable?*

The above stated questions will be answered in the remaining parts of this report. Together with the previous chapter, the basis of this project is now founded. The motivation for investigating this topic of low-energy realisation of FIR filters are described in the introduction and here the specifications and scope of the project are made.

3 Evaluation Set

The following chapter will introduce the evaluation set used in project. The evaluation set has the purpose of making it possible to evaluate the different filter implementations and hereafter compare these in an easy way. The same evaluation set is used for evaluation of both the software part and hardware part of the project.

As this project take an offset in already known methods of designing low-power filters, it is aimed for an evaluation set easily comparable to what is used in other literature on the topic. Literature shows evaluation sets consisting of low-pass filters with a fixed cut-off frequency and transition bandwidth, while the filter length is varied [Mehendale et al., 1998][Maskell, 2007][DeBrunner, 2007]. Other literature does not explicitly state what kind of FIR filters they use in their experiments, they do however present results for various filter lengths [Mehendale and Sherlekar, 2001][Meher, 2010].

Based on the evaluation sets used in other literature on the topic and to narrow the evaluation set to a sensible size, the evaluation set contains low-pass filters of different lengths. The cut-off frequency and transition band are kept fixed and the phase property is kept linear. Table 3.1 summarises the evaluation set.

Filter type	**Low-pass filter**
Cut-off frequency	$\pi/2$ rad
Transition bandwidth	0.1 rad
Phase property	Linear
Coefficient bits	16 bits
Filter length	8 to 120 (step-size 8)

Table 3.1: Filter parameters of the evaluation set.

As can be seen in table 3.1, the evaluation set contains a low-pass filter, where the filter length and thereby the stop-band attenuation is varied. The low-pass filters desired frequency response is seen on figure 3.1, where the stop-band attenuation is not given a value as this will change by the changing filter length. 16 bits will be used as the number of bits in the binary representation of the coefficients. The appropriate number of bits varies from application to application, and it is therefore simply chosen to use 16 bits implementations.

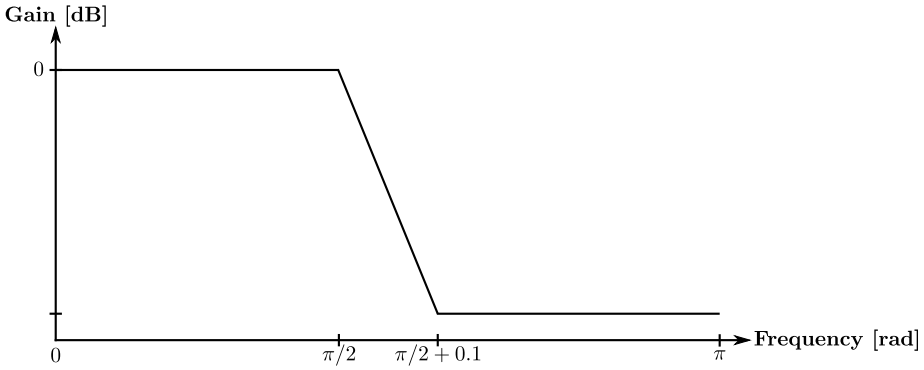

Figure 3.1: The desired frequency response of the evaluation set filters.

To find the filter coefficients, it is found to be convenient to use the Parks-McClellan method (described in appendix B on page 163), which is called the *remez*-function in Python.

As an example on how to use this function, the filter with 8 taps from the evaluation set can be created by executing the following:

```
1 filter = remez(8, [0, pi/2, pi/2+0.1, pi], [1, 0], Hz=2*np.pi)
```

Listing 3.1: Example of using the remez function in Python.

Here the cut-off frequency is $\pi/2$ and the transition band is 0.1. The remez-function creates a coefficient set which will give a filter characteristic where the attenuation will be 1 from 0 to $\pi/2$ rad, and as low as possible from $\pi/2+0.1$ to π rad.

To confirm that the remez-function gives the proper output, the filters are simulated. To avoid showing all filters in the evaluation set, the filters with filter 8, 40, 80 and 120 taps are the only ones shown here. The amplitude and phase response of these four filters are shown on figure 3.2 and 3.3 respectively.

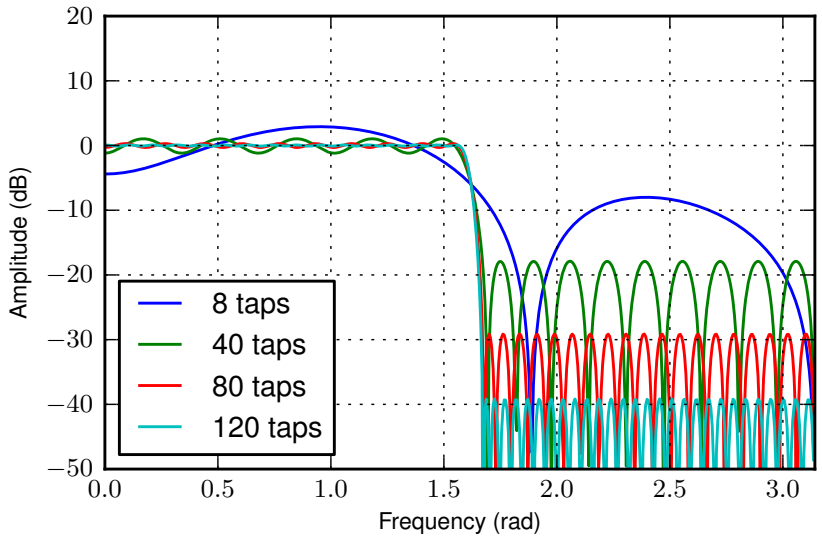

Figure 3.2: The amplitude response of the reference filters of length 8, 40, 80 and 120.

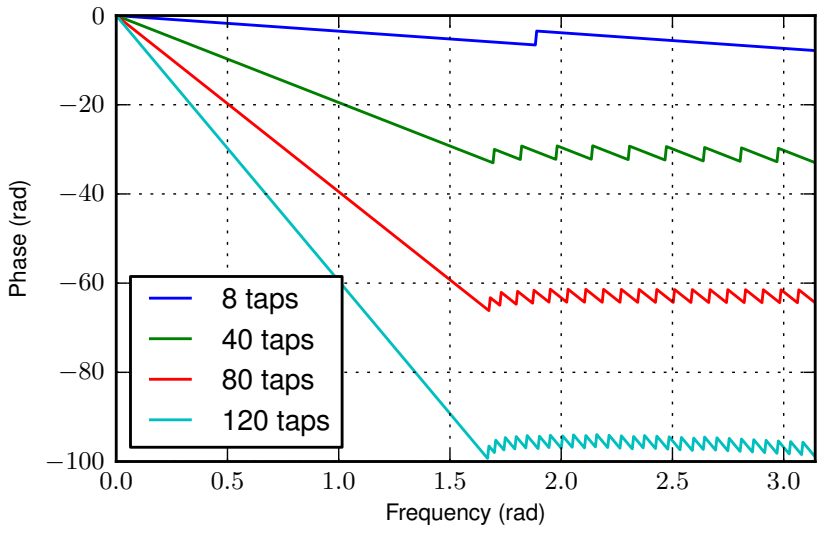

Figure 3.3: The phase response of the reference filters of length 8, 40, 80 and 120.

It is seen that the amplitude response shows a pass-band attenuation of approximately 0 dB and a stop-band attenuation which is approximately 8, 18, 39 and 49 dB for the filters with 8, 40, 80 and 120 taps respectively. The phase response is linear which is a general property of the filters created by the Parks-McClellan method.

The evaluation set is now introduced and the coefficients can now be used in the simulations, implementations and evaluations in both software and hardware. It will hereafter be possible to compare the different optimisation methods, as the same set of coefficients are used to evaluate these.

Part II

Software Optimisation Methods

4 Introduction to Software Filters

This part of the report contains the analysis, simulation and evaluation of the optimisation methods for software programmable platforms. The methods associated with software optimisation, are focused on reducing the switching activity or changing the structure of the filter. This is different from hardware optimisation as that will focus on the architectural aspects of the implementation. When changing the coefficients or the structure of the filter, either the switching activity or the computational complexity is changed. Reducing one of these factors introduces an overall reduction in energy dissipation.

The methods evaluated in this part of the project are coefficient optimisation, coefficient scaling and coefficient ordering. These methods are chosen as [Mehendale et al., 1998] introduces and simulates these, but does not implement and evaluate them on an actual platform. In addition, the multirate filtering method is also analysed, implemented and evaluated in this project. Also this method is suggested by [Mehendale et al., 1998]. The main conclusions from the following chapters will be on the comparison between the theory and the actual implementation of the method, as this is an interesting point to follow.

The evaluation of the methods will be one of the key features of the work performed in the this project. This, as the evaluation of the methods is actual energy measurements on a state-of-the-art microcontroller. This approach is different from previous work made by [Mehendale et al., 1998], which as far as our knowledge goes, does not look at the actual measurements of the methods implemented on a programmable platform, for other than the multirate realisation.

Although the project specification states that a DSP will be used for the software implementation, the project turns to use a microcontroller. The microcontroller used for implementation will be a Microchip dsPIC, which is actually a microcontroller with DSP functionalities. The reason for choosing a dsPIC, is that the Microchip microcontrollers is a well-known, high-performing platform, which is used in many different industries all over the world. This makes it interesting, as it is one of the most popular microcontrollers on the marked. Another important factor is that the architecture of this microcontroller, as this has the bus structure required for full utilisation of the some of the following optimisation methods. This makes it very general and a good platform to get a general view on how the optimisation methods will work.

4.1 Reference filter

The reference filters used as reference point in the evaluation of the software optimisation methods, are general direct form filters. The filter is illustrated on figure 4.1.

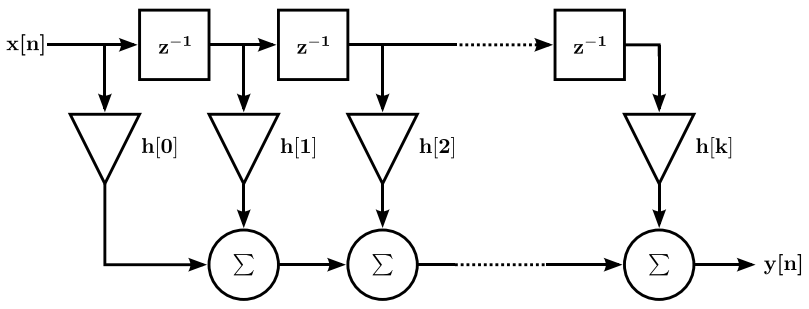

Figure 4.1: Block diagram of the direct form FIR filter.

The direct form FIR filter is described as:

$$y[n] = \sum_{k=0}^{N-1} h[k]x[n-k] \tag{4.1}$$

where y and x are the output and input respectively, and h are the coefficients of the filter.

The filter can be written as a C program, by the following lines of code:

```
1 x[oldest] = sample();
2 y = 0;
3 for (k = 0; k < N; k++)
4 {
5     y += h[k] * x[(oldest + k) % N];
6 }
7 oldest = (oldest + 1) % N;
```

Listing 4.1: C implementation of a digital filter.

In this implementation, the input is sampled and put into a ring buffer of length N overwriting the oldest sample. Next, the *for*-loop multiplies and summarise to get the output y. In the end, the index of the oldest sample is counted one up for the next sample to enter the code. In this project, the implementations has been carried out by assembly code, but the C code is more illustrative and describes the same general structure.

5 Multirate Decimation

By using decimation and interpolation a filter can be divided into several subfilters of lower order thereby leading to lower computational complexity of the filter. This is known as multirate filtering. The following chapter will contain descriptions of the necessary techniques used in multirate filtering, leading to an analysis of the complexity of such. The last part of this chapter will make a simulation which will compare the output from a direct form filter and the multirate decimated filter.

5.1 Changing the sample rate

The change in sampling rate must be considered when designing multirate filter, and is briefly described here.

A continuous-time signal can be converted to a discrete-time signal by sampling it with a time period, T, as:

$$x[n] = x_c(nT) \tag{5.1}$$

The sampling frequency (time period) can be changed in discrete-time as follows:

$$x_d[n] = x[nM] = x_c(nMT) \quad \text{(Decimation)} \tag{5.2}$$

$$x_i[n] = x[n/L] = x_c(nT/L) \quad \text{(Interpolation)} \tag{5.3}$$

Decimation is the technique where the sampling frequency is lowered or down-sampled. The decimated signal described in eq. (5.2) could e.g. decimate with $M = 2$ thereby deleting every second sample. The opposite is the interpolation where the sampling frequency is increased or up-sampled. This is described in eq. (5.3) and will for $L = 2$ insert a value equal to zero between all the original samples.

The re-sampling factors M and L can only be integers, but a re-sampling by non-integer numbers are possible by inserting both decimators and interpolators after each other.

5.2 Polyphase decomposition

This section describes polyphase decomposition which is the key element of multirate filtering.

Polyphase decomposition can be obtained by representing a sequence in M sub-sequences, where each subsequence contains every M'th value of the a successive delayed original sequence. If the wish is to decompose a filter impulse response into M sub-sequences it will look as:

$$\begin{aligned} H(z) = &\sum_{k=-\infty}^{\infty} h(kM) z^{-kM} \\ &+ z^{-1} \sum_{k=-\infty}^{\infty} h(kM+1) z^{-kM} \\ &\vdots \\ &+ z^{-(M-1)} \sum_{k=-\infty}^{\infty} h(kM+M-1) z^{-kM} \end{aligned}$$

which in a more compact form could be written as [Vaidyanathan, 1993]:

$$H(z) = \sum_{i=0}^{M-1} z^{-i} E_i(z^M) \tag{5.4}$$

where:

$$E_i(z) = \sum_{k=-\infty}^{\infty} e_i(k) z^{-k} \tag{5.5}$$

with:

$$e_i(k) \stackrel{\triangle}{=} h(Mk+i), \quad 0 \leq i \leq M-1 \tag{5.6}$$

from which it can be seen that the sub-sequences are successive decimated versions of the filters impulse response. The complete equation will for $M = 2$ look like:

$$\begin{aligned} H(z) &= \sum_{i=0}^{1} z^{-i} E_i(z^2) \\ &= E_0(z^2) + z^{-1} E_1(z^2) \\ &= \sum_{k=-\infty}^{\infty} e_0(k)(z^2)^{-k} + z^{-1} \sum_{k=-\infty}^{\infty} e_1(k)(z^2)^{-k} \\ &= \sum_{k=-\infty}^{\infty} h(2k)(z^2)^{-k} + z^{-1} \sum_{k=-\infty}^{\infty} h(2k+1)(z^2)^{-k} \end{aligned} \tag{5.7}$$

5.3 Multirate FIR filter

A FIR filter can be decomposed into a multirate FIR filter as proposed in [Mehendale et al., 1996]. The following will show the general example, where a 1-level multirate filter will be created from a N-tap FIR filter.

The first thing to do is to define the FIR filter as:

$$y[n] = \sum_{i=0}^{N-1} a_i \cdot x[n-i] \tag{5.8}$$

This can be Z-transformed:

$$Y(z) = \sum_{i=0}^{N-1} A_i \cdot X(z) \cdot z^{-i} \tag{5.9}$$

and the transfer function of the filter will then look as:

$$H(z) = \frac{Y(z)}{X(z)} = \sum_{i=0}^{N-1} A_i \cdot z^{-i} \tag{5.10}$$

The transfer function can then be split up into odd and even coefficients:

$$\begin{aligned} H(z) &= \sum_{k=0}^{N/2-1} A_{2k} \cdot z^{-2k} + \sum_{k=0}^{N/2-1} A_{2k+1} \cdot z^{-(2k+1)} \\ &= \sum_{k=0}^{N/2-1} A_{2k} \cdot (z^2)^{-k} + z^{-1} \cdot \sum_{k=0}^{N/2-1} A_{2k+1} \cdot (z^2)^{-k} \qquad (5.11) \\ &= H_0(z^2) + H_1(z^2) \cdot z^{-1} \qquad (5.12) \end{aligned}$$

It is important to see that this decomposed version of the transfer function can be obtained using the polyphase decomposition described in eq. (5.4) to (5.6). This can also be seen by comparing eq. (5.7) and eq. (5.11). The two new filters H_0 and H_1 of $N/2$-taps generates the same output as the original FIR filter with N-taps when used as eq. (5.12).

The input signal X(z) is in the same way decomposed in odd and even samples. This can easily be done as the input sequence is defined as:

$$X(z) = \sum_{i=0}^{N-1} X_i \cdot z^{-i} \tag{5.13}$$

By comparing eq. (5.13) with eq. (5.10) and eq. (5.12) it is seen that the input sequence can be decomposed in the same way:

$$X(z) = X_0(z^2) + X_1(z^2) \cdot z^{-1} \tag{5.14}$$

where X_0 and X_1 both are $N/2$ of length and represent the even and odd samples respectively.

The output Y(z) can now be written as:

$$\begin{aligned} Y(z) &= (H_0 + H_1 \cdot z^{-1}) \cdot (X_0 + X_1 \cdot z^{-1}) \\ &= H_0 \cdot X_0 + (H_0 \cdot X_1 + H_1 \cdot X_0) \cdot z^{-1} + H_1 \cdot X_1 \cdot z^{-2} \\ &= C_0 + C_1 \cdot z^{-1} + C_2 \cdot z^{-2} \end{aligned} \tag{5.15}$$

where:

$$\begin{aligned} C_0 &= H_0 \cdot X_0 \\ C_2 &= H_1 \cdot X_1 \\ C_1 &= (H_0 \cdot X_1 + H_1 \cdot X_0) \end{aligned}$$

It is noted in [Mehendale et al., 1996] that C_1 can be rewritten as:

$$C_1 = (H_0 + H_1) \cdot (X_0 + X_1) - C_0 - C_2 \tag{5.16}$$

to obtain a desirable multirate architecture. As the input signal is split up, so is the output signal:

$$Y(z) = Y_0(z^2) + Y_1(z^2) \cdot z^{-1} \tag{5.17}$$

The two output samples are then calculated as:

$$Y_0(z) = C_0 + C_2 \cdot z^{-2} \tag{5.18}$$

$$Y_1(z) = (H_0 + H_1) \cdot (X_0 + X_1) - C_0 - C_2 \tag{5.19}$$

which is illustrated on figure 5.1 in terms of a data-flow graph (DFG).

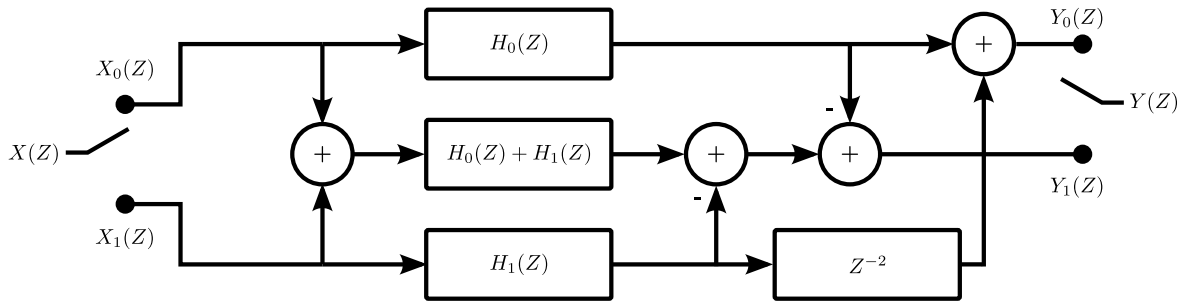

Figure 5.1: DFG of the decimated multirate 1-level architecture. [Mehendale et al., 1996]

The input and output switches on the figure distributes the even and odd samples, as to split and connect the samples, respectively.

It is possible to expand the multirate architecture by further decimating each of the subfilters in figure 5.1. By doing so the architecture will be increased to gain a multirate level-2 architecture.

5.3.1 Computational complexity

The following will analyse the computational complexity for both non-linear and linear phase filters.

Non-linear phase

The computational complexity of the N-tap direct form FIR filter seen on figure 5.2 is N multiplications and $N-1$ additions.

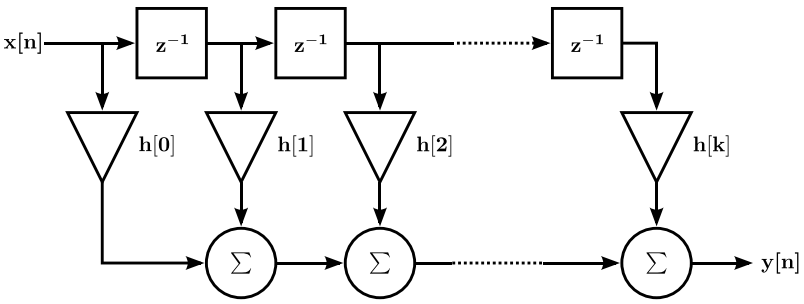

Figure 5.2: The direct form FIR filter.

By introducing the multirate 1-level FIR filter structure the computational complexity of each of the subfilters is $N/2$ multiplication and $N/2-1$ additions. As seen on figure 5.1 there are introduced 4 additional addition to compute two outputs. The complexity in the case with even number of taps rounds up to a total of:

$$3 \cdot \frac{N}{2} \cdot \frac{1}{2} = \frac{3N}{4} \qquad \text{multiplications}$$

$$3 \cdot \left(\frac{N}{2} - 1\right) \cdot \frac{1}{2} + \frac{4}{2} = \frac{3N+2}{4} \qquad \text{additions}$$

The factor of one divided by two, comes from the fact that we have decimated the signal in to even and odd samples, and the filters need only to do computations on every second sample. In the same way, the case with odd number of taps gives a complexity of:

$$\left(2 \cdot \frac{N+1}{2} + \frac{N-1}{2}\right) \cdot \frac{1}{2} = \frac{3N+1}{4} \qquad \text{multiplications}$$

$$\left(2 \cdot \frac{N}{2} + \frac{N-2}{2}\right) \cdot \frac{1}{2} + \frac{4}{2} = \frac{3N+6}{4} \qquad \text{additions}$$

For both cases the multirate structure has a lower computational complexity than the direct form structure. This is an important result, as it can be exploited in order to design low-energy filters.

Linear phase FIR filter

The computational complexity of the N-tap linear phase FIR filter is $N/2$ multiplications for even number of taps and $(N+1)/2$ multiplications for odd number of taps and $N-1$ additions in both cases. The number of additions is the same as for an even number of taps non-linear filter, and will therefore give no reduction in terms of additions. The multiplication complexity is about half the one for non-linear phase, since the symmetric property of the coefficients can be exploited to lower the number of multiplications. Due to this reduction in complexity the multirate architecture is not as beneficial for linear phase filters as for non-linear phase FIR filters. This comes clear by summing up all multiplications needed. For linear phase filters with even N and even $N/2$ the complexity is:

$$\left(2 \cdot \frac{N}{2} + \frac{N}{4}\right) \cdot \frac{1}{2} = \frac{5N}{8} \qquad \text{multiplications} \tag{5.20}$$

In the case where $N/2$ is odd the number of multiplications is:

$$\left(2 \cdot \frac{N}{2} + \frac{N+2}{4}\right) \cdot \frac{1}{2} = \frac{5N+2}{8} \qquad \text{multiplications} \tag{5.21}$$

In both these cases the number of multiplication is higher than in the direct form structure. Therefore it can be concluded that this first level multirate architecture is not suitable for linear phase FIR filters.

By continuous decimation of the subfilters and thus increasing the multirate level, the computational complexity can be further reduced. This goes for both non-linear phase filters and for linear phase filters. Further decimation makes the architecture more complex to work with and is therefore not considered here. Based on the previous analysis of computational complexity for both non-linear phase filters and linear phase filters, some conclusions can be drawn. The first level multirate architecture is more beneficial for non-linear phase filters than for linear phase filters when exploiting the symmetry.

5.3.2 Power analysis of multirate architectures

In this section a power analysis provided by [Mehendale et al., 1996] is presented. This analysis focuses on how the reduction in computation complexity further can reduce the power dissipation in the multirate architecture.

As stated previously in chapter 1, the power dissipation due to switching is the controllable source of power dissipation. The power reduction ratio using first level multirate architecture can be expressed in terms of the switching power equation like this:

$$\frac{P_{multirate}}{P_{direct}} = \frac{C_{multirate}}{C_{direct}} \cdot \left(\frac{V_{multirate}}{V_{direct}}\right)^2 \cdot \frac{f_{multirate}}{f_{direct}} \tag{5.22}$$

The total switching capacitance is assumed to be proportional to the total area of the multipliers and adders [Mehendale et al., 1996]. If the area ratio is the same as the delay ratio and the frequency of the multirate architecture is designed to maintain the filter throughput, the C_{direct} and $C_{multirate}$ are the same. This is an assumption made in [Mehendale et al., 1996]

Since the multirate realisation of FIR filters use less computations per output it can run on a longer clock period, which also means that it can run on a lower frequency, while maintaining the same throughput. The frequency reduction in the multirate architecture translates directly into lower power dissipation. The frequency ratio between the multirate and the direct form structure is given by the total delay of the multiplications and additions in the different structures. For a N-tap non-linear phase FIR filter the frequency ratio can be expressed as:

$$\frac{f_{multirate}}{f_{direct}} = \frac{3N/4 \cdot \delta_m + (3N+2)/4 \cdot \delta_a}{N \cdot \delta_m + (N-1) \cdot \delta_a} \tag{5.23}$$

where δ_m and δ_a are the delay of multiplications and additions respectively.

Reducing the frequency, corresponds to increasing the clock period. Since the clock period is increased, the logic delays are allowed to increase correspondingly without affecting the overall throughput. The supply voltage is one of the factors that affects the delays. The logic delays are allowed to increase according to the relationship

$V_{dd}/(V_{dd}-V_T)^2$. V_T is the threshold voltage of the transistor and the V_{dd} is the supply voltage. This relationship is illustrated in figure 5.3, for $V_T = 0.8V$. The delay values are normalised with respect to $V_{dd} = 5V$.

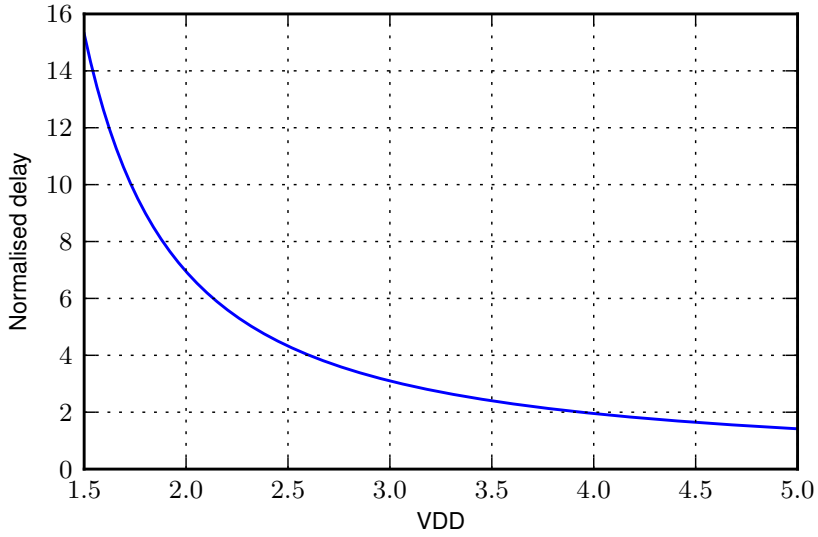

Figure 5.3: Normalised delay versus supply voltage.

To sum up, the power reduction using multirate architecture mainly depend on the amount the frequency can be lowered. This also makes it possible to reduce the voltage, which also has a great impact on the power dissipation. However; for the further work on this topic only the reduction in computational complexity will be considered. It will be investigated to what extent the reduced complexity results in a more energy efficient implementation of the filter.

5.4 Simulation

The simulation of the multirate technique takes the output from the structure described in the two equations, eq. (5.18) and (5.19), which are illustrated on figure 5.1, and compares it to the output from a direct form FIR filter. This is done by coding both filter structures and generating output by putting the same data into the filters.

The direct form FIR filter can be coded using the Python programming language as:

```
1 y = list(zeros(len(x)))
2 for j in range(len(x)):
3     for i in range(len(coef)):
4         y[j] = y[j] + coef[i]*x[j][i]
```

Listing 5.1: Python implementation of a direct form FIR filter.

having a list of coefficients, *coef*, and an input array, *x*. The input array is a simulated ring buffer, where each row is a time instance of the buffer. The columns are the time delayed versions of the input signal. The first row in the array is the buffer when one value has been clocked into the filter. The second row, is when two values has been clocked in, and so on. As an example, the array is showed, where the first three input values are 1, 2 and 3 and the following are zero. The array is created for a 8 tap filter, thereby meaning that the array has 8 columns and rows.

The array looks as:

$$x = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 2 & 1 \end{bmatrix}$$

In the same way, the multirate filter can be programmed in Python as:

```
y0 = []
y1 = []
f1_old = 0
for j in range(len(x1)/2):
    f0, f1, f0f1 = 0, 0, 0
    for i in range(len(h0)):
        f0 = f0+h0[i]*x0[j][i]
        f1 = f1+h1[i]*x1[j][i]
        f0f1 = f0f1+h0h1[i]*(x0[j][i]+x1[j][i])
    y0.append(f0+f1_old)
    f1_old = f1
    y1.append(f0f1-f1-f0)
```

Listing 5.2: Python implementation of a multirate FIR filter.

where the direct form filter coefficients are split into even, *h0*, odd, *h1* and added even and odd, *h0h1*. The output from the subfilters, is saved in the variables, *f0*, *f1* and *f0f1* respectively. The output from the multirate filter is saved in even and odd samples, *y0* and *y1* respectively. The delay element used in the multirate structure is denoted *f1_old*. The input for the multirate filter, is two input arrays, *x0* and *x1*, which is created using the same idea as for the direct form filter input. The difference is that the two arrays hold the even and odd samples instead of all the samples. Using the same example as form the direct form filter input, the arrays looks as:

$$x0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 0 & 3 & 1 & 0 \\ 0 & 0 & 3 & 1 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad x1 = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

By using the three arrays from the two previous examples of input arrays, and a filter created by the *remez* function in Python, the output from the filters can be generated. The generated filter has a cut-off frequency of $\pi/2$ and is created by the following function as:

```
filter = remez(8, [0, pi/2, pi/2+0.1, pi], [1, 0], Hz=2*np.pi)
```

Listing 5.3: Example of using the remez function in Python.

The output created by the direct form FIR filter and multirate FIR filter is found to be:

$$y_{direct} = \begin{bmatrix} -0.089713390262198878 \\ -0.43022586025639231 \\ -0.58229138537522085 \\ 0.077624914901158693 \\ 1.9247256276653326 \\ 2.4540882666074277 \\ 1.4854796030579729 \\ -0.025970715100093356 \end{bmatrix} \qquad y_{multirate} = \begin{bmatrix} -0.089713390262198878 \\ -0.43022586025639226 \\ -0.58229138537522085 \\ 0.077624914901158665 \\ 1.9247256276653324 \\ 2.4540882666074277 \\ 1.4854796030579729 \\ -0.025970715100093411 \end{bmatrix}$$

it is seen that the 8 output samples are mostly equal, but at sample 2, 4, 5, and 8, the last decimals are different. The errors are in the size of 10^{-17}. This small error is assumed to be introduced by rounding errors when calculating the filters. Due to the size of the errors, they are not seen as a flaw in the theory of multirate filtering, but rather from the way the filters are calculated.

By generating simulated outputs from coded versions of the direct structure and multirate FIR filters, the output can be concluded to be equal which means that they work in the same way.

6 Evaluation of Multirate Filtering

The following chapter is conducted in order to evaluate the energy consumption of the multirate filter compared to the normal direct form filter. The chapter will introduce the term energy model as an estimate on the energy consumption as this is wanted prior to the actual energy measuring. In the end a comparison between the estimated energy model and the measurement of actual filters implemented on a platform, is shown.

As the multirate architecture reduces the computational complexity, this is exploited in order to achieve a reduction in the energy consumption, while maintaining the throughput of the FIR filter. The following section will first make an estimate of the energy consumption, based on energy models introduced by [Tiwari et al., 1996], and thereafter make an actual energy measurement of filters with different length.

6.1 Energy model

The energy model will help to make an estimate on how much energy is consumed when using multirate filters in proportion to the direct form filter. The model gives an approximation of how much energy is consumed by each instruction used in the assembly program for the filters.

When the overall energy model has to be generated, the following three costs are introduced by [Tiwari et al., 1996]:

Instruction base costs: The base cost is the primary part of the energy model and is the energy consumed by a given instruction. The base energy cost can be found by calculating the energy from a measurement of the average power.

Effects of circuit state: The effects of circuit state are introduced, as the energy of an instruction is dependent on which instruction is executed before it. Since more switching activity leads to higher energy consumption, switching between different parts of the DSP, when using different consecutive instructions, leads to a higher energy consumption, than when the instructions are the same.

Other inter-instruction effects: Other inter-instruction effects are e.g. buffer stalls, pipeline stalls and cache misses. Such scenarios could happen if all instructions depend on a single instruction to finish. The DSP will then need to insert wait-states or stalls (also called bubbles) into the pipeline, until a new instruction can be fetched into the pipeline.

Of the three costs, the instruction base cost is the primary part, and this is why the following will be focused on using this to estimate the energy consumption. Effects of circuit state will also be taken into account, but the inter-instruction effects will be neglected. The reason for simplifying the model, by not including the inter-instruction effects, is that these are hard to find and are not considered to introduce a larger amount of energy dissipation.

The procedure for estimating the total energy consumed by the filter is now defined as:

$$E_P = \sum_{i=0}^{L} B_i \cdot N_i + \sum_{i=0 j=0}^{L} O_{i,j} \cdot N_{i,j} \tag{6.1}$$

where E_P is the energy of the program, P, and L is the number of different instructions. B_i is the base energy cost of each instruction, i, which is multiplied by the number of times the instruction is executed, N_i. O is the circuit state overhead for each pair of consecutive instructions i, j, multiplied with the number of times the pair is executed.

To find the base energy cost of each instruction, a loop of several instances of the instruction is executed on the DSP. The loop has to contain several instances of the instruction, to make the branch/jump instruction at the bottom

of the loop negligible. The literature on this topic [Tiwari et al., 1996] suggest about 200 of the same instruction should be sufficient. For the mathematical instructions or other instruction requiring a number as operand, such as MOV, numbers containing as many ones and zeros will be used, for instance $0xFF00$. This has been found to give an average of the instructions energy consumption. With the loop running on the DSP, the average power can be found by measuring the voltage over a shunt resistor as described in the measuring procedure in appendix C on page 167. The DSP used in these measurements is actually a microcontroller from Microchip, the state-of-the-art microcontroller dsPIC30F3012 with DSP functionality. When the average power is known, the energy consumed by a single execution of the instruction can be found by calculating:

$$E = P \cdot N \cdot \tau \tag{6.2}$$

As for the effects of circuit state overhead the assembly code is looked into for the purpose of finding out which consecutive instruction exist. Two consecutive instructions are then alternating in a loop, also this consisting of about 200 pairs of instructions. The circuit state overhead of an instruction pair is defined as the difference between the measured energy of the two instructions and the sum of the base cost of the two instructions.

When the base cost energy and circuit state overhead are found for all used instructions, the total energy can be found by using eq. (6.1).

6.1.1 Results

This section provides the results of the energy consumed by the multirate filter compared to a direct form implementation, by the use of the energy model presented above.

The dsPIC30F3012 is standardised with a 7.37 MHz crystal oscillator. This frequency is multiplied by 8 internally, which then increase to a frequency of 58.89 MHz. This converts to a clock period of 16.95 ns. Each instruction cycle uses four clock cycles, which results in an instruction period of 67.82 ns.

When implementing filters on a microcontroller, the multiply and accumulate (MAC) instruction is used, instead of multiplications and additions as is used when calculating the computational complexity theoretically. The relationship between the multiplication combined with an addition and the MAC has to be considered. Table 6.1 show the base cost of additions, multiplications and MACs coded in assembly.

Instruction	Power [mW]	Cycles	Energy [nJ]
ADD	149.29	1	10.13
MUL	154.87	1	10.51
MAC	201.21	1	13.65

Table 6.1: Base cost of multiplication and addition.

The addition consumes 10.13 nJ and the multiplication consumes 10.51 nJ. As the MAC instruction replaces the two other instructions in an actual implementation, it uses more energy than the two other, it consumes 13.65 nJ, which is not equivalent to the addition and multiplication combined. As of this the MAC is however used as a replacement of the addition and multiplication in the following implementation and testing. A direct form filter will then have N MACs instead of N multiplications and $N-1$ additions.

Table 6.2 describes the base cost of the instructions used in the assembly program for direct form and multirate realisation of FIR filters. This will be used later to calculate the energy consumption of the different filter realisations. For some instructions the base cost can not be found, this is because some instructions can not be executed consecutively. This goes for the REPEAT instruction, as one REPEAT can not follow another REPEAT. In that case the base cost for the repeat instruction can not be found and will not be considered in the energy model.

Instruction	Functionality	Power [mW]	Cycles	Energy [nJ]
ADD	Addition	149.29	1	10.13
MUL	Multiplication	154.87	1	10.51
MAC	Multiply and Accumulate	201.21	1	13.65
SAC	Store accumulator	148.93	1	10.10
SUB	Subtraction	149.99	1	10.17
CLR	Clear	185.85	1	12.61
MOV	Move	150.53	1	10.21
BRA	Branch	195.12	2	26.47

Table 6.2: Base cost for instructions used.

The assembly program for both the direct form realisation and the multirate has been investigated and the consecutive instructions are noted. The circuit state overhead for these instructions are listed in table 6.3. For some instructions the circuit state overhead can not be found, this is because the base cost of the instruction is not found, as is the situation with the REPEAT instruction. The base cost of the instruction combination MOV followed by a BRA, can neither be found. This is due to the nature of the BRA instruction, which will branch out of the loop and preventing only the two instructions to run in the loop. The instruction combinations used, but not resulting in an circuit state overhead is marked in the table with -. These circuit states will not be included in the total energy model either.

Instruction	MAC	SAC	SUB	CLR	MOV	BRA	ADD
MAC		6.87					
SAC					11.94		
SUB							
CLR							
MOV			12.37	8.68		-	11.94
BRA							
ADD							

Table 6.3: Circuit state overhead between consecutive instructions. Values are in nJ.

There will also be a circuit state overhead between two similar instruction, though this is not possible to calculate and will not be taken into account when computing the energy model.

With both the base costs and circuit state overhead found, the overall instruction level energy model for direct form and multirate realisation can be found using eq. (6.1). For direct form realisation the total energy model is found as:

$$E_P = 3 \cdot B_{MOV} + B_{CLR} + B_{SAC} + B_{BRA} + N_{taps} \cdot B_{MAC} + O_{MOV\&CLR} + O_{MAC\&SAC} + O_{SAC\&MOV} \tag{6.3}$$

For the multirate realisation the energy model is found as:

$$\begin{aligned} E_P =& (5 \cdot B_{MOV} + 3 \cdot B_{CLR} + 3 \cdot B_{SAC} + B_{ADD} + 2 \cdot B_{SUB} + B_{BRA} + 3 \cdot N_{taps}/2 \cdot B_{MAC} \\ &+ 2 \cdot O_{MOV\&CLR} + 3 \cdot O_{MAC\&SAC} + 3 \cdot O_{SAC\&MOV} + 2 \cdot O_{MOV\&ADD} + 2 \cdot O_{MOV\&SUB})/2 \end{aligned} \tag{6.4}$$

The multirate energy model includes a division by two, as the multirate filter is decimated in two and hereby computes two outputs.

The total energy found for the direct form and multirate realisation is illustrated in figure 6.1. The filter length is increased from 8 up to 120 with steps of 8, as is described in the evaluation set in chapter 3 on page 13. From this

estimate the multirate realisation consumes more energy than the direct form for the two first filters. For longer filters the direct form consumes more energy than the multirate, as expected.

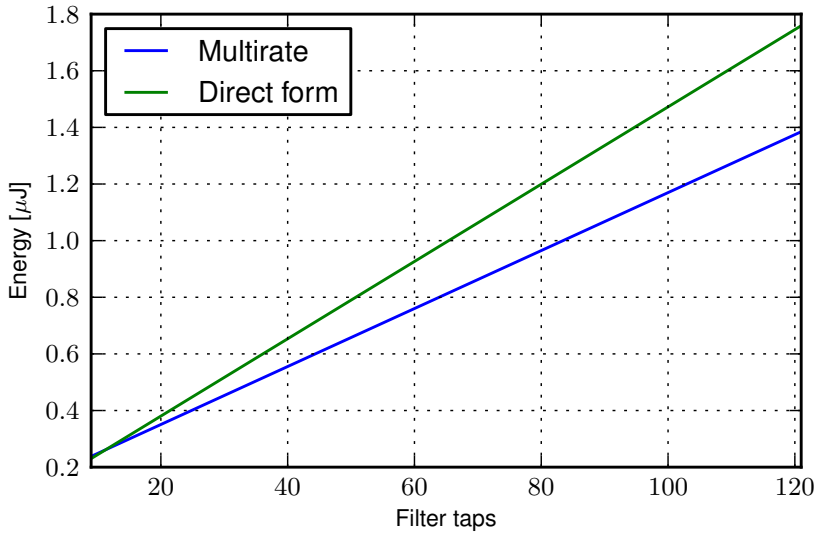

Figure 6.1: Graphical illustration of the energy model for direct form and multirate filters when implemented on a dsPIC30F3012.

The reason why the direct form consumes less than the multirate filter for the first two filters might be due to the circuit state overhead in the multirate filter. As the multirate realisation uses three filters of length $N/2$, it includes almost three times more shifting between the consecutive instructions used, thus increasing the total circuit state overhead, this might be a reason for the relatively large energy consumption in smaller filters. The circuit state overhead is not affected by the filter length, so as the filter length increases this does become more and more insignificant. Figure 6.2 illustrates the base cost and circuit state overhead in and between the instructions in the assembly program. The REPEAT instruction determines the number of times the MAC instruction is to be executed, though it does not affect the circuit state overhead between them.

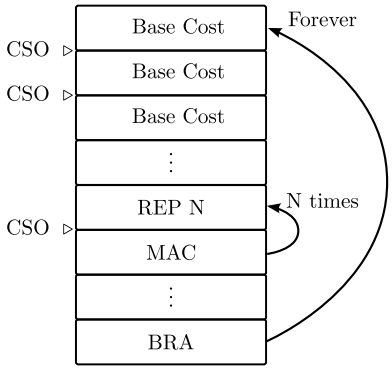

Figure 6.2: Base cost and circuit state overhead in a program.

The difference between the multirate filter and the direct form filter increases as the filter length increases. This is a clear effect of the reduction in computational complexity from direct form to multirate, as the multirate filter only uses $\frac{3N}{4}$ MACs when the direct form uses N MACs.

The possible optimisation to achieve using multirate filters instead of direct form filters is illustrated in figure 6.3. The optimisation if found as the difference in energy in the two realisations compared to the direct form:

$$\text{Optimisation} = \frac{E_{direct} - E_{multirate}}{E_{direct}} \cdot 100\% \tag{6.5}$$

The possible optimisation increases asymptotically, due to the reduction in computational complexity. Based on this estimate, optimisation is not possible for very small filters, but about 21 % optimisation is however possible for larger filters.

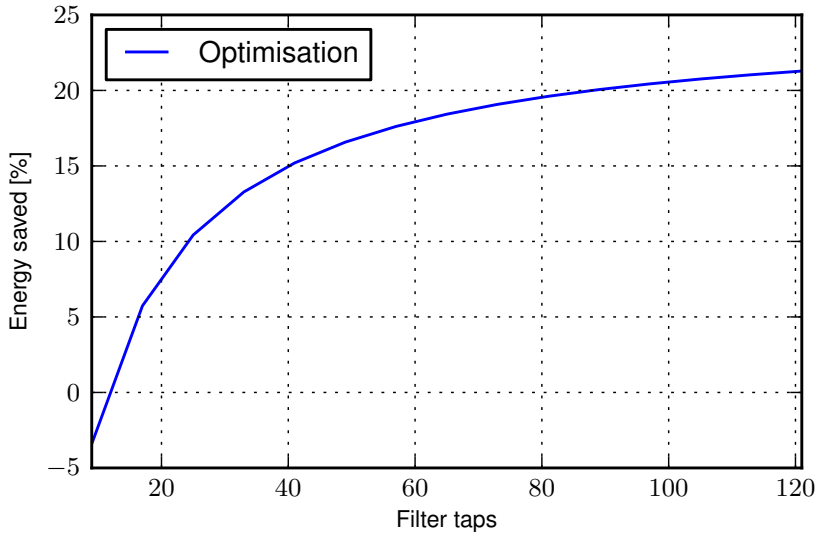

Figure 6.3: Optimisation achieved if replacing direct form filters with multirate filters.

6.2 Energy measurement

The described energy model can be used to estimate the energy used by the microcontroller, but the actual energy consumption has to measured by making an actual implementation of the filters. The following section will conduct measurements of the direct form filter and the multirate filter, to be able to compare the actual energy consumptions of the implementations. In addition, the measurements will also be compared to the energy model in order to evaluate the accuracy of this.

The energy measurements of the filters are performed as described in appendix C on page 167. From the measurements a list of average power consumptions of the filters, with changing number of filter taps, is found. This average power can be converted into energy by using eq. (6.2). The total number of cycles the filter uses to execute can be found by summing the cycles every instruction in the program takes. The number of cycles each instruction takes can be found in the *dsPIC30F Programmers Reference Manual*.

The filters used are defined in the evaluation set in chapter 3 on page 13. These filters have linear phase, but the symmetry in these are not exploited, thus this corresponds to using filters without linear phase. As the goal of multirate filtering is to reduce the computational complexity by decimating the filter, the optimisation would be the same regardless of the filter characteristics being low-pass, high-pass or band-pass.

6.2.1 Results

The results from the power measurements on actual filters implemented on the microcontroller, are shown in figure 6.4.

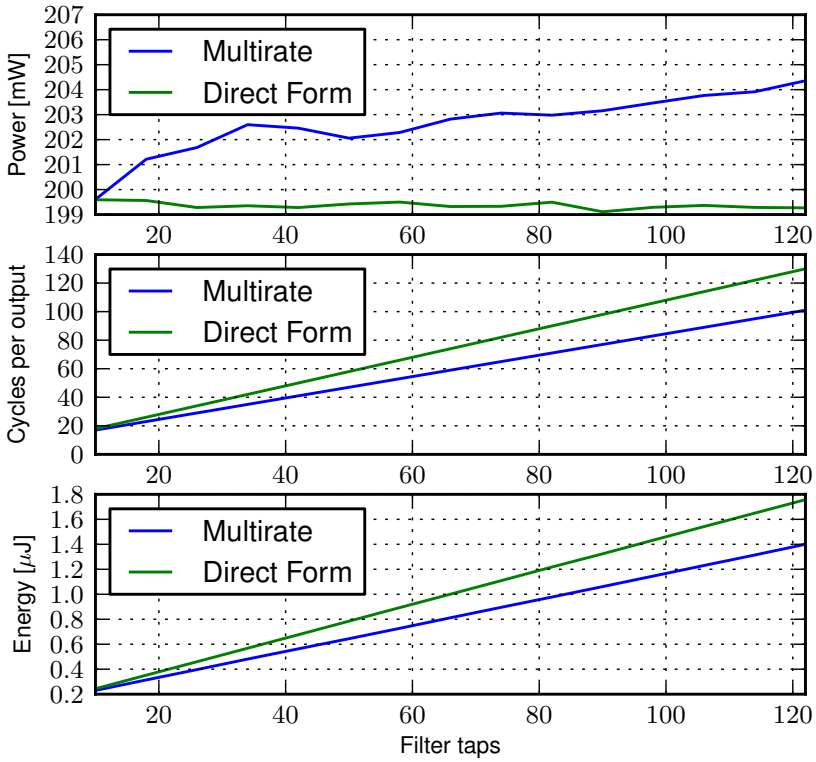

Figure 6.4: Comparison of differences between direct form and multirate filters of increasing number of taps.

As can be seen on the figure, the power consumption of the direct form filters are approximately the same, while the power of the multirate filters increases with the number of taps in the filter. The reason why the power of the multirate filter increases, is that the effects of circuit state is much bigger as the overhead of the filter is bigger. The overhead is bigger, as the decimated filter contains smaller filters, which each needs overhead instructions. The difference in overhead can be found by counting the instructions that are not multiply-accumulate instructions. For the direct form filter, the count of overhead instructions is 8, while it is 21 for the multirate filter.

When looking at the cycles per output, the number of cycles needed for the multirate filter is smaller than the direct form filter. It is also noted that the difference between the two filters becomes larger and larger, as the number of taps increases. This can be explained as the number of multiply-accumulate of the multirate filter is 75 % of the direct form filter, which then introduces higher differences as the number of taps increases.

The energy consumed by each execution of the filters is plotted in the last graph on figure 6.4. It is seen that the graph is very similar to the comparison of cycles per output. The savings in energy increases as the number of filter taps increases, which again is only natural, as the length of the multirate filter is 75 % of the direct form filter.

The total energy saved by using multirate filters instead of the normal direct form filter, can be seen on figure 6.5.

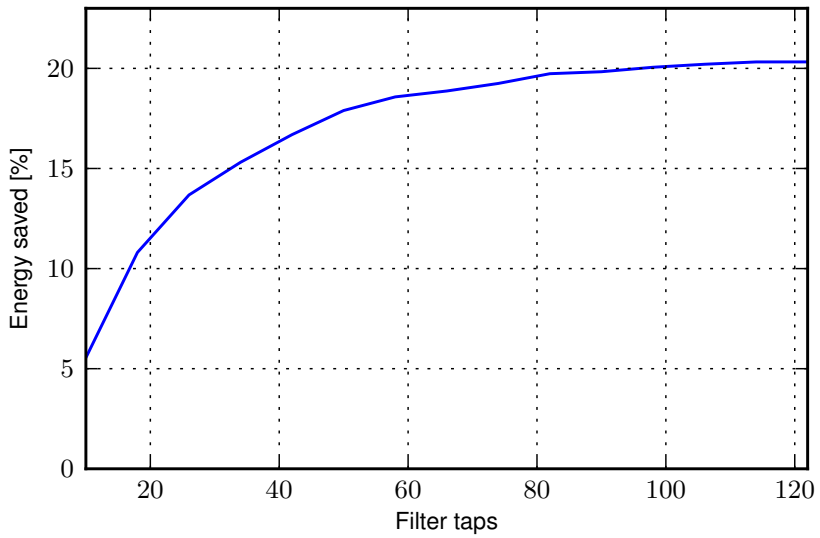

Figure 6.5: The percentage of energy saved when using multirate filters instead of direct form filter.

The graph is calculated by looking at the difference in energy compared to the energy of the direct form filter, which can be written as:

$$\text{Optimisation} = \frac{E_{direct} - E_{multirate}}{E_{direct}} \cdot 100\% \tag{6.6}$$

As can be seen on the figure, the trend is asymptotic. This can be explained by the overhead of the filters becoming more and more insignificant in relations to the whole length of the filter. In the end, the energy saving stops at 20.3 %, which can be seen as the maximum possible energy optimisation on this implementation.

The energy optimisation reached by [Mehendale et al., 1998] is as much as 40 % for filters over 42 taps. It is however presumed that this work uses frequency and voltage scaling as to get a higher optimisation than presented in this section. The theory on frequency and voltage scaling is shortly introduced in section 5.3.2 on page 25 and also in [Mehendale et al., 1998]. In addition, the work in [Mehendale et al., 1998] uses another DSP platform than used in this project. In the article a DSP from Texas Instruments is used.

It should be noted, that the graphs and results shown in this section will vary when the implementation is altered. The changes in implementation could e.g. be the choice of DSP or the way the filter is coded. By changing the platform or code, the energy optimisation could change to either better or worse implementations, thereby yielding that even higher or lower energy savings is a possibility.

6.3 Comparison

The following section will compare the results found by the energy model and the energy measurements. The energy consumption for the energy model and energy measurements are shown on figure 6.6.

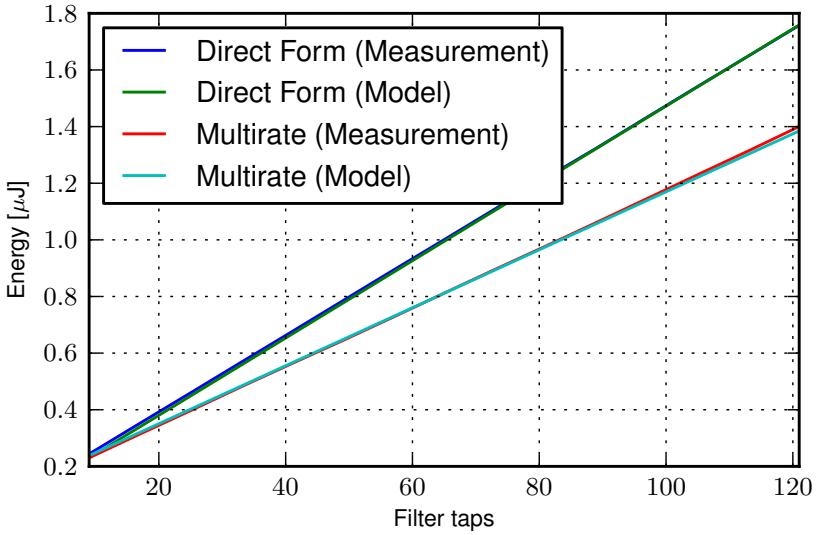

Figure 6.6: Comparison of the energy consumption for both the energy model and energy measurement.

It can be concluded from the figure, that the model gives a very good estimate on what can be expected when making a real measurement. The graphs of the model and measurements of the direct form are very similar. The average difference between them are 0.65%. As for the multirate filter the graphs of the modelled and measured energy consumption are also very similar, with an average difference of 1.9%. A conclusion on the comparison between the energy model and the energy measurement, is that they are very similar and it is possible to get a very good estimate on the energy consumption by using a model instead of conducting actual measurements.

When looking at the possible energy optimisation the model however does not give a very good estimate. A comparison between the energy model and energy measurements can be seen on figure 6.7.

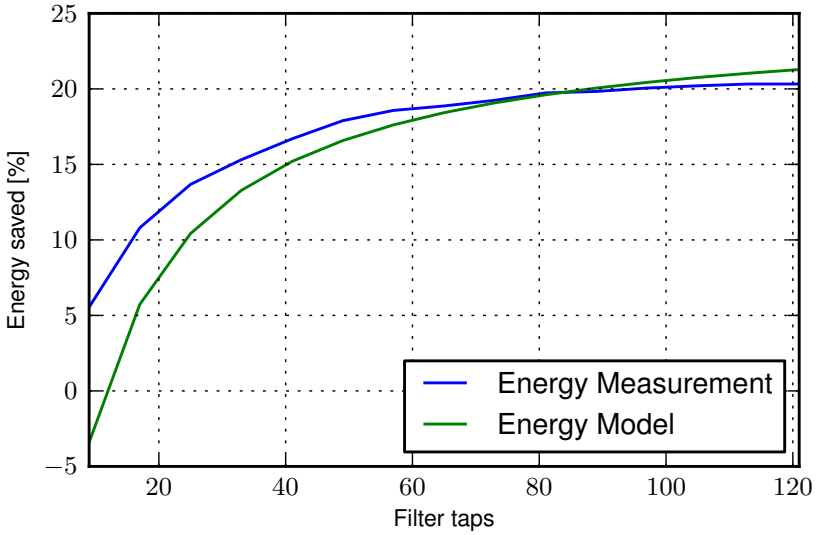

Figure 6.7: Comparison of the energy optimisation for both the energy model and energy measurement.

It can be seen that the energy model still has the asymptotic trend as the energy measurement, but starts at a negative value. The overall difference between the graphs causes a problem, as this means that the energy model can not be used to estimate how much energy that is saved when using multirate filters instead of direct form filters. This

is clear as the model will give wrong optimisation percentages compared to the actual measurements, especially at low number of taps. It should however be noted that the model can be used if a margin is introduced when specifying the optimisation percentage.

6.4 Conclusion

This section summarises and concludes on the multirate filter implementation.

The multirate filtering method has been investigated and the analysis showed that a reduction in computational complexity is possible for non-linear phase filters.

The created energy model gives good approximations to the energy consumption of the multirate and direct form filters. The model is however not that good at estimating the level of optimisation at smaller filters. All in all, it is noted that the energy model is a good approach when many different implementations has to be performed on a fixed platform, as it makes it easy to approximate the energy consumed, without making measurements every time.

A significant saving was discovered when using the multirate filtering method. A maximum energy optimisation of 20.3 % is considered as a very good result.

7 Coefficient Optimisation

This chapter presents the main reasons why coefficient optimisation can be used in order to design low-energy FIR filters. Two algorithms for coefficient optimisation are described and simulated, these being coefficient optimisation using steepest decent and coefficient optimisation using genetic algorithm.

For FIR filters implementations, switching activity leads to significant energy dissipation in the buses [Mehendale et al., 1998]. As these toggling bits are energy consuming, optimisation of the Hamming distance between the filter coefficients is wanted. The Hamming distance is a measure of switching activity between two consecutive signals, and is thus a measure of energy dissipation in buses. Hamming distance between two N-bit binary numbers is mathematically defined as:

$$HD = \sum_{i=0}^{N-1} a_i \oplus b_i \tag{7.1}$$

As an example of Hamming distance calculation, the two binary numbers 10010 and 11001 are considered. The Hamming distance is the number of bits that are different in these two numbers. In this case the Hamming distance is 3. The smaller the Hamming distance the more similar the binary represented numbers are. Similar binary numbers reduces the switching activity in the buses. This can be exploited in implementations, though some requirements on the architecture has to be defined. For implementation of FIR filters the coefficients are transferred by buses to computational units, such as multipliers and adders, as illustrated in figure 7.1. Requirements to the architecture, is that all the coefficients are gathered in one memory and proceeds directly after each other on a single bus. The more similar the coefficients are, the less switching activity is experienced on the coefficient bus, and thus the energy consumption is reduced. In an actual implementation, the filter coefficients may be stored in the same memory as the program instruction. In this case they might be transferred on the bus interleaved, and the reduced Hamming distance between the coefficients is expected to have no or little effect on the energy dissipation. This architecture is not beneficial for implementation in this project as it is crucial that the coefficients proceeds directly after each other on the bus.

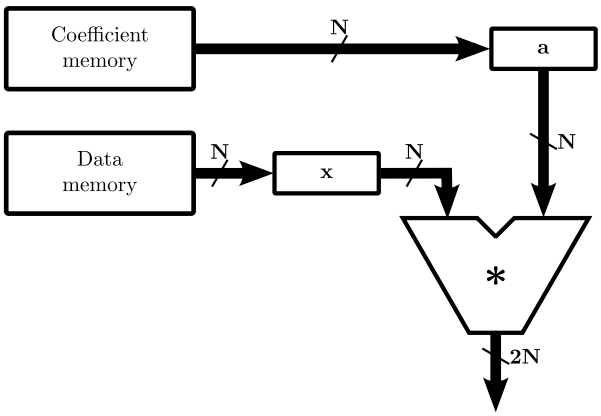

Figure 7.1: Architecture showing the buses to transfer coefficient to the functional operators.

Another measure that influence the energy consumption in buses is adjacent signal toggles (AST)[Mehendale et al., 1998]. An adjacent signal toggle is when two adjacent bits are opposite and both of them changes at the same time, i.e. changing from *10* to *01*. This measure is directly affected by the Hamming distance between the signals as the more similar the signals are, the less adjacent signal toggles are possible. The reason why AST affects the energy consumption is due to the crosstalk between the lines in the buses.

AST is defined mathematically as:

$$AST = \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} ((a[i][j] \oplus a[i][j+1]) \wedge (a[i][j] \oplus a[i+1][j]) \wedge (a[i][j+1] \oplus a[i+1][j+1])) \tag{7.2}$$

for M adjacent coefficients of N bits. The coefficients are defined by two index entries, where the first is the coefficient placement and the second is the bit placement.

Reducing the Hamming distance between the filter coefficients is therefore a method for potentially reducing the energy consumption in a FIR filter implementation. This chapter will present methods for minimising the Hamming distance between the coefficients in FIR filters.

In low-energy filter design, *optimisation* of the filter coefficients is a way of reducing the Hamming distance in between the filter coefficients, thus reducing the energy consumption. In this technique consider a N-tap filter fulfilling a filter specifications. Using this technique the original coefficients will be replaced by new coefficients with reduced Hamming distance, while still satisfying the filter specifications with a certain margin.

Optimising the filter coefficients is an iterative process, where the coefficients are modified in order to reduce the Hamming distance between them, and still comply with the desired filter characteristics. This can be seen as a COP (Combinatorial Optimisation Problem) since the optimal solution is a combination of several optimisation factors. A general description of COP is given in appendix A on page 157. In this COP the objective function includes the filter specifications and Hamming distance.

In the following two sections, two different algorithms using different approaches are described. First one is an approach using the steepest decent technique, suggested by [Mehendale et al., 1995]. The other approach uses genetic algorithm (GA), and is suggested by the project group. GA is presented in detail in appendix A on page 157.

7.1 Coefficient optimisation using steepest decent

In this approach the problem of finding the optimal filter coefficients is considered as a local search problem. Consider a N-tap FIR filter fulfilling the filter specifications. It is desirable to make adjustments to the coefficients, A_i, such that the Hamming distance between them are minimised, while the transfer function still fulfils the desired filter specifications. By considering the coefficient optimisation as a local search problem the optimal coefficient values are searched for in the neighbourhood of the original coefficients.

7.1.1 Algorithm

The flow of the algorithm is described in figure 7.2.

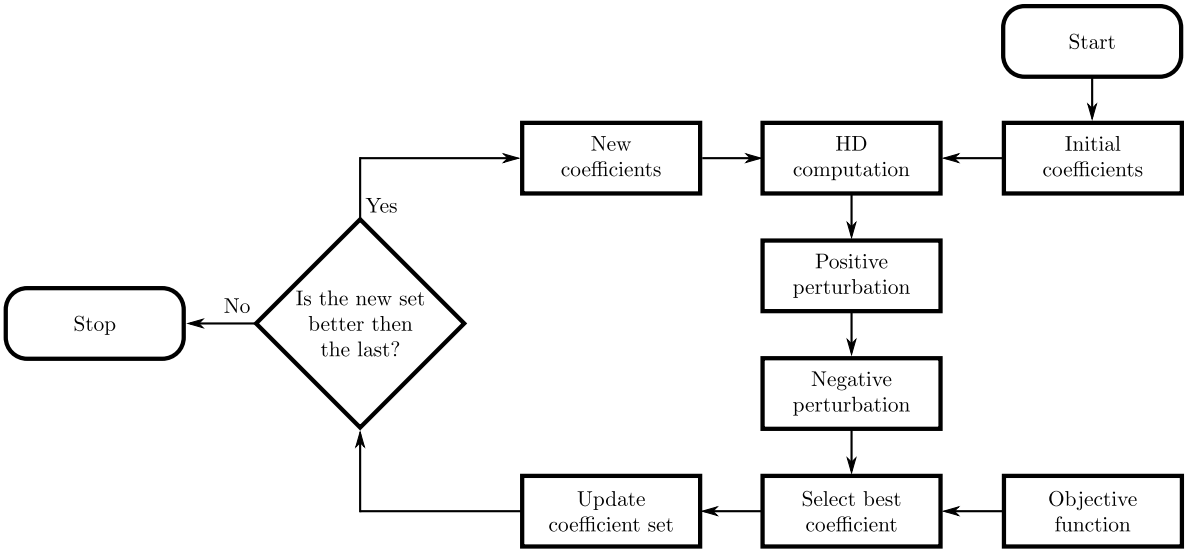

Figure 7.2: The flow diagram describing the coefficient optimising using steepest decent algorithm.

The first step of this algorithm is to calculate the total Hamming distance (HD) of the initial coefficients. Consider the original set of five coefficients:

$$A_0 \quad A_1 \quad A_2 \quad A_3 \quad A_4 \tag{7.3}$$

The total HD between these coefficients is found by:

$$TotalHD = A_0 \oplus A_1 + A_1 \oplus A_2 + A_2 \oplus A_3 + A_3 \oplus A_4 \tag{7.4}$$

Secondly; the coefficients are perturbed positive one by one, by adding LSB to the filter coefficients represented in 2's complement form. The perturbation is done until a new coefficient is found which will result in a reduced total HD between the new coefficient, A_{i+} and the original coefficients, A_i. The new coefficients are denoted as:

$$A_{0+} \quad A_{1+} \quad A_{2+} \quad A_{3+} \quad A_{4+} \tag{7.5}$$

The same procedure is executed to generate the negative perturbed set, A_{i-}:

$$A_{0-} \quad A_{1-} \quad A_{2-} \quad A_{3-} \quad A_{4-} \tag{7.6}$$

The change in the coefficient needs to be as small as possible in order to affect the filter characteristics minimally. In [Mehendale et al., 1995], there is not suggested any limit for adding or subtracting LSB, or a stop criteria if no better coefficient is found. The project group therefore inserted a count limit, which is the maximal number of allowed iterations. If a coefficient is incremented by LSB until it reaches the count limit, without reducing the total Hamming distance between the coefficients, the initial coefficient A_i will be used as new coefficient A_{i+}. Assuming that two coefficients are not the same, the HD between them can be minimised in at most $2^N - 1$ steps, by adding LSB to one of the coefficients. N is the word length of the coefficients. This meaning that the count limit should be maximum $2^N - 1$.

As the coefficients are represented in 2's complement, adding LSB to the highest positive value $(1 - \Delta)$ result in the most negative value (-1). Likewise, subtracting LSB from the lowest negative value (-1) will result in the highest positive value $(1 - \Delta)$. To avoid this, the coefficients are saturated at the highest and lowest possible value, in order to avoid that a large value is perturbed to a small value and visa versa.

The perturbation finds the nearest higher and nearest lower coefficient that reduces the total HD. An illustration of this is shown in figure 7.3. This figure shows the Hamming distance of the original coefficient and how the perturbed coefficients reduce the Hamming distance.

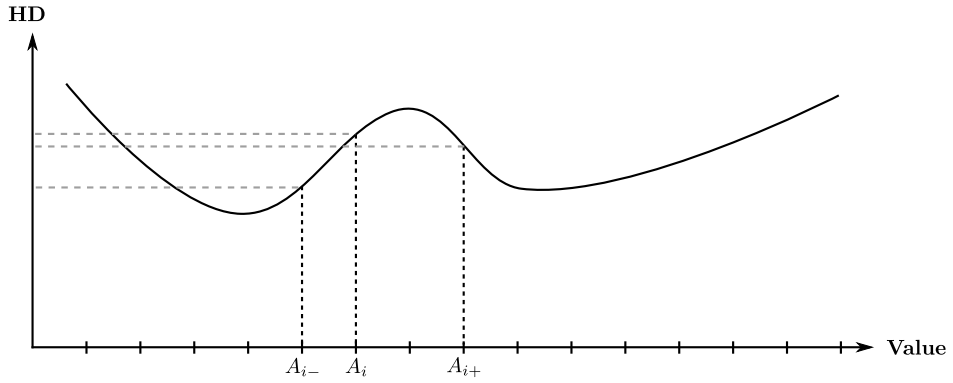

Figure 7.3: LSB is added to, and subtracted from the coefficient A_i in order to reduce the total HD.

After perturbation of the original coefficients, two new sets of coefficients are created, as seen in eq. (7.5) and (7.6).

The new coefficients are used to generate $2N$ new sets, in this example $2 \cdot 5$ new sets. The original coefficients are one at a time replaced by the new coefficients found by perturbation. This procedure generates $2N$ new sets of coefficients, during each iteration. From these $2N$ sets, the set that maximises a given objective function is chosen and further used as the new set for the next iteration. The objective function determines how good this new solution is, in terms of maintaining the filter characteristics and reducing the Hamming distance. The objective function is given by:

$$O = \left(\frac{Pdb_{req} - Pdb}{Pdb_{req}} + \frac{Sdb - Sdb_{req}}{Sdb_{req}} \right) \cdot HD_{red} \tag{7.7}$$

In this function the Pdb_{req} is the maximum accepted pass-band ripple and Pdb is the pass-band ripple achieved with the new set of coefficients. Sdb_{req} is the minimum accepted stop-band attenuation and Sdb is the stop-band attenuation achieved with the new set of coefficients. All these measures are illustrated in figure 7.4 on the next page. The measures are sat according to the initial filter. The pass-band ripple requirement determines the acceptable gain the filter should be within in the pass-band. As for the stop-band attenuation, this states the upper limit for the stop-band, all values below this are acceptable. In this case a negative number of higher numeric value than the limit will fulfil the requirement, while a smaller negative number will not. The HD_{red} is the Hamming distance reduction from the current set of coefficients to the new set.

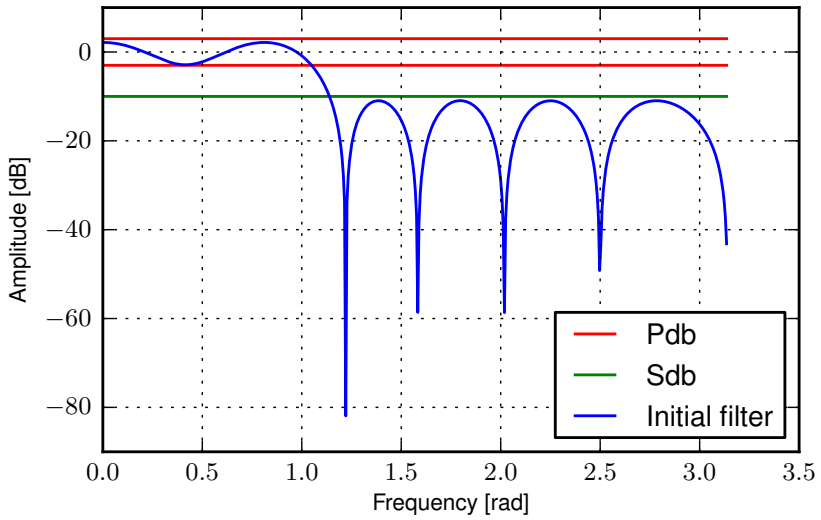

Figure 7.4: Filter with pass-band ripple and stop-band attenuation requirements.

How these measures are found can be seen in listing 7.1, where the filter has a pass-band from 0 to $\pi/3$ and stop-band from $\pi/3+0.1$ to π. Here it is presented how the pass-band ripple and stop-band attenuation for the new set of coefficients are found, and also the requirements these has to fulfil in order to proceed as possible solutions.

```
1  freq_new, response_new = freqz(coef_dec) # Calculate the frequency response
2  Pdb = 20*log10(abs(response_new[0:(512/3)-1])) # Calculate the amplitude of the pass-band
3  Sdb = 20*log10(abs(response_new[(512/2.6):512])) # Calculate the amplitude of the stop-band
4  hd_insert = hamming_distance(coef_insert) # Calculate the HD
5  hd_red = (1.-(float(hd_insert)/hd_orig))*100 # Calculate the HD reduction in percent
6  if max(Sdb) < Sdb_req and max(abs(Pdb)) < Pdb_req: # If the stop-band and pass-band is within
       specifications
7      tolerance = (Pdb_req - max(abs(Pdb)))/Pdb_req + (max(Sdb) - Sdb_req)/Sdb_req # Calculate
           the tolerance
8  else:
9      tolerance = 0 # Else, set the tolerance to zero when the filter does not comply with the
           specifications
10 obj_func_new = tolerance*hd_red # Calculate the objective function
```

Listing 7.1: Stop-band attenuation and pass-band ripple calculations.

The set which maximises the objective function is selected and replaces the initial coefficients. This fulfils the first iteration of the algorithm. The same procedure is followed for further improvement of the coefficients. The algorithm usually runs until no further optimisations can be done. Pseudo code for the algorithm is given in listing 7.2.

```
while optimisation can be done:
    calculate total HD of existing solution A_i (HD)
    for each coefficient A_i:
        while HD_new ≥ HD and within count limit
            add LSB to the current coefficient to generate A_{i+}
            calculate new HD (HD_new)
            add one to counter
    for each coefficient A_i:
        while HD_new ≥ HD and within count limit
            subtract LSB from the current coefficient to generate A_{i-}
            calculate new HD (HD_new)
            add one to counter
    for all new coefficients, A_{i+} and A_{i-}:
        insert one of the new coefficients in existing solution
            if Pdb and Sdb are within specifications
                calculate objective function
                keep best solution
                best solution = existing solution
```

Listing 7.2: Coefficients optimisation algorithm.

7.1.2 Simulation

The following section presents a simulation of the steepest decent algorithm for coefficient optimising. The simulation is done in order to investigate the reduction in HD achieved by this algorithm. The simulation is done for one filter only. Several parameters are tested, in order to observe their affect on the performance of the algorithm.

The initial filter is a 16th order low-pass filter, with pass-band from 0 to $\pi/3$ and stop-band from $\pi/3+0.1$ to π. The Python function *remez* is used to generate the filter coefficients. This function generates a Chebyshev filter by the use of Parks-McClellan method presented in appendix B. The gain of the filter is sat to be one in the pass-band and as low as possible in the stop-band. The initial filter is shown in figure 7.4 on the preceding page and as Python code in listing 7.3. Choosing a higher order filter than 16th order, leads to a more attenuated stop-band and less significant ripples in the pass-band, but for simulation purposes a filter order of 16 is found sufficient. During these simulations the symmetry of linear phase property has not been exploited in the optimisation algorithm, meaning that the coefficients are perturbed individually and not in pairs.

```
filt = ss.remez(17, [0, np.pi/3, np.pi/3+0.1, np.pi], [1, 0], Hz=2*np.pi)
```

Listing 7.3: Filter specifications.

The algorithm changes one coefficient in A_i each iteration, causing slow changes from the initial filter from iteration to iteration. The whole algorithm iterates as long as the Hamming distance between the filter coefficients can be minimised and the transfer function for the filter is within the stop-band attenuation and pass-band ripple requirements. In this test the Sdb_{req} is equal to -10 dB and the Pdb_{req} is sat to be ± 3 dB. There has been used a count limit of 2^{10}, to minimise the execution time. The count limit means that if no reduction of HD is found under perturbation within the limit, the algorithm stops perturbation of the coefficient.

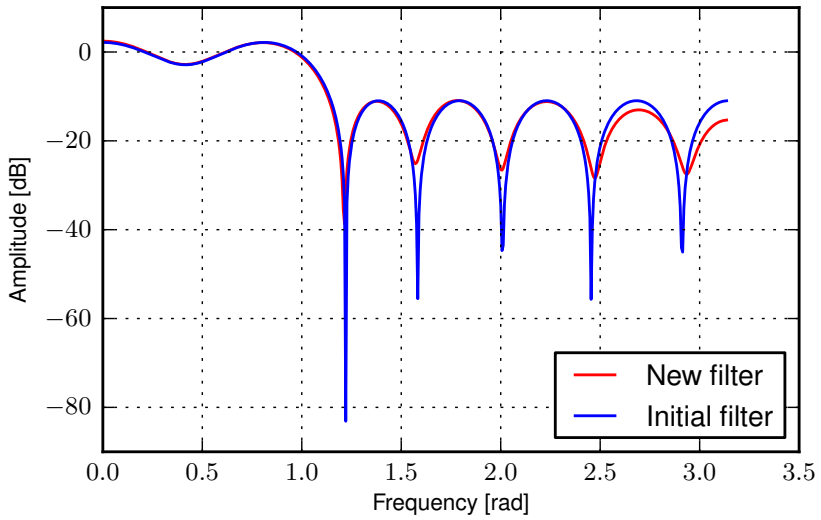

Figure 7.5: The initial filter and the filter with optimised filter coefficients, Sdb_{req} = -10, Pdb_{req} = $\pm$3.

Figure 7.5 shows the initial 16th order filter in blue and the optimised filter in red. This new filter does not differ significantly from the initial filter in terms of pass-band ripple and stop-band attenuation, due to the relatively weak requirements to the Pdb_{req} and Sdb_{req}. The results in Hamming distance reductions is shown in listing 7.4.

```
1 Pdb_req:      3
2 Sdb_req:      -10
3 Initial HD:   126
4 Best HD:      43
5 Optimised:    65.87%
```

Listing 7.4: Output from coefficient optimising algorithm, Sdb_{req} = -10, Pdb_{req} = $\pm$3.

The initial filter has a Hamming distance between its coefficients of 126. By the use of this algorithm the HD can be reduced to 43, corresponding to a reduction of 65.87 %.

7.1.3 Count limit considerations

A factor worth evaluating is the count limit. This determines the interval in which the algorithm is able to search for a new coefficient which reduces the HD. With a high count limit the algorithm searches a larger interval for finding coefficients which reduces the HD, it is therefore expected that a large count limit can reduce the HD more than a small count limit.

As discussed earlier, the coefficient which minimises the HD can be found in maximum $2^N - 1$ step, by adding LSB in every step. By using a 16 bit coefficient representation, the maximal count limit can be sat to 2^{15}. Figure 7.6 shows how the optimisation evolves as the count limit is increased with a fixed filter specification.

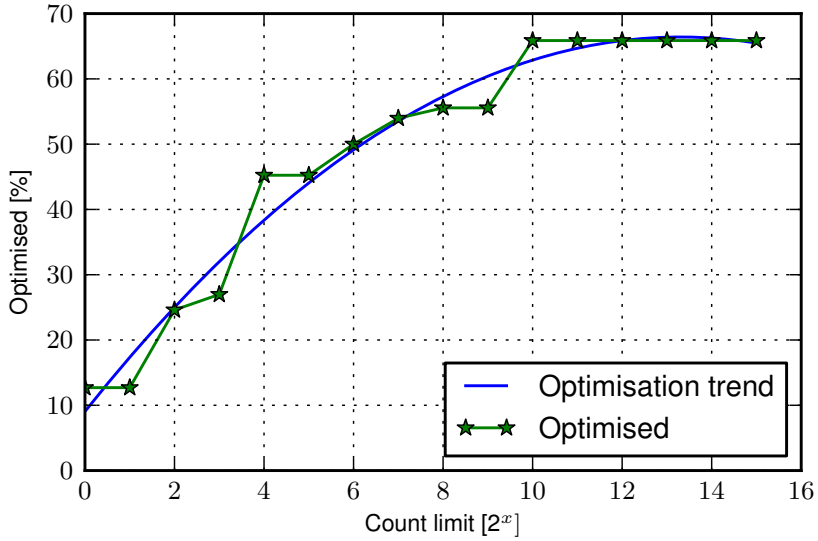

Figure 7.6: Optimisation as a function of count limit.

As seen in figure 7.6 the optimisation increases as the count limit increases and then tops at 66 %. As the count limit reaches 2^{10} the HD is not further optimised, this might be due to the saturation preventing the coefficients to change from maximal positive to maximal negative, or opposite. It may also be, that no further optimisation is possible.

As the count limit is increased the time for executing the algorithm increases. The perturbation of the coefficients is the most exhaustive part of the algorithm and while the interval of perturbation is increased, so is the execution time.

Next section is a description of how the coefficient optimisation can be done using GA.

7.2 Coefficient optimisation using GA

The general description of GA is given in appendix A on page 157, this should be read for general understanding of the algorithm and explanation of the terminology used here.

The use of GA for coefficient optimising has several advantages. It can be used both to find coefficients complying to the filter specifications and optimising them in terms of reducing the Hamming distance. As starting point, an initial population is found consisting of randomly selected filters. The coefficients in this population will be optimised using GA. Filter specifications like linear phase property, pass-band ripple and stop-band attenuation has to be included in the fitness computations. In this case the fitness computation must also include functionality that evaluates the Hamming distance between the filter coefficients. The total fitness function can be written as a function of the Hamming distance and the filter specifications:

$$F = g(spec, HD) \tag{7.8}$$

With a random selected population, the fitness is calculated and selection, crossover and mutation is done according to the description in appendix A. As the algorithm iterates, the filters in the population converges towards an optimal solution and will be more and more similar until there are several optimal filters in the population. The GA cycle is executed until the population is within the specifications of the filter and the Hamming distance is minimised, or to a predefined number of iterations is met [Haupt and Haupt, 2004].

7.2.1 Algorithm

This section will describe the algorithm in more detail, using parts of the generated code and pseudo code in order to describe the different functionalities of the algorithm. In this presentation of the algorithm the GA cycle will be used as an outset.

The ideal filter used in this algorithm is the same as for coefficient optimisation using the steepest decent algorithm, defined in listing 7.3. Compared to this filter we are trying to find optimal coefficients which fulfils the filter specifications and at the same time minimises the Hamming distance between the coefficients. The initial population consists of 200 filters, with randomly chosen filter coefficients.

After generating the initial population, fitness evaluation is done of all the filters. The function used to evaluate the fitness is called *feval* and is realised with the Python code in listing 7.5.

```
1  def feval(hd, nbits, pop):
2      hamd = []
3      error = []
4      fitness = []
5      for i in range(len(pop)): # For all filters in the population
6          w, h = ss.freqz(pop[i]/sum(pop[i])) # Calculate the frequency response
7          hamd.append(hamming_distance(pop[i], nbits)) # Calculate the HD
8          error.append(sum(abs(abs(hd)-abs(h)))) # Calculate the error as the absolute
               difference between the initial filter and this one
9          fitness.append(1./(error[i]+hamd[i])) # Calculate the fitness
10     return fitness, hamd, error # Return the parameters found in the for-loop
```

Listing 7.5: Fitness evaluation.

The fitness evaluation function uses the population, *pop*, the filter response of the initial filter, *hd*, and the number of bits in each coefficient, *nbits*, as inputs. In the fitness evaluation both the Hamming distance between the coefficients and the error compared to the ideal filter is calculated. The fitness is found as a sum of these two factors. In the algorithm these are weighted equally. The possibility of emphasising one of the factors more than the other is considered and several weights has been tested, but is not found to give a significant difference in the result.

In the selection part of this algorithm, both elite selection and roulette wheel methods are used. Only two filters are chosen by the elite selection and the rest are selected by the roulette wheel method. The elite selection selects the two strings with the highest fitness value and puts them into the next population. The function for solving the roulette wheel selection is shown in listing 7.6. Before this function is called, the fitness of the filters in the population is cumulative added up and stored in the list *comsum*. *comsum* is the input to the roulette wheel function. In the roulette wheel function a random number within the 0 and the maximum number in the *comsum* list is found. Thereafter, the corresponding filter for this value is found and it's index value is returned.

```
1  def wheel(comsum):
2      rannum = sp.rand()*comsum[-1] # Find a randum number within the range of the comsum list
3      j = 0 # Initialise the variable j
4      while comsum[j] < rannum: # While the j'th number in the consum list is less than the
           random number
5          j = j+1 # Increase j by one
6      return j # Return the index (j) of the filter which has been selected
```

Listing 7.6: Roulette wheel selection.

The crossover function coded in Python can be seen in listing 7.7. This function has both parents as input as well as the number of bits the coefficients are represented by. Likewise are the crossover and mutation rates input to the function. The amount of crossover and mutation are specified by these rates. In this algorithm the crossover rate

has been 0.6 and the mutation rate has been 0.01, as suggested in [Goldberg, 1989]. The function selects a random point in the parent, if the crossover rate allow it to, and combine the two parents from this point. This function also calls the mutation function, if allowed by the mutation rate. The mutation chooses a random point in the child and inverts the current bit.

```
1  def crossover(parent1, parent2, bits, pcross, pmutation):
2      parent1, parent2 = encode(parent1, bits), encode(parent2, bits) # The two parents are
           encoded to a binary representation
3      child1, child2 = [], [] # The children are initialised
4      child1.extend(parent1) # The first child is created as a copy of the first parent
5      child2.extend(parent2) # The second child is created as a copy of the second parent
6      if flip(pcross) == 1: # The flip command returns 1 as a function of the probability
           defined by the crossover rate
7          cpoint = pl.randint(0, len(parent1)-1) # The crossover point is defined as a random
               index
8      else:
9          cpoint = len(parent1)-1 # The crossover point is sat as the last index, which
               generates no crossover
10     for i in np.arange(0, cpoint): # The first part of the children are swaped
11         child1[i] = mutation(parent1[i], pmutation) # The bits are mutated with a probability
               defined by the mutation rate
12         child2[i] = mutation(parent2[i], pmutation)
13     for i in np.arange(cpoint, len(parent1)): # The second part of the children are swaped
14         child1[i] = mutation(parent2[i], pmutation) # The bits are mutated with a probability
               defined by the mutation rate
15         child2[i] = mutation(parent1[i], pmutation)
16     return decode(child1,bits), decode(child2,bits) # The children are decoded to base 10 and
           returned by the function
```

Listing 7.7: Crossover function.

This is continued until a new population is generated. The new population replaces the old, and the number of iterations is increased by one. This procedure is followed until the predefined maximum number of iterations are reached.

7.2.2 Simulation

The following section presents the results of the simulation of the algorithm described above. With an outset in the 16th order lowpass filter generated by Parks-McClellan algorithm by the function *remez* in Python, the algorithm has been executed in order to reduce the HD between the filter coefficients. By using GA the result will vary from time to time, as most of the initial population is chosen randomly and the result of the selection also will vary. This is the nature of heuristic algorithms.

Fitness calculations in this algorithm consists of a combination of the error between the new filters and the ideal filter and the HD reduction. While the fitness only increases, both the HD and the error fluctuates. How the error and HD evolves over the iterations can be seen in figure 7.7. The error is the difference between the ideal filter and the best filter from the population in the GA algorithm. The fitness is shown in figure 7.8. In this simulation 1000 iterations has been used.

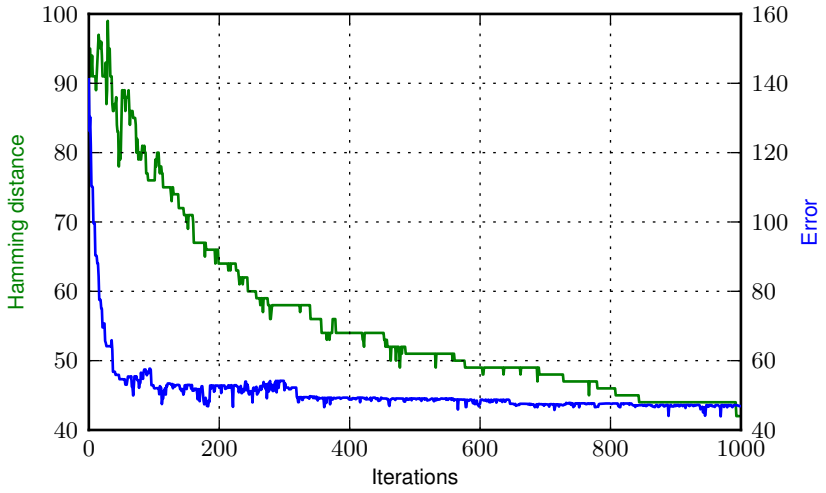

Figure 7.7: HD and error progress over iterations.

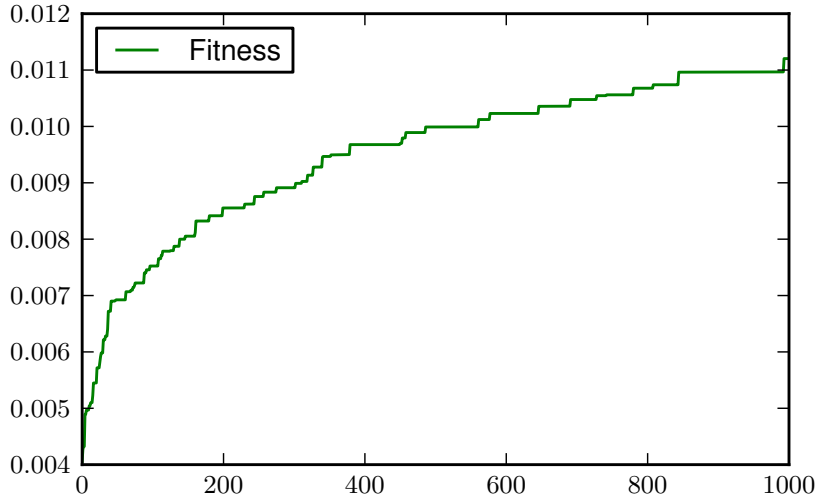

Figure 7.8: Fitness progress over iterations.

Listing 7.8 shows the output from the GA. With starting point in a Hamming distance of 126, this can be reduced to 42 by the GA. This corresponds to a reduction of 67 %. The ideal and final filters are shown in figure 7.9.

```
1 Initial HD:  126
2 Final HD:    42
3 Optimised:   66.67%
```

Listing 7.8: Output from coefficient optimising GA algorithm

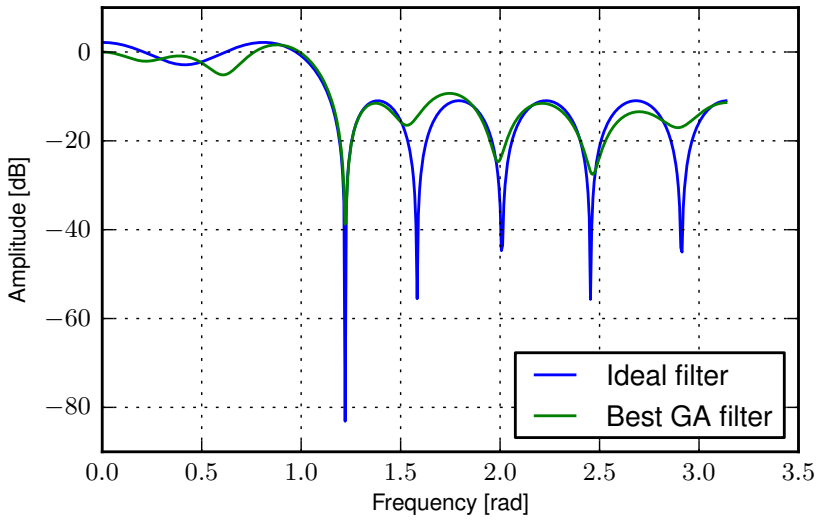

Figure 7.9: Ideal filter and final filter found by GA.

7.3 Conclusion

The following section will summarise and conclude on the coefficient optimisation methods described above.

This chapter has been concerning coefficient optimisation using either a steepest decent algorithm proposed by [Mehendale et al., 1998] and modified by the project group or a genetic algorithm proposed by the project group. The steepest decent algorithm work by perturbation of an initial set of coefficients in order to find a new set of coefficients which suites the desired filter specifications but with a lower HD. The genetic algorithm works by altering a random population of coefficient sets. This is done by using crossover and mutation operations, in order to end up at a set of coefficients, which fits an ideal filter as good as possible but with a lower HD.

The main difference between the two algorithms is that the steepest decent algorithm was given some desired specifications where the GA tries to fit some known filter response. The steepest decent algorithm uses a set of known filter coefficients fulfilling the desired filter specifications, which is then changed so the HD is lowered. In the other case, GA starts from a random picked population, which is processed until the filter characteristics are close to a desired response while the HD is as low as possible.

The algorithms gave in the above executed examples, using a 16 order low-pass filter, a result where the HD was lowered by respectively 66% and 67%. It was though noted, that the frequency response of the two resulting filters are very different. So in the end, two different approaches has been programmed and executed, and both approaches gives approximately the same result. This shows that both approaches are valid and can therefore be used to optimise a given filter or filter specification in the sense of minimising the HD.

8 Coefficient Scaling

This chapter introduces coefficient scaling as a way of reducing the HD between the coefficients in a filter, and thereby reduce the switching activity, thus the energy dissipation of the filter. An algorithm for doing this is designed and the results of this algorithm are presented.

The principle of coefficient scaling is to uniformly scale the coefficients in order to reduce the Hamming distance (HD) between consecutive coefficients. Reducing the HD between the filter coefficients, reduces the switching activity on the buses, thus reducing the energy dissipation. Reduced energy dissipation lead to low-energy filter realisations. Coefficient scaling is a technique that advantageously can be used in combination with coefficient optimisation. Since all the coefficients are scaled uniformly, the improvement is not expected to be significant, but this technique can be used to improve the starting point of the coefficient optimisation. Scaling the whole filter preserves the filter characteristics like specifications of pass-band ripple and stop-band attenuation, but result in an overall magnitude gain or attenuation equal to the scaling factor. By including the scaling factor K in the equation for the filter:

$$Y_n = \sum_{i=0}^{N} (A_i \cdot X_{n-i}) \tag{8.1}$$

it can be seen that this scaling factor is multiplied with all filter coefficients as well as the output:

$$K \cdot Y_n = K \cdot \sum_{i=0}^{N} (A_i \cdot X_{n-i}) = \sum_{i=0}^{N} ((K \cdot A_i) \cdot X_{n-i}) \tag{8.2}$$

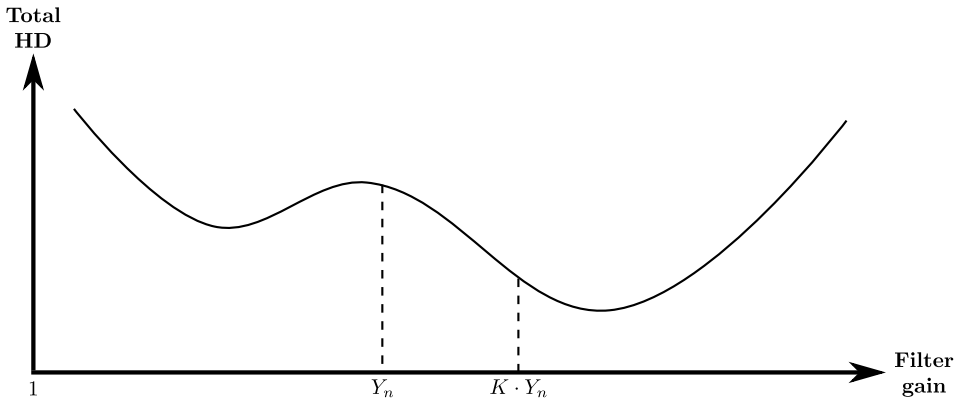

Figure 8.1: Principle of coefficient scaling.

Coefficient scaling can be used when a scaling factor exist, which will reduce the total HD between the filter coefficients. Figure 8.1 shows the principle of coefficients scaling. Y_n is the initial gain of the filter, having a certain total HD between its coefficients. With a gain equal to one, the sum of the coefficients is also one. The goal of coefficient scaling is to find the value K that uniformly scale the coefficients such that the total HD is minimised.

For scaling the coefficients an iterative search algorithm, which inserts different scaling factors and calculates the total HD, can be used to find the best factor. The algorithm might run for a predefined number of iterations or until a minimum is found. The algorithm might be given an allowable range of scaling, for instance ± 3 dB gain, to find the optimal scaling factor within [Mehendale et al., 1998]. This allowable range prevents the overall magnitude to change more than acceptable, but is also used to limit the signal-to-noise ratio (SNR). The SNR is affected, when the gain of the filter is changed. The main source of noise is assumed to be due to the error which appears when

quantising after multiplication or accumulation. The error is e.g. introduced when two N bit strings are multiplied and the result is a 2N bit string, which is usually quantised to N bit to fit the bus. This quantisation is usually either rounding or truncation. If the introduced change in the gain lowers the signal so much, that the quantisation error becomes a dominant factor, the filter performance will suffer from this. The lower limit of the search range may thus be defined to prevent the SNR to become too small.

8.1 Algorithm

The following section will introduce a general description of how the algorithm is build and how it performs. The algorithm is coded in Python and can be found on the enclosed CD. This section will use parts of the code to describe how the algorithm works.

The algorithm is build as a search algorithm over the scaling factor, K, which is changed within an interval to find the lowest possible HD. The interval at which K is changed, is specified as the maximum deviation which is accepted in the gain of the filter. As an example, the maximum deviation in gain can be set to ± 3 dB, at which the interval will be:

$$\begin{aligned} \text{interval} &= [-3;3]\,\text{dB} \\ &\approx [0.5;2]\,\text{times} \end{aligned}$$

The step-size used to step through the interval can be defined in many possible ways. This can be big to perform a quick search or small to do a more extensive and thorough search. The step-size is here defined as the smallest number which can be represented by the number of bits, denoted *nbits*. If *nbits* is set to e.g. 16 bit, the step-size would be:

$$\begin{aligned} \text{step-size} &= \frac{1}{2^{nbits-1}} \\ &= \frac{1}{2^{16-1}} \end{aligned}$$

The algorithm has to have a filter to conduct the search algorithm upon. In this algorithm the filter is specified by using the *remez*-function in Python. The function uses the Parks-McClellan algorithm to generate the optimal Chebyshev FIR filter with a given set of characteristics. By using this as the filter to optimise the HD on, the algorithm iterates over the loop seen in listing 8.1.

```
1 while 1/10**(maxdev/10) ≤ K ≤ 10**(maxdev/10): # While scaling factor is within the gain
      limits do
2     hd_new = hamming_distance(filt*K, nbits) # Compute the new HD
3     if hd_new < hd: # If the new HD is better than the previous
4         hd = hd_new
5         filt_new = filt*K # Set filter coefficients to be initial coefs*K
6     K = K+stepsize # Increase K with step-size
```

Listing 8.1: Main loop in the search algorithm.

On listing 8.1 the *while*-loop is true as long as K is inside the interval. Inside the loop we calculate the HD for the filter coefficients multiplied by K. If this distance is better than any of the previous values, the HD is saved and so is the filter coefficients. In the end we add the step-size in order to get a new K.

The HD is calculated by the function *hamming_distance*, which is defined as on listing 8.2.

```
1 def hamming_distance(filt, nbits):
2     filt_encoded = encode(filt, nbits)
3     hd = 0
4     for i in range(len(filt_encoded)-1):
5         hd = hd+sum(b1 != b2 for b1, b2 in zip(filt_encoded[i], filt_encoded[i+1]))
6     return hd
```

Listing 8.2: Function for calculating the Hamming distance.

The function takes two variables as input, a list containing the filter coefficients and an integer which defines the number of bits used to represent the coefficients. At first the filter coefficients are encoded into bit strings of the given length. The HD variable *hd* is then initialised. The function then starts by calculating the HD between the current coefficient and the next one in the list, and adds the distance to the variable *hd*. When the end of the list is reached, the function returns the total HD.

8.2 Simulation

The following section will present the results from the simulation of the above described algorithm.

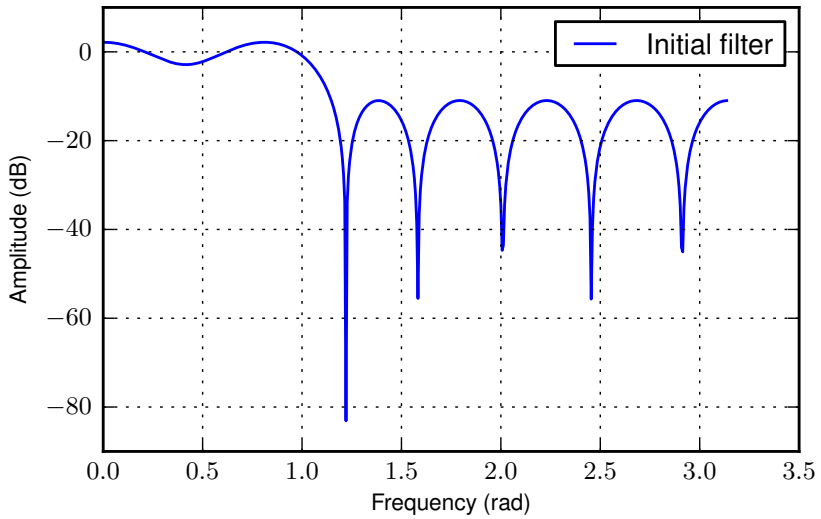

Figure 8.2: The initial filter used in the coefficient scaling method.

At first let us see how the initial filter in this example looks like. This is shown on figure 8.2 where the filter is realised with the following parameters:

```
1 filt = ss.remez(16+1, [0, np.pi/3, np.pi/3+0.1, np.pi], [1, 0], Hz=2*np.pi)
```

Listing 8.3: The parameters used to define the initial filter.

This is a 16th order filter, with pass-band from 0 to $\pi/3$ and stop-band from $\pi/3+0.1$ to π. The desired gain is 1 in the pass-band and as low as possible in the stop-band. The filter coefficients are found by using the Python function *remez*, which uses the Parks-McClellan algorithm to find an optimal Chebyshev filter. The Parks-McClellan algorithm is described in appendix B on page 163.

It is interesting to see how the HD is varying over each iteration, as this gives a view on how it is optimised. This is shown on figure 8.3.

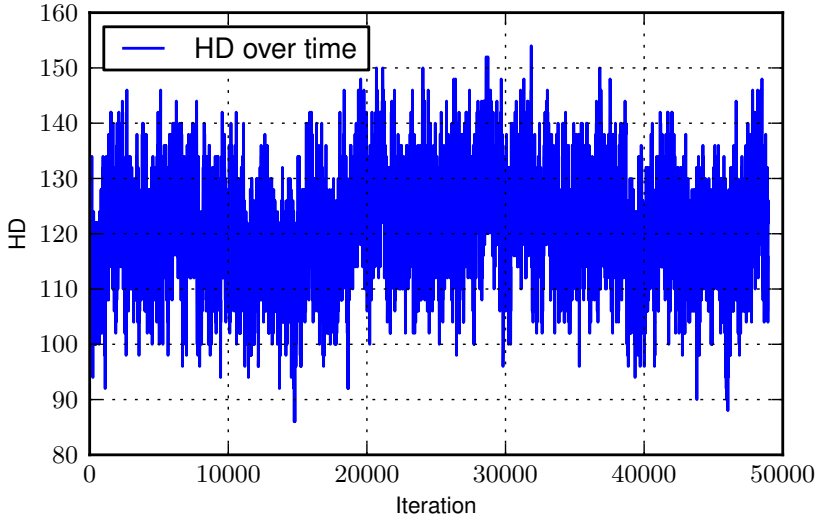

Figure 8.3: The Hamming distance for every iteration of the algorithm.

As can be seen on figure 8.3 the HD is fluctuating rapidly which is expected, as the scaling factor, K, changes the binary representation of all the coefficients. Every time the scaling factor is changed, a new set of coefficients are generated, which leads to a new HD. The output from the algorithm is the following:

```
1  Initial HD:  126
2  New best HD: 116
3  New best HD: 110
4  New best HD: 108
5  New best HD: 106
6  New best HD: 104
7  New best HD: 98
8  New best HD: 96
9  New best HD: 94
10 New best HD: 92
11 New best HD: 86
12 ----------------
13 Best HD:     86
14 Optimised:   31.75%
```

Listing 8.4: Output from the coefficient scaling algorithm.

which prints the initial HD of the filter, the falling HD over the iterations, the final HD and a percentage of how much the HD is optimised. Here it is seen that the initial HD is 126 and the final is 86, which gives a total optimisation of approximately 32 %.

The new filter can then be compared with the initial one, which is seen on figure 8.4.

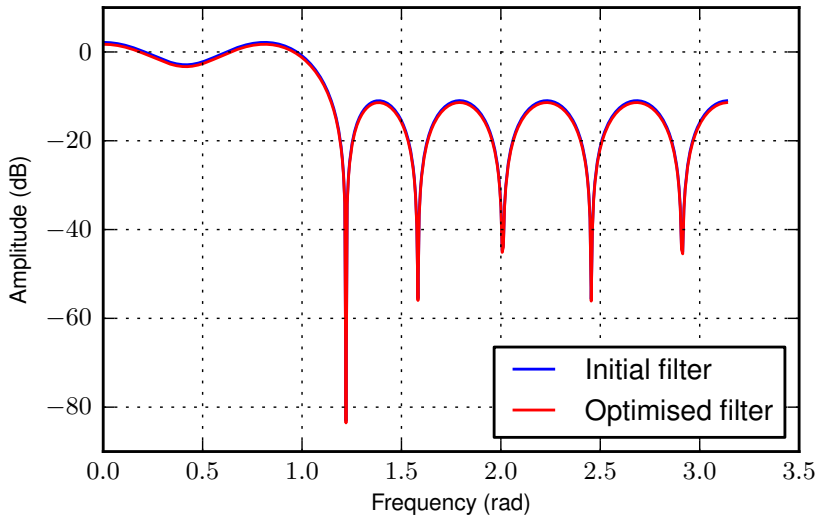

Figure 8.4: A comparison between the initial and the optimised filter.

It is seen, that the new filter has the same frequency characteristics as the initial one, but is moved slightly down in the graph, which means that it has an overall lower gain.

8.3 Additional considerations

The following section will show a more thorough analysis of the algorithm and its parameters. This is done as to investigate how the different parameters of the algorithm affects the results and outcomes from the algorithm.

The first thing to investigate, is how the length of the filter impacts the overall optimisation of the HD. A graph showing the improvement of HD as a function of filter order, when keeping the other filter specifications fixed, is seen on figure 8.5. The remaining parameters of the algorithm are the same as previous described.

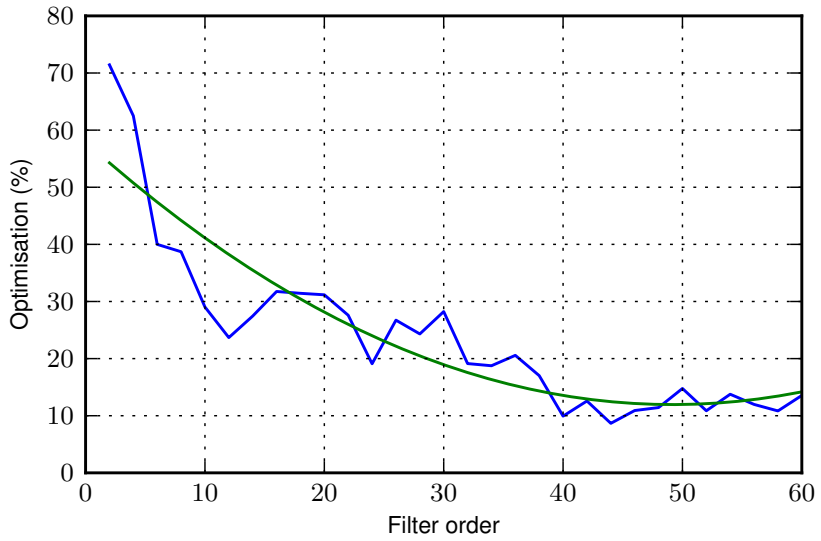

Figure 8.5: The correlation between the improvement in HD versus the filter order. The green line shows the second order regression.

On figure 8.5 it is seen that the overall improvement in HD is falling with rising filter order. This can be explained as for every new filter coefficient the optimisation problem is becoming a little more complex, which results in a less improvement in the HD.

Changing the cut-off frequency will not change the possible percentage of optimisation, as can be seen on figure 8.6. The data shown on the figure is fluctuating between 30 and 50 percent, except for a couple of values. These values are presumed to be generated as the scaling algorithm gets lucky to find a constant which makes the coefficients very similar.

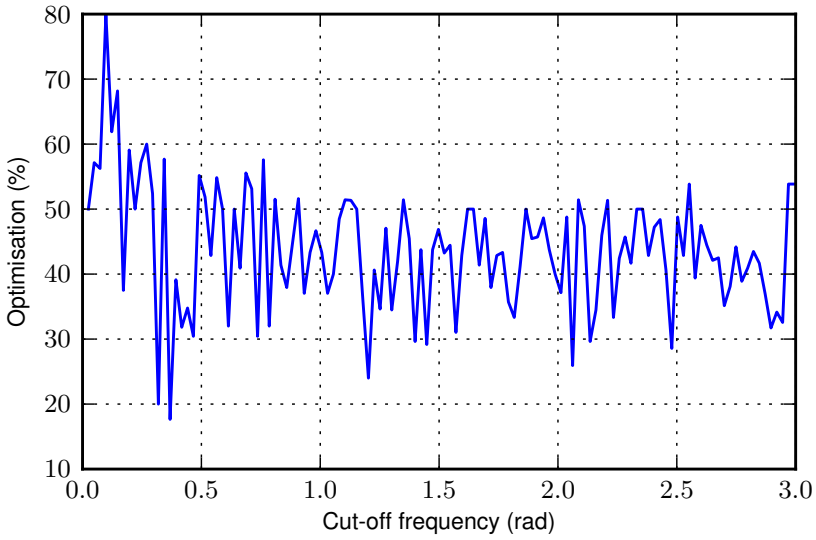

Figure 8.6: The percentage of optimisation as a function of the cut-off frequency.

By increasing the maximum deviation of the gain to an increasing amount of decibel will improve the HD optimisation. This comes clear as the search span is becoming bigger and bigger, while the step-size is kept fixed. With a greater search span it is more likely to find a more optimal solution. It is though noted, that while increasing the search span and keeping the step-size fixed, the time of the search is increasing massively. It is also noted that it probably is very few applications which allow a large deviation of the gain.

If the step-size is varied and the deviation kept fixed it is observed that a lower step-size gives a bigger improvement in HD. This can be seen on figure 8.7.

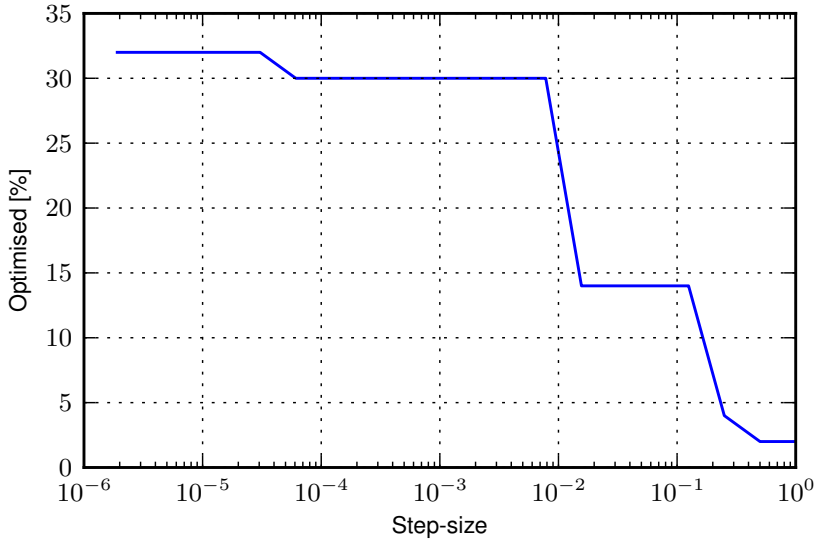

Figure 8.7: The correlation between the improvement in HD versus the step-size.

This comes clear in the same way as increasing the deviation, as the search span is increased. This however, is not increasing the interval of the search, but the number of points evaluated in the interval. The time of the search will again increase massively, as the number of points to evaluate increases.

8.4 Conclusion

The following section will summarise and conclude on the coefficient scaling method described above.

The coefficient scaling principle is used here by creating an algorithm which search for a scaling factor within a given interval which minimises the HD. The interval is here given by a maximum deviation from the original filter response, which is a requirement to the desired filter specification. The interval is covered by introducing a step-size which is the value added to the scaling factor at each iteration.

This rather simple way of reducing the HD generates a result where the HD is minimised by almost 32 % with a filter order of 16. This is seen as a very good result, as the whole concept and algorithm is that simple. It should be noted, that the percentage of optimisation drops for higher filter order and smaller deviation interval.

9 Coefficient Ordering

In this chapter the principle of coefficient ordering as a method of designing low-energy FIR filters, is presented. Two algorithms for ordering the coefficients are designed and tested, these being coefficient ordering using simulated annealing and genetic algorithm.

Computing the output of a FIR filter is done by a sum of products. Summation is both commutative and associative, which means that it is independent of the order of adding the products together. As an example, the output from a 4th order filter can be computed independent of the order additions. The following two equations produce the same output.

$$Y_n = A_0X_n + A_1X_{n-1} + A_2X_{n-2} + A_3X_{n-3} + A_4X_{n-4}$$

$$Y_n = A_3X_{n-3} + A_4X_{n-4} + A_0X_n + A_1X_{n-1} + A_2X_{n-2}$$

The order of the summation determines the order of the coefficients appearing on the coefficient memory data bus. The summation is independent of the order of computing, when we look at floating-point calculations. If the summation is done in fixed-point, there will be introduced an error due to the difference in the order the quantisation is performed, but we will assume that this will not be of significant size. Since the order of the summation can be changed, this can be exploited in order to reduce the Hamming distance (HD) between consecutive coefficients [Mehendale et al., 1998]. The problem of finding the optimal coefficient order will be addressed here.

Finding the optimal order of the filter coefficient can be seen as a travelling salesman problem. In the travelling salesman problem a salesman has to visit N cities once and only once, travelling the shortest possible route. In the case of ordering the coefficients before summation, each coefficient is considered as a city. This is illustrated in figure 9.1.

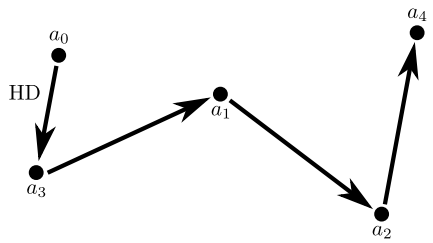

Figure 9.1: The travelling salesman problem, using filter coefficients as cities.

The coefficients are preselected and the task is to find the order which minimises the total distance between the cities. The distance between the cities is in the filter coefficient case, the HD between the numbers. The total distance between all cities is minimised when the total HD between the filter coefficients is minimised. There are many algorithms developed to solve the travelling salesman problem, most of them involves reordering the sequence of cities (coefficients) in an iterative process, to reduce the distance.

The following two sections will describe how simulated annealing (SA) and genetic algorithms (GA) are used to solve the travelling salesman problem.

9.1 Coefficient ordering using SA

SA is used to find near-optimal solutions for an optimisation problem. The basic idea of SA is to escape from local-minima by allowing solutions which are worse than the previously found. This is also known as hill-climbing and is illustrated on figure 9.2.

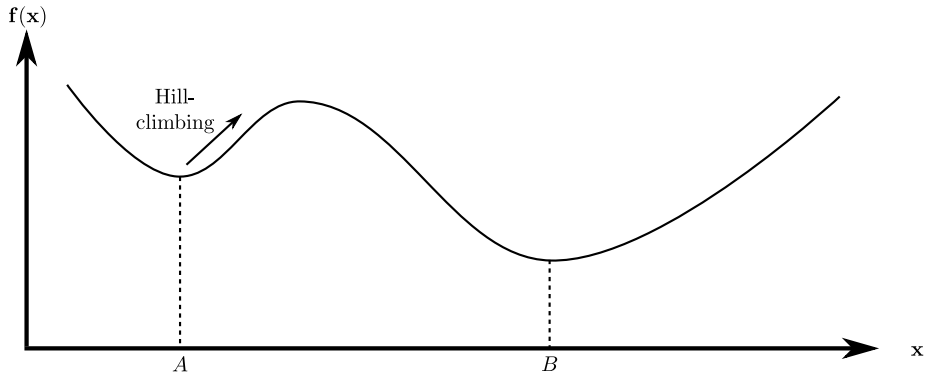

Figure 9.2: How simulated annealing escapes local-minima by hill-climbing.

On figure 9.2 it is seen, that to escape from the local-optimal point A, it is necessary to allow bad solutions in order to climb over the hill to end up at the global-optimal point B. In SA the action to use a bad solution is chosen by a certain probability following Boltzmann distribution, which is described as [Blum and Roli, 2003]:

$$exp\left(-\frac{f(s')-f(s)}{T}\right) \tag{9.1}$$

where s is the best found solution so far, s' is the newest found solution and T is called temperature. The temperature is decreased or cooled during the iterations of the algorithm by a so called cooling schedule. The temperature controls the progress of the algorithm, as a high temperature gives a high probability of accepting a bad solution, while a low temperature gives a low probability. The cooling schedule controls the temperature and can be defined in many ways. One of the must used schedules is:

$$T_{k+1} = \alpha T_k \tag{9.2}$$

which gives an exponential falling temperature, as α usually is below 1. The schedule does not even has to be monotonic and can therefore be defined as a piecewise function, which could both increase, decrease or be fixed at a certain value. The schedule is optimised by empirical adjustment from application to application. In the following, the exponential falling schedule is the only one considered.

SA is inspired from the annealing process of metals. As metal is heated the atoms becomes free and moves randomly in the material, this is much like the case in SA, where a high temperature makes the solution of the SA wander randomly around in the search space. As metal is cooled slowly the atoms gets ordered in a structure which holds a low internal energy. This is analogous to lowering the temperature in SA, thereby increasingly only choosing solutions with lower costs, which means that it converges towards a minima. The pseudocode of the SA algorithm is seen on listing 9.1.

```
s = initial solution
T = T0
while termination condition not met do
    s' = generate new solution from solution space
    if (f(s')<f(s)) or (exp(-(f(s')-f(s))/T > random number) then
        s = s'
    update(T)
```

Listing 9.1: SA algorithm outline.

As can be seen in the pseudocode of the algorithm, an initial solution is chosen either at random or by making a qualified guess. The temperature is thereafter initialised to a starting value, T_0. The *while*-loop which is the backbone of the algorithm runs for as long as a termination condition is not met. This condition can be e.g. maximum number of iterations or a temperature. A new solution, s' is generated by a transition function, which finds a new solution from the solution space. The transition function in the sense of coefficient ordering, will be a function which alters the ordering in some way. If this new solution is better in the sense of an objective function f, it is chosen as a new best solution. If it is not a better solution, the solution can still be chosen if the probability defined from eq. (9.1) is greater than a random number in the interval $[0,1]$. Before a new iteration is started, the temperature is updated.

9.1.1 Algorithm

The following section will introduce a general description of how the algorithm is build and how it performs. The algorithm is coded in Python and can be found on the enclosed CD. This section will use parts of the code to describe how the algorithm works.

The main loop of the algorithm is described by pseudocode in listing 9.1 and rewritten as Python code in listing 9.2.

```
path = np.arange(filter_order+1)
dist = hamming_distance(filt, nbits, path)
T = Tinit
while T > Tstop:
    path_new = movep(path)
    dist_new = hamming_distance(filt, nbits, path_new)
    if (dist_new ≤ dist) or (np.exp(-abs(dist-dist_new)/T)>sp.rand()):
        dist = dist_new
        path = path_new
    T = T*cool_rate
```

Listing 9.2: The main loop of the coefficient ordering SA algorithm.

The filter which is optimised is found by using the Parks-McClellan algorithm which is employed in Python by the function *remez*. In listing 9.2, the filter which is found by *remez* is denoted *filt*, and is as an example chosen to be a 16th order low-pass filter. The initial path or ordering of the coefficients is chosen to be the one found by the *remez* function. This ordering is denoted as a path starting from coefficient 0 going to coefficient 1, next to coefficient 2 and so on. The HD is initially calculated from this path by using the same HD function used in coefficient scaling in listing 8.2 on page 53.

The parameters for the algorithm has been found to be an initial temperature of 100 and a lower limit temperature of 0.1, which is used as the termination condition of the *while*-loop. The parameters has been chosen by empirical search and gives a good result to this problem. The first function to be used inside the loop, is the *movep*-function which is the transition function of the SA algorithm. The function can be seen in listing 9.3.

```
1 def movep(path):
2     i = pl.randint(0, len(path)-1)
3     j = pl.randint(0, len(path)-1)
4     item = path[i]
5     path = np.delete(path,i)
6     path = np.insert(path,j,item)
7     return path
```

Listing 9.3: Random rearranging of the coefficient ordering in SA algorithm.

The function generates two random integers, i and j, denoting two indices of the path. The item at index i is then moved to index j and the new path is then returned to the algorithm. The algorithm next calculates the HD of this new path and makes a decision whether to save this new solution, as described earlier. At the end of the loop, the temperature is updated. The new temperature is calculated by multiplying the previous temperature and a cooling rate, which here is sat to 0.999. This value is found by an empirical search, where this value gave consistent good results from.

9.1.2 Simulation

The previously described algorithm and parameter values are used in the following simulation to describe the results created by the algorithm. It should be mentioned before any results are presented, that since the rearranging of the coefficients is done randomly, the results are varying from execution to execution.

The filter used in this test is a 16th order filter generated using the *remez* function in Python. The filter specifications are the same as for the filter used in the simulation of the other HD reduction methods. A view of how the HD is varying over each iteration is shown in figure 9.3.

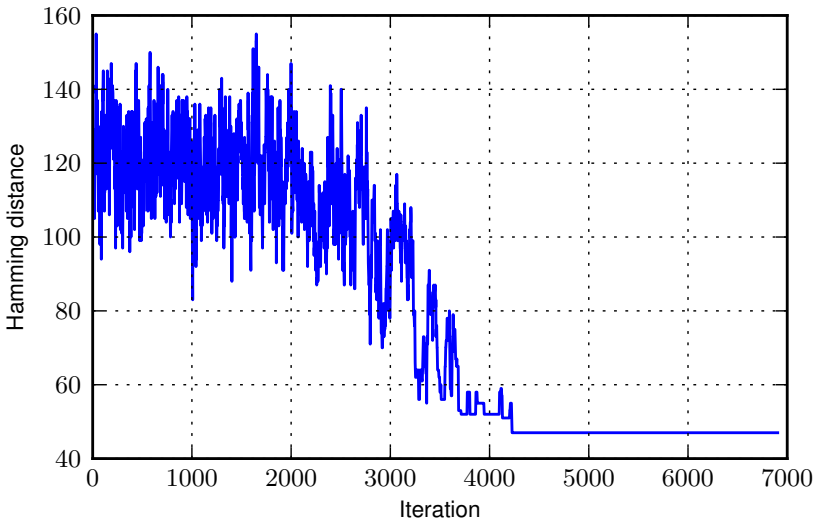

Figure 9.3: The varying HD over the iterations.

As can be seen, the HD is fluctuating rapidly in the first 2000 iterations. This is due to the relatively high initial temperature combined with a cooling factor close to one. When the temperature becomes closer to zero, the fluctuations gets smaller and the HD starts converging towards a steady state.

The temperature as a function of the iterations is shown in figure 9.4.

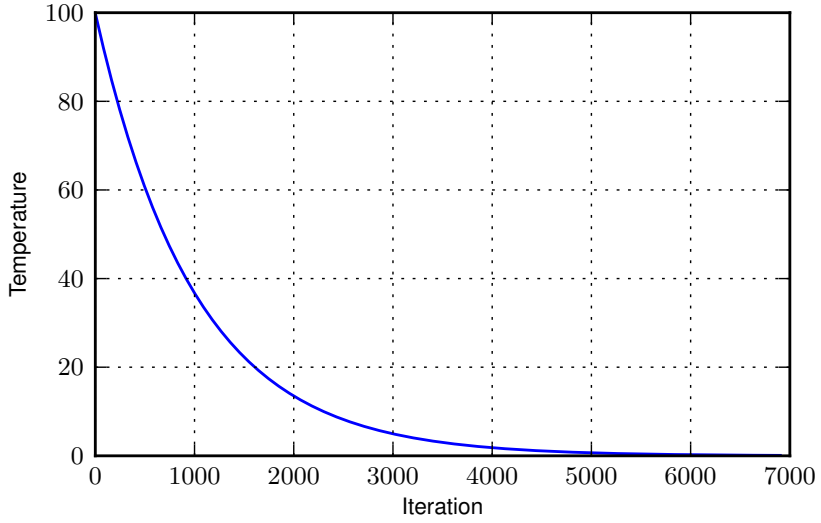

Figure 9.4: The temperature as a function of the iterations.

As it can be seen on figure 9.4, the temperature is falling exponentially from the initial value until it stops at the lower limit.

The output from the algorithm is as follows:

```
1 Time:         2.369 seconds
2 Initial HD:   126
3 Final HD:     47
4 Optimised:    62.7%
```

Listing 9.4: The output from the coefficient ordering SA algorithm.

This tells that the initial HD is calculated to be 126, the final HD is 47, which in the end gives an optimisation of almost 63 %. This optimised filter has the same characteristics as the initial one, but has a total HD which is about 63 % better than initial. This might lead to a more energy efficient filter implementation, which is investigated in a later chapter. The algorithm finished in just 2.3 seconds on a standard dual-core laptop.

9.1.3 Additional considerations

The following section will take a look at the parameters used in the SA algorithm, in order to map which parameters that are important to adjust to get a good solution.

The first thing to look at, is the temperature parameters, which are the cooling rate and the initial and stop temperature. The first thing to explore is how the optimisation of HD is affected by the cooling rate. The temperature is changed using the cooling schedule defined in eq. (9.2). The experiment is ran by keeping the other parameters fixed and then vary the cooling rate. This is seen in figure 9.5.

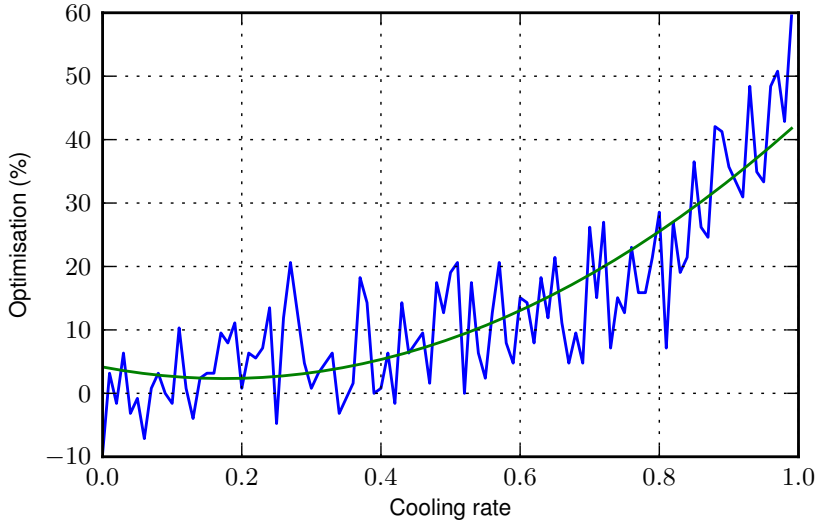

Figure 9.5: The optimisation of the HD compared to the cooling rate. The green line is the second order regression.

Due to the randomness included in the algorithm, the value of HD optimisation is expected to fluctuate. However, it is clear to see a trend in figure 9.5. The closer the cooling rate is to the value one the higher optimisation percentage. This can be explained, since a slower decreasing temperature gives more time to explore the search space.

In the same way, it is possible to see how the initial and stop temperature affects the final HD, which is seen on figure 9.6 and 9.7 respectively.

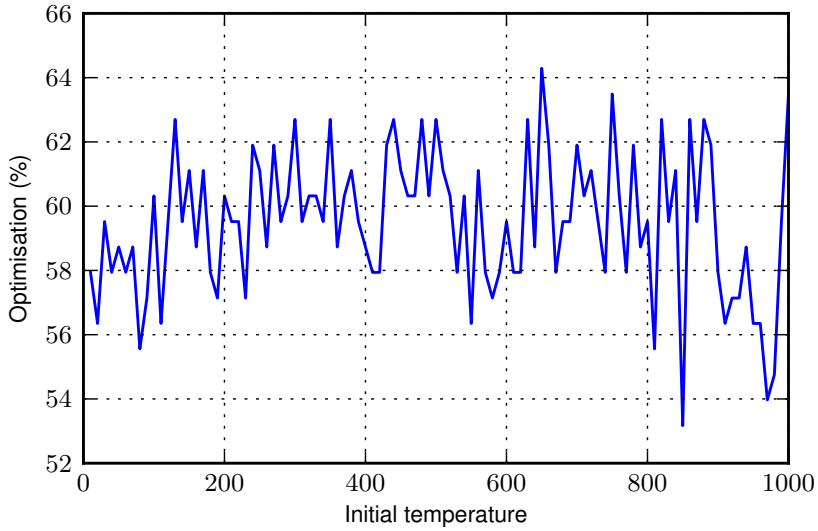

Figure 9.6: The optimisation of the HD compared to the initial temperature.

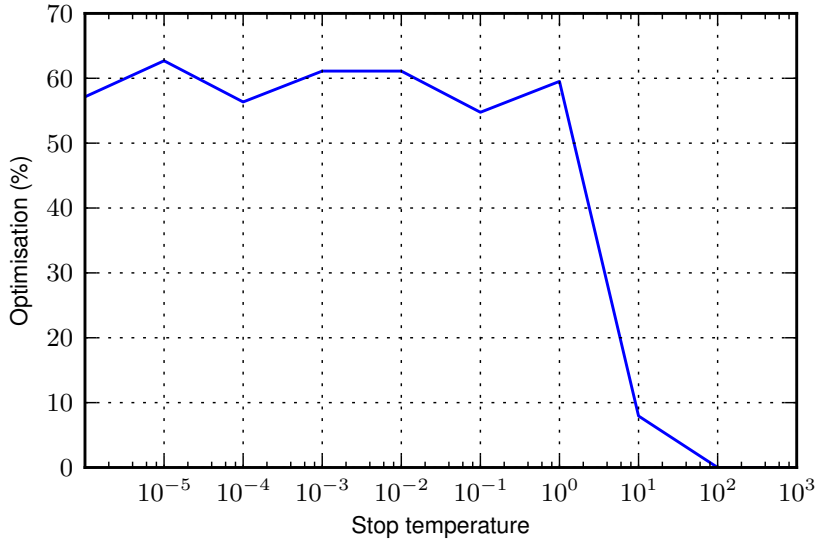

Figure 9.7: The optimisation of the HD compared to the stop temperature.

From both figures, no clear trend can be observed. It is though observed, that a high stop temperature affects the final HD in a bad way. This can be explained as the converging trend that SA uses, does not gets to fully converge before the process is stopped. This will clearly give a worse result.

9.2 Coefficient ordering using GA

Coefficient ordering can also be implemented using GA, which is presented in the following.

The general GA is presented in appendix A on page 157, however some modifications has to be made in order to use it to solve a travelling salesman problem. In this case, modified versions of the genetic operators crossover and mutation are used. The initial population consists of structures which includes different orders of computing the summations, randomly selected. In this case it is assumed that the filter coefficients in the filter are generated according to the desired filter characteristics, such that it is only the order of summation that changes. Fitness computations are only concerning the HD between the coefficients, where the lower HD will result in a higher fitness value. Selection is done in the same manner as previously described, where the fittest structures are more likely to be selected, and the mating creates pairs of structures for reproduction. Using GA for coefficient ordering each structure only consists of one string, which holds the order at which the filter coefficients are to be used. Figure 9.8 illustrates the structure of the population.

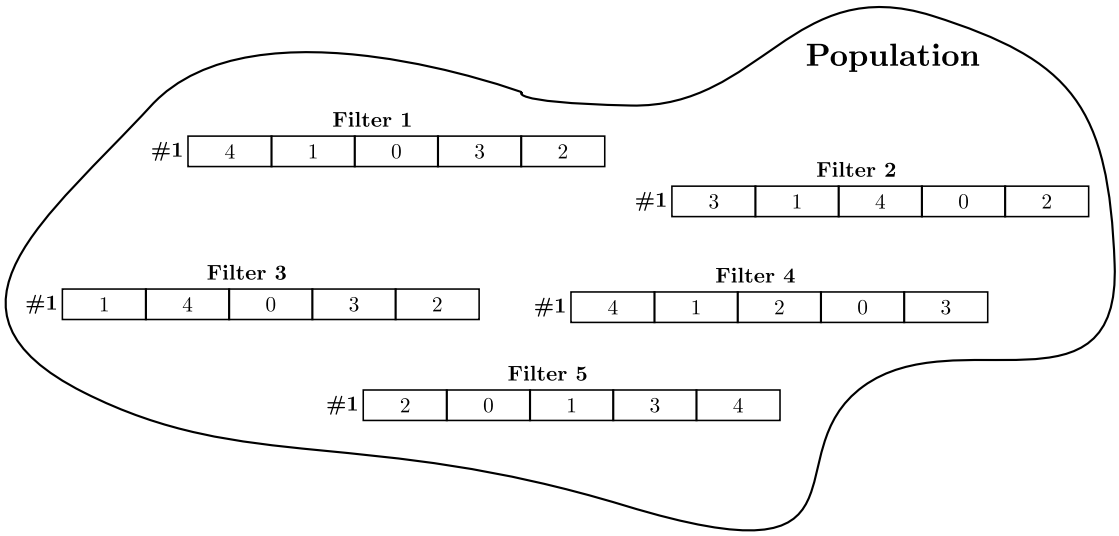

Figure 9.8: The structure of the population used in GA for coefficient ordering.

The structures on figure 9.8 are made so that the summation in e.g. the 4th order *Filter 1* is performed by calculating:

$$Y_n = A_4X_{n-4} + A_1X_{n-1} + A_0X_n + A_3X_{n-3} + A_2X_{n-2}$$

Since no coefficient number can be in the same structure twice, modified genetic operators has to be used. The modified version of the crossover operator used in this situation is called *order crossover*. The following will illustrate the crossover operator using a 10th order filter as an example.

Using this type of crossover, two random points in the two mating strings are selected and the values in between are copied to two new children. The remaining values of the first mate is ordered in terms of the second mates ordering and then inserted into the child. The same procedure is performed for the second mate. In table 9.1, the flow of the order crossover is illustrated.

	Initial	**Step 1**	**Step 2**
Parent 1	0 1 2 3 4 5 6 7 8 9	0 1 2 3 4 5 6 7 8 9	0 1 2 3 4 5 6 7 8 9
Parent 2	9 8 7 6 5 4 3 2 1 0	9 8 7 6 5 4 3 2 1 0	9 8 7 6 5 4 3 2 1 0
Child 1	* * * * * * * * * *	* * * **3 4 5 6 7** * *	**9 8 2** 3 4 5 6 7 **1 0**
Child 1	* * * * * * * * * *	* * * **6 5 4 3 2** * *	**0 1 7** 6 5 4 3 2 **8 9**

Table 9.1: Principle of order crossover

The two initial strings, *Parent1* and *Parent2* chosen for reproduction includes two different ways of ordering the ten coefficients. The ordering is chosen for illustrative purposes. In the first step two random points are found, and the values in between are copied to a new child. As two parents are chosen, two children are created in the first step. The second step takes the remaining numbers from the string, orders them by the other parent, and puts them into the new child. For *Parent1*, the remaining numbers are 8, 9, 0, 1 and 2. The numbers will then be reordered: 1, 0, 9, 8 and 2, as this is the order which the numbers appear from the same point in *Parent2*. The final two children are seen in table 9.1.

The mutation operator will also work in another way than previously described, because if one value in the string is randomly changed to another value, the result would probably have the same value two times. The solution is by randomly selecting one of the values and exchanging it by another randomly chosen value in the string. The remaining part of the GA, runs as described in appendix A.

9.2.1 Algorithm

The following section will describe the GA, which is used for coefficient ordering. The section will describe the GA in terms of the GA cycle which is shown in figure A.4 on page 159. The section will include samples of the code, to show some of the more important or interesting features of this algorithm. The full code is found on the enclosed CD.

The GA cycle starts by creating an initial population. This is done in the algorithm by filling a matrix with rows consisting of the numbers between zero and the filter order in a random ordering. This can be seen in listing 9.5.

```
1 pop = [] # Empty population
2 for i in range(popsize): # Runs until the population is full
3     row = [] # Empty row
4     while len(row) < filter_order+1: # Runs until the row is full
5         item = pl.randint(0, filter_order+1) # Generate a random integer
6         if item not in row: # If the integer is not in the row, it is appended to this.
7             row.append(item)
8     pop.append(row) # The row is appended to the population
```

Listing 9.5: The creation of the initial population for the coefficient ordering GA algorithm.

The next step of the GA cycle, is to evaluate the population by calculating the fitness of each row of the matrix. This is done by using the *hamming_distance* function shown in listing 9.6.

```
1 def hamming_distance(filt, nbits, path):
2     filt_encoded = encode(filt, nbits)
3     hd = 0
4     for i in range(len(filt_encoded)-1):
5         hd = hd+sum(b1 != b2 for b1, b2 in zip(filt_encoded[path[i]], filt_encoded[path[i+1]])
              )
6     return hd
```

Listing 9.6: The *hamming_distance* function in the coefficient ordering GA.

The function takes three arguments: *filt* which contains the filter coefficients, *nbits* which is the number of bits used in the binary representation and *path* which holds the ordering of the coefficients. The function iterates over the encoded coefficients, calculating the HD between the present and the next coefficient. The HD has to be calculated for every row in the population matrix and the fitness is thereafter found by calculating the reciprocal of the distance.

After the fitness computation, the algorithm has to select structures for the mating pool, but before this an elite selection is performed to guarantee that a few of the best structures are included in the new population. The selection is hereafter executed using a roulette wheel selection, described in section A.1.1 on page 160, which gives the fittest structures the highest probability of selection.

The genetic operators used in this algorithm are crossover and mutation operators. Further description of these is not considered necessary as the principle of these are described earlier in this chapter.

9.2.2 Simulation

This section will show the results generated by the simulation of the GA algorithm. Like with the SA algorithm the following results will vary, as the GA is random in nature, thereby leading to new results from execution to execution.

The GA will likewise optimise the HD of a 16th order low-pass filter generated by Parks-McClellan algorithm through the *remez*-function in Python. The cut-off frequency will be at $\pi/3$ and the filter coefficients will be represented by 16 bits. The GA parameters are sat to the following:

```
maxit = 300 # Number of iterations
popsize = 200 # Population size
elitsize = 2 # Elitist size
pmutation = 0.0001 # Mutation rate
pcross = 0.6 # Crossover rate
```

Listing 9.7: The GA parameters used for coefficient ordering.

The parameter values are found by an empirical search, where the different values are varied and the result is processed. The values chosen for this GA are found to be very effective and consistent. The HD and fitness over the iterations can be seen on figure 9.9.

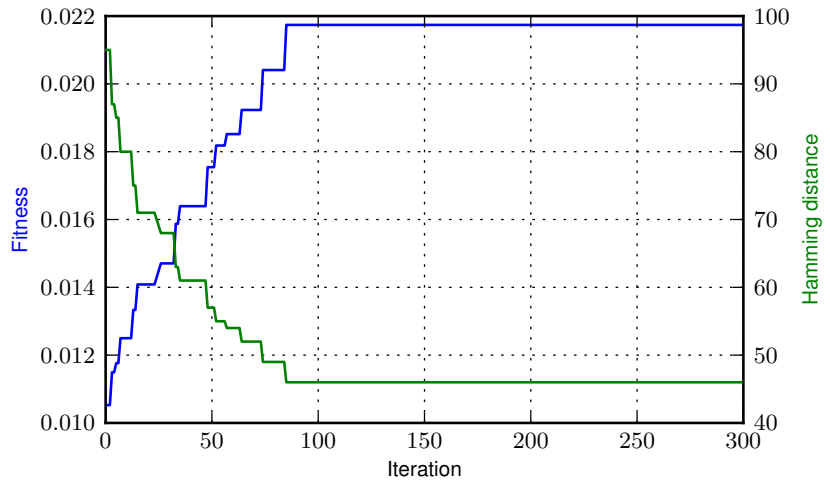

Figure 9.9: The varying HD and fitness over the iterations.

It is seen that the fitness and HD is connected, as the fitness is the reciprocal of the HD. Due to the elite selection used in the GA, the fitness is not fluctuating as the best filter ordering will always be present in the population. The final HD is found after less than 100 iterations, so the *maxit* parameter can in theory be halved. It is though a good idea to set the maximum iterations to a higher number, to be sure that the algorithm has converged to a final number.

The output from the algorithm is:

```
Time:         13.394 seconds
Initial HD:   126
Final HD:     46
Optimised:    63.49%
```

Listing 9.8: The output from the coefficient ordering GA.

Which tells that the algorithm has finished in approximately 13 seconds and gives an optimisation of 63 %. This optimisation is approximately the same as the SA algorithm. This should however again be compared to the random nature of these algorithms, so the SA will at some point get as good performance as the GA.

9.3 Conclusion

The following section will summarise and conclude on the coefficient ordering method described above.

The coefficient ordering method is used here by creating two different algorithms, SA and GA, which both solves the same problem. The reason for creating two different algorithms is to see if any difference between the two can be observed. The coefficient ordering principle rearranges the order of which the filter is calculated, to give a lower HD. The coefficient ordering does not change the filter response when the calculations are done in floating point. The situation may be different when a fixed point architecture is used, but is not assumed to be a problem.

Both algorithms works with a certain randomness, which means that the results will not be the same from execution to execution. The GA works by using the crossover and mutation operators, which works on a random population. The SA algorithm uses the theory of hill-climbing to escape from local minima and end up in a global minimum.

Both algorithms ends up with a total HD optimisation of about 63%, which also tells that the two algorithms are much alike as no significant difference is seen in the results. A difference is however seen in the execution time, which is about 5 times lower for the SA algorithm. This is however not an important difference since both times is within 15 seconds for a 16th order filter, which is rather quick.

10 Combined HD Reduction

This chapter combines all the methods presented concerning HD reduction in one simulation. The reason for combining them is to find the maximal possible HD reduction. This is here presented using the filter example used through the previous chapters, which is a 16th order low-pass filter with cut-off in $\pi/3$.

All previously described methods for reducing HD can be combined in order to maximise the HD reduction. This has been done for the reason to evaluate the total HD reduction when using coefficient optimisation, scaling and ordering. The algorithm starts out with creating the initial filter, created with the Python function *remez*, like described earlier. When combining the methods, the scaling is done prior to the optimisation and ordering, as this has been found to give the best overall HD reduction. The scaling is done used a step-size of $\frac{1}{2^{16-1}}$ and the maximal deviation in gain is set to ± 3 dB. The scaling improves the HD with 31.75 %, from a total HD of 126 to 86. This result can be seen in listing 10.1, which shows the output of the combined algorithms. The scaling improves the starting point of the coefficient optimisation.

Directly after the scaling the coefficient optimisation is executed. For the coefficient optimisation two algorithms were presented, designed and evaluated. They resulted in approximately the same optimisation. In this combined version the steepest decent approach is used. The steepest decent algorithm for coefficient optimisation uses the optimised coefficients from the scaling as the input of coefficients. The coefficient optimisation algorithm iterates as described in chapter 7.1. Although the scaling has reduced the HD between the coefficients significantly, some further optimisation of the HD happens. The coefficient optimisation algorithm reduces the HD with 2.33 % from 86 to 84, as seen in listing 10.1. The optimisation done by he steepest decent algorithm in this case is significant lower than what was achieved in the previous test on the initial coefficients created by the *remez*-function. The reason for this is that the coefficients are already optimised by scaling.

Coefficient ordering is the final element of the combined algorithm for HD reduction. This orders the coefficients from the optimisation in the order that minimises the HD between the successive filter coefficients. Also for coefficient ordering there were presented two algorithms. The coefficient ordering algorithm used here is made using simulated annealing.

```
1  Coefficient scaling:
2     Number of iterations: 48957
3     Initial HD:           126
4     Best HD:              86
5     Optimised:            31.75%
6  Coefficient optimisation:
7     Number of iterations: 2
8     Initial HD:           86
9     Best HD:              84
10    Optimised:            2.33%
11 Coefficient ordering:
12    Number of iterations: 6904
13    Initial HD:           84
14    Best HD:              34
15    Optimised:            59.52%
```

Listing 10.1: Result of combining all HD reduction methods.

From listing 10.1 it can be seen that the coefficient ordering reduced the HD from 84 to 34, which corresponds to a reduction of 59.52 %. In total the filter coefficients gets an overall HD reduction of 73.02 %. This result will vary from time to time as the coefficient ordering is made using SA, and will therefore give different results every time it is executed.

The difference between the optimised and original filter frequency response is seen in figure 10.1.

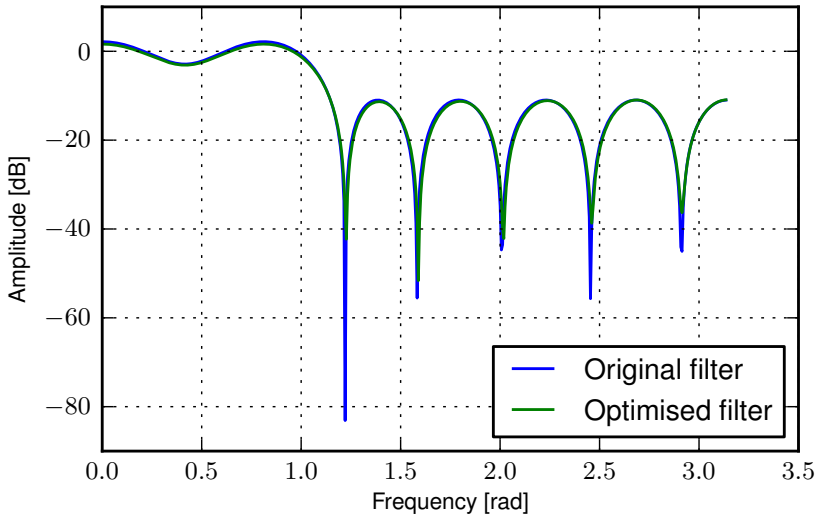

Figure 10.1: Comparison between the original and optimised filters.

It is seen that the difference in frequency response is very small even though the HD is 73 % smaller. The gain of the optimised filter is 0.54 dB lower than the original filter, which means that the output from the optimised filter will be lower. If the gain is considered to be too big, one suggestion is to reduce the acceptable gain used in the coefficient scaling algorithm or to reduce the count limit used in the coefficient optimisation algorithm. By reducing the count limit, one limits the possible change in the coefficient.

By investigating the frequency response of the original and optimised filter coefficients, the difference between them have been shown. The difference is depending on the parameters used in the HD optimisation methods. As the HD can be reduced by 73 %, it is assumed that this will lower the energy consumption of a filter implementation. Following chapters will conducts tests in order to find out whether the reduction in HD does corresponds to a reduction in energy.

11 Evaluation Set Considerations

This chapter provides an evaluation of how the cut-off frequency affects the HD between the coefficients in a filter. This is done in order to verify that the evaluation set in chapter 3 on page 13 is representative when evaluating the coefficient optimisation methods. The project group has found it necessary to conduct this evaluation to justify the choice of filters in the evaluation set.

Table 11.1 shows the specifications for the filters used in the evaluation. For this evaluation low-pass, high-pass and band-pass filter will be used. Only one filter order will be used, in order to limit the evaluation. The transition band is sat to 0.1 rad and the filters phase property is linear. The band-pass filter will have a bandwidth of $\pi/3$ rad. All the coefficients are encoded into 2's complement binary numbers with 16 bits.

Filter type	**Low-pass filter**	**High-pass filter**	**Band-pass filter**
Filter order	120		
Transition bandwidth	0.1 rad		
Phase property	Linear phase		
Bandwidth	-	-	$\pi/3$ rad
Coefficient bits	16		

Table 11.1: Filter characteristics for evaluating the effect of the cut-off frequency has on HD.

The three different 120th order filters are shown in figure 11.1, having a cut-off frequency of 1 rad.

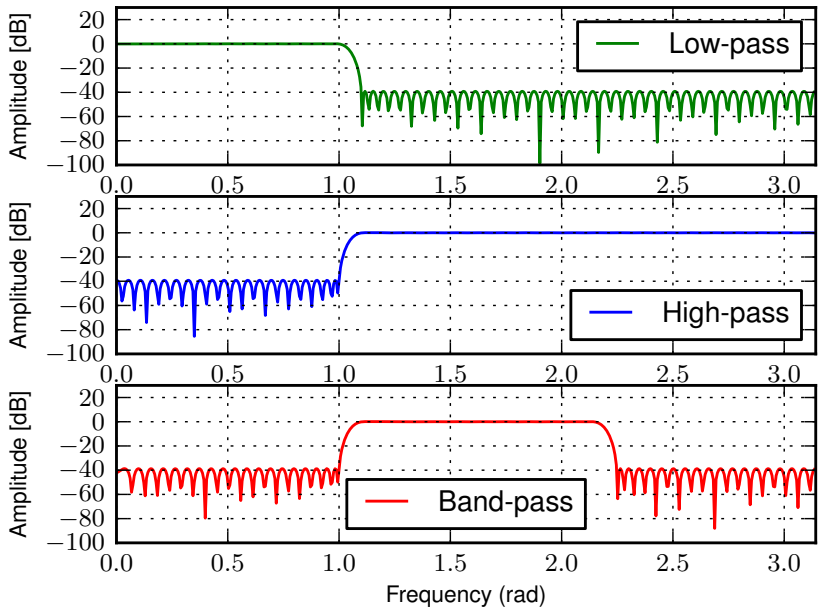

Figure 11.1: Filters used in the evaluation of the HD's connection to cut-off frequency.

The cut-off frequency will now be varied in steps of 0.01 rad from 0 to $\pi/3-0.1$ rad and for every cut-off frequency, the HD calculated. The HD is still defined as:

$$HD = \sum_{i=0}^{N-1} a_i \oplus b_i \tag{11.1}$$

Figure 11.2 shows the HD evaluation as the cut-off frequency is increased.

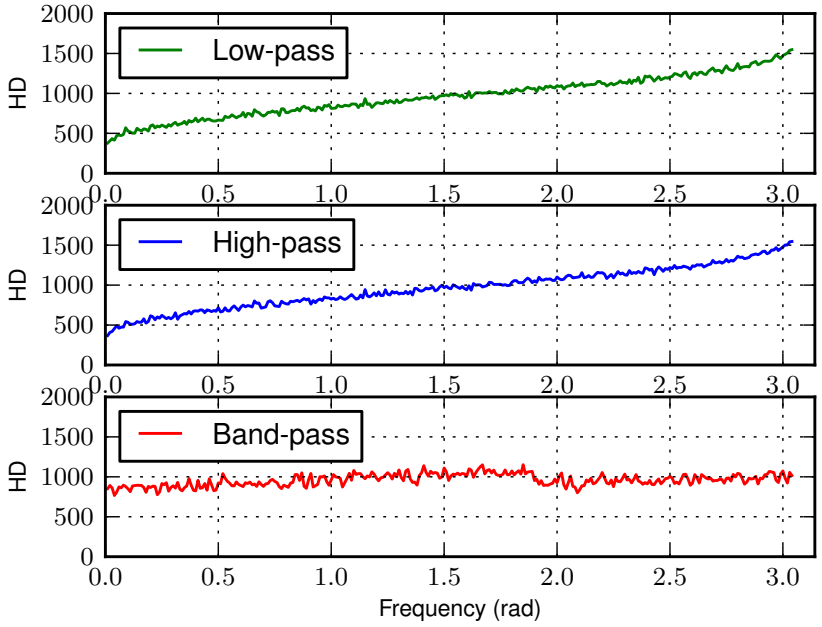

Figure 11.2: HD evaluation as cut-off frequency is increased for a 120th order filter.

The low-pass and high-pass graphs shows that the HD increases as the cut-off frequency increases. The similarities between the low and high-pass can be explained, as the filters are defined as:

$$H_{Low-Pass} = 1 - H_{High-Pass} \tag{11.2}$$

The band-pass graph shows approximately no change in HD as the cut-off frequency increases. To see what causes the trends seen on figure 11.2, the histograms in figure 11.3, 11.4 and 11.5 are created.

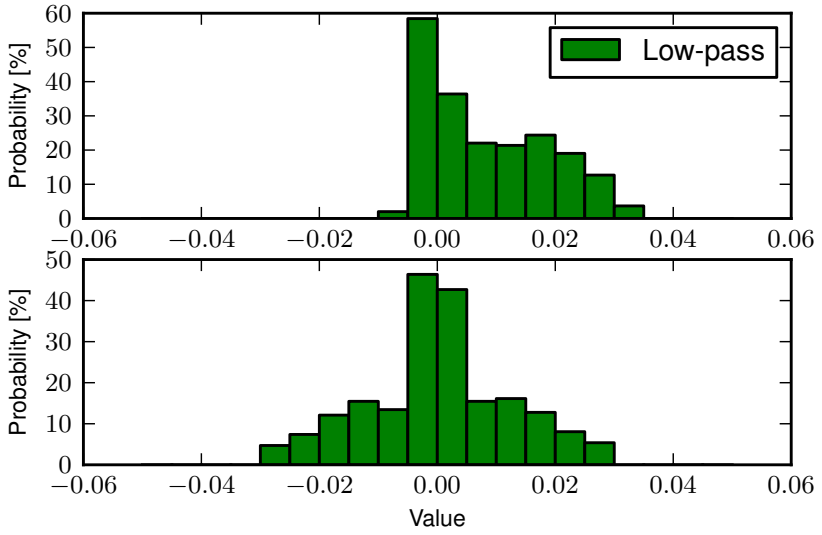

Figure 11.3: Histogram showing the low-pass filter coefficient values. The top graph show the distribution for filters with cut-off frequencies near zero and the bottom graph for cut-off frequencies near π.

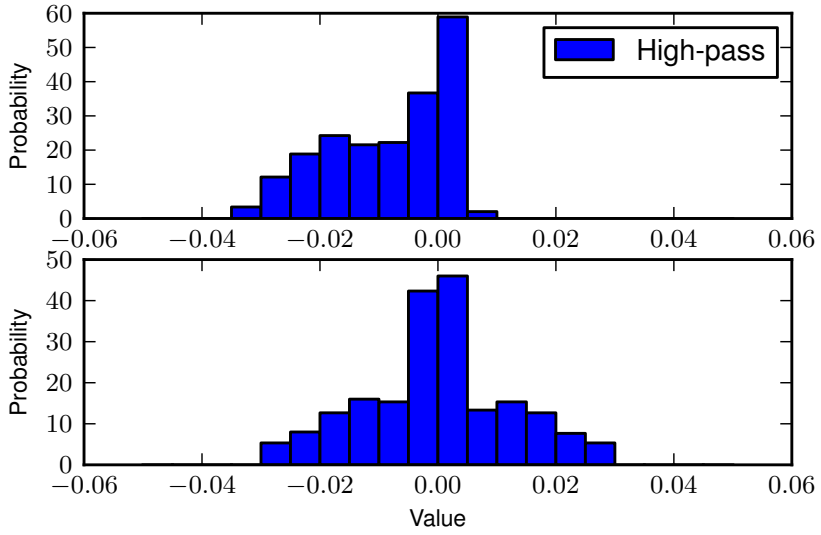

Figure 11.4: Histogram showing the high-pass filter coefficient values. The top graph show the distribution for filters with cut-off frequencies near zero and the bottom graph for cut-off frequencies near π.

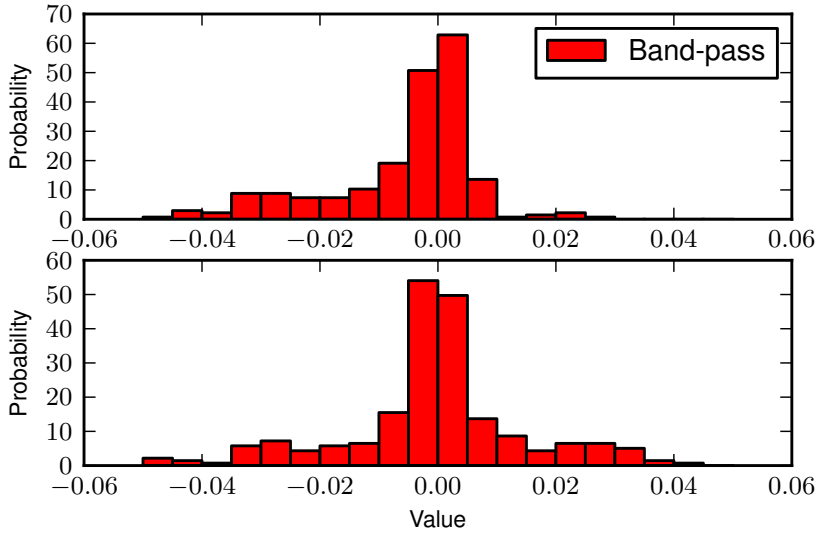

Figure 11.5: Histogram showing the band-pass filter coefficient values. The top graph show the distribution for filters with cut-off frequencies near zero and the bottom graph for cut-off frequencies near π.

The histograms are created by using the coefficients for the five filters closest to 0 rad for the top graphs, and the five filters closest to π rad for the bottom graphs. On figure 11.3 it is seen that the coefficients tend to distribute above 0 when the cut-off frequency is close to zero (top graph). The same trend is seen on figure 11.4, just with coefficients distributed mostly below 0. In the case where the cut-off frequency is close to π (bottom graphs), the coefficients tend to distribute on both sides of 0, this goes for both the low-pass and high-pass filters. When the coefficients are distributed almost equally on both sides of 0, then the HD will most likely be higher, than when they are distributed more dominant on the positive or negative side. This explains the trend seen on figure 11.2. As the distribution of the band-pass coefficients in figure 11.5 are distributed on both sides of 0, for frequencies close to both 0 and π, the HD will not differ significantly, which is also the case from figure 11.2.

This evaluation has shown that the cut-off frequency affects the HD between filter coefficients. This is however in a different degree dependent on the filter type. The low-pass and high-pass filters has an increasing total HD for increasing cut-off frequency. The HD of the band-pass filters is approximately the same independent of change in cut-off frequency. The filters described in the evaluation set in chapter 3 on page 13, has a fixed cut-off frequency, which makes it possible to use these as filters in an evaluation of different coefficient optimisation methods. The cut-off frequency for the evaluation set is chosen to be $\pi/2$. It should however be noted that the above performed analysis showed that if the cut-off frequency is changed, so is the HD.

12 Evaluation of HD Optimisation Methods

This chapter presents the evaluation of the HD optimisation methods described above, compared to filters using the unoptimised coefficients. The chapter consists of two main parts, a preliminary test and a test on actual filters.

The optimisation simulations described in the chapters 7, 8, 9 and 10 are based on platform independent simulations, concerning reduction in HD only. A reduction in HD corresponds to a reduction in energy consumption in an actual implementation [Mehendale et al., 1998], but can presumably not be transferred directly, i.e. a HD reduction of 50 % does not give a 50 % reduction in the overall energy dissipation. Therefore; actual implementation of filters will be used in order to conduct the following tests. The reduction in energy consumption will be tested using a microcontroller from Microchip. The used microcontroller is a dsPIC30F3012, which is a relatively simple microcontroller, with DSP functionalities. Of these are worth mentioning the multiply-accumulate function, which is very effective when dealing with filters. This microcontroller has the advantage that it has two separate data buses, so the filter coefficients can be transferred directly after each other, as described in chapter 7. The FIR filter that is implemented onto the microcontroller is written in the assembly coding language, and can be found on the enclosed CD.

The test is conducted by measuring the voltage drop over a shunt resistor and from this calculating the energy consumption used by the DSP with different filters implemented once at a time. How to measure and compute the energy can be seen in appendix C on page 167. The test is divided into two phases, a preliminary test and an actual test of different filters, figure 12.1 illustrates this.

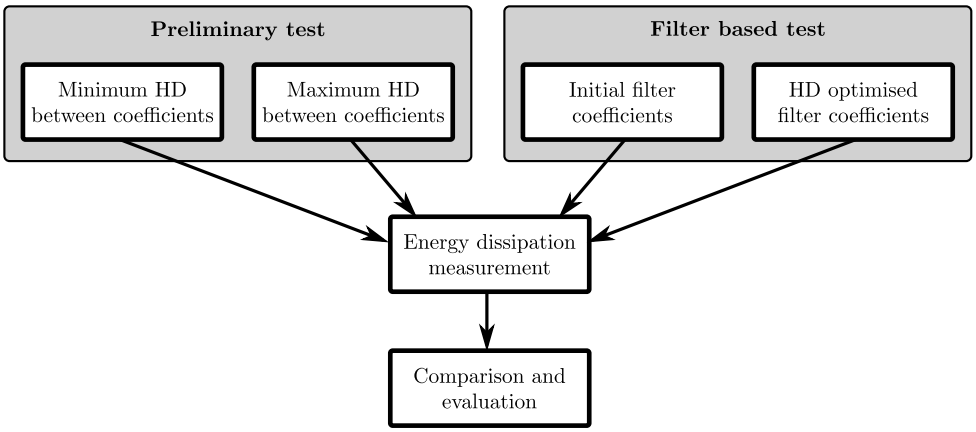

Figure 12.1: Test procedure of the coefficient optimisation methods.

12.1 Preliminary test

The preliminary tests are performed to see that the energy dissipation depend on HD between the successive filter coefficients. In this test two cases are tested. Best case, having all filter coefficients identical and worst case where the filter coefficients are as different as possible in terms of HD. Another reason for doing this test, is to obtain the maximal and minimal optimisation possible and to get knowledge about the energy that is expected to be consumed by the circuit.

In the best case, all filter coefficients are identical. This represents the best case, as the HD between the successive coefficients is zero. The test will be conducted using three different sets of coefficients. One set only consisting of zeros, another set consisting of the number *F0F0* and a third set consisting of *FFFF*. The coefficients used in each set are listed in table 12.1. The reason for testing several coefficient sets all having the same HD is to investigate whether the value of the coefficients has any effect on the energy consumption.

	Coefficients
BC 1	0000000000000000
BC 2	1111000011110000
BC 3	1111111111111111

Table 12.1: Best case coefficients

For the worst case, the filter coefficients are chosen to have maximum HD. Also this case will be tested using three sets of coefficients. On the bus, both HD between successive coefficients and Adjacent Signals Toggles (AST) in opposite directions has impact on the energy dissipation [Mehendale et al., 1998]. Therefore; in the first set every other filter coefficients in this case is sat to be *5555*, while the coefficient in between is *AAAA*. This gives the maximal HD and maximal toggles in opposite direction between two adjacent signals. The second set consists of the filter coefficients *00FF* and *FF00*. In the third set the coefficients are sat to be all zeros and all ones in every other coefficient. With these sets of coefficients the effect of the AST in terms of energy dissipation can be investigated. The coefficients used in each set are listed in table 12.2.

	Odd coefficients	**Even coefficients**
WC 1	0101010101010101	1010101010101010
WC 2	0000000011111111	1111111100000000
WC 3	0000000000000000	1111111111111111

Table 12.2: Worst case coefficients

The preliminary test is done for several filter orders, from 8 to 120, increasing the filter order with 8 every time. It is expected that actual filter coefficients will have an energy dissipation between the best and the worst case, and that the optimised filter coefficients will have a lower energy dissipated than the unoptimised coefficients.

The following results are generated by following the instructions described in appendix C on page 167. The results from the preliminary test is shown in figure 12.2. As it is hard to separate the lines in this figure, table 12.3 are included to emphasise the findings in figure 12.2. The table shows the energy consumption for a small selection of filter orders.

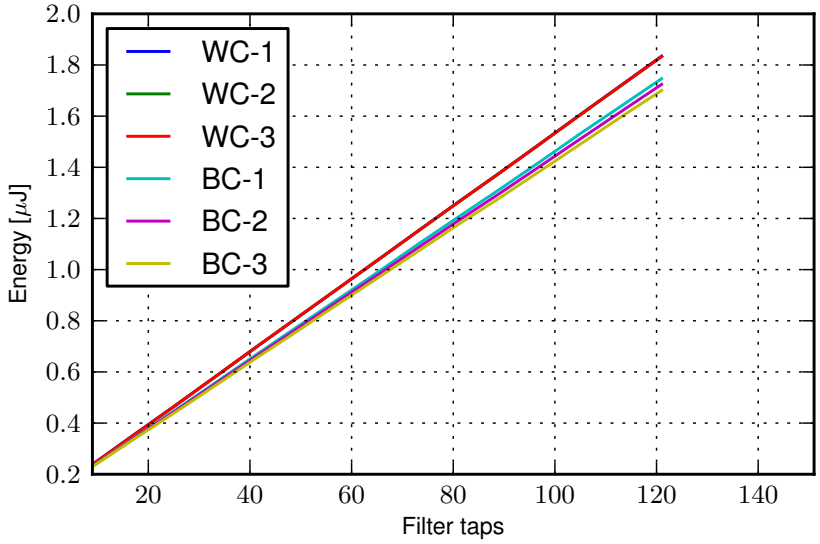

Figure 12.2: Energy dissipation for worst and best case HD.

	WC 1	**WC 2**	**WC 3**	**BC 1**	**BC 2**	**BC 3**
8. order	0.238 μJ	0.238 μJ	0.238 μJ	0.232 μJ	0.231 μJ	0.230 μJ
32. order	0.579 μJ	0.579 μJ	0.579 μJ	0.555 μJ	0.551 μJ	0.544 μJ
64. order	1.035 μJ	1.035 μJ	1.036 μJ	0.988 μJ	0.978 μJ	0.965 μJ
120. order	1.834 μJ	1.833 μJ	1.833 μJ	1.744 μJ	1.724 μJ	1.700 μJ

Table 12.3: Energy consumption for the different worst and best case coefficients

First considering the best case coefficient sets, where the HD is zero in all cases. Figure 12.2 and table 12.3 shows that there is a difference between the three sets. The set with coefficients equal to zero, dissipates the most energy out of the three best case sets. While the set consisting of only ones consumes the least amount of energy. The coefficient set containing the coefficient *F0F0* lies in between the two others. The difference between BC1 and BC2 is approximately the same as the difference between BC2 and BC3. In total there is an average difference between BC1 and BC3 of 1 %.

For the worst case coefficients, the difference between the energy dissipations are not significant. They different sets dissipated approximately the same amount of energy, as seen in figure 12.2 and table 12.3. Based on the results presented here, it can be concluded that for this microcontroller, AST does not affect the energy dissipation. This can be seen as the WC1, with 15 AST, and WC3, having zero AST, consumes the same amount of energy. However; what seems to have influence on the energy consumption on this DSP is the number of ones in the coefficients. In the worst case sets the total number of ones in the coefficients are fixed, and the energy dissipation does not differ significantly. While for the best case sets the number of ones are changing. BC1 does not contain any ones, and consumes more energy than BC3 which only consists of ones. Therefore; when referring to the best case in the following sections, the best case will be BC3 which only consists of coefficients equal to *FFFF*.

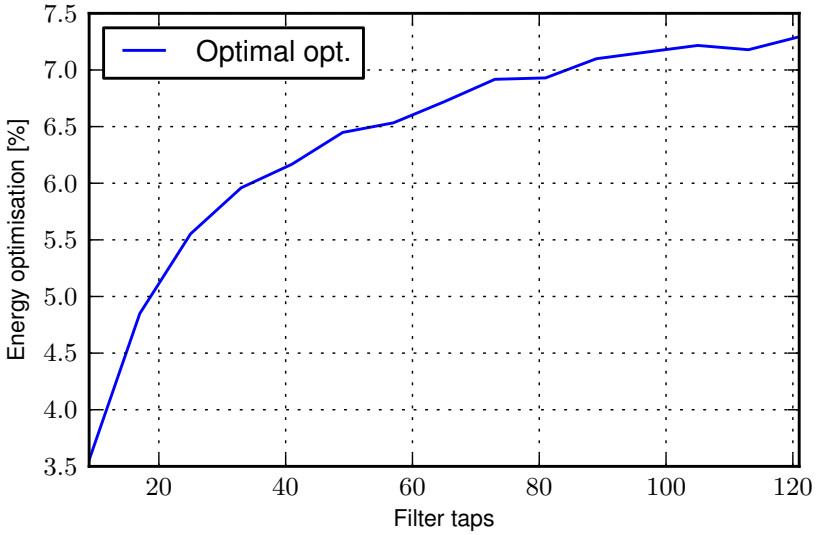

Figure 12.3: Maximal possible optimisation.

From figure 12.2 it is seen that the best case coefficients gives an overall lower energy dissipation than the worst case. This means that energy can be saved when minimising the HD between adjacent filter coefficients, which is an important result as it shows that the theory on coefficient optimisation holds in an actual implementation. The optimisation when replacing the worst case coefficients with the best case coefficients can be seen in figure 12.3. It is seen that the slope of the graph is larger in the start than in the end. This is due to the overhead of the filter, which is a significant part of the filter at smaller filters, but gets more and more insignificant as the filter length increases.

Based on the observation that the coefficients containing all ones dissipated less energy than the coefficient set containing zeros, improvements to the scaling and coefficient optimisation algorithms are suggested. The coefficient optimisation algorithm can favour coefficients containing as many ones as possible, or it can be designed to favour ones in the coefficient perturbation. As for the coefficient scaling both the number of ones and the HD can be considered as factors in order to determine the optimal solution.

It should be noted, that the energy values illustrated on figure 12.2 are the energy consumed by the entire microcontroller. This means that the energy values are the overall energy dissipation of the chip. As all other parameters in the implementation is kept fixed, thereby only changing filter order and coefficients, the difference in energy between worst case and best case are only a small part of the whole chips energy dissipation.

12.2 Filter based test

The test on actual filters is conducted in order to evaluate the HD optimisation methods investigated previously, see chapter 7, 8 and 9. All these methods are simulated with regard to HD reduction. The percentage of HD reduction does however not convert directly into a similar reduction in energy consumption in an actual implementation. Based on this test the actual energy optimisation achieved with the optimisation methods described earlier, is found.

The unoptimised filter coefficients are created by the *remez* function in Python. For the optimised filters, all the previous HD optimisation methods have been applied as described in chapter 10 on page 71.

As in the preliminary test, several filter orders are used. This is done in order to see how the energy dissipation and the optimisation evolves as the order and size of the filter is increased. Filter orders from 8 up to 120, in steps of 8, are used in the test. The filter specifications given to the *remez* function are kept fixed though the order is varied.

The filter characteristics is a low-pass filter with a cut-off frequency of $\pi/2$. The filters used for testing are further defined in the evaluation set in chapter 3 on page 13.

The result from the test performed on actual filters is shown in figure 12.4, in addition to the best case and worst case from the preliminary test. As supplement to the figure, table 12.4 states the energy consumption of a selection of filters. On the figure it is seen that the unoptimised and optimised filters uses less energy than the worst case and more energy than the best case, as expected. Likewise; the optimised filters has lower energy dissipation than the unoptimised filters.

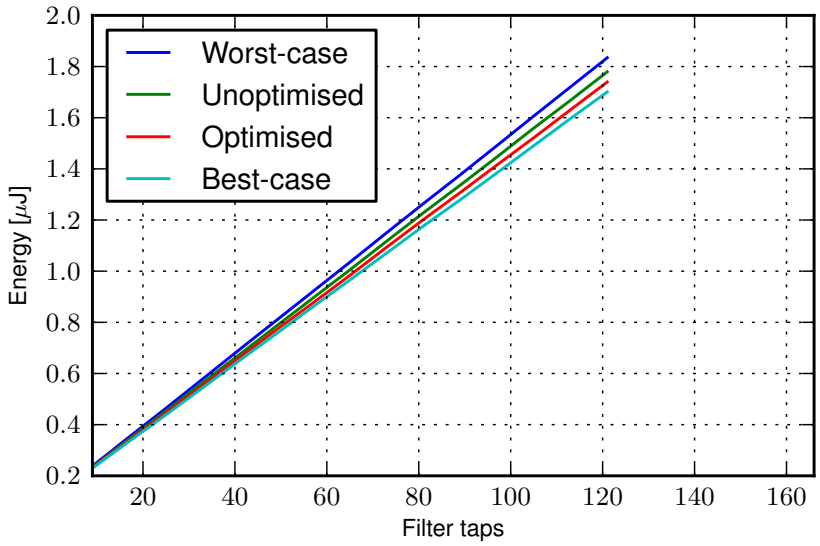

Figure 12.4: Energy consumption for a range of filters.

	Worst case	**Unoptimised**	**Optimised**	**Best case**
8. order	0.238 μJ	0.234 μJ	0.232 μJ	0.230 μJ
32. order	0.579 μJ	0.564 μJ	0.554 μJ	0.544 μJ
64. order	1.035 μJ	1.005 μJ	0.983 μJ	0.965 μJ
120. order	1.834 μJ	1.779 μJ	1.739 μJ	1.700 μJ

Table 12.4: Energy consumption of worst case, best case, unoptimised and optimised coefficients.

The optimisation in percent is shown in figure 12.5. This figure shows the energy optimisation when going from the worst case to best case (optimal optimisation), as in the preliminary test, and the actual energy optimisation achieved by applying the HD optimisation methods to filters of different order (actual optimisation).

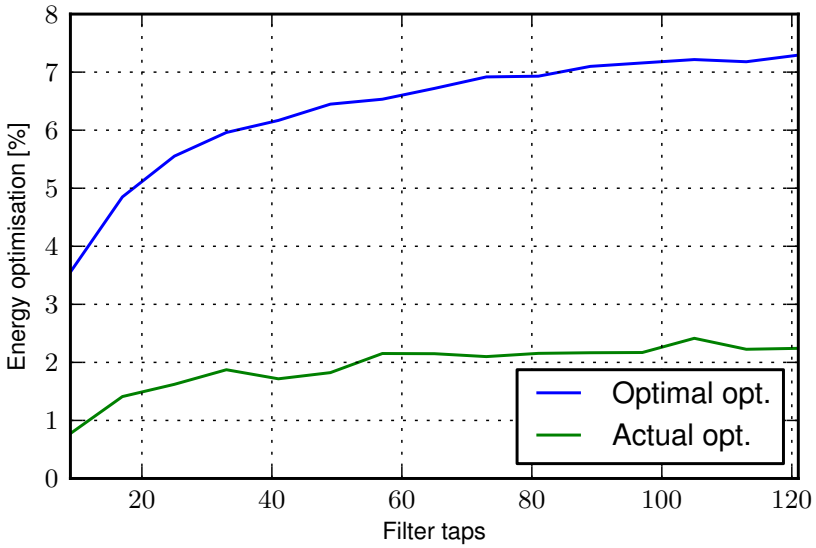

Figure 12.5: Optimisation in energy consumption between unoptimised and optimised filters.

The optimal optimisation is found to be increasing from about 3.56 % to 7.29 %, while for the actual optimisation of the filters the optimisation is found to increase from 0.77 % to 2.24 %. Both graphs has a larger slope in the beginning and then gets a small slope in the end. This is probably due to the size of the overhead in the filters, which may influence the overall optimisation at smaller filters. When the filter overhead gets less significant, the slope seems to be constant.

12.3 Conclusion

This section draws conclusions on the three HD reducing methods as ways of designing low-energy FIR filter, based on the preliminary and filter based test.

By considering the preliminary test, where the worst case and best case coefficients are investigated, a possible improvement of 3.6 to 7.3 % is found. This represents the best possible optimisation and this will presumably never be the case in an actual implementation situation.

The testing of the actual filter coefficients on the other hand, represents the actual possible optimisation and is tested for filter orders from 8 to 120 in steps of 8. As a conclusion on this test, it is noted that an overall energy optimisation is observed between 0.77 and 2.24 %. This is seen as a good result, as the change in HD and thereby switching activity of the coefficients, is just a very small part of the overall energy consumption in a microcontroller. The results presented here are based on energy measurements of the entire microcontroller, contrary to [Mehendale et al., 1998] which presents only the reduction in HD and AST.

It is observed that the potential energy optimisation rises with rising filter order, which is a result generated by larger and larger HD difference between unoptimised and optimised coefficients. Based on the conducted tests it can be concluded that the three methods for reducing HD can be used to design low-energy FIR filter, though the possible optimisation is limited due to the fact that the optimisation in HD is just a very small part of the overall energy consumption.

Part III

Hardware Optimisation Methods

13 Multiplierless Filters

This chapter will provide an introduction to the following chapters on the design of multiplierless filtering methods. The multiplier is expected to be the most energy consuming operation in digital filters. This leads to a focus on eliminating the multiplication by various techniques, as to lower the overall energy consumption of the filter.

Distributed arithmetic and a method of eliminating multiplications by replacing them with shifts are presented as two methods of designing multiplierless filters. In addition to these two methods, an implementation combining multirate and shift based filtering is also investigated. In relation to the shift based multiplication method, Common Subexpression Elimination (CSE) techniques are introduced in order to bring down the number of functional units such as adders and shifters. Distributed arithmetic and shift based filtering are chosen for this project due to their commonness and popularity in various studies. In addition, both methods are still research fields, which makes them highly relevant.

All multiplierless filtering methods are implemented and evaluated on a FPGA along with a reference filter, which is an implementation of a general transversal direct form FIR filter with multiplications. The filters are implemented on a FPGA and are evaluated by measuring the energy dissipation of the FPGA core. The FPGA used, is the state-of-the-art Cyclone 3 which is mounted on a development board from Altera. The Cyclone 3 board is chosen, as it has pins to measure the energy dissipation of the core and are easily accessible. The board is equipped with a 50 MHz oscillator, which can be scaled internally as the FPGA contains phase-locked loop (PLL) circuits. Through the rest of the project, the clock used will be the 50 MHz multiplied by four by a PLL circuit, which results in a clock frequency of 200 MHz. The reason for scaling the oscillator clock is to increase the dynamic energy dissipation and thereby getting larger differences between the evaluated filters. An example of this will be shown in the following preliminary test.

13.1 Preliminary test

The following section will introduce two types of preliminary tests in order to analyse the effects of using the PLL and the energy consumption of different filter characteristics, these being low-pass, high-pass and band-pass filters.

The following tests are executed with a parallel transposed direct form FIR filter structure, which is described in section 14.2. The input to the filter implementations is a pseudorandom signal generated by the pseudorandom generator described in appendix D on page 171. Using a pseudorandom input signal instead of a fixed signal has the benefit that it uses long time before it repeats it selves and therefore generates random signal through the filter. This is a necessary part of the implementation, as no specific application and thereby input is introduced in this project.

The first thing to look at is the effect of the PLL. The PLL which is used in this project multiplies the original clock frequency by four. The effect of this should be that the dynamic energy of the core should also be multiplied by four. To see if the PLL has the effect wanted, four filters of different length has been implemented and tested with and without PLL. The values in this test are estimated using Altera's PowerPlay tool. The energy measures from a single sample period are seen in table 13.1.

Filter length	Core energy dissipation	
	With PLL	Without PLL
8	0.66 nJ	0.28 nJ
40	1.23 nJ	0.41 nJ
80	1.83 nJ	0.54 nJ
120	2.15 nJ	0.65 nJ

Table 13.1: Test of the PLL effect.

As can be seen from the table, the core energy dissipation when using PLL is between 2.3 and 3.4 times bigger than the implementation without. The difference between the filter implementations with the PLL is now bigger, which makes it easier to see the difference between the filters energy dissipation, as this is bigger. This illustrates the idea of why the PLL is used in this project. The total energy consumption of the chip is actually higher in reality, as the PLL also consumes energy. The PLL's energy dissipation is subtracted from the total chip consumption to see the difference in core energy dissipation.

The next thing to look at is the effect of different filter characteristics. The filter characteristics are expressed by the coefficients, which is the parameter which is changing in this test. To see the influence of changing filter characteristics, three different filter types are used with varying length. The three filter types are created by the *remez*-function in Python after the characteristics in table 13.2.

Filter type	**Low-pass filter**	**High-pass filter**	**Band-pass filter**
Cut-off frequency	$\pi/2$ rad	$\pi/2$ rad	$\pi/3$ to $2 \cdot \pi/3$ rad
Transition bandwidth	0.1 rad		
Phase property	Linear		
Filter length	8, 40, 80 and 120		

Table 13.2: Filter characteristics for testing the effect of these on a FPGA.

The values in this test are measured values, and are found as described in appendix E on page 175. The measured values are seen in table 13.3.

Filter length	Low-pass filter	High-pass filter	Band-pass filter
8	0.66 nJ	0.67 nJ	0.64 nJ
40	1.23 nJ	1.22 nJ	1.16 nJ
80	1.83 nJ	1.84 nJ	1.86 nJ
120	2.15 nJ	2.53 nJ	2.28 nJ

Table 13.3: Test of the effect of changing filter characteristics.

From the table, it is seen that the values for the same filter lengths are very similar. The values will change due to the changing amount of signal toggles, which is introduced as different coefficients are used and the output of the filter will differ. Due to the relative small span of the values, this justifies that the evaluation set in chapter 3 on page 13 only contains low-pass filters of various lengths.

13.2 Logic elements in FPGA technology

As the filters will be implemented and evaluated on a FPGA, this section will give a short introduction to the FPGA technology with focus on the fundamental building block of this, the logic element.

The FPGA contains a number of programmable blocks called logic elements (LE). The LE contains a number of functionalities which can be put together in order to generate e.g. logic gates like *AND* and *OR* gates. The LE's in the Cyclone 3 FPGA used in this project, can work in two different modes, normal or arithmetic. In normal mode, the LE consist of a LUT and a programmable register. The LUT can e.g. be used to make combinational functions, and the register can function as a D, T, JK or SR flip-flop. In arithmetic mode, the LE consist of two LUT's, a 2-bit full-adder and a carry chain. The arithmetic mode can be used to program e.g. counters, adders and accumulators [Altera, 2011].

The Cyclone 3 FPGA contains almost 25000 LE's. These are ordered in groups of 16 LE's in so called logical array blocks (LAB). The internal connection of LAB's is illustrated on figure 13.1.

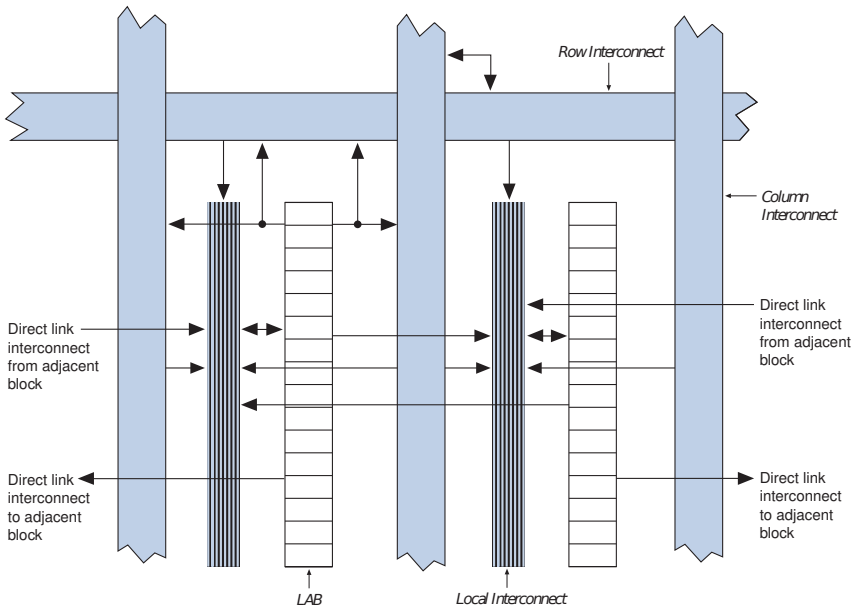

Figure 13.1: The structure and connections of LAB's in a FPGA [Altera, 2011].

The LE's in a LAB are connected together by a local interconnection. This assures that signals can be transferred from one LE to another. The registers in the LE's are connected by register chains, so the value of one register can be moved to the adjacent register.

The LAB's are connected in different ways to make area and speed efficient implementations. Every LAB has a direct link to the adjacent LAB. In addition, row and column interconnection makes it possible to transfer signals from one LAB to another LAB which is not necessarily right next to the current LAB.

This chapter introduce the term multiplierless filtering and performs a preliminary test on the FPGA used in this project. In addition, the fundamentals of FPGA technology was described briefly.

The preliminary test contained a test to see the effect of using a PLL circuit to scale the original clock to the FPGA. The PLL scaled the core energy dissipation, which will make it easier to see the difference between the multiplierless filters introduced in the following chapters. In the preliminary test the energy dissipation when using different filter characteristics was also investigated. This test showed that no significant difference between the filters, and due to this the evaluation set introduced earlier is representative.

14 Reference Filter

The following section will describe the implementation and evaluation of the reference filter, which is a general FIR filter with normal multiplications.

The filter is written in the VHDL programming language and is described in the following. The VHDL is written with the purpose of implementing the filter onto a Cyclone 3 FPGA mounted to a development board from Altera. The FIR filter block looks as shown on figure 14.1.

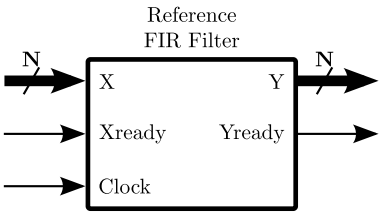

Figure 14.1: The reference filter block.

The filter has an input port, X, which has a width of N bits. Equally the output, Y, has a width of N bits. To indicate that the input is ready to be clocked into the filter, the $Xready$ signal is set high when X is ready. In the same way, the filter sets the $Yready$ signal high when the filter has finished a calculation and an Y is ready to be read. When $Yready$ is set, it is also an indication on that a new X can be loaded into the filter. A clock signal, $Clock$, is also necessary for the filter to run.

Before the internal parts of the reference filter block is explored, the term parallelism has to be discussed. As VHDL is a hardware description language, circuitry can be programmed to perform more tasks at the same time. This rises a new view at a normal FIR filter, as e.g. all the multiplications can be computed at the same time. To do this, the filter circuit needs to have as many multipliers as taps in the filter, which then uses more space on the FPGA. As the space and also activity of the circuitry increases so does the power, but contrary to a sequential filter implementation, the processing of an input will be much faster. The following implementation of the general FIR filter, will cover both a parallel and a sequential versions of calculating the filter output. This as to evaluate on the difference and meaning of using the one over the other.

The reference filters are implemented as transposed FIR filters, which is a very common way of implementing FIR filters on FPGAs [DeBrunner, 2007], as this makes the handling of the delayline more convenient, in the way that it can be calculated in a single clock cycle. This will be shown, when the implementations of the filters are explained later on.

The FPGA contains a fixed number of optimised embedded multipliers, which can be used by the designer. The embedded multipliers use less power than multipliers programmed by LE's in the FPGA [Oliver and Boemo, 2011]. The following implementation will not turn to use the embedded multipliers, as to see how non-optimised multipliers work compared to other ways of calculating a multiplication.

14.1 Sequential FIR filter

This section describes how the implementation of the sequential FIR filter is structured and how the code for this is written.

The sequential FIR filter version only uses a single multiplier when calculating the multiplication of the input by the coefficients. The DFG of an 8 tap transposed filter structure used to implement the FIR filters, is shown on figure 14.2.

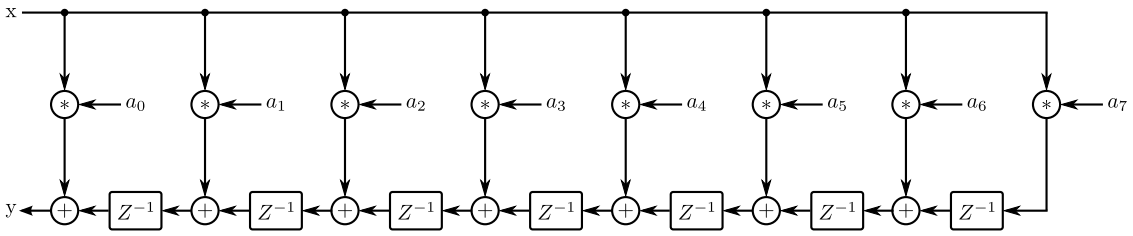

Figure 14.2: DFG of the 8 tap reference FIR filter.

It is seen that the input signal x is multiplied by every coefficient and inserted into the delayline. The DFG is the same for both the sequential and parallel versions of the FIR filter, so to see the actual difference, the scheduled precedence graph (PG) has to be visualised. On beforehand, the implementation is allocated a single multiplier and $N-1$ adders. The scheduled PG is seen on figure 14.3.

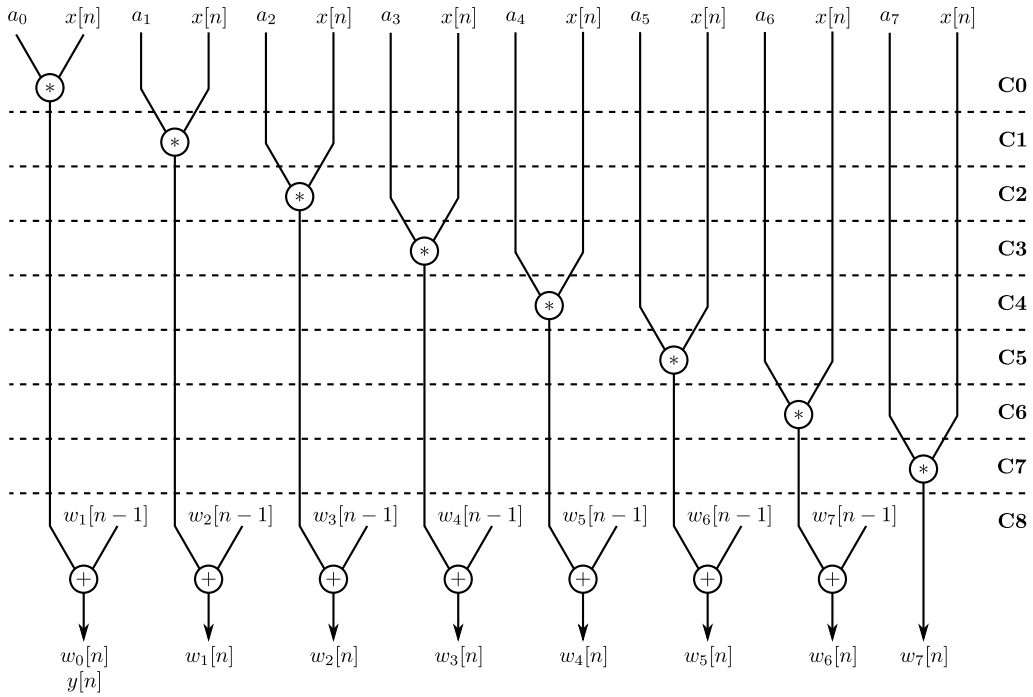

Figure 14.3: PG of the sequential 8 tap reference FIR filter.

On the PG, the clock cycles are denoted to the right from C0 to C8, which means that this 8 tap filter can be calculated in 9 clock cycles. The first 8 cycles from C0 to C7 calculates every tap of the filter. The sequentiality is seen as the taps are calculated sequentially after each other. This is convenient as only a single multiplier is needed. As earlier stated, the transposed filter structure makes it easier to handle the delayline. This can be seen in the bottom of the PG, where the delayline is updated. The variables w_0 to w_7 are the values in the delayline and it is seen that the old values are used to calculate the new. The first element in the delayline, w_0, is also the output, $y[n]$.

The VHDL coding of the update process is realised as seen in listing 14.1.

```
1 for j in 0 to taps-1 loop
2     if (j < taps-1) then
3         w(j) <= subres(j)(30 downto 15) + w(j+1);
4     else
5         w(j) <= subres(j)(30 downto 15);
6     end if;
7 end loop;
```

Listing 14.1: VHDL code describing the update process of the delayline.

The *subres* variable holds the truncated values from the multiplications. The reason why the *subres* variable is truncated, so the only bits used are those from position 30 down to 15, is that the multiplication performed generates an output of Q2.30 in Q notation. The wanted binary representation is Q1.15, which makes it necessary to truncate the *subres* variable. The truncation could be moved, so the only value truncated is the output y. This will give a more precise calculation, but will also use more wiring on the FPGA. The delay element, $w(j)$, is updated by calculating the addition of the proper *subres* value by the next delay element. This with the exception of the last element in the delayline, as this is set to the last *subres* value. This procedure is seen in both the VHDL code, the DFG and the PG.

The RTL design of the described update procedure is seen on figure 14.4. It is seen that the implementation of the delayline is very cost efficient, as the amount of components are kept at a minimum.

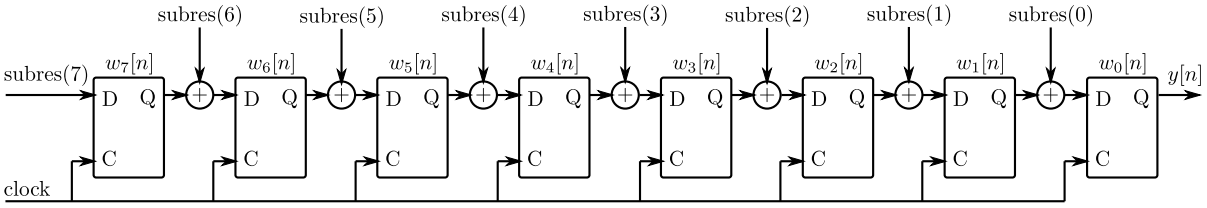

Figure 14.4: RTL design of the delayline.

The RTL implementation contains 7 adders and 8 latch arrays. The latch arrays are designed to hold a 16 bit wide value. At every clock cycle, the value from all latch arrays except w_0, is added with a *subres* value and stored in the next latch array. This means, as illustrated on the PG, that the delayline can be updated in one clock cycle.

The sequential FIR filter implementation is made as a simple state machine, using four states. The action of each state is:

State 0 waits for the *Xready* signal to be sat high. When this happens, the *Yready* is sat to low, as to indicate that the filter output is being computed. A counter i is sat to 0. The current input X is saved into a register.

State 1 calculates the *subres* and increases the i variable by one every time to fetch the correct coefficient.

State 2 updates the delayline as described earlier.

State 3 sets the first element from the delayline as output from the filter, Y, and the *Yready* signal is sat high. The state machine starts over again.

14.1.1 Simulation

The following section will show a simulation of the reference filter block to verify that it works as intended.

To simulate the VHDL code of the reference filter, the Altera ModelSim application has been used. To generate the input to the filter, a testbench has been constructed. A testbench implements the filter block seen on figure 14.1 and generates the proper input. The VHDL code for the testbench is as:

```
1 input := X"4000";
2 if Yready = '1' and Xready = '0' then
3     X <= input;
4     Xready <= '1';
5     input := input - X"2000";
6 elsif Xready = '1' then
7     Xready <= '0';
8 end if;
```

Listing 14.2: The code which defines the core of the testbench.

As can be seen from listing 14.2, a new X is set if $Yready$ is high and $Xready$ is low, and else set $Xready$ low. The X is set with the value from a variable $input$, which the first time is hexadecimal value 4000. When X is set to the value, the $input$ variable is subtracted by the hexadecimal value 2000. The last thing which is needed in the test bench is a process which generates the appropriate clock signal. This can be generated as:

```
1 clock: process is
2 begin
3     Clock <= '0';
4     wait for 10 ns;
5     Clock <= '1';
6     wait for 10 ns;
7 end process;
```

Listing 14.3: The process which defines the clock of the testbench.

The process starts by setting the clock signal $clock$ to 0, thereafter wait for $10ns$, then set the clock signal to 1 and again wait $10ns$. This process is executed over and over again, so the clock signal is constant all the time. In the implementation a PLL clock is also implemented, as stated in chapter 13. This speeds up the clock by a factor of four.

The simulation is executed, with a coefficient set only containing the hexadecimal value 4000, which is the same as the decimal value 0.5. This makes it easy to verify the output from the filter, as this should just be the half of the sum of the last 8 input values. The filter in this simulation is a filter with 8 taps, which makes the filter definition look like:

$$\frac{1}{2}\sum_{k=0}^{7} x[n-k] = y[n] \tag{14.1}$$

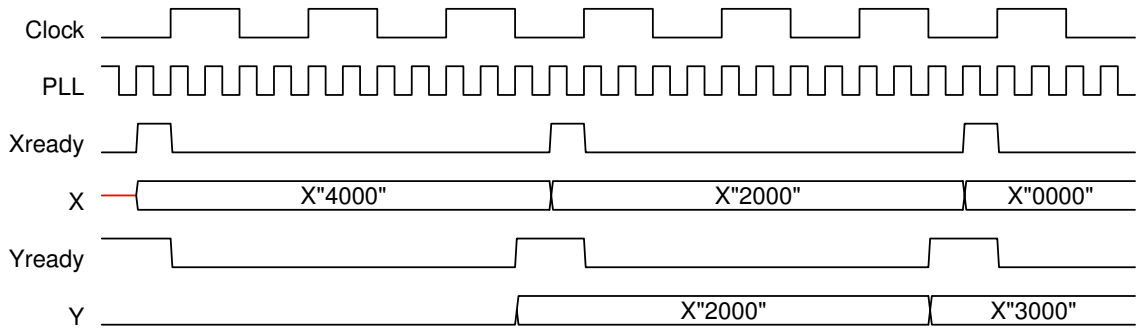

Figure 14.5: The timing diagram of the reference filter block.

Figure 14.5 illustrates the two first periods of the sequential FIR filter realisation. The figure includes both the PLL clock and the original clock signals. Here it is seen that the timing is increased by four when using the PLL clock instead of the initial clock. This will be used for all implementations and will from here on only be referred to as clock.

This simulation starts with a delayline which is initialised to zero, why the *Y* signal starts being low. It is seen that the input value is clocked into the filter when the *Xready* signal goes high. One clock cycle later, the *Yready* signal goes low, which indicates that the filter is processing the input. In the following 11 clock cycles, the filter is initialised and the output is calculated. The number of cycles depend on the number of taps. The filter will calculate an output in N+3 cycles, where N is the number of taps. When the filter is done processing the input, the result is sat as the *Y* signal and the *Yready* is sat high. The output from the filter should here be the half of the sum of the delayline, which it is as the only input value in the filter is the hexadecimal value 4000 and the output is 2000. The next input is hex 2000, which now generates an output of hex 3000, which is correct, as the sum of the delayline is hex 6000.

If zooming out on the timing diagram it will look as on figure 14.6.

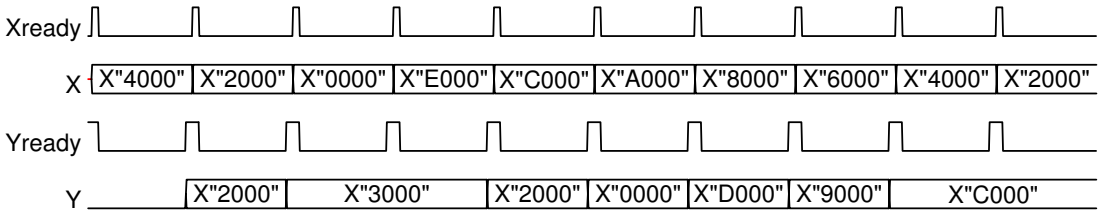

Figure 14.6: The timing diagram of the sequential reference filter block for 10 periods.

It is seen that the output *Y* is always the half of the sum of the last 8 input values. As the filter has 8 taps and the input value repeats itself after 8 values, the output becomes constant after the 9th input value. The output will stay at hex C000 from now on.

It is seen that the reference filter works as expected and can now be programmed to the FPGA to be evaluated.

14.2 Parallel FIR filter

The following section presents another way of designing the reference filter, by computing several multiplication in parallel.

A parallel FIR filter can be build by using several multipliers, thereby calculating the multiplications in the filter in one clock cycle. The DFG of the parallel filter is the same as the sequential filter, shown on figure 14.2 on page 90. The PG of the parallel filter is shown on figure 14.7.

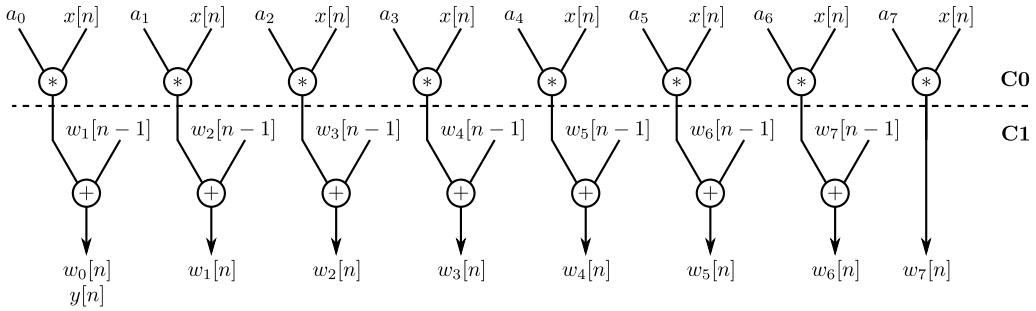

Figure 14.7: PG of the 8 tap parallel FIR reference filter.

The parallel filter structure calculates an output in only 2 clock cycles. As can be seen on figure 14.7, all multiplications are calculated in only one clock cycle. Next, the delayline is also updated in a single clock cycle, just as for the sequential filter.

The VHDL code for calculating the multiplications in parallel and storing them in a register array, *subres*, can be written as on listing 14.4.

```
1 for j in 0 to taps-1 loop
2     subres(j) <= signed(coef(j)) * signed(input);
3 end loop;
```

Listing 14.4: VHDL code for calculating filter taps in parallel.

As the sequential filter, the filtering computation is controlled by a simple state machine of 4 states, which is described in the following:

State 0 waits for the *Xready* signal to be set high. When the signal arrives, the *Yready* signal is set low, and the value of the input, X, is saved in a register.

State 1 calculates the multiplications of the input with the coefficients in one clock cycle. The result of every multiplication is saved in the register array *subres*.

State 2 updates the delayline, by setting every i'th value in the delayline to the i'th *subres* added with the next delayline value.

State 3 sets the delayline value at position 0 as the output from the filter, Y, and *Yready* is set high. The state machine then starts over.

14.2.1 Simulation

The simulation of the parallel FIR filter is performed in the same way as the sequential filter, and the timing diagram is seen on figure 14.8.

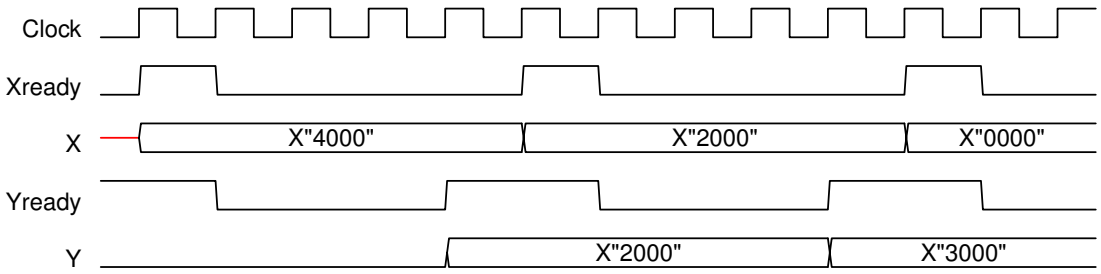

Figure 14.8: The timing diagram of the parallel reference filter block for the two first periods.

The difference from the sequential filter, is that the filter is done processing after 4 cycles. The processing will always take 4 cycles from an input is set and until an output has been set on the output port. This is contrary to the sequential filter which will take N+3 cycles and is therefore dependent on the number of taps.

In the following, the evaluation of both the sequential and parallel filters will be carried out.

14.3 Results

This section will analyse the results from the evaluation of the implementation of the sequential and parallel reference filters. The two reference filter will be used in order to compare their energy consumption with the energy consumption of the other implementation methods which are intended for low-energy filter design. The results will contain actual energy measurements and energy estimations generated by Altera's PowerPlay Power Analyzer.

In addition to energy comparison, the number of consumed LE's will also be used to compare the two filter. The measurements and estimations are performed as described in appendix E on page 175. For the evaluation there will

be used a pseudorandom input signal generated by the pseudorandom number generator described in appendix D. This is a necessary part of the implementation, as it is not wanted to specify which kind of signal enters the filter, due to the project specification which states that no specific application will be in focus. The filters used are those described in the evaluation set in chapter 3 on page 13.

The comparison between the measured and estimated energy consumption of the sequential and parallel reference filters is shown on figure 14.9.

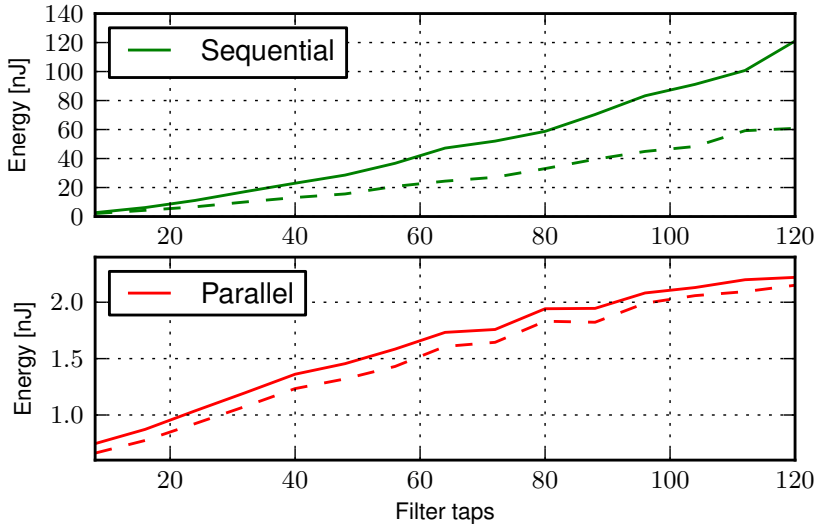

Figure 14.9: Energy comparison of the sequential and parallel reference filters. The dashed lines denotes the estimated energy based on values from Altera PowerPlay.

Considering the energy consumption of the two filters, a major difference is seen. The sequential filter consumes significant more energy than the parallel reference filter. A reason for this is that the sequential filter uses $N+3$ cycles to compute an output, while the parallel filter only use four cycles, independent of the number of filter taps. The parallel filter has an energy consumption increasing from 0.87 nJ to 2.22 nJ, while the sequential filter increases from 6.19 to 121.05 nJ, which is significantly higher than the parallel filter.

The estimated (dashed line) and measured energy (solid line) of the parallel implementation is following each other rather well. This is however not the case at the sequential implementation. At 120 taps, the difference between the estimated and measured values of the sequential filter is about $60nJ$. The PowerPlay tool is used in this project to see how it performs in comparison with actual implementations. Altera themselves claims the PowerPlay to estimate the power consumption within 10 % of the actual power consumption [Altera, 2010]. This can in this case directly be transferred into energy, as seen on the figure. For the parallel reference filter the difference between the estimated energy and the measured energy varies from 11.3 % to 3.1 % as the filter length increases. For the sequential filter the estimated energy is within 22.6 % and 49.8 %. For the sequential filter the difference between the estimated and measured value increases as the filter length is increased. The reason why the difference between the estimated and measured energy in the sequential is significant larger than for the parallel filter is not directly apparent, but shows that PowerPlay is more optimistic, than what is really the case for this type of implementation.

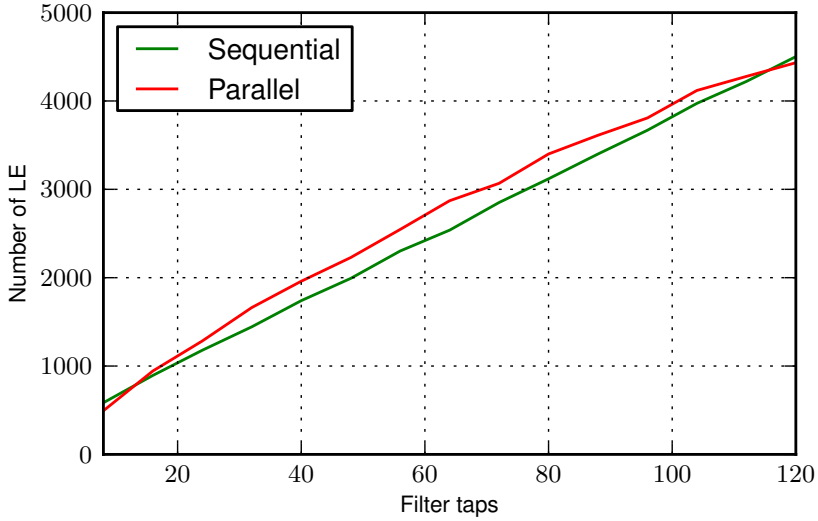

Figure 14.10: Comparison of the LE consumption of the reference filter implementations.

The LE usage of the two reference filters is shown in figure 14.10. It was expected that the consumed number of LE's was higher for the parallel implementation than the sequential, due to the fact that the parallel filter uses as many muliplers as needed in order to compute the output in four cycles, while the sequential filter uses one multiplier no matter which filter length is used. An unforeseen effect of the implemented sequential filter makes the number of LE increase in almost the same way as the parallel implementation. The sequential implementation contains only a single multiplier, but the size of the delayline and the logic to control which coefficient that should be multiplied by the input, uses a considerable amount of LE. The relatively simple design of the parallel filter, is much more space efficient since no logic is needed to control the coefficients. It should be mentioned, that the coefficients are not implemented in the on-chip memory on the FPGA and are instead implemented in registers which consumes LE. The amount of LE needed to implement the coefficients are however approximately the same for both the parallel and sequential implementations.

The results of the implementation of the reference filters are carried out and can now be used to compare different types of alternative implementations of the FIR filters.

14.4 Conclusion

This section summarises and concludes upon the implementation and results of the reference filters. Two reference filters are designed, so it is possible to do make a further comparison of different filter types with these. The sequential filter is designed using only one multiplier, and the parallel filter will use all necessary multipliers in order to do the computations as fast as possible.

Simulation of the filters has shown that both filter realisation computes the correct result. An important difference noticed here, is the number of cycles the filters needs to compute the output. The sequential filter needs $N+3$ cycles to complete, while the parallel filter only needs four cycles no matter what the filter length is. The effects of this can be seen as the energy consumption of the two filters is measured and estimated. Both estimates and measurements shows that the sequential filter has significant higher energy consumption than the parallel realisation.

The estimates and measurements of the parallel filter get along very good, with a maximal difference of approximately 11 %. However, for the sequential filter there is not an accordance between the estimated and measured energy. The result here has shown a great difference between the parallel and sequential reference filter. Both filters will be used as reference filters for the following comparison with other implementation methods.

15 Distributed Arithmetic Filtering

One possible multiplierless filter approach is distributed arithmetic (DA) which uses pre-calculated subresults stored in a look-up-table (LUT) to eliminate the multiplier in the filter. The following chapter will describe the principle of DA, thereafter conducting a simulation of the structure leading to an implementation and evaluation of the method.

DA is expected to have several advantages compared to normal multiplier-accumulate structures, e.g. high throughput and smaller energy consumption. The energy consumption is reduced as the switching activity introduced by memory-fetches is also reduced [Meher, 2010]. The principle of DA takes an outset in the general description of a FIR filter as a sum of products:

$$y = \sum_{n=0}^{N-1} a_n \cdot x[n] \tag{15.1}$$

where x can be rewritten if defined in 2's-complement binary representation as:

$$x[n] = -b[n,0] + \sum_{k=1}^{K-1} b[n,k] \cdot 2^{-k} \tag{15.2}$$

where $b[n,k]$ is the kth bit of the nth input, with $k = 0$ being the sign-bit. K is the number of bits in the 2's-complement binary representation. By combining the previous two equations, the following expression will appear:

$$y = \sum_{n=0}^{N-1} a_n \cdot \left[-b[n,0] + \sum_{k=1}^{K-1} b[n,k] \cdot 2^{-k} \right] \tag{15.3}$$

which can be rewritten as:

$$y = \sum_{k=1}^{K-1} \left[\sum_{n=0}^{N-1} a_n \cdot b[n,k] \right] \cdot 2^{-k} + \sum_{n=0}^{N-1} a_n \cdot (-b[n,0]) \tag{15.4}$$

By looking at the term in brackets, it is evident that this term only includes 2^N possible values. By pre-calculating these values and storing them in a memory, the filter can be represented by a memory block and an accumulator. The basic DA structure is illustrated on figure 15.1.

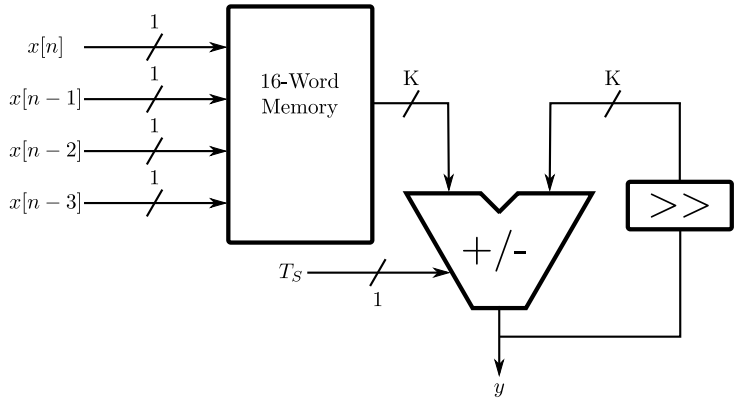

Figure 15.1: Basic DA structure with a filter length of 4 taps.

On figure 15.1 the structure for a filter with a length of 4 is shown. The 4 inputs are serially fed into the memory as address bits (LSB first), which finds the appropriate value to feed further into the adder/subtracter. The memory contains the LUT, which in this example will be $2^4 = 16$ words long. The adder/subtracter is controlled by the

signal T_S, which indicates when the MSB (the sign-bit) value is entering the adder/subtracter. The adder/subtracter should add all values except the sign-bit value which should be subtracted. The output from the adder/subtracter is forwarded into a shifter, which shifts the value one position to the right. As the result is shifted to the right, the shifter has to use sign extension in order to end up at a correct result. The shifter on figure 15.1 is also a register to hold the previous result. The right shift ensures that the result is assigned the proper weight, so the decimal point is in the right position in the end. When the subtraction of the sign-bit value has taken place, the result can be moved to the output y.

The LUT for the filter of length 4, will contain all the possible combinations of sums of the coefficients. The LUT is shown on table 15.1.

$\mathbf{b}[\mathbf{0},\mathbf{k}]$	$\mathbf{b}[\mathbf{1},\mathbf{k}]$	$\mathbf{b}[\mathbf{2},\mathbf{k}]$	$\mathbf{b}[\mathbf{3},\mathbf{k}]$	**LUT Contents**
0	0	0	0	0
0	0	0	1	a_3
0	0	1	0	a_2
0	0	1	1	a_2+a_3
0	1	0	0	a_1
0	1	0	1	a_1+a_3
0	1	1	0	a_1+a_2
0	1	1	1	$a_1+a_2+a_3$
1	0	0	0	a_0
1	0	0	1	a_0+a_3
1	0	1	0	a_0+a_2
1	0	1	1	$a_0+a_2+a_3$
1	1	0	0	a_0+a_1
1	1	0	1	$a_0+a_1+a_3$
1	1	1	0	$a_0+a_1+a_2$
1	1	1	1	$a_0+a_1+a_2+a_3$

Table 15.1: LUT of a filter with a length of 4.

The bits $b[0,k]$ to $b[3,k]$ are the bits from inputs $x[n]$ to $x[n-3]$ respectively.

To make it more clear how the DA filter works, a small example is considered. The example shows a filter with two taps, where the coefficients of these are called a_0 and a_1. The filter is considered when two input samples has been clocked into the filter. The inputs are:

$$x[0] = 10010110$$
$$x[1] = 11110101$$

and the LUT is formed as:

$\mathbf{b}[\mathbf{0},\mathbf{k}]$	$\mathbf{b}[\mathbf{1},\mathbf{k}]$	**LUT Contents**
0	0	0
0	1	a_1
1	0	a_0
1	1	a_0+a_1

Table 15.2: The contents of the LUT of a filter with a length of 2.

The first address for the LUT, is put together by the LSB of the two input values, which means that this is 01. The value for this address is a_1. This value goes into the accumulator and the output from this is then shifted once to

the right. The output is again a_1, as this is the first value from the LUT. The next address will then be 10, which contains the value a_0. This is fed into the accumulator, where the output from this then will be $a_0 + a_1 \gg 1$. This routine is performed for all bits in the input samples. The MSB is subtracted, why the final output from the filter can be explained as:

$$y[1] = -(a_0 + a_1) + a_1 \gg 1 + a_1 \gg 2 + (a_0 + a_1) \gg 3 + (a_0 + a_1) \gg 5 + a_0 \gg 6 + a_1 \gg 7$$

The DA method is usually seen as a slower way of calculating a sum of products due to the serial input, but if the number of bits, K, is lower than the number of taps in the filter, N, then it will actually be faster. This of course if parallel solutions is not taken into account. A problem is clearly the exponentially increasing memory size, which increases with the number of filter taps. This is illustrated on figure 15.2, where every value in the LUT is represented by two bytes, thereby yielding a 16 bits representation of the coefficients. The figure hereby illustrates the fact that the LUT must contain 2^{N+1} bytes when N is the number of taps.

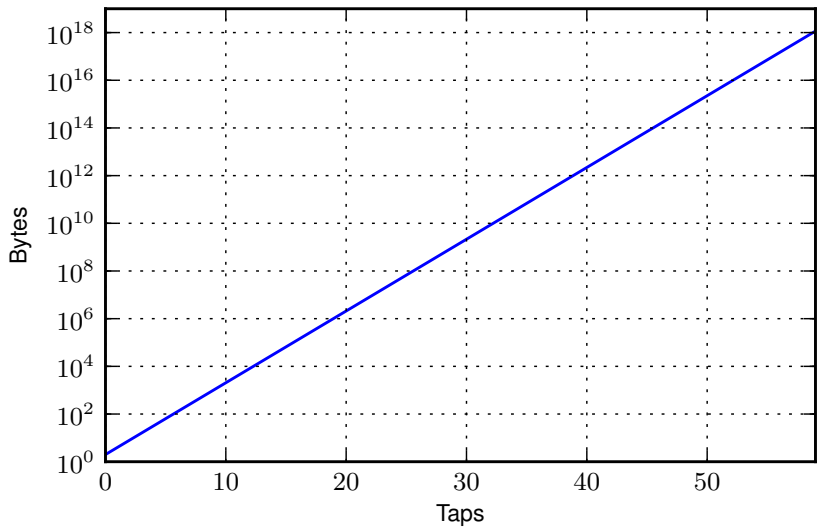

Figure 15.2: The relationship between taps and memory size.

It is seen on the figure, that the memory size is increasing exponentially. This causes a problem as the needed memory is at megabytes at 20 taps and at gigabytes at 30 taps. A way to overcome big memory blocks is to use several smaller blocks, as illustrated on figure 15.3.

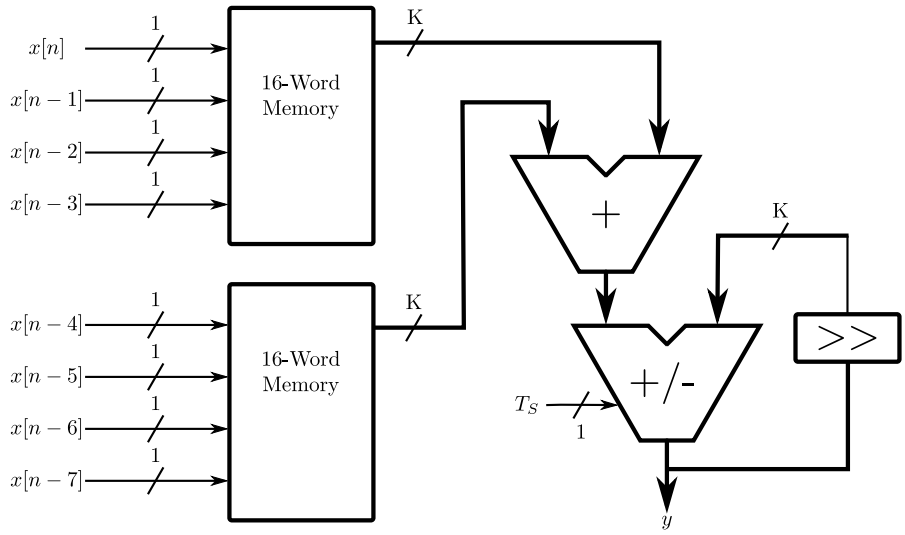

Figure 15.3: DA structure with a filter length of 8 and 2 memory blocks.

The overall memory size is still the same, but the size of the blocks will be smaller. To overcome the extra delay introduced by inserting an extra addition, pipeline registers can be inserted after every extra memory block and adder. By inserting pipeline registers, the numbers will ripple through the structure, generating the same throughput.

Several approaches for reducing the overall memory consumption of the DA method can be found in the literature [Meher, 2010], [White, 1989], [Mehendale and Sherlekar, 2001], but is not included here due to the time frame of the project.

15.1 Algorithmic simulation

The following section will simulate a DA structure to show that it produces the same output as the traditional multiplier solution.

At first, the simulated delayline can be created. This is done in the same way as in the simulation of multirate filtering in section 5.4 on page 26. The delayline is simulated as a ring buffer, where every row is a new time instance of the buffer and every column holds the time delayed versions of the input signal. The simulation is conducted by using an 8 tap filter, which means that the delayline is an 8 by 8 matrix. The input to the filter is three values: 0.1, 0.2 and 0.3, starting from time instance zero. The delayline matrix will then look like:

$$x = \begin{bmatrix} 0.1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.2 & 0.1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.3 & 0.2 & 0.1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.3 & 0.2 & 0.1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.3 & 0.2 & 0.1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.3 & 0.2 & 0.1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.3 & 0.2 & 0.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.3 & 0.2 & 0.1 \end{bmatrix}$$

The delayline is encoded into a 16 bit 2's complement representation, as the binary value should be used to find the proper address in the LUT.

The coefficients used in the filter is created by using the remez function like so:

```
1 filter = remez(8, [0, pi/2, pi/2+0.1, pi], [1, 0], Hz=2*np.pi)
```

Listing 15.1: Example of using the remez function in Python.

The coefficients is further rearranged in a LUT in the same way as described in table 15.1 on page 98, just extended to contain all the taps.

When the delayline and coefficients has been created, the simulation can calculate the output by using either DA or traditional direct form filtering. The DA algorithm looks as on listing 15.2.

```
1  y_da = list(np.zeros(len(dl)))
2  for k in range(len(dl)):
3      accm = 0.
4      for i in np.flipud(range(bits)):
5          address = ''
6          for j in range(taps):
7              address = address+dl[k][j][i]
8          if i != 0:
9              accm = (accm + lut.get(address)) * 2**(-1)
10         else:
11             accm = accm - lut.get(address)
12     y_da[k] = round(accm, 15)
```

Listing 15.2: Python implementation of a DA FIR filter.

The outermost *for*-loop runs through all the time instances. Under each time instance, the accumulator is resat, next, the address for the LUT is found by constructing a string which holds the bits at position i in the binary encoded delayline (dl). When the address has been created, the value is found from the LUT, added to the accumulator and shifted once to the right. When i reaches the MSB, the value from the LUT is subtracted from the accumulator and the output is saved in the list *y_da*.

The output from the direct form filter is created in the traditional way by computing the sums of product. The output is saved in a list called *y_df*. The output is calculated as:

```
1 y_df = list(np.zeros(len(dl)))
2 for j in range(len(dl)):
3     for i in range(len(filt)):
4         y_df[j] = y_df[j]+filt[i]*decode(dl[j][i], bits)
```

Listing 15.3: Python implementation of a DF FIR filter.

To compare the output from the two different filter, they are shown here:

$$y_{df} = \begin{bmatrix} -0.008971886593299 \\ -0.0430252119159 \\ -0.058229954726695 \\ 0.007770619055506 \\ 0.192478559415113 \\ 0.245409976849615 \\ 0.148543198576557 \\ -0.002602980967037 \end{bmatrix} \qquad y_{da} = \begin{bmatrix} -0.008971886593299 \\ -0.0430252119159 \\ -0.058229954726695 \\ 0.007770619055506 \\ 0.192478559415113 \\ 0.245409976849615 \\ 0.148543198576557 \\ -0.002602980967037 \end{bmatrix}$$

The output from the filters is equal which makes it possible to conclude that the DA filter works and can therefore be used as a multiplierless filter solution. It is noted that the delayline here is encoded as a 16 bits 2's complement number, but the rest of the calculations is performed in floating point. When converting the filters to fixed-point representations, rounding errors will be introduced. This is however expected and will equally be introduced in the direct form filter.

To summarise, the DA filter has been analysed and simulated, and it is seen that the DA filter generates the same output as the direct form filter. The DA filter can now be implemented and evaluated to see the difference in energy consumption.

15.2 Implementation

The following section will cover the implementation and hereunder the simulation at RTL level of the DA filtering method. The implementation will be performed on the Cyclone 3 development board from Altera, which is described in chapter 14 on page 89. The following will describe the VHDL code which generates the filter.

The overall block design of the DA filter, is the same as for the reference filter, and is shown on figure 15.4.

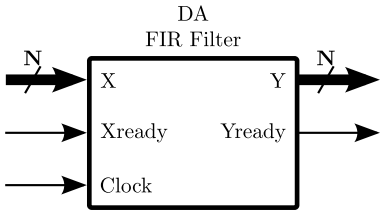

Figure 15.4: The DA filter block.

The overall block design holds three inputs and two outputs. The first input is the input to the filter, which is denoted *X* and is N bits wide. The second input is a signal, which is set high when the input is ready. This signal is denoted *Xready* and holds only a single binary value. The last input is the clock, which makes the filter run. At the output side, the first output is the output from the filter, which is denoted *Y* and is N bits wide. The second output is a signal to show when the output signal is ready. This is denoted *Yready*.

When looking inside the implementation of the DA filter, two very basic components are present. The first, an accumulator and second a RAM which holds the LUT. The two component blocks are shown on figure 15.5.

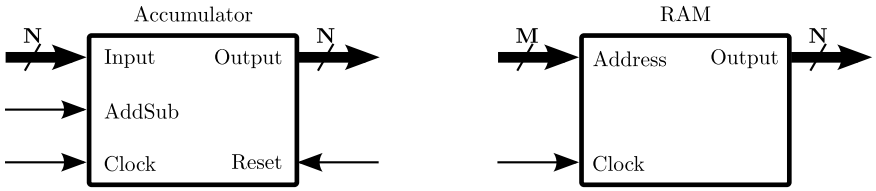

Figure 15.5: The RAM and accumulator blocks.

The accumulator has four inputs and a single output. The first input is the input to the accumulator, which is denoted *Input* and is N bits wide. The second input, *AddSub*, is a signal which indicates whether the input signal should be subtracted from or added to the accumulated value. When the signal is low, the input value is added and when the signal is high, the input value is subtracted. The last two inputs are signals to reset the accumulated value and the clock signal. The only output from the accumulator is the accumulated value, which is denoted *Output* and is a N bits wide signal. Before the accumulated value is moved to the output port, it is shifted one to the right, as indicated on figure 15.3 on page 100. The fundamental VHDL code of the accumulator is seen on listing 15.4.

```
1  if (Reset = '1') then
2      Oldresult <= X"0000";
3      Output <= X"0000";
4  else
5      if (AddSub = '0') then
6          Result := shift_right(Oldresult, 1) + signed(Input);
7      else
8          Result := shift_right(Oldresult, 1) - signed(Input);
9      end if;
10     Oldresult <= Result;
11     Output <= Result;
12 end if;
```

Listing 15.4: The fundamental code of the accumulator.

It is seen that when the *Reset* signal is high, the output and accumulated value (*Oldresult*) is set to zero. If the *Reset* signal is low, then the *Input* will be either added to or subtracted from the shifted accumulated value. When the accumulated value is shifted, there will be used sign extension in order to get a correct value. In the end, the result is saved and moved to the *Output* port.

The RAM has two inputs and a single output. The first input is the address at which the RAM should find an appropriate value. The address signal is M bits wide, where M is the number of taps. The second input is the clock signal. The output from the RAM is the value which is found at the address. The output is a N bits wide signal. When the RAM is implemented in VHDL, it can be designed as to use the on-board memory in the FPGA by using the Altera MegaWizard Plug-In Manager. If the RAM is not implemented in the on-board memory, then every bit will be implemented as a latch and will therefore use significant amount of space in the FPGA.

Another important part of the filter, is the delayline, which is build by an array of registers. The array holds N bits for every value of the M taps long delayline. The array is therefore created by $N \cdot M$ registers.

The DA filter is build as a simple state machine with four states. The four states are described in the following list:

State 0: The state waits for the *Xready* signal to be sat high. Thereafter; the filter is initialised and the first address to the RAM is composed. The first address is as described earlier the LSB of the values in the delayline. The initialisation sets the *Yready* and *AddSub* signals low and initialises thereafter the delayline pointer to the second bit. The delayline pointer points at the bit position of the values in the delayline, which should be composed to an address in the RAM.

State 1: This state is hold in $N-1$ clock cycles, as to run through the bits of the delayline values. The address for the RAM is changed at every clock cycle. This is done by increasing the pointer to the bit location by one, thereby changing the bits which the address is composed of.

State 2: The state sets the *AddSub* signal to high, which means that the output from the RAM is subtracted from the accumulated value. At the next clock cycle nothing is done, as this gives the accumulator time to calculate the final accumulated value.

State 3: The state moves the accumulator to the filter output, sets the *Yready* signal high, and resets the accumulator. The delayline is also affected, as every signal is moved one down in the array, to make room for a new input value.

When state 3 is done, state 0 is instantiated once again.

15.3 Simulation

The following section will describe the simulation performed on the VHDL code which is described in the above.

The following simulation will illustrate the output from a DA filter with 8 taps. As for the reference filter, the simulation will be performed in a testbench environment. The testbench is created in VHDL code and executed in Altera ModelSim. The testbench is designed in the same way as for the reference filter. This means that the inputs: *Clock*, *Xready* and *X* are generated, and the coefficients are programmed to all have the decimal value 0.5, which as a hexadecimal value will be 4000. The input, *X*, is starting from the hexadecimal value 4000 and this is subtracted by hex 2000 for every new input. The output from the testbench is shown on figure 15.6.

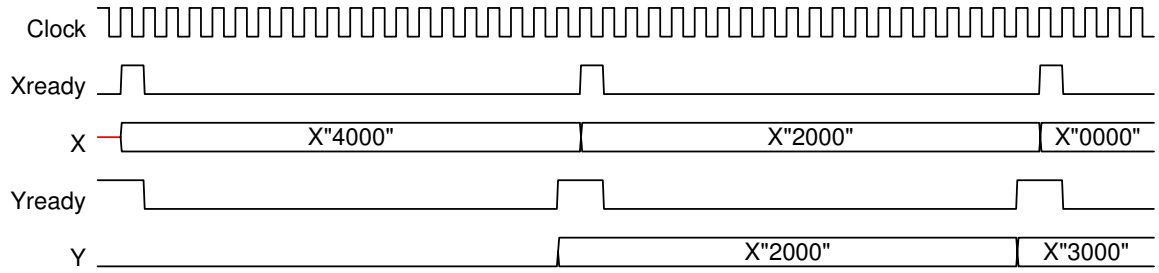

Figure 15.6: The timing diagram of the DA filter block for the two first periods.

On figure 15.6 it is seen that a new *X* is present at the time that *Xready* goes high. The filter start processing an output from the point where *Xready* goes high. When the filter is done processing, *Yready* goes high again as the output *Y* is set to the output value. The filter will always take 19 cycles to process a single input, as the number of bits in the implementation is kept fixed at 16 bits.

To assure that the output is correct, the simulation is executed for a longer period of time than on figure 15.6. This is seen on figure 15.7.

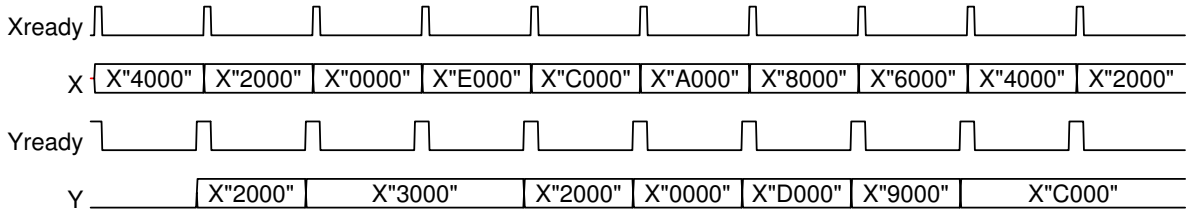

Figure 15.7: The timing diagram of the DA filter block for 10 periods.

As the filter contains a delayline of 8 values, and the output always is the half of the sum of the values in the delayline, it is seen that the output is correct. Due to the fact that the delayline contains 8 values, and the input is ramp signal which starts over after 8 values, the filter output will be the same after 8 periods, which is also seen on figure 15.7.

It is seen that the DA filter works as expected and can now be programmed to the FPGA and next evaluated.

15.4 Results

This section will present the results of the DA implementation in terms of energy consumption and use of LE's. The energy evaluation will be a comparison between the estimated energy computed from Altera's PowerPlay and measured energy from the filters implemented on Altera's Cyclone 3 FPGA. As input signal to the filters the pseudorandom number generator, described in appendix D on page 171 is used.

As described in the evaluation set in chapter 3 on page 13, there will be used a range of different filters lengths in the tests. However; this caused a problem in the evaluation of the DA implementation. This, as the memory requirement is exponentially increasing with the filter taps, the FPGA could not contain the LUT for filters with more than 14 filter taps. Therefore; the DA can not be tested as intended with the same evaluation set used for the other methods. It will instead be tested using filter taps from 2 to 14, increased in steps of 2. The measurements and estimations are conducted as described in appendix E.

The estimated and measured energy can be seen in figure 15.8. From this figure the relationship between estimated and measured energy can clearly be seen. They both follow the same trend, with a moderate increase in energy usage for the first 5 filters, upto 10 taps filter. For the filters with larger number of taps both the estimate and the measured energy indicates a more significant increase as the number of filter taps increases. Both the measured and estimated energy indicates that the energy dissipation increases rapidly at higher filter lengths. This can be explained, as the consumed space of the on-chip RAM increases in the same way. This is seen on figure 15.2 where the number of bytes increases exponentially as a function of the number of taps. As the RAM consumes more energy as more data is stored on it, this will explain the trend which is seen on figure 15.8. The average difference between the measured and estimated values is 11.78 %.

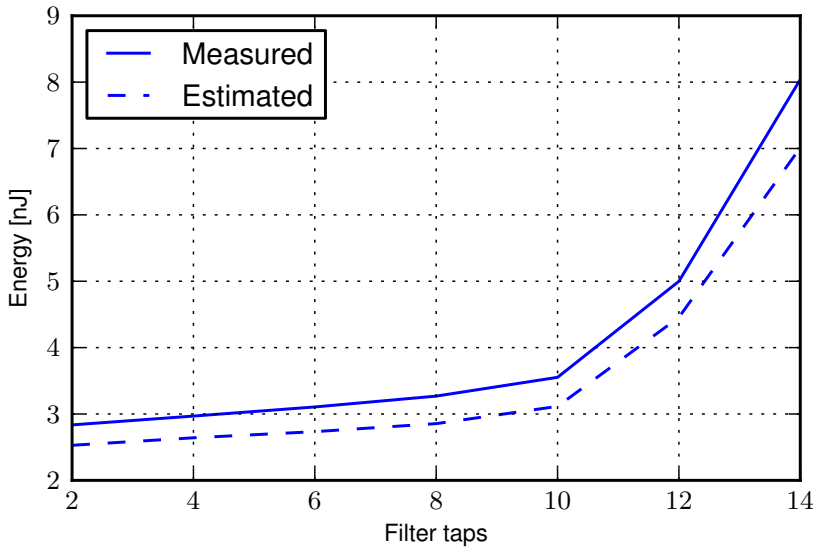

Figure 15.8: Measured and estimated energy dissipation from DA filter implementation.

Figure 15.9 shows the number of LE's used by the DA implementation. The use of LE's increases pretty much linearly as the number of filter taps is increased. This is due to the design, where the number of taps is reflected on the size of the delayline.

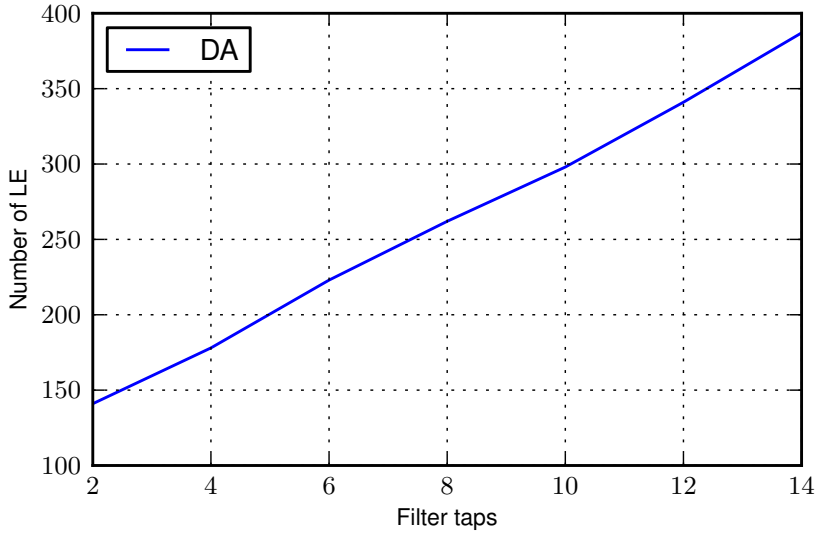

Figure 15.9: The number of LE's used in the DA implementation.

The measured energy consumption of the DA implementation is found to be between 2.84 and 8.04 nJ. The fact that the memory requirements increases exponentially is a major drawback for the method, as it can be used only for lower filter orders, using the Cyclone 3 FPGA. Using this implementation method, methods for reducing the overall area should be applied and ROM memory is recommended.

This concludes the results of the DA filter. More evaluation on the topic will be done in a later section, where all the hardware methods will be compared and their performance discussed.

15.5 Conclusion

This section will summarise and conclude on distributed arithmetic as a way of designing multiplierless filters.

The chapter starts out with a presentation of the method, it is here made clear that this method has exponentially growing memory requirements as the filter length is increased. The method is simulated with a 8 tap filter. The simulation shows that the method generates the same filter output as a traditional direct form implementation. The implementation was carried out, using an accumulator block and a RAM block. The simulation of the implementation showed that the input and output signals were timed expected and that the implementation generated the correct output.

The implementation of the DA filter on the Cyclone 3 FPGA immediately revealed problems introduced by the high memory requirement as filters above 14 taps could not be implemented. Because of this, the initial evaluation set could not be used. Instead the evaluation of the DA filter, considered filters with lengths from 2 to 14, increasing the filter length in steps of two taps. The estimates provided by Altera's PowerPlay Power Analyzer Tool and the measured energy are highly correlated and the difference between these were on average 11.78 %. Based on these tests the energy is found to be exponentially increasing as the filter length is increased. The energy consumption increases from 2.84 to 8.04 nJ, for filter lengths from 2 to 14.

Due to the fact that the DA filter only can be implemented on the Cyclone 3 for filter lengths upto 14, indicated that memory optimisation methods has to be used in future implementations. In addition, a ROM memory block will be recommended, as the type of RAM block in the FPGA consumes more energy when more space are being used.

16 Shift Based Filtering

The following chapter will analyse shift based filtering as a multiplierless filtering method and show a simulation of this. When analysing shift based multiplications common subexpression elimination (CSE) methods is naturally addressed in order to bring down the number of functional units of the filter. After simulating the shift based filter and verifying that it works as expected, an implementation on FPGA is carried out in order to evaluate the method compared to a traditional filter.

The following will introduce the principle of shift based multiplications.

Computing the output of a FIR filter is traditionally done by a summation of products. For instance; consider the coefficients $a_0 = 0.0111011$, $a_1 = 0.0101011$, $a_2 = 1.0110011$ and $a_3 = 1.1001010$ represented in 2's complement 8 bit fixed point representation [Mehendale and Sherlekar, 2001]. The output of the filter can be calculated using the weighted sum as:

$$y[n] = a_0 \cdot x[n] + a_1 \cdot x[n-1] + a_2 \cdot x[n-2] + a_3 \cdot x[n-3] \tag{16.1}$$

The weighted sum is realised as seen in figure 16.1. This is the most common way of realising FIR filters, where the input x is delayed before the multiplication with the coefficients. When replacing the multiplications with shifters and adders the output can be computed as:

$$\begin{aligned} y[n] =& x[n] \gg 2 + x[n] \gg 3 + x[n] \gg 4 + x[n] \gg 6 + x[n] \gg 7 \\ &+ x[n-1] \gg 2 + x[n-1] \gg 4 + x[n-1] \gg 6 + x[n-1] \gg 7 \\ &- x[n-2] + x[n-2] \gg 2 + x[n-2] \gg 3 + x[n-2] \gg 6 + x[n-2] \gg 7 \\ &- x[n-3] + x[n-3] \gg 1 + x[n-3] \gg 4 + x[n-3] \gg 6 \end{aligned} \tag{16.2}$$

In this realisation each non-zero bit in each coefficient are represented by shifting the value the appropriate number of times, depending on the bit placement in the coefficient. It is therefore seen that by using shifters and adders the multiplication can be eliminated.

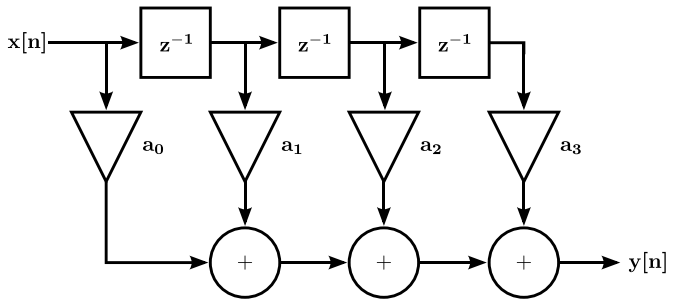

Figure 16.1: Data flow graph of weighted sum realisation of FIR filter.

Another way of realising the FIR filter is by using the transposed filter, which can be derived by re-timing the direct form filter. This realisation multiplies the same x to all coefficients, and delays them afterwards according to the appropriate coefficient. This realisation is illustrated in figure 16.2.

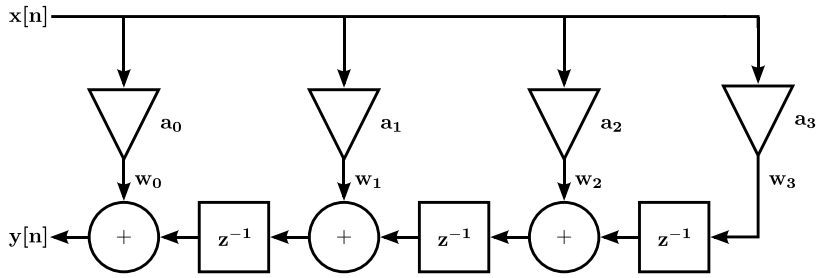

Figure 16.2: Data flow graph of the transposed realisation of a FIR filter.

The output of the transposed filter is computed as follows:

$$y[n] = w_0 + w_1[n-1] + w_2[n-2] + w_3[n-3] \tag{16.3}$$

where:

$$\begin{aligned} w_0 &= a_0 \cdot x[n] \\ w_1 &= a_1 \cdot x[n] \\ w_2 &= a_2 \cdot x[n] \\ w_3 &= a_3 \cdot x[n] \end{aligned} \tag{16.4}$$

as also denoted on figure 16.2. By replacing the multiplications by additions, subtractions and shifts the output yields:

$$\begin{aligned} w_0 &= x[n] \gg 2 + x[n] \gg 3 + x[n] \gg 4 + x[n] \gg 6 + x[n] \gg 7 \\ w_1 &= x[n] \gg 2 + x[n] \gg 4 + x[n] \gg 6 + x[n] \gg 7 \\ w_2 &= -x[n] + x[n] \gg 2 + x[n] \gg 3 + x[n] \gg 6 + x[n] \gg 7 \\ w_3 &= -x[n] + x[n] \gg 1 + x[n] \gg 4 + x[n] \gg 6 \end{aligned} \tag{16.5}$$

Using the previous shown weighted sum realisation of the filter, the output can be computed in 17 additions/subtractions and 16 shifts. The transposed realisation results in the same amount of logic operators, as this uses 17 additions/subtractions and 16 shifts, where two subtractions and one addition are used in calculation of the delay-line. The term *logic operators* (LO) will in the following be used instead of additions and subtractions.

In the computations of the output using the multiplierless realisations in eq. (16.5), the shifts corresponds to each of the non-zero bits in the coefficients. This meaning that the more non-zero bits in a coefficient the more LO's and shifters are required. As of this, the complexity of the coefficients dominates the complexity of the FIR filter. Several methods that aim at reducing the number of shifters and LO's exist and one will be presented in the following.

16.1 Common subexpression elimination

The following section presents CSE using an example before going into detail with two types of CSE.

Common subexpression can be used to reduce the number of LO's and shifters in the computation of FIR filters. Generally speaking, CSE aims at determining identical parts between coefficients or within a coefficient, and compute this part once and for all, in order to eliminate redundant computations. This is shown here using the coefficients mentioned previously with an example.

Considering the computation of the output of the transposed filter, some similarities are found between some of the outputs. The expression $x[n] \gg 2 + x[n] \gg 3$ is used to compute the multiplication within two of the coefficients. LO's can be saved by pre-computing this expression. Calling it x_{23} the output can be computed as:

$$\begin{aligned} w_0 &= x_{23} + x[n] \gg 4 + x[n] \gg 6 + x[n] \gg 7 \\ w_1 &= x[n] \gg 2 + x[n] \gg 4 + x[n] \gg 6 + x[n] \gg 7 \\ w_2 &= -x[n] + x_{23} + x[n] \gg 6 + x[n] \gg 7 \\ w_3 &= -x[n] + x[n] \gg 1 + x[n] \gg 4 + x[n] \gg 6 \end{aligned} \tag{16.6}$$

These computations uses 16 LO's, including the addition in computing x_{23} and the delayline, and 14 shifts, including the two needed to compute x_{23}. The subexpression $x[n] \gg 6 + x[n] \gg 7$ is used in three of the coefficients, and can be be precomputed as $x_{23} \gg 4$ resulting in the computation:

$$\begin{aligned} w_0 &= x_{23} + x[n] \gg 4 + x_{23} \gg 4 \\ w_1 &= x[n] \gg 2 + x[n] \gg 4 + x_{23} \gg 4 \\ w_2 &= -x[n] + x_{23} + x_{23} \gg 4 \\ w_3 &= -x[n] + x[n] \gg 1 + x[n] \gg 4 + x[n] \gg 6 \end{aligned} \tag{16.7}$$

Including this common subexpression results in computations requiring 13 LO's, including the one needed to compute x_{23} and the delayline, and 11 shifts. The 11 shifts include two for computing x_{23}. Further reduction in LO's and shifters can be done by combining $x[n] \gg 4 + x_{23} \gg 4$ in x_{234}, since this combination is used in two of the coefficients:

$$\begin{aligned} w_0 &= x_{23} + x_{234} \\ w_1 &= x[n] \gg 2 + x_{234} \\ w_2 &= -x[n] + x_{23} + x_{23} \gg 4 \\ w_3 &= -x[n] + x[n] \gg 1 + x[n] \gg 4 + x[n] \gg 6 \end{aligned} \tag{16.8}$$

The final computation require 12 LO's and 9 shifts, where the original computation required 17 LO's and 16 shifts. This example eliminated common subexpression within a coefficient (CSWC). The general procedure for reducing LO's and shifters, includes two general forms of common subexpressions. That is CSWC and common subexpression across coefficients (CSAC). The following two sections will describe both types of subexpressions and show examples on how to eliminate some of these to reduce the overall number of LO's.

16.1.1 Common subexpression within coefficient (CSWC)

CSWC is also known as horizontal CSE in some literature. This method finds common subexpression within the coefficients, as can be seen from coefficient $a_2 = 1.0110011$. In this coefficient the subexpression 11 is found two places, in bit location 2 and 3, and in 6 and 7. Because of this, the expression for bit location 2 and 3 are computed as $x_{23} = x[n] \gg 2 + x[n] \gg 3$ and used for calculating both subexpressions. The initial subexpression, w_2, as shown in eq. (16.5), is:

$$w_2 = -x[n] + x[n] \gg 2 + x[n] \gg 3 + x[n] \gg 6 + x[n] \gg 7 \tag{16.9}$$

Using CSWC the expression is reduced to:

$$w_2 = -x[n] + x_{23} + x_{23}[n] \gg 4 \tag{16.10}$$

The new subexpression requires 3 shifts and 4 LO's, compared to the initial realisation which required 4 shifts and 5 LO's.

In a CSWC elimination algorithm other subexpressions would normally also be searched for, such as the bit combinations [111], [101], [1001] and [1111] etc..

16.1.2 Common subexpression across coefficients (CSAC)

Following section describes the CSAC. It also provides a short simulation to illustrate the effects of CSWC and CSAC.

CSAC is also known as vertical CSE, which refers to subexpressions being used in several coefficients. This can be seen from the coefficients used previously and now shown in table 16.1, where now the goal is to eliminate redundant computations across coefficients. In table 16.1 the first row numbers indicate the number of right shifts needed in the computation.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
a_0	0	0	1	1	1	0	1	1
a_1	0	0	1	0	1	0	1	1
a_2	1	0	1	1	0	0	1	1
a_3	1	1	0	0	1	0	1	0

Table 16.1: Filter coefficients put into table-form. Each column refers to the number of shifts needed.

This initial computation use 17 LO's and 16 shifts. When looking for common subexpressions across the coefficients in table 16.1, the subexpression $\begin{bmatrix}1\\1\end{bmatrix}$ are found between many of the coefficients. The subexpressions are marked in table 16.1. The subexpression refers to the following:

$$\begin{bmatrix}1\\1\end{bmatrix} = x[n] + x[n-1] = x_2 \tag{16.11}$$

The output, y, is formed as previously described by eq. (16.3).The coefficients are searched for the subexpression $\begin{bmatrix}1\\1\end{bmatrix}$, finding it in bit position 2, 4, 6 and 7 for the coefficients a_0 and a_1, and in bit position 0 and 6 for coefficient a_2 and a_3. Using this subexpression $\begin{bmatrix}1\\1\end{bmatrix}$ the output can be found as:

$$\begin{aligned} y = & x_2[n] \gg 2 + x[n] \gg 3 + x_2[n] \gg 4 + x_2[n] \gg 6 + x_2[n] \gg 7 \\ & - x_2[n-2] + x[n-2] \gg 2 + x[n-2] \gg 3 + x_2[n-2] \gg 6 + x[n-2] \gg 7 \\ & + x[n-3] \gg 1 + x[n-3] \gg 4 \end{aligned} \tag{16.12}$$

The subexpressions found between the two first coefficients are not delayed, while for the subexpressions in the two last coefficients that are delayed. The delayed subexpressions is written as $x_2[n-2]$. The bits that are not included in a subexpression, are assigned the appropriate delay depending on which coefficient they are a part of. Using $\begin{bmatrix}1\\1\end{bmatrix}$ as a common subexpression, as described above, the number of LO's can be reduced to 12, including the one needed in order to compute the subexpression. The number of shifts are reduced to 11.

Developing an algorithm for CSE, CSWC elimination are often done prior to CSAC elimination. Because of this the effects of the CSAC elimination are minor. This can be seen in figure 16.3. In this figure the reduction in percent of the initial number of LO's achieved by the CSWC and CSAC elimination are shown. Investigating filters with orders up to 100 the largest reduction in LO's caused by CSAC elimination was found to be 2.08 % of the initial number of LO's, while for the CSWC elimination was found to be 60.6 %. Due to the minor significance of the CSAC elimination, this might be omitted without affecting the final result significantly. CSAC is omitted in the following implementation of the CSE method.

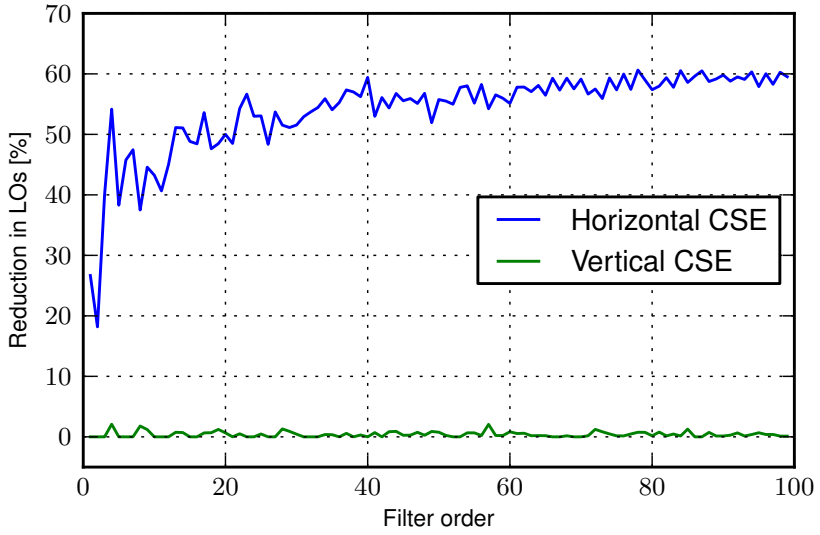

Figure 16.3: Reduction of LO's by CSWC and CSAC elimination.

16.2 Canonical signed digit vs. binary 2's complement

Following section provides a discussion on whether canonical signed digits or binary 2's complement is the most efficient number representation concerning common subexpression elimination.

Many algorithms for CSE uses canonical signed digit (CSD) for representation of the filter coefficients. CSD is another way of representing fixed point binary numbers by using three states; -1, 1 and 0 instead of the normal two; 0, 1. For instance will 23 be represented as +10-100-1 as this expands to $2^5 - 2^3 - 2^0$, where binary 2's complement would have been represented as 10111.

As mentioned earlier, the number of ones in the coefficients determine the number of LO's and shifts required to compute the output of the FIR filter. Using CSD, the number of non-zero bit are reduced by 50 % on average compared to binary 2's complement form [Mahesh and Vinod, 2008]. This is why many chose to represent the filter coefficients as CSD.

As the CSD has reduced number of ones, there are also a reduced possibility to form common subexpressions compared to binary representation, thus leaving more unpaired bits, as illustrated in figure 16.4. Unpaired non-zero bits are bits that are not included in any of the subexpressions and is therefore expensive as they will need one LO each. Seen from figure 16.4, the number of unpaired bit are significant higher using CSD representation compared to binary representation for the higher filter values. The number of unpaired bits using CSD is larger for all filter orders, and the difference increases as the filter order increases. As for the binary representation the number of unpaired bit does not increase significant.

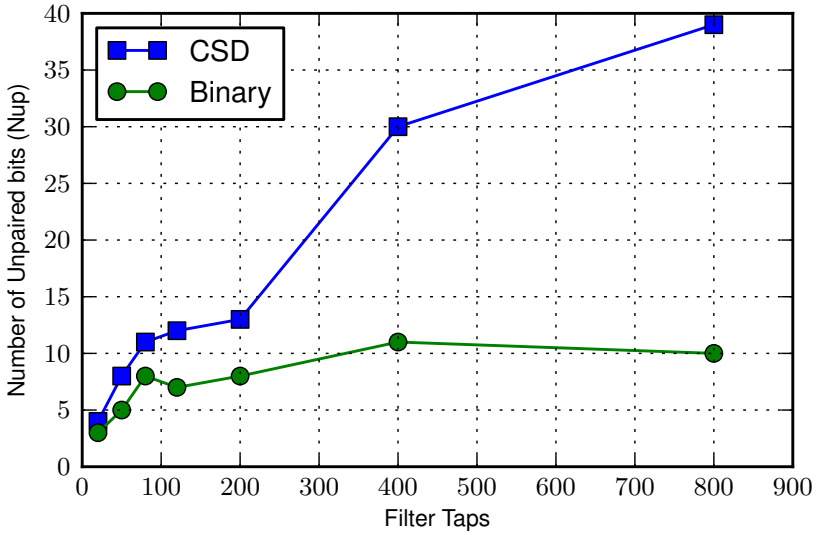

Figure 16.4: Number of unpaired bit using CSE on CSD and binary coefficients [Mahesh and Vinod, 2008].

[Mahesh and Vinod, 2008] describes three factors for the LO cost of a CSE method. These being the total number of non-zero bits in the coefficients, the number of common subexpressions that can be formed and the number of unpaired bits. Using these factors an expressions of the number of LO's needed can be formed as:

$$N_{LO} = \alpha \cdot N_{nz} - \beta \cdot N_{cs} + \gamma \cdot N_{up} \tag{16.13}$$

where N_{nz} is the number of non-zero bits, N_{cs} is the number of common subexpressions and N_{up} is the number of unpaired bits.

[Mahesh and Vinod, 2008] investigates the significance of each factors using FIR filters of different lengths and with coefficients of different word lengths. The filters used by [Mahesh and Vinod, 2008] are mainly intended for wireless communication receivers which are relatively narrow banded, as the adjacent channels are normally located close to the current band. The filters are designed with different pass-band and stop-band, all having relatively narrow transition band. The filters are believed to be representative for more applications than just wireless communication and will be found representative for this project also. Based on these filters the statistical means of α, β and γ is found in [Mahesh and Vinod, 2008] to be:

$$N_{LO} \approx 0.2345 \cdot N_{nz} - 0.6643 \cdot N_{cs} + 4.0487 \cdot N_{up} \tag{16.14}$$

As can be seen from this equation the number of unpaired bits is the most significant factor in terms of determining the total number of LO's. As CSD will have more unpaired bits than the binary counterpart, a binary representation of the filter coefficients will based on these results result in a lower number of LO's. For that reason; the further investigation on this topic will only concern binary representation, despite the initial statement that coefficients having the least amount of ones would be preferable. Due to the relative big difference between the first two weights, α and β, and the last weight, γ, a change of application is not expected to change these three statistical means, such that α and β will become more significant than γ. Due to this expectation, the weights are found to be representative.

16.3 Binary subexpression elimination

This section presents an algorithm for binary CSE, from here on called binary subexpression elimination (BSE). The result of the algorithm is presented using a 4 tap filter as example.

The algorithm for BSE take an offset in the methods of CSWC and CSAC. The following will illustrate the algorithm by an example. The following algorithm will only address CSWC due to the earlier discussion on CSAC. The filter coefficients are created by the Python function *remez* and are therefore created with linear phase property. The initial number of LO's used by the filter, is found as the number of ones in the binary representation of the filter coefficients minus one:

```
1 for i in range(len(coef)):           # For all coefficients
2     init_lo += coef[i].count('1')    # Count number of ones
3 init_lo -= 1                         # Subtract one
```

Listing 16.1: Python implementation of the number of LO's needed for direct implementation with adders ans shifts.

Using this as a starting point, the elimination of CSWC is used to eliminate the common subexpressions within the coefficients. This is done by a search for the subexpressions 11, 101, 1001, 111, 1111 and 11101, replacing these expressions with the numbers 2, 3, 4, 5, 6 and 7 respectively. This as to make it easier to find the subexpressions later on in the algorithm. The subexpressions are chosen, as these are found to be the ones which appear most often in filter coefficients [Mahesh and Vinod, 2008]. The subexpressions are searched for in this specific order, as this favours the subexpressions with the least amount of LO's. The algorithm is formed as:

```
1 for i in range(len(subexpression)):      # For all subexpressions
2     for j in range(len(coef)):           # For all coefficients
3         coef[j] = coef[j].replace(subexpression[i], replacement[i])
4                                          # Replace the subexpressions by their proper number
```

Listing 16.2: Python implementation of elimination CSWC.

where the coefficients are noted as a list, $coef$, the subexpressions and their respectively replacement is also denoted as lists.

The result from elimination of CSWC of a 4-tap filter using 8 bit representation can be seen in listing 16.3. The elimination of CSWC has reduced the number of LO's from 13 to 10, as can be seen here:

```
1 Initial coefficients: ['00010110', '01000111', '01000111', '00010110']
2 Initial number of LO's: 13
3 Coefficients after CSWC: ['00010200', '01000500', '01000500', '00010200']
4 Number of LO's after CSWC: 10
```

Listing 16.3: Result of elimination CSWC.

Using this relatively simple example using a 4-tap filter representing the filter coefficients using 8 bit, gives the reader an overview over the methods of eliminating CSWC. In this example the number of LO's are reduced with 23 %.

16.4 Software simulation

The following section will show a simulation of the shift based filtering method and the BSE algorithm, which is introduced to see that the method produce the same output as a normal direct form filter.

As a starting point, the delayline is created in the same way as in section 15.1 on page 100. The filter is here simulated as a 8 tap filter. This creates a delayline which looks as:

$$x = \begin{bmatrix} 0.1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.2 & 0.1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.3 & 0.2 & 0.1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.3 & 0.2 & 0.1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.3 & 0.2 & 0.1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.3 & 0.2 & 0.1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.3 & 0.2 & 0.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.3 & 0.2 & 0.1 \end{bmatrix}$$

where every row is a time instance of the delayline.

The coefficients is created by using the *remez* function in Python and are encoded into 2's complement representation with a length of 16 bits. When both delayline and coefficients has been created, the filter output can be calculated by the shift based filter and the direct form filter, in order to compare these. The shift based multiplication is simulated in Python in floating point as:

```
1  count = coef_copy[j].count(item[k])
2  while count != 0:
3      shift = coef_copy[j].rfind(item[k])
4      if (shift > 0):
5          if (item[k] == '1'):
6              output += dl[i][j]*2**(-shift)
7          if (item[k] == '20'):
8              output += dl[i][j]*2**(-shift) + dl[i][j]*2**(-shift-1)
9          if (item[k] == '300'):
10             output += dl[i][j]*2**(-shift) + dl[i][j]*2**(-shift-2)
11         if (item[k] == '4000'):
12             output += dl[i][j]*2**(-shift) + dl[i][j]*2**(-shift-3)
13         if (item[k] == '500'):
14             output += dl[i][j]*2**(-shift) + dl[i][j]*2**(-shift-1) + dl[i][j]*2**(-shift-2)
15         if (item[k] == '600'):
16             output += dl[i][j]*2**(-shift) + dl[i][j]*2**(-shift-1) + dl[i][j]*2**(-shift-2) +
                   dl[i][j]*2**(-shift-3)
17         if (item[k] == '700'):
18             output += dl[i][j]*2**(-shift) + dl[i][j]*2**(-shift-1) + dl[i][j]*2**(-shift-2) +
                   dl[i][j]*2**(-shift-4)
19         tmp = list(coef_copy[j])
20         tmp[shift] = '0'
21         coef_copy[j] = "".join(tmp)
22     if (shift == 0):
23         output -= dl[i][j]
24         tmp = list(coef_copy[j])
25         tmp[shift] = '0'
26         coef_copy[j] = "".join(tmp)
27     count = coef_copy[j].count(item[k])
```

Listing 16.4: The simulation of the shift based filter.

It can be seen that the simulation iterates over the *count* variable, which denotes the number of times a subexpression (here denoted as *item*) is found in the coefficient list, $coef\_copy$. If the *count* variable is not zero, then the

position of the subexpression is denoted in the *shift* variable. The *shift* is then used in connection with the subexpression to add the correct shifted value of the appropriate delayline value. It is noted that when the shift position is zero, then the sign-bit is set and should therefore be subtracted instead of added. This is not directly mentioned in the article on BSE [Mahesh and Vinod, 2008] and is therefore explicitly mentioned here. When the value has been added to or subtracted from the *output* variable, then the subexpression is removed from the coefficient list and the *count* variable is updated. When the coefficient list, $coef\_copy$, only contains zeros, the *output* variable is saved as the filter output, and the next time instance is instantiated.

The direct form filter is created by the following relatively short algorithm:

```
1 y_df = list(np.zeros(len(dl)))  # Instantiate a list of zeros
2 for j in range(len(dl)):        # For all rows in the delayline matrix
3     for i in range(len(coef)):  # For all taps
4         y_df[j] = y_df[j]+decode(coef_init[i], nbits)*dl[j][i]
5                                 # Decode the coefficients, multiply them by the delayline and
                                      sum the results
```

Listing 16.5: The simulation of the direct form filter.

As the coefficients is encoded in 2's complement representation, they has to be decoded as done by the *decode* function.

The output from the two filters, at the time instances introduced by the delayline, is:

$$y_{df} = \begin{bmatrix} -0.00897216796875 \\ -0.043023681640625 \\ -0.058230590820313 \\ 0.007763671875 \\ 0.192471313476563 \\ 0.245407104492188 \\ 0.14854736328125 \\ -0.002597045898438 \end{bmatrix} \qquad y_{sb} = \begin{bmatrix} -0.00897216796875 \\ -0.043023681640625 \\ -0.058230590820313 \\ 0.007763671875 \\ 0.192471313476563 \\ 0.245407104492187 \\ 0.14854736328125 \\ -0.002597045898437 \end{bmatrix}$$

It is seen that the outputs are equal with exception from the last decimal in some of the values. This result makes it clear to see that the shift based filter works in the correct way, as it creates the same output as the direct form filter. The error observed is neglectable due to the size of this.

It should be noted that the above simulated results are found by implementations of the filters in a floating point environment. Except from the coefficients which is encoded in 16 bits 2's complement, all other variables are floating point. When the filter is implemented in a fixed point environment, truncation errors will appear.

To summarise, the shift based implementation of a multiplication has been analysed and this has been optimised by the BSE method to lower the amount of logic operators. The simulation of the shift based filter showed that it gives the same output as the direct form filter and it can therefore be concluded that the shift based filter can be used as a multiplierless alternative to the direct form filter.

16.5 Implementation

This section describes the implementation of the shift based filter using VHDL onto a FPGA. The implementation is done on the Cyclone 3 development board from Altera. This includes a Cyclone 3 FPGA which is briefly described in chapter 13 on page 85.

As an example of how the filter can be realised, figure 16.5 shows a realisation of a 8 tap filter with coefficients encoded by 8 bits. The coefficients are generated by the *remez* function in Python. The coefficients used are:

['11110101', '11100000', '00011000', '00111010', '00111010', '00011000', '11100000', '11110101']

BSE is used to lower the amount of logic operators. The coefficients are now:

['17000001', '12000000', '00020000', '00700000', '00700000', '00020000', '12000000', '17000001']

Figure 16.5 shows the DFG of the filter realisation. In this figure the numbers next to the vertical lines represents the appropriate shift for realising the current subexpression or *w*.

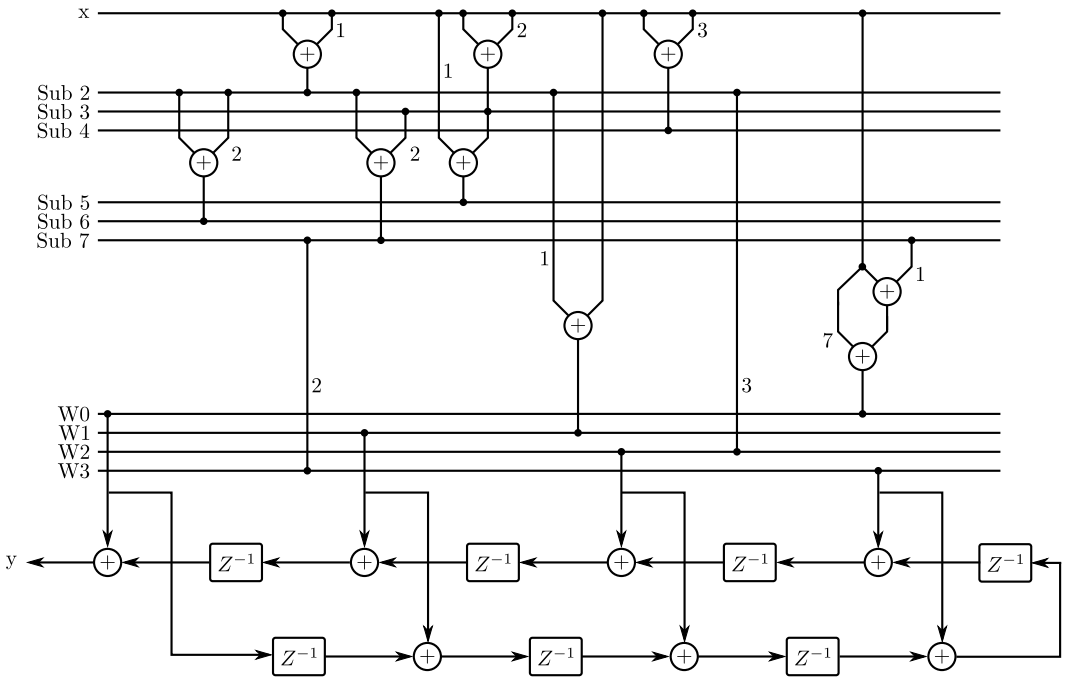

Figure 16.5: DFG of BSE Realisation.

Based on this DFG the VHDL, for implementing this structure on the FPGA, is written. The shift based filter block can be seen in figure 16.6. The filter has the input signal, *X*, which is N bits wide. Likewise, the output signal, *Y*, is N bits wide. *Xready* and *Yready* indicates respectively when the input and output signals are ready to be used. The filter is driven by a clock which also is an input to the filter.

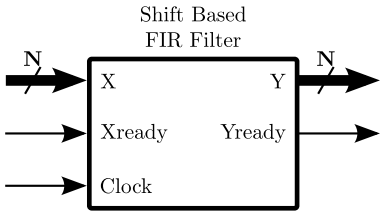

Figure 16.6: The shift based filter block.

The shift based filter is realised as a state machine using three states. The algorithm starts with *State 0* where the three first subexpressions are computed, 11, 101 and 1001. These are computed in the first state as all these only include two none-zero bits. The computation of these are shown in listing 16.6.

```
1 sub_2 := x + shift_right(x,1);
2 sub_3 := x + shift_right(x,2);
3 sub_4 := x + shift_right(x,3);
4 current_state <= S1;
```

Listing 16.6: VHDL code for defining subexpressions 11, 101, 1001.

After computing these subexpressions the current state is sat to *State 1*. *State 1* compute the remaining subexpressions: 111, 1111, 11101, based on the computation of the first subexpressions. The computation of these can be seen in listing 16.7. After computing these the current state is sat to be *State 2*.

```
1 sub_5 := shift_right(x,1) + sub_3;
2 sub_6 := sub_2 + shift_right(sub_2,2);
3 sub_7 := sub_2 + shift_right(sub_3,2);
4 current_state <= S2;
```

Listing 16.7: VHDL code for defining subexpressions 111, 1111, 11101.

In *State 2* the subexpressions are used to compute the w for all coefficients. By pre-computing the subexpressions they are computed once and for all, in order to use the same subexpressions in several of the coefficients without introducing additional additions. The calculation of w is done in parallel. In *State 2* the w for all coefficients are computed and directly included in the delayline. This can be seen in listing 16.8, where all w are computed using the appropriate shift of x or a subexpression. In addition to this, the next w is also added to the current w, this implements the delayline of the transposed filter directly. The first coefficient, $w(0)$, is used as the output from the filter, y.

```
1  Y <= -x+shift_right(sub_7,1)+shift_right(x,7)+w(1);
2  w(1) := -x+shift_right(sub_2,1)+w(2);
3  w(2) := shift_right(sub_2,3)+w(3);
4  w(3) := shift_right(sub_7,2)+w(4);
5  w(4) := shift_right(sub_7,2)+w(5);
6  w(5) := shift_right(sub_2,3)+w(6);
7  w(6) := -x+shift_right(sub_2,1)+w(7);
8  w(7) := -x+shift_right(sub_7,1)+shift_right(x,7);
9  Yready <= '1';
10 current_state <= S0;
```

Listing 16.8: VHDL code computing the w and delayline.

After computing all w and y, *Yready* is sat to high in order to signalise that an output is ready and that a new input x can be loaded into the filter. The current state is sat to *State 0* again.

16.6 Simulation

The simulation of the implementation of the shift based filter is presented here. The shift based filter is simulated using the Altera ModelSim tool in order to verify that the signal flow in the design is as expected. To test the filter the same testbench used for the reference filter is used to generate the input signal. The same testbench can be used as the filters inputs and outputs are the same as the reference filter. The testbench provides the input x, the *Xready* signal and the clock.

Figure 16.7 shows the timing diagram for two periods of the shift based filtering. For illustration purposes, the filter coefficients will all have a value of 0.5 or the hexadecimal value 4000. The input x will be the same as in the simulation of the reference filter in section 14.1.1, as it easily can be verified that the result generated from these two is correct.

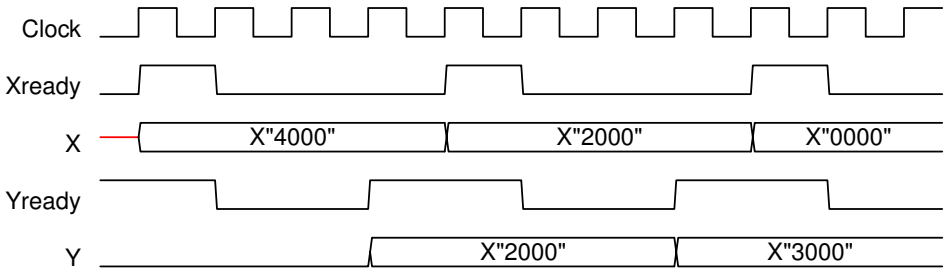

Figure 16.7: The timing diagram of the shift based filter block for the two first periods.

When *Xready* toggles to high, a new input value in introduced. *Yready* is low during the processing of the input. When the *Yready* signal switches to high a new output is ready on *Y*. The filter takes three clock cycles to compute, as the filter start computing immediately as the *input* is available. As figure 16.7 only illustrates two inputs figure 16.8 is made in order to verify that all resulting outputs are correct. Figure 16.8 shows 10 periods of the input signal.

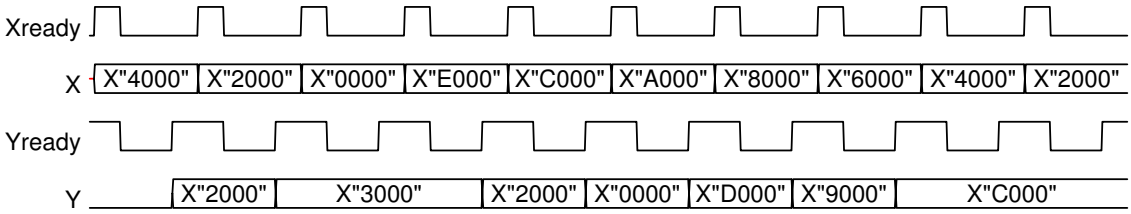

Figure 16.8: The timing diagram of the shift based filter block for 10 periods.

This simulation results in the same output as for the reference filter, using the same input and coefficient set. It can hereby be concluded that the shift based filtering implementation works as expected.

16.7 Results

This section contains the results of the implementation of the shift based filtering method in terms of energy and use of logic elements. The energy dissipation is found both by estimates and by measurements. The estimate is provided by Altera's PowerPlay Power Analyzer and the measurements are done according to the procedure described in appendix E.

Filters of different length will be evaluated. Filters with taps from 8 to 120 will be used, where the number of taps will be increased in steps of 8. This evaluation set is presented in chapter 3 on page 13. For the further testing, a pseudorandom input signal will be used. This is described in appendix D. Using a pseudorandom input signal instead of the input signal used for simulations has the benefit that it uses long time before it repeats itself and therefore generates random signal toggles in the filter. This is a necessary part of the implementation, as no specific application and thereby input is introduced in this project.

Figure 16.9 shows the estimated and measured energy dissipation of the shift based filters when implemented on an Cyclone 3 from Altera. From figure 16.9 it is clear to see that there is a correlation between the estimated and measured energy, as they follow the same trend. Both the estimated and measured energy increases as the number of filter taps is increased.

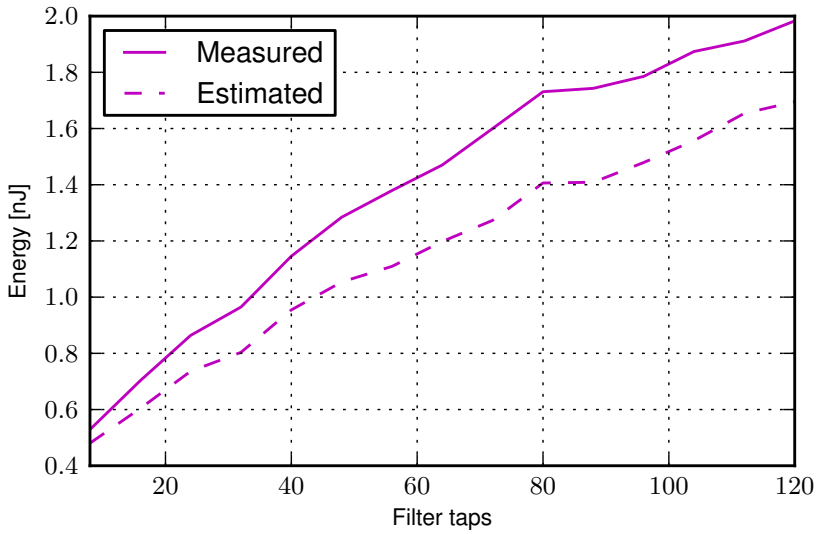

Figure 16.9: Estimated and measured energy from shift based filter implementation.

The measured energy is found to be between 0.49 and 1.98 nJ for filters with 8 to 120 taps. Altera claims that the PowerPlay Power Analyser can estimate the energy usage for a design which is within $\pm 10\%$ of the actual usage [Altera, 2010]. This is however not the case in this implementation. The average error for the filters shown on the graph is 16.5 %, which is higher than the 10 % specified by Altera. The error spans from 9 to 20 %. From this it can be concluded, that the claim of a precision within 10 % by Altera, is not the case for this specific implementation. The estimation is however seen as a good result, as it describes the trend and fits within a 20 % margin.

When evaluating the shift based filter in terms of LE's, the number of LE's is provided by the Altera compiler. The consumption of LE's can be seen in figure 16.10. The LE usage increases approximately linearly.

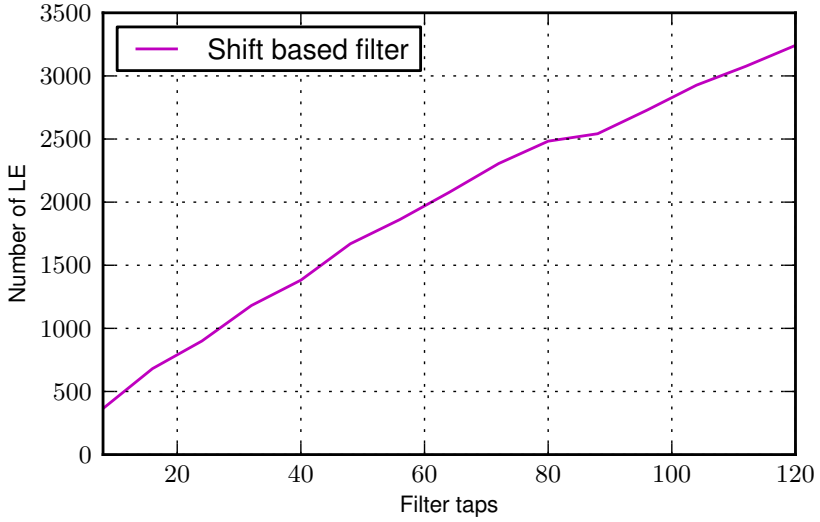

Figure 16.10: Logic elements used in the shift based filter implementation.

The reason why the number of elements increases is due to the fact that the shift based filter is implemented with parallel computations, computing all w at the same time. The output of the filter is computed in three clock cycles independent of the number of taps in the filter or the number of bits representing it. This also means that more LE's must be allocated when implementing a larger filter compared to a filter with lower filter order. However; designing the shift based filter in a sequential manner would result in a smaller amount of LE's, on the compromise of an increased computation time.

16.8 Conclusion

This section serve the purpose of summarising and concluding on the shift based filter as a method of designing multiplierless filters.

The chapter starts out by presenting the principle of how multiplications can be replaced with shifts and additions. Common subexpression elimination is introduced as a method of further reducing the number of shifts and logic operators, by finding common elements in the filter coefficients. Two types of common subexpression elimination were investigated, common subexpressions within coefficients, and common subexpressions across coefficients. Software simulations did however show that the effects of eliminating common subexpressions across coefficients were so small compared to the effects of eliminating common subexpressions within coefficients, that it could be left out of the algorithm.

For this reason the further simulation and implementation of the shift based filtering method only considered common subexpressions within coefficients. The software simulation is done with a 8 tap filter and this verifies that the shift based filter computes the same output as a direct form filter. The implementation presents the structure of the VHDL code and the simulation of this focuses on the timing of the signals in the filter. It is verified that the input and output signals are available at the correct clock cycles and that the output is correct.

The results showed a comparison of the estimated and measured energy dissipation of filters with lengths from 8 to 120. The tests shows that the estimates provided from Altera's PowerPlay Power Analyzer Tool are a fairly good estimate (within 20 %). The energy dissipation increases linearly as the filter length is increased.

17 Multirate Filtering

As described earlier in chapter 5 on page 21, multirate decimation can be used to lower the overall computational complexity of a filter. The evaluation of the software implementation showed how the energy dissipation can be optimised up to 20 %. This is a good argument for also trying to use the method in a hardware implementation. The following chapter will consider the method implemented on a hardware programmable platform. In addition to the multirate filter described in the software part of the report, the following filter will contain decimated filters which are calculated by using the shift based filtering method described in the previous chapter. This way of constructing a multiplierless filter is not seen before and is a method proposed by the project group.

Due to the resemblance in the methods, the following chapter will not repeat the background and theory of multirate filtering. In stead the chapter will jump directly to the description of the implementation. After this description, a simulation of the implementation will be carried out. In the end of the chapter, the results gathered from evaluation of the filter implementation will be shown and a conclusion will reflect on the findings.

17.1 Implementation

This section describes the implementation of the multirate filter onto a FPGA. The implementation is performed on the same Cyclone 3 development board used and described in the previous chapters. The overall DFG of the multirate filter is seen on figure 17.1.

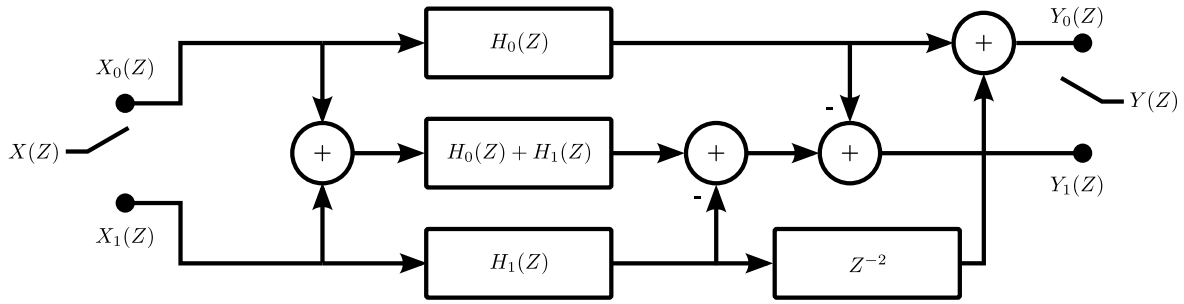

Figure 17.1: DFG of the decimated multirate 1-level architecture. [Mehendale et al., 1996]

The outputs from the filter are Y_0 and Y_1 which are the even and odd samples respectively. These are calculated as:

$$Y_0(z) = C_0 + C_2 \cdot z^{-2} \tag{17.1}$$
$$Y_1(z) = (H_0 + H_1) \cdot (X_0 + X_1) - C_0 - C_2 \tag{17.2}$$

where:

$$C_0 = H_0 \cdot X_0$$
$$C_2 = H_1 \cdot X_1$$
$$C_1 = (H_0 + H_1) \cdot (X_0 + X_1) - C_0 - C_2$$

The even and odd input samples are here denoted by X_0 and X_1 respectively. The decimated filters are seen on figure 17.1 denoted by H_0, H_1 and $H_0 + H_1$. The implementation of these decimated filters will each be shift based filters. The shift based filtering method is described in chapter 16 and an example of a DFG of such a filter is

illustrated in figure 16.5 on page 116. The reason for using the shift based filtering method in the multirate filter is that it is wanted to keep the filters multiplierless as the multipliers are the most power consuming component to implement. The multirate filter block is seen in figure 17.2.

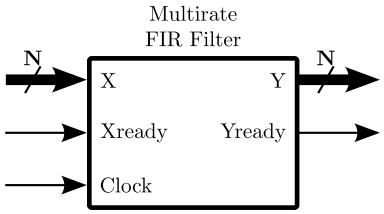

Figure 17.2: The multirate filter block.

As it is seen on figure 17.2, the multirate filters basic outline is the same as the other filters presented earlier. This, with input and output signals and their respective ready-signals. The input clock pin is likewise necessary to make the filter run.

The multirate filter is implemented using a state machine with seven states. The states are divided into two overlapping groups, the first from state 0 to 3 and the second from state 3 to 6. The reason for dividing the states into two groups is that they will calculate the even and odd output values respectively. The first group calculates the H_0 filter and the even output Y_0. The second group calculates both the H_1 filter, the $H_0 + H_1$ filter and the odd output Y_1. The number of clock cycles used to calculate a single output is four, as every group contains four states. This will be shown in the following section when the simulation of the hardware is carried out.

17.2 Simulation

This section will present the simulation of the hardware implementation of the multirate filter. The implementation is simulated using the Altera ModelSim tool, to get a correct picture of the signal flow of the filter. The testbench used in this simulation, is the same as for the previous hardware simulations.

The simulation is executed with a filter with 8 taps, where the coefficients are set to 0.5 or hexadecimal 4000. The input to the filter is 0.5 (4000 hex) to start with and is subtracted 0.25 (2000 hex) every time. The timing diagram for the multirate filter is seen in figure 17.3.

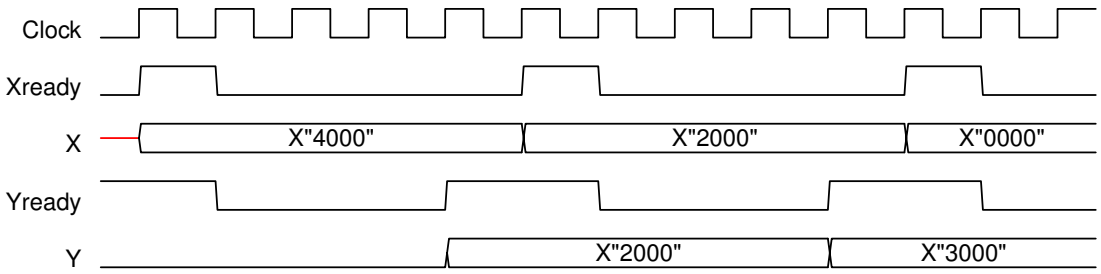

Figure 17.3: The timing diagram of the multirate filter block for the two first periods.

As can be seen from the figure, the filter takes four clock cycles. This can be seen by counting the clock cycles from the *Xready* signal is set high until the *Yready* signal is set high. This comply with the implementation, which has to go through four states before an output is fully calculated. The output from the filter is the half of the sum of the last 8 input values, this is a consequence of the chosen input and coefficient values. From this it is seen that the signal flow of the filter is correct and that the first two inputs are compliant with the inputs.

The timing diagram for 10 filter computation is seen on figure 17.4. This timing diagram is created as to see if the correct output is calculated, when more than 8 inputs has been put into the filter.

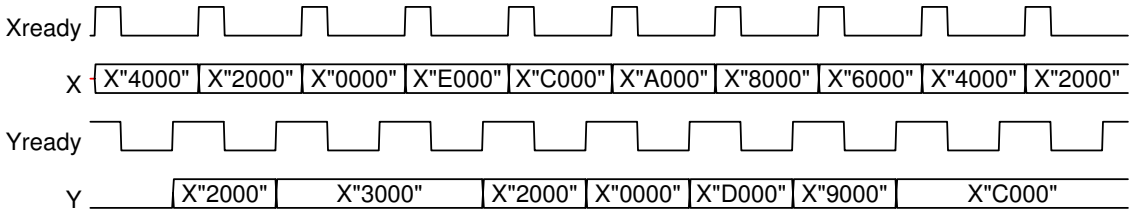

Figure 17.4: The timing diagram of the multirate filter block for 10 periods.

It is seen from the figure, that the output complies with the input of the filter. As for the previous hardware simulations, the output gets static after 8 filter computations. This is due to the fact that the input repeats itself after 8 computations and the filter has 8 taps. From this it is seen that the filter generates the expected output.

The multirate filter has now been implemented and simulated, which makes it ready for evaluation.

17.3 Results

This section contains the results from the evaluation of the multirate filter implementation. The energy dissipation is found by measurements conducted on the Cyclone 3 development board and from estimates calculated by the Altera PowerPlay tool. Both ways of finding the energy dissipation are described in appendix E.

The evaluation set used to define the filter characteristics is described in chapter 3 on page 13. The evaluation set describes low-pass filters with 8 to 120 taps with a step-size of 8 taps. In addition to the evaluation set, the input to the filters is a pseudorandom signal, which is described in appendix D. The pseudorandom signal assures that the signal toggles throughout the filter is random, which is a necessary part of the implementation, as no specific application and thereby input is introduced in this project. The estimated and measured energy consumption for a filter to compute a single output, is seen on figure 17.5.

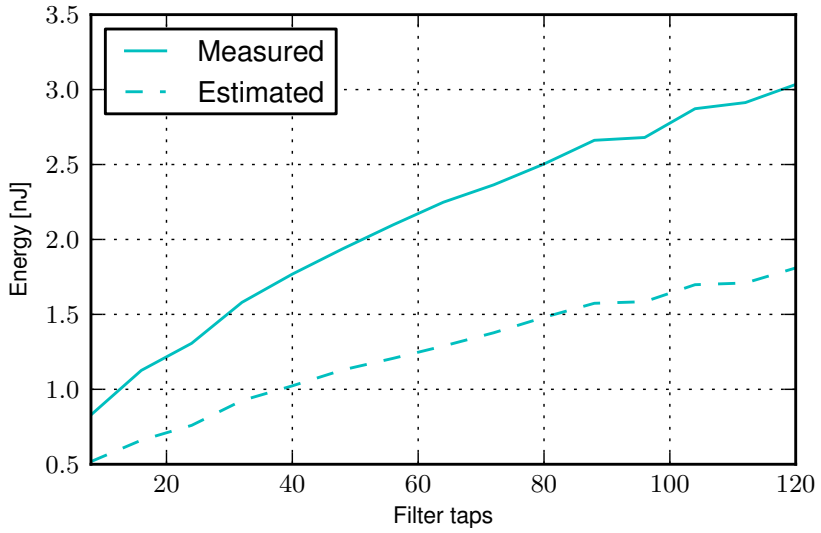

Figure 17.5: Estimated and measured energy of the multirate filter implementation.

There is a clear correlation between the estimated and measures values, but the estimated energy has a smaller slope. This shows that the PowerPlay estimator is too optimistic when calculating the energy estimates. The av-

erage error is 41.2 %, which is much higher than the specified 10 % margin which the manufacture claims. The reason for this large difference is not easy to find, since access to the source code of the PowerPlay tool is not possible.

The amount of LE's used in the implementation is illustrated on figure 17.6.

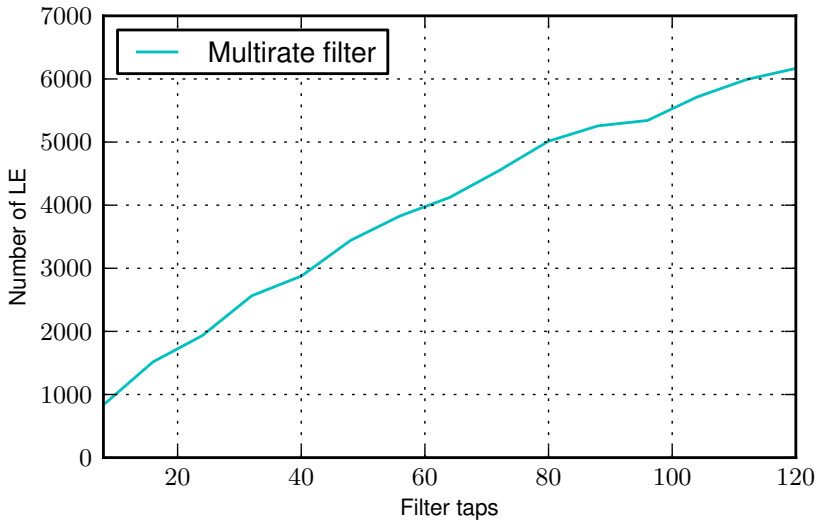

Figure 17.6: Logic elements used in the multirate filter implementation.

The LE consumption is increasing linearly as a function of the filter length. This is due to the increasing size of the delaylines in the filters and as more control logic is needed in the overall implementation. The FPGA contains 24624 LE's, which means that the filter with 120 taps use approximately 25 % of the total LE's possible. This is a relatively large amount of LE's compared to the other implementations investigated in this report. The reason for this relatively large amount of LE's is that a lot of control logic is needed to calculate three filters rather than one.

17.4 Conclusion

This section serve the purpose of summarising and concluding on the multirate filter as a method of designing multiplierless filters.

The method introduced in this chapter is a combination of two earlier described methods, why an analysis of these were not necessary. The chapter therefore starts out by describing the implementation of the multirate shift based filter. The VHDL implementation of the filter is described in terms of the states in the finite state machine used. By simulating the implementation, it is verified that the filter works in the proper way.

The results from the evaluation of the implementation shows that the estimated and measured values describes an increasing trend as a function of increasing filter length. The estimated values are on average 41.2 % lower that the measures, which most likely is because the estimator is too optimistic in its estimations.

18 Evaluation of Multiplierless Filters

This chapter contains the evaluation and comparison of the optimisation methods investigated and implemented in the hardware part of this report. This part of the report has looked into multiplierless filters using distributed arithmetic, shift based and multirate filters. The previous chapters have presented the method, simulation, implementation and evaluation of the methods. The results consist of a comparison of the estimated energy consumption and the actual energy consumption. In this chapter these results are compared with the two reference filters, these being one sequential implementation and one parallel implementation of a direct form FIR filter. The comparison will later be used in order to make a set of guidelines on how to design low-energy digital filters.

As the DA filter is evaluated for lower filter orders this will be compared with the other filters using the same orders. This will evaluate the filters for lower orders and especially evaluate the DA implementation in comparison to the other implementation methods. There will also be an evaluation and comparison of the shift based, multirate and reference filters for filters up to 120 taps, as stated in the evaluation set in chapter 3 on page 13.

In figure 18.1 the measured energy dissipation from the DA, multirate, shift based and reference filters are shown.

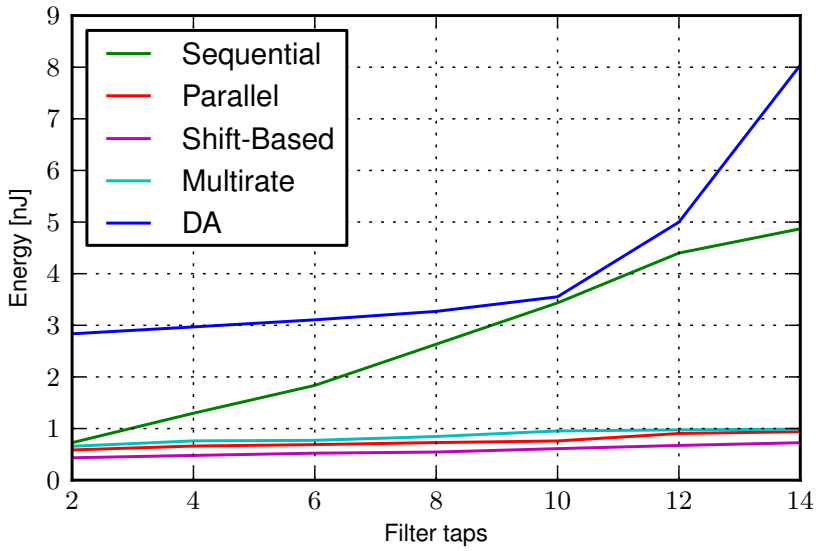

Figure 18.1: Measured energy consumption from DA, shift based, multirate filters and the reference filters.

From this figure it is seen that the energy consumption of the DA filter is significantly higher than for the other methods. The reason for this, is that the on-chip RAM memory consumes a significant amount of energy. The literature uses simple ROM blocks, where the amount of data does not directly affect the energy consumption. This was however not possible in terms of the off-the-shelf Altera Cyclone 3 development board used in this project. The problem could have been solved by connecting auxiliary ROM blocks to the I/O pins on the FPGA, but this has not been investigated due to the time limit of the project.

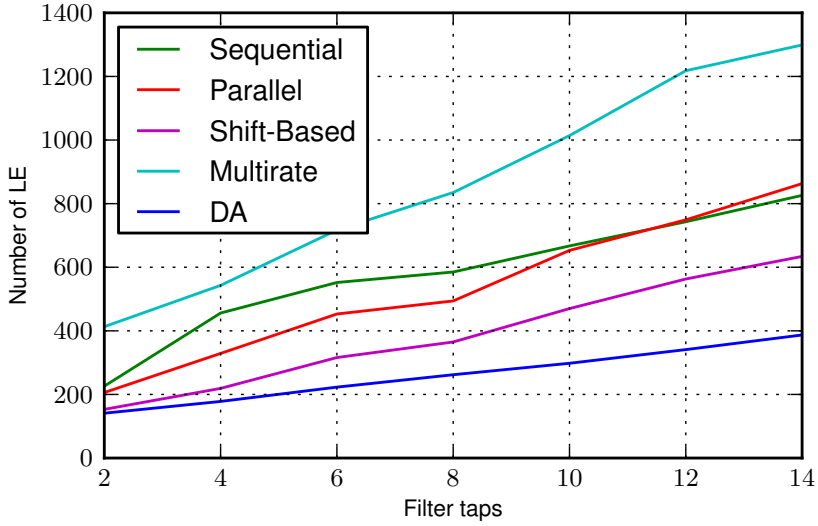

Figure 18.2: Number of LE's used by DA, shift based, multirate filters and the reference filters.

Comparing the LE usage of the different methods implemented with filters from 2 to 14 taps, the LE usage increases for all of the methods, as shown in figure 18.2. The LE usage for the multirate filter is significantly higher than the other filters. This is due to the fact that it contains three filters instead of just one, and does therefore require more LE's. The two reference filters has a higher LE consumption than the other two multiplierless implementations. The DA implementation uses the least amount of LE's, which means that if the implementation had contained external ROM memory, then the DA filter would have had the potential of being a low-energy implementation.

The following tests will compare the shift based and multirate filter implementations with the two reference filters. In these measurements, filters containing 8 to 120 taps are used, as presented in the evaluation set in chapter 3. Figure 18.3 shows the energy dissipation from the two reference filters, the multirate and the shift based filter method, implemented on a Cyclone 3 FPGA. The sequential reference filter dissipates energy to such an extent that it is hard to compare the energy consumption of the other implementations. Therefore; figure 18.4 only shows the shift based filter, the multirate filter and the parallel reference filter.

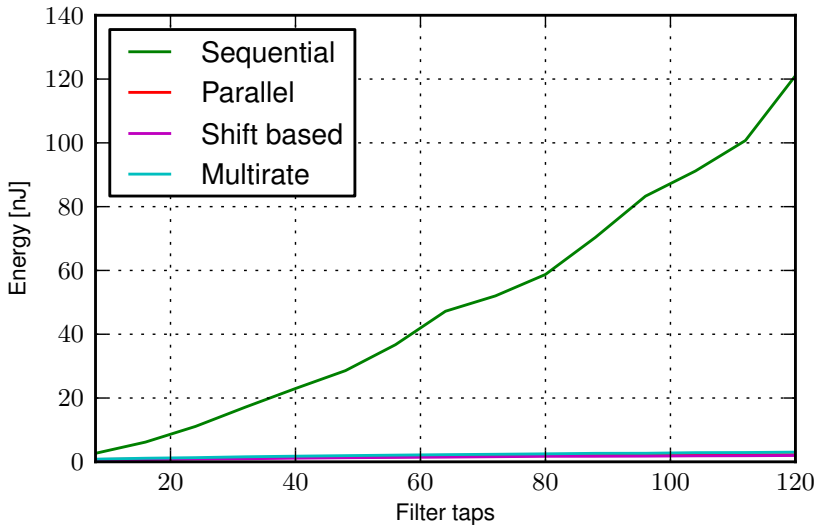

Figure 18.3: Measured energy from shift based, multirate and the two reference filters.

By omitting the sequential reference filter, the rest of the implementations can easier be evaluated. Seen in comparison with the parallel reference filter it is clear to see that the shift based realisation uses the less energy and the multirate uses more energy.

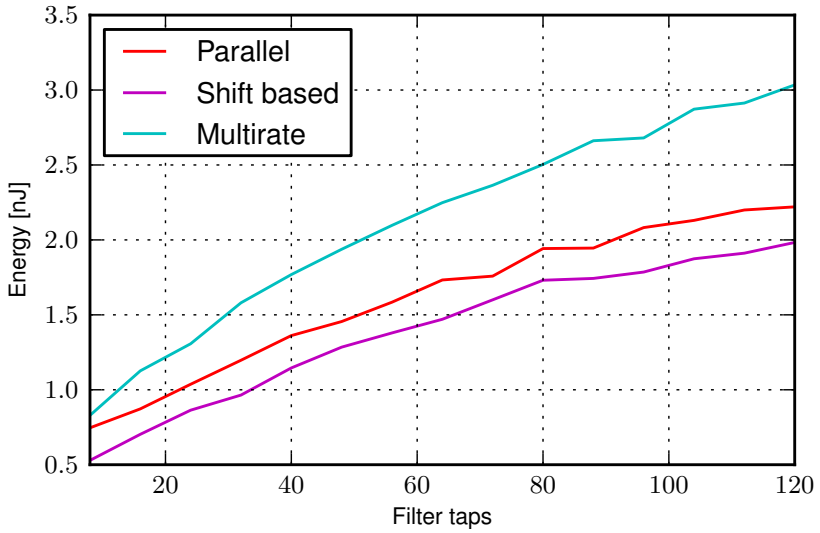

Figure 18.4: Measured energy from shift based, multirate and the parallel reference filters.

The optimisation achieved when replacing the reference filters with the shift based and multirate filters is shown in figure 18.5.

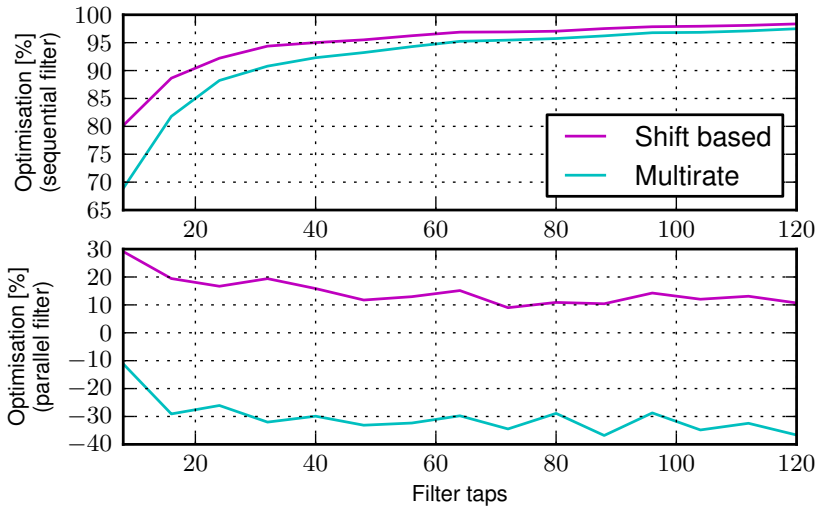

Figure 18.5: Optimisation achieved when employing shift based and multirate filters instead of the two reference filters. The top graph shows the comparison between the two multiplierless filters and the sequential reference filter. The bottom graph shows the comparison with the parallel reference filter.

The possible optimisation when replacing the sequential filter with shift based filter or multirate filter increases as the number of filter taps is increased. This is a direct result of the energy dissipation progression of the sequential reference filter, as this increases with a higher rate than the shift based filter. The average optimisation is found to be 94 % for the shift based filter and 92 % for the multirate filter. The average optimisation is found as the average of the data points of the graphs in figure 18.5. Optimisation above 90 % is seen as an extremely high optimisation percentage and clearly shows the positive effect of introducing energy efficient filters. The high

optimisation percentage is mainly due to the fact that the sequential filter needs more cycles to compute an output as the filter length is increased, and hereby has a higher energy consumption. This while the shift based filter and multirate filter only uses three and four cycles respectively to compute an output no matter what the filter length is. The lowered energy consumption is of course also a result of the effective way of implementing the methods. In addition to the number of cycles needed to complete the computations, the sequential filter is implemented using only one multiplier while the shift based and multirate filters does not use any multipliers. This is an important factor to why the multiplierless filters are more energy efficient implementation methods than the sequential filter using multipliers.

When replacing the parallel reference filter with the shift based and multirate realisations, this results in poorer optimisation than that for the sequential reference filter. For the multirate filter, the optimisation is negative, since the parallel filter has a lower energy consumption. The average optimisation for the shift based filter compared to the parallel reference filter is 15 %, while it for the multirate filter is -30 %. The reason for the high efficiency of the shift based filter, is that the implementation does not use any multipliers and has a much simpler design than the multirate filter. The shift based filter is therefore the most energy efficient implementation compared to the other methods shown above.

Figure 18.6 shows the LE usage for the four filter realisation.

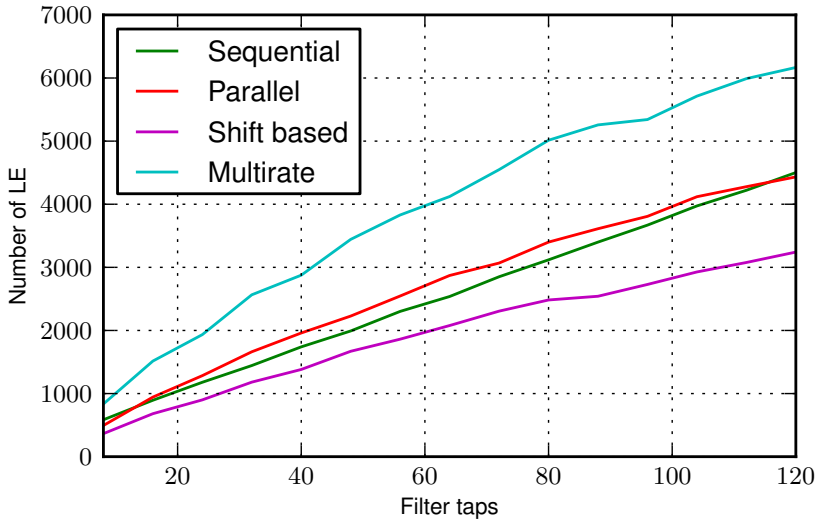

Figure 18.6: LE usage of the shift based filter, the multirate filter and the two reference filters.

The multirate filter uses the most LE's of all the filters, which can be explained with its more complex design. The complex design is introduced due to the fact that the three subfilters uses more control logic than a single filter would do. The LE consumption of the reference filters are very similar, with the parallel version just above the sequential version for most of the filter lengths. This is expected as the parallel filter would require more LE's since it includes several multipliers. It is however not expected that the sequential filter would consume almost the same amount of LE's as the parallel, since it only uses a single multiplier. The reason why it consumes this much, is that the amount of control logic to ensure that the correct coefficient enters the multiplier, consumes a relatively large part of the LE's. The shift based filter consumes the least LE's which is due to it's relatively simple design and that it does not use multipliers.

It is worth mentioning that the results achieved here are depending on several factors. One is the compiler, as this is build on heuristic rules and since it is not possible for the designer to control this. The compiler can for instance perceive and utilise symmetry in the filter implementation due to the coefficients. Another factor is the programmer, as there are several ways to program VHDL code. How the different filter implementations are

coded affects the LE usage and energy dissipation. The results provided in this part can therefore not be perceived as the unambiguous truth. However; for this platform, these implementations and when accepting the effect of varying results from the compiler, the result can been seen as satisfying in the sense that the optimisation methods introduced can be used to develop filters which use less energy than the sequential direct form FIR filter.

18.1 Conclusion

This section concludes on the multiplierless methods as a way of designing low-energy FIR filters. Previously in this chapter the methods are compared and evaluated. This section will draw conclusions based on this evaluation.

The comparison of the multiplierless filtering methods with the two different implementations of the reference filter, had to be split in two parts. This was due to the memory requirements of the DA implementation. This method could not be implemented in the FPGA used in this project for filters with more than 14 taps, so the intentional evaluation set could not be used. The DA filter was instead compared with the other filter implementation for filter lengths from 2 to 14. The results show that the DA implementation does not result in a more energy efficient solution. This is due to the drawbacks of the memory technology chosen for the implementation. The on-chip RAM consumes a significant amount energy, which affects the results. To overcome this problem, the energy consuming on-chip RAM should be replaced by a ROM memory. This will probably introduce a much more energy efficient filter implementation.

Further investigation of the sequential, parallel, multirate and shift based filters were conducted. Results of this was that the shift based implementation was the most energy efficient solution compared to the two reference and multirate filters. This shift based implementation lead to an average optimisation of 94 % compared to the sequential filter and compared to the parallel reference filter the average optimisation achieved was 15 %. The multirate filter gained a similar optimisation compared to the sequential reference filter, as this had an average optimisation of 92 %. In comparison to the parallel reference filter, the multirate filter did not perform as good and was actually less energy efficient. The average difference was -30 % in favour of the parallel reference filter. This difference could potentially be decreased my using a higher level of multirate decimation. The implementation performed here, is a level-1 decimation, and by using a higher level the subfilters will again be multirate filters. This will decrease the computational complexity, which therefore might lower the energy dissipation. A drawback from this will be a more complex structure, which then introduces more LE's.

The results achieved in these tests, shows that the shift based filtering method can be used as a very efficient method for designing low-energy FIR filters. The DA filter, as it is implemented here, is not considered as a suitable way of implementing low-energy filters. It is recommended that the DA filter should be implemented using large ROM memory to gain a more efficient implementation. The multirate filter was a worse implementation than both the parallel reference filter and the shift based filter. This while the implementation also exceeded all the other implementations in LE consumption. The multirate filtering method was introduced as an alternative to DA and shift based filters, and combined both the shift based filtering method along with the multirate structure. This resulted however in a very high consumption of LE's due to the advanced structure of the filter. The multirate filter is more energy efficient than the sequential reference filter and is on the level of both the parallel and shift based implementations. This is seen as a success as this means that the method introduced by the project group can be used to implement low-energy filters. Further studies and development of this method could potentially make it as energy efficient as the other methods presented here. In this implementation only 2nd order polyphase decomposition was applied. Further decomposition could be applied to the filter and thereby may lead to a more energy efficient solution.

The overall conclusion is that the shift based filtering method is, based on the comparison in this project, the most energy efficient way of implementing hardware filters on a FPGA.

Part IV

Conclusion

19 Overall Evaluation

The following chapter will assess all the energy optimisation methods in this project. The chapter will contain a brief summation of the results presented in chapter 6, 12 and 18. In addition, the results will be compared and the pros and cons of each method will be discussed. The results used in this chapter will solely be the measured energy.

The first thing to look at, is the software optimisation methods presented in the second part of the report. Figure 19.1 shows the optimisation possible by using multirate filtering and Hamming distance (HD) optimisation methods.

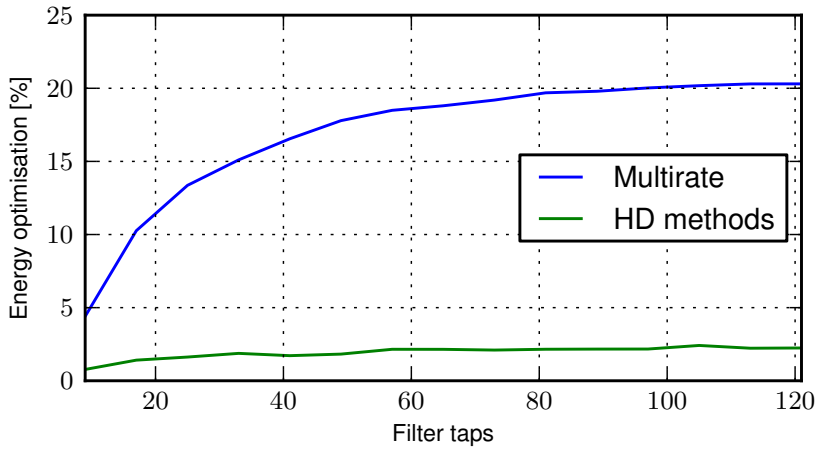

Figure 19.1: Energy optimisation compared to the direct form FIR filter implementation.

Figure 19.1 shows the clear advantage of using a multirate filter as an alternative to the direct form FIR filter. It shows an increasing improvement as the filter length is increased. The optimisation stops at 20 %, which is seen as the best possible optimisation for this implementation. This energy efficient implementation is a result of the multirate decimation technique which lowers the computational complexity. The computational complexity can be converted into the amount of clock cycles which the filter uses to compute an output. The clock cycle difference between the multirate filter and the direct form FIR filter is illustrated in percent on figure 19.2.

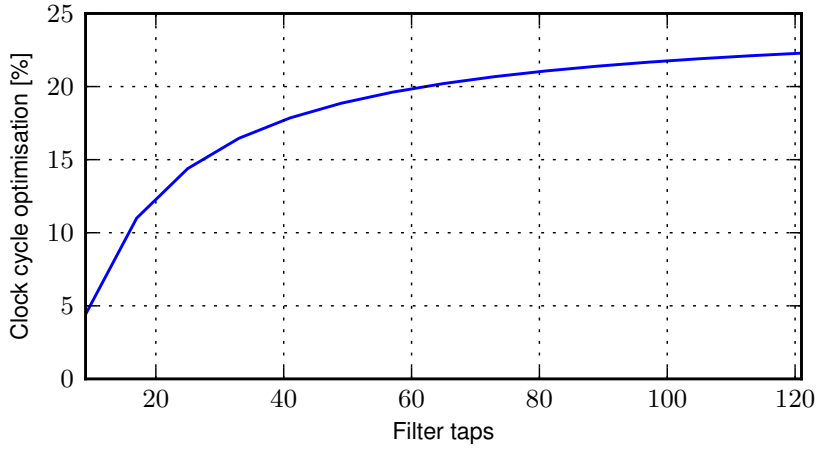

Figure 19.2: Clock cycle difference as a function of the filter length as compared to the direct form FIR filter.

From this figure, it is clear to see, that the two graphs of the multirate energy optimisation and clock cycle difference are correlated. This also means that the if it is possible to lower the amount of clock cycles needed by the filter, then this will transfer directly into a lower energy consumption.

Compared to the 20 % optimisation when using multirate, the HD optimisation methods does not give a similar amount of optimisation. The HD optimisation lead to an average energy optimisation of 2 %. The combined HD optimisation methods were able to optimise the HD of the coefficients by at least 61 % and on average approximately 66 %. This is shown on figure 19.3.

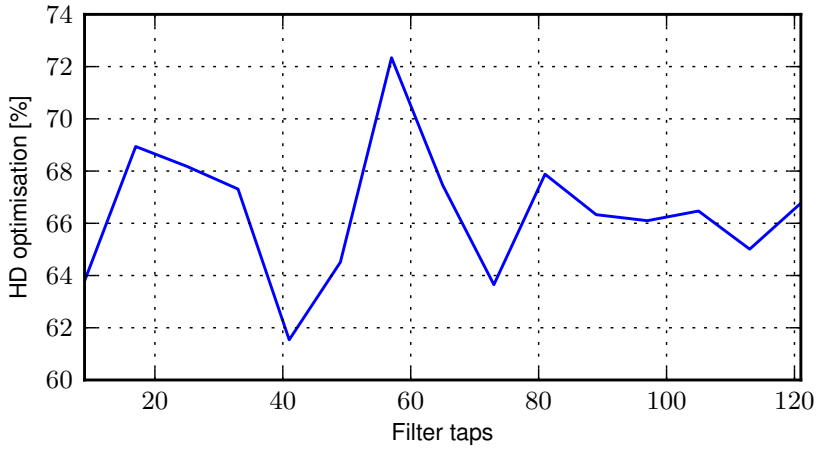

Figure 19.3: HD optimisation as a function of the filter length as compared to the direct form FIR filter.

Even though the HD optimisation is significant, the energy optimisation did not result in an equal significant result. This was however not expected, since changing the switching activity on the coefficient bus, is a very small part of the overall energy consumption of the microcontroller. The 2 % optimisation is a predominant good result, as decreasing the HD of the coefficients is a relatively small improvement in the overall implementation.

The overall assessment of the software optimisation methods is that both methods give a measurable optimisation, which means that both can be used to introduce low-energy optimised filters. When considering the multirate filtering method, this gives a significant optimisation, which would be preferable in many low-energy applications. When considering the HD optimisation methods, the optimisation is only a few percent, but when dealing with low-energy applications, even a few percent can be considered as a significant result. The work load of optimising the coefficients is not time consuming when the optimisation algorithms has been developed. This makes HD optimisation a recommendable step in developing low-energy filters. The two methods can be combined, by employing HD optimisation methods to the coefficients in the three subfilters of the multirate realisation.

The second thing to look at, is the hardware optimisation methods presented in the third part of the report, which solely focused on multiplierless implementations. The reference point consisted of both a sequential and a parallel implementation of a direct form FIR filter. The difference in energy consumption between these were very clear, as the sequential filter consumes significantly more energy than the parallel filter. The difference between the sequential and the parallel implementation is seen on figure 19.4. In some cases the parallel implementation can be seen as low-energy filter.

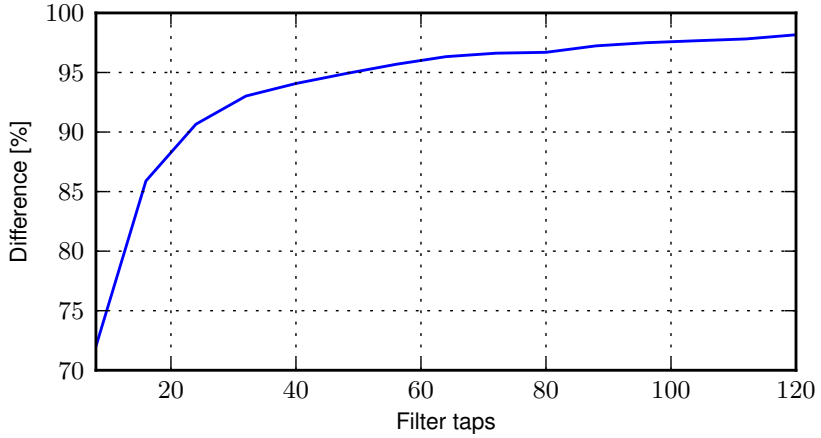

Figure 19.4: Difference between the parallel and sequential filter implementations..

The first hardware optimisation method introduced were the DA filter. The implementation suffered from the restrictions of the Cyclone 3 development board used in this project. The restrictions were introduced in the form of low memory capacity, which meant that the original evaluation set could not be used. The evaluation took instead offset in filter with a smaller amount of taps. The outcome of the evaluation is illustrated on figure 19.5.

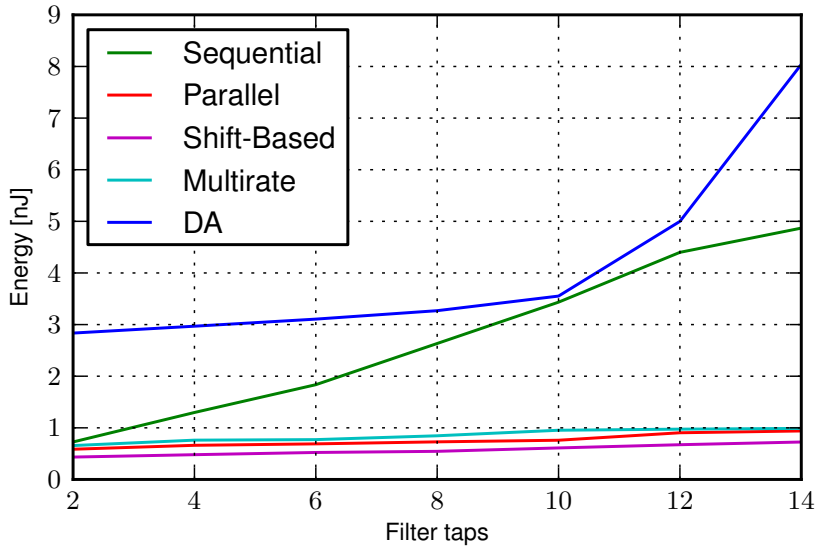

Figure 19.5: Measured energy consumption from DA, shift based, multirate filters and the reference filters.

From the figure it is clear to see that the DA implementation used the most energy of all the implementations. In addition it seems that the energy consumption would rise in an undesirable way, which would restrict the use of this method. The implementation did suffer from the on-chip memory, which at first was too small and secondly affected the energy consumption. This is seen from figure 19.5, as this realisation consumes more energy than all the other. It is expected, that if the implementation had been carried out by using external ROM instead of on-chip RAM, then the result would be very different. The energy consumption of the external ROM should of course then be added to the energy consumption of the FPGA, as to get the overall energy consumption of the complete implementation. For the further evaluation the DA filtering method is not included. This is because it can not be implemented, due to memory requirement, using the evaluation set.

The two other methods introduced in this project has been the shift based and multirate filtering methods. These two methods performed as illustrated in comparison to the parallel reference filter on figure 19.6. The sequential reference filter has not been included in this comparison, as this has significantly higher energy dissipation than all other realisations, which complicates the comparison of the other methods.

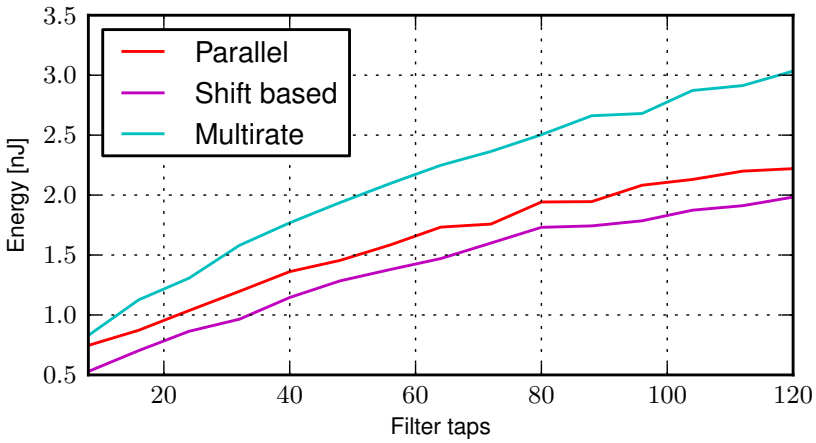

Figure 19.6: Measured energy from shift based, multirate and the parallel reference filters.

The optimisation of the two methods compared to the parallel reference filter is seen on figure 19.7.

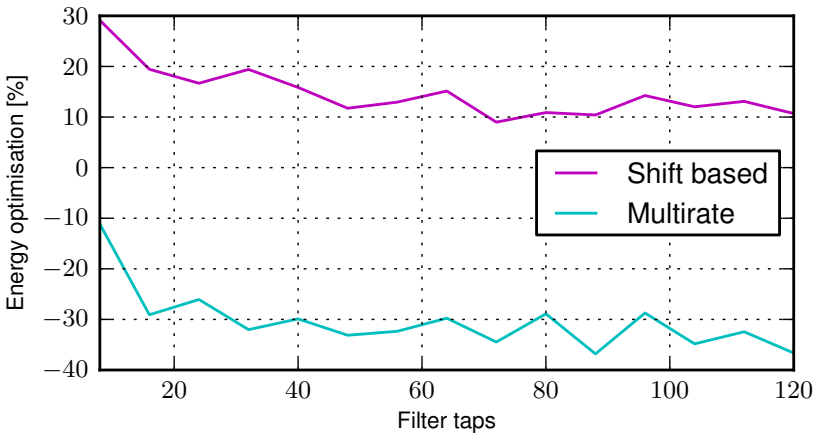

Figure 19.7: Energy optimisation of the shift based and multirate filters compared to the parallel reference filters.

The shift based implementation performs the best and is on average 15 % more energy efficient than the parallel reference filter. This result, in combination with the 20 % optimisation achieved by the software implementation of the multirate filter, lead to the idea of combining these into multirate filter implementation on hardware. This implementation combined the theory of the shift based and multirate filters, into an implementation which should introduce an even more energy efficient solution. However; this was not the case as the multirate filter was on average 30 % less efficient than the parallel reference filter. This was presumably due to the more complex design which the multirate method introduced. This assumption is supported by the comparison of the LE consumption seen on figure 19.8.

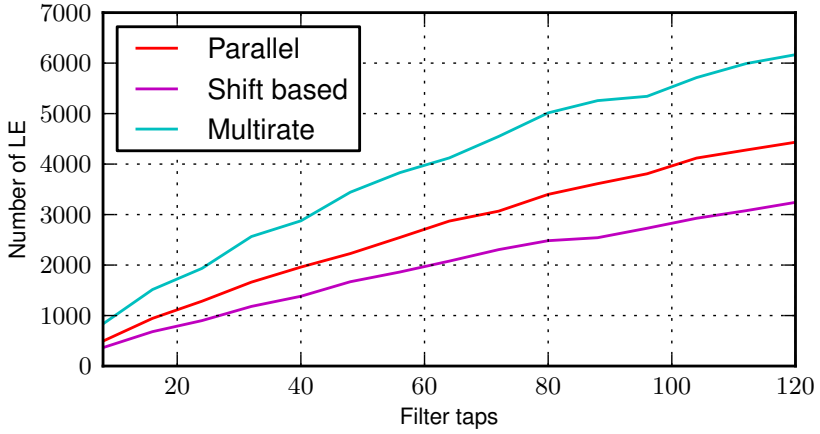

Figure 19.8: LE consumption of the shift based, multirate and parallel filters.

By considering the LE consumption of the different implementations a connection between this and the energy consumption showed on figure 19.6 is evident. This means that if the LE consumption is brought down, then the energy consumption will most likely do the same. If this assumption is transferred to the evaluation of the DA filter, then this would be more energy efficient than all the other filters, as the LE consumption of this is lower than the other filters. This assumption does however not take the energy consumption of the memory into considerations. The LE consumption for all the filter implementations is seen on figure 19.9.

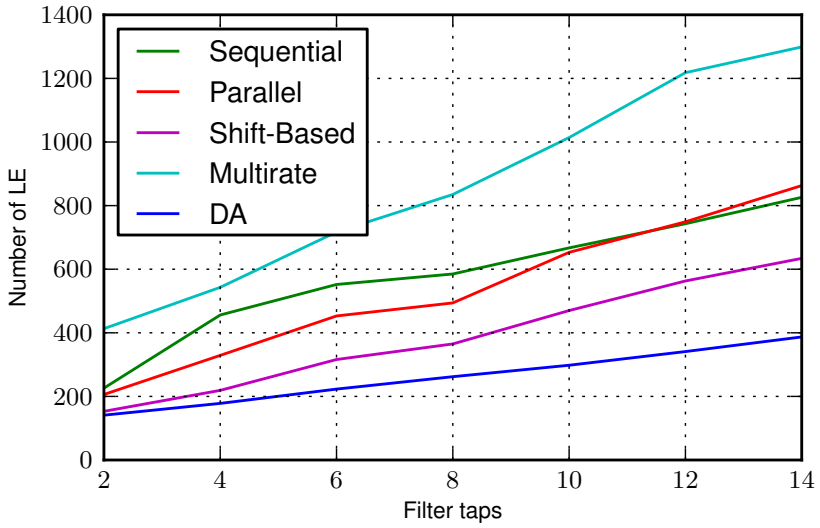

Figure 19.9: Number of LE's used by DA, shift based, multirate filters and reference filters.

As a summary, figure 19.10 shows both the energy and LE consumption in compliance to the sequential filter. The sequential filter is defined as 100 in both energy and LE, and the other methods are normalised in relation to this. This relates directly to percent, as the numbers on the axis corresponds to the percent the various methods are in relation to the sequential filter.

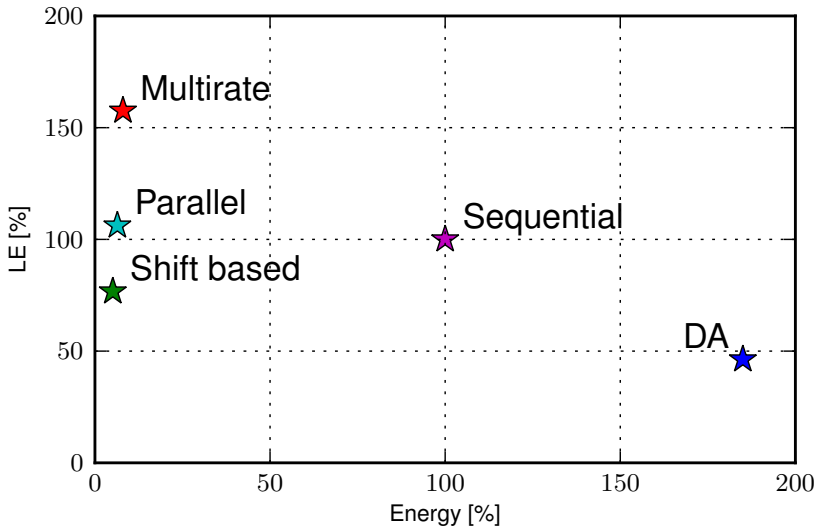

Figure 19.10: Multiplierless filters in terms of energy and LE consumption.

The conclusion on the above overall evaluation is that the software and hardware optimisation methods, showed very different performance as to lower the energy consumption of the filters. With the exception of the DA filter, all of the methods introduced an overall energy improvement. It is however expected that the DA filter would have the possibility to do the same, when the memory has been implemented in a more applicable manner.

Based on the findings from evaluation several methods for low-energy filters a set of guidelines are conducted. These guidelines are intended to guide designers on how to develop low-energy filters. These guidelines follows in the following chapter.

20 Guidelines

The following presents a set of guidelines on how to design low-energy filters. Based on the optimisation methods investigated in this study, these guidelines are intended to direct the designer on how low-energy filters can be designed. The methods presented here are investigated and presented as a part of a larger study, and the full report should be read to get a deeper understanding of the methods.

As digital filters are implemented on different platforms depending on the application, this study has taken both software and hardware implementation into consideration. Therefore; the guidelines will be divided into two main parts, where the first has focus on optimisation methods for software implemented filters and the other on hardware implemented filters. The implementation in this study has turned to use a microcontroller with DSP functionality for the software implementation and a FPGA for the hardware implementation. As a closure to the guidelines a comparison is made, and pros and cons by both the implementation platforms and the methods are discussed.

20.1 Software implemented filters

When optimising software implemented filters for low-energy use, two different approaches has been introduced in this study. First the multirate filtering method, which decimates the filter into subfilters in order to lower the computational complexity. The second approach aims at reducing the switching activity in the coefficient bus. Switching activity is introduced as bits and thereby signals are toggled, and a measure for this is called Hamming distance (HD). Reducing the Hamming distance between the coefficients can therefore be used as a method for designing low-energy filters. In this study three HD optimisation methods are investigated: coefficient scaling, optimisation and ordering.

The outset for these methods, both multirate and HD optimisation methods, is a filter specification. By the use of the Parks-McClellan algorithm the appropriate filter coefficients can be found. The *remez*-function in Python or Matlab can be used to find the filter coefficients by the use of the Parks-McClellan algorithm.

The optimisation methods shown here as a part of a software implementation are not competitive methods, meaning that they can all be combined. This is illustrated in figure 20.1, which is a rewritten graph from [Mehendale et al., 1998]. This graph emphasises the necessary design considerations associated with the different methods.

The graph starts out with a specification of the filter, which is used in the Parks-McClellan method to generate the filter coefficients. The first choice to consider, is whether it is acceptable to use some additional program and data memory. This choice has to be taken, as the multirate filter uses additional space due to the more complex filter structure. If this is acceptable, then a multirate architecture can be used.

Next choice to make is whether an overall change in gain is acceptable. The coefficient scaling method can be applied by defining a maximum gain change, why the developer can control the maximum change in the gain. If this is acceptable, the coefficient scaling method is applied. If the multirate structure has been utilised, changes to the algorithm presented in this project has to be applied. This as the scaling algorithm should consider the a common scaling factor, which minimises the HD of all three filters.

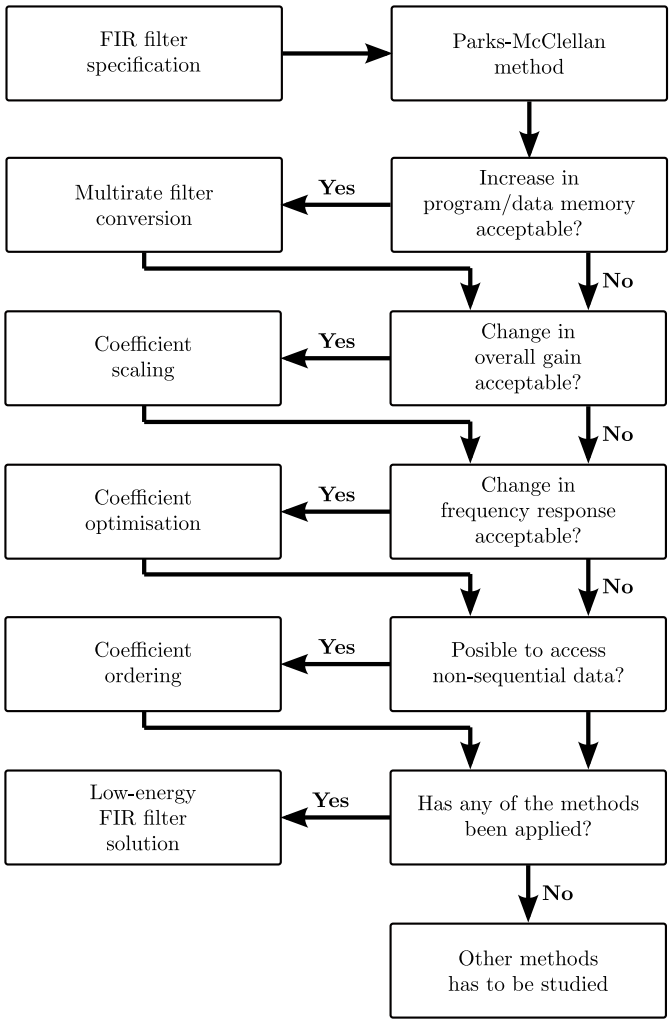

Figure 20.1: Development procedure showing the possible software optimisation methods and the choices to make before employing them.

The next choice is whether a small change in the frequency response is acceptable. The coefficient optimisation method generates a new set of coefficients, optimised in terms of both the filter specifications and HD. Two different algorithms has been developed to utilise coefficient optimisation, a steepest decent algorithm and a genetic algorithm. The difference between these, is that the former takes already known coefficients as input, which the latter does not. If a change in the frequency response of the filter is acceptable, the coefficient optimisation is applied. If the multirate architecture is already applied, the algorithm for coefficient optimisation has to be the steepest decent algorithm, as the multirate architecture consist of three subfilters and the desired characteristics of these filters are not know, which means that the genetic algorithm approach can not be used.

The last choice to make, is whether it is possible or suitable to make a filter which access the coefficients in a non-sequential way. As the coefficient ordering rearranges the coefficients, so the HD is minimised, the filter has to take this reordering into account. If this is possible, then the coefficient ordering can be applied. If the multirate structure has been applied, the coefficient ordering algorithm, must be applied on the subfilters one by one.

If none of the optimisation methods can be applied, then the procedure will of course not result in a low-energy optimised FIR filter. In this case, other optimisation methods has to be found.

It is evident, that the multirate filtering method will introduce the biggest change in energy consumption, as this

results in an optimisation of 20 % compared to the direct form and the HD optimisation methods only achieved 2.24 % at its best. This does however not eliminate the HD coefficient methods, as it does has a positive effect on the energy consumption. The HD optimisation methods has not been tested in connection to the multirate architecture. However; as the HD optimisation methods in that case are employed on three filters with half the length, the possible optimisation of the methods are most likely greater than for employing them on a single filter with the full length.

20.2 Hardware implemented filters

When optimising hardware implemented filters, several approaches can be used. This project introduced optimisation methods with focus on implementing multiplierless alternatives to the generic direct form filter. On the contrary to the methods investigated in the software implementation part, these methods are competing methods. Multiplications are expected to be the most energy consuming operation in digital filters, which is why a designer wants to eliminate them to gain a lower energy consumption.

Three multiplierless methods were investigated. These being shift based filters, distributed arithmetic (DA) and multirate filters. Two reference filters were designed in order have a frame of reference to compare the multiplierless methods with. These reference filters uses multipliers to compute the filter output. One of the filters use a single multiplier to compute the output, and is called the sequential reference filter, while the other reference filter employs as many multipliers as necessary in order to compute all multiplications in parallel. The transposed filter structure was utilised in all the filters, as this is the most common implementation of FIR filters on FPGA [DeBrunner, 2007].

The two reference filters were implemented on the FPGA. Measurements showed that the sequential filter dissipated significantly more energy than the parallel filter. This is due to the fact that the sequential filter uses one multiplier and thereby needs more cycles to compute larger filters. As a consequence of this, the parallel reference filter is also seen as a way of optimising the sequential filter in terms of energy dissipation.

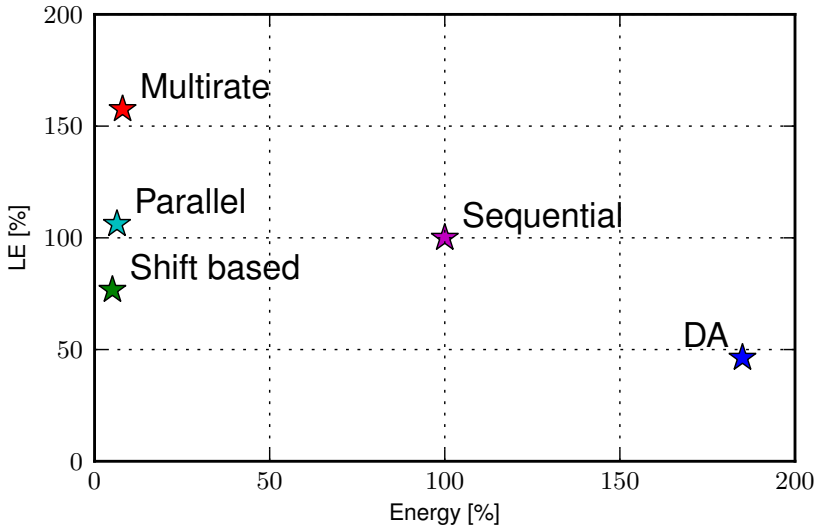

Figure 20.2: Multiplierless filters in terms of energy and LE consumption.

Figure 20.2 shows the different methods in relation to the sequential reference filter. The sequential filter is defined as 100 % in both logic element (LE) consumption and in energy dissipation. The other methods are hereafter plotted in relation to this. With an offset in the sequential reference filter, it is seen that three different methods can be used to make a significant decrease in the energy consumption, seen from figure 20.2. These methods being the

shift based, parallel reference and multirate filters. The energy consumption is almost the same, but the shift based filter does however consume the least amount of energy. The energy consumption of the DA filter can potentially be lowered if the implementation is done with a more energy efficient memory solution.

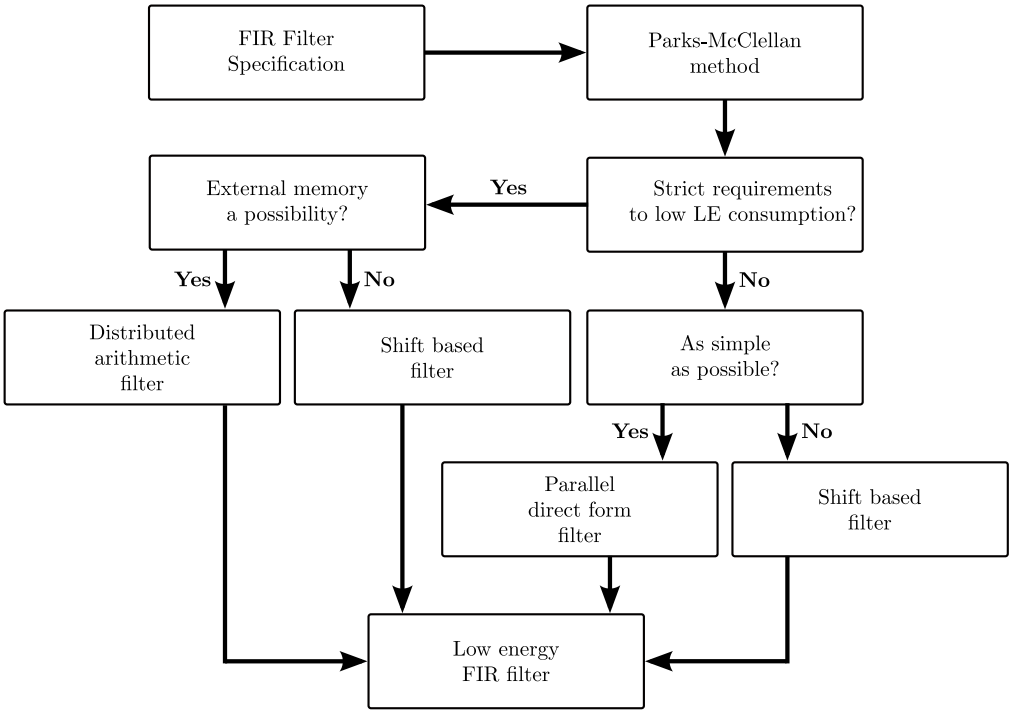

Figure 20.3: Development procedure showing the possible hardware optimisation methods and the choices to make before employing them.

For the purpose of helping the designer to chose the most suitable method, figure 20.3 is conducted. In most cases the designer has a set of filter specifications available, and can based on these and by the use of the Parks-McClellan algorithm design a set of filter coefficients. Descriptions of the application often includes a cost function, which is helpful to chose the most preferable implementation method.

It is common that the cost function include requirements for the allowable area of the implementation. For implementation on a FPGA this is measured in LE's. If the applications has strict requirements to the use of LE's this eliminates some of the methods. With strict LE requirements the designer should investigate the possibility for utilising an external memory, if this is available the DA is suggested as the best solution. If external memory is not available, shift based filtering is the most appropriate.

If the cost function only have moderate requirements to the LE usage, but requires a very simple architecture which can be programmed quickly, this might be due to short time to marked, the parallel reference filter is preferable. This is easy to comprehend, easy to implement, and is a much more energy efficient realisation than the sequential alternative. If this is not the case the shift based filter is a good solution also here. The shift based filter is more comprehensive as the coefficients has to be converted to shifts and logic operators (additions and subtractions). In addition, binary subexpression elimination (BSE) has to be applied as to remove redundant shifts and logic operators.

The figure does not recommend the multirate filter as a solution. In the way that the multirate filter is implemented at the moment, it is not particular LE efficient and it is not as energy efficient as the shift based or the parallel filter. Compared solely to the sequential filter, a significant improvement is achieved but other realisation are found to more efficient. It is however believed that by improving the method further, the performance can become better.

The guidelines presented here are intended for designers who are in need of energy efficient implementations of FIR filters. To give the designer a pointer to whether he should choose a software or hardware programmable platform this depends on several factor. First of all, the choice of implementation platform is not always up to the designer to decide, it is often decided from the application or specifications of the application. However; if the designer has to chose platform, some recommendation on the choice will be made. Based on the measurements done on the microcontroller and FPGA, a major difference in energy consumption is discovered. For the implementation on the microcontroller the energy consumption for the direct form implementation was between 0.25 and 1.75 μJ, while for the implementation of the parallel reference filter on the FPGA the energy consumption was between 0.7 and 2.2 nJ. Based on these measurements the FPGA is a much more energy efficient implementation platform. Whether this is an appropriate implementation platform is to a large extent based on the application, as other signal processing is assumably also needed. It should be noted that many signal processing applications are implemented on DSP's, which is assumed to have a lower energy consumption.

The project does also introduce the techniques of constructing energy models on software programmable implementations and of making power estimations of FPGA implementations. These methods can be used by the designer, as performing actual measurements can be a long and monotonous job. The energy model has proven to give energy estimations which fits the measurements to a great extent. The energy estimations has not proven to be a consistent way of making estimates which fits the measurements and has to be used with caution.

It is of course possible to use several other approaches for introducing energy efficient filtering, this study did however present some of the more popular and fundamental solutions. Further development of the methods presented, will surely give a higher energy optimisation, and the designer is therefore encouraged to do so.

21 Conclusion and Discussion

The following chapter will conclude on the project in its final form and make a discussion on how to improve this. It will also contain a general summation of the project and a short resume of the chapters. The discussion will contain a description of the improvements which could be carried out to improve the results found in the project and suggestions to other approaches to reduce the energy consumption.

The starting point of this project is the article [Mehendale et al., 1998] which, like this project, takes an offset in several methods for designing low-power filters. Only one of the methods were implemented and tested, which laid the foundation of this project, as this was seen as a shortage to make a general comparison between a set of methods. The opinion in the project group is that, to make an evaluation and comparison of energy optimisation methods, actual measurements has to be performed.

This offset has shown that the scope of the project has been very wide and that some strict limitations to the amount of methods has been necessary, as to keep the general literature study to a reasonable size. The literature survey conducted in this project showed that several other approaches are potential candidates for inclusion in the overall comparison, and the project could have been expanded to include more methods.

The project specification stated the problem formulation which is answered throughout the report and in this conclusion. The problem statement asked, how the energy dissipation of FIR filters in software and hardware implementations could be minimised. The answer to this has been investigated by analysing, simulating, implementing and evaluating several methods for this purpose. The main procedure of getting to an evaluation of a method, has been carried out as it is assumed to be by an engineer in a company would precede. This has been very awarding, as this made the process very quick, efficient and practical.

An evaluation set is defined in the very beginning of the project, as to have a common reference point to evaluate the methods upon. As no specific application is considered in this project, the evaluation set had to be as simple and still wide-ranging as to make the final results as general as possible. Tests has been conducted in order to verify that the evaluation set is representative for comparing the optimisation methods with the reference filters.

Concerning the representativeness of the evaluation set, a preliminary test conducted in the software part of the report showed that the Hamming distance (HD) between the coefficients is dependent on the cut-off frequency, yet the evaluation set was not altered. A more thorough analysis would require an expansion of the evaluation set, as to include several cut-off frequencies. In that case the new evaluation set should have been used for the evaluation of all the methods in the project. A comprehensive evaluation set will in all cases be more representative for comparing and evaluation the different low-energy methods. Both linear and no linear filters could also been included in the evaluation set, as to compare the energy optimisation achieved by exploiting the linear phase property. The reason for not considering this change of the evaluation set, is the extensive time it would take to make the measurements.

The methods for the software programmable platform deals generally with decreasing the switching activity or the computational complexity of the filter. This is two very fundamental approaches for reducing the energy consumption. The multirate filter deals with lowering the computational complexity and HD optimisation methods deals with lowering the switching activity. The methods introduced in this part of the project are also found in [Mehendale et al., 1998], why the analysis took an offset from this.

The multirate filtering approach deals with polyphase decomposition, which is used to decrease the computational complexity. From analysis of the computational complexity it was found that that the amount of multiplications

and additions can be lowered by approximately 25 % when a level-1 decimation structure was used, compared to a direct form realisation. It was noted, that the computational complexity can be lowered even further, by using a level-2 decimation, where the subfilters from the first level of decimation are decimated once more. This however results in a more complex filter structure, which challenges the implementation and has therefore not been applied.

Before implementing the multirate filter, an energy model has been constructed based on the theory from [Tiwari et al., 1996]. An energy model can be created by gathering knowledge on how every instruction performs on the microcontroller. The energy computed by the model corresponded very well with the actual measurements, even though the model was simplified by only including two of the three possible measures. A full utilisation of the model could be beneficial if the model was to be applied to more extensive implementations. This will also mean that a full analysis of all the instructions in the instruction set, will have to be carried out. The reason for not doing this in this project, was the size of the implementation, which was relatively small. An idea could be to make it a requirement for the manufactures to make these measurements, as to put together a database of energy models. This will make it easier for the developers and designers to make a quick assessment of a prototype implementation.

The reason for introducing the energy model is to present an alternative to making actual measurements. It can be a rather long and monotonous job, so to spare the designer of this job, it can be beneficial to make a complete energy model, which then will be able to make a qualified guess of the energy dissipation. The energy model was found to be relatively easy to construct and has the potential of saving the designer for a lot of time. It should however be noted, that larger differences was seen between the energy model and the measurements at lower filter orders (< 30 taps), which means that if the designer chose to use the energy model in stead of measurements, some variations has to be expected at lower order filters.

Both the energy model and the measurements showed an energy reduction upto 20 %, compared to the direct form implementation. This is considered to be very satisfying, seen in context to the theoretical reduction in the number of multiplications which is 25 %.

As the computational complexity is decreased, the throughput of the filter is increased. The gain in throughput could be neutralised by lowering the clock frequency of the implementation. As described in section 5.3.2, this decrease in clock frequency means that it is also possible to lower the supply voltage, as the logic delay can be increased. The decrease in supply voltage gives a significant energy saving, as the switching power is a function of the squared supply voltage. It is also noted that the switching power is also dependent on the clock frequency, which in this case also is lowered. This technique of performing frequency and voltage scaling is a very different approach than the explored method in this study and was therefore not applied, though it would introduce a significant energy optimisation.

All in all, the multirate filtering approach gave a significant energy saving and is therefore considered as a very usable method for optimising energy consumption. The result from this method was as expected, due to the connection between the savings in multiplications and energy.

The HD optimisation methods presented in this project are coefficient optimisation, scaling and ordering. The common purpose of these three methods is to lower the switching activity on the data bus. This is done by lowering the HD between the adjacent coefficients, which means that the amount of bits toggling between every coefficient is lowered.

Coefficient optimisation takes an outset in the given filter specification and coefficients, and generates a new set of coefficients which is optimised in the sense of the filter specification and HD. As this method changes the filters frequency response, this has to be allowed by the filter specification. The coefficient optimisation method was performed by two different approaches. The first approach was presented in [Mehendale et al., 1998] and uses the principle of steepest decent, while the second approach was presented by the project group and uses genetic algorithm.

Simulation of the two algorithms showed that that they performed approximately the same. The HD optimisation of a 16th order filter was 66 and 67 % for the steepest descent and genetic algorithm respectively. The reason for introducing the genetic algorithm was to make an alternative to the steepest descent algorithm, and as the performance of the algorithms are approximately the same, this is a success. The genetic algorithm was found to be easier to implement, but this of course depends on the designer.

Coefficient scaling finds a scaling factor which scales the coefficients uniformly for thereby finding a solution which has a lower HD. Contrary to coefficient optimisation, this method does not alter the frequency response of the filter, but it changes the overall gain of the filter. As the gain of the filter is changed, this has to be within the allowable limits of the filter specification. The coefficient scaling method has been utilised as a simple search algorithm developed by the project group. The result from the coefficient scaling was that it was able to generate a 32 % optimisation in the HD of a 16th order filter.

Coefficient ordering alters the coefficient order so the HD between the coefficients is as low as possible. This method does not change the frequency response or gain of the filter, but the algorithm requires that this way of changing the order of the additions is possible on the implementation platform. The method can be seen as a *travelling salesman problem*, which can be solved by several heuristic algorithms. The method is utilised by two different heuristic algorithms: simulated annealing and genetic algorithm. Two algorithms has been developed, as to discover the pros and cons of both approaches and to see if they converge toward the same solution.

The result from the algorithms was very similar and both ended with a HD which was optimised by approximately 63 % for a 16th order filter. A difference between the two algorithms was the execution time, where the genetic algorithm was 5 times slower. This favours the simulated annealing approach, as execution time will become larger and larger for increasing number of taps in the filter.

The three HD optimisation methods were combined as to complement each other and end up with a minimised HD. Coefficient scaling was done first. Further on the coefficient optimisation. The steepest descent approach was used for the coefficient optimisation as this takes an offset in already known coefficients. The simulated annealing approach was chosen for the coefficient ordering. The combined HD optimisation algorithm resulted in an overall HD reduction of 73 % for a 16th order filter. This result is seen as the maximum possible without changing the filter characteristics undesirably.

The evaluation of the HD optimisation methods was utilised in two tests. The first, a preliminary test applied coefficients with a HD of 0 and 100 %, in order to see the minimum and maximum energy dissipation respectively. The second test used the unoptimised and optimised filter coefficients from the evaluation set, in order to see the actual difference in energy dissipation. The result from the tests showed that the optimisation from the maximum HD to the minimum was between 3.5 and 7.3 %. In addition, the optimisation between the unoptimised and optimised coefficients was between 0.8 and 2.2 %. This optimisation in energy dissipation was found by running a simple direct form filter on the microcontroller and simply changing the coefficients transferred on the data bus. This form of optimisation deals with a very small part of the implementation, so a maximum optimisation of 7.3 % is considered as a very good result. When using actual coefficients instead of artificial, the optimisation tops at 2.2 %, which again is considered to be a good performance compared to the small area of optimisation.

All in all the HD optimisation methods did not give similar energy savings as the multirate methods, this was however as expected. The HD optimisation methods can be used in applications, where small optimisations is relevant, but in other cases this method is not considered as an effective method.

Implementation of the HD optimisation method has been considered for the software programmed filters but not for hardware. This is clearly a topic of further investigation and possible reduction of the energy consumption.

The two approaches for optimisation of software implemented filters gave two very different results of optimisation. The multirate method showed an optimisation of up to 20 % while the HD optimisation methods were able to optimise up to 2.2 %. Both results should be seen in their respective extent of optimisation, where the multirate filter changed the whole structure of the filter, contrary to the HD optimisation which introduced small changes to the coefficients, such that a smaller switching activity was achieved. It is clear to see that the multirate approach gives the highest amount of energy saving, but the HD optimisation is also considered as a relatively easy way of optimising the energy consumption if not only a few percent.

The software programmable platform has in this project been a microcontroller with DSP functionalities. This will presumably not be the preferred platform for all signal processing applications and a true DSP would therefore be recommended for further studies. It should though be mentioned, that to use the HD optimisation methods, the platform has to employ a Harvard architecture.

The methods introduced for optimisation of hardware implemented filters, has been focused on multiplierless filters. The multiplier is considered as the most energy consuming part of a filter implementation, why the project presents several methods for eliminating this. The implementation of these methods has been carried out on a FPGA development board, where pins for measuring the energy consumption is easy to access. On beforehand, the effects of using different filter types, such as low-pass, high-pass and band-pass, was explored as to conclude whether the evaluation set was representative. No significant difference was seen, why the evaluation set was considered to be representative.

When considering the reference point for comparison of different multiplierless implementations, a reference filter was constructed. This filter is a transposed direct form filter. The transposed design was introduced as this is the most popular approach [DeBrunner, 2007] and is found to be a convenient way of implementing the filter. It should however be noted that the transposed filter structure is not the most energy efficient solution, as all values in the delayline is updated every time. By using circular buffers in a non-transposed filter, the switching activity when updating the delayline could be significantly decreased.

The term parallelism was introduced, as it is possible for hardware circuits to make several calculations at the same time. Due to this, two different reference filters were constructed. A sequential version, which allocates just a single multiplier, and a parallel version which allocates as many multipliers as there are taps. By doing so, the sequential filter use N+3 clock cycles to calculate the filter output, and the parallel filter use only 4 clock cycles.

Measurements of these two filters, showed that due to the increasing number of clock cycles consumed by the sequential filter, this consumed significantly more energy than the parallel counterpart. This was not expected and the result from this is that the other methods, compared to the sequential reference filter, are much more energy efficient. Analysis of the implementation has not shown any problems with the programming, so the sequential filter simply use much more energy. As a consequence of significant higher energy consumption, the comparison has been focused on the optimisation methods in relation to the parallel reference filter. Despite this the sequential implementation is not seen as faulty.

One of the multiplierless methods was the distributed arithmetic (DA). In this method a multiplierless filter is constructed by saving subresults in a look-up-table. The address in the look-up-table is constructed by forming a bit string from the values in the delayline. The bit string contains as many bits as taps in the filter, which makes the number of words in the memory rise exponentially as the number of taps is increased. The execution time of the DA filter is directly connected to the number of bits used for the values in the delayline.

The implementation and simulation of the DA filter showed that the filter can be calculated in M+3 clock cycles, where M is the number of bits. The exponentially increasing memory requirement caused problems in the implementation, as the on-chip memory was too small to contain the look-up-table for filters with more than 14 taps. To overcome this limitation, the evaluation of the filter implementation was covered by evaluating filters with 2 to 14

taps, increasing the number of taps in steps of two. This alternative evaluation set was also used in addition to the original one, in the evaluation of the other hardware optimisation methods. This, as to make a comparison of these in relation to the DA filter.

Actual measurements were conducted to evaluate the DA filter. The trend showed another problem with the implementation of the look-up-table, as the energy consumption increased exponentially as the number of taps was increased, and were quickly much higher than the sequential reference filter implementation. This is not desirable, and shows that further consideration on the memory solution has to be carried out to use this type of filter in an actual application. It was argued that the on-chip memory was an expensive and energy consuming solution.

Larger low-energy memory could be connected to the FPGA as to lower the energy consumption of the DA filter. This could e.g. be flash memory, ROM or static RAM. In addition to another memory implementation, several methods for decreasing the memory requirement can be applied. If the FIR filter has linear phase property, then the size of the LUT can be reduced by 50 %, by exploiting the symmetry of the coefficients. [Mehendale and Sherlekar, 2001] introduce several methods for memory optimisation which could be interesting to explore.

The overall conclusion of the DA filtering method is that further development of the implementation has to be considered, as to use this in actual applications. It is evident that a solution to the memory problem has to be found in order to make a correct comparison between the implementations. It was argued, that by using external memory blocks, this limitation could be eliminated, but due to the time frame of the project this has not been done.

The second method introduced as a multiplierless alternative to the general direct form filter, is the shift based filter. In the shift based filter the multiplications are replaced by shifts, additions and subtractions. This can be done, as the non-zero bits in a coefficient represent a shift of the input signal determined by the bits respective position in the bit string.

Analysis showed that the amount of non-zero bits directly converts into logic operators. As to lower the amount of logic operators in the implementation, binary subexpression elimination (BSE) was applied. This algorithm found common subexpressions in the coefficients, and eliminated these as to remove redundant calculations. BSE was implemented and is seen as a significant part of the shift based filtering method.

The implementation calculates as many operations as possible in parallel, which result in an implementation that always takes 3 clock cycles to execute. The reason for doing this was that the parallel reference filter performed much more efficient than the sequential filter, and due to this it was more convenient to implement the filter in this way. The evaluation showed that the measured energy consumption was on average 15 % lower than the parallel reference filter. The lower energy consumption was expected as the method employs a very efficient and simple multiplierless alternative to the parallel reference filter.

A third and last method introduced for multiplierless filtering, is the multirate filter. In addition to a multirate structure, the filter also employed the theory of shift based filtering, as to eliminate the multiplier. This way of making a hardware implemented filter is an idea suggested by the project group, and use already introduced principles to end up with a multiplierless filter with a lower computational complexity.

The time to calculate a single output from a single input, was 4 clock cycles independent of the amount of taps. The evaluation showed that the measured energy consumption was on average 30 % higher than the parallel reference filter. It was hoped, that a decrease in computational complexity, in addition to utilisation of a shift based filter, would give a better result than the normal shift based method. This was however not the case, and is assumably due to the more complex structure of the multirate filter. The complexity of the structure is reflected by the consumption of the logic elements, which is the highest of all the hardware implemented filters. A further decimation of the filter could give an even lower computational complexity, but this would again give a more complex filter structure. Whether further decimation will give a more energy efficient solution is therefore hard to say.

It is believed that by further development of this method, the energy consumption could become lower than the parallel reference filters. Whether it is possible to make it better than the shift based solution, is however unlikely since the difference in logic elements is substantial.

The overall evaluation of the hardware implementation methods, showed that the DA filter performed significantly poorer than all the other filter types. This in addition to the sequential reference filter, were the two filters which dissipated the most amount of energy. The multirate, parallel reference and shift based filter, were on approximately the same level of energy dissipation compared to the two other filters. By closer comparison, the multirate filter used on average 30 % more energy than the parallel reference filter. Compared to the parallel reference filter, the shift based filter used 15 % less energy.

Comparison of the logic elements which each of the implementation consumed on the FPGA, the DA and shift based filters used the least amount. The DA filter does in addition use a large amount of memory, which also should be taken into account. The shift based filters were expected to have smallest area on the FPGA, as this method utilised the BSE, to lower the amount of logic operators. In the other end of the scale, the multirate and parallel reference filter used the most amount of logic elements. These were expected to have the largest area, as the multirate filter use three filters instead of one, and also needs more control logic to assure the correct signal flow. The parallel filter, implements the same amount of multipliers as taps and does therefore also consume a significant amount of logic elements.

From this, the most energy efficient solution is the shift based filter, but this does however require more complex VHDL code than the parallel reference filter, which is recommended for quick and efficient implementation of a low-energy filter. This result can be seen as a drawback, as the reference filters was expected to have a higher energy consumption than the proposed methods. As it turns out, the simplicity and parallel structure which the parallel reference filter employs, is a very energy efficient way of implementing filters on a hardware programmable platform.

A general comment, which goes for all the implementations in this project, is that the linear phase property of the generated filters is not exploited. By exploiting the natural symmetry of a linear phase FIR filter, the number of multiplications can be significantly reduced and this will give a measurable difference in the energy consumption. Considering the hardware implemented filters, it is observed that the compiler discovers the symmetry of the coefficients and exploits this to some extent, but further optimisation is possible. Exploiting the linear phase is believed to effect the energy consumption of both the software and hardware implementations, and is therefore a recommended optimisation to follow.

The overall evaluation of the optimisation methods on both software and hardware platforms, has shown that several approaches makes it possible to lower the energy consumption of a filter. In software programmed filters, the multirate method gave the most significant energy saving. This at the expense of a more complex filter structure, which will increase the requirement to program and data memory. The HD optimisation methods, showed that by optimising the filter coefficients, the energy consumption could be decreased by a few percent. This is however a reasonable saving, as this optimisation deals with a very small part of the switching activity of the whole microcontroller. When optimising the coefficients in terms of HD, some margins would have to be introduced in the filter specifications, as to allow the algorithms to change the coefficients. This is presumably not suitable in all applications. In hardware programmed filters, the shift based filter resulted in both significant energy and logic element savings. When looking at the energy consumption of the parallel reference filter and the multirate solution, they both achieved good performance, but at a higher expense of logic elements. The sequential reference filter consumed a significant larger amount of energy, compared to the other solutions. It was noted that the DA filter needed a more suitable implementation, to be comparable with the other methods. When looking at the overall evaluation of the methods, it is evident to see the correlation between the number of clock cycles needed to execute the filter and the energy consumption. This suggests that parallel implementations gives better energy efficiency

than the sequential ones.

The estimates computed by Altera's PowerPlay tool gave very different measures of accuracy in relations to the actual measurements. Several articles and manuals on the PowerPlay tool argued that the estimations fit within a 10 % margin. This was however not the case in the majority of the experiments carried out in this project. Some of the estimations fitted the measurements in a reasonable way, but this is not the overall impression of the results found in the project. The trend has been that the estimations has been optimistic in terms of energy consumption. For the filters implemented, the biggest difference has been approximately 50 %, which is definitely not close to the specified 10 %. It is therefore strongly recommended, not to rely completely on the estimations, but to perform measurements if true measures are to be considered.

As a result of this project a set of guidelines is developed. The guidelines takes the overall evaluation of the project into consideration, as to give suggestions on how to develop low-energy filters. The guidelines are aimed at the designer in the situation of designing a low-energy FIR filter either on a software or hardware programmable platform. The guidelines emphasises the pros and cons of the various methods in order to give the designer the best possible basis of making the right choices for implementation purposes.

A set of guidelines is always usable for a designer and as this study is build on state-of-the-art platforms, they are very representative for nowadays development within the field of low-energy filtering. As the software part of the guidelines is made from assessments on a microcontroller, the results will most likely be different from an actual DSP implementation. This has to be taken into account when using the guidelines.

As part of the guidelines, the choice whether to chose the software or hardware based solutions is also discussed. It is here shown that the FPGA solutions has a much lower energy dissipation, which is expected as the filter is implemented directly into hardware and not in software which has to run on a fixed hardware platform. From this, the FPGA solution is the most energy efficient solution. It is however not assumed that the designer has to take this choice, as it is often the application which defines which platform is used.

The time and complexity of making a software or hardware implementations solely relies on the experience of the designer. For the project group, it has been easier to utilise software implementations as this has been tried before. The learning curve for VHDL programming is found to be somehow steeper than usual software programming, as terms as timing and parallelism has to be considered while defining an architecture.

Among the suggestions for future work, the memory solution for DA filter is considered to be the most important one, as this will complete the studies performed in this project. The DA filter implementation does not work as a low-energy method due to this and should therefore be the first to see improvements.

All in all, the problem statement has been answered, as evaluation of several methods has shown that the energy consumption can be decreased in comparison to a general reference point. The project has covered optimisation on both computational complexity, switching activity and general filter structure. These are considered as the most fundamental and important approaches to energy efficient designs. On the more academic side, heuristic search algorithms and energy models has served as an alternative to the approaches presented in the literature. The academic approach complemented the straight forward approach of exploring the optimisation methods in terms of what an engineer would do in an actual application, which has been the procedure in this project.

Bibliography

Agilent (2002). *AC Voltage Measurement Errors in Digital Multimeters*. AN 1389-3.

Altera (2010). *FPGA Power Management and Modeling Techniques*. WP-01044-2.0.

Altera (2011). *Logic Elements and Logic Array Blocks in the Cyclone III Device Family*. CIII51002-2.3.

Blum, C. and Roli, A. (2003). Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Survey, Vol. 35, Issue 3*, page 268-308.

Chandrakasan, A. P. and Brodersen, R. W. (1995). Minimising power consumption in digital CMOS circuits. *Proceedings of the IEEE, Vol. 83, Issue 4*, page 498-523.

DeBrunner, L. S. (2007). Reducing complexity of FIR filter implementations for low power applications. *Conference Record of the Forty-First Asilomar Conference on Signals, Systems and Computers, 2007. ACSSC 2007.*, page 1407-1411.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Inc. ISBN: 0-201-15767-5.

Haupt, R. L. and Haupt, S. E. (2004). *Practical Genetic Algorithms*. John Wiley & Sons, Inc., 2. edition. ISBN: 0-471-45565-2.

Koomey, J. G., Berard, S., Sanchez, M. and Wong, H. (2011). Implications of historical trends in the electrical efficiency of computing. *IEEE Annals of the History of Computing, Vol. 33, Issue 3*, page 46-54.

Mahesh, R. and Vinod, A. P. (2008). A new common subexpression elimination algorithm for realizing low-complexity higher order digital filters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 27, Issue 2*, page 217-229.

Maskell, D. L. (2007). Design of efficient multiplierless FIR filters. *Circuits, Devices & Systems, IET, Vol. 1, Issue 2*, page 175-180.

Mehendale, M. and Sherlekar, S. D. (2001). *VLSI Synthesis of DSP Kernels - Algorithmic and Architectural Transformations*. Kluwer Academic Publishers. ISBN: 0-7923-7421-5.

Mehendale, M., Sherlekar, S. D. and Venkatesh, G. (1995). Coefficient optimization for low power realization of FIR filters. *1995. IEEE Signal Processing Society Workshop on VLSI Signal Processing, VIII*, page 352-361.

Mehendale, M., Sherlekar, S. D. and Venkatesh, G. (1996). Low-power realization of FIR filters using multirate architectures. *VLSI Design 9th Conference Proceedings*, page 370-375.

Mehendale, M., Sherlekar, S. D. and Venkatesh, G. (1998). Low-power realization of FIR filters on programmable DSP's. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 6, Issue 4*, page 546-553.

Meher, P. K. (2010). New approach to look-up-table design and memory-based realization of FIR digital filter. *IEEE Transactions on Circuits and Systems I: Regular Papers, Vol. 57, Issue 3*, page 592-603.

Narendra, S., De, V., Borkar, S., Antoniadis, D. A. and Chandrakasan, A. P. (2004). Full-chip subthreshold leakage power prediction and reduction techniques for sub-0.18-μm CMOS. *IEEE Journal of Solid-State Circuits, Vol. 39, Issue 3*, page 501-510.

Oliver, J. P. and Boemo, E. (2011). Power estimations vs. power measurements in cyclone III devices. *2011 VII Southern Conference on Programmable Logic (SPL)*, page 87-90.

Oppenheim, A. V. and Schafer, R. W. (1999). *Discrete-time Signal Processing*. Prentice-Hall, Inc., 2. edition. ISBN: 0-13-083443-2.

Srinivas, M. and Patnaik, L. M. (1994). Genetic algorithms: A survey. *Computer, Vol. 27, Issue 6*, page 17-26.

Tiwari, V., Malik, S. and Wolfe, A. (1996). Instruction level power analysis and optimization of software. *Ninth International Conference on VLSI Design*, page 326-328.

Vaidyanathan, P. P. (1993). *Multirate Systems and Filter Banks*. Prentice-Hall, Inc. ISBN: 0-13-605718-7.

White, S. A. (1989). Applications of distributed arithmetic to digital signal processing: A tutorial review. *IEEE ASSP Magazine, Vol. 6, Issue 3*, page 4-19.

Wong, K. D. (2011). On RMS voltage measurement. Website: `http://www.kerrywong.com/2011/08/22/on-rms-voltage-measurements/`.

Part V

Appendix

Combinatorial Optimisation

In many optimisation problems, there is a goal of finding the best combination and configuration of variables in order to solve a problem. In the case where the solution to this problem is encoded with discrete valued variables, the problem is called a combinatorial optimisation problem (COP). In such problems the optimal solution is searched for in a finite set. The COP can be defined as:

$$P = (S, f) \tag{A.1}$$

where S is all possible solutions and f is the objective function. The problem can be defined by several factors e.g.:

- Set of variables $x = \{x_1, x_2, \cdots, x_n\}$
- Constraints among the variables
- Objective function to be minimised

The task is to find a solution $s* \in S$ using the minimum objective function value $f(s*) \leq f(s)$ where $s \in S$. $s*$ is the globally optimal of (S, f).

One typical COP is the travelling salesman problem. The principle of the travelling salesman problem is illustrated by an example on figure A.1. In the general problem a salesman has to visit N cities. He should visit each city exactly once travelling the shortest possible way. In this problem there are $N!$ possible routes, so finding the optimal solution is challenging and time consuming when N becomes large. Adding one extra city, increases the complexity of the problem significant.

Such problems are said to be NP-complete and no algorithm exist to find a solution of this within polynomial time. This class of problems are non-efficiently computable, however it is possible to find a near-optimal solution or a solution within the constraints of the problem. Approximating algorithms are used to solve NP-complete problems. These algorithms has no guarantee for finding the optimal solution, but will find good solution in bounded time.

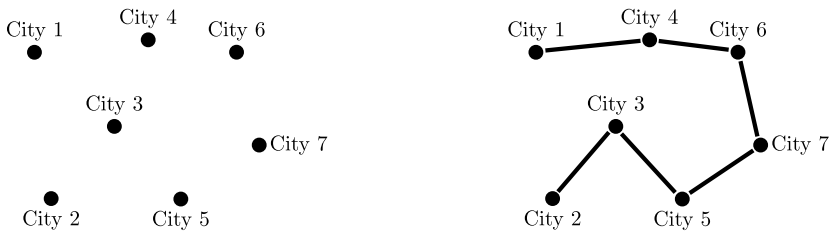

Figure A.1: An example of the travelling salesman problem. To the left all the cities are seen and to the right the shortest route between the cities.

Designing FIR filters can be considered as COP with the filter coefficients as the variables and an objective function consisting of the desired filter characteristics. As one coefficient is changed, this changes the whole filter characteristic, and most likely other coefficients has to be adjusted, while trying to minimise the objective function. The objective function can include e.g. requirements to the pass-band ripple, stop-band attenuation, phase property etc.. The objective function is minimised in the sense of the error between the filter characteristics of the found solution and the requirements.

The genetic algorithms can be used to solve COP and is one of the more popular approaches. The next section describes the genetic algorithms.

A.1 Genetic algorithm

This section describes the basic principle of genetic algorithms (GA). It includes a general presentation of how GA works and the methods that can be used within the GA. This is a general description which introduces the algorithm and terminology used.

GA is an iterative optimisation technique based on principles of evolution and "survival of the fittest". By the use of natural search and selection processes the foundation of GA is made. In nature, the fittest individuals will survive the less fit as the ability to adapt to the environment is essential for surviving. These are also the main principles in GA. As the fittest individuals survive, so will the fittest genes and they will be passed on to the next generation. The next generation is created when two parents reproduces and generates offspring.

GA works from several encoded sets of parameters which fills up a population. This means that the search algorithm looks at a whole bunch of points in the solution space, which is one of the advantages of GA. GA then finds the best points from the population by using an objective function. This means that it is not necessary to calculate e.g. gradients, which in some applications would result in far from optimal solutions [Goldberg, 1989]. The chosen points are then mixed together or altered in some way, to create new points, which then all together gather up and creates the new population.

Since GA is closely related to both natural genetics and computer science, the terminology tend to get mixed up in the literature. The following will describe the link between the artificial and the natural world.

In the natural world we deal with a number of *genes*, which contain some values called *alleles*. These genes are formed in a structure called a *chromosome*. In the artificial world, the chromosome is called a *string*, which then contains *features* which has different *values*. The *position* of the features in the string is contrary the *locus* of the gene in the chromosome. The chromosomes gather up and creates a *genotype* also called a *structure* in the artificial world. The *phenotype* of the genotype is the physical properties of the genotype, where the structure transforms into a *parameter set* or *point* in the solution space. It is these physical properties that GA evaluates over to get to the so called fitness value, which is the measure of how good the structures or genotypes are. The fitness value is calculated by the objective function. The terminology of GA is summarised in table A.1. The relationship between the structure, parameter set and the fitness value is illustrated on figure A.2.

Natural genetics	**Genetic algorithms**
Chromosome	String
Gene	Feature
Allele	Feature value
Locus	String position
Genotype	Structure
Phenotype	Parameter set

Table A.1: Comparison of terminology of natural genetics and genetic algorithms

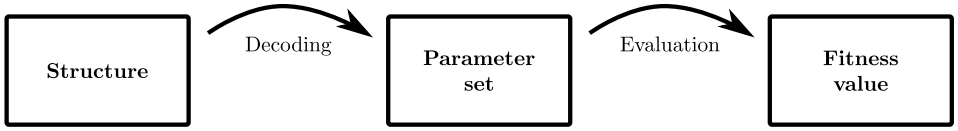

Figure A.2: The relations between structure, parameter set and the fitness value.

The work with GA in this project will be concentrated around the terminology created from the artificial world, which is summarised in figure A.3.

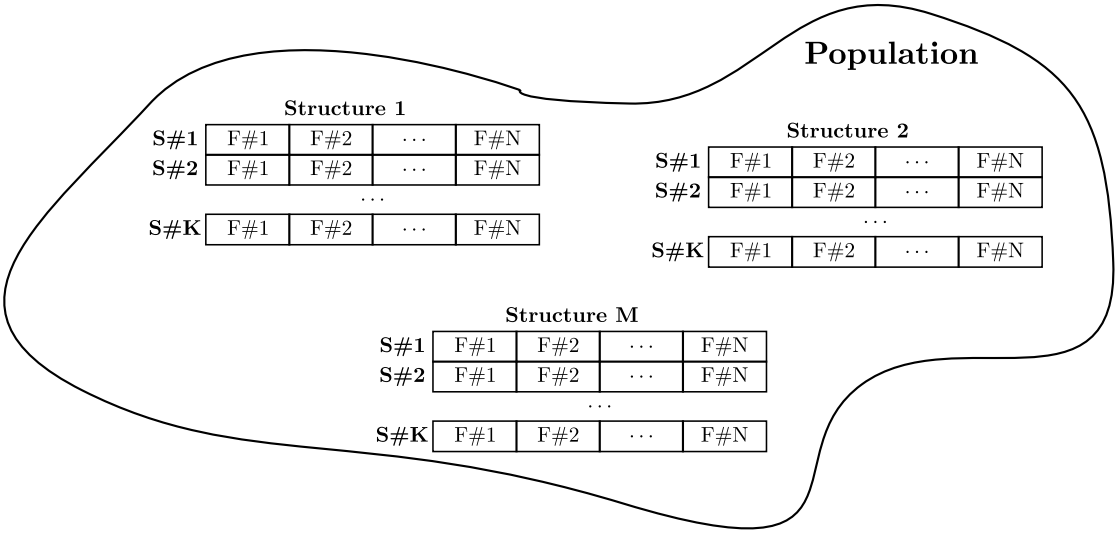

Figure A.3: The structure of the population and what it is composed of.

In terms of filter optimisation, the population contain filters (structures) which is considered for selection and reproduction. Each filter is made from strings (e.g. *S#1*), which contains the variables of the filter which is optimised upon. This could e.g. be the number of coefficients, gain, number of bits used etc.. For the example with a string containing the coefficients, each coefficient will be represented by a single feature (e.g. *F#1*).

The following will address the general flow of GA.

A.1.1 Flow of the algorithm

Figure A.4 illustrates the flow of GA which is called the GA cycle.

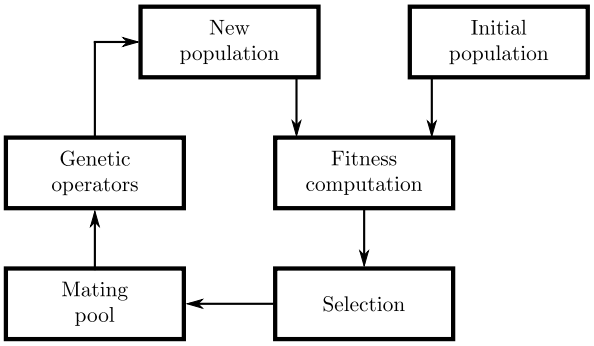

Figure A.4: The GA cycle.

As seen in figure A.4, the algorithm starts out with an initial population, which consists of structures composed of strings containing a number of features (variables). The initial population can be selected fully randomly or it can be based on some initial guesses. A well considered initial population, will in most cases lead to a quicker convergence towards a near-optimal solution, but is many times hard to determine.

Fitness computations

From the initial population the fitter and weaker structures must be found. This is done by a fitness computation of the parameter set from the structures. How to evaluate the parameter set, depend on the objective function that is to be optimised. From application to application this function may generate many different values, but the fitness value should always be high for a good performing structures and low for a bad performing structures. By the use of this function the fitness of the parameters will be the value used for further selection of parents for the next population.

Selection

The next block in the chain of the GA cycle is the *Selection*, this is where the fittest structures are selected for reproduction. The fitter structures will have more offspring than the weaker ones which are thereby more likely to perish. There are several ways of doing the selection. A possible method is the *proportionate selection method*, this uses both the structures fitness values and the average fitness value. Using this method a structure with fitness f_i is allocated f_i/F offspring, where F is the average fitness value of the population. A structure with fitness value above average are allocated to have more than one offspring, while a structure with fitness below average will have less than one offspring. Using this method most structures are assigned a fractional number of offspring, which is not possible. To overcome this, the *roulette wheel method* can be used to determine which structures to choose. This method tends to be the most popular method and can easily be illustrated, as in figure A.5. In the roulette wheel the structures in the population are normalised and placed in a pie chart so the size of the sector they span is determined by the fitness of their respective parameter sets.

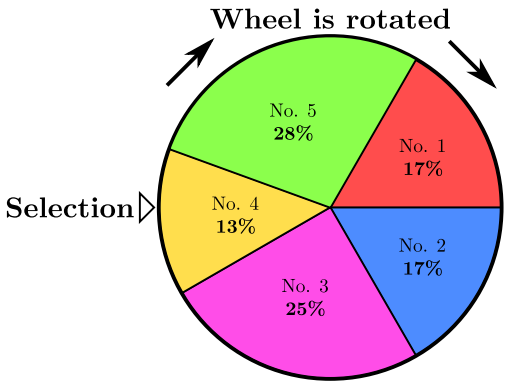

Figure A.5: Roulette wheel used for selection in GA.

As the wheel is spun it is most likely to end up at the fittest structure, but a structure with very low fitness might also be selected in rare cases. This method gives a fair selection, though there are no guaranty for selecting the most fit structures for the mating pool.

Another selection method is the *elite selection*. By using this method it is guaranteed that the most fit structures will be selected for mating. This method is often used in cooperation with the roulette wheel selection, where the best structures will be put into the mating pool, if they are not already selected by the roulette wheel method.

The structures selected for the mating pool are the ones that will reproduce. Introducing a selection rate such that only a part of the population is selected for mating can be done in order to prevent bad performance to evolve. However; letting too few structures reproduce, limits the offspring of the next population [Haupt and Haupt, 2004]. By introducing a selection rate, it is decided to discard some structures. There is a high chance, that the poorest structures are discarded, and this makes room for possible better new offspring created later by the genetic operators.

Genetic operators

Genetic operators are applied to the structures in the mating pool. Several genetic operators can be used in GA and the following will introduce the most popular and important ones.

The *crossover* operator is probably the most important operator, as it states the very core of genetics. In crossover, pairs from the mating pool are selected and their strings are interchanged. This can be done by single-point crossover or with several crossover points. Single-point crossover is illustrated in figure A.6. This figure shows two chosen strings and how their offspring will look like. The bits after the crossover point are interchanged in the two offspring strings, while the bits prior to this point remains the same. This method can be expanded to include several crossover points. In that case the strings are split into several parts and interchanged. Where in the string the crossover point is sat is chosen randomly, both for single-point crossover and for n-point crossover.

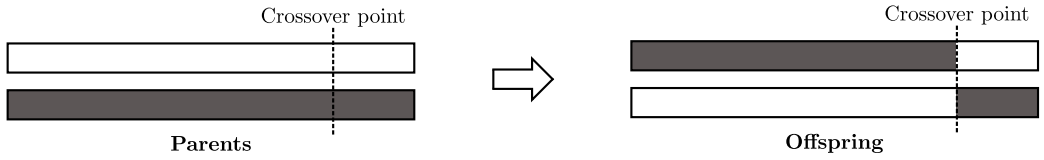

Figure A.6: The genetic operator crossover.

The structures will be paired to produce two offspring. This will be repeated until there are enough offspring to replace the discarded structures. How to select which structures to be paired can be chosen in several ways. The first method for selecting mates is by pairing from top to bottom. The structures are first listed according to their fitness, the two best suited structures will be paired together, and the two second best structures will be paired together and so on.

Another approach for pairing is random pairing. The simplest version of this method randomly pairs two structures in the population. Variations of this random pairing exists. One variation of this method is called random weighting. In this method the pairing uses probabilities for selecting, just like the roulette wheel selection does. The structures are paired randomly but the structures with higher fitness has higher probability to be chosen.

The crossover operation should not necessarily be done for all structures, so a crossover rate (P_C) determining how often crossover should be executed, can be chosen[Srinivas and Patnaik, 1994]. This rate determines the probability of how often crossover should be done. The crossover operator does not bring any new information to the population, as it uses only the available information.

Another popular genetic operator is *mutation*. Also this can have a probability rate (P_M) in the same way as the crossover. Mutation can be done in combination with crossover or fully independent of any other genetic operator. Mutation implies that one or several bits in the strings is altered. The bits are flipped independently and the bits to alter are chosen randomly. An illustration of the principle of mutation is shown in figure A.7. This method introduces new information to the population, contrary to crossover. Mutation is a known phenomenon from natural genetics and introduces an unexpected change in the string. The mutation rate is in the order of one mutation per thousand bit from the total strings in the population [Goldberg, 1989].

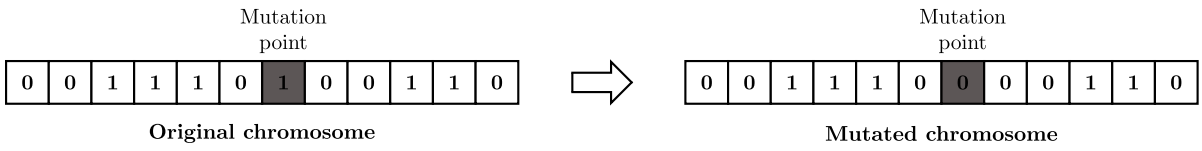

Figure A.7: The genetic operator mutation.

A third genetic operator is simply copying. This operator copies one structure from the old population and directly into the new. This is done in some cases and is generally used in connection with elitist selection, hereby copying the best structures directly into the new population. By treating the fittest structures as every other structures and apply crossover and mutation on them, unnecessary errors might be generated in already good structures. Letting them proceed to the next population by simply copying them maintains their overall good fitness.

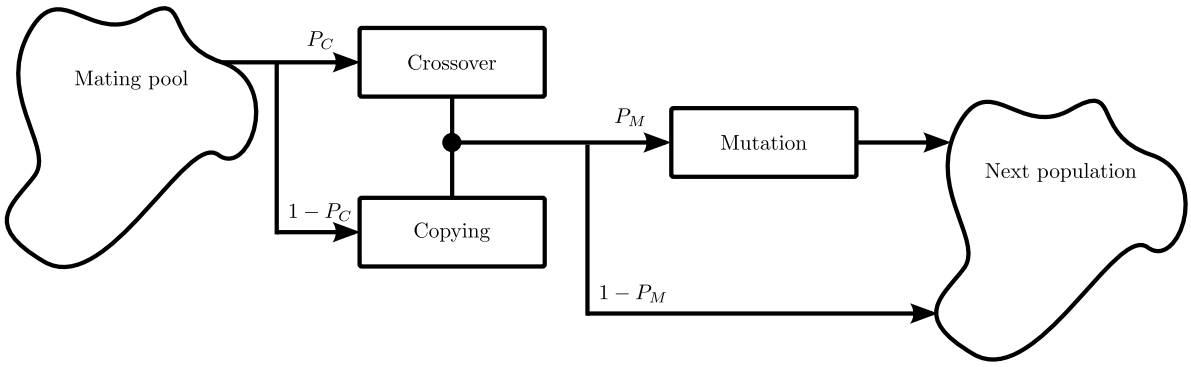

Figure A.8: The flow-diagram over genetic operators with their respective probabilities.

Figure A.8 show how the genetic operators can be linked together and how the respective probabilities are used. GA can be modelled in an infinite number of ways, as different size of population, crossover rate, mutation rate etc. could be used. The variables will change from application to application. The various algorithm variables, can be optimised as to find the set of variables which optimises the convergence rate. This can again be seen as a COP, which will take even longer time to find a near-optimal solution.

This appendix has presented the term combinatorial optimisation problem (COP) and the travelling salesman problem as an example of a COP. Further; was the genetic algorithm (GA) presented as a method for solving COP's. A general description of the GA was presented. This will be used as a basis in other chapters of the report.

B Parks-McClellan Method

FIR filters can be designed in many ways. The following appendix will introduce and describe some of the basic features of the Parks-McClellan method, which is also known as the Remez exchange method. The Parks-McClellan method is a popular method for designing linear phase FIR filters. The method is build on the more general Remez method, in the way that it restricts the Remez method to linear phase FIR filters.

The appendix will be a short summation of the description of the Parks-McClellan method from [Oppenheim and Schafer, 1999].

The usual method used to design FIR filters are the window method, where the ideal impulse response of the filter is truncated to a specific filter order M. This method is the best mean-square approximation of the filter, since it minimises the error:

$$\varepsilon^2 = \frac{1}{2\pi}\int_{-\pi}^{\pi} |H_d(e^{j\omega}) - H(e^{j\omega})|^2 d\omega \tag{B.1}$$

between the desired filter H_d and the designed filter H.

This method is good in the sense that it is simple to employ, but it has some limitations. These limitations could be such as discontinuities at the ends of the window and no way to optimise the error in e.g. stop and pass-bands. This is why methods as e.g. Parks-McClellan can be used instead, which uses a minimax optimisation strategy. Minimax optimisation is the minimisation of the maximum errors.

The wish is to design a linear phase FIR filter, so let us start out by considering a zero-phase filter:

$$A_e(e^{j\omega}) = \sum_{n=-L}^{L} h_e[n]e^{-j\omega n} \tag{B.2}$$

where:

$$h_e[n] = h_e[-n] \tag{B.3}$$

due to the zero-phase property.
If the filter has $L = M/2$ as an integer (even order), the filter can be rewritten as:

$$A_e(e^{j\omega}) = h_e[0] + \sum_{n=1}^{L} 2 \cdot h_e[n] cos(\omega n) \tag{B.4}$$

which can be delayed by $L = M/2$ samples, to get the causal frequency response:

$$H(e^{j\omega}) = A_e(e^{j\omega})e^{-j\omega M/2} \tag{B.5}$$

or the impulse response:

$$h[n] = h_e[n - M/2] = h[M - n] \tag{B.6}$$

The Parks-McClellan method is an approximation method as it makes a polynomial approximation of the cosine function in eq. (B.4). The polynomial is more specific a Chebyshev polynomial, which is defined as:

$$T_n(x) = cos(n \cdot cos^{-1}(x)) \tag{B.7}$$

and the cosine function is rewritten as a sum of powers of cosines as:

$$cos(\omega n) = T_n(cos(\omega)) \tag{B.8}$$

where T_n is a nth order Chebyshev polynomial. Eq. (B.4) can now be expressed as:

$$A_e(e^{j\omega}) = \sum_{k=0}^{L} a_k(cos(\omega)^k \tag{B.9}$$

where a_k is the values of the impulse response. If $x = cos(\omega)$ then:

$$A_e(e^{j\omega}) = P(x)|_{x=cos(\omega)} \tag{B.10}$$

where P(x) now is the Lth order polynomial:

$$P(x) = \sum_{k=0}^{L} a_k x^k \tag{B.11}$$

Let us now take a look at the minimax error optimisation problem:

$$\underset{h_e[n]:0\leq n\leq L}{min} \left(\underset{\omega\in F}{max} \; |E(\omega)| \right) \tag{B.12}$$

where F is in a closed subset of $0 \leq \omega \leq \pi$, which for a low-pass filter will be $0 \leq \omega \leq \omega_p$ or $\omega_s \leq \omega \leq \pi$. These intervals can also be seen on figure B.1.

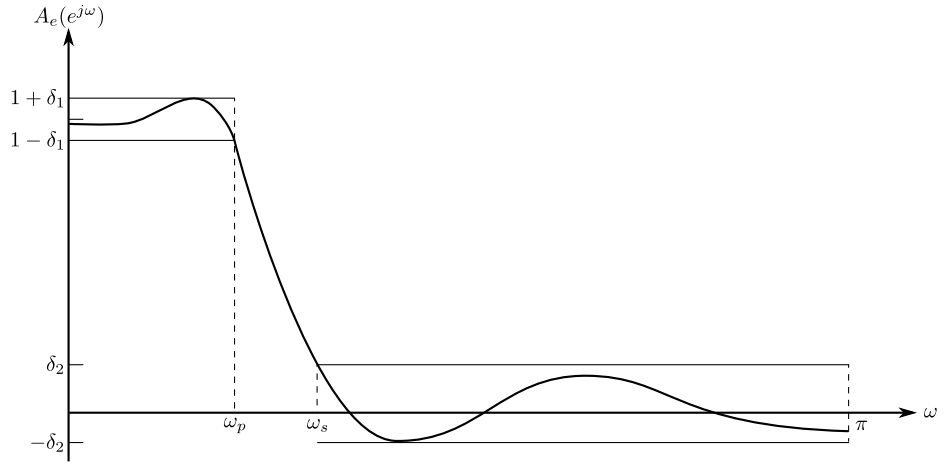

Figure B.1: Frequency response meeting the specifications from the Parks-McClellan method [Oppenheim and Schafer, 1999].

The error function is defined as:

$$E(\omega) = W(\omega)[H_d(e^{j\omega}) - A_e(e^{j\omega})] \tag{B.13}$$

which employs a weighting function, W, a desired frequency response, H_d, and the approximating function, A_e. All of these functions are not defined in the transition band, $\omega_p < \omega < \omega_s$, which is the interval where it does not matter how the frequency response looks. The desired frequency response can be designed to find the best solution:

$$H_d(e^{j\omega}) = \begin{matrix} 1, & 0 \leq \omega \leq \omega_p \\ 0, & \omega_s \leq \omega \leq \pi \end{matrix} \tag{B.14}$$

which is the ideal low-pass filter.
The weighting function is inserted to weight the error in different bands:

$$W(omega) = \begin{matrix} \frac{1}{K}, & 0 \leq \omega \leq \omega_p \\ 1, & \omega_s \leq \omega \leq \pi \end{matrix} \tag{B.15}$$

where $K = \delta_1/\delta_2$ which is seen on figure B.1. The δ_1 and δ_2 is allowable pass-band and stop-band deviations respectively. The Parks-McClellan method minimises the approximating function by minimising the allowable deviations, with the filter order, pass- and stop-band edges fixed.

In a simplified form the Parks-McClellan algorithm runs as the following*:

1. Guess the positions of the extrema as being evenly spaced in the pass and stop-band.
2. Perform polynomial interpolation and re-estimate positions of the local extrema.
3. Move extrema to new positions and iterate until the extrema stop shifting.

This is a rather simple way of describing how the algorithm works, as further analysis is considered outside the scope of this project. The algorithm moves the extrema of the frequency response, until the curve is just touching the allowable deviations.

This appendix gives a presentation of the Parks-McClellan algorithm. It describes the theory of it and a simplified version of how the algorithm runs. For the use in this project the project group has turned to use the *remez* function in Python as to compute filters with linear phase property.

*From the Wikipedia site on Parks-McClellan algorithm: `http://en.wikipedia.org/wiki/Parks-McClellan_filter_design_algorithm`

Energy Measurement on dsPIC

The following chapter contains a description on how to replicate the measurements performed on the dsPIC30F3012, which is used in this project. The chapter will describe the general set-up and procedure of the measurement, and elaborate on which errors and uncertainties this set-up gives.

C.1 Objective

The objective of this measurement will be to measure the energy that the dsPIC30F3012 consumes when different filters is programmed on it. The energy is found by measuring the voltage drop over a shunt resistor with a known resistance. The energy can thereby be calculated as:

$$E = V_{MCU} \cdot \frac{V_R}{R} \cdot T \tag{C.1}$$

where V_{MCU} and V_R are the voltage drop over the microcontroller and the shunt resistor respectively. T is the time it takes for the filter to calculate an output.

C.2 Equipment

The equipment used in this measurement is used to supply the dsPIC30F3012 with power and measure the voltage drop over the shunt resistor.

Name:	**Description:**	**AAU number:**
Microchip MPLAB ICD3	In-circuit debugger and programmer	-
Agilent E3630A	Triple Output DC Power Supply	56781-00
Agilent 34401A	6 1/2 Digit Multimeter	52858-00
Agilent 54825A	Infiniium Oscilloscope	52856

Table C.1: Equipment used for measuring energy consumption.

In addition to the equipment in table C.1, the group has made a measurement board for the dsPIC30F3012, which is shown on figure C.1.

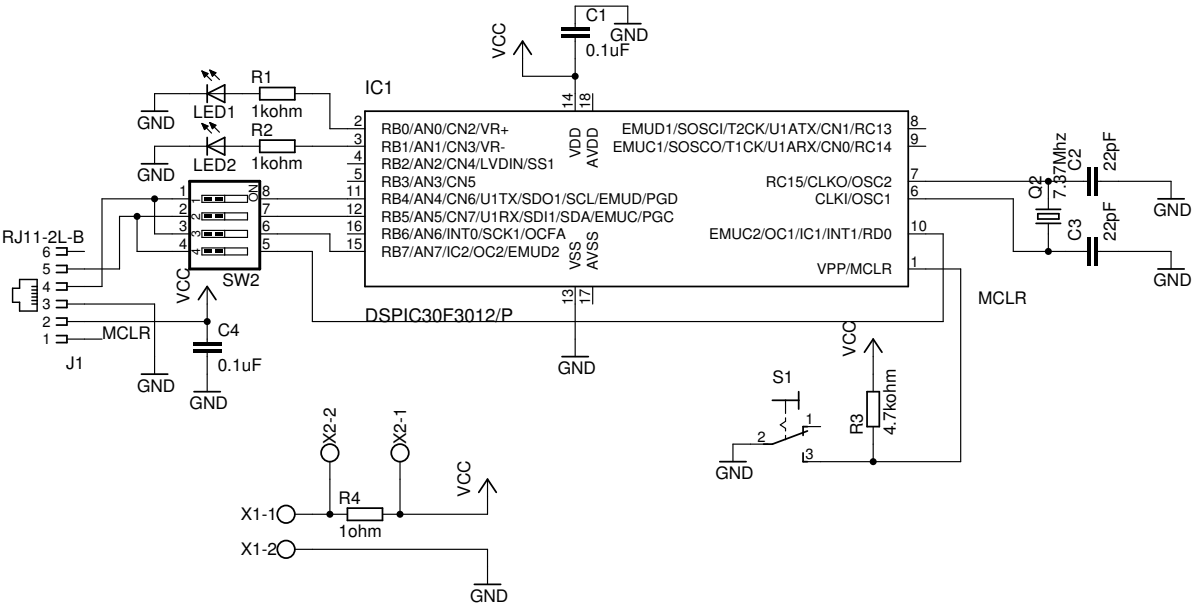

Figure C.1: The schematic set-up for the dsPIC30F3012 board.

Important connectors on figure C.1 are the RJ11-2L-B connector called J1, which is placed on the board in order to program it via the MPLAB ICD3 in-circuit debugger and programmer. The terminals X1 (1 and 2) and X2 (1 and 2) are used to supply the board with power and to measure the voltage drop respectively. It should be noted that resistor R4 is the shunt resistor of size 1 Ω with tolerance $\pm$ 1 %.

C.3 Set-up

The set-up of the measurement is as described in the following:

- The power supply is set to an output voltage of 5 V.
- Connect the power supply: VCC to terminal X1-1 and ground to terminal X1-2.
- The measurement instrument is connected to the board with measurement probes: the high probe to terminal X2-2 and ground probe to terminal X2-1.
- The ICD3 debugger and programmer is connected to the RJ11 connector J1 on the board and to a computer using an USB cable.

C.4 Procedure

The procedure of measuring the energy consumption of the dsPIC30F3012 is as the following:

1. Measure and note the resistance of the resistor R4 on the board, using a mΩ meter.
2. Power up the board by turning on the power supply.
3. Load the program, which is going to be programmed to the microcontroller, into MPLAB X.
4. Program the dsPIC30F3012 by pushing the *Program* button in e.g. MPLAB X development application for the Microchip PIC's.
5. Now measure and note the voltage drop over the resistor R4, with the measurement instrument.
6. Calculate the energy consumption by using eq. (C.1).
7. If more measurements are required, jump to step 3.

C.5 Preliminary test

A preliminary test is conducted in order to compare the measurements done by an oscilloscope and a multimeter. Table C.3 shows the measurements done for four filters with different length. The oscilloscope is set to calculate the DC RMS, which includes both the DC and AC component. The total RMS value is found with the multimeter, by measuring both AC and DC measures, and the total RMS voltage is found by calculating:

$$V_{RMS} = \sqrt{V_{DC\,RMS}^2 + V_{AC\,RMS}^2} \tag{C.2}$$

Table C.2 shows the AC and DC measures from the multimeter.

Filter length	AC RMS	DC RMS
8	0.003 mV	45.871 mV
40	0.021 mV	45.899 mV
80	0.009 mV	45.966 mV
120	0.005 mV	45.930 mV

Table C.2: AC and DC multimeter measurements.

The AC component measured with the multimeter is significantly lower than the DC component. In the computation of the RMS voltage the AC component does not affect the result to a great extent.

When using the multimeter several uncertainties has to be taken into account. As stated by [Agilent, 2002] and [Wong, 2011], the AC reading can vary significantly when the signal is not sinusoidal and symmetric. This give reasons to believe that the measurements of the AC component is not accurate.

As seen in table C.3 the measures from the oscilloscope is slightly larger than for the multimeter. However; they evolve in the same manner and the difference is very consistent.

Filter length	Voltage drop over shunt resistor		Difference
	Oscilloscope	Multimeter	
8	47.57 mV	45.871 mV	3.57 %
40	47.59 mV	45.899 mV	3.55 %
80	47.83 mV	45.966 mV	3.90 %
120	47.63 mV	45.930 mV	3.57 %

Table C.3: Comparison of oscilloscope and multimeter measurements.

Based on these measurements both the oscilloscope and the multimeter are considered appropriate choices of measuring devices. They do not give the exact same measuring results but due to the relatively small variation in the difference both are seen as possible measuring devices. However; when comparing results, the same measuring device has to be used in order to get an accurate comparison. If the multimeter is used as measuring device one can restrict oneself to only measure the DC component, as the AC component is neglectable compared to the DC component.

It is chosen to use the multimeter to find the energy dissipated by the microcontroller.

C.6 Source of errors and uncertainties

A source of error and uncertainty is that the multimeter used to measure the voltage drop measures the RMS value. This average value may not be sufficient in all cases, and will therefore generate errors. In most of the measurements conducted in this project, the average value will however be sufficient and will therefore not be the source of error.

When doing RMS measurements an important factor is noise. Noise can be introduced in this test set-up at several points. The cables used in the test set-up can be subject of radiated noise from e.g. other electrical devices. The cables are however shielded which will minimize the amount of radiated noise. The board is equally a subject of radiated noise, especially as this is not shielded and the pins and solderings are therefore exposed.

The temperature of the surroundings will also affect the measurements, as the energy dissipation will vary as a function of the temperature of the components at use.

Pseudorandom Number Generator

D

This appendix will describe how the input signal to the filters is generated, as this is an important factor when measuring energy dissipation. For measuring the energy dissipation the input to the filters needs to be as random as possible, to remove inactive signals which would affect the energy dissipation. The input signal is generated by a pseudorandom number generator (PRNG).

A pseudorandom number generator generates numbers that appears to be random, but in reality is not, as they will get periodic after a long period of time. The important thing here is that the generated sequence gets periodic after sufficiently long time. The reason to use pseudorandom numbers is that they are easier to generate than truly random number.

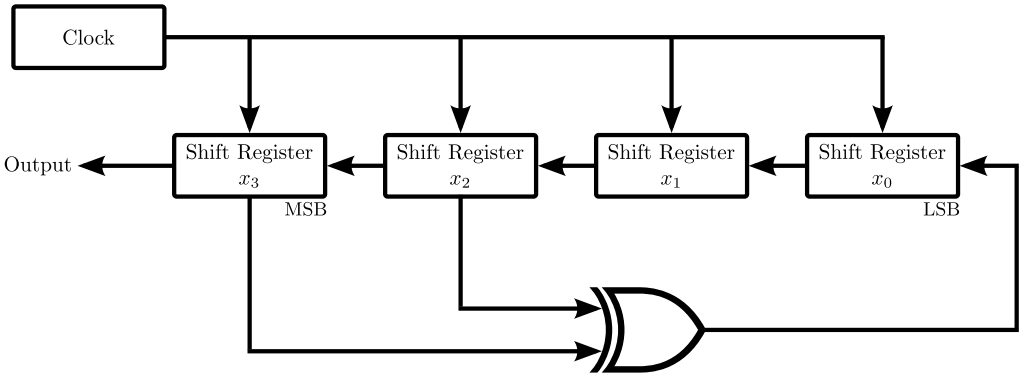

Figure D.1: DFG of linear feedback shift register.

Pseudorandom numbers can be designed in many ways, but here the method called linear feedback shift register (LFSR) will be presented, as this is one of the most common ways of implementing PRNG. In LFSR the output from the register is a linear combination of its previous state. Usually a XOR-gate (exclusive-or) is used as the linear function to produce the next state*. A simple LFSR is shown in figure D.1. Here it is illustrated how the LFSR uses the previous state in order to create the next. For more complex LFSR several XOR-gates can be used in various combinations. Figure D.2 illustrates two clock cycles in the LFSR.

*Wikipedia on Linear Feedback Shift Register, 27. March 2012, http://en.wikipedia.org/wiki/Linear_feedback_shift_register

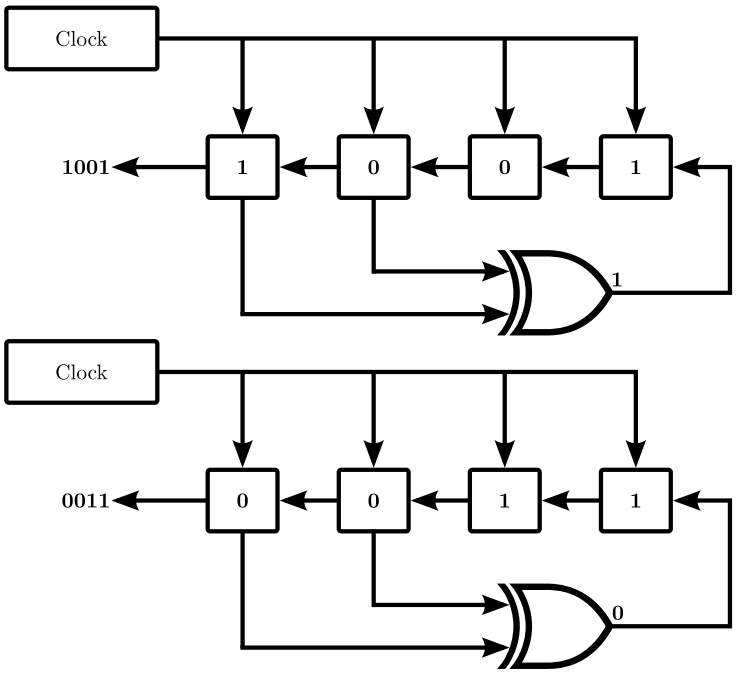

Figure D.2: Example of linear feedback shift register.

All LFSRs need a starting point, this is called a *seed*. The seed can be any number except zero, as this will not generate a new value other than zeros. Therefore; all other seed then zero can be used. For some applications the seed can be determined by real-time processes for instance time or date. When using time as a seed, the pseudorandom generator will only create the same sequence at very rare instances. This might for some applications be meaningful, but for the purposes in this project, a static predefined number would be sufficient. In that case the user also have to accept that the pseudorandom number sequence will repeat in a predefined pattern, and that the same sequence of number will be created every time starting the generator. The number of bits used in the implementation defines the periodicity of the pattern. The number of clock cycles before the sequence repeats it self, will for the implementation on figure D.1, be determined by $2^N - 1$ where N is the number of bits.

The PRNG implemented in this project uses a simple LFSR as illustrated above, using 16 bits. Using 16 bits the sequence will be 65535 long, before it repeats it selves. This is considered to be long enough for testing purposes in this project. The seed is predefined to be the hexadecimal value 0001.

Listing D.1 show the main part of the PRNG implemented in VHDL. The generator is driven by the clock and uses the psuedorandom number that is generated as output. The seed is on line 2 defined as a variable, *rand_temp*, setting LSB to one and the following to zero. At each rising edge of the clock the two MSB are lead together in an XOR-gate and the result of this is stored in a variable *temp*. Next is the shift operations, all bits except the MSB is moved one up, towards MSB. Following is the result of the XOR operation set as the LSB. This procedure is repeated at every clock cycle.

```
process(clk)
    variable rand_temp : signed(bits-1 downto 0) := (bits-1 => '1', others => '0');
    variable temp : std_logic := '0';
begin
    if(rising_edge(clk)) then
        temp := rand_temp(bits-1) xor rand_temp(bits-2);
        rand_temp(bits-1 downto 1) := rand_temp(bits-2 downto 0);
        rand_temp(0) := temp;
    end if;
    random_num <= rand_temp;
end process;
```

Listing D.1: The code for pseudorandom number generator.

In this appendix the pseudorandom number generator is presented. It is explained how it works and the advantages of using it. The pseudorandom number generator designed in this project is described with supplement of the VHDL code of it.

E Energy Measurement and Estimation on Cyclone 3

The following chapter will contain a description on how to replicate the measurements and estimations performed on the Cyclone 3 development board. The chapter will describe the general set-up and procedure of the measurement and estimation, and elaborate on which errors and uncertainties this set-up gives.

E.1 Objective

The objective of this test will be to measure and estimate the energy that the Cyclone 3 FPGA consumes when different filters are implemented on it. The measured energy is found by measuring the voltage drop over a shunt resistor with a known resistance multiplies with the time for execution. This set-up is illustrated on figure E.1.

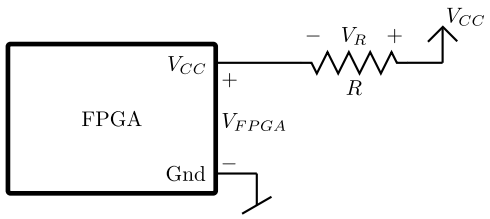

Figure E.1: The measuring set-up for tests on FPGA.

The energy can thereby be calculated as:

$$E = V_{FPGA} \cdot \frac{V_R}{R} \cdot T \tag{E.1}$$

V_{FPGA} is the voltage drop over the FPGA, which in this case will be $1.2V - V_R$. R is the shunt resistance and is mounted on the board. The size of the resistor is in these tests is 1Ω. T is the time which the filter takes to calculate an output. The mathematical expression above can now be simplified as:

$$E = (1.2 - V_R) \cdot V_R \cdot T \tag{E.2}$$

E.2 Equipment

The equipment used in this test are:

Name:	Description:	AAU number:
Cyclone 3 FPGA Starter Kit	FPGA development board	NJ14-3-015-1434-01
Agilent 34401A	6 1/2 Digit Multimeter	52858-00
Agilent 54825A	Infiniium Oscilloscope	52856

Table E.1: Equipment used for measuring energy consumption on the FPGA.

An important connector on the FPGA development board is JP6, which holds a pin on each side of the shunt resistor. V_R is found by measuring the voltage drop over the pins in JP6. The FPGA's core supply pins are connected to the shunt resistor, which means that all current supplying the internal core runs through this resistor.

E.3 Set-up

The set-up of the measurement is as described in the following:

- The power supply is connected to the FPGA development board.
- The USB cable is connected to the FPGA development board and to a computer.
- The measurement instrument is connected to the JP6 connector with a measurement probe.

E.4 Procedure

The procedure of measuring and estimating the energy consumption of the Cyclone 3 FPGA is as the following:

1. Compile the Quartus project which contains the VHDL or Verilog code of the filter which is going to be tested.
2. Estimation:
   (a) In Quartus settings window, under EDA Tool Settings -> Simulation, glitch filtering is enabled to get a high accuracy in PowerPlay.
   (b) Perform a gate-level simulation to create a Value Change Dump (VCD) file. The simulation is in this project executed in Altera Modelsim. To generate a VCD file, the code in listing E.1 is executed in Modelsim.
   (c) In Quartus settings window, under the PowerPlay settings tab, the created VCD is added to the input file list.
   (d) The PowerPlay process can now be started and the output can be watched in the Compilation Report (described below).
3. Measurement:
   (a) Turn on the development board and the measurement instrument.
   (b) Set the oscilloscope to show $10mV/div$ vertically and $100\mu s/div$ horizontally. The oscilloscope is in addition also set to show the RMS DC voltage.
   (c) Program the FPGA using the programming tool in Quartus.
   (d) The mean RMS DC voltage drop over the shunt resistor is now showing on the measurement instrument. The value is noted.

```
1 vsim gate_work.c3
2 force -freeze sim:/c3/osc_clk 1 0, 0 {10000 ps} -r 20ns
3 vcd file ../../c3.vcd
4 vcd add -r /*
5 run 2us
6 vcd checkpoint
7 quit -sim
8 quit
```

Listing E.1: Code for creating a VCD file in Altera ModelSim.

E.4.1 PowerPlay report

This section will elaborate on how to read out the results from the PowerPlay report and put into context to the actual measurement.

When the PowerPlay process is finished analysing the compiled and simulated VHDL code, a summary report can be found in Quartus. The summary could look as in table E.2.

PowerPlay Power Analyzer Status	Successful - Thu Apr 12 10:55:52 2012
Quartus II 32-bit Version	11.1 Build 173 11/01/2011 SJ Web Edition
Revision Name	c3
Top-level Entity Name	c3
Family	Cyclone III
Device	EP3C25F324C8
Power Models	Final
Total Thermal Power Dissipation	169.15 mW
Core Dynamic Thermal Power Dissipation	69.06 mW
Core Static Thermal Power Dissipation	87.83 mW
I/O Thermal Power Dissipation	12.26 mW
Power Estimation Confidence	High: user provided sufficient toggle rate data

Table E.2: Example of how a PowerPlay summary report could look.

The first thing to check is that the confidence of the power estimation is high, which indicates that PowerPlay had enough data to make a sufficiently good estimation. The top of the report down to and with *Power Models* holds information of the date, Quartus version, project name, top-level name, device and which power model that has been used. The following power measures are the total power estimation of the whole FPGA. The current running through the shunt resistor is however only to the internal core, why this has to be found in the PowerPlay section called *Current Drawn from Voltage Supplies*. The report in that section looks as in table E.3.

Voltage Supply	Total Current Drawn	Dynamic Current Drawn	Static Current Drawn
VCCINT	84.03 mA	79.59 mA	4.44 mA
VCCIO	7.21 mA	3.41 mA	3.80 mA
VCCA	33.45 mA	2.19 mA	31.26 mA
VCCD	11.11 mA	5.59 mA	5.52 mA

Table E.3: Example of how the PowerPlay *Current Drawn from Voltage Supplies* section could look.

The current running through the shunt resistor is that noted under *VCCINT* and *VCCD*, which denotes the internal core and digital supplies respectively. By reformulating eq. (E.1) the energy consumed can hereby be calculated.

E.5 Preliminary test

As in section C.5, a test is conducted in order to compare the measurements done by an oscilloscope and a multimeter. Table E.5 shows the measurements done for four filter with different lengths. The oscilloscope is set to calculate the DC RMS, which includes both the DC and AC component. The total RMS value is found with the multimeter, by measuring both AC and DC measures, and the total RMS voltage is found by calculating:

$$V_{RMS} = \sqrt{V_{DC\,RMS}^2 + V_{AC\,RMS}^2} \tag{E.3}$$

Table E.4 shows the AC and DC measures from the multimeter.

Filter length	AC RMS	DC RMS
8	2.552 mV	40.169 mV
40	5.031 mV	92.611 mV
80	7.028 mV	128.68 mV
120	6.890 mV	183.29 mV

Table E.4: AC and DC multimeter measurements.

Seen from table E.5, the differences between the measurements made by the oscilloscope and the multimeter are relative small.

Filter length	Voltage drop over shunt resistor		Difference
	Oscilloscope	Multimeter	
8	39.75 mV	40.250 mV	1.242 %
40	94.08 mV	93.746 mV	0.355 %
80	129.86 mV	128.872 mV	0.761 %
120	184.23 mV	183.420 mV	0.440 %

Table E.5: Comparison of oscilloscope and multimeter measurements.

Based on these measurements both the oscilloscope and the multimeter are considered appropriate choices of measuring devises. They do not give the exact same measuring results but due to the relative small variation in the difference, both are seen as possible measuring devices. The measurement uncertainties related to the AC measurements on the multimeter might favour the oscilloscope compared to the multimeter. The AC component does affect the total voltage to some extent, and with the uncertainties in this measure the multimeter might give inaccurate results.

It is chosen to use the oscilloscope to find the energy dissipated by the FPGA.

E.6 Source of errors and uncertainties

This section will describe the most evident sources of errors and uncertainties in the way that the measurement has been conducted.

A source of error and uncertainty is that the oscilloscope and multimeter used to measure the voltage drop measures the RMS value. This RMS value may not be sufficient in all cases. In most of the measurements conducted in this project, the RMS value will however be sufficient as all measurements are conducted in this way, which will eliminate this as an error.

When doing RMS measurements an important factor is noise. Noise can be introduced in this test set-up at several points. The cables used in the test set-up can be subject of radiated noise from e.g. other electrical devices. The cables is however shielded which will minimize the amount of radiated noise. The board is equally a subject of radiated noise, especially as this is not shiAs mentioned in section C.5 measurements of the AC component made by the multimeter may not be accurate due to the asymmetrical shape of the signal.elded and the pins and solderings are therefore exposed.

The shunt-resistor mounted on the development board is from Altera set to have a resistance of $1\Omega \pm 1\%$. The resistor has a tolerance and is affected by its temperature, which will give an uncertainty in the measurements. It can however be noted that the resistance will most likely not change when performing the measurements in this project, so the comparison of energy will still hold. When comparing the measurements with the estimations, uncertainties has to be expected.

The temperature of the surroundings will also affect the measurements, as the energy dissipation will vary as a function of the temperature of the FPGA.