TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

JUSSI KILPELÄINEN

AUTOMATED FLOW FOR GENERATING HARDWARE DESCRIPTION OF FILTERS

Master of Science thesis

Examiner: Prof. Timo D. Hämäläinen
Examiner and topic approved by the Faculty Council of the Faculty of Computing and Electrical Engineering on 5th November 2014

ABSTRACT

JUSSI KILPELÄINEN: Automated flow for generating hardware description of filters
Tampere University of Technology
Master of Science thesis, 53 pages, 0 Appendix pages
April 2015
Master's Degree Programme in Electrical Engineering
Major: embedded systems
Examiner: Prof. Timo D. Hämäläinen
Keywords: filter, generation, fir, rtl

Digital filters are used to process signals in many fields like telecommunications, image processing and in medical equipment. They are so omnipresent that engineers are building and verifying those all the time, using a lot of resources.

As the structure of a basic filter is quite simple, savings could be made by automatizing the creation of filters. In this Thesis the requirements of Nokia Networks are analyzed to build an automatized filter generation flow. Different tools are evaluated, but finally a custom generator is built. It crafts filters from pieces of hand-written RTL.

The end result is an automated flow which supports single and multichannel FIR filters with constant coefficients. The user has to input the coefficients to a Matlab script with the desired data widths. The filter is then generated and verified by running the script. The flow supports both ASIC and FPGA technologies.

TIIVISTELMÄ

JUSSI KILPELÄINEN: Automatisoitu vuo suodinten laitteistokuvauksen tuottamiseen
Tampereen teknillinen yliopisto
Diplomityö, 53 sivua, 0 liitesivua
huhtikuu 2015
Sähkötekniikan koulutusohjelma
Pääaine: sulautetut järjestelmät
Tarkastaja: Prof. Timo D. Hämäläinen
Avainsanat: suodin, generointi, fir, rtl

Digitaalisia suotimia käytetään signaalien käsittelyyn monilla eri tekniikan alueilla, kuten telekommunikaatiossa, kuvankäsittelyssä ja lääketieteellisissä laitteissa. Ne ovat niin yleisiä, että insinöörit käyttävät paljon aikaa ja resursseja niiden toteuttamiseen ja verifioimiseen.

Koska yleisimpien suotimien rakenne on melko yksinkertainen, niiden luominen voidaan automatisoida generaattorin avulla. Tässä diplomityössä Nokia Networksin vaatimukset kartoitetaan automatisoidun suodinten laitteistokuvauksen tuottamisvuon kehittämiseksi. Erilaisia tuottamismenetelmiä vertaillaan, mutta lopulta päädytään kehittämään oma generaattori. Se luo suotimia yhdistelemällä osia käsinkirjoitetusta RTL:stä.

Lopputuloksena on automatisoitu vuo, joka tukee vakiokertoimilla varustettuja, yhden tai useamman kanavan FIR-suotimia. Käyttäjän tulee syöttää kertoimet ja haluttu datanleveys Matlab-skriptiin. Ajettaessa skripti luo suotimen ja verifioi sen. Vuo tukee sekä ASIC- että FPGA-teknologioita.

PREFACE

The work in this Thesis was carried out at Nokia Networks in Tampere, Finland between October 2014 and February 2015.

In the words of John Donne, "No man is an island". There were many persons who helped me do this Thesis. I am very grateful to the supervisor of this Thesis, Dr Tuomas Järvinen, whose advice and experience were very valuable throughout the process. I also regard the technical advice given by Juha Nousiainen very highly. Professor Timo D. Hämäläinen from Tampere University of Technology examined this Thesis and gave a lot of helpful comments. I also appreciate my line manager Didier Capiten for giving me the opportunity for doing this Thesis while being employed. Thanks to you all.

This is the culmination of a long process, which started the day I went to school in 1997. I wish to take this opportunity to thank two teachers who really helped me develop my thinking: Mikko Pörsti from Valtimon lukio and Seppo Toivanen from Rautavaaran lukio.

Most of all, thanks to my friends and family for all the support during my studies and while doing this Thesis.

Tampere, 25.02.2015

Jussi Kilpeläinen
jussi.kilpelainen@iki.fi

TABLE OF CONTENTS

1. Introduction 1

2. Filter theory 4

2.1 Basic topologies 4

2.1.1 Direct FIR filter 5

2.1.2 Mapping block diagrams to hardware 5

2.1.3 Transposed FIR filter 6

2.2 Structural optimizations 7

2.2.1 Folded FIR filter 7

2.2.2 Half-band filter 8

2.2.3 Coefficient area 9

2.2.4 Resource sharing 9

2.2.5 Multichannel filter 11

2.3 Multiplier optimizations 13

2.3.1 Shift and add algorithm 13

2.3.2 Canonical signed digit 14

2.3.3 Reduced adder graph 14

2.4 Current research 15

3. Requirements 17

3.1 Constraints 17

3.2 Existing tools for generating filters 18

3.2.1 Generators by research groups 18

3.2.2 Generators by FPGA vendors 19

3.2.3 High level synthesis tools 20

3.2.4 Own implementation 20

3.3 Verification 21

3.4 The flow 21

3.4.1 From Matlab model to RTL 24

3.4.2 Verifying the RTL . . . 26

4. Filter generator . . . 29

4.1 Multiplier block test syntheses . . . 29

4.2 Building a filter by hand . . . 32

4.3 Automating the generation . . . 36

4.4 Results . . . 38

4.4.1 FPGA synthesis . . . 38

4.4.2 ASIC synthesis . . . 40

5. Extension to multichannel filters . . . 42

5.1 Functionality . . . 42

5.2 Implementation . . . 46

5.3 Results . . . 48

6. Conclusions . . . 49

Bibliography . . . 51

LIST OF FIGURES

1.1 Sine wave that gets corrupted by noise and is then filtered. 1

1.2 A noisy image (left) that has been filtered (right). 2

1.3 Filtering an electrocardiogram. 2

2.1 A four-tap FIR filter. 5

2.2 A transposed four-tap FIR filter. 6

2.3 A folded four-tap FIR filter. 7

2.4 A folded five-tap FIR filter. 8

2.5 A multiply-accumulate unit (MAC). 10

2.6 Distributed arithmetic filter. 10

2.7 A multichannel filter as described by Ming and Chao [12]. 11

2.8 A multichannel filter as described by Kukkala [8]. 12

2.9 A RAG multiplier block with coefficients 66, 905, 294, 2282 and 5032. 15

3.1 Screenshot of Altera FIR Compiler II 19

3.2 The filter generation flow. 22

3.3 Directory structure used by the flow. 22

3.4 Floating point and fixed-point implementations of a filter. 25

4.1 Eight input binary tree adder (design III). 30

4.2 Maximum operating frequency on an FPGA. 39

4.3 Number of logic elements used on an FPGA. 39

4.4 Number of embedded multiplies used on an FPGA. 40

4.5 Silicon area as a function of the filter length. 41

5.1 Structure of the multichannel filter. 43

5.2 Timing diagram for the multichannel filter. 44

5.3 Delay line register stages sharing the same memories. 45

5.4 Dividing the delay line to different memories. Context switch period is 8 cycles. 46

5.5 Simplified example of the data flow from memory to the shadow registers. 47

LIST OF TABLES

3.1 User settings in the Matlab script. 23

4.1 Synthesis results (μm^2) 31

4.2 Hooks in the simple filter template 37

4.3 FPGA synthesis results 38

4.4 ASIC synthesis results 40

5.1 Effects of choosing the context switch period. 45

5.2 Memory addresses in context memory. 45

5.3 Multichannel filter synthesis results. Areas in mm^2. 48

LIST OF ABBREVIATIONS

ASIC	Application specific integrated circuit
CD	Compact disc
CSD	Canonical signed digit
CSE	Common subexpression elimination
DA	Distributed arithmetic
DAC	Digital to analogue converter
DFE	Digital front-end
DUT	Device under test
ECG	Electrocardiogram
EEG	Electroencephalogram
FEC	Focused expression coverage
FIR	Finite impulse response
FPGA	Field-programmable gate array
HDL	Hardware description language
HLS	High level synthesis
IIR	Infinite impulse response
IP	Intellectual property
LE	Logic element
MAC	Multiply and accumulate
PHP	PHP: Hypertext Preprocessor (a programming language)
RAG	Reduced adder graph
RAM	Random access memory
RNS	Residue number system
RTL	Register-transfer level
UVM	Universal verification Methodology
VHDL	VHSIC hardware description language
VHSIC	Very high speed integrated circuit
VIP	Verification intellectual property
XML	Extensible markup language

1. INTRODUCTION

Filters are used in signal processing to change the spectral content of a signal. Traditionally they have been implemented in analogue domain using passive components and operational amplifiers. Figure 1.1 gives an example of a sine wave which gets corrupted by noise. The noisy signal is then filtered with a low-pass filter, resulting in the signal in the right side of the figure. The bottom part of the figure shows the magnitudes of the signals in frequency domain.

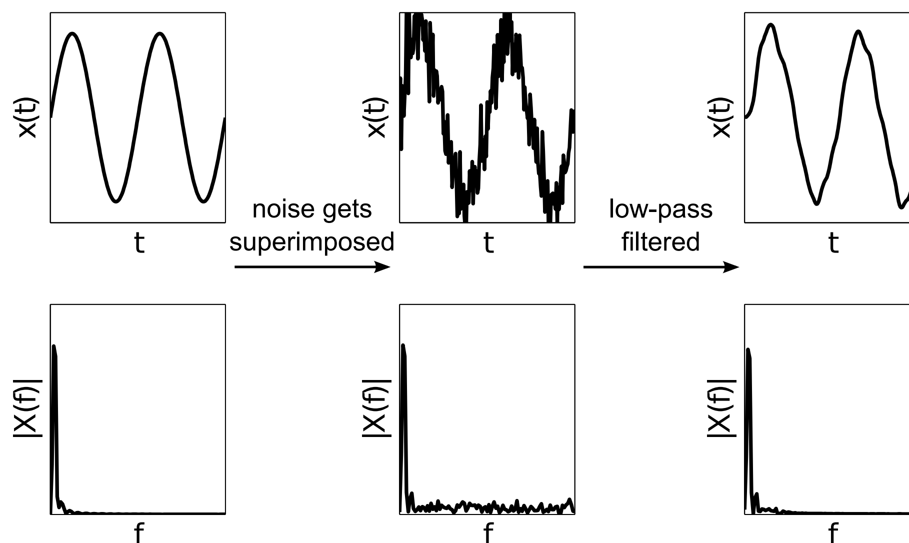

Figure 1.1 *Sine wave that gets corrupted by noise and is then filtered.*

Digital filters are superior to their analogue counterparts in many ways. They are cheaper and more immune to the tolerance of component values or changes in the environment. Implementing them is easy in software. They can also be synthesized to a field-programmable gate array (FPGA) or an application-specific integrated circuit (ASIC). Some applications, like software defined radios, make use of their capability of being adjustable on the fly.

Decreased cost is the reason why digital filters are used in many compact disc (CD) players. A CD player converts the digital information from the disc to an analogue output signal. To prevent signal aliasing, a reconstruction filter is required. A

common trick in the industry is to oversample the audio signal before the digital-to-analogue converter (DAC). The signal can then be filtered effectively in the digital domain, and the final analogue filter can be simpler. A very steep analogue filter would also cause phase changes, worsening the reproduction quality.

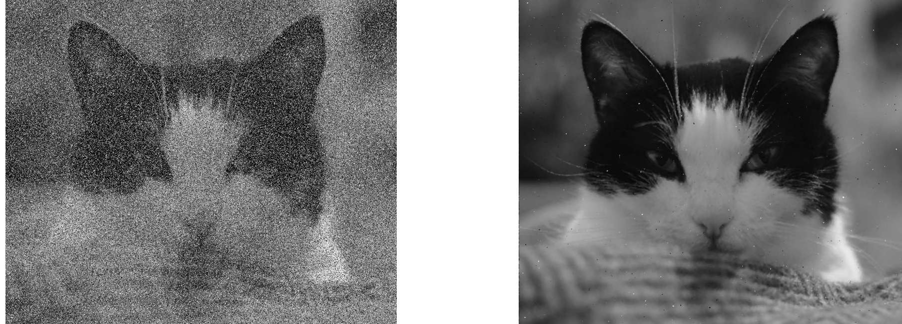

Figure 1.2 *A noisy image (left) that has been filtered (right).*

Besides audio, the same filtering principles can also be applied to image processing. For example, transmission errors or defects in the image sensor may cause salt and pepper noise, which can be removed by applying a median filter like shown in figure 1.2.

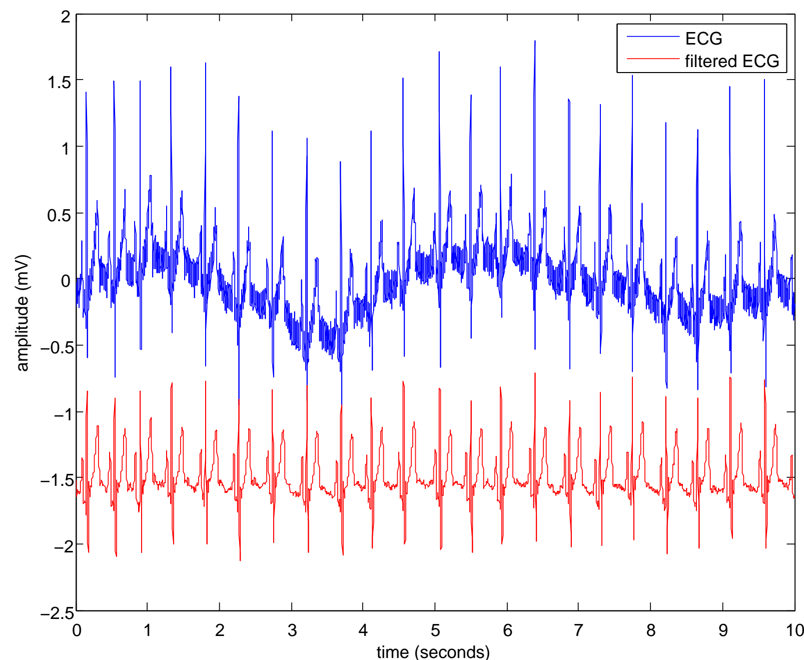

Figure 1.3 *Filtering an electrocardiogram.*

Filters are also used in medical applications. Biosignals like electrocardiograms (ECG) and electroencephalograms (EEG) are often very weak and susceptible to all kinds of interference from the environment. Perspiration, respiration and muscle

activity can also have a negative effect in the signal. Getting rid of these unwanted phenomena requires a lot of filtering. See figure 1.3 for an example.

Telecommunications have a lot of applications for filters as well. When a listener tunes her radio receiver to a particular station, she changes the center frequency of the channel filter. The filter removes unwanted signal from the demodulator input, improving the signal-to-noise ratio. Similar functionality exists in mobile phones and network base stations, for example.

Given how ubiquitous digital filters are, it is no wonder that implementing them is routine for many engineers. At Nokia Networks in Finland it was deemed that too much engineering effort was put in implementing simple filters from scratch over and over again. Moreover, as there was no generic test bench available, verification took a long time as well. This Thesis solves this problem by introducing an automated flow for generating and verifying the VHDL (VHSIC Hardware Definition Language, where VHSIC stands for Very High Speed Integrated Circuit) descriptions for the most commonly used digital finite-length filters in telecommunication applications.

The problem is approached by collecting the requirements and evaluating different filter toolkits. There are lots of pre-existing generators and a lot of research papers with interesting optimizations, but to keep things as simple as possible, a custom generator is built. The effort included choosing a suitable filter topology for automatization and performing trial syntheses on arithmetic blocks to get a reasonable balance between simplicity and efficiency. A Python generator script was written to combine the pieces of hand-written RTL into a filter. For ease of use, the generator is placed in a Matlab-driven flow which was developed for this application and includes an automatically created, self-checking test bench.

Besides documenting how the generator and the flow were created, this Thesis will also describe how to use them so that it can be used as a user manual. Maybe it could also help people with a computer engineering background and little experience in signal processing to get started with turning the requirements into designs. The focus is kept on filter implementations, not on the filter design.

The structure of this Thesis is as follows: Chapter 2 sets the theoretical background for the Thesis. The requirements are collected and a the flow described in Chapter 3. Chapter 4 tells about the RTL designs and their test synthesis results. In Chapter 5 the generator is expanded to support multichannel filters. Chapter 6 concludes the Thesis.

2. FILTER THEORY

There are two types of digital filters: finite impulse response (FIR) and infinite impulse response (IIR) filters. The names are very descriptive: a FIR filter has a finite impulse response - it will settle to zero in a finite time. On the other hand, IIR filters perform an infinite sum which does not become zero past a certain instant. [11, p.165]

Early on, a decision was made to concentrate on FIR filters. They have a lot of good properties, like inherent stability and the ease of designing a filter with linear phase [17]. Therefore they are more often used in projects at Nokia. Moreover, their simplicity makes implementing the automated flow easier. Support for IIR filters could then be added later if there is a need. Only FIR filters are considered in this and all subsequent chapters.

2.1 Basic topologies

The output of a FIR filter is a weighted sum of the past and present inputs. The output does not depend on the previous outputs, [1] making the filter non-recursive.

The relation can be written formally as

$$y[n] = \sum_{k=0}^{M-1} h[k]x[n-k], \tag{2.1}$$

where y is the output, x the input, h the filter coefficients and M the number of taps. The number of taps is equal to the number of coefficients. The filter order is one less than the tap count[14, p.432]. The filter coefficients h are often referred to as the impulse response of the filter. [9, p.159]

In its most general form without any optimizations a FIR filter has two common hardware topologies. They are presented in this section.

[1] With the exception of cascaded integrator-comb filters [11, p.166], which are outside the scope of this Thesis.

2.1.1 Direct FIR filter

Figure 2.1 represents the simplest type of the FIR filter, which is called a direct filter. The input $x[n]$ is fed into a delay line. Each tap of the delay line is multiplied by a coefficient, and the products are summed together to form the output $y[n]$. Because of the architecture these filters are sometimes called tapped delay lines [11, p.166] [14, p.436]. The coefficients may then be referred to as tap weights.

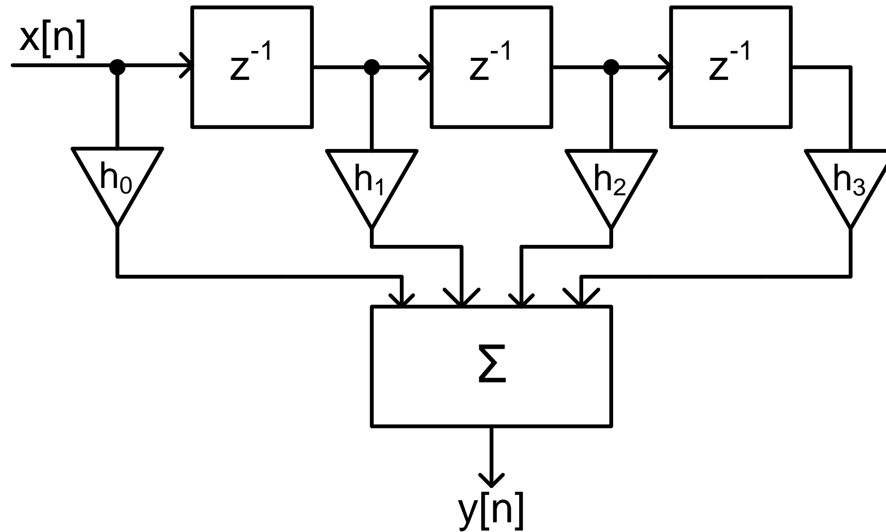

Figure 2.1 *A four-tap FIR filter.*

Applying the formula 2.1 for the filter in figure 2.1 results in

$$y[n] = h_0 \cdot x[n] + h_1 \cdot x[n-1] + h_2 \cdot x[n-2] + h_3 \cdot x[n-3]. \tag{2.2}$$

In literature formula 2.1 is often written using a convolution

$$y[n] = h[k] * x[n], \tag{2.3}$$

which corresponds to a multiplication $Y(z) = H(z)X(z)$ in frequency domain. [9, p. 160]

2.1.2 Mapping block diagrams to hardware

Creating a hardware implementation of the block diagram of simple filters is straightforward. The delay line can be implemented using a shift register. Addition is naturally done by adders. Multiplier components can take care of different weights of the taps.

However, things get a bit more difficult if there are strict speed or area restrictions that the filter must fulfill. Multipliers and adders must then be pipelined to improve throughput.

Optimizing the multipliers is the key to an efficient design because multiplication is a much more complex operation than addition [7], resulting in a larger silicon area or a longer latency. There are also a few methods that can be used in the filter design phase to decrease the number of mathematical operations required. Moreover, the multiplications can be optimized further if the coefficients are constant and known at the time of implementation. In that case the multipliers can be replaced with implementations that utilize bitshifting. [3, p.131]

2.1.3 Transposed FIR filter

The direct FIR filter can be transposed by turning it around: the input and the output change places, signal flow direction is changed and forks are replaced with adders and vice versa. [11, p. 167] The result can be seen in figure 2.2. Input is multiplied with all the coefficients, and the products are summed in the delay chain with the output of the previous delay element.

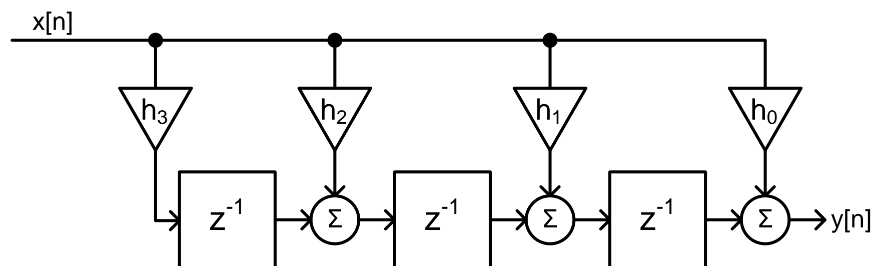

Figure 2.2 *A transposed four-tap FIR filter.*

It can be argued that this structure is simpler than the direct topology presented before. The final summation of the direct filter requires a pipelined adder tree whereas the transposed structure doesn't [11, p.167]. It also opens up many interesting optimization methods; as all the multipliers have the input as the other operand, there is possibly a lot of redundancy in the multipliers. On the other hand, the registers it requires are large because they store the multiplier outputs in full precision instead of just the delayed input. This is, however, less of a concern in FPGA implementations where registers are plentiful [11, p. 188]. Moreover, the multiplication results are often rounded to save logic at the expense of increased noise.

2.2 Structural optimizations

The basic structure of a FIR filter was presented in Section 2.1. By designing the filter carefully, some logic in the basic implementations can be made redundant. It is also possible to perform optimizations by reusing the same hardware for multiple filters.

2.2.1 Folded FIR filter

Often it is desired that the group delay of a filter is constant. It means that it takes the same amount of time for different frequency components to go through the filter and no phase distortion has been added to the signal.

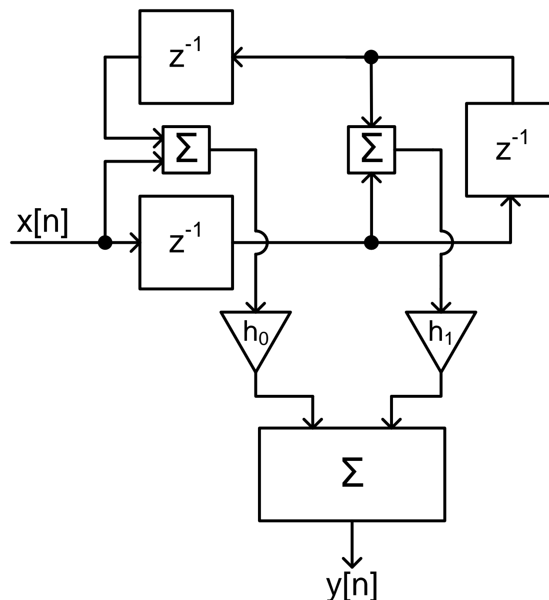

Figure 2.3 *A folded four-tap FIR filter.*

It can be easily shown that in order to have a linear phase, the FIR coefficients must be symmetrical $h[k] = h[(M-1)-k]$ or antisymmetrical $h[k] = -h[(M-1)-k]$, where M is the number of taps. [9, p.503][11, p.171][14, p.436][17, p.164] For example, if the four coefficients in figure 2.1 satisfy the criteria $h_0 = h_3$ and $h_1 = h_2$, equation 2.2 can be written as

$$\begin{aligned} y[n] &= h_0 \cdot x[n] + h_1 \cdot x[n-1] + h_1 \cdot x[n-2] + h_0 \cdot x[n-3] \\ &= h_0(x[n] + x[n-3]) + h_1(x[n-1] + x[n-2]). \end{aligned} \tag{2.4}$$

As a result, two of the multiplications can be replaced with summations. Generally speaking, if there are M taps, the direct topology requires M multipliers whereas folded topology needs only $\lceil \frac{M}{2} \rceil$. This fits in extremely well with the desire to minimize the number of multipliers in the filter. A block diagram of the folded filter can be seen in figure 2.3.

If the number of coefficients is odd, the coefficient in the middle will perform as the symmetry axis and therefore the middlemost tap will have no pair to be summed with. The middle tap will have to be multiplied alone and added to the final sum. For an example, see figure 2.4.

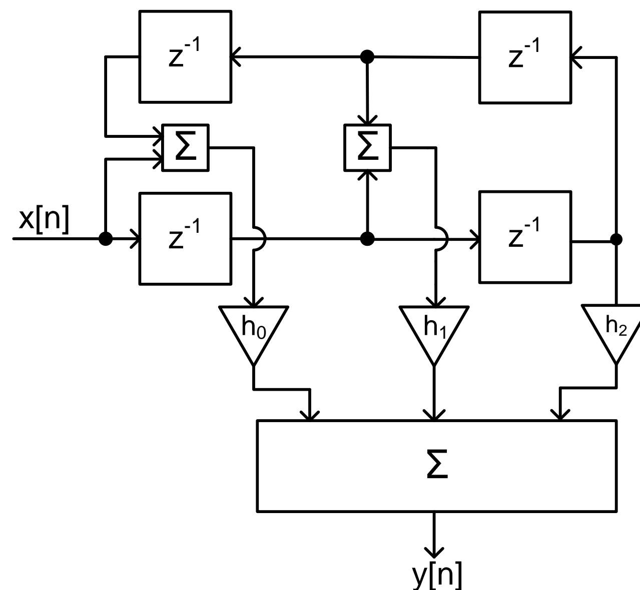

Figure 2.4 *A folded five-tap FIR filter.*

2.2.2 Half-band filter

In non-trivial systems there are likely to be multiple points where the sample rate is changed in order to interface between different subsystems or to decrease the computational load. Increasing the sample rate is called interpolation, and it is done by adding new samples between the existing ones. Reducing the sample rate is called decimation, and it can be accomplished by discarding samples. [9, p. 381][14, p.50]

There is a type of FIR filter, a half-band filter, that is often used as an image removal filter when changing the sample rate [9, p.188]. They have been designed so that their frequency response is symmetrical around the frequency $f_s/4$, where f_s is the sampling frequency. This symmetry causes a very useful phenomenon to occur in

time domain: every other filter coefficient becomes zero except for the middlemost one. [9, p.188][14, p.782]

Half-band filters do have a limitation though: if the number of taps is M, M+1 must be a multiple of four [9, p.188]. This is however a small downside compared to the fact that almost half of its coefficients are zeroes. The amount of multipliers required is therefore also nearly halved.

2.2.3 Coefficient area

As described in formula 2.1, a FIR filter consists of multiplying the delayed input samples with the coefficients and adding the results together. Given an input bit width w_i and a coefficient width w_c, the multiplication output will be of size $w_r = w_c + w_i$. Summing M of those together will result in a number of width $w_r + \lceil log_2(M) \rceil$.

However, these calculations result in the number of bits that is required in order to not have overflow when the input and the output may have any value within the range. With fixed coefficient FIR filters a better estimate can be calculated by taking the magnitudes of the coefficients into account. To find out the maximum output value of the filter, the input data must be assumed to be the worst case of x_{max}, namely the highest value that will fit in the number of bits. Now formula 2.1 can be rewritten to give the maximum output value

$$\begin{aligned} y_{max} &= \sum_{k=0}^{M-1} |h[k]| \cdot x_{max} \\ &= \alpha \cdot x_{max}, \end{aligned} \tag{2.5}$$

where α is the sum of the absolute values of coefficients. This value is sometimes called the *coefficient area* of the filter. [23] In most cases, determining the output width using the coefficient area will result in a smaller number than the analysis done by using the number of taps. Accumulator registers can then be made smaller, and silicon area is saved.

2.2.4 Resource sharing

The direct, folded and transposed filters perform all the calculations in parallel and are therefore very fast. However, sometimes the speed requirements are not so high

and saving silicon area is more important. In those cases it is reasonable to perform the calculations using less hardware over a longer period of time.

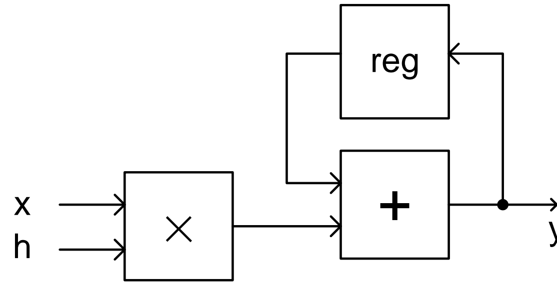

Figure 2.5 *A multiply-accumulate unit (MAC).*

One way of doing this is by using multiply-accumulate units (MACs) as shown in figure 2.5. Control logic, which is not drawn in the figure, feeds in the values for h and x from formula 2.1 for each clock cycle, and the result is accumulated in y. For a filter with N taps the operations can be done in N cycles with a single MAC unit. The designer can balance the speed and area costs by using several MAC units in parallel.

Distributed arithmetic (DA) filters utilize another method of sharing resources by calculating the outputs in a bit-serial way. Their main idea is to perform the calculations one bit at a time for all the coefficients and then do a weighted sum as shown in figure 2.6. The operations are done using bit shifts, lookup tables and an adder. [10][11, p.189]

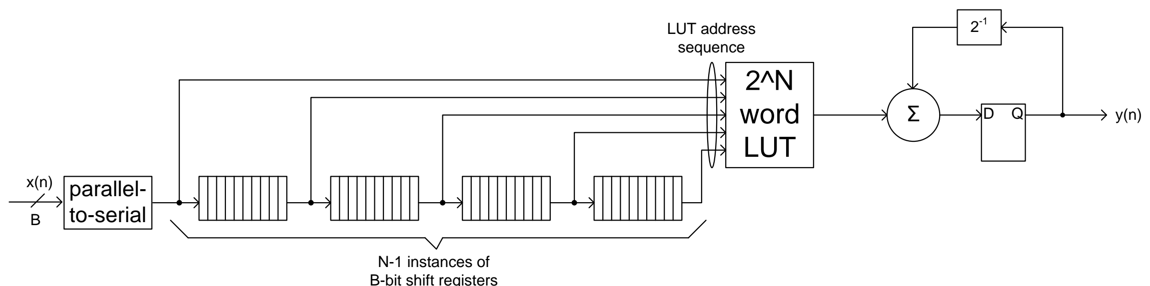

Figure 2.6 *Distributed arithmetic filter.*

The number of clock cycles required to calculate the output for a single input is dependent on the coefficient bit width. Increasing the filter length does not worsen the performance, like it does with single MAC filters [11, p.114], but it does increase the lookup table size. [21]

This architecture is most popular in field-programmable gate array (FPGA) implementations because it maps well to the architecture [10]. For short filters the internal

lookup tables (LUTs) can be used. Longer filters tend to utilize memory blocks, or use muxing with LUTs. [11, p.192]

A lot of research goes into trying to optimize the memory usage in distributed arithmetic filters. Meher et al. [10] describe a method of decomposing the operations in order to be able to use smaller memories.

Uwe Meyer-Bäse has done a comparison of optimized, transposed FIR filters and DA filters on FPGAs. On average, transposed filters require 71% less logic but have 8% lower maximum operating frequency. [11, p.208] On the other hand, DA filters offer a predictable implementation - its cost and performance are not dependent on how well the coefficient values can be optimized. Switching between different coefficient banks is also easy as it only requires selecting the appropriate lookup table with a multiplexer.

2.2.5 Multichannel filter

Distributed element filters presented in Section 2.2.4 offered a simple way of supporting multiple sets of coefficients in a single filter. These kinds of filters are called multichannel filters, and the different sets of coefficients are often referred to as coefficient banks. There are also ways of implementing them with direct, folded and transposed topologies.

Ming and Chao describe [12] a multichannel filter in which the contents of the delay line are saved to RAM. It is illustrated in figure 2.7.

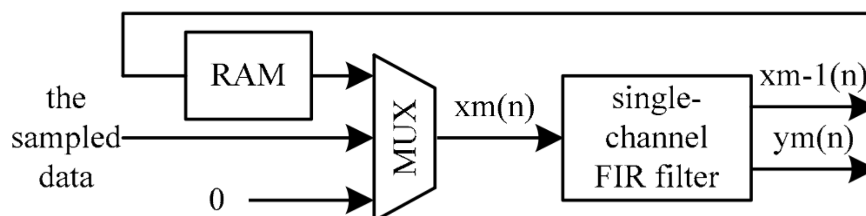

Figure 2.7 *A multichannel filter as described by Ming and Chao [12].*

During reset, the internal state of the filter is loaded with zeroes. In normal operation, the sampled data of channel m is fed into the multiplexer, producing output $ym(n)$. When changing the channel, the context — the contents of the delay line $xm - 1(n)$ — is saved to the memory. The context is then loaded when the channel is being processed the next time. Naturally, this solution requires a lot of control

logic related to managing the memory operations and controlling the multiplexer, none of which is drawn in the figure.

The filter supports any kind of a single-channel FIR filter design. However, support for multiple coefficient sets is not mentioned in the paper. The multiplier blocks could be built so that they support choosing the coefficients with a multiplexer or reading the multipliers from memory alongside the context.

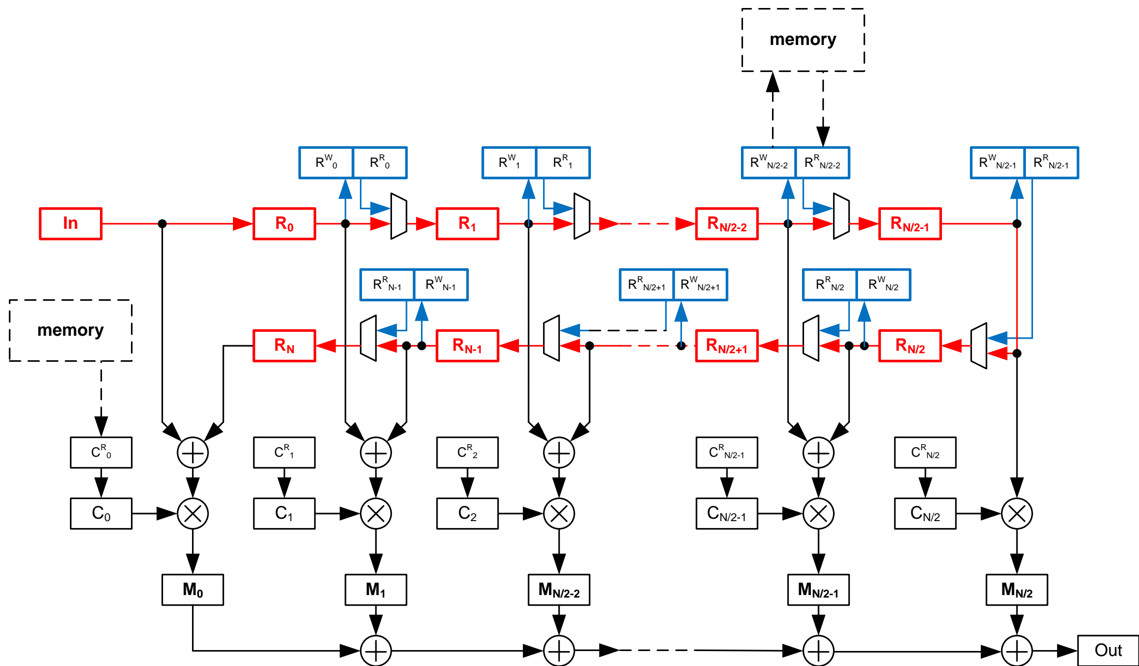

Figure 2.8 A multichannel filter as described by Kukkala [8].

A more complex multichannel filter has been developed at Nokia [8]. It has a folded delay line. When the channel changes, it saves the context to memory much like the Ming and Chao filter does. The filter also has support for multiple coefficient banks — the coefficients for each channel are loaded from the memory. The filter with its memory interfaces is depicted in figure 2.8.

The data path of a conventional FIR is emphasized with red in the figure. The blue paths are related to the context switching. The bottom part of the figure deals with the arithmetic operations.

When the channel changes, the contents of the registers R_0 ... R_{N-1} are saved in the R^W registers. The new context is loaded to R_1 ... R_N from the R^R registers. The data traffic between the memory and the shadow registers R^W and R^R is handled over multiple clock cycles during the normal operation of the filter.

One use case for a multichannel filter is the channel filter of a radio module. The input data comes from a streaming interface and each sample has a channel ID associated with it. The samples must be filtered with different coefficients depending on the carrier type.

The output of a multichannel filter is indistinguishable from the result that is got from parallel filters. However, a multichannel filter will generally have a smaller area and lower power consumption because it reuses the multipliers and has a lot of the registers replaced with memories. On the other hand, a multichannel filter requires shadow registers and a lot of control logic, and therefore parallel filters may be a more efficient implementation when the filters are short and there are only a few coefficient banks.

2.3 Multiplier optimizations

As mentioned before, the arithmetic cost is dominated by the multipliers. It can be reduced if the coefficients are known beforehand.

2.3.1 Shift and add algorithm

Multiplying a number with a power of two can be performed by bit shifting its binary representation to the left. This idea can be expanded further: each number can be represented as a sum of powers of two, so any multiplication can be done as a sum of bit shifts. For example, multiplying $7_{10} = 111_2$ with $5_{10} = 101_2$ can performed as

$$
\begin{aligned}
7 \cdot 5 &= (4 + 2 + 1) \cdot 5 \\
&= 5 \cdot 2^2 + 5 \cdot 2^1 + 5 \cdot 2^0 \\
&= (5 << 2) + (5 << 1) + (5 << 0) \\
&= 20 + 10 + 5 \\
&= 35.
\end{aligned}
\tag{2.6}
$$

Bit shift is often considered as a free operation as it requires just reconfiguring some of the wiring instead of adding gates. Therefore a shift and add multiplication is really appealing when compared to a full multiplier.

2.3.2 Canonical signed digit

The number of additions required in a shift and add multiplier increases as the number of '1'-bits in the number increases. However, the number of operations can be reduced 33% by adding subtractions to the set of operations. [6, p.157] The format is called canonical signed digit (CSD).

In CSD additions are marked with 1 and subtractions with a $\bar{1}$. For example, the CSD representation of 31_{10} is $10000\bar{1}$, which stands for $2^6 - 2^0$. Compared to the binary representation 11111 it has three less non-zero digits, and therefore three less bit shifts to be summed to complete the multiplication.

Converting a binary representation to CSD can be done by looking at patterns in the bits. Starting from the least significant bit, sequences of '1' longer than two are replaced with $10...0\bar{1}$, where the number of zeroes is one less than the number of ones in the pattern. At the same time patterns 1011 must be replaced with $110\bar{1}$. After the first iteration, patterns $10\bar{1}$ must be replaced with 011, starting from the most significant bit.[11, p.59]

2.3.3 Reduced adder graph

As mentioned in Chapter 2.1.3, transposed filters make it easy to look for redundant operations in the multiplications, which is something CSD does not utilize. By using a method called reduced adder graph (RAG), an improvement of 16 – 26 % can be done over CSD [4]. To help grasping the subject it is beneficial to think of all the parallel multipliers as a single block, with the data input of the filter as the input and all its multiples as the outputs.

Optimization begins by reducing all the coefficients to unique, positive-valued fundamentals. Those form the input set. The amount of adders required to implement each coefficient in the input set is checked from a pre-computed lookup table. Coefficients with an adder cost of zero or one are moved from the input set to the graph set, where the coefficients that the algorithm has realized are stored.

From there on, the algorithm tries to implement coefficients from the input set by adding, subtracting and bit shifting the coefficients from the graph set. When all combinations are exhausted, the algorithm moves more complex coefficients from the input set to the graph set and tries again. [11, p.183]

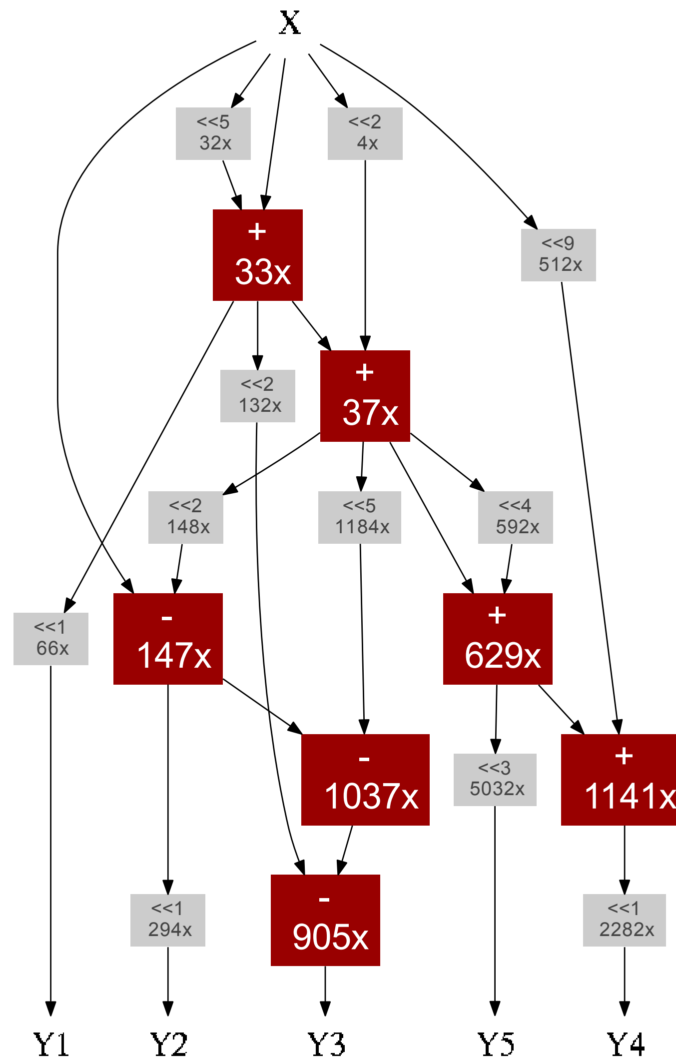

Figure 2.9 *A RAG multiplier block with coefficients 66, 905, 294, 2282 and 5032.*

Figure 2.9 presents a RAG multiplier block generated with SPIRAL multiplier block generator[15]. X is the input and Y1 through Y5 are the outputs. The block requires seven adders and has a depth of four.

2.4 Current research

There are endless little details that can be optimized in FIR filters, and they are therefore still a popular research topic. For example, Mirshekari et al. have experimented with residue number system (RNS) arithmetics in order to improve the

speed of arithmetic operations. [13] RNS introduces more parallelism and fewer carry operations than standard arithmetics.

Besides coming up with new or improved filter topologies, there is another popular branch of research which is automatizing filter generation. Research articles about this topic are addressed in Section 3.2.1.

3. REQUIREMENTS

The automated filter generation flow is meant to be used by Nokia, so the needs of the company were kept in mind throughout the process. To get a clear view of the requirements, a look was taken on a digital front-end (DFE) ASIC project to see what kind of filters are needed.

3.1 Constraints

Most of the filters in the designs are constant coefficient half-band filters used in decimation and interpolation blocks. Therefore that is the main type of filter that must be supported. The generated filters will be used with both ASIC and FPGA technologies. At the moment, most filter designs have a maximum clock speed of about 500 MHz.

There are no specific area constraints, but naturally the resulting filter shouldn't be much larger than the existing designs. Therefore using an existing design as a benchmark is reasonable.

Simplicity of the resulting VHDL file and of the scripting used is also very important. Engineers must be able to debug the VHDL and do modifications to make the filter fit the application. To guarantee a clear and consistent implementation when integrated into a module, the generated code must follow Nokia's VHDL guidelines.

It is also possible that maintaining the script will be someone else's responsibility in the future, and keeping everything as simple as possible decreases the learning curve and maintenance effort. Simple code has also most likely fewer bugs. As the HDL gets simpler and less optimized, the silicon area tends to increase. Managing this will require some trial synthesis runs.

It is also important that there is a bit-exact Matlab model of the filter. That way the flow user can be sure that the generated filter is functionally identical to the fixed-point model, and therefore meets the requirements. However, there is no simple way of making sure that the fixed-point model matches the original floating-point model.

The quantization that has been done may have had a negative effect on stopband attenuation, and it needs to be checked manually.

The whole purpose of automatization is to save the time of the engineers. Therefore the flow should be easy to learn and take little time to complete. It also means that the end user shouldn't have the need to know everything that's happening under the hood. The test bench must be self-checking.

There is a set of options the flow must support. Coefficients and the input and output widths and their fraction lengths must be freely adjustable. The generator must also support multichannel filters with fixed coefficients. The amount of subsequent input samples that need to be related to the same channel must be adjustable. Same applies to the number of coefficient banks and their lengths.

3.2 Existing tools for generating filters

There is nothing inherently complicated in programming FIR filters, and therefore building generic libraries or HDL generators has been done many times before. For example, Savela documents in his Master of Science thesis [18] a library of generic FIR and IIR filters.

3.2.1 Generators by research groups

Rosa et al. describe a complete flow for generating optimized FIR filters from transfer function to synthesizable VHDL. The generator uses common subexpression sharing (CSE), much like the reduced adder graph presented in Section 2.3.3. The tool also fine-tunes the coefficients to make the hardware implementation as simple as possible while maintaining the desired design constraints of the filter. [16]

In his Master of Science thesis Howard describes a PHP based VHDL generator for minimized adder graph optimized filters. The main focus is on FPGA devices, and a mix of embedded multipliers and common subexpression elimination is proposed as the most optimal solution. [5]

Verma and Chien introduce a generator, which produces efficient decimating filters. The logic has been optimized so that no effort is put in calculating values that are ignored by the decimator. Canonical signed digit (CSD) format is also used to save silicon area. [20]

3.2.2 Generators by FPGA vendors

FIR generators by Xilinx and Altera have been used in numerous Nokia's FPGA projects. The experiences have been mixed: while they do create a design that is a good match for the FPGA platform, the resulting HDL is very difficult to read and understand.

Both Altera FIR Compiler II and Xilinx FIR Compiler v5.0 were evaluated for this project to see if they would fit the flow. They both support interpolators, decimators, half-band filters and resource sharing [1] [22], to name a few. Both generators create distributed arithmetic filters and therefore support multiple coefficient banks.

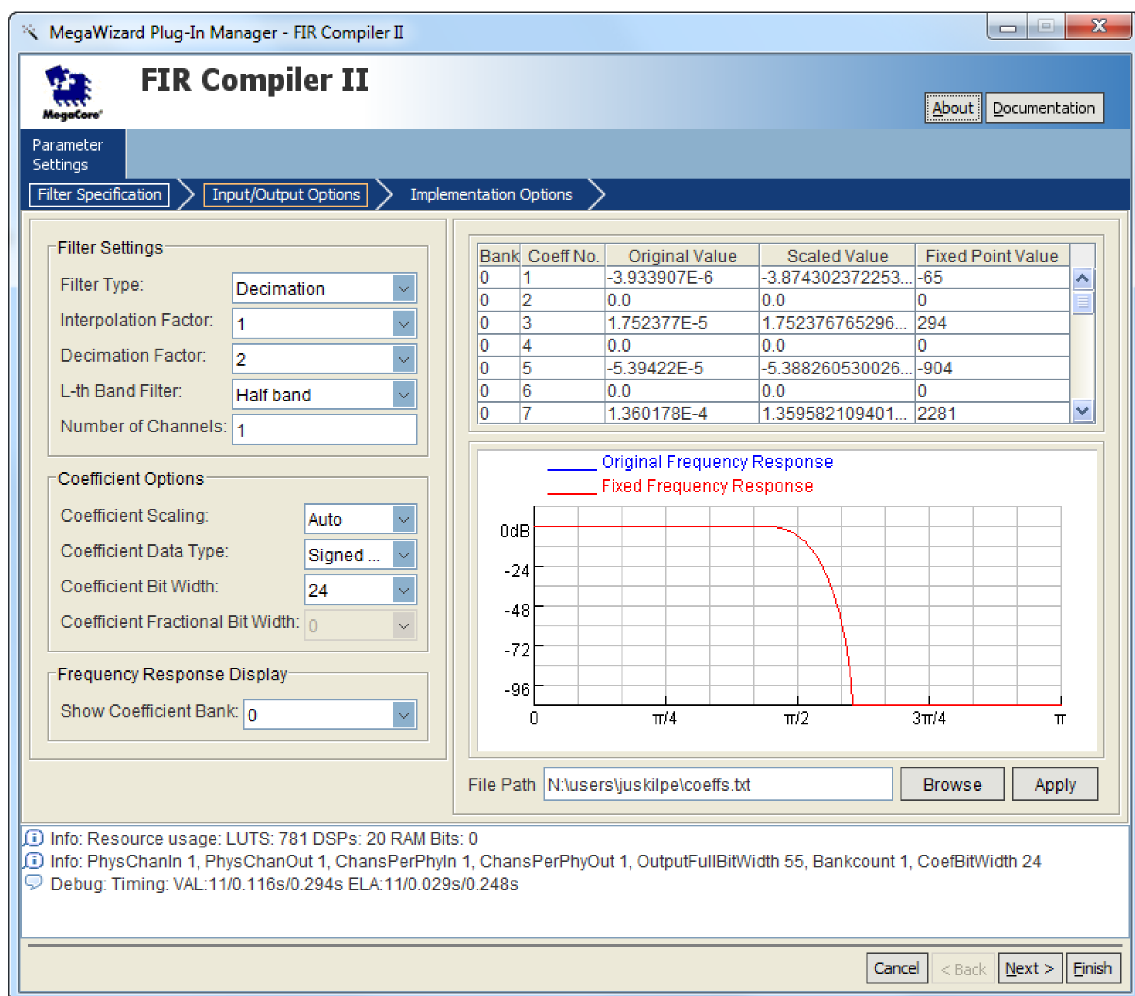

Figure 3.1 *Screenshot of Altera FIR Compiler II*

They gave valuable information about the different parameters that are required to generate filters and what kind of a experience the user may expect. Figure 3.1 is a screenshot of the Altera tool.

However, it was very difficult to get ideas for implementation as Xilinx VHDL was encrypted and Altera code was too complex. The core of the reference filter created with the Altera tool was a single 250 kilobyte VHDL file full of on-chip memories and similar FPGA-specific intellectual property (IP) blocks. It is simply not suitable for ASICs.

3.2.3 High level synthesis tools

There are numerous high level synthesis (HLS) tools on the market which can be used for generating filter HDL. For example, Calypto has Catapult-C, MathWorks offers HDL Coder and Cadence's tool is called C-to-silicon. During the course of this Thesis the tool vendors gave demonstrations of the tools, and it was quickly deemed that they are quite complex for simple filters.

First of all, it is a lot easier to control the synthesis results by generating VHDL than it is by generating C code and running it through a high level synthesis tool, for example. Moreover, if the flow depended on the HLS tool, its updates would have a high risk of breaking the flow. That could be balanced by decreasing the level of automation so that more is done in the HLS tool manually. That would be something that contradicts the usability requirements.

3.2.4 Own implementation

The tools presented before were deemed to be unsuitable for the generator. They were either too complex or lacking in ASIC support.

Availability of the tools is also something that must be considered. There may be publications about filter generators, but the generators are still not freely available or require a special license for commercial usage. From the company point of view, spending money on licenses is something that must be carefully considered, especially if the needs are just for simple filters. It is also not enough just to have the licenses available for the initial design period as a license would be required any time an engineering change is requested.

For all those reasons it was decided that it would still be better to program an implementation of our own. Instead of creating very generic VHDL the flow will utilize a scripting layer to increase the level of automation. For example, the script can generate the reduced amount of registers for a symmetrical half-band filter instead of using if-generate statements or a vast library of VHDL files to support each topology.

The programming language was chosen to be Python which is a modern language that a lot of engineers are comfortable with. It's also well supported in the computer environment of the company.

3.3 Verification

As mentioned before, all the filters must have a bit-exact Matlab model, which the RTL can be verified against. MathWorks has a product called HDL Verifier, which makes it easy to control the DUT (device under test) inputs and outputs from a Matlab script. It supports Modelsim, Questasim and Incisive as the simulation engine. Having the verification tool inside the Matlab environment reduces the need to move data from one program to another.

The benefits of using HDL Verifier are clear when verifying more complex blocks like multichannel filters. Creating the stimulus for each context and then switching the context from time to time is tedious to program with VHDL.

The code will also be run through Spyglass. The lint checks are the most interesting and will help to improve code quality.

3.4 The flow

The generator is placed in an automated flow illustrated in figure 3.2. The flow is built so that the design information has to be input only once and the information goes through the flow automatically. To make things easier for the user, all the script files are the same and used in the same way for both single-channel and multichannel filters.

The flow will be discussed in more detail later in this section, but in short, the floating point model of the filter is first converted to fixed point. Data about this fixed point model is saved into a Extensible Markup Language (XML) file, which is given to the generator. Using some files from its HDL library, the generator produces an HDL model. The HDL model is then verified against the original Matlab fixed-point model in HDL Verifier, which uses Modelsim as the simulation engine. The stimulus for the verification is generated inside the Matlab environment.

The flow is built around a Matlab script and a Python generator script, which is presented in the following chapters. The Matlab script covers the entire flow from the floating point model to verification. It also takes care of calling the generator script.

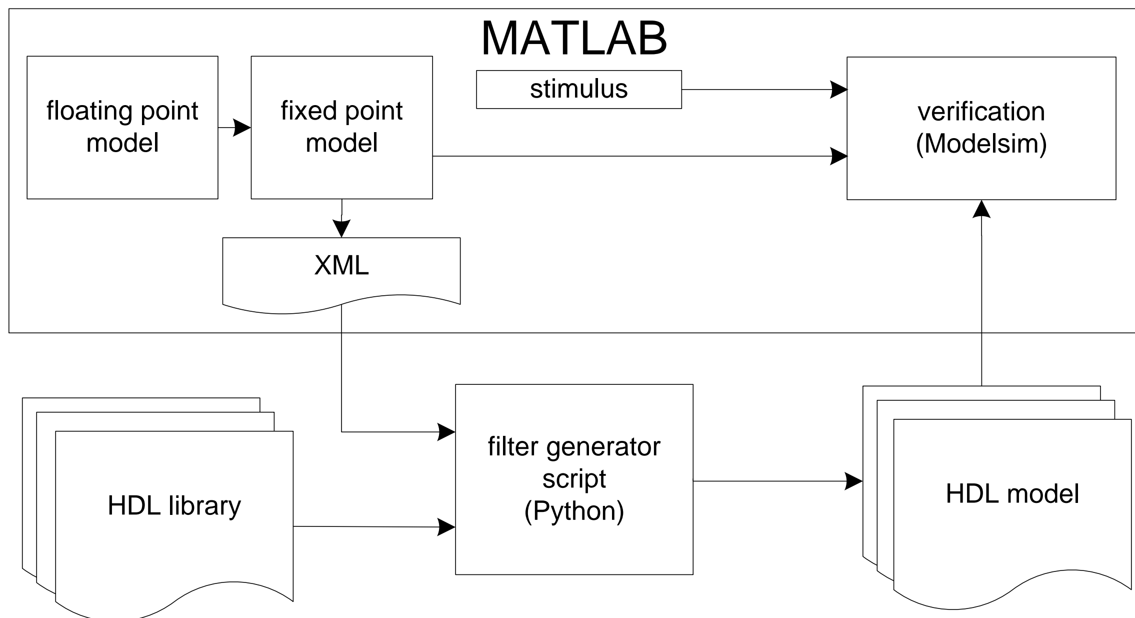

Figure 3.2 *The filter generation flow.*

The intended environment for the flow is one of Nokia's Unix development servers. Like with many other scripts in Nokia environment, the first step in using the flow is setting the environment variable `$MODULES_PATH` so that the generator can use absolute paths with ease. It is the parent of the directory `fir_gen`, which contains all the files related to the generator. Refer to figure 3.3 for the directory structure.

- $MODULES_PATH/fir_gen/ — root directory for the generator
  - designs/
    - MyDesign/ — subdirectory that is created for each user design
      - hdl/ — contains HDL files
      - sim/ — contains test bench, makefile and simulator files
  - fir_gen.py — filter generator script that gets called by Matlab
  - hdl_lib/
    - channel_filter/ — VHDL templates for multi-channel filters
    - common/ — VHDL templates for both filter types
    - single_fir/ — VHDL templates for single-channel filters
  - launcher.sh — launcher script, startpoint for the flow

Figure 3.3 *Directory structure used by the flow.*

All the rest of the startup steps are taken care of by a simple startup script called `launcher.sh`. The only command line parameter it takes is the name of the filter, and it is mandatory to specify it. The script loads Matlab, Modelsim and Python into

the environment, creates the directory structure, copies the Matlab script template to a subdirectory and opens it in Matlab editor.

After the editor has loaded, it is time to edit the Matlab script. There are a few properties that the user must change. They are listed in table 3.1.

Table 3.1 *User settings in the Matlab script.*

name	description
CHANNELS	Number of channels in filter (1–32). Value of one creates a single-channel filter, otherwise a multichannel filter is created.
COEFF_WIDTH	Width of the FIR coefficients in bits (4–32).
COEFF_FRACTION_WIDTH	Number of fraction bits in the coefficients (0–COEFF_WIDTH). Determined automatically for single-channel filters when the value is *auto*.
CONTEXT_SWITCH_PERIOD	Number of clock cycles per context switch period in a multichannel filter (4–64, only powers of two).
FILTER_NAME	VHDL entity name. Set by the launcher script.
INPUT_WIDTH	Width of the input samples in bits (4–32).
INPUT_FRACTION_WIDTH	Number of fraction bits in the input samples (0–INPUT_WIDTH).
OUTPUT_WIDTH	Width of the output samples in bits (4–32). Determined automatically from coefficient area if the value in this field is *auto*.
OUTPUT_FRACTION_WIDTH	Number of fraction bits in the output samples (0–OUTPUT_WIDTH).
SHOW_RESPONSE	Boolean. If set to 1, the frequency response of the fixed point and floating point designs are shown. Valid only for a single-channel filter.
STIMULUS_LENGTH	Number of stimulus input samples per channel. Must be a multiple of CONTEXT_SWITCH_PERIOD.

Algorithm developers have often defined the required data and coefficient widths. If that is not the case, the coefficient widths can be determined by using the Matlab function *minimizecoeffwl* [19].

3.4.1 From Matlab model to RTL

The filter can be designed in the Matlab environment by using all the design tools available. For example, to create a low-pass filter with 12 taps, a cut-off frequency of 10 MHz and a sampling frequency of 50 MHz, a suitable set of commands could be

```
d = fdesign.lowpass('N,Fc', 11, 10e6, 50e6);
hd = design(d, 'fir');
b_fp{1} = hd.numerator;
```

In a typical scenario the coefficients are given to the RTL designer without the need to touch the filter design tools at all. In that case the coefficients can simply be placed in the vector:

```
b_fp{1} = [-66 0 294 0 -905 0 2282 0 -905 0 294 0 -66] ./ 2^15;
```

For multichannel filters the different coefficient banks must be set in *b_fp* indices 1 — *CHANNELS*. After the coefficients have been specified, the script no longer needs user input. The floating point model is then quantized to the desired bit width using the Matlab fixed point toolkit:

```
for n=1:CARRIERS
  b_fi{n} = fi(b_fp{n}, 1, COEFF_WIDTH, COEFF_FRACTION_WIDTH);
end
```

If the filter is of single-channel type and *SHOW_RESPONSE* was set, the user is presented with the frequency response of the filter as shown in figure 3.4. This functionality can be used to observe the negative effects of quantization like reduced attenuation in the stop band.

At this stage the script stores the filter design into a XML file to be forwarded to the generator. See program 3.1 for a sample of the format. All the elements in the sample XML are mandatory, and no other elements are supported at the moment. However, some data may be unused in the generator, like the information about the context switch period when generating a single channel filter.

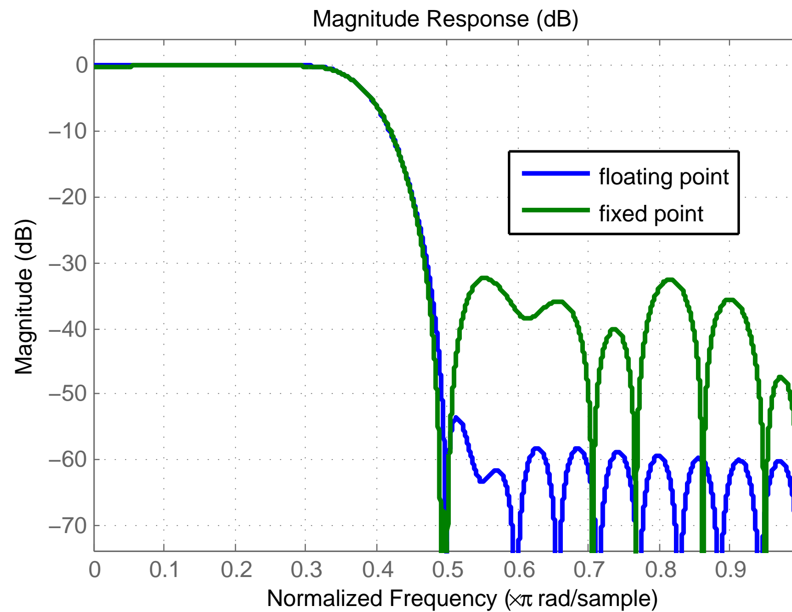

Figure 3.4 *Floating point and fixed-point implementations of a filter.*

```
<?xml version="1.0" encoding="utf-8"?>
<filter>
    <name>sg_filter_test</name>
    <fixed-point>
        <coefficient-set channel="0">-66 0 294 0 -905 0 2282 0 5032
          2282 0 -905 0 294 0 -66</coefficient-set>
        <settings>
            <coefficient-width>25</coefficient-width>
            <coefficient-fraction-width>24</coefficient-fraction-width>
            <input-data-width>24</input-data-width>
            <input-data-fraction-width>0</input-data-fraction-width>
            <output-data-width>24</output-data-width>
            <output-data-fraction-width>0</output-data-fraction-width>
            <context-switch-period>4</context-switch-period>
        </settings>
    </fixed-point>
</filter>
```

Program 3.1 *Example of the XML file created by the Matlab script.*

The generator is called by using Matlab's `unix` command and giving the path to the XML file as a parameter. The generator output is printed in the Matlab window. If all goes well, the RTL design of the filter will be created in the `hdl/` directory.

3.4.2 Verifying the RTL

The design is compiled using the makefile the generator has saved in the `sim/` directory. After that the simulation object has to be created and the simulator launched. The object and the launcher script have been initially created using the `cosimWizard` in HDL Verifier. In this flow the generator has taken care of copying them to the `sim/` folder, and all that is required is calling them in Matlab:

```
delete('vsim.wlf');
fir_sysobject = hdlcosim_FIR_CHANNEL_FILTER;
launch_hdl_simulator_FIR_CHANNEL_FILTER;
while exist('vsim.wlf', 'file') ~= 2
  pause(1);
end
```

There is no handshaking in the connection between Matlab and the simulator. Therefore the script waits for the simulator files to appear before proceeding so that it is ready to accept commands from Matlab.

For each channel an input sequence of length `STIMULUS_LENGTH` is created. It is fixed point random data which has the properties set with `INPUT_DATA_WIDTH` and `INPUT_DATA_FRACTION_WIDTH`. Matlab `filter` function is called to create the reference data to which the RTL output is compared:

```
for n = 1:CARRIERS
  fi_result{n} = filter(b_fi{n}, 1, x_fi{n});
end
```

See program 3.2 for the test bench main loop. The simulation is continued for as long as there are channels with input data. Then a channel is chosen at random, and its data is fed to the filter under test for one context switch period or more. For single channel filters there is only one channel with input data, and the context is therefore never switched.

In some possible use cases any number of subsequent samples may be invalid at any given point of time. To mimic this behaviour, the test bench randomly marks some samples as invalid.

The randomizations help to ensure correct functionality in all use cases. Special cases include having the same channel for two multiple context switch periods, or having another channel for one period in between periods of one channel. The latter

situation requires a special case in RTL because the context memory is read before it is written.

The `step` function, which advances the simulation, is called on line 20. It takes the inputs to the filter as a parameter and returns the outputs directly to Matlab variables.

```
while any(carrier_pointer)
  % Randomly choose the next carrier
  while true
    context = randi(CHANNELS, 1);
    if carrier_pointer(context) > 0
      break;
    end
  end

  n = 1;
  while n <= CONTEXT_PERIOD
    % Set 1 in 20 samples as invalid
    valid_in = (randi(20,1,1) ~= 7);

    % Calculate array index
    sample_n = STIMULUS_LENGTH+2*CONTEXT_PERIOD + ...
      1-carrier_pointer(context)

    % Advance simulation
    [outdata,outvalid,outchannel] = step(fir_sysobject, ...
      fi(x_fi{context}(sample_n), 1, ...
        INPUT_DATA_WIDTH, INPUT_DATA_FRACTION_WIDTH), ...
      fi(context-1, 0, log2(CHANNELS), 0), ...
      fi(valid_in, 0, 1, 0));

    % Update data pointer
    if valid_in == 1
      carrier_pointer(context) = carrier_pointer(context) - 1;
      n = n + 1;
    end

    % Append the output
    if outvalid == 1
      outchannel = outchannel.double;
      outbuf{outchannel+1} = [outbuf{outchannel+1} outdata];
    end
  end
end
```

Program 3.2 *Test bench main loop.*

The final step is comparing the reference output in `fi_result` to the DUT output in `outbuf`. The last line of output from the script tells whether the RTL matches the fixed point model.

4. FILTER GENERATOR

In the previous chapter the requirements were collected and the flow was introduced. This chapter describes the script used to generate the VHDL. First a series of test syntheses is performed to get an idea of the optimization level required, then these results are utilized to build a filter by hand. Finally the generation is automated and a series of test syntheses is performed on the generated filters.

4.1 Multiplier block test syntheses

For the multiplier block test syntheses a reference filter was selected from a current project. It is a 71-tap, folded half-band filter. All the different candidates were compared to this in terms of area. Code readability was evaluated in weekly meetings. Transposed filter was chosen as the basis of all the different variants. Its structure is very simple, yet it has interesting optimization possibilities like using reduced adder graph. Utilizing coefficient symmetry is just a matter of connecting the multiplier outputs to two adders. It maps well to both FPGA and ASIC technologies with easily controllable timing paths.

To shed some light on an appropriate implementation of the multiplier block, four different candidates were programmed. The resulting filters were then synthesized using Synopsys DC Ultra with a wireload model.

In the simplest multiplier block the multiplications were implied with a multiplication sign. This was the most readable option and therefore it was interesting to see how well the synthesis tool optimizes it. To give an idea of the implementation, program 4.1 presents how a few of the multiplications were done. The `INPUT_TIMES` vectors had been optimized to minimum width as determined by the input width and the multiplier value.

```
  INPUT_TIMES_66 <= DATA_IN * 66;
2 INPUT_TIMES_294 <= DATA_IN * 294;
  INPUT_TIMES_905 <= DATA_IN * 905;
```

Program 4.1 *Multipliers using multiplication sign (design I).*

A bit more advanced approach was to perform the CSD coding in Python and specify the bit shifts, additions and subtractions in VHDL. Program 4.2 clarifies this method.

```
1 INPUT_TIMES_66 <= DATA_IN*64 + DATA_IN*2;
  INPUT_TIMES_294 <= DATA_IN*256 + DATA_IN*32 + DATA_IN*4 +
3                DATA_IN*2;
  INPUT_TIMES_905 <= DATA_IN*1024 - DATA_IN*128 + DATA_IN*8 +
5                INPUT_TIMES_1;
  INPUT_TIMES_2282 <= DATA_IN*2048 + DATA_IN*256 - DATA_IN*32 +
7                DATA_IN*8 + DATA_IN*2;
  INPUT_TIMES_5032 <= DATA_IN*4096 + DATA_IN*1024 - DATA_IN*128 +
9                DATA_IN*32 + DATA_IN*8;
```

Program 4.2 *CSD multiplications (design II).*

A variant of the CSD method used a binary tree of adders like shown in figure 4.1 instead of implying the operation with an addition sign. The purpose of this was to be able to break the long combinational paths that may be synthesized with big adder structures. The coefficients had at most nine bit shifts to be summed so the binary trees were four levels deep.

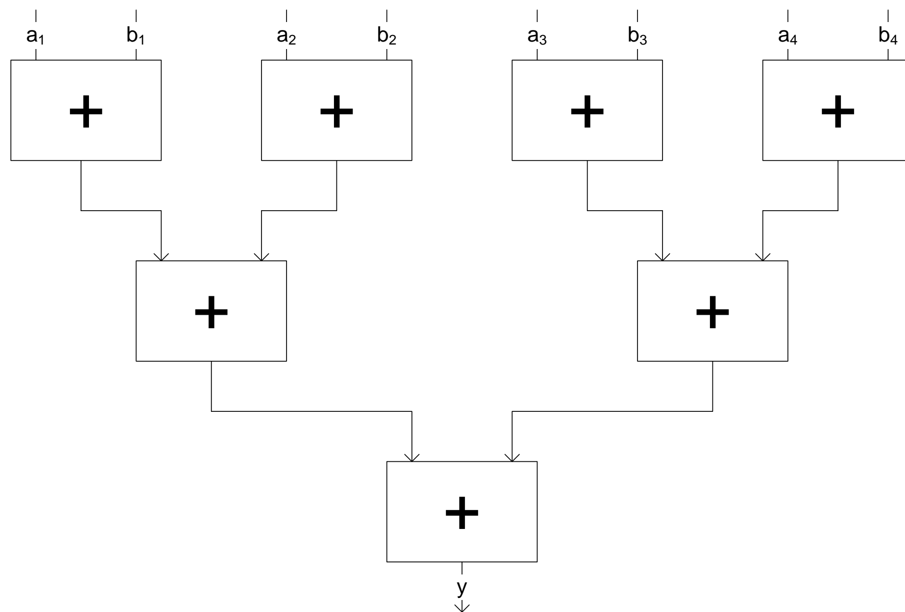

Figure 4.1 *Eight input binary tree adder (design III).*

Fourth implementation used a reduced adder graph created by SPIRAL multiplier block generator [15]. The generator output was a file with information about the required adders, which was then parsed in Python to generate the VHDL. A part of the VHDL can be seen in listing 4.3.

```
1 RAG_COEFF_33 <= DATA_IN*32 + DATA_IN;
  RAG_COEFF_37 <= DATA_IN*4 + RAG_COEFF_33;
3 RAG_COEFF_147 <= RAG_COEFF_37 * 4 - DATA_IN;
  RAG_COEFF_629 <= RAG_COEFF_37 * 16 + RAG_COEFF_37;
5 RAG_COEFF_1141 <= DATA_IN*512 + RAG_COEFF_629;
  RAG_COEFF_3657 <= RAG_COEFF_629*4 + RAG_COEFF_1141;
7 RAG_COEFF_3620 <= RAG_COEFF_3657 - RAG_COEFF_37;

9 INPUT_TIMES_66  <= RAG_COEFF_33 * 2;
  INPUT_TIMES_294 <= RAG_COEFF_147 * 2;
11 INPUT_TIMES_905 <= RAG_COEFF_3620 / 4;
  INPUT_TIMES_2282 <= RAG_COEFF_1141 * 2;
13 INPUT_TIMES_5032 <= RAG_COEFF_629 * 8;
```

Program 4.3 *RAG multiplications (design IV).*

All the different multiplier block implementations used the same test bench and the same delay line structure. There was one register stage after the multiplier in each of the multipliers. The designs had no internal pipeline registers with the exception of the CSD binary tree variant, which had one extra register stage.

The synthesis results are collected to table 4.1. Multiplier block name is abbreviated in the first column. The second column states the amount of non-combinational logic in the design whereas the area of combinational logic is listed in the third one. Fourth column is the total area reported by the synthesis tool, and calculated in the fourth column is the relative area compared to the reference.

Table 4.1 *Synthesis results (μm^2)*

	design	non-comb.	comb.	area	relative
II	CSD	15143	11639	32674	-12 %
I	MULT	16218	13268	36240	-3 %
	REF	18919	11456	37194	0 %
IV	RAG	23059	14070	45273	22 %
III	CSD_BINTREE	46718	11351	69882	88 %

There turned out to be some benefit in specifying the CSD algorithm in VHDL. Synopsys documentation hints that the compiler does it automatically with constant coefficients [24], but for some reason doing it by hand reduced the area by 12%. Trying to implement the additions by a pipelined adder tree was a bad idea - it

increased the area of the registers dramatically. Moreover, adding extra register stages inhibits some compiler optimizations.

Surprisingly the reduced adder graph implementation had no problems with achieving timing closure even with eight additions per register stage. However, in terms of area it performed surprisingly badly given its level of sophistication. Possibly the structure is too irregular to be efficiently optimized by the compiler.

While there was only one set of coefficients tried, it was deemed that the results were conclusive enough to give a clear guideline on how to implement the multiplication. Implying the operation with a multiplication sign is the most readable option and the compiler does a good job optimizing it.

4.2 Building a filter by hand

The results of the trial syntheses were studied to create a hand-written filter which was used as the basis of the generation. To support different number formats, a decision was made to utilize the VHDL 2008 fixed point package [2]. It also has a compatibility package for VHDL 1993, which was required as none of the tools supported the new features, with the exception of Modelsim. Even with the compatibility package some issues had to be solved in collaboration with the tool vendors.

Using the fixed point package frees the programmer from the burden of tracking the place of the binary point manually. The bounds of the results of operations can be determined by using the `sfixed_high` and `sfixed_low` functions. They take the operand bounds and the type of operation as the parameters and return the output bounds.[2]

These calculations are done in a package file. This makes it possible to use the calculated output width in the entity. The package file also contains the filter coefficients and the bit width of each coefficient. By stating the coefficient width explicitly it is possible to force the synthesis tool to use a multiplier that is of exactly the right size. Whereas the smallest multiplier is preferred in ASIC designs, in FPGA designs it may be desired to use a multiplier size that is supported by the embedded multiplier blocks.

It was decided that generics are not used as the filter is usually designed to fit in an existing processing pipeline with frozen parameters. Therefore the parameters are rarely changed in the integration phase. Coefficient width, number of taps, coefficient area, filter topology et cetera are automatically calculated in the generator script. To get an idea of what's included in the package file, see program 4.4.

```
-- Input data width
constant W_IN_C          : positive := 24;
-- Place of binary point in the input
constant BP_IN_C         : integer  := 0;
-- Coefficient width
constant W_COEFF_C       : positive := 25;
-- Coefficient binary point location
constant BP_COEFF_C      : integer  := -24;
-- Number of multiplications needed
constant N_UNIQ_COEFF_C : positive := 19;

-- Coefficient area is calculated in Python script
-- to be 28079036 >> -24, requiring 26 bits
constant COEFF_AREA_HIGH_C  : integer := 26 + BP_COEFF_C - 1;
constant COEFF_AREA_LOW_C   : integer := BP_COEFF_C;
constant DATA_IN_HIGH_C     : integer := W_IN_C + BP_IN_C - 1;
constant DATA_IN_LOW_C      : integer := BP_IN_C;

constant ADDER_HIGH_C       : integer := sfixed_high(
        DATA_IN_HIGH_C, DATA_IN_LOW_C, '*',
        COEFF_AREA_HIGH_C, COEFF_AREA_LOW_C);
constant ADDER_LOW_C        : integer := sfixed_low(
        DATA_IN_HIGH_C, DATA_IN_LOW_C, '*',
        COEFF_AREA_HIGH_C, COEFF_AREA_LOW_C);

-- Constant coefficients and their widths
constant COEFF        : T_CONST_COEFF   := (
        -66,294,-905,2282,-5032,10060,-18633,32442,-53683,
        85171,-130580,194987,-286134,417594,-617599,959358,
        -1712825,5317569,8388608);
constant COEFF_LEN    : T_CONST_LENGTH := (
        8,10,11,13,14,15,16,16,17,18,18,19,20,20,21,21,22,24,25);
```

Program 4.4 *An incomplete example of a package file.*

Even though some of the existing filter implementations round the multiplication results before summing them, the generator will always use full precision arithmetic. It was done in order to guarantee bit exactness between Matlab and RTL - modeling the intermediate roundings in Matlab would be more complex.

Another issue to ensure compatibility was the rounding and saturation behaviour. It has been set to *round nearest and saturate* in both Matlab and the filter RTL. The VHDL-2008 fixed point package provides the rounding routines.

In a single FIR filter there is only one other file besides the package file. That file contains the fimplementation of the filter. The functionality is divided to three

processes. All the processes require the ENABLE input to be high in order to process any data. It will also allow the filter to process data that has been decimated by toggling the enable signal. It also enables automatic clock gating to save power in situations where there is no meaningful data to be processed. In normal operation a lot of bits will change at each clock cycle, consuming a lot of power.

The first process registers the input and converts it to fixed point type. The scalb function is used to perform a bit shift. See program 4.5.

```
P_REGISTER_INPUT: process (CK, XR)
begin
  if(XR = '0') then
    DATA_IN_R <= (others => '0');
  elsif(CK'event and CK='1') then
    if(ENABLE = '1') then
      DATA_IN_R <= scalb(to_sfixed(DATA_IN, W_IN_G-1, 0), BP_IN_C);
    end if;
  end if;
end process P_REGISTER_INPUT;
```

Program 4.5 *First process: registering the input.*

Program 4.6 represents the second process which performs the multiplications. As the coefficients are given as constants in the package file, the synthesis results will be well-optimized. The multiplier outputs are registered to decrease the length of the combinational path.

```
P_INPUT_MULTIPLES: process(CK, XR)
begin
  if(XR='0') then
    MULTIPL_OUT <= (others => (others => '0'));
  elsif(CK'event and CK='1') then
    if(ENABLE = '1') then
      for I in 0 to MULTIPL_OUT'length - 1 loop
        MULTIPL_OUT(I) <= resize(DATA_IN_R * scalb(
            to_sfixed(COEFF(I), COEFF_LEN(I)-1, 0), BP_COEFF_C),
          MULTIPLIER_HIGH_C, MULTIPLIER_LOW_C);
      end loop;
    end if;
  end if;
end process P_INPUT_MULTIPLES;
```

Program 4.6 *Second process: performing the multiplications.*

The third process advances the delay line and sums the multiplier outputs into it. The implementation is different for each filter type as some symmetry-based optimizations are done based on the filter index. For an example see program 4.7, which is used for symmetrical coefficients. As the bounds of `SHIFTREGISTER` have been calculated in the package file to be large enough not to overflow, the resize functions use truncation and no overflow protection to minimize the amount of logic.

```
P_SHIFTREGISTER: process (CK, XR)
begin
  if(XR = '0') then
    SHIFTREGISTER <= (others => (others => '0'));
    OUTPUT_R      <= (others => '0');
  elsif(CK'event and CK='1') then
    if(ENABLE = '1') then
      SHIFTREGISTER(0) <= MULTIPL_OUT(0);
      for I in 1 to L_C-1 loop
        --First half of the filter
        if (I < L_C / 2) then
          SHIFTREGISTER(I) <= to_slv(
            resize(
              arg            => SHIFTREGISTER(I-1) +
                                MULTIPL_OUT(I),
              left_index     => ADDER_HIGH_C,
              right_index    => ADDER_LOW_C,
              round_style    => fixed_truncate,
              overflow_style => fixed_wrap));
        --Second half of the filter
        else
          SHIFTREGISTER(I) <= to_slv(
            resize(
              arg            => PREVIOUS_SR_STAGE +
                                MULTIPL_OUT(L_C - 1 - I),
              left_index     => ADDER_HIGH_C,
              right_index    => ADDER_LOW_C,
              round_style    => fixed_truncate,
              overflow_style => fixed_wrap));
        end if;
      end loop;
      OUTPUT_R <= resize(SHIFTREGISTER(SHIFTREGISTER'high),
                    ADDER_HIGH_C, 0);
    end if;
  end if;
end process P_SHIFTREGISTER;
```

Program 4.7 *Third process: summing the multiplier outputs into the delay chain.*

4.3 Automating the generation

The basis of the generator are two template files as described before. Information about the coefficients will be written into the package file. The most optimal `P_SHIFTREGISTER` process in the implementation file will also be chosen among the candidates based on the properties of the coefficients. There are different shift register processes for non-symmetric, symmetric, antisymmetric and half-band filters.

The system works by having special hooks in the template files, which will be replaced with the desired contents. For all the hooks in the single channel FIR template file, see table 4.2. The first column in the table stands for the name of the hook and the second one is the type of data that is inserted at that point. The third column is a description, which clarifies the function of each hook.

All this processing is done in the Python script. The script gets the coefficients, input and coefficient bit widths and their binary point locations from XML generated by the Matlab script. The data interface between these two scripts is described in more detail in Section 3.4.

First of all, the scripts checks the coefficients for symmetry. For a half-band filter, every coefficient with an odd index must be zero with the exception of the middlemost tap. The number of taps incremented by one must also be divisible with four as described in Section 2.2.2. Symmetrical and anti-symmetrical properties of the coefficients are checked by comparing the lists with their reversed counterparts. If a filter is not a half-band filter and has no symmetry or anti-symmetry, it is deemed a non-symmetrical filter, for which no optimizations can be made.

The script also calculates the coefficient area of the coefficients using formula 2.5. The number of taps is the same as the number of coefficients given to the generator. The amount of bits required to represent these numbers is also calculated.

Depending on the type of filter that was detected, the script determines the number of multipliers that is needed. For example, a filter with symmetrical coefficients requires less multipliers than a filter with non-symmetrical coefficients even if they have the same number of taps. The coefficients for the multipliers are saved into an array. In case of half-band filters the zeroes are not saved.

The last step of generation is reading the file containing the correct `P_SHIFTREGISTER` implementation to the memory. After that it is just a matter of replacing the hooks in the template files and writing everything to the files.

Table 4.2 Hooks in the simple filter template

name	type	description
COEFFICIENT_WIDTH	integer	width of largest coefficient in bits
COEFFICIENT_BINARY_POINT	integer	location of binary point in coefficients
COEFFICIENT_AREA	integer	area of coeffiecients
CONSTANT_COEFFICIENTS	string	coefficients in VHDL array format
CONSTANT_LENGTHS	string	coefficient widths in VHDL array format
ENTITY_NAME	string	entity name
FILTER_SPECIFIC_CONTENT	code	implementation-specific constants in package file
INPUT_DATA_BINARY_POINT	integer	location of binary point in input data
INPUT_DATA_WIDTH	integer	width of filter data input in bits
LATENCY	integer	latency of the filter in cycles
LOG2_COEFFICIENT_AREA	integer	bit width of coefficient area
NUMBER_OF_TAPS	integer	number of taps
NUM_UNIQUE_COEFFICIENTS	integer	number of multipliers required
OUTPUT_DATA_BINARY_POINT	integer	location of binary point in output data
OUTPUT_DATA_FRACTION_WIDTH	integer	fraction width of output data (additive inverse of binary point)
OUTPUT_DATA_WIDTH	integer	width of filter data output in bits
PACKAGE_NAME	string	name of the VHDL package file
REGISTER_ASSIGNMENTS	code	contents of process `P_SHIFTREGISTER`

4.4 Results

Test bench has 100% statement and branch coverage. The only unverified signal transition in the reports is toggling the reset from '1' to '0'.

To further characterize the properties of the filters, some more syntheses were run. A symmetric FIR filter with 16-bit wide inputs, outputs and coefficients was generated with 4, 16, 64 and 128 taps. Both FPGA and ASIC performance was evaluated.

4.4.1 FPGA synthesis

The FPGA syntheses were run using Quartus II v14.0. The destination chip was selected to be Cyclone IV EP4CE115. For reference, the filter designs were implemented using Altera's FIR Compiler II as well. The Altera designs include the Avalon streaming interface, which is automatically bundled with the filter cores. The results have been collected to table 4.3, and they cover the maximum operating frequency, the number of logic elements (LEs), used 9-bit embedded multipliers and the size of the VHDL file containing the filter core.

Table 4.3 *FPGA synthesis results*

	taps	f_{max} (MHz)	LEs	multipliers	file size (kB)
The generator	4	151	191	4	5
	16	126	603	16	5
	64	134	2322	56	5
	128	140	4765	94	5
FIR Compiler II	4	163	314	4	15
	16	151	974	6	48
	64	154	3934	18	185
	128	168	4678	34	375

With all tap counts the FIR Compiler II designs fare better in terms of maximum operating frequency. This can be seen in figure 4.2. The clock frequency seems to be independent of the tap count with both generators.

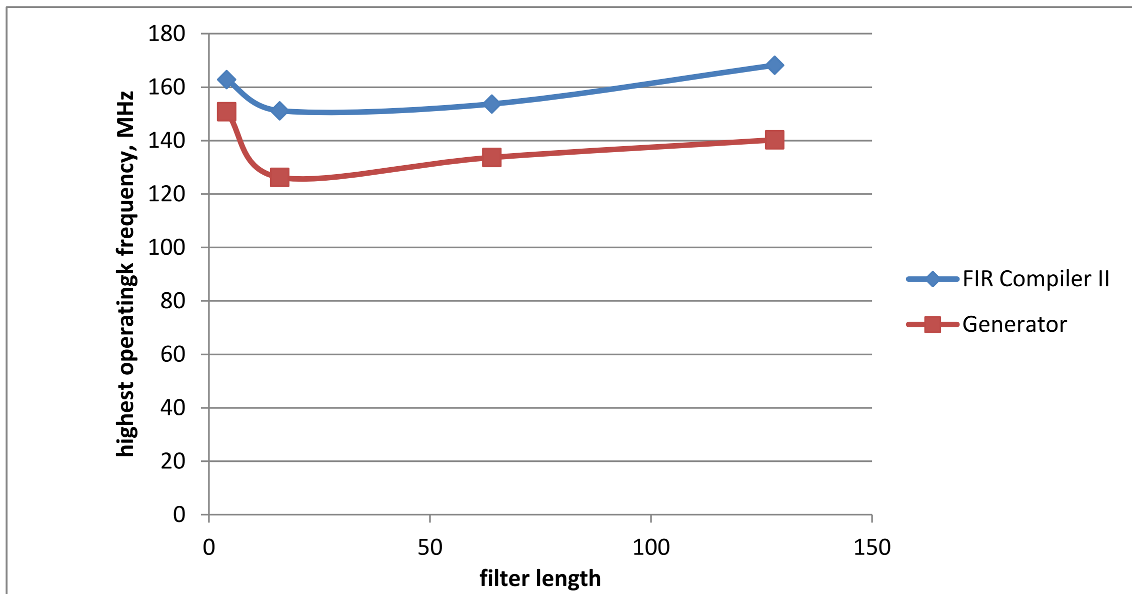

Figure 4.2 *Maximum operating frequency on an FPGA.*

With the generator the number of logic elements used seems to increase proportionally to the filter length. The amount is equal to or lower than the what FIR Compiler II designs require as can be seen in 4.3.

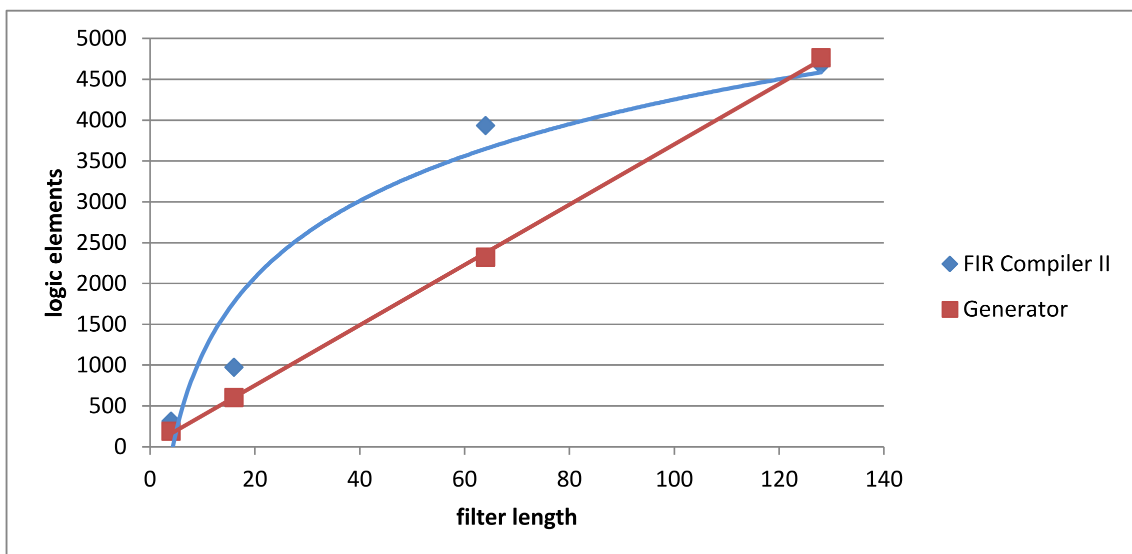

Figure 4.3 *Number of logic elements used on an FPGA.*

However, FIR Compiler II is able to make better use of the embedded 9-bit multipliers. With both tools the number of block required increases in linearly with the filter length. See figure 4.4.

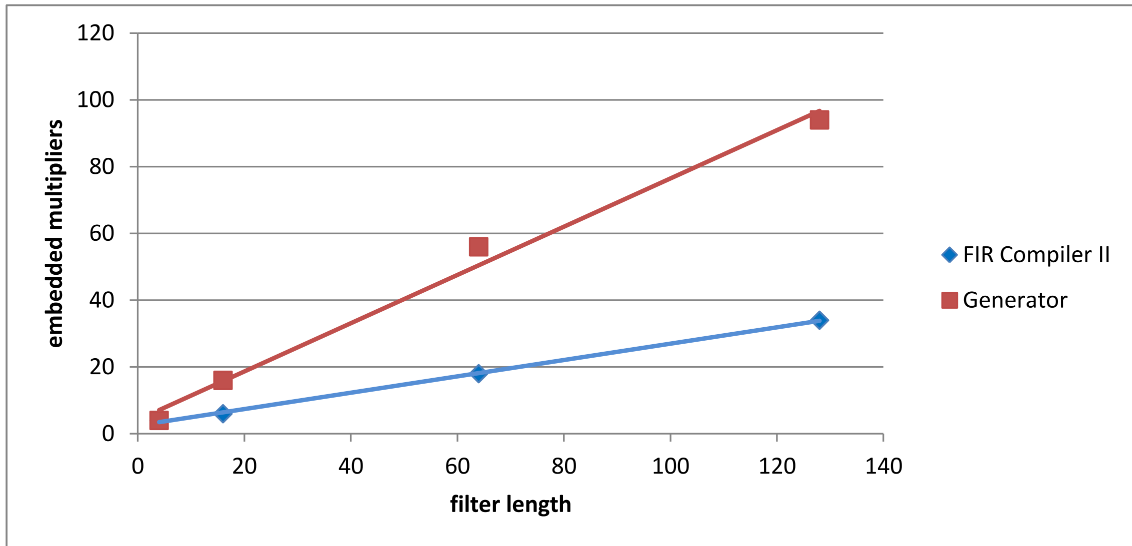

Figure 4.4 *Number of embedded multiplies used on an FPGA.*

According to the results, the FIR Compiler II is able to generate designs that are better optimized to the platform. However, as the file sizes suggest, the designs are much more complex.

The filters created by the generator seem to scale well with the filter length. There is no significant change in the maximum clock frequency, and the amount of resources increases proportionally to the filter length.

4.4.2 ASIC synthesis

The same designs were synthesized using the same synthesis environment as in Section 4.1. All designs achieved timing closure with 500 MHz. The resulting silicon areas are collected in table 4.4.

Table 4.4 *ASIC synthesis results*

taps	area (μm^2)
4	1772
16	8966
64	27587
128	54578

These results are similar to the ones received in FPGA synthesis: the silicon area grows linearly with the filter length as shown in figure 4.5. Moreover, there is no significant performance penalty that comes with the increasing tap count.

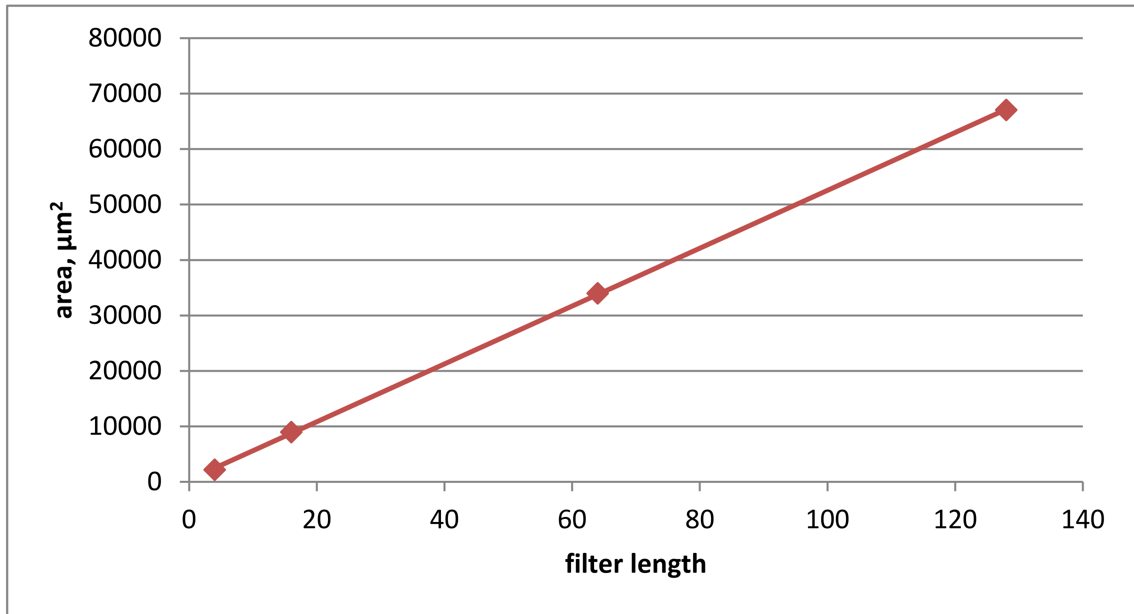

Figure 4.5 *Silicon area as a function of the filter length.*

5. EXTENSION TO MULTICHANNEL FILTERS

Chapter 4 dealt with automating the generation of simple FIR filters with a single coefficient bank. As the support for multichannel filters was one of the initial requirements, the generator was then expanded to support those kinds of filters as well. The main idea of multichannel filters was introduced in Section 2.2.5. While the core of the filter is essentially the same as with a single channel filter, a lot of control logic must be added in order to move data from one place to another.

The Nokia multichannel filter explained in 2.2.5 was used as the basis for the implementation [8]. Particularly the method of saving the context and loading it back was replicated very faithfully. However, this was a complete rewriting effort with some key differences: the previous implementation used a folded topology with a fixed filter length (64 taps) and a fixed context switch period (8 cycles). In this implementation the filter has a transposed structure with an adjustable context switch period and adjustable filter lengths. The previous version also loaded its coefficients from memory, whereas the newer version has constant coefficients in the banks.

5.1 Functionality

The multichannel filter is illustrated in figure 5.1. The design contains three distinct parts: the FIR core, the context memories and the state machine that controls the data flow. The input data samples and the related channel ID are fed into the input, and filtered data appears at the output. Although not drawn in the figure, the module also has a channel output, which indicates which channel the output data belongs to.

A few changes had to be done to the FIR core compared to the single FIR structure presented in figure 2.2 and Chapter 4. First of all, as there are multiple coefficient banks, there must be more generic, non-optimized multipliers or a larger amount of well-optimized, constant coefficient multipliers. Choosing the actual implementation is left up to the synthesis tool, but for simple implementations it is most likely more efficient to multiplex the outputs of multiple multipliers which use bit shifting techniques.

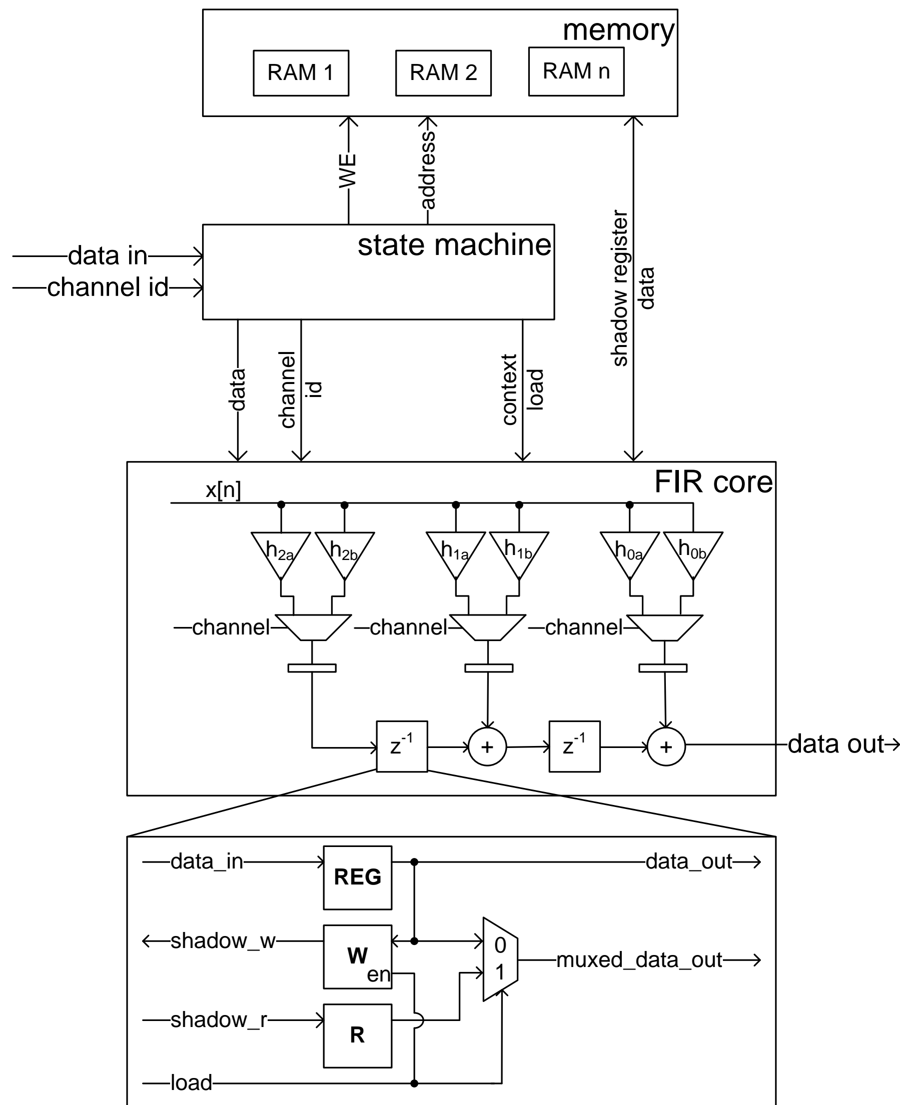

Figure 5.1 *Structure of the multichannel filter.*

As can be seen in the bottom part of figure 5.1, the delay element has also changed. Instead of being a simple register stage, there is now a total of three registers. The register marked `REG` is functionally identical to the register stage in a basic, transposed filter.

The other two registers are related to the context switching. The `W` register holds the data of `REG` after the context has been changed and while the old context is being written to the memory. Likewise, the new context is loaded to the `R` register so that it is ready to be loaded to the `REG` register further down in the chain when the `load` signal is asserted. The change is done seamlessly so that data is processed all the time without any wasted clock cycles.

The delay elements are connected in the FIR core so that the `data_in` of each stage is connected to the `muxed_data_out` of the previous stage. The signal `data_out` of the last delay element is used as the output of the whole filter. A separate output is needed so that the output value will not be written over by the new context.

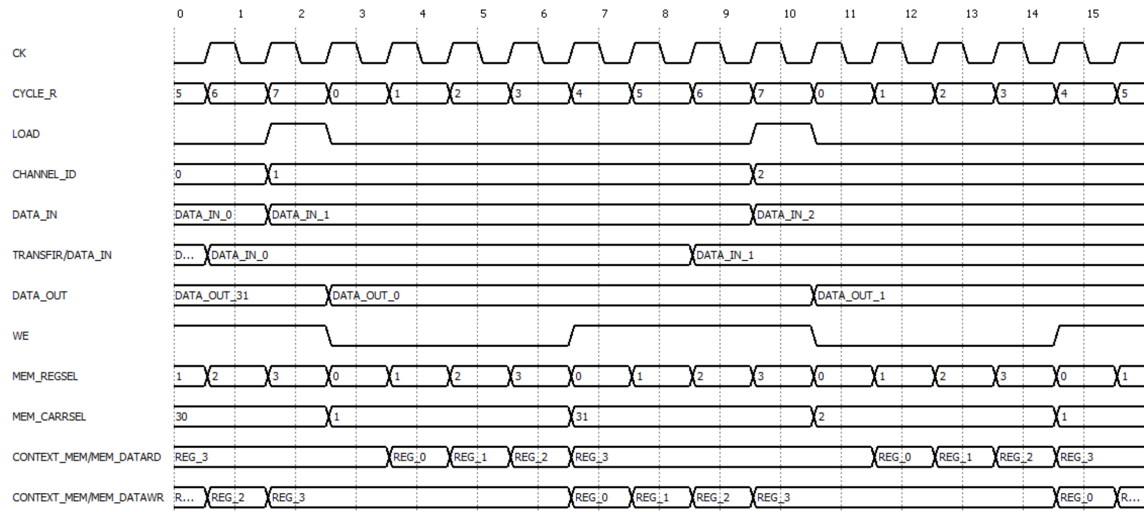

Figure 5.2 *Timing diagram for the multichannel filter.*

The context memories and the state machine are easiest explained with the help of the timing diagram in figure 5.2. The key parameter in the timing is the number of subsequent cycles that must be related to the same channel, or the context switch period.

The context switch period may be changed freely from 4 to 64 in powers of two. In the timing diagram of figure 5.2 the context switch period has been chosen to be eight clock cycles. In that scenario writing the previous context to memory and loading the next one may therefore take at most eight clock cycles. When using single port memories, there are four clock cycles for reading and four for writing. Consequently, four delay elements are using the same memory, and the number of memories is equal to the number of taps divided by four. This has been illustrated in figure 5.3.

By making the period longer the designer can reduce the memory bandwidth and the number of memories that is instantiated. This phenomenon is described in table 5.1. Of course, making the period too long may increase the latency of the system as the system is not able to change the channel at a short notice.

During the first half of the period the new context is loaded from memory, and the old context is written in the second half. In the middle of the period there is a time when the output of the memory access done in the previous clock cycle is read

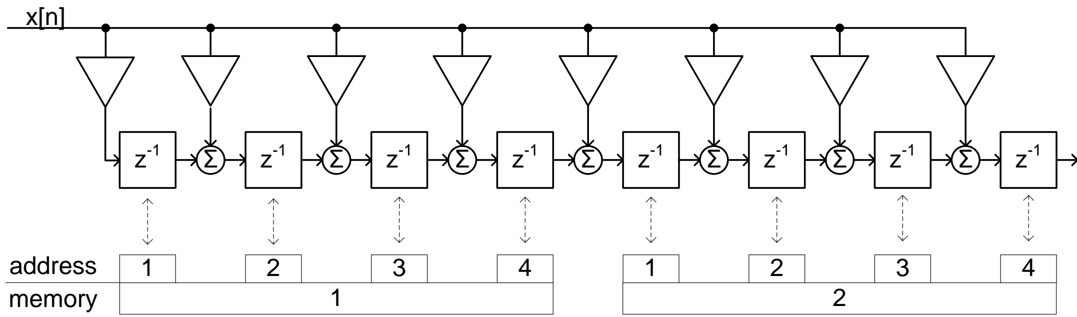

Figure 5.3 *Delay line register stages sharing the same memories.*

simultaneously with writing the memory. The memory addresses are derived from the channel id and the index of the delay element inside the memory block (0-3) like shown in table 5.2.

Table 5.1 *Effects of choosing the context switch period.*

period	number of memories	relative bandwidth
4	number of taps / 2	1
8	number of taps / 4	0.5
16	number of taps / 8	0.25
32	number of taps / 16	0.125
64	number of taps / 32	0.0625

In order to be able to load the right context before processing data, it is necessary to buffer the input data for a period of time that is one clock cycle shorter than one complete context switch period. The data is fed to the filter one clock cycle before the `load` signal is asserted so that the multiplication result stored in the register is ready when `load` is asserted. The result can then be summed with the new context in the `muxed_data_out` signal during the next clock cycle, and the first output related to the new context is then available at the first cycle of the new context switch period.

Table 5.2 *Memory addresses in context memory.*

address	content
0x7f	channel 31, register 3
0x7e	channel 31, register 2
...	...
0x01	channel 0, register 1
0x00	channel 0, register 0

5.2 Implementation

All the parameters that are related to context switching, like the number of channels and the length of the context switching period, were added to the package file introduced in program 4.4. They are the only new implementation specific things, and therefore the role of the generator is just to write the FIR core and package files just like with single filters. The additional VHDL files that contain the state machine, the delay elements and the memory wrapper are copied to the destination directory with only the package name changed in them.

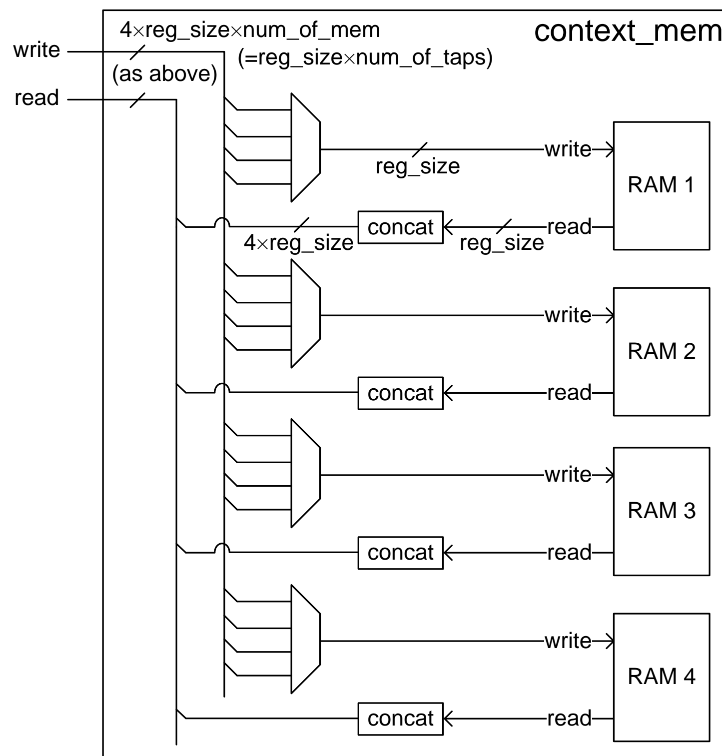

Figure 5.4 *Dividing the delay line to different memories. Context switch period is 8 cycles.*

To avoid problems with verification and synthesis tools, care was taken to use only `std_logic` and `std_logic_vector` signals. Using VHDL-2008 fixed point types and nested `std_logic_vector`s would have made things a lot easier especially at the memory interface where large vectors are sliced to smaller ones, which are then written to memory.

The memory connections can be seen in figure 5.4. The two buses coming in from the left are conneted directly to the shadow registers. The write bus is simply all the `W` registers of figure 5.1 in one big vector. Similarly, the read bus is connected to all the inputs of `R` registers of the same picture.

In the picture the size of the delay line registers is referred to as *reg_ size*. The total number of memories is indicated by *num_ of_ mem*, and *num_ of_ taps* is equal to the number of taps. The width of both write and read buses is the same — the number of taps multiplied by the width of a single accumulator register. If the number of taps is not a multiple of the number of delay stages that have to share the same memory, there will be some wasted space in one of the memories. Unused memory regions are not visible in the interface towards the shadow memories.

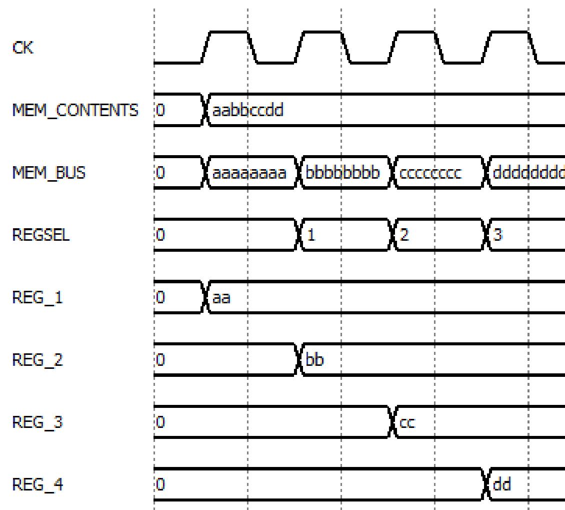

Figure 5.5 *Simplified example of the data flow from memory to the shadow registers.*

The write bus is sliced so that the parts that belong to the same memory are grouped together. The resulting signal is then further sliced to the signals representing the different delay stages inside the same memory. These signals are then connected to a multiplexer which connects them to the memory write bus. The multiplexer is controlled by the state machine, as is the write enable signal and the address bus. This is illustrated in figure 5.4.

The read signal is of size *reg_ size* when it comes from the memory read bus. It represents the contents of a single delay line register stage. It is then concatenated with itself so that the memory read bus output is connected to the inputs of all shadow read registers, which are connected to the same memory. In figure 5.4 the resulting output width is 4 times *reg_ size*, because four delay line stages share the same memory.

The state machine then toggles the appropriate shadow read register read enable signals to ensure that the new value is written to the register only when it is what has been read from the memory. For an example, see figure 5.5.

5.3 Results

The automatically generated test bench has 100% statement and branch coverage. To make the RTL synthesizable, a few non-achievable conditions had to be added to the code. Therefore the focused expression coverage (FEC) reaches only 99.0%. During tests the toggle coverage was above 99%.

To characterize the silicon area, 16- and 32-channel filters were synthesized and compared to a single channel filter. The filters had 27 bits wide inputs, outputs and coefficients. There were four different sets of coefficients, each with 64 coefficients. All filters achieved timing closure at 500 MHz. The results have been collected in table 5.3.

Table 5.3 *Multichannel filter synthesis results. Areas in* mm^2.

design	combinational	non-comb.	memory area	total
single filter	0.025	0.014	0	0.048
channel filter, 16 ch.	0.076	0.037	0.088	0.227
channel filter, 32 ch.	0.076	0.037	0.102	0.241

The results support the claim made in Section 2.2.5. Multichannel filters are more efficient when the number of channels increases. The 32-channel filter takes only five times the area of the single channel filter.

The multichannel filters have roughly three times as many registers as there are in a single channel filter, reflecting the addition of read and write shadow registers. With an increasing number of channels only the memory size increases. The overall savings will be even greater if the coefficients would not be constants. In those cases the filter would also reuse the costly multipliers.

6. CONCLUSIONS

This Thesis presented a flow for automated generation and verification of RTL designs for most commonly used FIR filters. According to some experienced designers it takes approximately two days to design and verify a filter. This flow reduces the time to a few minutes. A typical DFE project has dozens of filters, so the savings at project level are in the order of man-months. Therefore this flow will be introduced to the DFE team in the near future.

Besides the savings in design and verification time, the flow brings with it all the benefits of IP based development. The design is guaranteed to achieve timing closure, reducing the amount of work required from the backend team. In the documentation phase less effort needs to be put into describing the design as the filter implementation is not unique. It is very probable that the flow will save enough time to justify the four months it took in studies, development and documentation.

During this Thesis the flow has been built up to a point where it fulfills the original requirements. They were achieved without depending on any high level synthesis tools, keeping complexity and license costs low. The implementation is generic enough to be synthesized to both ASIC and FPGA platforms. It supports single and multichannel filters with adjustable data widths and freely choosable filter lengths and coefficients. The resulting silicon area is about the same as with existing, handwritten designs, and increases linearly with the filter length. The required clock frequency of 500 MHz was achieved in trial syntheses.

Code quality has been monitored with both code reviews and automated lint checks. The resulting code conforms to Nokia VHDL guidelines and is virtually indistinguishable from handwritten RTL.

However, there is still plenty of room for further development. Applications typically require some sort of custom logic associated with the filter that the flow doesn't currently support. For example, there may be marker signals that must be passed through the filter with an appropriate latency. The generated filters don't also implement decimation or interpolation by themselves.

Only very simple optimizations are used in the generator, leaving a lot of room for improvement. Were the generator to support decimation in the future, clever tricks could be utilized to reduce unnecessary calculations. On another note, in a typical application the filters are used to filter IQ data where the two branches can share some logic.

In some applications a folded filter could be preferred instead of a transposed one. For instance, when applying different sets of coefficients for the same input data, the arithmetic units can share the same delay line. The smaller register size requirements when compared to the transposed topology could also mean that a folded multichannel filter would consume less memory. Alternatively, rounding the multiplication results could be investigated more thoroughly.

The most urgent expansion to the generator will be the support for programmable coefficients. During discussions it turned out that algorithm developers are wary of fixing the coefficients as the requirements may change. For example, the product could be taken to a new geographic location, requiring some adjustments in the filtering.

From the generator point of view, programmable coefficients are not very interesting. Supporting them requires full multipliers, and the implementation has to be so generic that very little optimizations can be made. Coefficient analysis in its current form is naturally out of the question as the coefficients are not known. VHDL will have enough expression power for nearly all of the required settings.

Programmable coefficients will also require a test bench that is very different from the one presented in this Thesis. The coefficients are set via a register bank connected to a system bus, and they must be initialized in the beginning. To make the transactions easy, an Universal Verification Methodology (UVM) test bench with a proper verification IP (VIP) for the bus should be used.

With all these ideas about expanding the generator, it is important to be careful about adding in new features. Doing some exotic optimizations for a specific application results in an increased level of complexity for the generator. Automatizing the generation of something is only worth it if the feature is needed more than a couple of times.

BIBLIOGRAPHY

[1] Altera Corporation, *Fir Compiler II IP Core*, 2014. [Online]. Available: http://www.altera.com/literature/ug/ug_fir_compiler_ii.pdf , accessed 2015-02-24.

[2] D. Bishop, *Fixed point package user's guide.* [Online]. Available: http://www.eda.org/fphdl/Fixed_ug.pdf , accessed 2015-02-24.

[3] P. P. Chu, *RTL hardware design using VHDL : coding for efficiency, portability, and scalability.* Hoboken, NJ: Wiley-Interscience, 2006.

[4] A. G. Dempster and M. D. Macleod, "Constant integer multiplication using minimum adders," *Circuits, Devices and Systems, IEE Proceedings* -, vol. 141, no. 5, pp. 407–413, 1994.

[5] C. D. Howard, "Minimizing FIR Filter Designs Implemented in FPGAs Utilizing Minimized Adder Graph Techniques," Master's thesis, Tallahassee, 2008.

[6] K. Hwang, *Computer arithmetic: principles, architecture, and design.* New York: John Wiley & Sons, 1979.

[7] A. Karatsuba, "The Complexity of Calculations," in *Proceedings of the Steklov Institute of Mathematics*, vol. 211, 1995, pp. 169–183.

[8] P. Kukkala, *AXI-Stream FIR reference design study*, 2013, Nokia Sharenet D509623246. Limited availibility.

[9] R. G. Lyons, *Understanding digital signal processing*, 2nd ed. Upper Saddle River, NJ: Prentice Hall PIR, 2004.

[10] P. K. Meher, S. Chandrasekaran, and A. Amira, "FPGA Realization of FIR Filters by Efficient and Flexible Systolization Using Distributed Arithmetic," *Signal Processing, IEEE Transactions on*, vol. 56, no. 7, pp. 3009–3017, 2008.

[11] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*, 3rd ed. Berlin, Germany: Springer Berlin Heidelberg, 2007, 774 p.

[12] L. Ming and Y. Chao, "The Multiplexed Structure of Multi-channel FIR Filter and its Resources Evaluation," in *Computer Distributed Control and Intelligent Environmental Monitoring (CDCIEM), 2012 International Conference on*, 2012, pp. 764–768.

[13] A. Mirshekari and M. Mosleh, "Hardware implementation of a fast FIR filter with residue number system," in *Industrial Mechatronics and Automation (ICIMA), 2010 2nd International Conference on*, vol. 2, May 2010, pp. 312–315.

[14] S. K. Mitra, *Digital signal processing : a computer based approach*, 3rd ed. New York: McGraw-Hill Higher Education, 2006.

[15] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code Generation for DSP Transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[16] V. S. Rosa, F. F. Daitx, E. Costa, and S. Bampi, "Design flow for the generation of optimized FIR filters," in *Electronics, Circuits, and Systems, 2009. ICECS 2009. 16th IEEE International Conference on*, 2009, pp. 1000–1003.

[17] T. Saramäki, "Finite impulse response filter design," in *Handbook for Digital Signal Processing*, S. K. Mitra and J. F. Kaiser, Eds. John Wiley & Sons, 1993, pp. 155–277.

[18] V. Savela, "Comparison of digital filter architectures using synthesizable VHDL," Master's thesis, Tampere University of Technology, Tampere, 1996.

[19] The MathWorks, Inc., *Optimized Fixed-Point FIR Filters.* [Online]. Available: http://se.mathworks.com/help/dsp/examples/optimized-fixed-point-fir-filters.html , accessed 2015-02-24.

[20] V. Verma and C. Chien, "A VHDL based functional compiler for optimum architecture generation of FIR filters," in *Circuits and Systems, 1996. ISCAS '96., Connecting the World., 1996 IEEE International Symposium on*, vol. 4, 1996, pp. 564–567 vol.4.

[21] Xilinx, Inc., *Distributed Arithmetic FIR Filter v9.0*, DS240, 2005. [Online]. Available: http://www.xilinx.com/ipcenter/catalog/logicore/docs/da_fir.pdf , accessed 2015-02-24.

[22] Xilinx, Inc., *IP LogiCORE FIR Compiler v5.0*, DS534, 2011. [Online]. Available: /http://www.xilinx.com/support/documentation/ip_documentation/fir_compiler_ds534.pdf , accessed 2015-02-24.

[23] R. Yates, *Practical Considerations in Fixed-Point FIR Filter Implementation*, PA5, 2010. [Online]. Available: http://www.digitalsignallabs.com/fir.pdf , accessed 2015-02-24.

[24] R. Zimmermann, "Datapath synthesis for standard-cell design," in *Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on*, 2009, pp. 207–211.