

**Design and Implementation of Sampling Rate Converters
for Conversions between Arbitrary Sampling Rates**

by

**Fedor Merkelov,
Yaroslav Kodess**

**LiTH-ISY-EX-3520-2004
Linköping, 2004**

**Design and Implementation of Sampling Rate Converters
for Conversions Between Arbitrary Sampling Rates**

by


**Fedor Merkelov,
Yaroslav Kodess**

LiTH-ISY-EX-3520-2004

Supervisor: Håkan Johansson

Examiner: Håkan Johansson

Linköping, 26th of March, 2004

	Avdelning, Institution Division, Department Institutionen för systemteknik 581 83 LINKÖPING	Datum Date 2004-03-26
---	---	------------------------------------

Språk Language Svenska/Swedish X Engelska/English	Rapporttyp Report category Licentiatavhandling X Examensarbete C-uppsats D-uppsats Övrig rapport _____	ISBN	
		ISRN LITH-ISY-EX-3520-2004	
		Serietitel och serienummer Title of series, numbering	ISSN _____

URL för elektronisk version
<http://www.ep.liu.se/exjobb/isy/2004/3520/>

Titel Title	Design and Implementation of Sampling Rate Converters for Conversions between Arbitrary Sampling Rates
Författare Author	Fedor Merkelov, Yaroslav Kodess

Sammanfattning
 Abstract
 In different applications, in digital domain, it is necessary to change the sampling rate by an arbitrary number. For example Software Radio which should handle different conversion factors and standards. This work focuses on the problem of designing and implement sampling rate converters for conversions between arbitrary sampling rates. The report presents an overview of different converter techniques as well as considers a suitable scheme with low implementation cost. The creating VHDL generator of Farrow-based structure to speed up the design process is the main task of this work. The suitable design technique which is the most important thing in any design work is presented in the report as well. The scheme which is considered to be suitable is created by VHDL generator and tested in MATLAB. The source code is attached to the report. And some results from tests of the implemented scheme.

Nyckelord
 Keyword
 VHDL code generator, digital filter, ajustable delay, Farrow, sampling rate converter

Abstract

In different applications, in digital domain, it is necessary to change the sampling rate by an arbitrary number. For an example Software Radio, which should handle different conversion factors and standards.

This work focuses on the problem of designing and implement sampling rate converters for conversions between arbitrary sampling rates.

The report presents an overview of different converter techniques as well as considers a suitable scheme with low implementation cost. The creating VHDL generator of Farrow-based structure to speed up the design process is the main task of this work. The suitable design technique, which is the most important thing in any design work, is presented in the report as well.

The scheme, which is considered to be suitable, is created by VHDL generator and tested in MATLAB. The source code is attached to the report. And some results from tests of the implemented scheme.

Acknowledgements

This Master thesis has been written at the Department of Electrical Engineering (ISY), Linköping University as a final work of the International Master Program in SoCware.

We would like to thank Håkan Johansson and Henrik Ohlsson for providing us with source code of filter generator that we used in our thesis work as well as for their help in any other questions.

Our special thanks are to our friend Oleg Zakaznov for his suggestions in different aspects.

Table of contents

1 Introduction	1
1.1 Background.....	1
1.2 Motivation	1
1.3 Purpose	1
1.4 Summary of main results.....	2
1.5 About this document	2
<i>Literature review</i>	2
2 Different converters techniques	5
2.1 Introduction	5
2.2 Basic concepts and definitions	5
2.3 Schematics of sampling rate converters	8
2.4 Design techniques.....	14
3 Implementation.....	19
3.1 Introduction	19
3.2 Implementation details	19
3.3 Setting up the task	19
3.3.1 <i>System level approach</i>	19
3.3.2 <i>Signal flow graph level analysis</i>	23
3.3.3 <i>Prerequisites definition</i>	25
3.4 Task solution	25
3.4.1 <i>Step-by-step adaptation and simplification of the SFG.</i>	25
3.4.2 <i>Final schematic description</i>	30
3.5 VHDL code structure.	34
3.6 Generator C code structure.....	36
3.7 Configuration file description	38
4 Design Example and Experimental Results.....	43
4.1 Introduction	43
4.2 Experiments on Benchmark	43
4.3 Internal behavioral simulator.....	45

4.4 User manual guide.....	46
5 Conclusion and future work.....	49
5.1 Conclusion.....	49
5.2 Future work	49
References	51
Abbreviations.....	53
Appendix A VHDL components interface description.....	55
Appendix B Configuration file example.....	63
Appendix C Command line parameters	67

Figure list

- Figure 1 Analog method of sample-rate conversion..... 6
- Figure 2 All-digital method of sample-rate conversion..... 6
- Figure 3 Time-domain view of sample-rate conversion 7
- Figure 4 The structure (a) no block structure (b) block structure 9
- Figure 5 Modified Farrow structure 11
- Figure 6. Modified transposed Farrow Structure I..... 12
- Figure 7. Transposed modified Farrow structure II 13
- Figure 8. Adjustable fractional delay filter 14
- Figure 9. Block definition of FAD 20
- Figure 10. Generator system level block diagram. 21
- Figure 11. SFG for FAD 23
- Figure 12 Delay chain 24
- Figure 13 FIR filter..... 24
- Figure 14. FAD SFG part and its expansion..... 26
- Figure 15. Introducing the common delay line..... 27
- Figure 16. General block schematic of parallel multiplier..... 28
- Figure 17 6-input adder tree example 28
- Figure 18 FAD fragment with expanded FIR filters..... 29
- Figure 19 FAD fragment with joined adder tree 29
- Figure 20 Simplified and adapted FAD block schematic. 31
- Figure 21 Block schematic of a 4-input adder 32
- Figure 22. Quantizer. Rounding case 33
- Figure 23. Quantizer. Extension case..... 33
- Figure 24. Quantizer. Pass through case 33
- Figure 25. FAD VHDL level hierarchical diagram 35
- Figure 26. Tool block schematic 37
- Figure 27. Design Example: Structure 43
- Figure 28. Design Example: Impulse response..... 44

1 Introduction

1.1 Background

Now the video and audio move more and more to completely digital processing of a signal. Hence, the problem of dealing with equipment with different sampling rates has become really severe. There was usually only a single digital processor in any particular signal chain and only Analog to Digital (A/D) or Digital to Analog (D/A) converters used in that kind of chains. But today it is common to find completely digital studio for processing audio or/and video signals. In that kind of studios the signal is digitized immediately after the receiver (original analog source). All operations such as editing and processing remain in the digital domain which is considered to be an advantageous solution in all future equipment. For all that was said above, there is a need for simple digital interfacing between different digital equipments.

1.2 Motivation

The main problem in designing such a system is the complexity of design process. The manual design usually provides the desired implementation results but leads to long design times.

During the design process a fast design technique is needed to reduce the time of the design process.

1.3 Purpose

The purposes of this thesis are:

- To study different converter techniques in order to work out the best solution for the problem at hand.
- To select the schematics which are suitable to implement with respect of small area and low power consumption.

- To find suitable techniques to design sample rate converters.
- To reduce the time of the design by creating an automatic tool to generate the VHDL code of the system with given parameters.

1.4 Summary of main results

In order to test created automatic tool, existing high level synthesis benchmark is used. In addition, a new one was created to provide testing with more precision and accuracy. Design examples and experimental results are presented in chapter 4 of this document.

1.5 About this document

Literature review

Information on this subject was obtained from academic papers found in electronic libraries, such as IEEE, from some books in this area, from some papers which focus on describing design techniques and tools. Most of the information was taken from academic papers. In addition there was used Internet databases to find some information related to programming and design techniques.

The bibliography section includes the links referenced in this document.

Prerequisites

It must be noticed that reader of this thesis is assumed to have general knowledge in digital signal processing.

Outline

- Chapter 2 provides detail concerning the different converter techniques. In addition, more precise definition of the problem is presented.

- Chapter 3 describes the system and tool implementation details. The architecture is presented.
- Chapter 4 concluded with the implementation by introducing the experimental results based on testing some design examples. It also includes a user manual.
- Chapter 5 presents some conclusions and proposes suggestions for future work.
- Appendix A contains VHDL components interface description.
- Appendix B provides with configuration file example.
- Appendix C presents command line parameters.

2 Different converters techniques

2.1 Introduction

The chapter considers different converter techniques. First some basic concepts and definitions will be considered. Then, some techniques will be introduced as well as some discussion of their advantages and disadvantages. Finally, some suggestion of suitable techniques will be presented.

2.2 Basic concepts and definitions

Interpolation and decimation are operations used respectively to increase and reduce the sampling rate or frequency, usually by an integer factor. Increasing of a sampling rate requires that new values, not presented in the signal, be computed and inserted between the existing samples. The new value is estimated from a neighborhood of the samples of the original signal. Similarly, in decimation a new value is calculated from a neighborhood of samples and replaces these values in the lower sampling rate. Integer factor interpolation and decimation algorithms may be implemented using efficient Finite Impulse Response (FIR) filters and are therefore relatively easy to implement. Alternatively, interpolation by non-integer factors typically uses polynomial interpolation techniques resulting in more complex solutions.

There are two classes of sample-rate converters. The first class is synchronous and the second one is asynchronous. In synchronous sample rate converters, the sample rate of incoming signal is converted to a new sample rate by an integer factor. It is suitable in many applications but if irrational conversion factors are needed the problem appears. Its digital output is producing the output samples at a fixed rate related to the input

rate. On the other hand, asynchronous sample rate converters produce output samples at rate, which is independent from the input rate.

There are two methods of sample rate conversion. The first one is analog that is the simplest in principle but not in practice. The idea is to use a D/A converter in combination with a “brick wall” filter. The “brick wall” filter removes all signal images. Then output A/D converter converts signal back to a digital format. The A/D converter runs at the output sampling rate. Figure 1 shows the block-diagram of that design. [1]

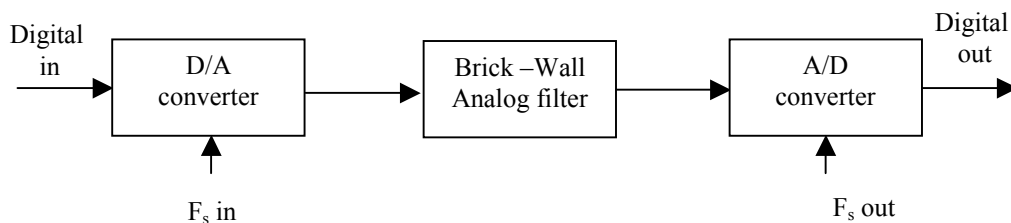


Figure 1 Analog method of sample-rate conversion

But the main problem of Analog method is that Analog functions are more difficult to implement than digital functions. On that reason, the all-digital solution is more preferred.

The general principle of all-digital sample rate converter is almost the same but the analog filter is replaced by a digital interpolation filter. The Figure 2 shows the block-diagram of all-digital sample-rate converter. [1]

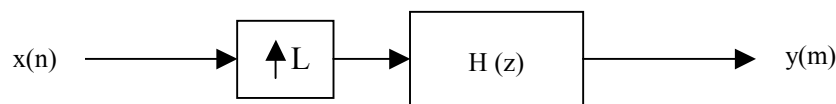


Figure 2 All-digital method of sample-rate conversion

The sample-rate conversion problem may be formulated using the interpolation/decimation model in the time-domain view shown in Figure 3. The output sample rate (trace C) is higher than the original input sample rate (trace A). It can be done by first interpolating by A and then decimating by B. The interpolated values are fed into a zero-order-hold

and then resampled by the output switch (trace D). The output values appear to be representation of the values produced by interpolation filter (which is the nearest in time). Because the output sampling switch is not closing in exactly the time corresponding to a point on the fine time grid of the interpolated output there appear some errors, which could be made small by increasing the filter order.

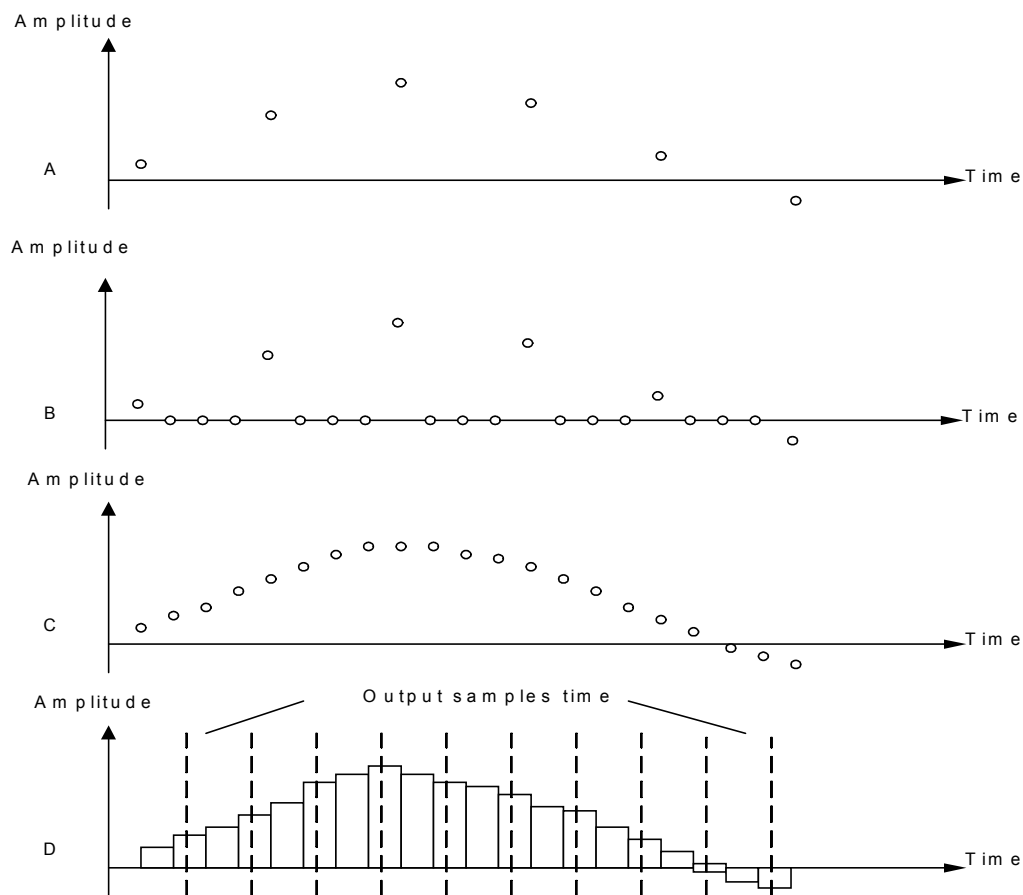


Figure 3 Time-domain view of sample-rate conversion

Sampling rate conversions, performed between arbitrary sampling rates, tends to make the sample rates conversion factor to be a ratio of two very large integers or even an irrational number.

The ratio of f_1 and f_2 can be expressed as M/N , where M/N is computed by the least-common-multiple of the two sampling rates f_1/f_2 . Hence, the sample rate of the input signal is first interpolated by a factor of M using digital filters and decimated by a factor of N to obtain the final

sampling rate. If the integers M and N are manageable numbers less than 10, for example, it is possible to solve this problem by combination of interpolator and decimator. But there are cases when this number can be irrational or too high to implement in simple way. For example, the ratio of 44.1 and 48 kHz can be expressed as 147:160. Hence, the sample rate of the input signal is first interpolated by a factor of $M=160$ using digital filters and decimated back by a factor of $N=147$, but resulting filter would become rather big. To overcome this problem $147:160$ can be accomplished by breaking this into the conversion procedures of $3 : 2$ and $7 : 5$. The overall conversion filter can be obtained by taking the cascade combination of all these. Hence, the overall filter should consist of cascaded subfilters.

2.3 Schematics of sampling rate converters

N.Aikawa and Y.Mori proposed an interpolation kernel filter [2]. It is approximated in each sampling section piece by using a quadratic functions in equation (1). The block structure of the proposed kernel shown in Figure 4(b).

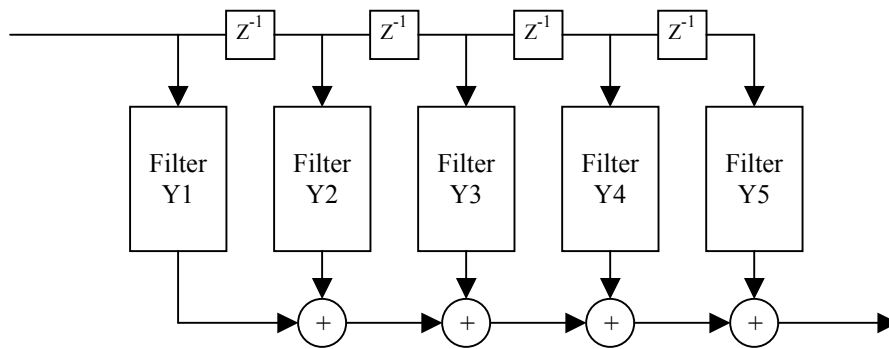
The linear combination of the input data and reconstruction kernel is:

$$f(x) = \sum_i f_i \cdot y(x-i)$$

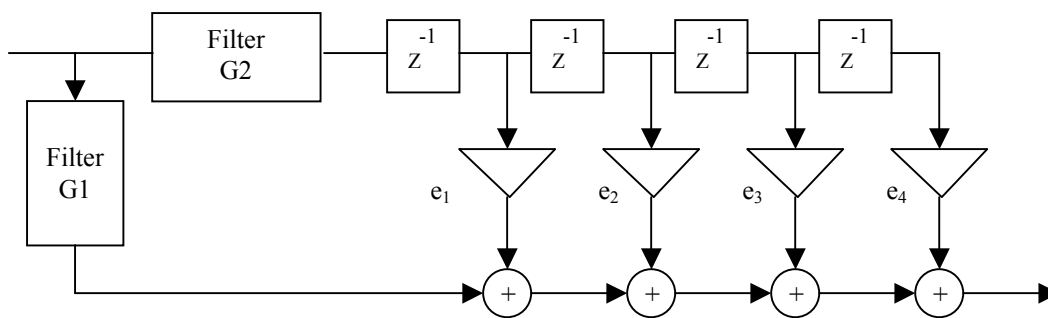
where f_i are the sample values and $y(x)$ is the reconstruction kernel.

$$y(x) = \begin{cases} a_{1,l}x^2 + b_{1,l}x + c_{1,l} & (0 \leq |x| \leq 1/N) \\ \vdots & \\ a_{1,r}x^2 + b_{1,r}x + c_{1,n} & ((n-1)/N \leq |x| \leq 1) \\ \vdots & \\ a_{s,r}x^2 + b_{s,r}x + c_{s,n} & (s-1+(n-1)/N \leq |x| \leq s-1 + n/N) \\ \vdots & \\ a_{s,N}x^2 + b_{s,N}x + c_{s,N} & (s-1+(n-1)/N \leq |x| \leq s) \end{cases} \quad (1)$$

The block diagram of the filter presented in Figure 4 shows the structure with 5 sampling sections.



(a)



(b)

Figure 4 The structure (a) no block structure (b) block structure

Kernel with block structure is the modification of the kernel without block structure. Changing coefficients of multiplication e_i allows changing the ratio of sampling rate conversion.

The proposed Kernel of the general form is:

$$g(x) = \begin{cases} a_{1,1}x^2 + b_{1,1}x + c_{1,1} & (0 \leq |x| \leq 1/N) \\ a_{1,n}x^2 + b_{1,n}x + c_{1,n} & ((n-1)/N \leq |x| \leq 1) \\ e_l(a_{2,l}x^2 + b_{2,l}x + c_{2,l}) & (1 \leq |x| \leq 1+1/N) \\ e_l(a_{2,N}x^2 + b_{2,N}x + c_{2,N}) & (1+(N-1)/N \leq |x| \leq 2) \\ e_{s-l}(a_{2,n}x^2 + b_{2,n}x + c_{2,n}) & (s-1+(n-1)/N \leq |x| \leq s-1+n/N) \\ e_{s-l}(a_{2,N}x^2 + b_{2,N}x + c_{2,N}) & (S-1+(N-1)/N \leq |x| \leq S) \end{cases} \quad (2)$$

To produce a useful filter from the proposed general form, some restrictions to equation (2) should be applied.

- 1) $g(x) = g_{l,0}$ for $x=0$
- 2) $g(x) = g_{2,n}$ for $x=n/N$
- 3) $g(x) = 0$ for $x=s$
- 4) C_0 – continuous.

The design problem is to find coefficients e_j (Figure 4(b)), quadratic coefficients a_{ij} and g_{ij} which satisfy the equation:

$$-W(\omega) \cdot \sum_{j=0}^{S-1} e_j \cdot \sum_{i=jL}^{(j+1)L} G(iT) \cos(i\omega T) - \delta^2 < -W(\omega)D(\omega)$$

Where L is a number of evaluation points in one sampling section, $T=1/L$, $W(\omega)$ is weighting function, $D(\omega)$ is ideal frequency characteristic and δ is the maximum allowable approximation error.

When the difference between δ_i and δ_{i+1} is less than 10^{-5} , the filter can be a FIR filter with the found transfer function. Otherwise an order and coefficients recalculation is needed.

An advantage of the digital kernel technique is that there is no necessity to redesign filter, whenever the sampling rate is changed. Another advantage is that the kernel obtained is easy to implement. The disadvantage of this method is high computational complexity. This is because of the large number of quadratic functions, which are needed to obtain a high attenuation in the stopband.

Another implementation of sample rate converter is based on the Farrow structure. Some modifications that can be produced to achieve low area and efficiency of it according to Djordje Babic, Jussi Vesma, Tapio Saramuki and Markku Renfors from Nokia and Tampere Universitet of Technology, Finland. [3]

Firstly, it is modified Farrow structure shown in Figure 5 that has fixed filter coefficients as a benefit. The only changeable parameter is the fractional interval μ :

$$\mu = k/R - [k/R]$$

Where R is the decimation factor.

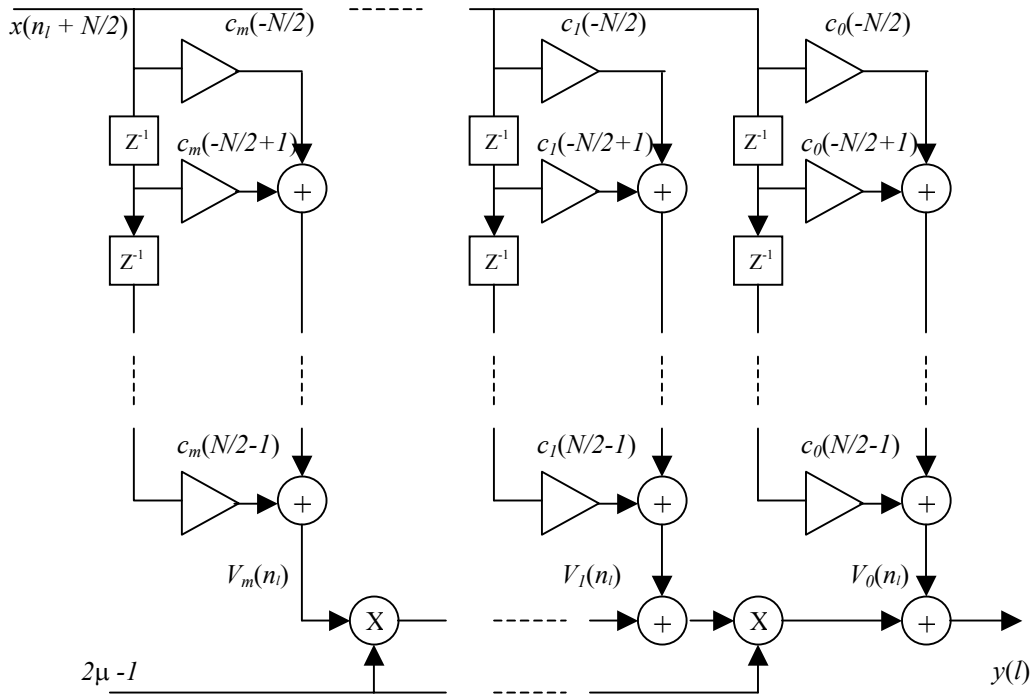


Figure 5 Modified Farrow structure

The output samples are:

$$v_m(n) = \sum_{k=0}^{N-1} x(n-k + \frac{N}{2}) c_m \cdot (k - \frac{N}{2})$$

of the $m+1$ FIR filters. Where m is the quantity of the filters c is multiplication coefficient. Each of the FIR filter has the transfer function:

$$C_m(z) = \sum_{k=0}^{N-1} c_m \cdot (k - N/2) \cdot z^{-k}$$

Where N is the parametrical coefficient.

Structure in the Figure 6 contains $(m+1)N+m$ multipliers working at the input sampling rate f_1 , N integrators as well as dump circuits and $N-1$ delay elements at the output sampling rate f_2 .

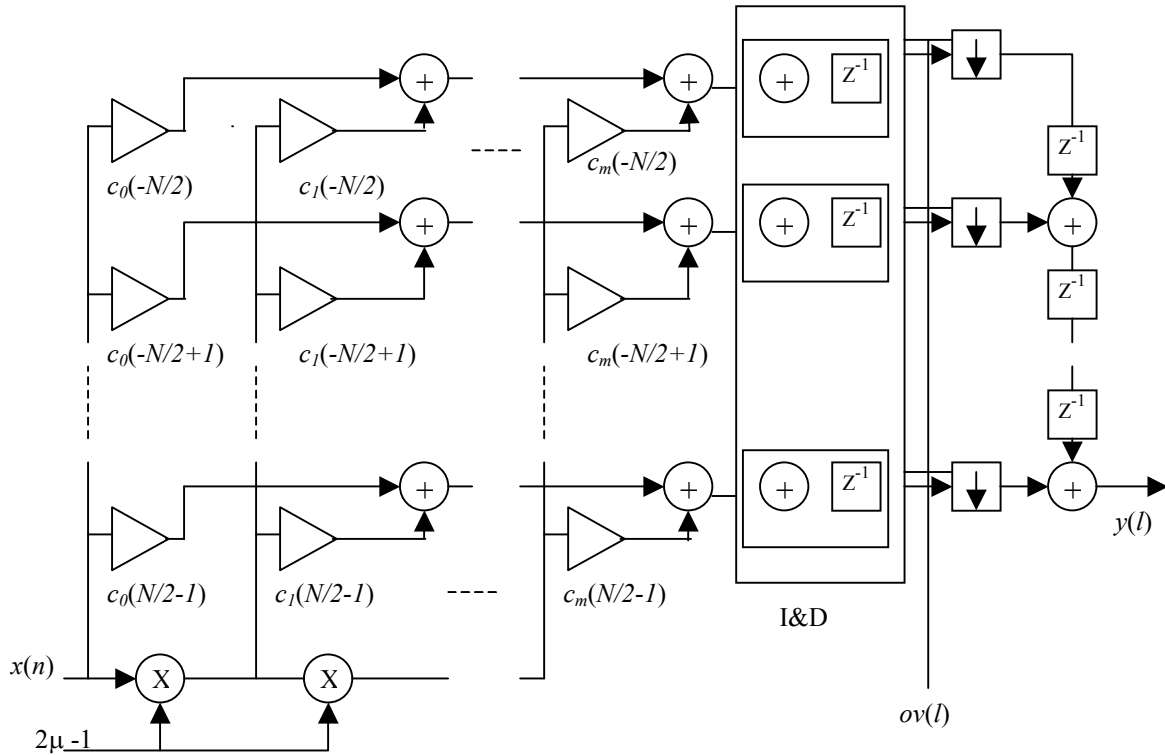


Figure 6. Modified transposed Farrow Structure I

The fractional interval here is:

$$\mu_{k+1} = \mu_k + 1/R - [\mu_k + 1/R]$$

where R is decimation factor.

and the signal $ov(l)$ is needed to indicate an overflow of the accumulator.

The fractional interval for each sample is:

$$\mu_k = (kT_1 - n_k T_2)/T_2$$

where k is number of sample, T is period of sample.

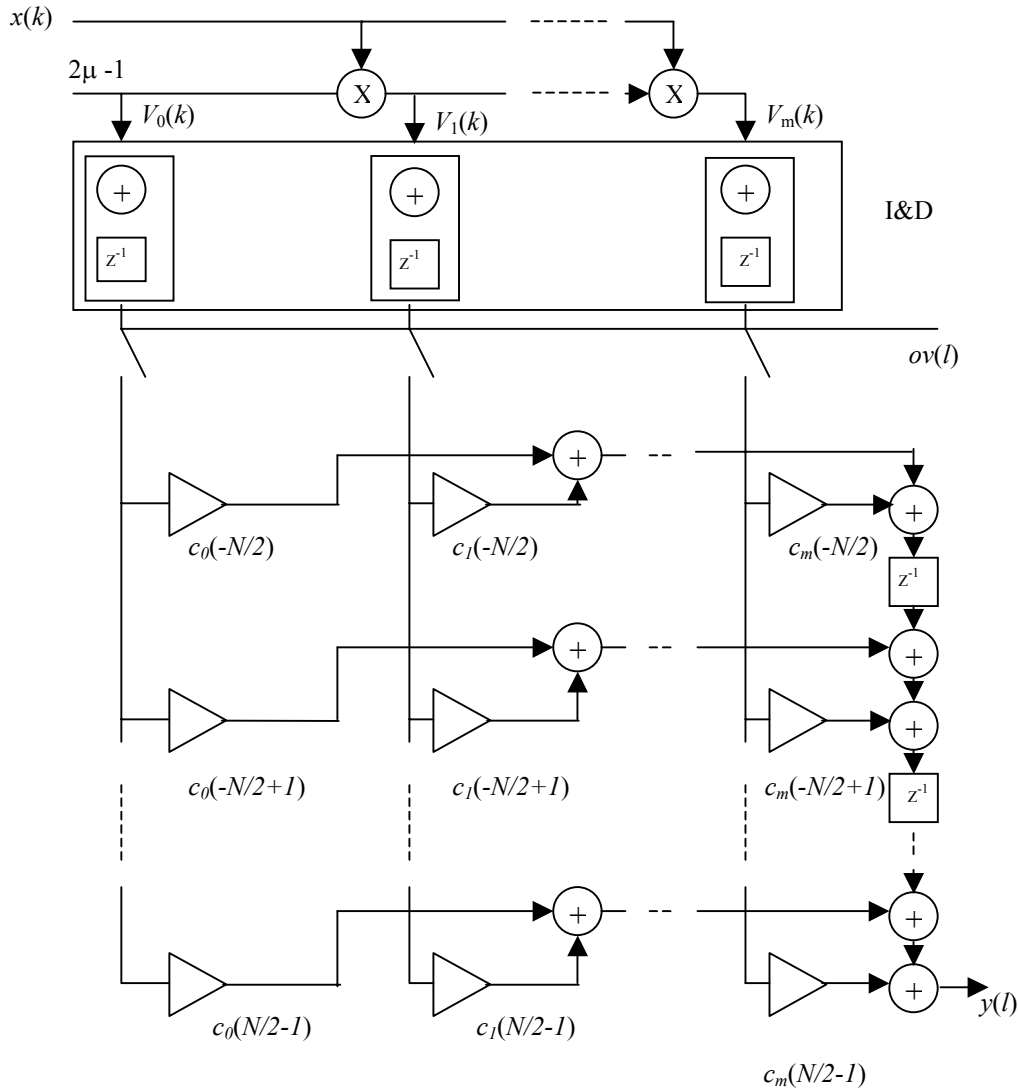


Figure 7. Transposed modified Farrow structure II

The structure shown in Figure 7 has $N(m+1)$ multipliers working at the output sampling rate and m multipliers working at the input sampling rate. In addition, there are m I&D circuits working at the input sampling rate as well as $N-1$ delay elements working at the output sample rate.

The structures shown in Figures 5, 6 and 7 have almost the same performance but the second one consumes less power during conversion from higher sample rate to lower.

The main advantage of those structures is that the filter coefficients are fixed which is very good for designing and producing. The fractional interval is the only one changeable parameter.

The modification of the Farrow structure shown in Figure 5 is in fact adjustable fractional delay filter. It is suitable for interpolation, while transposed Farrow structures (Figure 6 and 7) are suitable for decimation. Adjustable fractional delay filter has the low area cost due to possibility to share delay elements among subfilters. Due to this it is considered to be very suitable to implement. The designing process of that structure will be discussed in the next chapter.

2.4 Design techniques

This structure shown in Figure 8 consists of several parallel subfilters, which are linear-phase FIR filters. Due to the coefficient symmetry of such filters the number of required multiplications may be reduced.

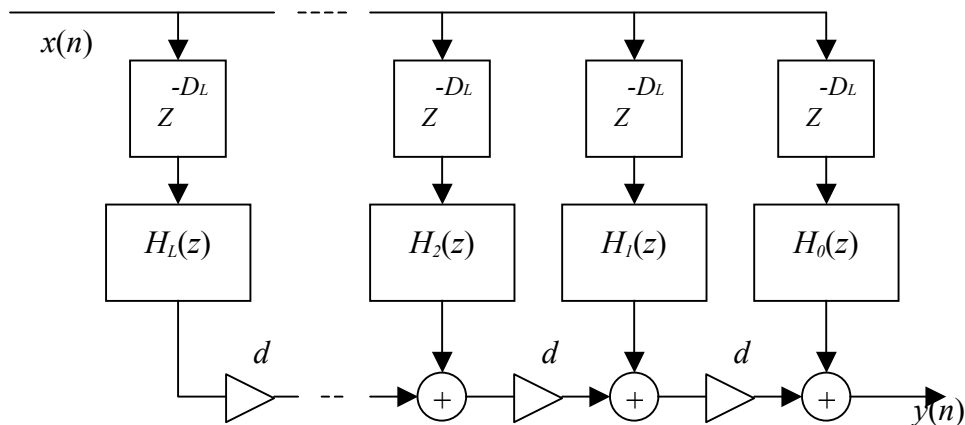


Figure 8. Adjustable fractional delay filter

The transfer function of the overall filter is:

$$H(z) = \sum_{k=0}^L d^k \cdot z^{-D_k} \cdot H_k(z)$$

Where $d=[-0.5;0.5]$ is a multiplication coefficient, D_k is the delay of k -th subfilter, L is the quantity of subfilters and $H_k(z)$ are subfilters frequency responses:

$$H_k(e^{j\omega T}) = e^{-jN_k \frac{\omega T}{2}} \cdot H_{kR}(\omega T)$$

Where N is order of subfilter.

The desired real function $H_{kR, des}(\omega T)$ is:

$$H_{kR, des}(\omega T) = \begin{cases} \frac{(-j\omega T)^k}{k!}, & k \text{ is even} \\ -j \frac{(-j\omega T)^k}{k!}, & k \text{ is odd} \end{cases}$$

Where $k = [0;L]$.

Each subfilter should approximate the frequency response of an N_k -th order differentiator.

The advantage of such a structure is that there is no need in redesigning the subfilters but just change the single coefficient d . The subfilters are designed only one time.

Håkan Johansson and Per Lowenborg [4] recommend separate optimization of the subfilters. This design technique is based on distributing the allowable errors in the error functions.

This approach results in subfilters that can have different order. That allows to reduce arithmetic complexity comparatively with the case where all subfilters are of equal orders.

The specification is:

$$|H_e(e^{j\omega T}, d)| \leq \delta$$

where the complex error $H_e(e^{j\omega T}, d)$ is:

$$H_e(e^{j\omega T}, d) = H(e^{j\omega T}) - H_{\text{des}}(e^{j\omega T}),$$

$$\omega T \in [0, \omega_c T], \quad |d| \leq 0.5$$

To satisfy the specification the selection of L and separately optimization $H_{kr}(\omega T)$ is needed so that:

$$|\delta_k(\omega T)| \leq \frac{2^k \delta}{C(1+\sqrt{2})}$$

$$\frac{(0.5 \cdot \omega_c T)^{L+1}}{(L+1)!} \leq \frac{\delta}{1+\sqrt{2}}$$

where

$$C = \begin{cases} [L/2], & k \text{ is odd} \\ [L/2]+1, & k \text{ is even} \end{cases}$$

This optimization can be solved in MATLAB using function *remez.m*. The specification ε is set that:

$$-\varepsilon \leq \frac{C(1+\sqrt{2})}{2^k \delta} \leq \varepsilon$$

When $\varepsilon \leq 1$ the specification is satisfied. To find the subfilters orders it is easy to design several filters by increasing the filter order until the specification is met. But in this approach the overall filter will be overdesigned for the reason that the interaction between filters could appear and this technique does not consider this. Than the order of filters and the complexity could be higher than necessary.

The other proposed technique is filter design via simultaneous optimization of the subfilters. To reduce the complexity of the overall structure some nonlinear optimization routine should be used. Relaxed specification is used in order to allow iteration of this design procedure. The relaxed specification might have differences between two cases ripples such as a factor of ten. The nonlinear optimization problem could be solved in MATLAB as a standard function *minimax*. Then the relaxation of the design specification by factor Δ is used to find less complex structure. The Δ is chosen that the number of optimization iterations is rational.

The main idea is to use some kind of automatic implementation tools in low level and furthermore in FPGA to reduce design time. The overall filter consists of subfilters which orders and coefficients are the only changeable parameters before complete implementation. Hence, it is necessary to create the program, which will do all kind of “smart” implementation with respect to the low area. The main problem with it is that automatic process should use techniques, which are optimal from engineer’s point of view. To reach that aim it should be created some base of suitable designing solutions. For example, it is the sharing of the delay elements among the subfilters and suitable schematics for signed multiplication. These improvements are playing the main role in the reducing the area. Thus, the important first step is implementation of the single-rate fractional-delay filter and creating the VHDL code generator for it.

Then the appropriate way must be chosen the suitable way to find that kind of parameters which are suitable for adjustable sampling rate conversion.

Assumption that all subfilters have the same parameters seems to be simple because it allows to operate with only two changeable parameters

FIR filters coefficients and fractional delay. But mathematically it is not that simple problem.

3 Implementation

3.1 Introduction

The primary topic of the work is developing a software VHDL code generator for a filter with adjustable delay (FAD).

This chapter describes the FAD generator implementation.

3.2 Implementation details

The tool is written in C programming languages. The output data is a VHDL code which can be further implemented as a register transfer level (RTL) model (in FPGA, for instance). Also simulation in MATLAB was performed for FAD structure to test the generated VHDL model.

3.3 Setting up the task

3.3.1 System level approach

To determine starting design conditions, the main initial factors of the further tool were studied, such as:

1. *Software platform – Solaris;*
2. *Application field assumed – sampling rate converter generation for its further study and hardware implementation;*
3. *User qualification – experienced **Unix** user, scientist, programmer;*
4. *The generator sources will be apparently used as a key part of the higher level generators;*
5. *High portability for using in variety of software environments and operation systems;*
6. *High efficiency of using, fast evoking, fast output generation;*
7. *High functionality;*
8. *Efficient verification mechanism.*

Basing on these key characteristics the following solution for the system level is suggested:

1. *Input data is supplied in noninteractive mode with configuration file read by the tool. It doesn't bring many problems for a Unix user because this way is the most commonly used for this software environment.*
2. *User specifies the necessary operating mode with command-line keys;*
3. *The programming language being used is ANSI C because it is supported by all the software platforms known and provides high degree of portability even for recent operation systems. Special supporting libraries (like input-output library, for instance) from the side software makers are not being used.*
4. *Minimal configuration file syntax.*

The block definition of ready FAD is presented in the Figure 9:

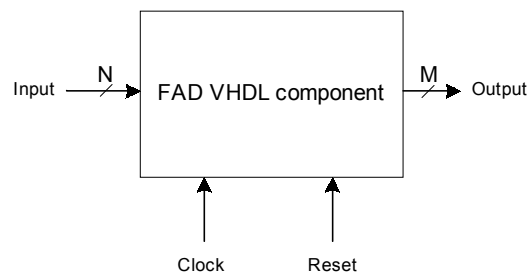


Figure 9. Block definition of FAD

Here:

“N” and “M” are an input and output bus width respectively.

“Reset” sets the block into initial state.

The rate at that an input information is accepted is controlled by a clock signal.

The system level block diagram, showing how the main blocks of the tool are interconnected is added in the Figure 10:

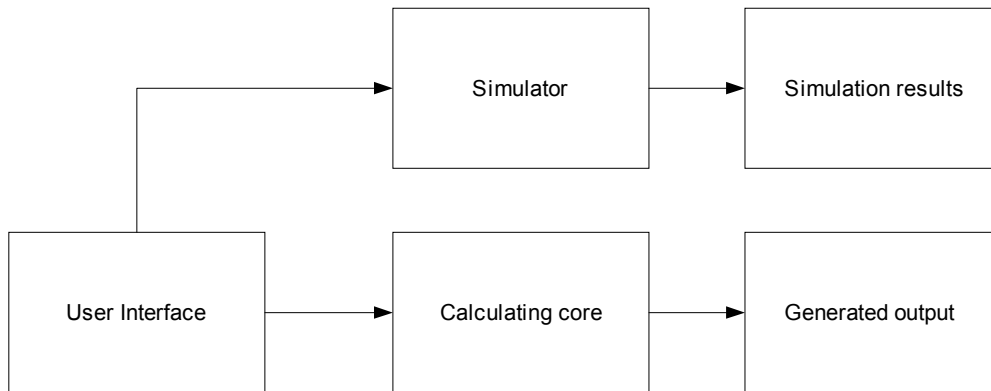


Figure 10. Generator system level block diagram.

Short blocks description:

User interface block is responsible for the interaction with a user. It gets initial data supplied in some certain format (in configuration file in our case) and transforms them into the internal representation;

Calculating core performs all the necessary tool functionality and generates the output;

Simulator provides the mathematical model of the generated schematic and makes possible the comparison between the mathematical simulation and VHDL level simulation. The identity of the results from these two types of simulations guarantees (at certain level of reliability) that produced VHDL level model is equal to the signal flow graph (SFG) required.

Simulation results are real decimal and binary representation of a simulator output given to the user via terminal console or with another type of interface.

Generated output is the set of VHDL files representing the desired structure.

It has been already mentioned that the simulator provides mathematical model of the desired schematic. When certain test vector is applied to the input (it might be a single impulse that is the case here),

only certain output sequence is expected. The equality between mathematical simulation and VHDL modeling guarantee that the equal transformation has been obtained and hence, the current realization is correct.

Simulation can be performed in different ways:

- *System level modeling in Simulink, Ptolemy and in similar frameworks;*
- *Mathematical model in MATLAB, Mathcad, Excel etc;*
- *Integrated simulator.*

When a supplementary modeling tool is implied, it requires that, at first, it must be installed into user environment. Secondly, user must be able to use it (that is though not a serious problem for the user assumed). Multi-purpose visual simulators often do not provide the necessary flexibility and adjustability for the new input data (number of stages, for instance). Evoking the environment with considerable functionality for executing the couple-of-line code seems redundant.

The FAD mathematical model is quite simple, therefore integrating the simulation facilities into software being developed seems an optimal solution. Since the VHDL code generator and the simulator use the single configuration file, it gives additional guarantee of that the mathematical and VHDL models represent the same entity.

The additional advantage is that it is very fast and may be performed in parallel with generation process.

Disadvantages of the approach:

- *A simple verifying simulator complicates, for example, parameters adjusting for given output. It is good only for verifying.*
- *Often user trusts more the well-known systems like MATLAB, especially if he can see the block schematic or the model source code that he examines.*

3.3.2 Signal flow graph level analysis.

Consider general view of a SFG for FAD (Figure 11):

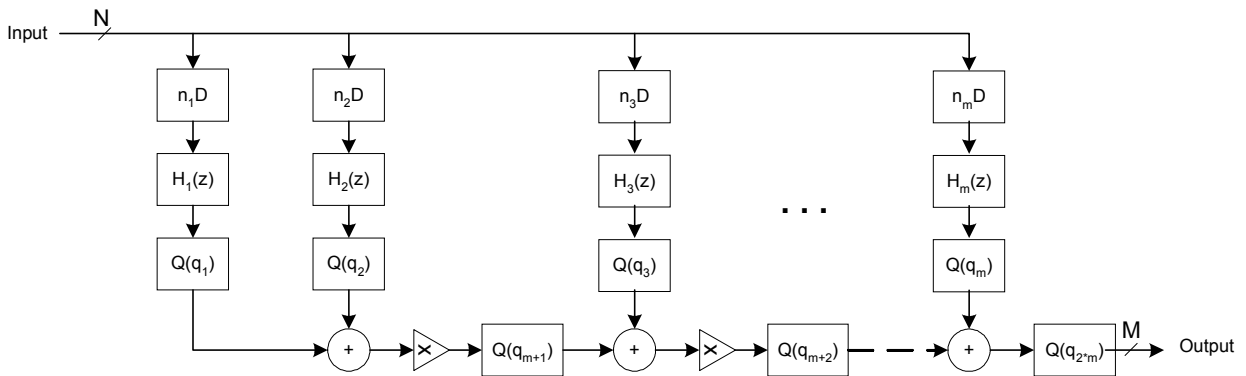


Figure 11. SFG for FAD

Input information is applied with signed parallel code. Output sequence, produced by a FAD has also signed parallel representation. Input bus is connected to the set of delay chains. There can be arbitrary delay before the input information reaches a filter in each stage.

We used the following shortening symbols for the delay chain (Figure 12) and for FIR filter (Figure 13) in this schematic:

1.

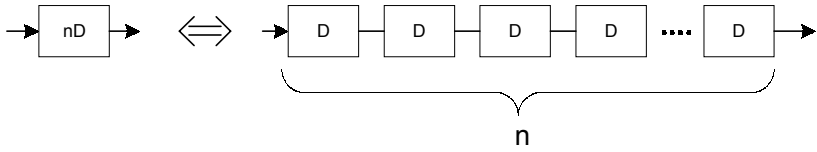


Figure 12 Delay chain

2. Filters denoted as H_i are linear phase FIR filters.

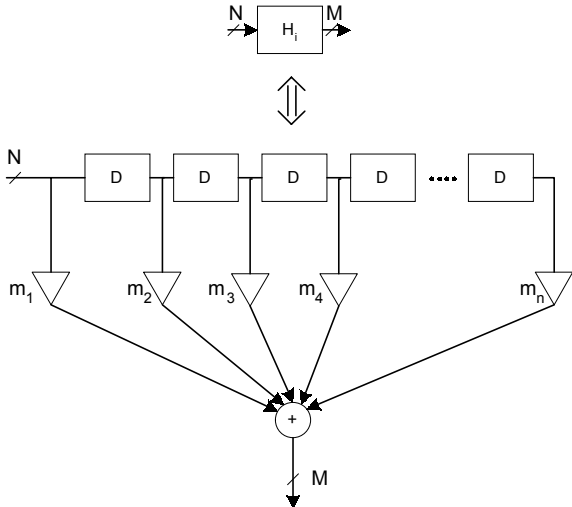


Figure 13 FIR filter

Output bus width depends on the input bus width and on the maximum coefficient length.

3. Quantizers $Q(q)$ manage with internal bit width. They can be used either for intermediate rounding (truncation) or for bus width extension according to the specification. It will be carefully discussed below.

4. Multipliers perform the multiplication of the input value by the fixed coefficients.

3.3.3 Prerequisites definition

The given structure implies that the following input information must be provided by a user:

1. Number of sections of FAD;
2. Input word length;
3. Prefilter delays for each section (number of delays in chain $\{\mathbf{n}_i\}, i = 1..m$;
4. Quantizers parameters (input/output word length) $\{\mathbf{Q}(q_j)\}, j = 1..2*m$;
5. Filter parameters:
 - a. Number of sections k (filter order);
 - b. Multipliers coefficients;
6. Coefficients for multipliers of a FAD.

This list includes essential input information for the FAD generator. It will be replenished as may be necessary.

3.4 Task solution

3.4.1 Step-by-step adaptation and simplification of the SFG.

The SFG, mentioned in the previous chapter, primarily illustrates the schematic to be realized but cannot be implemented directly. It does not contain detailed information about the parts that it is composed of, about bus structure and their sizes, etc. Therefore the model must be analyzed and optimized at the lower hierarchical level.

The task of the current work is not to design a mathematically optimal solution that can be further implemented in FPGA with minimal cost. Therefore obvious simplification will be carried out.

a) Delay chain simplification.

Consider the part of FAD SFG and its expanded view, paying attention to the delay elements (Figure 14):

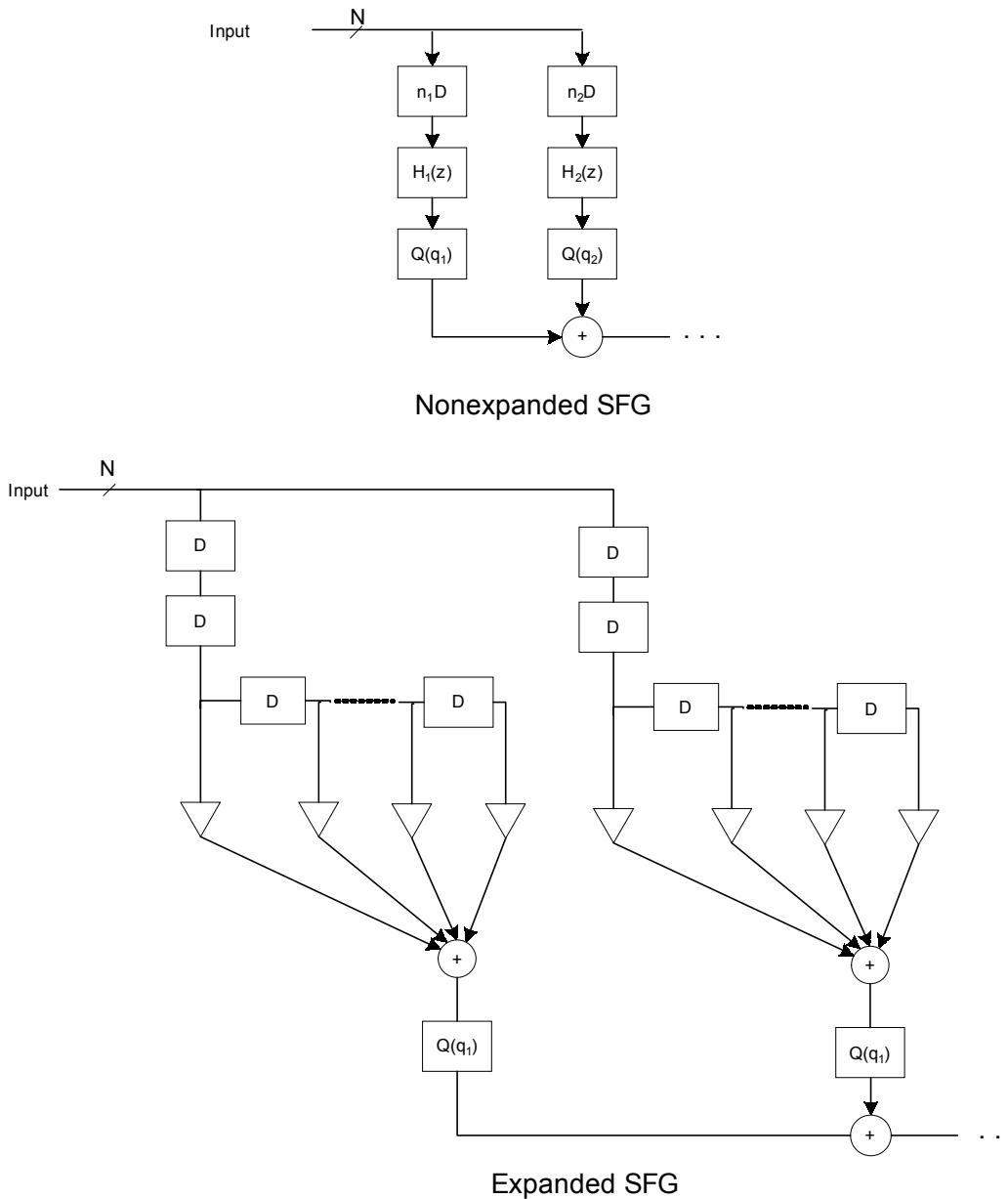
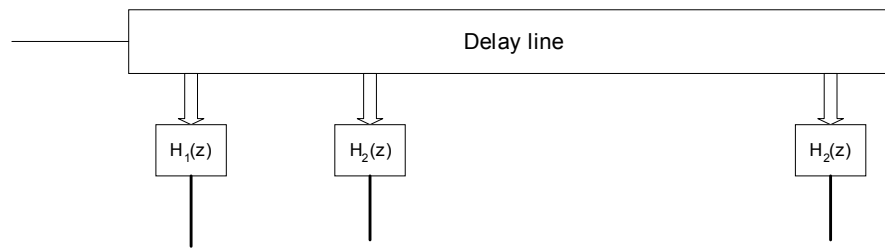


Figure 14. FAD SFG part and its expansion

It is evident that having individual delay lines for each filter and for prefilter delays is redundant. SFG hence can be reduced by introducing the single common delay chain: (Figure 15).



Where

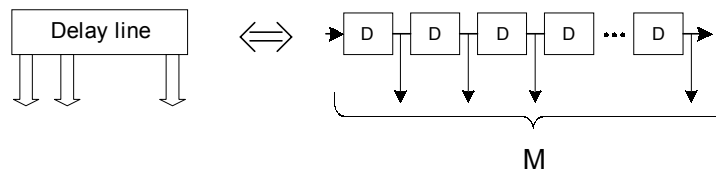


Figure 15. Introducing the common delay line.

Here $M = \max(n_i + k_i)$, for $i = 1 \dots m$ where

n_i – length of the i -th filter pre-delay;

k_i – number of i -th filter internal delays;

m – number of stages of FAD.

Now each filter loses its own delay chain and gets appropriately delayed samples from the common one.

b) Multipliers simplification.

There exist many types of parallel multipliers. Although this structure is quite regular from the implementational point of view, it is very large. But often it can be simplified, for example, if fixed coefficients are used like in our case.

General view of multiplier block schematic is shown in Figure 16:

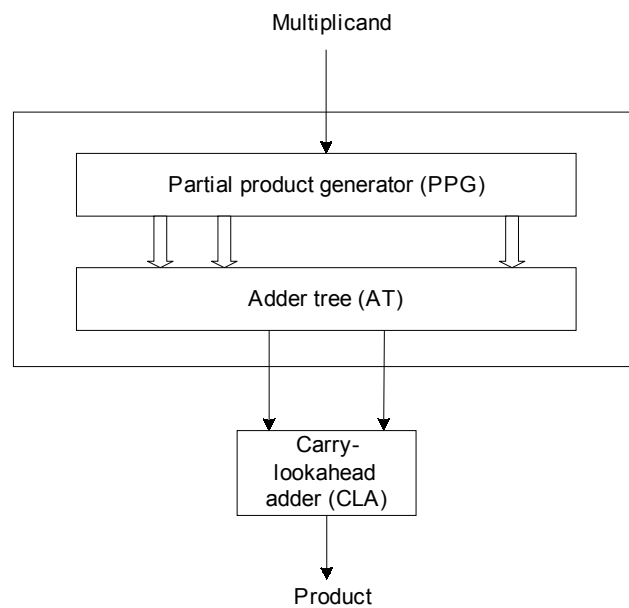


Figure 16. General block schematic of parallel multiplier

Number of partial product is equal to number of nonzero elements (binary ones) in binary representation of a coefficient. It can be reduced a lot by converting coefficient to canonic signed digit code (CSDC) where the number of nonzero bits is minimized [5,6].

Adder tree is based on carry save adders (CSA). Each CSA takes 3 numbers at its input, adds them and produces 2 output result. Thus if it is used in the tree structure, it reduces any number of inputs to 2. A 6-input (each input is multibit) adder tree is shown in Figure 17.

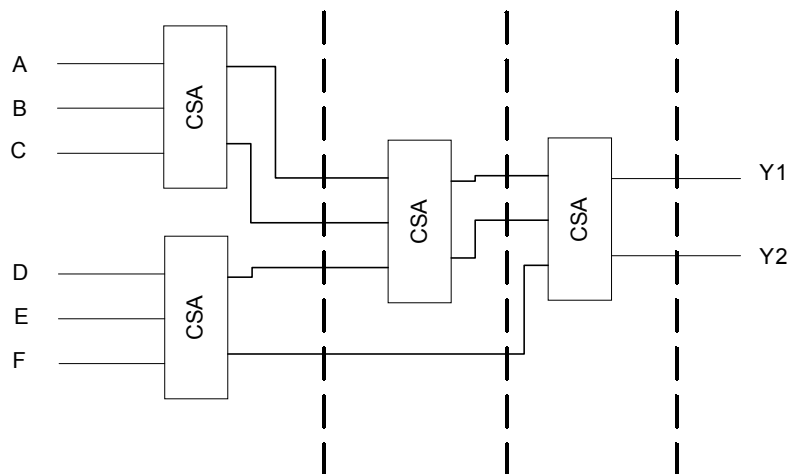


Figure 17 6-input adder tree example

This tree is able to add 6 multibit numbers with latency of $t=3 \cdot t_{CSA}$. t_{CSA} is equal to the propagation time of full adder. In order to decrease critical path, interlayer latches may be introduced (where dotted lines intersect outputs of CSAs).

CLA shown in Figure 16 is optional. It might be used if only final result needed of number of buses should be decreased.

c) Filters simplification.

According to the discussion above, part of the FAD SFG with FIR filters can be drawn like in Figure 18:

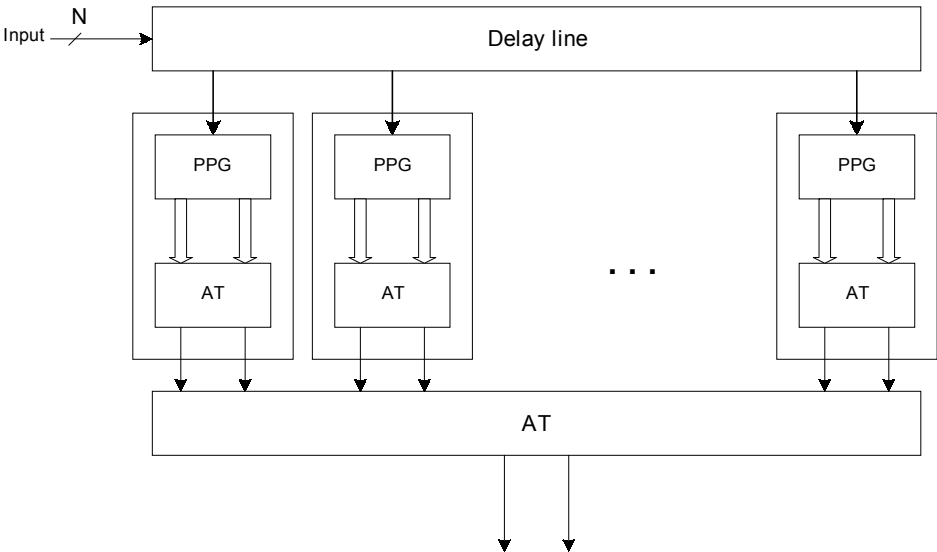


Figure 18 FAD fragment with expanded FIR filters

It is evident that the separated adder trees can be joined into a single tree and schematic can be structurally simplified (Figure 19):

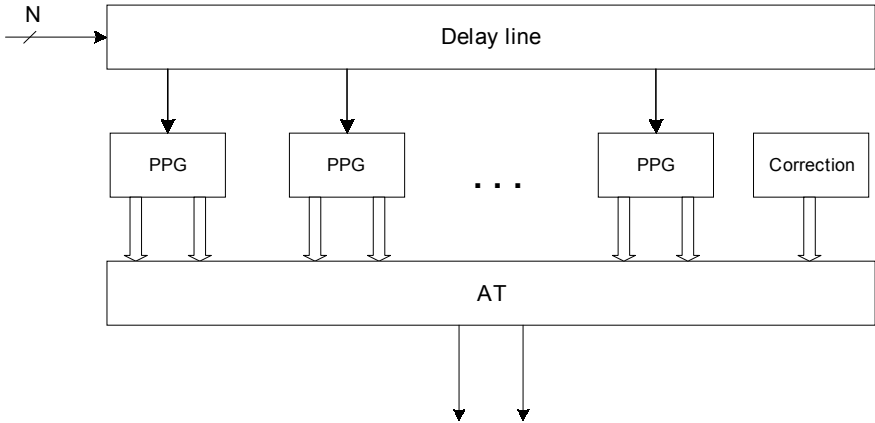


Figure 19 FAD fragment with joined adder tree

Correction block here is in fact a constant binary number. It is also used in previous case (Figure 18) in every infilter adder tree. Correction vector is needed to avoid a sign extension of the partial products.

Rules of partial product (PP) and the correction vector generation.

CSDC encoded coefficient is built according to the following BNF-notation:

$$\{\{0\}\{-1|1\}\}.$$

Denote the coefficient length as ‘m’, input number length as ‘n’ and input number as $x(n)x(n-1)\dots x(0)$

Initial value of correction is equal to 0.

- Iterate from the position 0 to m from right to left;
- If ‘0’ is met in the coefficient in position k, nothing is added to the correction and no PP generated;
- If ‘1’ is met in the coefficient in position k then:

$$PP_k = \langle 0^{m-k} \rangle \langle \overline{x(n-1)x(n-2)\dots x(0)} \rangle \langle 0^k \rangle$$

$$correction = correction - \langle 1^{m+1} \rangle \langle 0^{n-1} \rangle$$

- If ‘-1’ is met in the coefficient in position k then:

$$PP_k = \langle 0^{m-k} \rangle \langle \overline{x(n-1)x(n-2)\dots x(0)} \rangle \langle 0^k \rangle$$

$$correction = correction + \langle 1^{m+1} \rangle \langle 0^{n-1} \rangle + 1$$

Here 1^i and 0^i denote sequence of i ones or i zeros in binary form.

After the iterating through all the coefficient bits is finished, PP vector contains all the partial products, and the value of correction vector is completely calculated.

3.4.2 Final schematic description

After the suggested simplification has been carried out, the FAD schematic shown in Figure 20 is obtained:

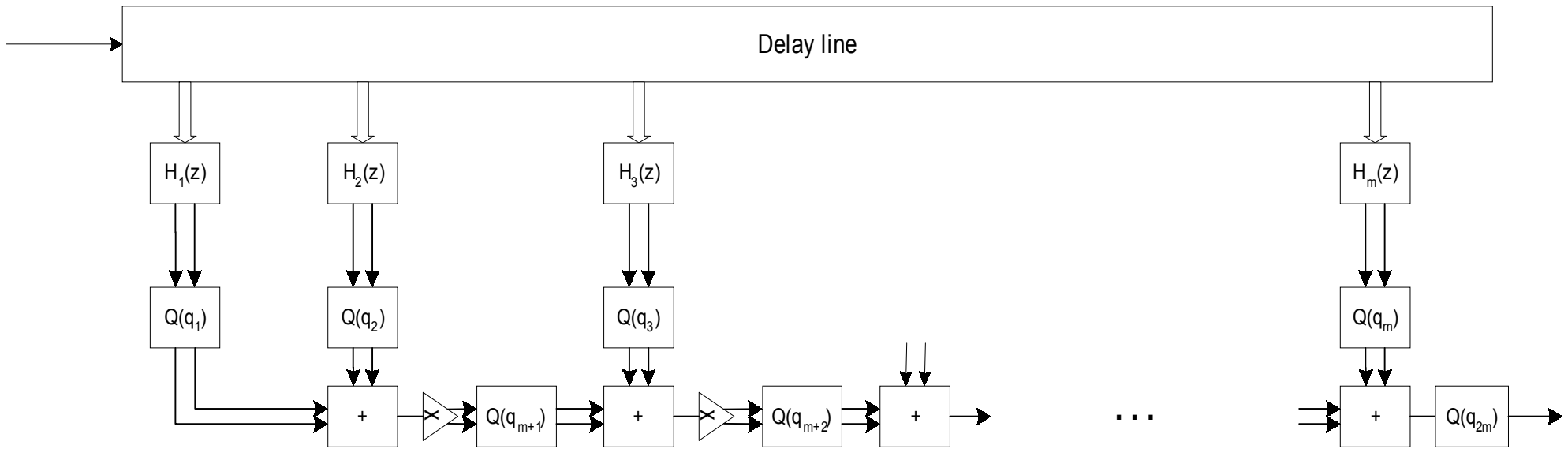


Figure 20 Simplified and adapted FAD block schematic.

This realization uses 2's complement fixed point binary representation. Most significant bit (MSB) is a sign bit. The 2's complement binary encoded number $\{x_i\}$ can be converted into decimal representation by the equation:

$$Q = (-1)^{x_0} + \sum_{i=1}^N 2^{-i} \cdot x_i$$

where x_i are binary number bits, N is a word length.

In this approach 4-input adders are used instead of CLA's at filter outputs. Inner structure of an adder is shown in Figure 21:

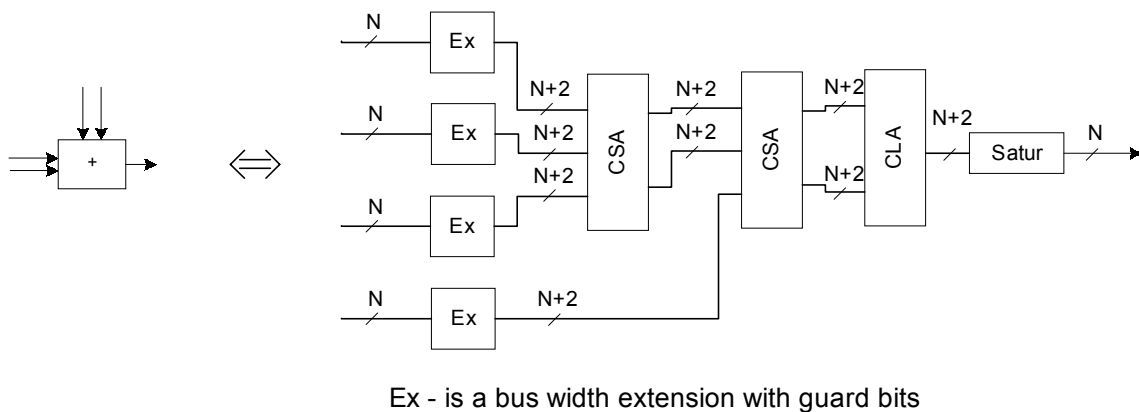


Figure 21 Block schematic of a 4-input adder

4-input adder in Figure 21 has one output. It is done to avoid multiplier duplication in the next stage. When a multiplier coefficient has many nonzero bits, a complex CSA tree must be used in it, which leads to an area consuming register transfer level (RTL) implementation. On the other hand, when a coefficient has few of nonzero bits the approach used may seem ineffective however implementation tool optimizes RTL representation and disposes unnecessary gates that compensate such a weakness. As it was mentioned in previous chapters this work is not focused on carrying out the accurate optimizations.

Quantizer might be realized differently depending on the input-output width ratio.

If n is the input bit width and m is the output bit width then:

a) When $(n/m) > 1$, rounding is performed.

Rounding schematic being implemented in this case is shown in Figure 22:

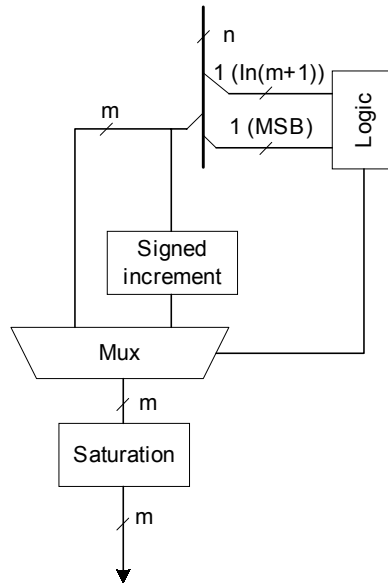


Figure 22. Quantizer. Rounding case

b) When $(n/m) < 1$, extension is performed.

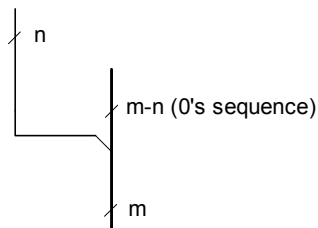


Figure 23. Quantizer. Extension case

Here n most significant bits are taken from the input, the less significant part is extended by zeros.

c) When $(n/m) = 1$, it is a pass through.

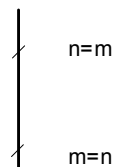


Figure 24. Quantizer. Pass through case

3.5 VHDL code structure.

System approach in design implies that system is decomposed into independent (or relatively independent) components that could be in turn developed separately.

VHDL language provides a special component approach. It means that device being designed can be figured as a set of logically independent blocks that further are joined together with special language means. Thus, developer obtain hierarchical structure of his project.

The FAD being designed can be hierarchically split in the following way (Figure 25):

Detailed input-output description for each component in realization is presented in Appendix A.

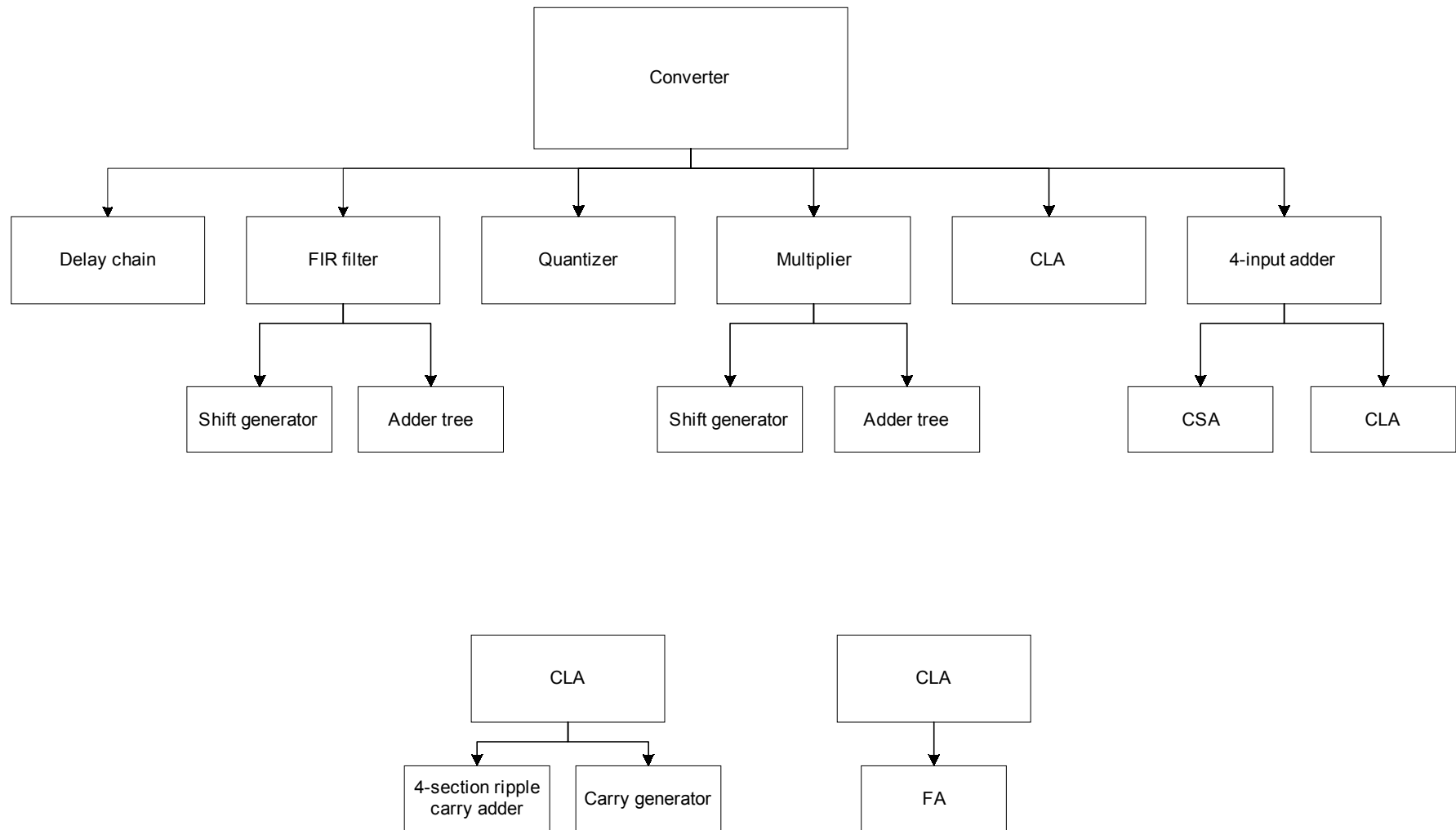


Figure 25. FAD VHDL level hierarchical diagram

3.6 Generator C code structure

Software VHDL component code generation is a process when target file is created, the necessary component functionality is put into it, obeying correct VHDL program structure. The standard C file input-output procedures are used to achieve this aim.

FAD code generator tool is built according to the system approach as a set of independent (loosely-coupled) procedures. This approach implies that each procedure can be written and debugged with individual test bench.

The final program consists of a set of several files (modules). Each file contains one or several procedures that intended for either certain component generation or supporting means like data reencoding, generation with templates etc.

Functional approach used in C language is quite old, however fast and effective if used in appropriate way. Data exchange between functions is only carried out by function parameters, no global variables are used for this. It also makes for the functions independency and increases code clarity.

Tool block schematic is shown in Figure 26:

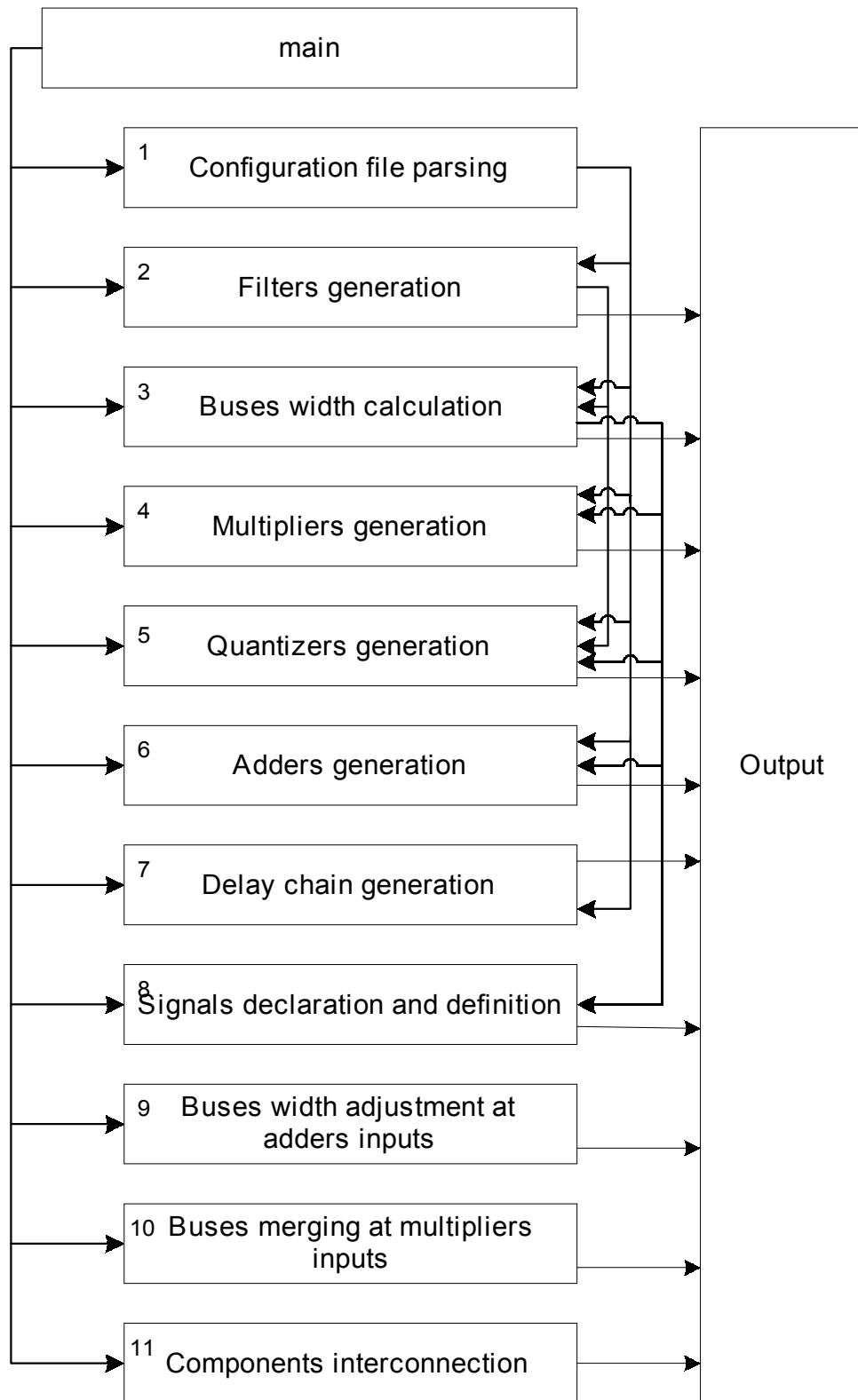


Figure 26. Tool block schematic

- Tool input information is taken from the configuration file written according to special simple syntax that will be carefully discussed in the next chapter. Procedure (1) that is responsible for that fills the internal structures with filter parameters information, multiplier coefficient values etc;
- Filters are generated only with information supplied in configuration file (procedure (2));
- Internal bus width depends only on the filter result width, quantizers parameters, adders input bus widths;
- Multiplier generation procedure requires that coefficient be given in input configuration file and bus widths be calculated at previous step;
- The quantizer input parameter that is taken from the configuration file is an output bus width while the input bus width is calculated. Number of quantizer channels is implied by the schematic to be implemented.
- The adder width is chosen as the broadest input width. Two guard bits are added inside the adder.
- Delay chain length is chosen so it can supply all the possible delays required for all the schematic and for each separate filter.

All the components are interconnected according to the schematic shown in the Figure 26. If FAD schematic should be adjusted, it would cause serious program code modification.

3.7 Configuration file description

Configuration file is the main (and generally the one) interaction means between a user and the tool. It should contain all the information about FAD that is necessary to successfully generate the schematic.

The sufficient information required (see figure 27):

1. Input data width;
2. Number of stages;
3. Input delay value;
4. Filter parameters:
 - 4.1. Number of stages;
 - 4.2. Multiplier coefficients for each stage;
5. Quantizers parameters;
6. Multipliers coefficients.

The logical information required:

1. Schematic name (to be used as a part of VHDL file name for the component);
2. Filter names.

Additional information contains details about filter realization.

Configuration file syntax gives an unambiguous interpretation of the information supplied. The information above may be put into the configuration description according the following rules (in Backus Naur notation (BNF)):

- 1) Configuration_file ::= <Header>
 {<Filter_description>}
 {<Multiplier description>}

2) Header ::= **converter** <space> <name> <space> <number>
word <space> <number>
[{<Additional param>}]
<Delays>

1-st line: converter name is supplied and FAD output bus width
(output quantizer parameter)

2-nd line: input bus width

3) Additional param:

<compensation>|<guardbits>|<pipelining>
<symmetry>|<vma>|<symmetrypipelinig>
<vmapipelining> “\n”

4) Delays ::= **delay** <space> {<number> <space>} <number>

Number of delays before each stage of FAD is supplied. All of
them must be entered.

5) Filter_description ::=

filter <space> <name> <space> <number>
<number>
{<coefficient> “\n”} <coefficient>

1-st line: filter name and filter output bus width (corresponding
quantizer parameter);

2-nd line: number of filter stages;

3-rd line starts a coefficient list. All of them must be included.

Number of coefficients must be equal to number of filter stages.

6) Multiplier description ::=

mul <space> <coefficient> <number>

Here the value of the coefficient is passed and the bus width after multiplier.

```
compensation ::= compensation <space> <Boolean>;  
guardbits ::= guardbits <space> <number>;  
pipelining ::= pipelining <space> <number>;
```

Compensation tells whether compensation vector should be added in the filter adder tree.

Guardbits points how many guard bits should be used in filter adder tree

Pipelining keyword precedes the number of pipeline stages in adder tree.

```
boolean ::= 0|1;  
number ::= {<digit>|<digit>};  
name ::= <letter> {<letter>|<digit>};  
coefficient ::= <float>;  
space ::= {“ “};
```

“digit” is a one-character digit;

“float” is a real number in any notation (including exponential);

“letter” is a single capital or lower-case latin letter.

Configuration file parser has a proof against the incorrect data typed by the user and strong syntax checking mechanism.

The sample configuration file is adduced in the Appendix B.

4 Design Example and Experimental Results

4.1 Introduction

In the previous chapters both the low area consuming architecture was introduced and the tool for producing it. This tool is used to generate the example of the design. This section shows the experimental results.

4.2 Experiments on Benchmark

For testing of the system (Figure 27) the testbench was created in MATLAB. The source signal is generated with special test vector generator component done in VHDL. Output response is saved and converted to MATLAB to be analyzed there.

The same structure in MATLAB was tested and showed the same results. The parameters of the Design example are shown in Appendix B.

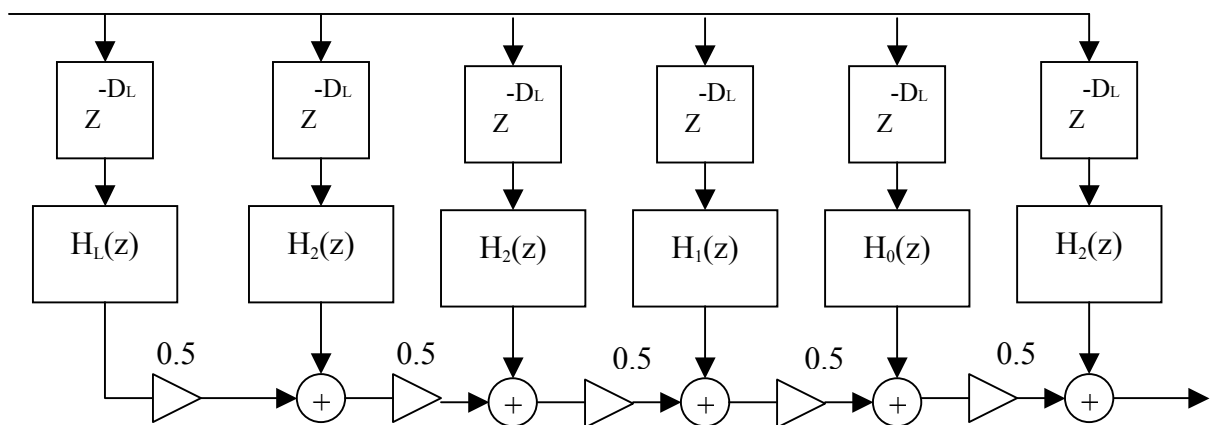


Figure 27. Design Example: Structure

The same input vector applied during VHDL and MATLAB modeling. The output impulse responses (Figure 28) were compared and found to be identical.

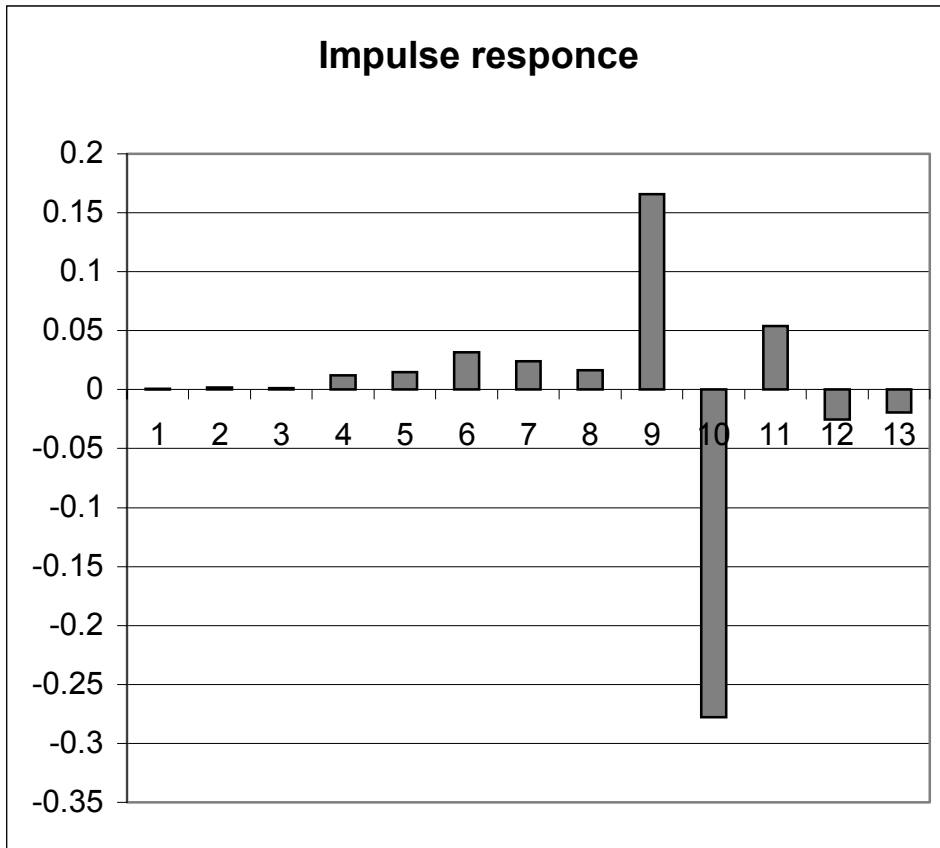


Figure 28. Design Example: Impulse response

The impulse response value shown below:

0.00061
0.001709
0.001343
0.012207
0.014893
0.03186
0.02417
0.016479
0.16577
-0.27783
0.053711
-0.02539
-0.01953

Three different examples were used as test information and results obtained proved the equality of the VHDL model output and MATLAB mathematical model.

In addition to simplify the testing for future work the internal behavioral simulator was created.

4.3 Internal behavioral simulator

Internal simulator emulates the single pulse of the highest positive value and applies it to the input of FAD. Such an input cause a certain output vector called impulse response. Vector length depends on the number of stages in filters the schematic is composed from and number of stages in FAD itself.

The comparison between the mathematical simulation results and test bench output gives a user an assurance that implemented schematic is correct or not.

The mathematical model looks like the following:

consider f_{ij} is a matrix, containing the FAD filters coefficients.

i – is a time instance within time period when FAD impulse response is at output.

$i=1..max[length_k+delay_k]$ for $k=1..m$,

where m is a number of FAD stages;

$length_k$ – number of stages for filter number k ;

$delay_k$ – delay value before the filter number k .

j – is a filter number being considered $j=1..m$

Matrix composition rule: column j is the list of the filter ' j ' coefficients. The list starts from the row number $delay_k$.

$$F_i = (\dots(((f_{i1} + f_{i2}) \cdot c_1 + f_{i3}) \cdot c_2 + f_{i4}) \cdot c_3 \dots) = \sum_{j=1}^{m-2} f_{i(j+1)} \cdot \prod_{k=j}^{m-2} c_k + f_{i1} \cdot \prod_{k=1}^{m-2} c_k + f_{im}$$

where F_i is the response value at i -th time instance ($i=1..\max[\text{length}_k+\text{delay}_k]$ for $k=1..m$).

To achieve the full accordance with VHDL model generated, it is necessary that rounding is introduced into the model in a way as it is performed in generated FAD.

Thus, the simulation algorithm is the following:

1. *Build the f_{ij} matrix according to the matrix composition rule above. The important think is that all the coefficients are rounded to the number of bits as implied by specification.*
2. *Loop for $i=1..N$. $N = \max[\text{length}_k+\text{delay}_k]$*

 - 2.1. *multiplicand= $f_{i1}+f_{i2}$;*
 - 2.2. *Loop for $j=3..M-1$. M is number of stages.*
 - 2.2.1. *multiplicand= $\text{round}(\text{multiplicand} \cdot c_{j-1}) + f_{ij}$;*
 - 2.3. *$F_i = \text{round}(\text{multiplicand} + f_{iM})$*

The result is a vector F. Simulation can be also useful during parameters selection and adjustment (for example quantizers width to achieve the proper accuracy). It is important that input parameters for the simulator are given by the same configuration file that makes the design cycle faster and decrease mistake probability (due to mistyping for example).

4.4 User manual guide

1. Locate the executable file (called ‘convgen’). All the generated VHDL code will be placed into the same directory as the program located.

2. Write the configuration file. Configuration file syntax is described in the respective chapter. Example of the configuration file could be found in Appendix B.

3. Place the configuration file into the same directory where the executable is.

4. Put the generator file into the execution giving the proper command line parameters. The list of possible parameters and their description is adduced in Appendix C.

5. If no simulation option passed, program produces a set of VHDL files representing HDL model for a FAD with specified parameters. If simulation mode is activated then simulation result file produced with a name specified as a parameter in command line together with generated VHDL code.

6. If it is needed, testbenching may be performed and compared with a simulation results.

5 Conclusion and future work

5.1 Conclusion

The suitable architecture with respect to low area cost has been chosen. The tool for automatic generation of the architecture with given parameters has been created. The testing of design generated by the tool was performed and proofed that generator works correctly.

5.2 Future work

This work implies using the multipliers with fixed coefficient. The variable coefficient multiplier may be introduced in order to increase the flexibility and get more functionality.

In order to be implemented more improvements and optimization must be done.

References

- [1] R. Adams, T. Kwan.: “A Stereo Asynchronous Digital Sample-Rate Converter for Digital Audio” IEEE Journal of solid-state circuits, vol. 29, NO. 4, pp 481 – 488, April 1994.
- [2] N. Aikawa, Y. Mori.: “Kernel with block structure for sampling rate converter” 0-7803-7663-3/03 2003 IEEE, ICASSP 2003, NO. VI pp 269 –272, 2003.
- [3] D. Babic, J. Vesma, T. Saramaki, M. Renfors.: “Implementation of the transposed Farrow structure” 0-7803-7448-7/02 2002 IEEE, NO. IV, pp 5-8, 2002.
- [4] H. Johansson, P. Lowenborg.: ”On the Design of adjustable Fractional Delay FIR Filters” IEEE, Transactions on circuits and systems – II: Analog and digital signal processing, vol. 50, No. 4, pp 164-169, 2003.
- [5] L. Wanhammar.: “DSP integrated circuits”, pp 468-470.
- [6] H. Ohlsson.: ”Studies on implementation of digital filters with high throughput and low power consumption”, Thesis 1031, LiU-Tec-Lic-2003:30, Linkoping University, 2003 p. 13.

Abbreviations

AT – adder tree

BNF – Backus-Naur form

CLA – carry look-ahead adder

CSA – carry save adder

CSDC – canonic signed digit form

FAD – filter with adjustable delay

FIR – finite impulse response

MSB – most significant bit

PP – partition product

PPG – partial product generator

SFG – signal flow graph

Appendix A VHDL components interface description

Converter component

```
entity converter_<converter_name>_<number> is
port (
  indata : in std_logic_vector((12-1) downto 0);
  ys : out std_logic_vector((41-1) downto 0);
  yc : out std_logic;
  reset : in std_logic;
  clk : in std_logic);
end converter_<converter_name>_<number>;
```

indata – input data to converter. Its width depends on the configuration file settings;

ys, yc – output result and output carry out;

clk, reset – clock and reset signals;

Carry lookahead adder

```
entity cla<width>_<converter_name>_0 is
port(
  in_a: in std_logic_vector(23 downto 0);
  in_b: in std_logic_vector(23 downto 0);
  c0: in std_logic;
  sout: out std_logic_vector(23 downto 0);
  cout: out std_logic);
end cla<width>_<converter_name>_0;
```

CLA width is used as a part of the component name and file name as well.

in_a, in_b – are inputs of CLA;
c0 – carry in;
sout, cout – output result and carry signal.

Carry save adder

```
entity CSA_<width> is
port ( in1, in2, in3: in std_logic_vector(20 downto 0);
       outc, outs: out std_logic_vector(20 downto 0));

end CSA_<width>;
```

CSA width is used as a part of the component name and file name as well.

in1...in3 – are input of adder;
outs, outc – outputs.

Four input adder

```
entity FourXAdd_<name>_0 is
port(
in1: in    std_logic_vector(30 downto 0);
in2: in    std_logic_vector(30 downto 0);
in3: in    std_logic_vector(30 downto 0);
in4: in    std_logic_vector(30 downto 0);
ys: out    std_logic_vector(31 downto 0);
yc: out    std_logic);
end FourXAdd_<name>_0;
```

This is an adder which produces sum for four items.

in1..in4 – are inputs of adder;
ys, yc – result of the addition.

Multiplier

```
entity mul_<convertor_name>_0 is  
port(  
  indata: in  std_logic_vector(21 downto 0);  
  ys: out   std_logic_vector(30 downto 0);  
  yc: out   std_logic_vector(30 downto 0);  
  clk: in   std_logic);  
end mul_<convertor_name>_0;
```

indata – number in parallel cod to be multiplied by fixed coefficient;

ys, yc – output result and carry signal.

Quantizer

2 channel:

```
entity rounder<num_of_chan>_<inp_width>x<out_width> is  
port(  
  inp1: in  std_logic_vector(13 downto 0);  
  inp2: in  std_logic_vector(13 downto 0);  
  ys: out   std_logic_vector(12 downto 0);  
  yc: out   std_logic_vector(12 downto 0));  
end rounder<num_of_chan>_<inp_width>x<out_width>;
```

Number of channels is included in component name and the file name as well. It affects the interface as well as the

functionality. File name also built from the input buses width and the output width, required by the specification.

inp1, inp2 – two independent channel inputs;

ys, yc – two independent channel outputs with required width.

1 channel:

```
entity rounder<num_of_chan>_<inp_width>x<out_width> is
port(
inp1: in    std_logic_vector(16 downto 0);
ys: out    std_logic_vector(15 downto 0));
end rounder<num_of_chan>_<inp_width>x<out_width>;
```

1 channel quantizer has the same name convention and interface as the 2 channel one.

FIR filter

```
entity connect_<name> is
port (
indata : in std_logic_array_delay_<name>;
yc, ys : out std_logic_vector((21-1) downto 0);
clk   : in std_logic);
end connect_<name>;
```

Component (file) name includes name of a filter specified in the configuration file.

indata – filter input in special format:

```
STD_LOGIC_ARRAY_DELAY_FILE IS
    ARRAY (<> DOWNTO 0) OF STD_LOGIC_VECTOR (<> DOWNTO 0);
```

In fact, it is the array of buses;

ys, yc – filter outputs;

clk – clock signal (meaningful if only tree is divided by pipeline registers);

Delay chain

```
entity delay_chain_<convertor_name>_0 is  
port (  
  a   : in std_logic_vector((12-1) downto 0);  
  y   : out std_logic_array_commondelay_fedor;  
  clk : in std_logic);  
end delay_chain_<name>_0;
```

a – input signal in parallel code;

y – output array containing input signal values delayed from 0 to maximum possible value.

Output type looks like the following:

```
STD_LOGIC_ARRAY_COMMONDELAY_FEDOR IS  
  ARRAY (<> DOWNTO 0) OF STD_LOGIC_VECTOR(<> DOWNTO 0);
```

clk – clock signal at which (rising edge) delay chain shift occurred.

Shift generator

```
entity shift_gen_<name>_0 is
port (
a   : in std_logic_vector(12-1 downto 0);
y   : out std_logic_array_shift_<name>;
clk : in std_logic);
end shift_gen_<name>_0;
```

It is a supporting component which generates partial products for multiplication. The rules how it composes the result are hardly coded in component code. FAD VHDL implementation contains many shift generators for different coefficients required. Component name includes <name> field that defines which component is the “owner” of shifter.

a – input value in parallel code;

y – output set of partial products. The type declaration is similar to the type, described in the filter section;

clk – clock signal.

Adder tree

```
entity tree_<name>_0 is
port (
clk: in std_logic;
in_0: in std_logic_array_shift_fill;
in_1: in std_logic_array_shift_fill;
in_2: in std_logic_array_shift_fill;
in_3: in std_logic_array_shift_fill;
in_4: in std_logic_array_shift_fill;
```

```
in_5: in std_logic_array_shift_fill;  
in_6: in std_logic_array_shift_fill;  
in_7: in std_logic_array_shift_fill;  
outc, outs: out std_logic_vector(21-1 downto 0));  
end tree_<name>_0;
```

Number of inputs depends on the number of partial products to be added.

in_0...in_7 – inputs to be added;

outs, outc – sum result;

clk – clock signal. Meaningful if only tree is pipelined.

Appendix B Configuration file example

converter fedor 16

word 12

5

delay 0 1 2 3 4

filter fil1 12

8

0.001953125

0.015625

-0.06640625

0.3046875

-0.3046875

0.06640625

-0.015625

-0.001953125

filter fil2 12

8

0.001953125

0.015625

-0.06640625

0.3046875

-0.3046875

0.06640625

-0.015625

-0.001953125

filter fil3 12
8
0.001953125
0.015625
-0.06640625
0.3046875
-0.3046875
0.06640625
-0.015625
-0.001953125

filter fil4 12
8
0.001953125
0.015625
-0.06640625
0.3046875
-0.3046875
0.06640625
-0.015625
-0.001953125

filter fil5 12
8
0.001953125
0.015625
-0.06640625
0.3046875
-0.3046875
0.06640625

-0.015625

-0.001953125

mul 0.5 13

mul 0.5 13

mul 0.5 14

mul 0.5 15

Appendix C Command line parameters

The only parameter that cannot be omitted is the name of a configuration file

The evoking syntax in BNF notation is the following:

```
<program name> <config file name> {[-m] | [-t name] | [-s name]}
```

The additional parameters can be given in any order after the configuration file name.

-m Creates makefile for VCOM. If such a parameter is given, the tool creates file called 'makefile', containing all the components of ready FAD, enumerated in proper order. It allows removing the interdependency problem during compilation the VHDL code. VCOM compiler should be evoked in the following way: *vcom -f makefile*.

-t Defines name used as a key part of converter type file name. If the option is not given, 'noname' is used as a part of typefile.

-s Simulates and writes simulation results into file with filename supplied. Simulation results are just a set of strings like the following:

0000000000000100	0.000122
0000000000101000	0.001221
1111111111001000	-0.001709
0000001000000000	0.015625
0000000110010000	0.012207
0000001100101000	0.024658
0000001000110000	0.017090
0001010101011100	0.166870
1101110010111000	-0.275635
0000011101110000	0.058105
1111110111100000	-0.016602
1111111111000000	-0.001953

Results are provided both in 2's complement binary notation and in decimal floating point form to simplify comparing with VHDL simulation in VSIM or other HDL simulator and design information.

På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: <http://www.ep.liu.se/>