NTNU – Trondheim
Norwegian University of
Science and Technology

Channel Filter Cross-Layer Optimization

Joar Nikolai Talstad

Master of Science in Electronics
Submission date: June 2015
Supervisor: Snorre Aunet, IET
Co-supervisor: Isael Diaz, Nordic Semiconductor
Jan Egil Øye, Nordic Semiconductor

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Title:	Channel Filter Cross-Layer Optimization
Student:	Joar Nikolai Talstad

Problem description:

The objective of this project is to analyze a channel filter in a number of design-dimensions, in order to arrive to an ultimate solution that is re-configurable and can potentially have low energy consumption and good performance at the cost of moderate complexity increase. The goal is the final implementation of a channel filter that compared to state-of-the-art in literature: consumes less energy, presents the same performance, does not occupy more than 30% additional footprint. The configuration time should not be longer than 2 clock cycles.

The methodology to be used consists of running multiple times a short design cycle with initial constraints. A candidate implementation is generated at the end of every cycle and saved for posterior evaluation. A comparison between the various implementation candidates is done by comparing each solution in terms of cost and performance. Finally, a number of candidates that best suit the initial constraints are merged into a single reconfigurable architecture. In this manner the final solution can adapt itself to the computational requirements in real time.

Responsible professor:	Snorre Aunet, IET
Supervisors:	Jan Egil Øye, Nordic Semiconductor
	Isael Diaz, Nordic Semiconductor

Abstract

The recent raise of Internet of Things has increased the demand of energy-efficient wireless devices. However, the design process of a low-energy, high-performance device for all operational cases is not trivial. Thus, in this thesis a cross layer optimization technique called algorithm-architecture co-design is used to optimize one of the most critical DSP blocks in any communication device, namely, the channel filter.

In order to effectively trade between similar RTL designs in terms of area and power dissipation, a fully automated tool-flow is created which performs RTL-simulation, synthesis, layout and power analysis. The tool-flow provides the results of a 500 gate design in less than 5 minutes, running on a computer of the current industry standard, and is considered to be very accurate based on the results of a previous study.

A digital low pass filter is first optimized through a constructed filter sorting algorithm. It generates a large number of theoretical filter solutions and sorts them based on how eligible they are for hardware implementation. The algorithm is made generic, and hence applicable for any filter requirement, and proves to find the most energy-efficient solution.

The hardware architecture of the most effective filter implementation is then thoroughly analyzed in two stages. Firstly, in order to find the filter's most effective quantization levels, and secondly, in order to make the architecture dynamic with regards to filter performance and power dissipation. In the latter, two main approaches are proposed.

The first approach adapts the filter order, and hence the stopband attenuation, according to the quality of the radio link. The best implementation of this approach manages to reduce the power dissipation of 28%, 55% and 88% for the constructed low power modes, with an increase of only 8% in area compared to the non-dynamic implementation.

The second approach adapts the quantization level, and hence the amount of noise introduced by the filter, according to the radio link. Here, the best implementation reduces the power dissipation of 11%, 32% and 81% for the low power modes, while increasing the area of only 18%.

Sammendrag

Konseptet Internet of Things har de siste årene økt etterspørselen av energieffektive, trådløse enheter. Å utvikle systemer som tilbyr lavt effektforbruk og høy ytelse for ethvert funksjonsområde er imidlertid ingen triviell oppgave. Denne oppgaven anvender derfor en kryssoptimaliseringsteknikk kalt algorithm-architecture co-design for å optimalisere en av de mest kritiske DSP-blokkene i ethvert kommunikasjons-system, nemlig kanalfilteret.

For å effektivt kunne veie like RTL design opp mot hverandre når det gjelder areal og effekttap, er det laget en automatisk verktøy-flyt som utfører RTL-simulering, syntese, layout og power analyse. Verktøy-flyten gir resultater fra et 500 gate design på mindre enn 5 minutter når den kjører på en datamaskin av dagens industristandard, og anses å være nøyaktig basert på resultater fra en tidligere studie.

Et digitalt lavpass-filter er først optimalisert gjennom en konstruert filtersorterings-algoritme. Den genererer et stort antall teoretiske filterløsninger og sorterer dem basert på hvor egnet de er for hardware-implementasjon. Algoritmen er generisk, og derfor anvendelig for alle filter-spesifikasjoner, og viser seg å finne den mest energieffektive løsningen.

Hardware-arkitekturen til den mest effektive filter-implementasjonen er deretter nøye analysert i to steg. Først for å finne filterets mest effektive kvantiseringsnivå, og deretter for å gjøre arkitekturen dynamisk med hensyn til filter-ytelse og effekttap. I det sistnevnte er det foreslått to hovedtilnærminger.

Den første tilnærmingen tilpasser filterets orden, og dermed dempningen i stopp-bånd, i henhold til kvaliteten på radioforbindelsen. Den beste implementasjonen av denne tilnærmingen reduserer effekttapet med 28%, 55% og 88% for de konstruerte laveffekt-modusene, med en økning i areal på bare 8% sammenlignet med den ikke-dynamiske implementasjonen.

Den andre tilnærmingen tilpasser kvantiseringsnivået, og dermed mengden støy introdusert av filteret, i henhold til radioforbindelsen. Her gir den beste implementasjonen en reduksjon i effekttap på 11%, 32% og 81% for de forskjellige laveffekt-modusene, med en økning på bare 18% i areal.

Preface

This thesis completes a Master of Science degree in Electronics, Design of Digital Systems, submitted to the Department of Electronics and Telecommunications at the Norwegian University of Science and Technology. The assignment was given by Nordic Semiconductor in January 2015, and the work was completed in June the same year.

Working with this thesis has been very interesting, but also challenging and time-consuming, as it required me to familiarize with a lot of different CAD tools, as well as theory regarding digital signal processing. The work has given me insight in modern IC development, and practice in designing and implementing low-power oriented systems, which I think will be useful in the future.

Firstly, I would like to thank my supervisor Professor Snorre Aunet at NTNU for his help and guidance throughout this last semester. I would also like to thank my supervisors at Nordic Semiconductor, Jan Egil Øye and Isael Diaz. Your continuous feedback throughout this process has been invaluable for me, and for that I am very grateful. Finally, I wish to thank my family for their support, and my fiancé Sissel Klakegg for all her love and encouragement over these 5 years.

Trondheim, 2015-6-10

Joar Nikolai Talstad

Contents

List of Figures

List of Tables

List of Acronyms

AFGEC Automated Filter Generation and Eligibility Calculation.

CMOS Complementary Metal-Oxide-Semiconductor.

CSV Comma-Separated Values.

DSP Digital Signal Processing.

FDA Filter Design and Analysis.

FIR Finite Impulse Response.

FOS First Order Section.

IIR Infinite Impulse Response.

IoT Internet of Things.

IP Intellectual Property.

RTL Register-Transfer Level.

SAIF Switching Activity Interchange Format.

SOS Second Order Section.

SQNR Signal-to-Quantization-Noise Ratio.

VCD Value Change Dump.

Chapter 1

Introduction

Energy efficiency has become one of the most important aspect in today's wireless technology, with increasing demands in both performance and battery life time. With the Internet of Things (IoT) at its staring point, which is expected to reach 50 billion connected devices over the next decade [5], it is more relevant than ever to come up with low power design methodologies that prolong the battery life time of wireless communicating devices. Sleep modes are used to shut down parts of the circuit whenever they are unused. But as important as saving power in sleep mode, is to save power when the device is computing.

1.1 Motivation

When designing an Intellectual Property (IP) in the industry, there is usually not time nor resources to compare tens or hundreds of different implementations in order to find the most eligible one. Thus, several sub-optimal solutions will be implemented. This thesis addresses this problem by taking a closer look at one of the IPs that may be implemented in numerous ways, namely the channel filter, in order to find an optimized implementation of the filter. The computational IPs are often implemented as *static*, in the meaning of providing the same performance at all times. The IP then needs to be able to provide the best level of performance for the worst case scenario, which implies that it will be overqualified in most cases. However, by making the IP *dynamic* it may adapt its performance to the computational requirements. This will require a slight increase in the amount of logic, but may potentially reduce the average power dissipation in the typical case.

1.2 Algorithm-architecture co-design

This thesis will explore a methodology called *algorithm-architecture co-design* which is described in the doctoral dissertation by Isael Diaz [4]. It states that "the best implementation for any application requires a fine tuning between both algorithm and architecture." Diaz explains that since design constraints vary considerably among different applications, a general abstraction model will result in sub-optimal implementations. Thus, he writes, "efficient implementations can only be realized if both algorithm and architecture are prepared towards a common goal."

The common goal in this work is to reduce the power dissipation of the channel filter, without increasing the area more than 30% and while maintaining the performance of the filter at an acceptable level. This thesis will try to achieve the optimal solution by addressing the algorithm and architecture aspects in the following way:

Algorithm optimization

Minimal power dissipation can be obtained by finding the simplest digital signal processing algorithm which provides the desired filter performance. Investigating different filter types and structures may then lead to find the implementation which is most eligible for hardware implementation with regards to area and consequently power dissipation. Some filtering algorithms may also be better suited for a dynamic implementation, and should therefore be prioritized.

Architecture optimization

Investigating the architecture of the hardware implementation may lead to finding possible simplifications. For instance, parts of the circuit may be bypassed in order to reduce the power dissipation, although this will degrade the performance of the filter and increase the total area due to additional control logic. Such trade-offs will be examined in order to find the optimal solution in terms of power dissipation.

1.3 Previous work

In the author's specialization project [14], the accuracy of two early stage power estimation methods is explored. The first method takes use of the commercial Design-for-Power tool PowerArtist by Ansys Apache, which estimates power dissipation from Register-Transfer Level (RTL). The tool yields promising results and accuracy, but its license is not available for the time being. The second method annotates switching activity from RTL simulation on post-layout netlists in Synopsys' PrimeTime-PX. This method proves to detect the power states of a multi-voltage full-chip design, and estimates its power dissipation within 1% deviation for low activity scenarios and within 13% deviation for high activity scenarios, using a sign-off power analysis

as reference. The specialization project report concludes that this method is well suited for making RTL-design tradeoffs based on power dissipation in multi-voltage designs, but that it will require a custom-made automated tool-flow. The automated tool-flow should first perform a simplified synthesis and layout of an RTL-design in order to generate a post-layout netlist, and then perform the power analysis in PrimeTime-PX which applies activity files from RTL.

1.4 Objectives

The work to be done can be split into the following objectives:

1. Create an automated tool-flow based on the previous work, which enables an efficient work-flow when making tradeoffs among a set of different RTL implementations. The tool-flow should create a scorefile for each implementation which extract details regarding area and power from the layout reports and power estimation reports.
2. Create a script that generates a large number of theoretical filters and sorts them based on hardware implementation friendliness.
3. Implement the highest scoring filters in Objective 2 in RTL, run them through the tool-flow in Objective 1, and verify or disprove the prediction. Settle for one implementation that will be used for further investigation.
4. Investigate the effect of quantizing at different levels. Make several RTL-implementations and verify using the tool-flow in Objective 1. Find the optimal quantization level in the tradeoff between quantization noise and area/power.
5. Investigate different approaches in making the channel filter dynamic. Implement in RTL and analyze the gain versus overhead introduced using the tool-flow in Objective 1.

The order and interrelation of the objectives are visualized in the master thesis workflow, in Figure 1.1.

1.5 Thesis overview

Chapter 2 presents the theory regarding digital filters, power dissipation, quantization noise and commonly used low-power design techniques, which is considered useful material for the reader.

Chapter 3 describes the automated area and power estimating tool-flow. Chapter 4 describes the automated filter generation and eligibility calculation algorithm, along

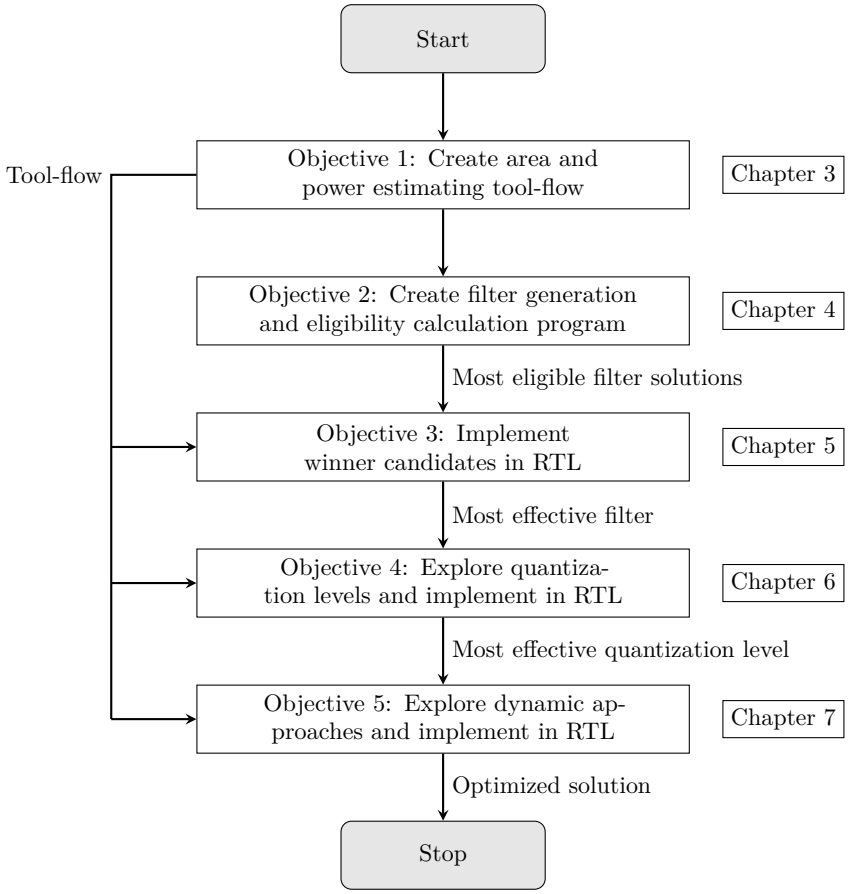


Figure 1.1: The master thesis workflow

with the results and conclusion with the most eligible solutions. Chapter 5 describes the RTL implementation of the winner candidates, and presents their resulting area and power dissipation. The results are compared to the predictions in Chapter 4, and a conclusion is made where the most efficient filter is found. Chapter 6 carries out a quantization noise analysis of the filter, and presents the results of different quantization level implementations. Chapter 7 presents two main approaches in making the filter dynamic. Several implementations are made for each approach, which are analyzed with regards to area, power and performance.

Chapter 8 discusses the overall result, and takes a look at the methodology from a more generic point of view. Finally, Chapter 9 presents the overall conclusions and lists the objectives for future work.

Chapter 2

Background and Theory

This chapter presents the material needed in order to understand how power is dissipated in CMOS circuits, some basic low-power design techniques, and some digital signal processing theory regarding digital filters.

2.1 The channel filter in wireless communication systems

An illustration of a basic wireless communication system is shown in Figure 2.1 [9]. Note that this is an abstracted model, and does not necessarily represent the system mentioned in this thesis.

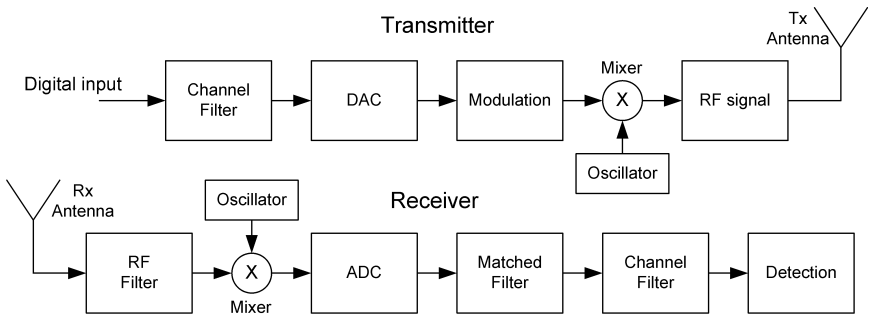


Figure 2.1: Basic transmit-receive procedures in digital communications

In such systems, a *channel filter* is typically placed in both the transmitter and the receiver, filtering the information in baseband. This is done in transmission to avoid leaking energy into the neighboring bands, and in reception to reject interfering signals outside the operating band, and unwanted products from mixers and amplifiers [10]. This is typically done in the digital domain, which is the case that will be explored in this thesis.

2.2 Digital filters

A digital filter is a Digital Signal Processing (DSP) system which alters the spectral content of an input signal in order to remove noise, improve signal quality or extract signal information, to mention a few common objectives [7]. DSP systems are defined as an implemented algorithm which performs a certain operation on digital signals in order to achieve a predefined task. The processing of digital signals, and hence digital filters, can be described by additions, multiplications and time shifts, using the building blocks in Figure 2.2 [7]. A digital filter can be implemented in software or hardware using these building blocks.

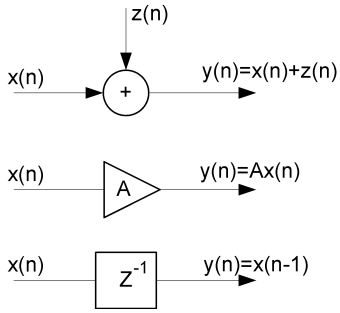


Figure 2.2: Building blocks of DSP systems: Adder, multiplier and delay element

Filter representations

Digital filters may be represented in several ways, and one of them is the I/O equation, or difference equation. The general constant-coefficient difference equation of discrete-time systems is defined as [11]:

$$y(n) = -\sum_{k=1}^{N} a_k y(n-k) + \sum_{k=0}^{M} b_k x(n-k) \tag{2.1}$$

where $x(n)$ is the input, $y(n)$ is the output, and a_k and b_k are the coefficients of the system. Here, the larger number of M and N is called the *order* of the system,

although N is usually selected to equal or exceed M. Another way of representing digital filters is by its system function [11]:

$$H(z) = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 + \sum_{k=1}^{N} a_k z^{-k}} \tag{2.2}$$

which is the ratio of the system input and output after performing the z-transform. The system function is defined by the coefficients a_k and b_k, which determines the poles and zeros of the system, and thereby its frequency response [11]. The impulse response of a digital filter can be obtained by applying the unit impulse function,

$$\delta(n) = \begin{cases} 1 & , n = 0 \\ 0 & , n \neq 0 \end{cases} \tag{2.3}$$

to the input of the system. Hence, by substituting $x(n)$ with $\delta(n)$ in Equation 2.1, the equation for the impulse response is found. Digital filters can be divided into two categories based on their impulse response. These are, Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters.

2.2.1 FIR filters

FIR filters are defined exclusively by the input $x(n)$ and the coefficients b_k, as the output $y(n)$ do not depend on previous output values. Hence, the I/O equation of FIR filters can be defined by:

$$y(n) = \sum_{k=0}^{M} b_k x(n-k) \tag{2.4}$$

The impulse response of an FIR filter can, as the name implies, be represented by a finite number of samples. These samples are identical to the coefficients of the filter, as can be verified by inserting $x(n) = \delta(n)$ into Equation 2.4:

$$y(n) = \sum_{k=0}^{M} b_k \tag{2.5}$$

The simplest structure for implementing an FIR filter, is called the direct-form. This realization requires $M - 1$ memory elements, M multiplications and $M - 1$ additions, which is illustated in Figure 2.3 [11].

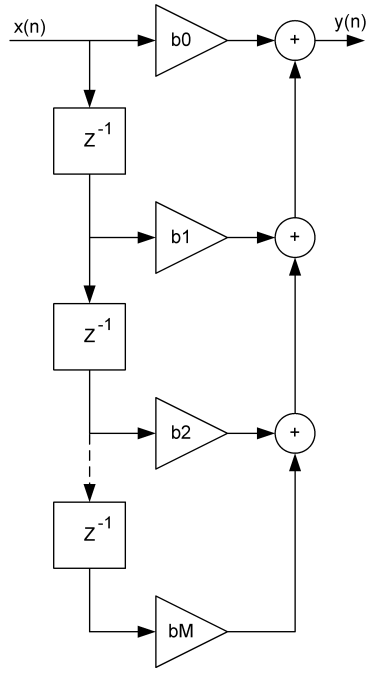


Figure 2.3: Direct Form realization of an FIR filter

The finite length of the FIR filter impulse response ensures that they are stable [7]. And when the system has linear phase, the coefficients also are symmetrical, which makes it possible to reuse half the coefficients. This may reduce the number of multiplications to $M/2$ [11]. However, the disadvantage with FIR filters is that they require a relatively high order (number of coefficients) in order to obtain steep curves in the frequency response, which implies a high complexity [7].

2.2.2 IIR filters

If the impulse response of a filter is not of finite length, it is called an IIR filter [7]. An easy way to recognize an IIR filter is that the output of the filter not only depends on previous input values, but also on previous output values, which creates feedback loops in the system. The I/O equation and system function of an IIR filter can be described by Equation 2.1 and 2.2, respectively.

There are several ways of realizing IIR filters. The Direct Form structures can be realized by splitting the system function in Equation 2.2 into an all-zero filter and

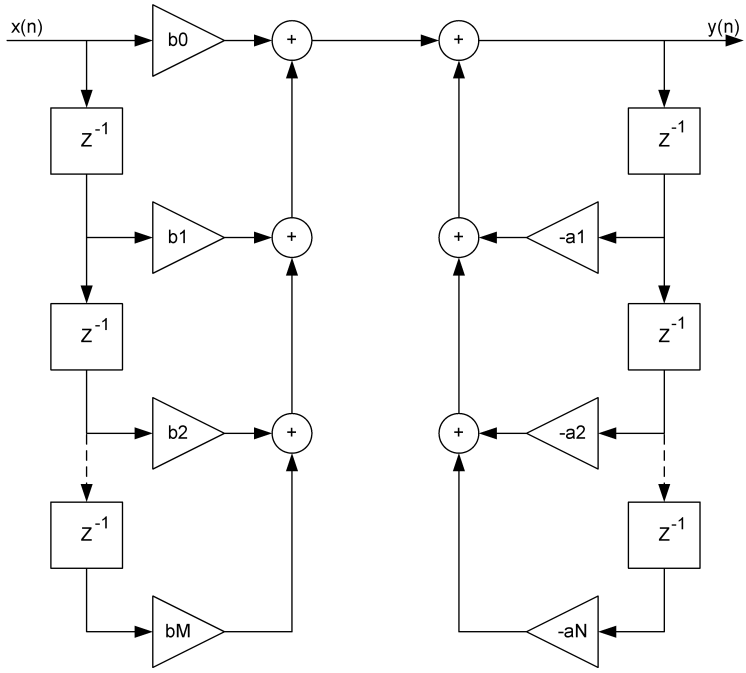


Figure 2.4: Direct Form I realization of an IIR filter

an all-pole filter, as follows [11]:

$$H(z) = H_1(z)H_2(z) \tag{2.6}$$

where

$$H_1(z) = \sum_{k=0}^{M} b_k z^{-k} \tag{2.7}$$

and

$$H_2(z) = \frac{1}{1 + \sum_{k=1}^{N} a_k z^{-k}} \tag{2.8}$$

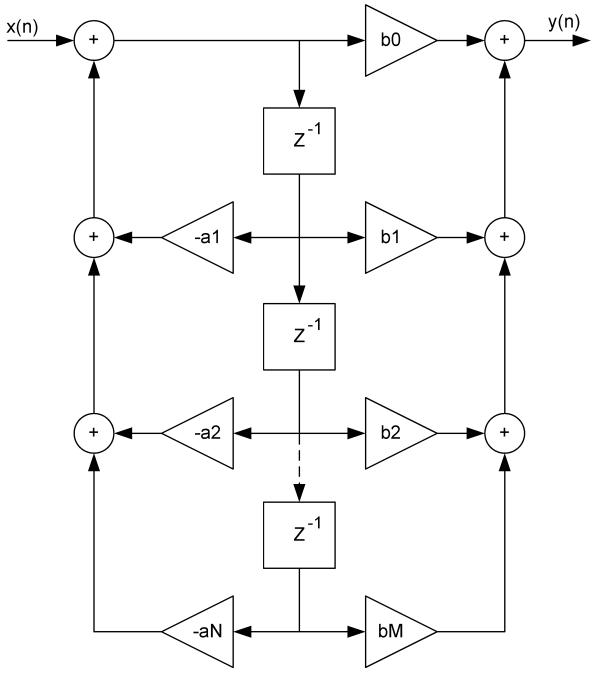


Figure 2.5: Direct Form II realization of an IIR filter

The Direct Form I structure

The Direct Form I structure, illustrated in Figure 2.4, is realized by first implementing the all-pole filter $H_1(z)$ followed by the all-zeros filter $H_2(z)$. Note that if the a_k coefficients are zero, this becomes an FIR filter. This structure requires $M + N + 1$ multiplications, $M + N$ additions and $M + N$ memory elements.

The Direct Form II structure

The Direct Form II structure, illustrated in Figure 2.5, is obtained by placing the all-pole filter in front of the all-zero filter. This is a more compact structure, as the memory elements may be reused. Consequently, it requires a maximum of $M + N + 1$ multiplications, $M + N$ additions, and $max(M, N)$ memory elements. Unfortunately, the Direct Form structures are extremely sensitive to quantization and are not recommended in practical applications, as small changes in the filter coefficients results in large changes in the placement of poles and zeros in the system if N is large [11]. To reduce this effect, high-order IIR filters are often factored into first- and second-order sections, or *biquadratic sections*, which are connected in cascade to form the overall filter [7].

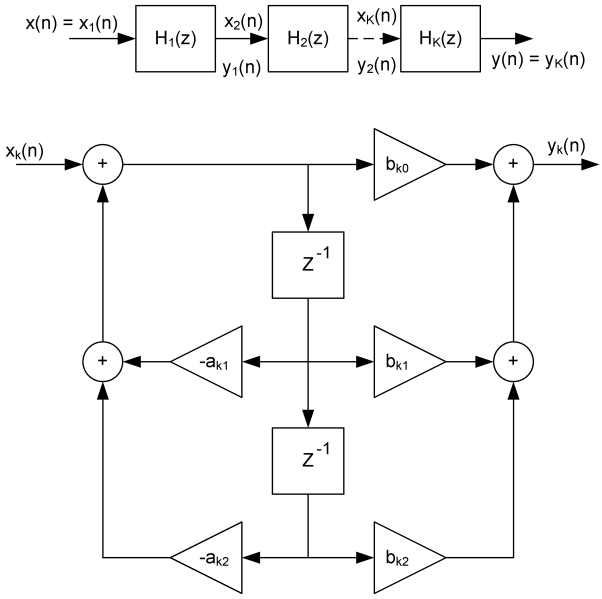


Figure 2.6: Cascade-Form realization of an IIR filter, with generic Second-Order Section

The Cascade-Form structure

The Cascade-Form structure of $H(z)$ can be expressed as

$$H(z) = \prod_{k=1}^{K} H_k(z) \tag{2.9}$$

where

$$K = \lceil \frac{N+1}{2} \rceil \tag{2.10}$$

and

$$H_k(z) = \frac{b_{k0} + b_{k1}z^{-1} + b_{k2}z^{-2}}{1 + a_{k1}z^{-1} + a_{k2}z^{-2}} \tag{2.11}$$

If $N > M$, some of the second-order sections will have b-coefficients that are zero. Also, if N is odd, one of the sections will have $a_{k2} = 0$ in order to form a first-order

section. Each of the sections defined by Equation 2.11 can be realized as either Direct Form I or II. There are also many ways of pairing the poles and zeros when factoring into second-order sections, and ordering the sections. For infinite-precision arithmetic all implementations are equivalent, but for practical applications the various implementations may differ significantly [11]. The Direct Form II realization of a cascaded second-order section is illustrated in Figure 2.6.

Commonly Used IIR Filters

When the structure of the filter is selected, the values of the coefficients need to be determined. This implies determining the poles and zeros of the filter, which is a well-developed field in analog filter design. There also exist several computer programs for designing digital filters. One example is the Filter Design and Analysis (FDA) tool in Matlab, which calculates the coefficients based on a set of design constraints and specifications. However, the tool needs to know which filter to generate, and this section presents the characteristics of the three most commonly used IIR filters:

Butterworth filters are all-pole filters where all the poles occur on a circle of a given radius, at equally spaced points. The frequency response of Butterworth filters are monotonic in both passband and stopband [11].

Chebyshev Type I filters are all-pole filters that have equiripple behaviour in passband, and monotonic behaviour in stopband. The poles is placed on an ellipse, and is easiest determined by first finding the points of an equivalent order Butterworth filter. The details of this can be found in [11].

Elliptic filters contain both poles and zeros, and have equiripple behaviour in both passband and stopband. The poles are placed according to the Jacobian elliptic function, and the zeros are placed on the imaginary axis. The Elliptic filters manages to spread the approximation error equally over the passband and stopband, and are therefore considered the most efficient from the view of yielding the smallest order filter for a given specification. A disadvantage with the Elliptic filter compared to Butterworth and Chebyshev, is that its phase response is more nonlinear in the passband [11].

2.3 Quantization noise

When performing fixed-point multiplications, you usually need to quantize a number via rounding or truncation in order to keep the same word length, which degrades the precision level of the signal [11]. The consequence of rounding and truncation is that a quantization error, the difference between the number prior and after quantization,

is introduced to the signal. The characteristics of this error depends on the number representation of the signal. We will focus on the *two's-complement* representation.

2.3.1 Truncation

For positive numbers, truncation results in a number that is smaller than the original. Hence, the quantization error E_t is within the region $-(2^{-b_t} - 2^{-b}) \leq E_t \leq 0$, where b and b_t are the number of bits before and after truncation, respectively. In two's complement, truncating a negative number increases the magnitude of the negative number. Hence, the truncation error is always negative, and falls within the region $-(2^{-b_t} - 2^{-b}) \leq E_t \leq 0$ [11]. The relationship between the original signal x, and the truncated signal $Q_t(x)$ is illustrated in Figure 2.7, where the quantization error is defined as $E_t = x - Q_t(x)$.

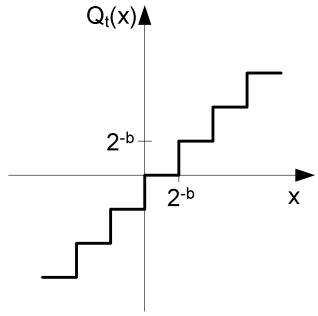


Figure 2.7: Relationship between original and truncated signal

If the quantization error is uniformly distributed in the range $-\Delta \leq E \leq 0$, the mean value of the error is $-\frac{\Delta}{2}$ and the quantization noise power is $\frac{\Delta^2}{3}$.

2.3.2 Rounding

In rounding, the maximum error that can be introduced is $\frac{1}{2}(2^{-b_r} - 2^{-b})$, where b_r is the number of bits after rounding. The round-off error can be either positive or negative, and falls within the region $-\frac{1}{2}(2^{-b_r} - 2^{-b}) \leq E_r \leq \frac{1}{2}(2^{-b_r} - 2^{-b})$. Thus the quantization error of rounding is symmetrical about zero, as illustrated in Figure 2.8. Here, the quantization error is defined as $E_r = x - Q_r(x)$, where x is the original signal and $Q_r(x)$ is the rounded signal.

If the quantization error is uniformly distributed in the range $-\frac{\Delta}{2} \leq E \leq \frac{\Delta}{2}$, the mean value of the error is zero and the quantization noise power is $\frac{\Delta^2}{12}$ [11].

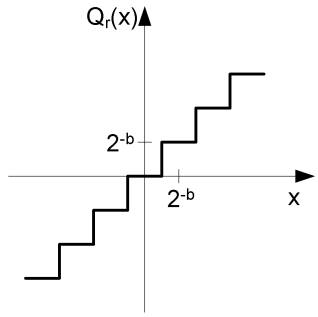


Figure 2.8: Relationship between original and rounded signal

2.4 CMOS Power Dissipation

The power dissipation of digital Complementary Metal-Oxide-Semiconductor (CMOS) circuits can be categorized as shown in Figure 2.9. The following sections will explain each component in detail, and present the resulting equation.

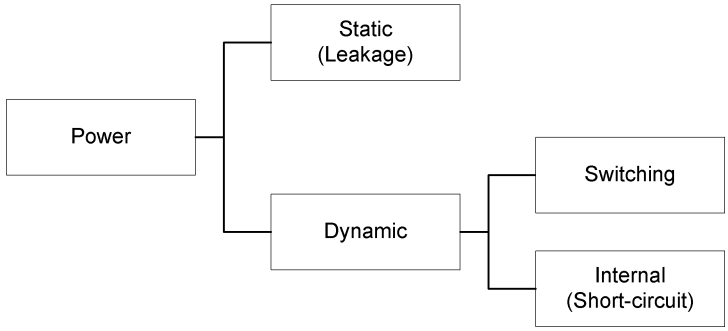


Figure 2.9: CMOS power dissipation categories

2.4.1 Static Power

The power dissipated by a gate when it is inactive, is called *static power* or *leakage power*. This is due to the fact that all static power components are caused by leakage currents. Most of the static power dissipation is due to the source-to-drain subthreshold leakage I_{sb}, which is caused by reduced threshold voltages that prevent the gate from turning off completely. The other component results from current leaking between the diffusion layers and substrate [12]. The leakage currents are denoted I_{lk} in Figure 2.10.

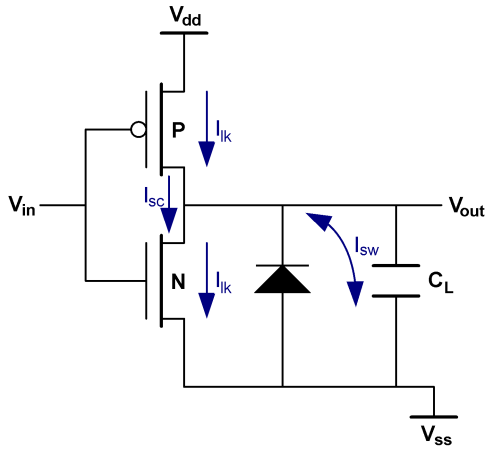


Figure 2.10: CMOS power dissipation circuit diagram

2.4.2 Dynamic Power

The power dissipated by an active gate is called *dynamic power*, and can be split into the following two categories.

Internal Power

Whenever the voltage on a net changes, a circuit dissipates power due to charging or discharging of internal capacitances. This is one of the components of *internal power*, together with the more dominating component called *short-circuit power*. The latter is caused by the short-circuit current going from V_{dd} to ground the short period of time the PMOS and NMOS transistors are simultaneously open during switching [12]. In Figure 2.10, I_{sc} is the short-circuit current rising when both transistor types are partially open.

Switching Power

The *switching power* is the power dissipated by charging and discharging the load capacitance C_L, which is given by the sum of net and gate capacitances at the output. In Figure 2.10, I_{sw} denotes the switching current charging and discharging the output capacitance. The switching power component is part of the following equation, which shows the total average power in a digital CMOS circuit:

$$\begin{aligned} P_{avg} &= P_{switching} + P_{short-circuit} + P_{leakage} \\ &= \alpha C_L V_{dd}^2 f_{clk} + I_{sc} V_{dd} + I_{lk} V_{dd} \end{aligned} \tag{2.12}$$

The switching component is calculated from the supply voltage V_{dd}, the load capacitance C_L, the clock frequency f_{clk} and the switching activity factor α. The factor α is defined as the average number of times in each clock cycle a specific node makes a power consuming transition, going from 0 to 1 [3].

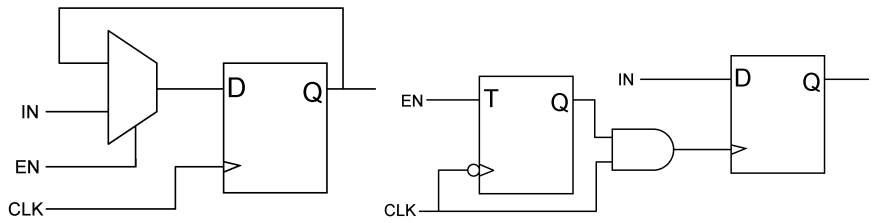
2.5 Low power techniques

This section presents some of the most common low-power techniques for reducing dynamic and static power dissipation, which will be part of the dynamic implementations in Chapter 7.

2.5.1 Clock gating

One of the most common low-power techniques is *clock gating*, which means to prevent the clock signal from propagating to parts of the circuit whenever it is not required [1]. This reduces dynamic power in the clock distribution tree, and in the sequential elements connected to that clock.

The most simple clock gate is realized with the clock signal and the enable signal connected to the inputs of an AND-gate. However, this structure is sensitive to glitches in the enable signal, that would create glitches in the gated clock signal as well. A better solution is obtained by using a latch in addition to the AND-gate, which filters the glitches in the enable signal [1]. This structure can be seen in Figure 2.11b, where it appears in front of a register, while Figure 2.11a shows the enabled register without clock gating.



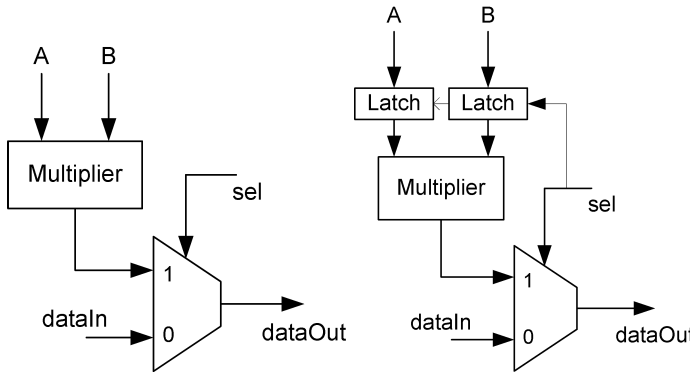
(a) Not implemented (b) Implemented

Figure 2.11: Enabled register with and without clock gating

2.5.2 Datapath gating

Significant amounts of dynamic power may be wasted in combinational datapaths, due to switching activity that has no contribution to the functionality of the circuit [1].

Guarded evaluation is a datapath gating technique which can be applied when the outputs of embedded combinational blocks are not used. Transparent latches are inserted at the inputs of the combinational block, and control logic is added to determine when the output of the block is unused. The control signal is then used to latch the inputs, preventing the combinational block from toggling [1]. An example of guarded evaluation is illustrated in Figure 2.12, where a multiplier represents the combinational block.

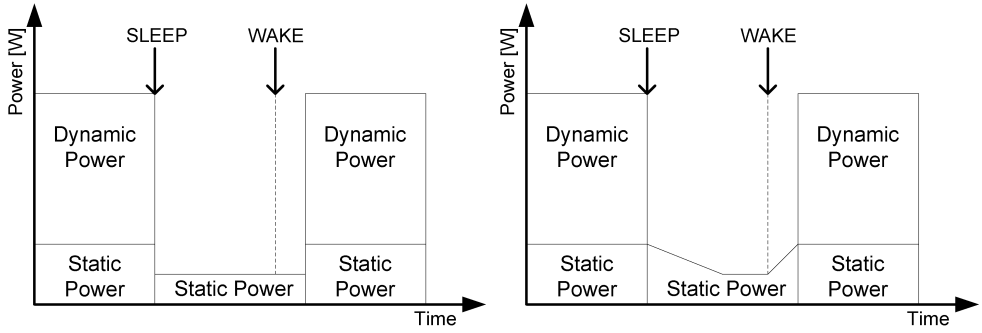


(a) Not implemented (b) Implemented

Figure 2.12: Datapath gating using guarded evaluation

2.5.3 Power gating

While clock- and datapath gating can be used to reduce the dynamic power dissipation of a circuit, *power gating* may be used to reduce static power. The basic strategy of power gating is to provide a low power mode and an active mode, and switch between the modes in order to maximize the power reduction while minimizing the impact on performance. A disadvantage of power gating is that it adds significant time delays to safely power up and down the circuits. Also, controlling a power domain requires one or more power switches, which will increase the area and also contribute with additional leakage power. Figure 2.13 illustrates the ideal versus the realistic effect of power gating [8].



(a) Ideal (b) Realistic

Figure 2.13: Ideal and realistic effect of power gating

Chapter 3

Automated area and power estimating tool-flow

This chapter describes the tool-flow that is created in order to obtain area and power numbers for each RTL implementation, addressing Objective 1 in Section 1.4. The tool-flow is based on the previous work of the author [14], where it was found that fast and accurate power estimations could be obtained by applying switching activity information from RTL simulation on post-layout netlists. The tool-flow is created using a Makefile of the GNU Make Utility[1] to structure a list of commands with certain dependencies. The Makefile can be read in Appendix A.1, and its function is illustrated in Figure 3.1. The tool-flow estimates the area and power dissipation of an RTL design automatically by simply providing it with the RTL filelist. The figure shows the dependencies between the steps in the tool-flow, and a simplified overview of how design files are passed on to succeeding tools. The tool-flow is

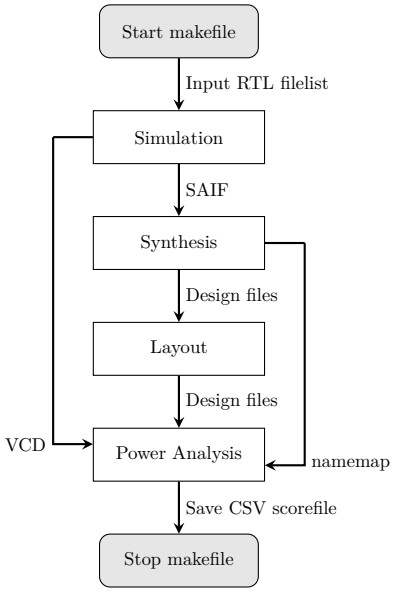


Figure 3.1: Automated area and power estimating tool-flow

[1] https://www.gnu.org/software/make/manual/make.pdf

run on a computer of the current industry standard, holding two 2.8 GHz six-core Intel Xeon E5660 processors and 32 GB RAM, running a 64 bit CentOS 5.10 operating system.

3.1 The steps of a complete design cycle

Figure 3.2 shows a complete design cycle in this work, from RTL design to illustrated results of the designs area and power dissipation, including the automated tool-flow. The steps in this design cycle will now be explained in detail.

3.1.1 RTL Design

A module is designed in SystemVerilog[2], and a filelist is written which lists all RTL files that makes the complete design. The filelist must have a unique name, representing that particular design, as it will be used throughout the whole tool-flow.

3.1.2 Commands in terminal

Two simple commands needs to be executed in terminal in order to start the automated tool-flow. The first command sets the enviormental variable `FILE_LIST` to the unique name of the current RTL design. The second command, `make runpow`, tells the Makefile to run the power analysis. Due to the dependencies, the Makefile will first complete the simulation, synthesis and layout, before starting on the power analysis.

3.1.3 Simulation

This is the first step of the automated tool-flow. The current RTL design is simulated in Mentor Graphics' Questa[3], running a testbench which applies several test frequencies to the data input of the module. Since all RTL designs are planned to have the same interface, the testbench will be reused for all cases. The test scenarios are listed in Table 3.1. The testbench SystemVerilog code can be found in Appendix A.2. The testbench also logs the data output, such that the frequency response may be calculated from the impulse response in simulation. Furthermore, the simulation tool generates both Value Change Dump (VCD) and Switching Activity Interchange Format (SAIF) activity files. VCD is a cycle accurate activity file format which will be used to annotate switching activity in power analysis. SAIF is an average activity file format which, in this case, only is used to generate a name-mapping file in synthesis. The name-mapping will ensure name consistency between the activity file and the netlist.

[2] http://www.eda.org/sv/SystemVerilog_3.1a.pdf
[3] http://www.mentor.com/products/fv/questa/

Test case	Power Scenario	Description
Impulse response	None	A single pulse is input to the system. This can be used to calculate the frequency response of the filter.
Inactive	Inactive	The clock is the only signal active. This can be used to measure power dissipation in idle state.
1 MHz input	Active	The data input is a 1 MHz sinus. Attenuation is measured.
2 MHz input	Active	The data input is a 2 MHz sinus. Attenuation is measured.
3 MHz input	Active	The data input is a 3 MHz sinus. Attenuation is measured.

Table 3.1: Overview of test cases and power scenarios

3.1.4 Synthesis

This is the second step of the automated tool-flow. When the Makefile encounters this step, it enters a new Makefile dedicated to synthesis, which can be found in Appendix A.3. The synthesis is performed in Design Compiler[4] using Synopsys' reference scripts for recommended methodology [13].

The RTL design, given by the filelist variable, is synthesized to a netlist of logic gates from a given cell library in Synopsys Design Compiler. The cell library used is of 180 nm technology. The synthesis is performed under certain design constraints, including input transition times, output capacitance load and clock signal details, which can be found in Appendix A.4. A name-mapping file is generated from the SAIF activity file created in simulation. This will be used in power analysis to map the signals from RTL simulation to the corresponding pins in the post-layout netlist. The synthesis creates a Verilog netlist, a Synopsys design file and an expanded constraint file, which all will be used in layout.

Slight changes are made to the recommended methodology scripts in order to read in the correct RTL design and perform the name-mapping. Important commands for this are listed below.

[4] http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/

```
set FILE_LIST [getenv "FILE_LIST"]
set DESIGN_NAME "IIRFilt"
set RTL_SOURCE_FILES "[N_ReadFileList␣[subst␣"../rtl/${FILE_LIST}.fl"]]
    "
analyze -format sverilog -define HINST_TSMC_180_ARM_1V2 ${
    RTL_SOURCE_FILES}
saif_map -start
read_saif -auto_map_names -input /pri/jota/workspace/master/ip/IIRFilt/
    sim/run/rtl/IIRFilt.saif -instance test_IIRFilt/u_IIRFilt -verbose
saif_map -type ptpx -write_map ${RESULTS_DIR}/${DESIGN_NAME}
    .mapped.SAIF.namemap
```

3.1.5 Layout

As the Makefile encounters the layout step, it enters the layout dedicated Makefile, which can be read in Appendix A.5. As for the synthesis step, the layout is also performed using Synopsys' reference scripts for recommended methodology [13].

The synthesized design files are passed on to Synopsys IC Compiler [5] which performs floorplanning, place and route, clock tree synthesis and optimization algorithms. The script is set up to create a quadratic core layout with a core utilization of 40%, with a vertical power supply through the center of the core. Key commands for this customization are listed below.

```
set DESIGN_NAME "IIRFilt"
set ICC_IN_VERILOG_NETLIST_FILE "/pri/jota/workspace/master/ip/IIRFilt/
    syn/results/$DESIGN_NAME.mapped.v" ;
set ICC_IN_SDC_FILE             "/pri/jota/workspace/master/ip/IIRFilt/
    syn/results/$DESIGN_NAME.mapped.sdc"
set ICC_IN_DDC_FILE             "/pri/jota/workspace/master/ip/IIRFilt/
    syn/results/$DESIGN_NAME.mapped.ddc"
create_floorplan \
        -control_type aspect_ratio \
        -core_aspect_ratio 1 \
        -core_utilization 0.4 \
        -left_io2core 5 \
        -bottom_io2core 5 \
        -right_io2core 5 \
        -top_io2core 5 \
        -start_first_row
create_power_straps  -direction vertical  -start_at 150.0 -nets  {
    DVDD_1V2 AVSS}  -layer METAL2 -width 1.0
```

[5] http://www.synopsys.com/Tools/Implementation/PhysicalImplementation/Pages/ICCompiler.aspx

The output of the layout is a post-layout netlist, a design constraint file, and a parasitic exchange file. It also generates several layout reports, from which details regarding cell count and area can be extracted. These will be included in the Comma-Separated Values (CSV) scorefile generated in the end of power analysis. The details are:

- Number of combinational cells
- Number of sequential cells
- Number of clock tree buffers/inverters
- Number of adders
- Combinational area
- Noncombinational area
- Total area

3.1.6 Power Analysis

When the Makefile reaches the final step, it enters the power analysis dedicated Makefile, given in Appendix A.6. The power analysis script is also based on Synopsys' reference scripts. However, the script can be read in its entirety in Appendix A.7 due to several customizations.

The design files from layout are passed on to Synopsys' PrimeTime-PX[6], which performs the power analysis. It analyzes the activity in different scenarios, defined by certain time windows of the VCD file. The power scenarios are given in Table 3.1, where *Active* is an average power analysis of three different time windows. PrimeTime uses the VCD activity file generated in RTL simulation to initiate toggling in as many nets as possible. The name-mapping file ensures that the toggling information reaches the correct pin. The nets that are not directly annotated, will get their toggling information from activity propagating throughout the circuit. The power analysis generates a power report for each scenario. The information extracted from these reports into the scorefile is:

- Total internal power
- Total switching power
- Total leakage power
- Total power

[6] http://www.synopsys.com/Tools/Implementation/SignOff/Pages/PrimeTime.aspx

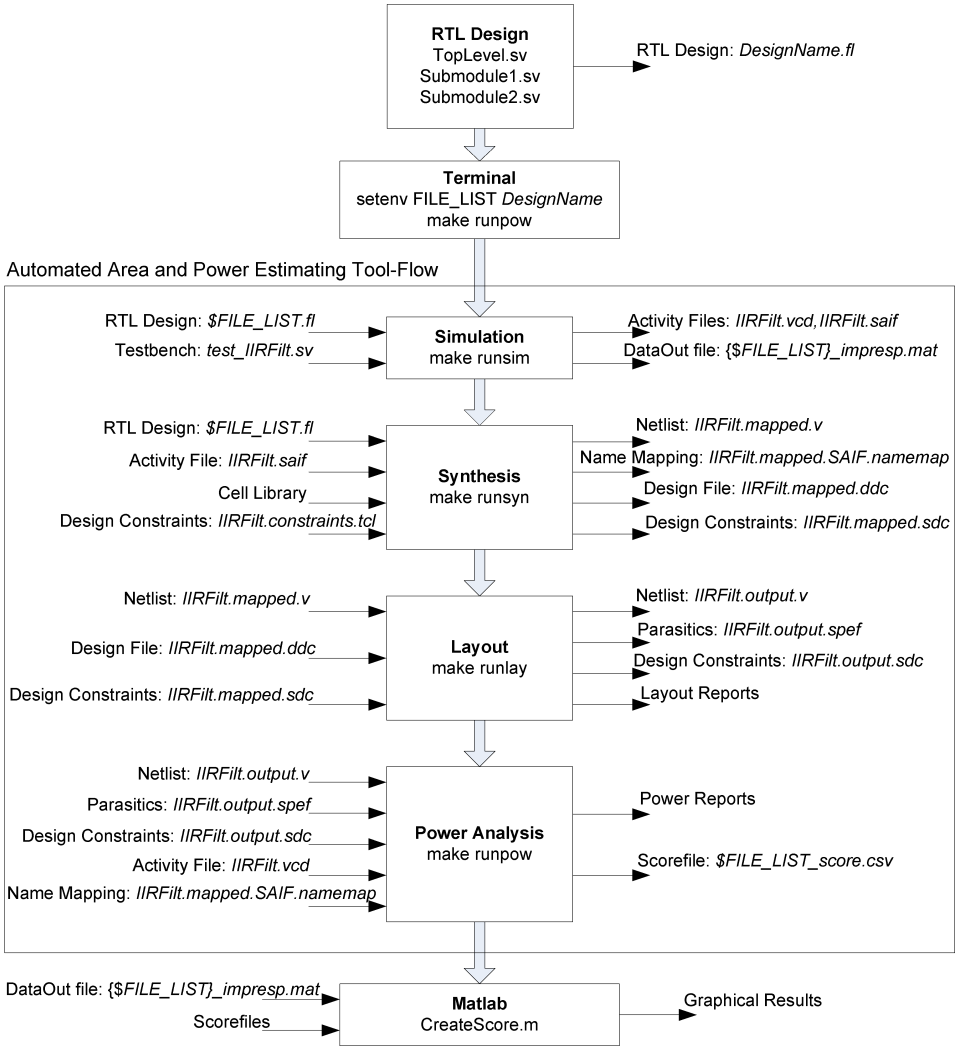


Figure 3.2: Detailed overview of the automated area and power estimating tool-flow

3.1.7 Visualization in Matlab

Finally, one or more CSV scorefiles may be input to the Matlab[7] script which is created to visualize the scores and easily compare the different RTL implementations. The script also reads the impulse response dumped from RTL simulation in order to plot the frequency responses of the implementations. The Matlab script can be read in Appendix A.8.

[7] http://se.mathworks.com/products/matlab/

Chapter 4

Automated filter generation and eligibility calculation

This chapter describes the Automated Filter Generation and Eligibility Calculation (AFGEC) algorithm, which is a Matlab script that generates a set of filters and sorts them based on how eligible they are for hardware implementation. The flow of the AFGEC algorithm is illustrated in Figure 4.1. The script can be read in its entirety in Appendix B.1. Section 4.1 and 4.2 describes the automated filter generation, Section 4.3 describes the calculation of the eligibility factor, while Section 4.4 carry out the results of the AFGEC algorithm.

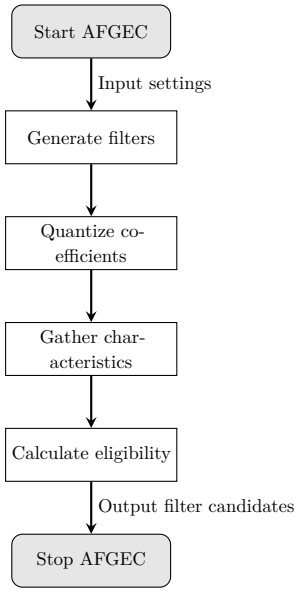


Figure 4.1: Automated filter generation and eligibility calculation flow diagram

4.1 Automated filter generation

This section explains the filter generation algorithm, states the filter requirements and elaborates on the filter types and structures that are chosen.

4.1.1 Filter requirements

The filter to be generated should be a low-pass filter with cut-off frequency close to 1 MHz and sampling frequency of 16 MHz. The gain/attenuation at 1 MHz should be as close to 0 dB as possible, within 3 dB deviation. The attenuation at 2 MHz and 3 MHz should be more than 6 dB and 12 dB, respectively. There are no strict requirements with regards to phase linearity of the filter, other than the fact that it should be close to linear in passband. The requirements are summarized in Table 4.1.

Factor	Requirement
Characteristic	Low pass
Sampling frequency	16 MHz
Cut-off frequency	1 MHz
Max att./gain at 1 MHz	3 dB
Min attenuation at 2 MHz	6 dB
Min attenuation at 3 MHz	12 dB

Table 4.1: Filter requirements

4.1.2 Filter type

An advantage of FIR filters is the possibility of having linear phase response. However, there are no strict requirements regarding phase linearity in this work, which makes the IIR filters just as applicable. As described in Section 2.2.1, a disadvantage of FIR filters is that they require a relatively high order in order to obtain steep curves in the frequency response. A high filter order implies a large amount of registers and multipliers, which again will have a significant impact on the area and power dissipation. Therefore, since energy is of the essence in this work, it is decided to focus on IIR filters. All of the commonly used IIR filters presented in Section 2.2.2 will be generated in the AFGEC algorithm. That is the Butterworth, Elliptic and Chebyshev Type I filters. It is interesting to see if either of the IIR filters are more eligible for hardware implementation than the others.

4.1.3 Filter structure

Obviously, implementing the IIR filters in a Direct Form I structure is a suboptimal solution. As described in Section 2.2.2, it requires the double amount of registers as for instance the Direct Form II structure, which clearly would effect the power dissipation. Moreover, we read that both Direct Form structures are extremely sensitive to quantization and was not recommended for practical applications. Hence, all the generated filters will be structured as first- and second-order sections in the Cascade-Form, which exploits the robustness of low order filters. This structure may

also be well suited for a dynamic approach, where subsections can be enabled or disabled based on the current filter requirement. This topic will be addressed in Chapter 7.

4.1.4 Filter generating algorithm

The filter generating algorithm is the first part of the AFGEC script, which can be read in its entirety in Appendix B.1. The script is a Matlab program which takes use of the functions `butter`, `cheby1` and `ellip` in order to design digital IIR filters.

Initial values

The purpose of the filter generating algorithm is to generate a large number of filters, which eventually will be compared against each other. However, the filters are not intended to range from very poor performance to excellent performance. It is rather important that the filters are just slightly different, with almost identical performance. Hence, when generating the filters, it is decided to keep the filter order n_{ord} fixed at 3 for all solutions, and rather focus on the more fine-grained variations that rise when sweeping the cut-off frequency of the filter over a certain interval. The initial frequency f_c is set according to the specifications, to 1 MHz, and the frequency step interval is set to 8 kHz. This is the interval which the cut-off frequency increases from one filter solution to the next. Hence, as 100 filters of each type are generated, their cut-off frequencies range from 1 MHz to 1.8 MHz. All filters are converted from transfer function representation to second-order section representation in Matlab, using the `tf2sos` function, before quantizing the coefficients. The initial values are summarized in Table 4.2.

Parameter	Value	Description
T_b	100	Number of Butterworth filters to be generated
T_c	100	Number of Chebyshev Type I filters to be generated
T_e	100	Number of Elliptic filters to be generated
n_{ord}	3	Filter order
f_c	1 MHz	The initial filter cut-off frequency
f_{step}	8 kHz	Frequency step interval

Table 4.2: Parameters in the filter generating algorithm

Definitions

A total of T filters are generated, grouped into a vector Y, defined as

$$Y = \{y^1, y^2, \cdots, y^{T-1}\}$$

where y^i is the ith filter solution, and T is derived from

$$T = T_b + T_c + T_e$$

The cut-off frequencies of the implementations are given by the initial value f_c, the step value f_{step} and the filter implementation number i, as

$$f_c^i = \begin{cases} f_c + f_{step}(i-1) & \text{if} \quad 0 < i \leq T_b \\ f_c + f_{step}(i - T_b - 1) & \text{if} \quad T_b < i \leq T_b + T_c \\ f_c + f_{step}(i - T_b - T_c - 1) & \text{if} \quad T_b + T_c < i \leq T \end{cases}$$

Each filter solution can be described by its N coefficients, defined in vector A^i, as

$$A^i = \{a_0^i, a_1^i, \cdots, a_{N^i-1}^i\}$$

The number of coefficients N in a cascaded biquadratic structured filter depends on the filter order n_{ord}, as

$$N = \lceil \frac{n_{ord}}{2} \rceil * 6$$

4.2 Coefficient quantization algorithm

The coefficient quantization algorithm is the second step of the AFGEC algorithm. Its purpose is to represent the filter coefficients from the previous section with as few bits as possible, without changing the original values more than a predefined tolerance value. The coefficient quantization algorithm is visualized in Figure 4.2.

The quantization algorithm starts with zero fraction bits, and increases the number of bits after the decimal point as long as the quantization error ϵ_j^i is greater than the tolerance τ. A smaller tolerance value implies more accurate coefficients and hence a higher number of bits per coefficient. In this work, a tolerance value of 2^{-3} is used. The number of fraction bits will also be increased if the filter after quantization has infinite gain for some frequency. This will occur if a pole is moved to the unit circle, i.e. if one of the feedback coefficients a_k are -1. Infinite gain can also occur at DC if the sum of a_k-coefficients are -1, since this implies that the denominator in Equation 2.2 is zero. If this occurs, the property will be tested again, with an increased number of fraction bits for all coefficients.

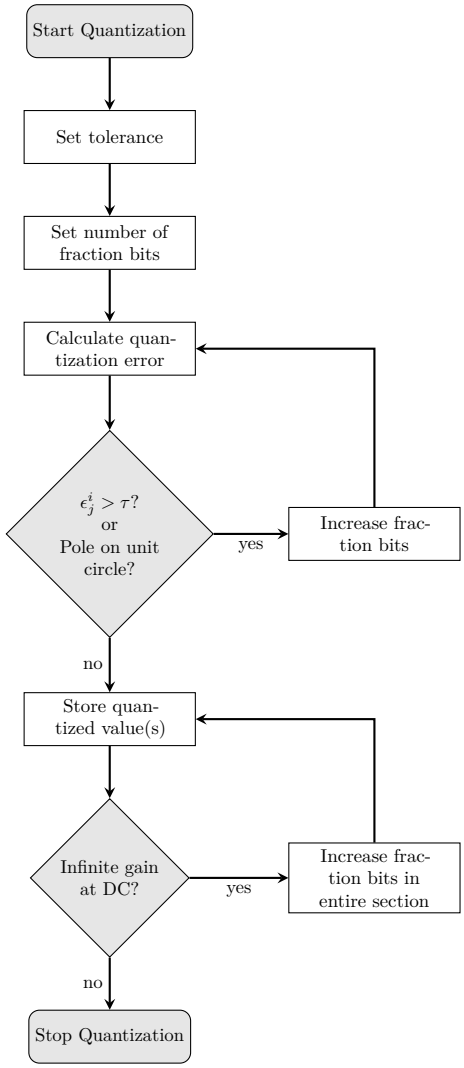


Figure 4.2: Flow chart of the coefficient quantization algorithm

Definitions

$\bar{Y}$ is the set of quantized filters, defined as

$$\bar{Y} = \{\bar{y}^1, \bar{y}^2, \cdots, \bar{y}^{T-1}\}$$

And $\bar{A}^i$ is the vector of quantized coefficients in the ith filter solution, defined as

$$\bar{A}^i = \{\bar{a}_0^i, \bar{a}_1^i, \cdots, \bar{a}_{N^i-1}^i\}$$

The quantization error of the jth coefficient in the ith filter implementation, ϵ_j^i, is defined as

$$\epsilon_j^i = a_j^i - \bar{a}_j^i$$

When each quantization is finished, the number of bits used to represent the coefficients in the ith filter is stored in the vector $\bar{B}^i$, which is defined as

$$\bar{B}^i = \{\bar{b}_0^i, \bar{b}_1^i, \cdots, \bar{b}_{N^i-1}^i\}$$

4.3 Eligibility calculation

As the set of filters is generated and quantized, it is of interest to sort them based on how eligible they are for hardware implementation. For this, it is first necessary to extract some characteristics that can indicate how expensive it will be to implement each filter solution with regards to area and power dissipation.

4.3.1 Characteristics

As the structure and order of the filters are selected, the coefficients are the only variables left which separate the solutions. The filter coefficients will define the number of multipliers, adders and shift operators used in the filter, and their complexity, which will have a significant impact on the filter's power dissipation. The following explains how these metrics can be extracted from the coefficient numbers.

Zero- and one-coefficients

The number of coefficients that are zero or one, are important parameters, as either of them implies that the circuit will require a multiplier less. For instance, if the coefficient b_{k2} in Figure 2.6 is 1, that multiplier can been removed, and the output from the second delay element can be wired directly to the following adder. And if b_{k2} is 0, not only the multiplier can be removed, but also the adder in front of b_{k1}. Hence, a zero-coefficient will also imply one less adder. The characteristics c_0^i and c_1^i are the rates of zero-coefficients and one-coefficients, respectively. These are defined as

$$c_0^i = \frac{1}{N^i} \sum_{j=0}^{N^i-1} \alpha(a_j^i)$$

and

$$c_1^i = \frac{1}{N^i} \sum_{j=0}^{N^i-1} \beta(a_j^i)$$

with the functions $\alpha(x)$ and $\beta(x)$ defined as

$$\alpha(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x \neq 0 \end{cases}$$

and

$$\beta(x) = \begin{cases} 1 & \text{if } x = 1 \\ 0 & \text{if } x \neq 1 \end{cases}$$

As the number of zero- and one-coefficients are found, the remaining coefficients should be investigated, as these are the ones that will be implemented as multipliers and shift operators. When excluding the zero- and one-coefficients in A^i, we get the set of remaining coefficients $\hat{A}^i$, defined as

$$\hat{A}^i = \{\hat{a}_0^i, \hat{a}_1^i, \cdots, \hat{a}_{M^i-1}^i\}$$

and the corresponding vector with number of bits after quantization

$$\hat{B}^i = \{\hat{b}_0^i, \hat{b}_1^i, \cdots, \hat{b}_{M^i-1}^i\}$$

where $M^i = N^i(1 - c_0^i - c_1^i)$.

Single-one coefficients

If a coefficient have a binary representation that is all zeros except for a single-one bit, it will be implemented as a shift operation. For instance, the binary representation

of the number 2 is 010.00, and the binary representation of the number 0.5 is 000.10. Coefficients that hold this property will have a smaller contribution to the power dissipation compared to a coefficient that is implemented as a multiplier, as a shift operator is less complex. The number of single-one-coefficients out of remaining coefficients is the characteristic c_2^i, defined as

$$c_2^i = \frac{1}{M^i} \sum_{j=0}^{M^i-1} \gamma(\hat{a}_j^i)$$

with the function $\gamma(x)$ defined as

$$\gamma(x) = \begin{cases} 1 & \text{if } x \in \{2^i\}; \text{ where } i \in \mathbb{Z} \\ 0 & \text{otherwise} \end{cases}$$

Bits per coefficient

The parameter affecting the complexity of the multipliers and the shift operators, is the number of bits per coefficient, as this determines the amount of combinational logic. This describes the last characteristic c_3^i which is the average number of bits in the remaining coefficients, defined as

$$c_3^i = \frac{1}{M^i} \sum_{j=0}^{M^i-1} \hat{b}_j^i$$

4.3.2 Eligibility function

The eligibility function E^i combines all the characteristics described above, in order to find the filter solution most eligible for hardware implementation. The solution yielding the highest number is considered most eligible. Parameters related to filter performance, like phase linearity, ripple and attenuation, are inspected visually from the results generated in Matlab. Note that c_3^i is inverted since fewer bits shall have a positive impact on the eligibility.

$$E^i = c_0^i + c_1^i + c_2^i + \frac{1}{c_3^i}$$

4.4 Results of the AFGEC

Figure 4.3 shows the calculated eligibility when generating 300 filters. Filter implementations 1 to 100 are Butterworth filters, 101 to 200 are Chebychev Type 1 filters, and 201 to 300 are Elliptic filters. The first implementation of each filter type, i.e. implementation number 1, 101 and 201, all have the same cut-off frequency of 1 MHz.

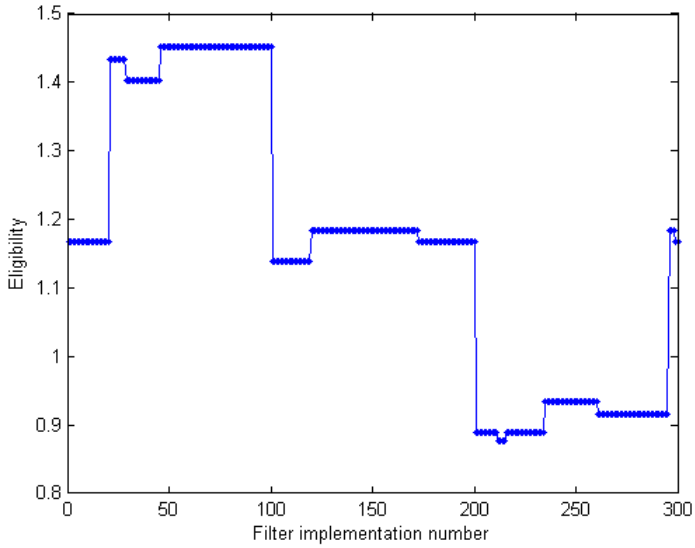


Figure 4.3: Calculated eligibility for each of the 300 filter implementations

It is observed that the Butterworth filters yield the overall best results, while the Elliptic filters have the worst results. The reason for this can be found when studying Figure 4.4, which illustrates each parameter affecting the eligibility function. First of all, it is observed that the number of zero-coefficients and one-coefficients are constant at 2 and 6, respectively. An odd order number implies that one first-order section will be present, which is represented as a second-order section with two zero-coefficients in Matlab. And since all filter solutions have their zeroes at 8 MHz, they all get the same number of one-coefficients.

Moreover, the number of single-one-coefficients range from 2 for some Butterworth filters, to 0 for some Elliptic filters. When looking at the corresponding characteristic c_2^i in Figure 4.5, it can be seen that this number makes half of the remaining coefficients in most of the Butterworth filters. These coefficients may be implemented as shift operators, which might have a significant impact on the area and the power dissipation. In the eligibility function in Figure 4.3, the contribution of c_2^i is easily recognized in the plot.

From Figure 4.4 it can be seen that the total number of bits spans from 14 for some of the Butterworth filters, to 19 for some of the Elliptic filters. The inverted characteristic c_3^i have a small but significant impact on the eligibility function, making it possible to trade between fairly similar solutions.

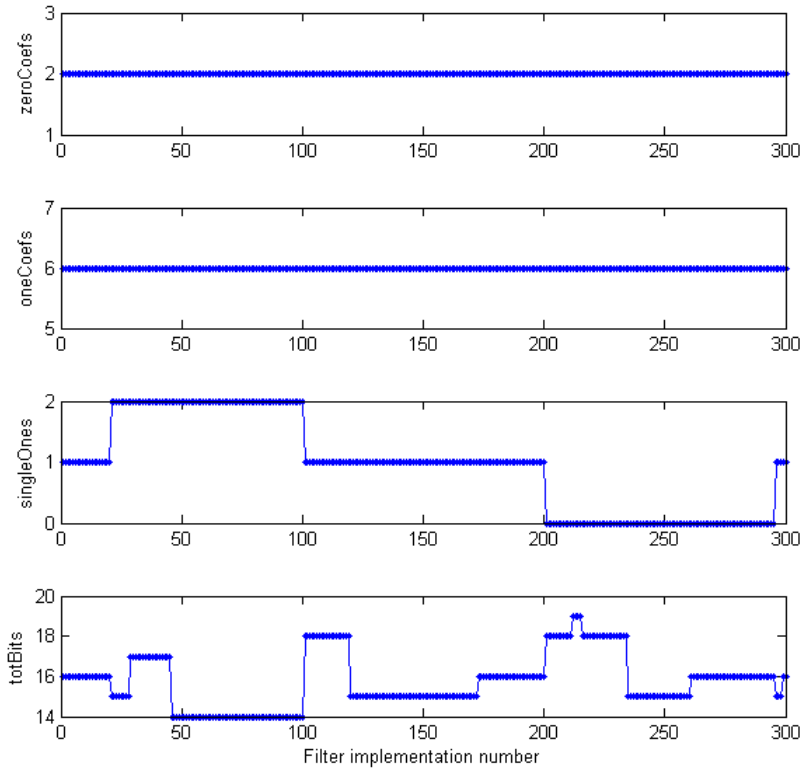


Figure 4.4: The parameters which influence the four characteristics, of each filter implementation

4.4.1 Winner candidates from the AFGEC

The six unique highest scoring filter solutions have the following filter implementation numbers: 46, 21, 29, 30, 120 and 296, where 46 is the best and 296 is the worst. These filters will be referred to as *IIRFilt46* to *IIRFilt296*, and their frequency response is depicted in Figures 4.6 to 4.11. Since all filters are of third order, they consist of one First Order Section (FOS) and one Second Order Section (SOS). The plots show the frequency response of each subsection separately and combined in order

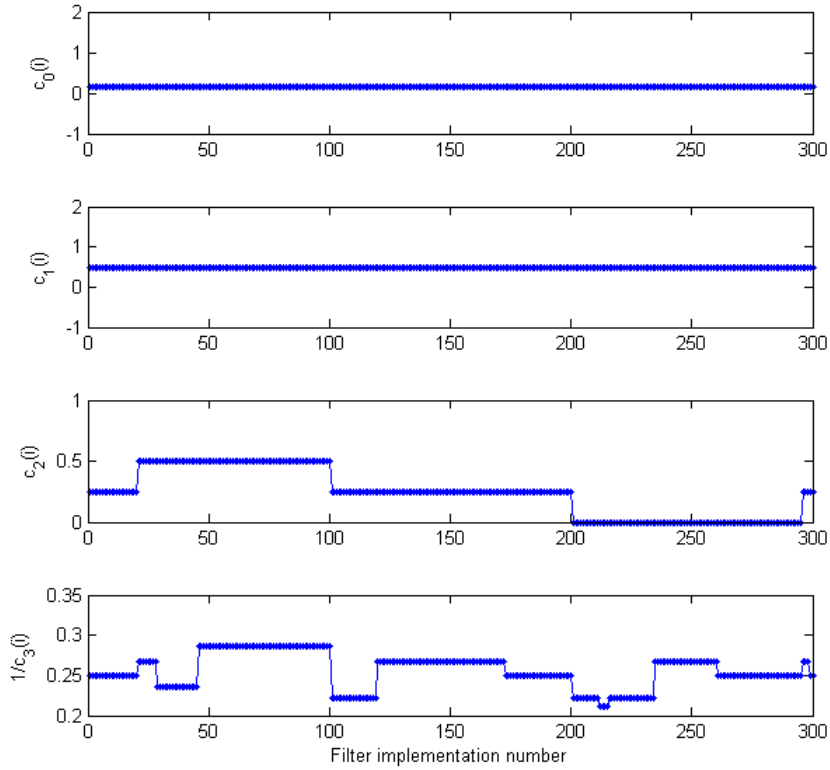


Figure 4.5: The four characteristics yielding the Eligibility function, of each filter implementation

to decide whether the filter has potential for dynamic implementation or not. The four highest scoring candidates are all Butterworth filters, while the fifth and sixth candidate are Chebyshev and Elliptic filters, respectively. The passband gain is not of relevance when comparing the filters, but rather the shape of the frequency response. All filters will be scaled such that they have unity gain when proceeding to RTL implementation.

Figure 4.6 shows the frequency response of *IIRFilt46*, a third order Butterworth filter with cut-off frequency at 1.36 MHz. This solution came out with the highest eligibility score, having two single-one-coefficients and only 14 bits in total. The frequency response is rather linear in passband, from 0 to 1 MHz, for both subsections and the resulting third order filter. The subsections have slightly different cut-off

frequencies, and less attenuation at 2 MHz, but may be used separately in a dynamic approach. Figure 4.12 shows the phase response in passband for all six candidates, and it is observed that *IIRFilt46* seems most linear in this interval. The attenuation from 1 MHz to 2 MHz is 8.4 dB.

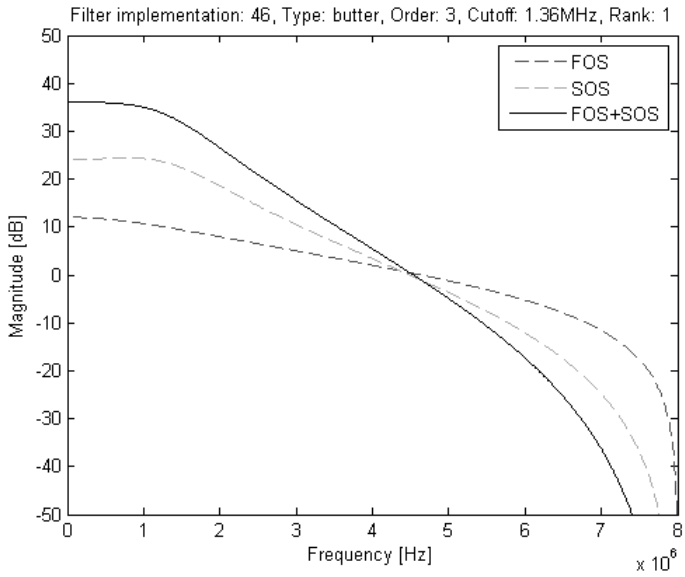


Figure 4.6: Frequency response of IIRFilt46

The second highest scoring candidate is *IIRFilt21*, whose frequency response is depicted in Figure 4.7. From the figure, a large bump at 1.3 MHz is observed, caused by the frequency response of the SOS. In Figure 4.12, its phase response is overlapped by the yellow line of *IIRFilt296*. The linearity is affected, but mostly after 1 MHz. The attenuation from 1 MHz to 2 MHz is 10 dB, slightly better than the *IIRFilt46*.

IIRFilt29 is the third most eligible Butterworth filter, with its frequency response showed in Figure 4.8. This filter has a slightly lower cut-off frequency than the *IIRFilt46*, which implies more attenuation in passband, but also more attenuation at 2 MHz. The attenuation difference from 1 to 2 MHz is therefore slightly better, of 13.5 dB. The subsections have the same qualities as for the *IIRFilt46*, which makes them suitable for dynamic implementation. *IIRFilt29* has however the least linear phase response of all candidates.

IIRFilt30 is the fourth highest scoring filter. Its quantization seems to have shifted the cut-off frequency of 1.23 MHz to a slightly higher frequency, which results in less attenuation at both 1 MHz and 2 MHz. This affects the relative attenuation at 2 MHz, of 8.9 dB. The filter's SOS is also slightly less linear in passband, which

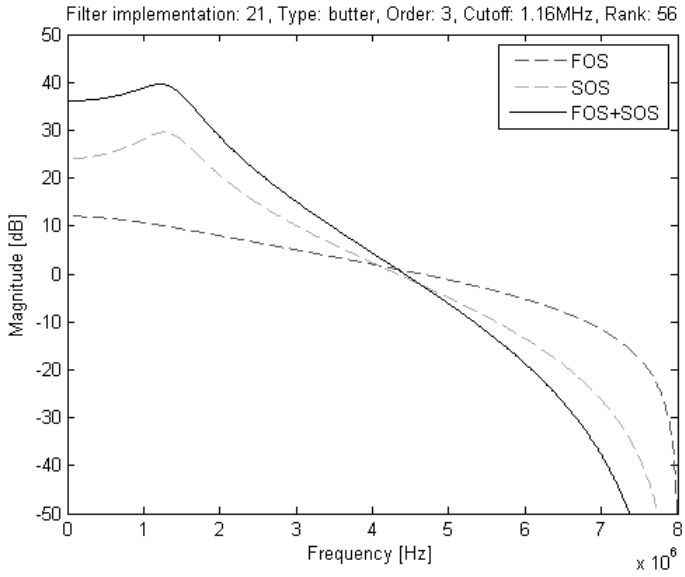


Figure 4.7: Frequency response of IIRFilt21

makes it less eligible for dynamic implementation. The phase response is rather linear.

IIRFilt120 is the highest scoring Chebyshev filter. It has a fairly linear passband, and a very steep curve in stopband. The attenuation from 1 MHz to 2 MHz is 12.4 dB, and the phase response is also linear in passband. The only drawback is that the frequency responses of the subsections are non-linear in passband, which makes the filter less suited for dynamic implementation.

IIRFilt296 is the highest scoring Elliptic filter. It has the largest attenuation from 1 MHz to 2 MHz, of 20 dB, but not as much attenuation at for instance 3 MHz. Moreover, its frequency and phase responses are quite linear in passband. However, its drawback is that the subsections are less eligible separately, similarly to the *IIRFilt120* and the *IIRFilt21*.

4.4.2 Conclusion of the AFGEC

Based on the frequency and phase responses of the six highest scoring filter solutions, two candidates remark themselves as particularly promising. These are *IIRFilt46* and *IIRFilt29*, which yield linear characteristics in passband for both their first- and second-order sections and their resulting third-order filter.

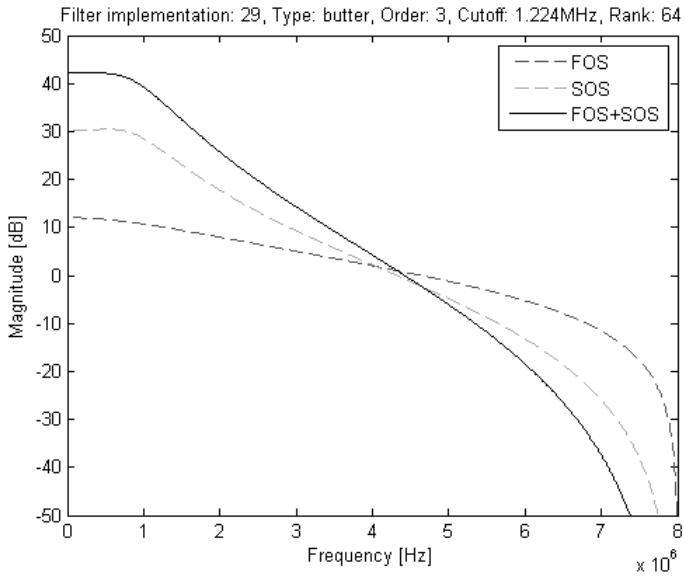


Figure 4.8: Frequency response of IIRFilt29

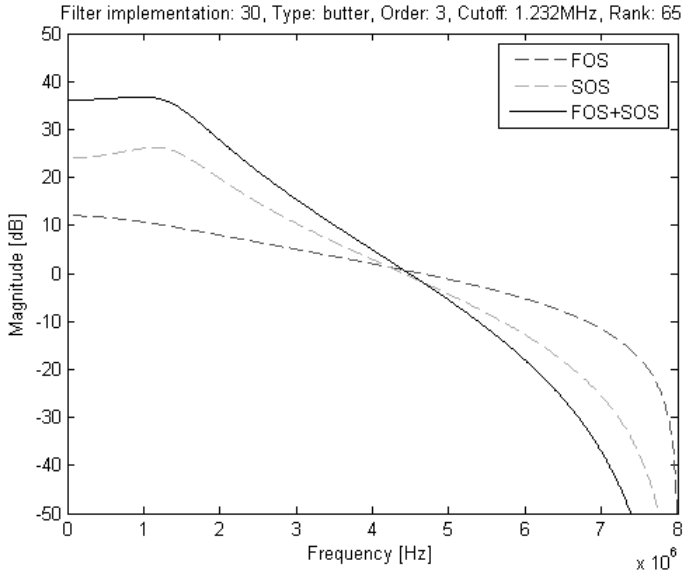


Figure 4.9: Frequency response of IIRFilt30

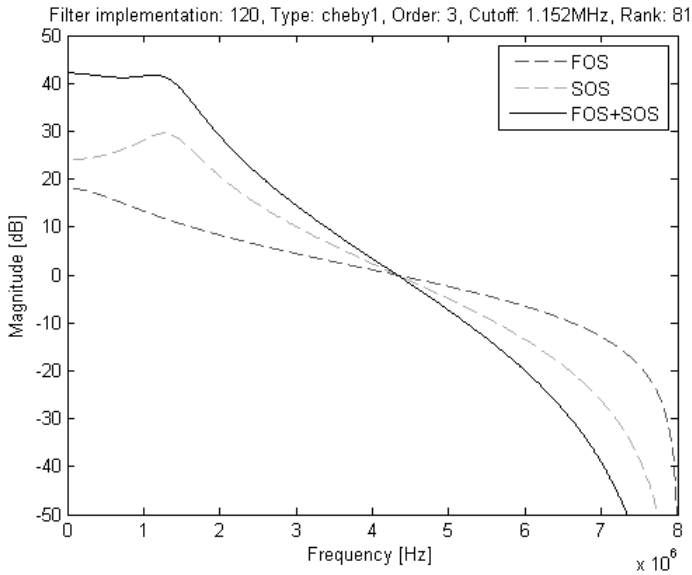


Figure 4.10: Frequency response of IIRFilt120

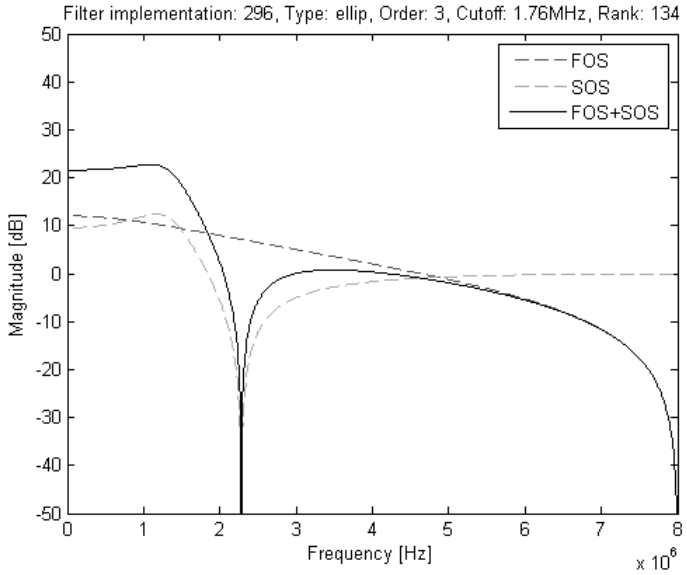


Figure 4.11: Frequency response of IIRFilt296

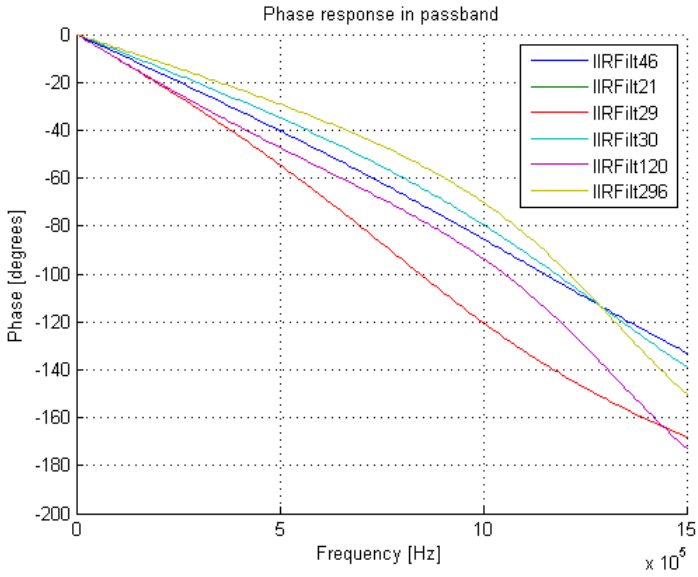


Figure 4.12: Phase response in passband of winner candidates

Chapter 5

RTL implementation of winner candidates

In order to ensure that the AFGEC algorithm in Chapter 4 makes precise predictions, each of the six highest scoring filter solutions are implemented in RTL. Then, the RTL implementations are passed through the tool-flow in Chapter 3, and compared against each other in terms of area and power dissipation.

5.1 Framework

Since all the candidates are third order IIR filters, they have the exact same framework consisting of one FOS and one SOS. The top level module is called `u_IIRFilt`, and is depicted in Figure 5.1. The main inputs and outputs of the top level module are the `dataIn` and `dataOut` buses, which hold 12 bits each. The inter-module bitwidth B is a local parameter which may be changed in order to affect the quantization noise in the filter. The subsections in the filter are designed to use the same input and output word lengths, and to perform their own separate scaling to unity gain of their data output. The reason for this is twofold. First of all, as every subsection has the same interface, it is simple to combine different biquadratic sections at a later stage. With unity gain, it is also possible to cascade as many biquadratic sections as desired without having to concern about the signal to over- or underflow. Secondly, the common word length and separate scaling allow us to more easily bypass parts of the filter in a dynamic approach. This will be adressed in Chapter 7.

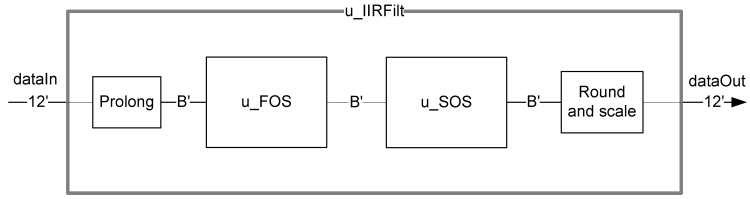


Figure 5.1: Top level block diagram of the *IIRFilt* implementations

The top level module has a `Prolong`-section, which prolongs the input signal to match the bitwidth of the `u_FOS` interface. It also has a `Round and scale`-section at the end, which downscales the output of the last biquadratic section to match the 12 bit output signal.

Figure 5.2 and Figure 5.3 show the framework of the FOS and the SOS, respectively. As mentioned, they have the same bitwidth externally, but internally they have some differences. The internal bitwidth is determined by the inter-module bitwidth, B, and the bitwidth of the coefficients, C. These variables decide the bitwidth of the multiplier outputs. Hence, the input to `u_FOS` is prolonged to $B + C$ bits, which is the internal word length of that section. In order to obtain full precision at the adder outputs, the bitwidth should be increased by one bit when adding two terms, and two bits when adding three terms. This is impossible to realise for the first adder, as it is part of a feedback loop. This is part of the quantization level exploration, which will be addressed in Chapter 6. The second adder in `u_FOS` however, may obtain full precision by increasing the output bitwidth of one bit. The FOS has a total of $2B + C$ registers: $B + C$ for the `Z01` registers and B for its output register `Zint`. The multipliers `A01` and `B01` may also be implemented as shift operators or simply not implemented at all, based on the coefficients determining the filter characteristics. This will be explained more thoroughly in the next section.

Due to the adders operating with three terms in `u_SOS`, the internal bitwidth is increased by one bit compared to `u_FOS`. Moreover, the output of the second adder is increased by two bits to obtain full precision. Counting the delay elements and the output register `Zout`, the SOS has a total of $3B + 2C + 2$ registers.

The inter-module bitwidth B of the winner candidate implementations is chosen to be 20 bits. This can be regarded as an initial value, and will be further investigated in Chapter 6.

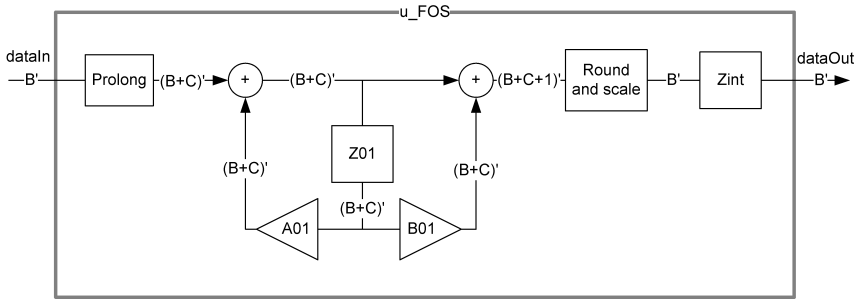


Figure 5.2: Block diagram of the first order section in *IIRFilt*

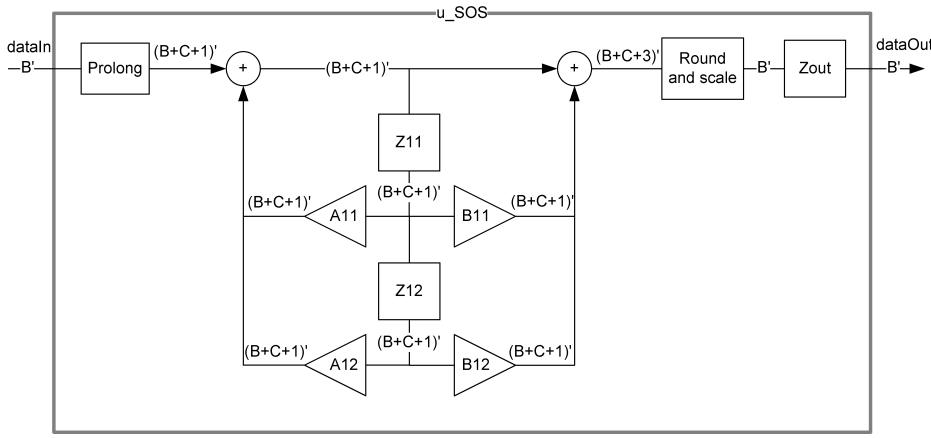


Figure 5.3: Block diagram of the second order section in *IIRFilt*

5.2 Coefficients

The only factor distinguishing one filter implementation from the next, is the set of coefficients shown in Table 5.1. As described in Chapter 4, the filters are sorted based on their coefficients, and how eligible they are for hardware implementation. Table 5.2 shows how the coefficients are implemented in RTL. Coefficients that are 1 implies that no multiplier is implemented. Coefficients that are powers of two are implemented as left or right shift operations, represented as << or >>, respectively. The remaining binary numbers imply that multipliers of the specified bitwidth are implemented. From the table, it can be seen that *IIRFilt46* and *IIRFilt21* are implemented with two shift operators and two 25-bit multipliers ($B = 20$, $C = 4$), while *IIRFilt29* and *IIRFilt30* need 26-bit multipliers. Moreover, *IIRFilt120* and *IIRFilt296* require three multipliers. The SystemVerilog implementation of *IIRFilt46* can be read in its entirety in Appendix C.1. The remaining filter implementations are not included as appendices, since all of them share the same framework.

Filter	A01	B01	A11	A12	B11	B12
IIRFilt46	0.50	1.00	1.250	−0.500	2.00	1.00
IIRFilt21	0.50	1.00	1.500	−0.750	2.00	1.00
IIRFilt29	0.50	1.00	1.500	−0.625	2.00	1.00
IIRFilt30	0.50	1.00	1.375	−0.625	2.00	1.00
IIRFilt120	0.75	1.00	1.500	−0.750	2.00	1.00
IIRFilt296	0.50	1.00	1.500	−0.750	−1.25	1.00

Table 5.1: Quantized filter coefficients in decimal

Filter	A01	B01	A11	A12	B11	B12
IIRFilt46	>>	1	01.01	11.10	<<	1
IIRFilt21	>>	1	01.10	11.01	<<	1
IIRFilt29	>>	1	01.100	11.011	<<	1
IIRFilt30	>>	1	01.011	11.011	<<	1
IIRFilt120	00.11	1	01.10	11.01	<<	1
IIRFilt296	>>	1	01.10	11.01	10.11	1

Table 5.2: Quantized filter coefficients in binary and hardware implementation

5.3 Power scenarios

The power scenarios in the analysis are called *active* and *inactive*. The *active* scenario performs an average power estimation of three time windows in simulation. The first window when applying a 1 MHz sine tone to the input of the filter, the second window when applying 2 MHz, and the last window for 3 MHz. The *inactive* scenario performs an average power estimation of the filter in a time window where the data input is zero for the entire period. However, the clock is still running in this scenario. How the power scenarios are connected to the simulation testbench is summarized in Table 3.1.

5.4 Results of the RTL implementations

Figure 5.4 shows the resulting area of the RTL implementations. It can be seen that the prediction made in AFGEC is reflected to a great extent: *IIRFilt46* has the smallest area, and the area increases in the order of the winner candidates in Section 4.4.1, at least for the Butterworth filters. However, *IIRFilt120* and *IIRFilt296* come out with smaller areas than *IIRFilt30*, even though the latter has one multiplier less. This may be the result of *IIRFilt30* having larger multipliers. The figure shows a small increase in sequential area for *IIRFilt29* and *IIRFilt30*, but the main

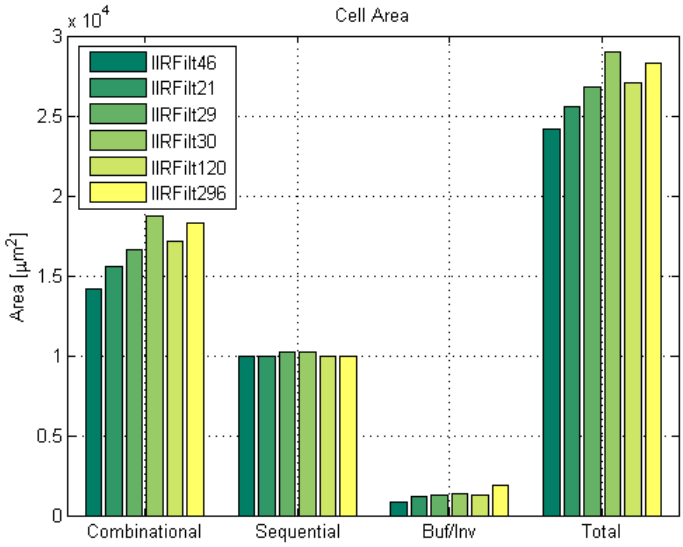


Figure 5.4: Cell area of *IIRFilt* implementations

contribution is due to combinational cell area.

Figure 5.5 shows that the number of sequential cells is right above 100. This corresponds to the calculation of $5B + 3C + 2$, which results in 117 registers for *IIRFilt29* and *IIRFilt30*, and 114 registers for the remaining implementations. The major difference is in the number of combinational cells, where the largest implementation contains 700, or 40% more than the smallest implementation. This indicates that the synthesis tool has not been able to do as many simplifications for the multipliers in *IIRFilt30*, as in *IIRFilt29* whose RTL is almost identical. It is noted that the implementations have slightly different number of clock tree buffers inferred. While *IIRFilt21* and *IIRFilt120* have 2 buffers, the rest have 3. This might affect the power dissipation.

The resulting power dissipation in the *active* scenario is depicted in Figure 5.6. The result looks very much as expected, reflecting the area in Figure 5.4. The only difference from the area results is that the power dissipation of *IIRFilt30* is slightly less than for the *IIRFilt296*. Apparently, the combinational logic causing the additional area of *IIRFilt30* do not have a high toggle rate in this scenario. When sorting the implementations with regards to power dissipation, the results yield the same order as predicted in Section 4.4.1, except for *IIRFilt120* which comes out slightly better than expected.

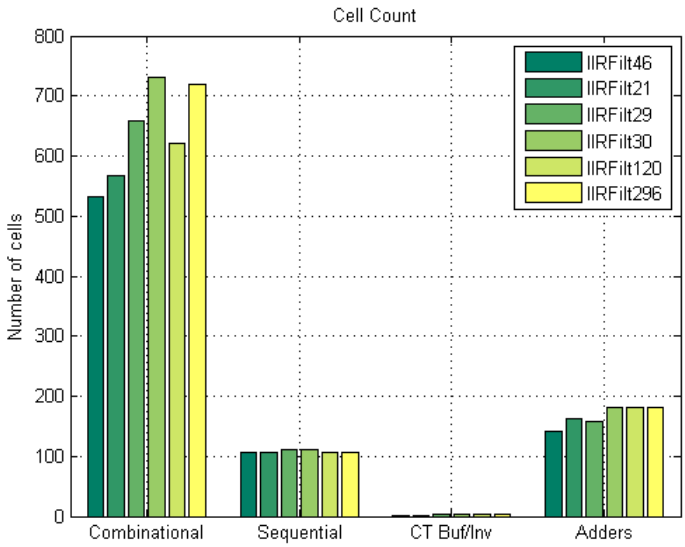


Figure 5.5: Number of logic cells in *IIRFilt* implementations

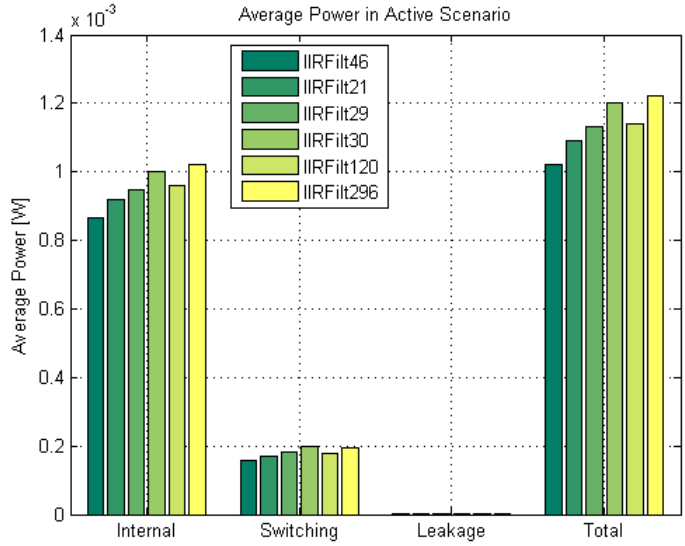


Figure 5.6: Average power dissipation in active scenario for *IIRFilt* implementations

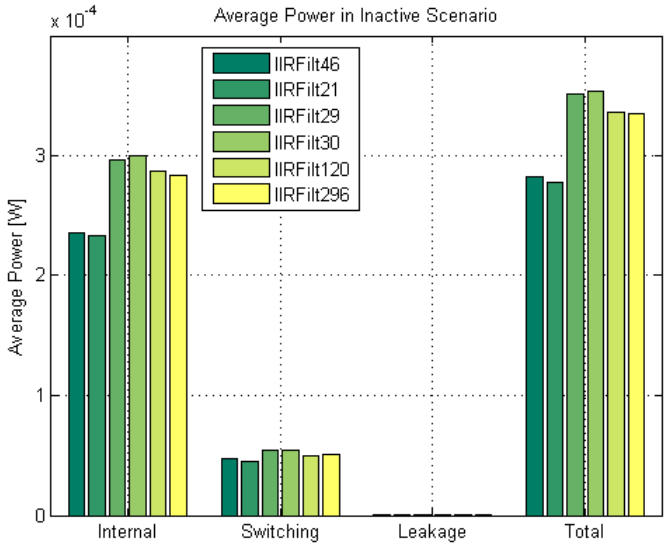


Figure 5.7: Average power dissipation in inactive scenario for *IIRFilt* implementations

The average power dissipation in the *inactive* scenario is shown in Figure 5.7. Since the reset and clock signals are the only sources of switching activity in this scenario, the power dissipation is mainly due to the clock distribution network. When comparing the results with the implementation details in Table 5.2, we see a clear pattern: *IIRFilt46* and *IIRFilt21* have the same number of registers and multipliers, and yield the same power dissipation. The same applies for *IIRFilt29* and *IIRFilt30*, which have a slightly larger number of registers, and for *IIRFilt120* and *IIRFilt296*, which have one additional multiplier.

The frequency responses in Figure 5.8 are computed from the filters impulse response extracted from simulation. This means that the plot is reflecting the function of the actually implemented filters. When comparing the frequency responses with the ones obtained in Chapter 4, the characteristic of each filter can be recognized. This indicates that the filters are implemented correctly.

5.4.1 Conclusion of the RTL implementations

The work in this chapter is carried out in order to confirm that the AFGEC algorithm is performing accurate predictions, and in order to ensure that the right candidate is chosen for further exploration. The AFGEC algorithm has proved to yield good indications of filter eligibility with regards to area and power dissipation. The most eligible candidate with regards to area and power dissipation is *IIRFilt46*, which

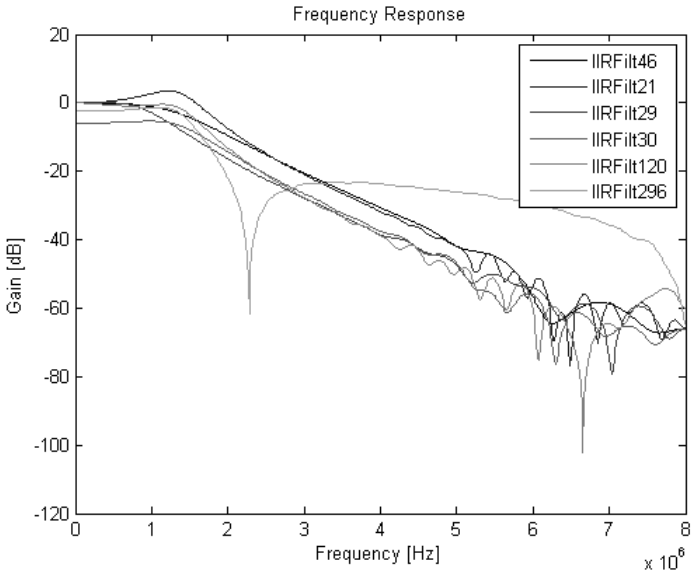


Figure 5.8: Frequency response of *IIRFilt* implementations, computed from impulse response in simulation

also was found by the algorithm. We also recall from Chapter 4 that *IIRFilt46* and *IIRFilt29* had the best potential for dynamic implementation, due to the linearity of their subsection frequency responses. This concludes that *IIRFilt46* is the most eligible filter at this point, and will be used as framework when investigating quantization levels and dynamic approaches in the next chapters.

Chapter 6

Quantization level exploration

This chapter describes the exploration of the signal bitwidths internally to the *IIRFilt46*, and the resulting amount of quantization noise that rise as these parameters are altered.

6.1 Quantization noise analysis

The quantization noise sources in *IIRFilt46* are shown in Figures 6.1 to 6.3. As described in Section 2.3, the amount of quantization noise depends on whether we perform rounding or truncation on a signal. $Q1$, $Q2$, $Q4$, $Q5$ and $Q6$ are quantization noise sources due to bit truncation, while $Q3$, $Q7$ and $Q8$ are sources due to rounding. This gives the following equation for the total quantiation noise power:

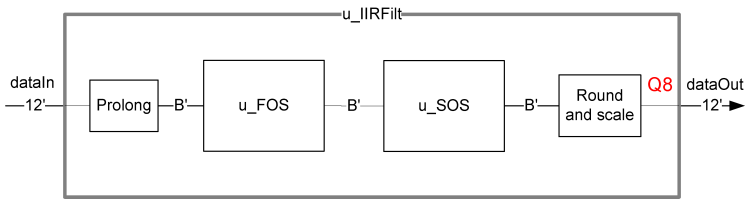


Figure 6.1: Quantization noise sources at top level

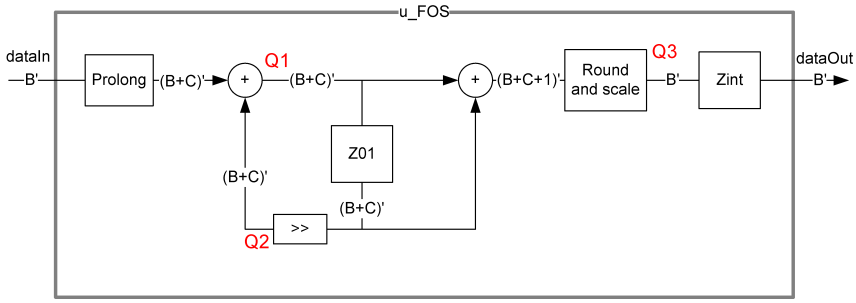


Figure 6.2: Quantization noise sources in the first order section

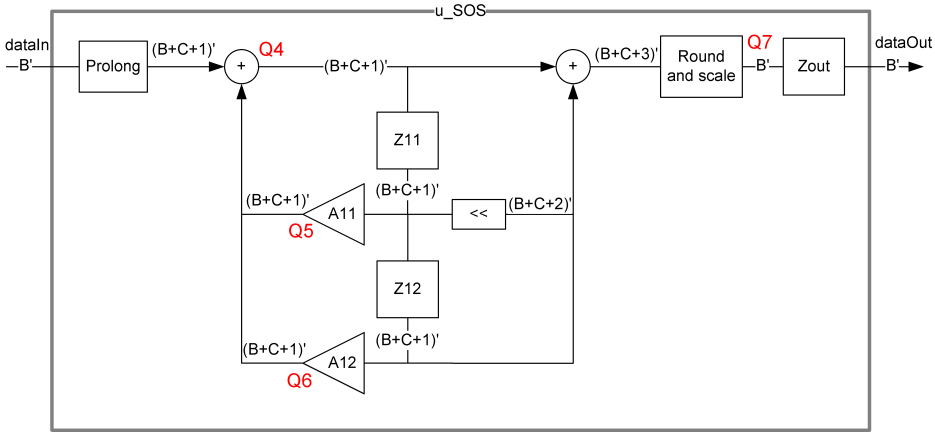


Figure 6.3: Quantization noise sources in the second order section

$$\sigma_e^2 = \frac{\Delta_1^2}{3} + \frac{\Delta_2^2}{3} + \frac{\Delta_3^2}{12} + \frac{\Delta_4^2}{3} + \frac{\Delta_5^2}{3} + \frac{\Delta_6^2}{3} + \frac{\Delta_7^2}{12} + \frac{\Delta_8^2}{12}$$

where Δ_i is the smallest difference between numbers represented by the ith quantization noise source. When filling in the bitwidth of the respective signals after quantization, we get the following equation:

$$\begin{aligned}\sigma_e^2 = &\frac{2^{-2(B+C-1)}}{3} + \frac{2^{-2(B+C-1)}}{3} + \frac{2^{-2(B-1)}}{12} + \frac{2^{-2(B+C)}}{3} \\ &+ \frac{2^{-2(B+C)}}{3} + \frac{2^{-2(B+C)}}{3} + \frac{2^{-2(B-1)}}{12} + \frac{2^{-2(11)}}{12}\end{aligned} \tag{6.1}$$

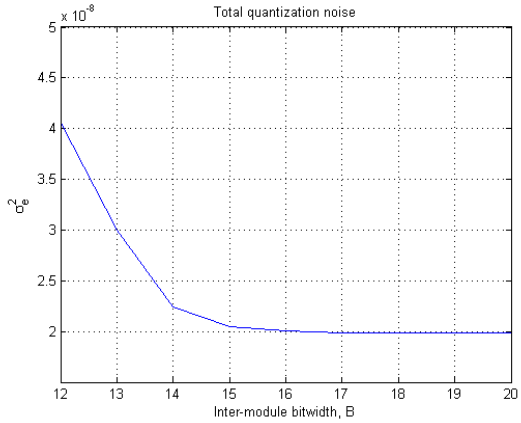
where B is the inter-module bitwidth, and C is the bitwidth of the coefficients in the multipliers. Figure 6.4a shows how σ_e^2 develops as the inter-module bitwidth is

increased from 12 to 20 bits, with C being fixed at 4. The Matlab source code of the quantization noise analysis and illustration, is found in Appendix D.1.

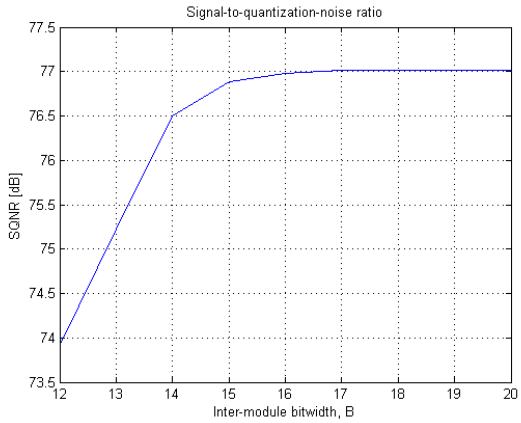
The Signal-to-Quantization-Noise Ratio (SQNR) is given by

$$SQNR = 10 \log_{10} \frac{P_x}{P_n}$$

where the power of the quantization noise $P_n = \sigma_e^2$, and the signal power is defined as $P_x = 1$. Figure 6.4b shows that the SQNR improves significantly in the interval 12 to 15 bits, and that the improvement flattens after this. The reason why the amount of noise flattens, is due to fact that the last term in Equation 6.1 is a constant. We may eliminate all other terms by increasing the inter-module bitwidth sufficiently, but due to the filter specifications we will always have to perform a rounding at 12 bits on the data output.



(a) Total quantization noise power



(b) Signal-to-quantization-noise ratio

Figure 6.4: Total quantization noise power and SQNR for different inter-module bitwidths

6.2 Implementation

In order to choose the optimal inter-module bitwidth, it is interesting to explore the reduction in area and power as B is decreased. Hence, six RTL implementations of *IIRFilt46* with different B-values are passed through the area and power estimating tool-flow, listed in Table 6.1. All implementations share the same RTL source code, shown in Appendix C.1, but with a different value for the inter-module bitwidth parameter, named `wlinout` in RTL.

Name	Inter-module bitwidth B
IIRFiltQ12	12
IIRFiltQ13	13
IIRFiltQ14	14
IIRFiltQ15	15
IIRFiltQ16	16
IIRFiltQ20	20

Table 6.1: Implementations during quantization level exploration

6.3 Results of the quantization level exploration

Figures 6.5 to 6.8 show the results of the implementations with different inter-module bitwidths. Figure 6.5 shows that the area is affected in a linear manner as B increases. Note that *IIRFiltQ17*, *IIRFiltQ18* and *IIRFiltQ19* are not implemented, which is the reason why there appears a larger step in the results between *IIRFiltQ16* and *IIRFiltQ20* compared to the rest of the implementations.

The power dissipation in the *active* and *inactive* scenarios are, similarly to the cell area, also developing in a linear manner as B increases. This is illustrated in Figure 6.6 and 6.7.

Figure 6.8 shows that the frequency response is maintained for all implementations. According to the figure, the increased amount of noise mostly affects the gain at frequencies above 5 MHz, where the attenuation is more than 40 dB anyway.

6.3.1 Conclusion of the quantization level exploration

Neither of the implementations stand out as extraordinary in terms of area, power dissipation or performance. Since the area and power dissipation develops linearly as B increases, it is chosen to determine the inter-module bitwidth based on the theoretical SQNR values. Hence, the inter-module bitwidth for further exploration is chosen to be 15 bits, since this is where the SQNR in Figure 6.4b saturates.

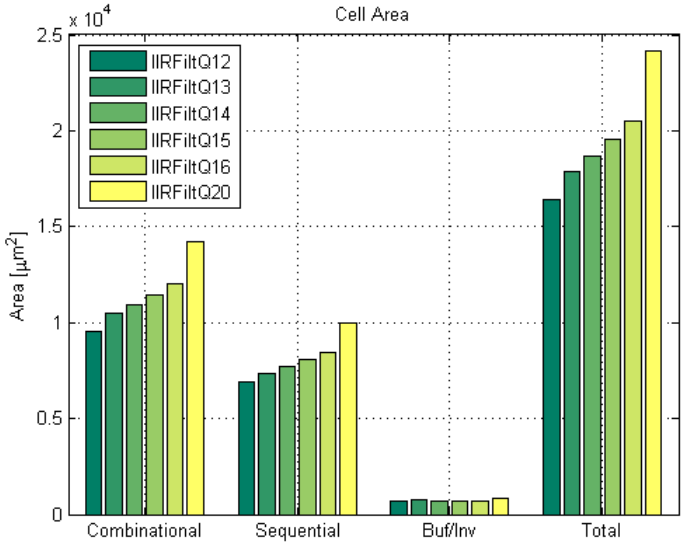


Figure 6.5: Cell area of *IIRFiltQ* implementations

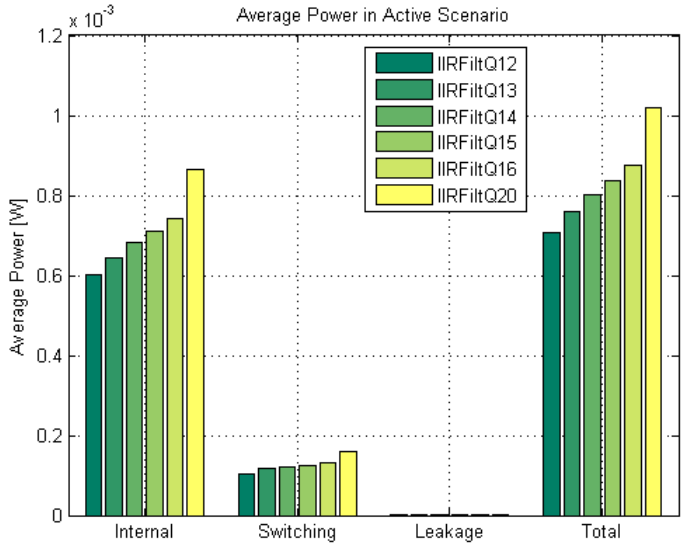


Figure 6.6: Average power dissipation in active scenario, for the *IIRFiltQ* implementations

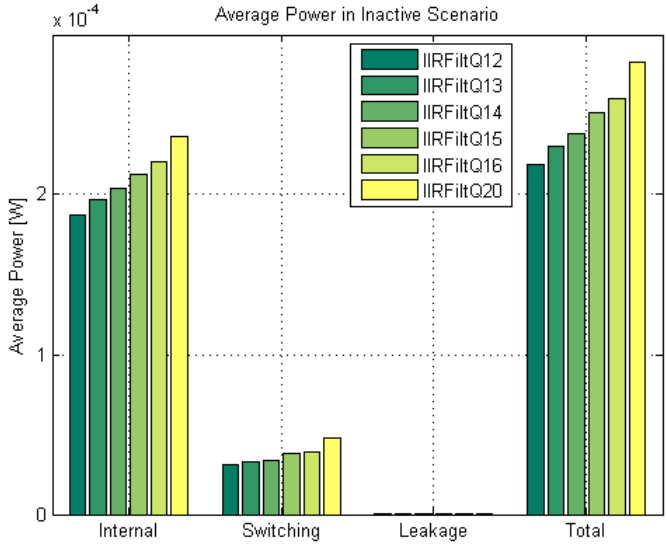


Figure 6.7: Average power dissipation in inactive scenario, for the *IIRFiltQ* implementations

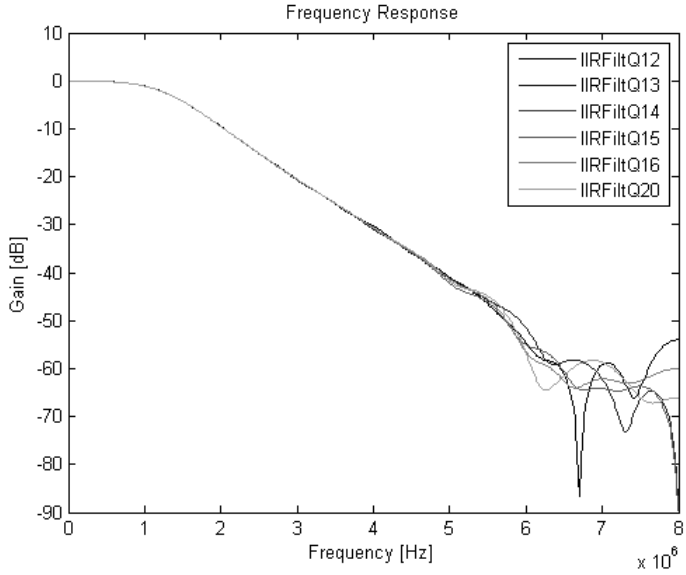


Figure 6.8: Frequency response of *IIRFiltQ* implementations, computed from impulse response in simulation

Chapter 7

Dynamic RTL

This part of the work aims to design a channel filter which adapts its performance dynamically according to its surroundings. For instance, the filter may degrade its performance under ideal conditions in order to save energy, while it under worse conditions might need to use all its resources in order to perform satisfactory. This chapter describes two main approaches in making the filter dynamic with regards to performance and power dissipation. Each approach is realised in several slightly different implementations, which is passed through the area and power estimating toolflow in order to find the optimal solution.

7.1 Implementation

An effective way of saving energy is to reduce the amount of switching activity. Switching activity in registers can be eliminated by preventing the clock to propagate, using integrated clock gating cells. Unwanted switching activity in combinational cells however, can be eliminated with datapath gating. As described in Section 2.5.2, this can be realised by inserting latches at the data input. However, other constructs can provide the same functionality. For instance, by inserting a MUX at the input of the combinational block which selects a signal of all zeros when disabled, or by clock gating a register at the input of the combinational block. The following two ideas take use of these methods in order to reduce the power dissipation while systematically adjusting aspects of the filter performance:

Dynamic Filter Order Disabling all switching activity in an entire biquadratic section with clock and/or datapath gating in order to reduce power dissipation. Disabling the sections will affect the order of the filter, and hence the slope of the frequency response. Increased area will be due to multiplexers, ICG cells and control logic.

Dynamic Quantization Noise Disabling the N least significant bits in every register in both subsections in order to save power. This will affect the arithmetic precision of the filter, and hence the amount of quantization noise. Increased area will be due to multiplexers, ICG cells and control logic.

7.1.1 Modes of performance

In order for the filter to know what mode of performance that is needed at all times, it is added an input control signal, `mode`, to the module. As illustrated in Figure 7.1, the control signal is assumed to originate from a *Mode selector*. The mode selector receives a signal from a module further down the signal chain, here called the *Link quality estimator*, which holds a quantized value of the quality of the radio link, for instance the *bit error rate*. This signal is passed on to the mode selector which then decides which mode of performance that is required. As seen in the figure, the `mode` signal is imagined to not only be passed on to the channel filter, but also to other dynamic modules using the same interface. This would allow an entire system to become dynamic, and hence save even more energy in the typical case. Designing the *link estimator* is considered to be out of the scope of this thesis, so the `mode` signal is therefore treated as a known input signal. The signal has two bits, which represents the following four modes:

High Performance The high performance mode will utilize all available resources, and hence consume the most power. It will provide the same performance as the non-dynamic filter *IIRFiltQ15* in Chapter 6, but is expected to have a slightly larger power dissipation due to the overhead of the dynamic logic. The channel filter will take use of this mode only when the radio link is poor.

Moderate The moderate mode will free some of the available resources. It will provide slightly worse performance, but is expected to have a noticeably lower power dissipation. It is assumed that the channel filter will use this mode of performance when the radio link is decent.

Low Performance The low performance mode will free a larger part of the available resources. It will just provide an adequate performance, but is expected to have a significantly lower power dissipation. It is assumed that the channel filter will use this mode when the radio link is very good.

Bypass The bypass mode will free all resources except the dynamic logic overhead. The performance is equivalent to not using any filter at all, and the power dissipation should therefore be close to zero. It is assumed that the channel filter will use this mode only when the radio link is so good that an all-pass filter will do the job.

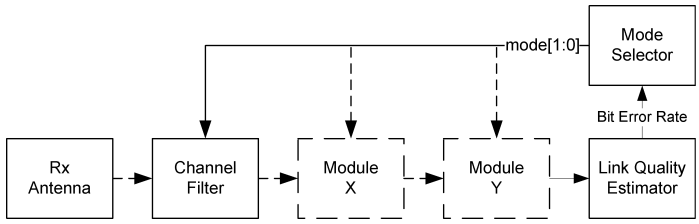


Figure 7.1: Link quality estimator and mode selector provides the dynamic modules with the current mode of operation

7.1.2 Dynamic filter order

All the dynamic filter order implementations take use of the same control signals, shown in Table 7.1. `enaSOS` is the control signal that enables the second-order section, while `enaFOS` enables the first-order section. The highest performance is achieved when enabling both sections, which yields a third order filter. The second best performance is achieved when enabling only the SOS, yielding a second order filter. The third best performance when enabling only the FOS, resulting in a first order filter. This can be realised with a direct mapping of the bits in the `mode` signal.

`mode`	Mode name	`enaSOS`	`enaFOS`
11	High performance	1	1
10	Moderate	1	0
01	Low performance	0	1
00	Bypass	0	0

Table 7.1: Control signals in dynamic filter order implementations

DynOrder

The basic implementation is called *DynOrder*, and is illustrated in Figure 7.2. It takes use of two integrated clock cells, one for each of the sections in the filter. The clock gates are enabled by the control signals in Table 7.1. As the registers are gated, they also need to be reset in order to ensure that the output freeze at zero. This is done by the `Reset Logic` blocks, which also are enabled by `enaFOS` and `enaSOS`. The reset logic blocks hold the simple boolean expressions:

```
rst_gate0 = rst OR !enaFOS
rst_gate1 = rst OR !enaSOS
```

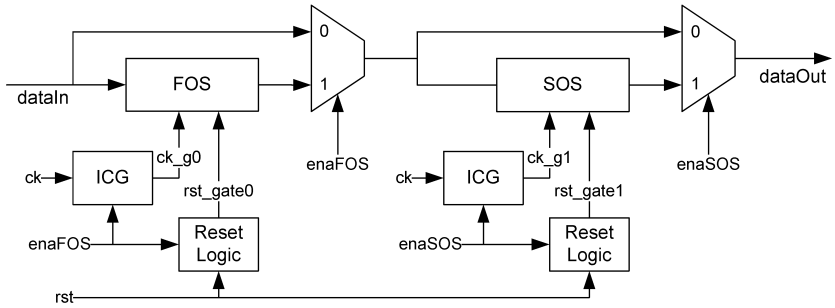


Figure 7.2: Implementation with dynamic filter order, *DynOrder*

As the registers are freezed at zero, no toggling will occur in the multipliers, as shown in Figure 7.3. This implies that `dataIn` will flow through the circuit, as only zeroes are added in both junctions. However, due to the `Round and scale` block, `dataOut` would have become a downscaled version of `dataIn` in the case where `u_SOS` is bypassed. This would have degraded the SQNR. Hence, the multiplexers at the output of the FOS and SOS in Figure 7.2 are added to the circuit. The drawback of this is that it will increase the total area, but the gain from it is twofold: Firstly, we may clock gate the output register in Figure 7.3 and save additional power. Secondly, the input of a bypassed section may be passed on to the output without any increase in noise.

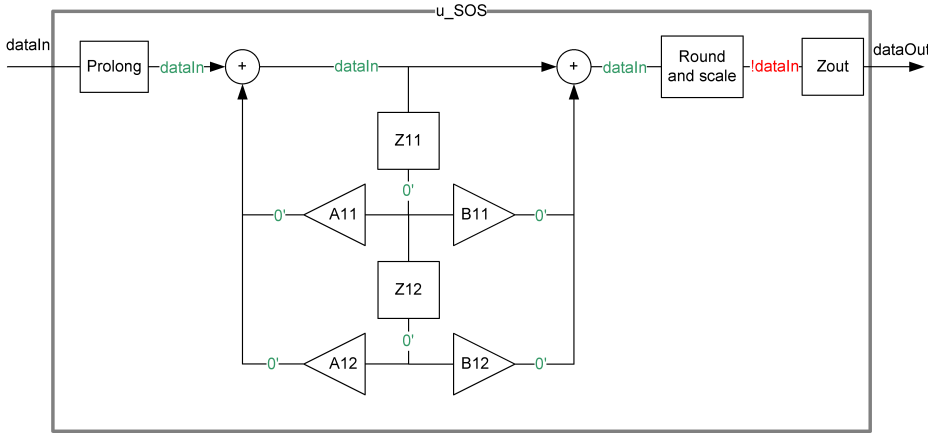


Figure 7.3: Analysis of second order section in *DynOrder* when inactive

DynOrderMux

The implementation named *DynOrderMux* is an extention of *DynOrder*, illustrated in Figure 7.4. It aims to reduce the switching activity even more by introducing additional logic. As seen in Figure 7.3, there will still occur some toggling in the adders and scaling logic even though the registers are clock gated. This toggling can be eliminated by gating the input datapath. This is done in *DynOrderMux* by inserting a multiplexer at the input of each section, which selects only zeroes if the section is disabled. The additional multiplexers will imply larger area and also slightly increased power dissipation, but the total power reduction are expected to be larger. Figure E.1 shows how the clock and reset signals are distributed to the registers of *DynOrder* and *DynOrderMux*. The SystemVerilog source code of *DynOrderMux* is shown in Appendix E.1.

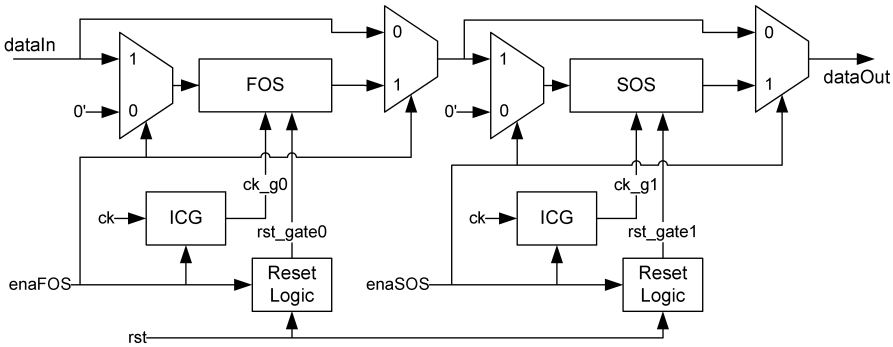


Figure 7.4: Implementation with dynamic filter order, including datapath gating using input MUX, *DynOrderMux*

DynOrderReg

The implementation named *DynOrderReg* has the same motivation as *DynOrderMux*, but has a different architecture. Instead of using input multiplexers to gate the datapath, it uses input registers. The output registers internally to each section are moved to the input of the section, as shown in Figure 7.5. Hence, the datapath is automatically gated as the input register is clock gated. However, when removing the output register of `u_SOS`, the combinational logic depth prior to the global output will increase. Hence, an additional register is inserted at the output of the filter to make the implementation more comparable. Figure E.2 shows how the clock and reset signals are distributed to the registers of *DynOrderReg*.

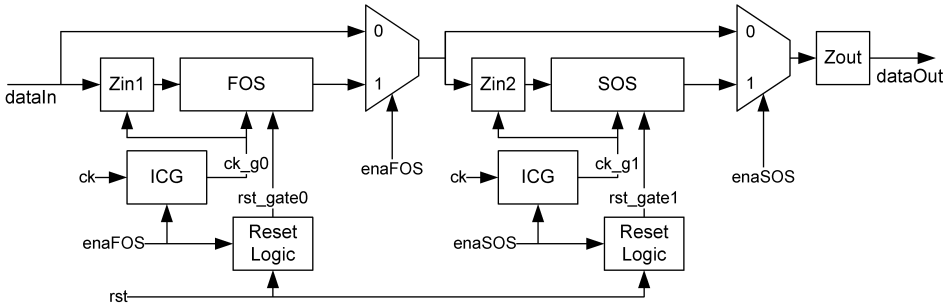


Figure 7.5: Implementation with dynamic filter order, including datapath gating using input registers, *DynOrderReg*

7.1.3 Dynamic quantization noise

All dynamic quantization noise implementations takes use of the same control signals, listed in Table 7.2. Here, each control signal is assigned to a set of registers based on their significance. `enaMSB` enables the most significant bits, `enaLSB` enables the least significant bits, and `enaMid` enables the remaining bits in the middle.

`mode`	Mode name	`enaMSB`	`enaMid`	`enaLSB`
11	High performance	1	1	1
10	Moderate	1	1	0
01	Low performance	1	0	0
00	Bypass	0	0	0

Table 7.2: Control signals in dynamic quantization noise implementations

Figure 7.6 illustrates how a register of wordlength *WL* is divided into sections *MSB*, *Mid* and *LSB* using the limit parameters *Llim* and *Rlim*. By changing the values of the limit parameters, the amount of quantization noise will change accordingly in the different modes of performance. Implementations with different limit values will be explored. Gating the *LSBs* or *Mids* implies a truncation at index *Rlim* or *Llim* respectively. This creates more quantization noise than rounding, but does not require any additional logic. Hence, implementations are made with both truncation and rounding in order to do trade-offs with regards to noise, area and power dissipation.

Table 7.3 lists all the implementations exploiting dynamic quantization noise, and states which limit values and quantization technique that are used in each case. All implementations have the same framework, illustrated in Figure 7.7. From the figure, it can be seen that three ICG cells are utilized: one for each section of significance in the registers. The ICG cells are enabled by a decoder module, which translates the mode signal into control signals according to Table 7.2. The same

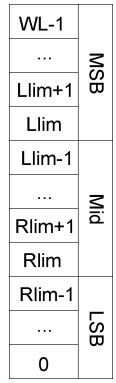


Figure 7.6: Register of wordlength *WL* is divided into sections of significance

Name	Llim	Rlim	Trn/Rnd
DynNoise8Trn	8	4	Truncation
DynNoise12Trn	12	6	Truncation
DynNoise8Rnd	8	4	Rounding
DynNoise12Rnd	12	6	Rounding

Table 7.3: Dynamic quantization noise implementations

control signals are passed on to the `Reset Logic` block, similar to the one in the *DynOrder* implementations. All reset and clock signals are then fed to both the FOS and the SOS, since all registers in the entire filter will be gated equally. The multiplexer at the end ensures that `dataIn` is passed through the filter as the *bypass* mode is selected. In Figure E.3 in Appendix E.3, a more detailed overview is given of how the clock and reset signals are distributed to the registers in the design. Here, the registers named `Zxx` represents both `Z11` and `Z12`. The SystemVerilog source code of *DynNoiseRnd12* is shown in Appendix E.2.

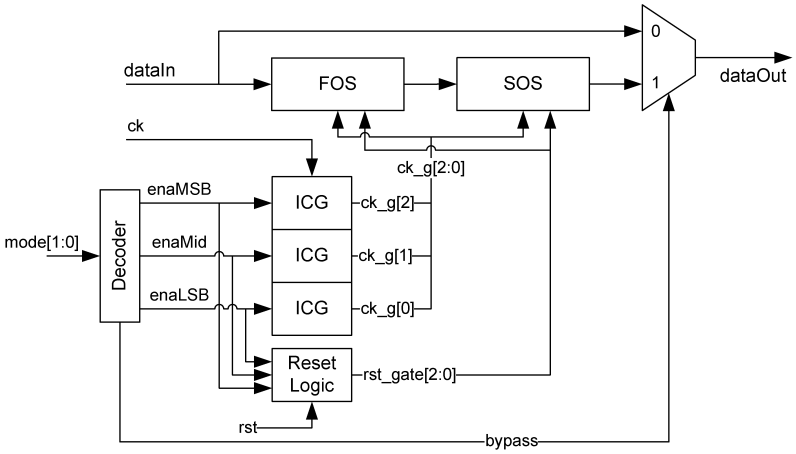


Figure 7.7: Implementations with dynamic quantization noise, *DynNoise*

7.1.4 Power gating

In order to further reduce the power dissipation in some of the performance modes, one could introduce power gating. This can be done by gathering all cells that are clock- and datapath gated by the same enable signal, into a separate power domain. One power domain should be on at all times, while the rest could be controlled by power switches. Inserting the power switches would introduce some leakage power and slightly increase the area. However, the power reduction obtained from shutting down all cells that are inactive, should be larger. A complete implementation of power gating is considered to be beyond the scope of this thesis. However, a theoretical analysis of how one of the implementations would be affected by power gating is carried out. This analysis is made based on the reports generated by PrimeTime-PX, which logs the power dissipation of every cell in the design. Estimating the effect of power gating is done in the following order:

1. Find all cells that have zero switching power, and set the internal and leakage power of these cells to zero as well. These cells are most likely clock gated, datapath gated or disabled ICG cells.

2. Find out how many power switches that are required in order to drive the amount of cells in the design. Add the leakage of the power switches to the total power dissipation. Table 7.4 lists some typical details of a power switch cell in 180 nm technology.

3. Find the unused clock tree buffers in each scenario, and set its switching-, internal- and leakage power to zero.

Parameter	Value
Leakage power	40 pW
Area	126 μm^2
Fanout	100 cells

Table 7.4: 180 nm technology power switching cell details

The analysis is carried out for the *DynOrderMux* implementation, and Figure 7.8 shows how the design would look with power switches and power domains included. Three power domains are established: `PD_FOS` and `PD_SOS` are assigned to each of the sections, and are controlled by separate power switches `PS_fos` and `PS_sos`. `PD_ON` in an *always on* domain, where the cells that should be powered on at all times should be placed, like the power switches and the output multiplexers of each section.

Note that isolation cells are not included in the analysis, although this should be done when implementing the power gated design. This will imply a slight increase in area.

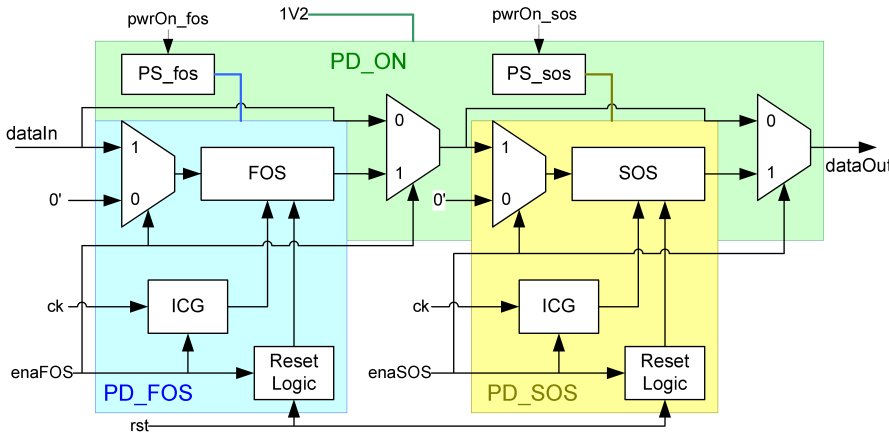


Figure 7.8: Power gating analysis of the *DynOrderMux* implementation

7.2 Results of the dynamic implementations

This section presents the results of the dynamic *DynOrder* and *DynNoise* implementations, and the theoretical results of the power gated implementation *DynOrderMuxPG*.

7.2.1 Dynamic filter order results

Figure 7.9 shows the resulting frequency responses of the dynamic filter order implementations. It can be seen that all implementations yield identical characteristics.

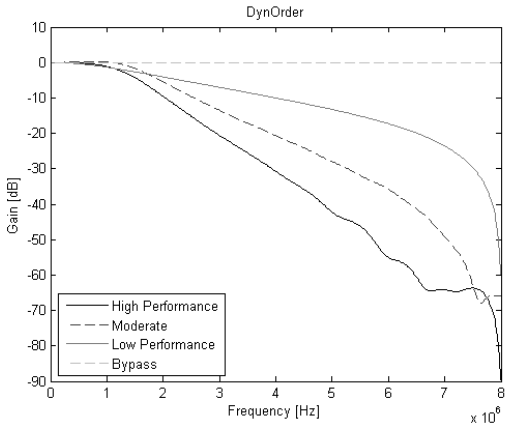
This is due to the fact that they all perform the same calculations with the same internal bitwidths, yielding the same impulse response from RTL simulation. What differentiates the implementations is only the way they perform gating of redundant switching activity.

It can be seen from the figure that the different modes of performance yield different levels of attenuation. *High Performance* has the steepest curve and the largest attenuation in stopband. The *Moderate* and *Low Performance* modes have more relaxed curves with less attenuation in stopband. These modes also provide evenly shaped frequency responses, without bumps or irregularities. The *Bypass* mode is indeed true bypass, which provides 0 dB attenuation for all frequencies.

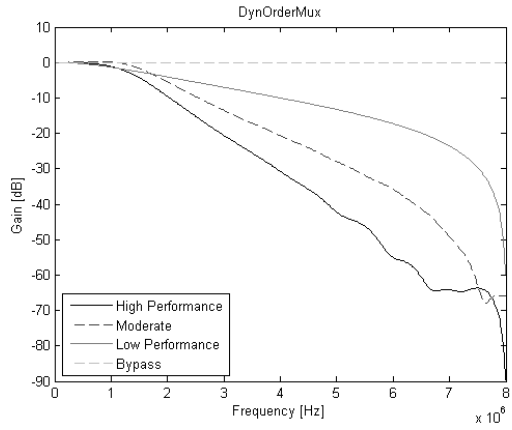
In Figure 7.10c, the total cell area in the dynamic order implementations are compared against the static reference implementation *IIRFiltQ15*, whose performance is identical to the *High Performance* mode in this chapter. It can be seen that *DynOrder* requires the smallest increase in area, since this implementation do not include datapath gating. *DynOrderReg* requires the largest increase in area due to additional output registers. However, all dynamic order implementations can be implemented with less than 10% increase in area.

Figure 7.10a shows the total average power dissipation of each of the dynamic filter order implementations, for each of the modes of performance. The horizontal red line marks the power dissipation of the reference implementation, *IIRFiltQ15*. The figure shows that all implementations consume around 10% more power in *High Performance* than the static reference. This is of course expected due to the overhead logic that is required to make the implementations dynamic, which is also seen in the total area results. Moreover, it can be seen that *DynOrderMux* yields the best results in terms of power reduction. Except from the *High Performance* mode, it has the lowest power dissipation in all modes of performance, yielding 28%, 55% and 88% reduction from reference.

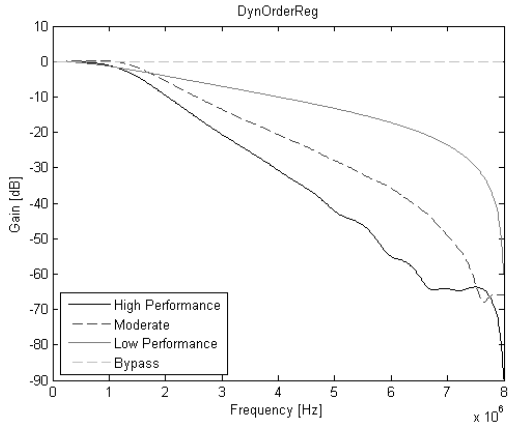
In the *inactive* scenario results, visualized in Figure 7.10b, we see that the power dissipation in idle state can be reduced drastically by implementing the dynamic order approach. *DynOrder* and *DynOrderMux* have the largest savings in this scenario, with a reduction of 77% from reference.



(a) *DynOrder* implementation

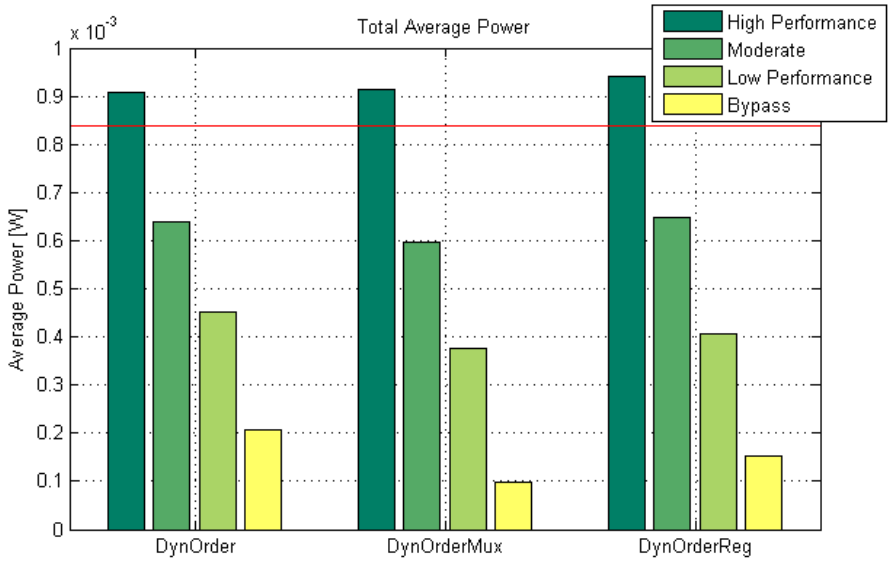


(b) *DynOrderMux* implementation

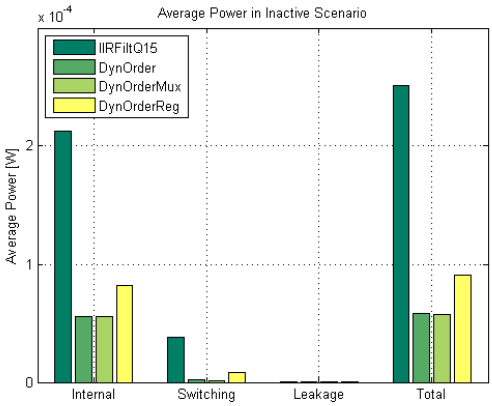


(c) *DynOrderReg* implementation

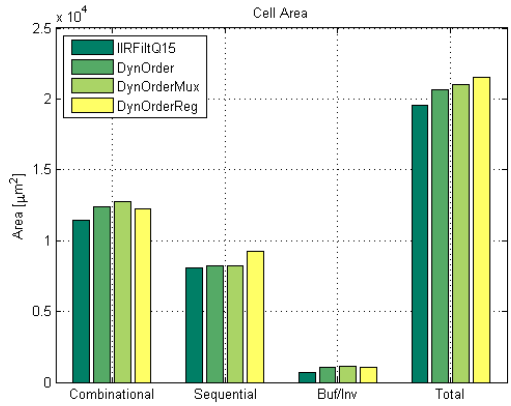
Figure 7.9: Frequency responses of the dynamic filter order implementations



(a) Total average power



(b) Total average power in inactive scenario



(c) Cell area

Figure 7.10: Power and area results of the dynamic filter order implementations

7.2.2 Dynamic quantization noise results

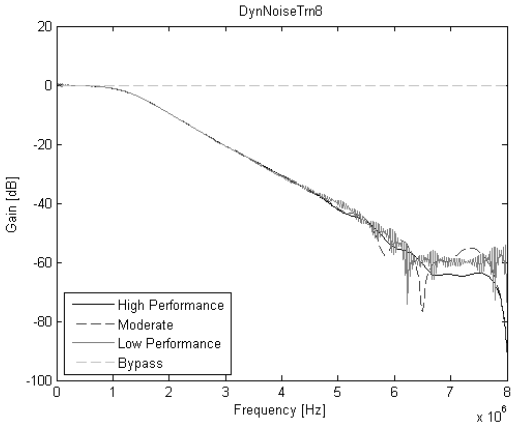
The frequency responses of the dynamic quantization noise implementations are shown in Figure 7.11. Unlike the dynamic *order* implementations, which provided different levels of attenuation, the dynamic *noise* implementations offer the same attenuation for every mode of performance. However, the *Moderate* and *Low Performance* modes introduce more quantization noise, which affects the impulse response and thus the frequency response.

It can be seen that *DynNoiseTrn12* and *DynNoiseRnd12* have larger fluctuations in the frequency response for the *Moderate* and *Low Performance* modes than *DynNoiseTrn8* and *DynNoiseRnd8*. This is due to the different number of inactive registers, given by *Mid* and *LSB*. Moreover, it is observed that *DynNoiseRnd12*, which perform rounding, yields a smoother frequency response in *Low Performance* mode than *DynNoiseTrn12*, which performs truncation. This is expected, as truncation generally implies more noise than rounding.

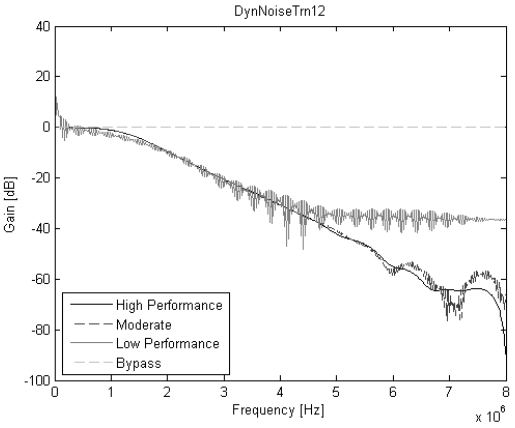
Figure 7.12c shows the total cell area of the dynamic noise implementations, compared against the static reference implementation *IIRFiltQ15*. The figure shows that the dynamic noise approach can be implemented with less than 5% overhead if truncation is used. However, if rounding is preferred, the overhead is increased to almost 20%. Anyhow, the amount of additional logic is at an acceptable level. The results also show that the partitioning of *MSB*, *Mid* and *LSB* have minimal impact on the total area. This is also expected, as the amount of registers and control logic are close to identical when performing the same quantization method.

The total average power dissipation of the dynamic noise implementations are shown in Figure 7.12a. The resulting power is given for each mode of performance, with the *IIRFiltQ15* reference illustrated as the horizontal red line. As for the dynamic *order* implementations, the dynamic *noise* implementations also yield increased power dissipation around 10% for the *High Performance* mode, naturally. Moreover, it can be seen that the *Moderate* and *Low Performance* modes are not able to provide the same reduction as in the dynamic *order* approach. However, the dynamic *noise* implementations offer the attenuation of a third order filter for every mode of performance. Out of the implementations in Figure 7.12a, *DynNoiseTrn12* and *DynNoiseRnd12* yield the best results in terms of power reduction. They both provide low power modes where the power dissipation is significantly reduced for each mode. More specifically, the reductions are 9%, 34% and 83% for *DynNoiseTrn12*, and 11%, 32% and 81% for *DynNoiseRnd12*.

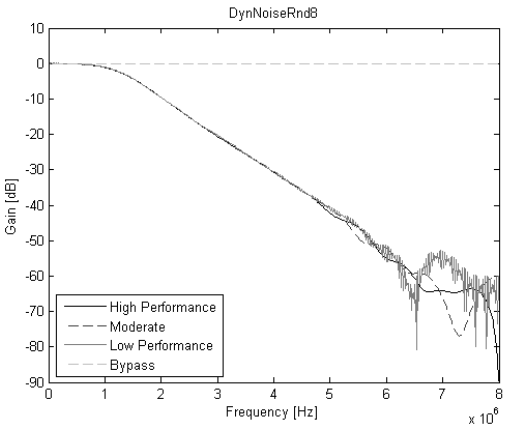
Figure 7.12b shows the total power dissipation in *inactive* scenario. It can be seen that all implementations succeed in reducing the power dissipation in this state, similarly to the dynamic *order* approach, all yielding reductions around 75%.



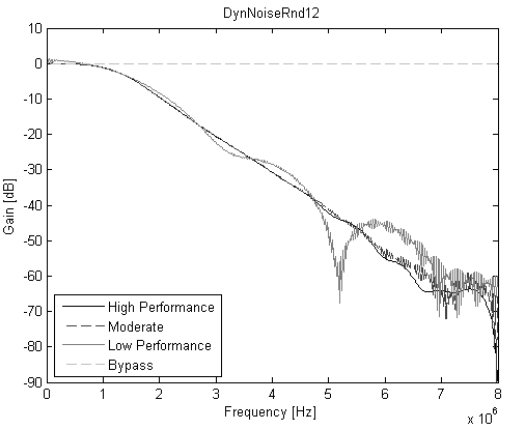
(a) *DynNoiseTrn8* implementation



(b) *DynNoiseTrn12* implementation

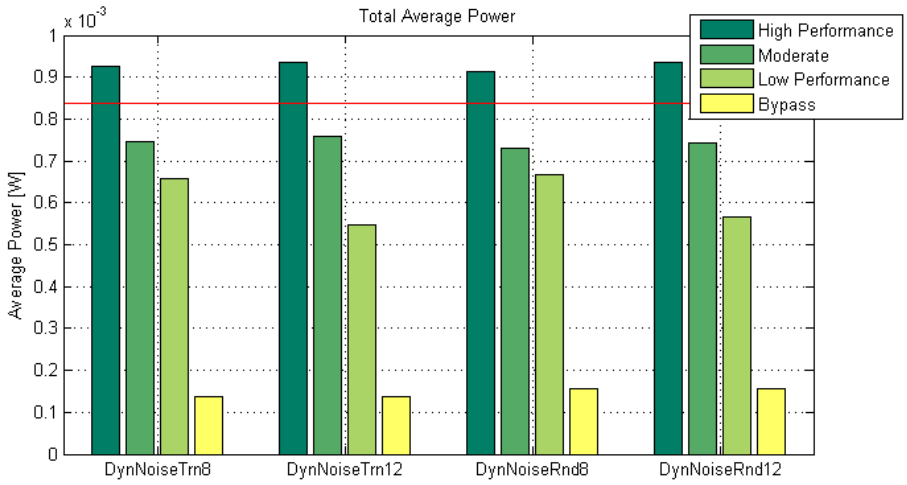


(c) *DynNoiseRrn8* implementation

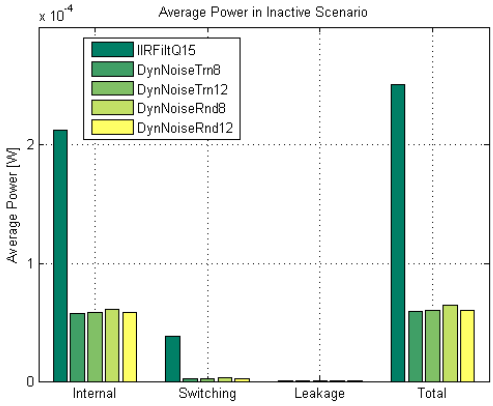


(d) *DynNoiseRrn12* implementation

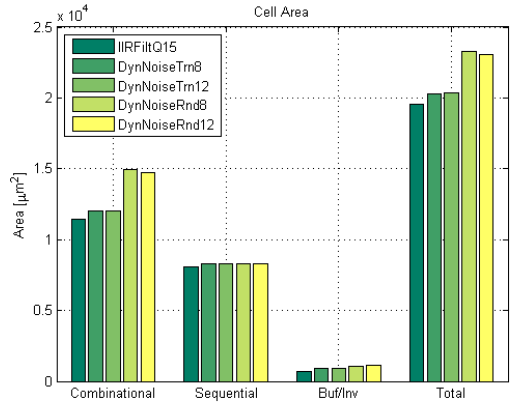
Figure 7.11: Frequency responses of the dynamic quantization noise implementations



(a) Total average power



(b) Total average power in inactive scenario



(c) Cell area

Figure 7.12: Power and area results of the dynamic quantization noise implementations

7.2.3 Power gating results

Figure 7.13 compares the results of the dynamic order implementation *DynOrderMux*, and the theoretical results of power gating the same implementation. Figure 7.13c shows that the increase in area is negligible. The increase is due to additional power switching cells, which is estimated to be one per hundred leaf cell. This adds up to 5 power switches in this particular design. However, it is important to take into account that the cells would have to be physically separated into different power domains. This will restrict the optimization in layout, which again will affect the total area. How much the area will increase due to this is however hard to predict, but an additional 5-10% can be assumed.

The total average power dissipation in each mode of performance is shown in Figure 7.13. From the figure, it can be seen that the power dissipation may be reduced by 58% in *Bypass* mode. The effect of power gating is negligible in the other modes of performance.

However, in the *inactive* scenario, the power gating may potentially eliminate the power dissipation completely, as shown in Figure 7.13b. Normally, the only activity in this scenario is due to the propagating clock signal, where the main consumer is the primary clock buffer. However, when the clock buffers are power gated, the power dissipation is driven to a minimum.

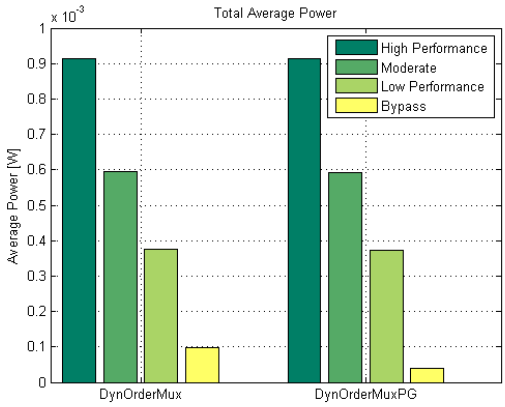
7.2.4 Conclusion of the dynamic implementations

All dynamic implementations yield promising results, with less than 10% increase in power dissipation for the *High Performance* mode, and within 20% increase in area. Most of the solutions provide significant power reduction in each of the performance modes.

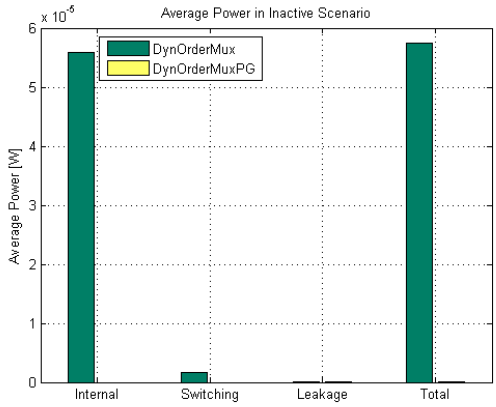
The dynamic *order* implementations yield the largest reductions in each mode. Especially *DynOrderMux*, which manages to reduce the power dissipation of 28%, 55% and 88% in each of the low power modes, with an area penalty of only 8%. However, the performance is reduced to second order, first order and all pass filters, respectively.

The dynamic *noise* implementations do not manage to provide reductions at the level of the dynamic *order* implementations. However, they provide the performance of third order filters in all modes, although the amount of noise is increased. *DynOrderRnd12* manages to maintain the frequency response in *Low Performance*, while providing significant power reductions in each mode, at an acceptable area increase of 18%.

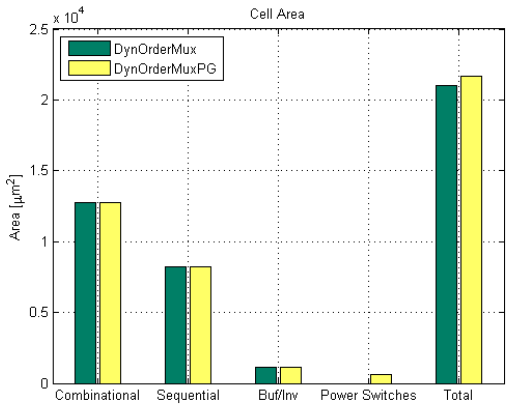
Power gating has shown to have minimal effect on the power dissipation in the



(a) Total average power



(b) Total average power in inactive scenario



(c) Cell area

Figure 7.13: Estimated power and area results of a power gated implementation

High, *Moderate* and *Low Performance* modes. In the *Bypass* mode however, it manages to reduce the power dissipation of 58% when the data input is active, and almost 100% when the data input is *inactive*. Although the power dissipation in this scenario already is of a small magnitude, the possible energy saving may become significant over time. The increase in area is also negligible at first glance, although the area increase of separating the logic into different power domains needs to be taken into account.

Chapter 8

Discussion

8.1 Algorithm-architecture co-design and platform level implementation

The fundamental challenge in algorithm-architecture co-design is to find the right balance between two disciplines, which requires in-depth knowledge in both. That is, just the right abstraction level in the algorithm, and just the right flexibility in the architecture. One step in the wrong direction for one discipline can diminish the achievements made on the other one.

In this work, the filter algorithm is generalized to a cascade of first- and second-order sections, whose abstraction allows for any filter requirement. This modular structure eases the flexibility in performance and energy consumption, which is required for dynamic implementation. The same reasoning can be applied to other parts of the radio, mainly to those handling the digital baseband processing. Typically, the modules in the digital baseband processing are dictated by a high throughput, with little or no control, and few, very specialized DSP cores.

The modular construction of the proposed solution allows for the various performance modes (*High Performance*, *Moderate*, *Low Performance* and *Bypass*) that can be shared among other parts of an entire product or platform, which was illustrated in Figure 7.1.

Most radio communication standards provide means to measure the quality of the

communication link, as Bluetooth's RSSI [2] and LTE's CQI [6]. While these metrics are typically used to dictate the amount of energy used on the power amplifiers, it can also be used to regulate internal energy consumption.

The number and types of modes needed in the entire platform are a consequence of the combination of two factors: namely, the communication quality metrics and the throughput requirements. The first given by the environment the radio is immersed into, and the latter given by the user or application requirements. These modes are used as potential energy regulators for the entire radio.

Similar studies to the one detailed in this thesis are needed to other parts of the radio in order to define the correct partition of all modules involved in the communication, so that a complete solution at platform level can be achieved.

8.2 Evaluation of dynamic implementations

8.2.1 Dynamic order

The dynamic filter order approach in Chapter 7 showed that the cascade-form structure made it easy to bypass parts of the filter in order to alter its performance. The results in the same chapter also showed that the saving in power dissipation was significant when disabling a biquadratic section.

The disadvantage with this approach is that the stopband attenuation becomes significantly decreased when shutting down an entire second-order section. However, this may be tolerated in some applications where there are small chances of interfering signals appearing on the channel.

The filter structure is also scalable, as the dynamic third order filter easily could have been extended to a dynamic filter of a higher order, by increasing the number of second-order sections in series. Instead of switching between filters of order 1, 2 and 3, one could implement three second-order sections and switch between filters of order 2, 4 and 6. By disabling two or more biquadratic sections for each mode of performance, instead of one, the modes could also provide larger differences in terms of power reduction. This would of course increase the total area and the power dissipation when all resources are active.

8.2.2 Dynamic noise

The dynamic quantization noise approach in Chapter 7 do not depend as much on the cascade-form structure. Its principle is more generic, and does not require a serially structured data flow. This approach could have been implemented in any module where the amount of quantization noise is an issue.

The disadvantage of this approach is that the power reductions are more moderate compared to the dynamic order approach. In order to obtain large power reductions, a relatively large part of the registers needs to be clock gated, which greatly affects the amount of quantization noise. For instance, we saw that in order to achieve a power reduction of 35% in *Low Performance* mode, as much as 60% of all registers needed to be gated. And if more than 60% of the registers were to be clock gated, the quantization noise would eventually become too dominant.

A great advantage with the dynamic noise approach is that it offers the same stopband attenuation for each mode of performance. Consequently, the filter may maintain its function in each of the modes, as long as a certain amount of noise can be tolerated. This may be preferred in some applications, where the attenuation at certain frequencies is of high importance.

The implementation could of course be extended to a filter of higher order in order to improve the stopband attenuation additionally. The power reduction principle would be the same in either implementation.

8.2.3 Possible combined solution

A possible cross-layer solution could be to combine the dynamic order approach and the dynamic noise approach. A natural implementation would be to provide different amount of quantization noise for each filter order configuration. With the current implementations, this would mean 4 performance modes for each of the 2 biquadratic sections, which would imply a total of 16 possible modes of performance. Such a large number of modes is of course inconvenient, and would increase the complexity of the control logic significantly.

However, the implementations could be adjusted to provide a more suiting number of modes. For instance, the first-order section could be implemented with the dynamic order approach, while the second-order section could be implemented with a two-step dynamic quantization noise approach. This would give the four modes given in Table 8.1, which would require one clock gate for each of the biquadratic sections.

Mode	FOS	SOS
High performance 3rd order	On	Minimal noise
Low performance 3rd order	On	Moderate noise
High performance 2nd order	Off	Minimal noise
Low performance 2nd order	Off	Moderate noise

Table 8.1: Possible combined dynamic order and dynamic noise solution

8.3 Thoughts around future work

8.3.1 Fine-tune the AFGEC algorithm

The filter generation and eligibility calculation algorithm in Chapter 4 managed to find the most area and energy efficient filter solutions. The six solutions with the highest eligibility score were implemented in RTL, where five of the solutions were ranked in the same order as predicted. However, the algorithm could perhaps be calibrated even better, such that six out of six solutions are predicted in the correct order. In order to improve the prediction, the eligibility calculation could be adjusted with scaling factors for each of the characteristics. A possible solution could be to decrease the impact of *single-one coefficients*, or increase the impact of *number of bits*. Such a fine-tuning of the eligibility calculation in the AFGEC algorithm could provide an even higher percentage of accurate predictions.

8.3.2 Include power gating in tool-flow and implement

The theoretical analysis of power gating in Chapter 7 showed that power gating might potentially eliminate the power dissipation of the channel filter when it is inactive, and that the increase in area might be negligible. However, it would have been interesting to put these results to the test by actually implementing the *DynOrderMuxPG*. But first, the area and power estimating tool-flow should be adjusted to support power gating. This could be done by creating a default setup of power domains and power switches according to the modes of performance. Afterwards, the power gated design could be run through the updated tool-flow.

8.3.3 Extended quantization level exploration

The quantization level exploration in Chapter 6 was helpful in selecting the optimal inter-module bitwidth. It showed that the SQNR could only be improved to a certain point when working with a fixed data output requirement of 12 bits, and that the improvement saturated around the inter-module bitwidth of 15. What could have been interesting however, was to analyze the design in a more fine-grained manner. This could be done by creating a program that analyzed every possible combination of bitwidths for each of the noise sources in the design. Certain constraints regarding the bitwidths should of course be set, like the maximum bitwidth allowed, in order for the program to finish within a reasonable amount of time. Such an extended quantization level exploration could lead to find an even more optimized solution in terms of SQNR, area and power dissipation.

8.3.4 Similar studies

The dynamic filter is envisioned to be a part of a larger dynamic system, which share the same control logic for the modes of performance. Reducing the power dissipation of a single channel filter results only in a small reduction in the total power dissipation of an entire radio. However, if the modules which share the modes of performance are many, the total power reduction may eventually become significant. In order for this to be realised, other IPs within the radio, whose performance may depend on the circumstances, should be evaluated for dynamic implementation. Similar studies to the one elaborated in this thesis should be performed for these as well.

8.3.5 Link quality estimator and mode selector

In this thesis, the control signal selecting the current mode of performance is considered as a known signal. This is a simplification which needs to be addressed for the dynamic approach to be realised in practice. This requires that a link quality estimator is defined, which provides a metric that is readable in digital logic and able to tell something about the quality of the radio link. When the source for calculating the quality of the radio link is found, the mode selector in Figure 7.1 should also be implemented. Its function should be to translate the link quality estimate into mode control signals.

Chapter 9

Conclusion

The methodology of algorithm-architecture co-design is used to find the most area and energy efficient filter solution, and to optimize the hardware architecture for energy efficiency. The main contributions of this work can be summarized as follows:

- An automated area and power estimating tool-flow is created, which has proven to obtain accurate and comparable results of 500 gate designs within 5 minutes. The tool-flow is generic, and may be used for Design-for-Power purposes, and as a tool when performing similar studies in the future.

- An automated filter generation and eligibility calculation algorithm is created and implemented as a Matlab program. The filter sorting algorithm manages to predict the filter solution that yields the smallest area and least amount of power dissipation. The program is also generic and may hence be used to find the optimized solution of any filter requirement.

- A dynamic filter architecture is proposed, which alters the *filter order*, and hence the stopband attenuation, in order to adapt its performance to the requirements in real time. The best out of three implementations is found: It requires only an 8% area increase, and a 10% increase in power dissipation compared to the non-dynamic filter architecture of the same performance, while providing low power modes with 28%, 55% and 88% reduction in power dissipation. This dynamic approach is applicable for any filter requirement.

- Another dynamic filter architecture is proposed, which alters the *quantization level*, and hence the amount of quantization noise introduced by the filter, in order to adjust the filter performance in real time. The best out of four implementations is found: It requires an 18% area increase, and a 12% increase in power dissipation compared to the non-dynamic filter architecture of the same performance, while providing low power modes with 11%, 32% and 81% reduction in power dissipation. This dynamic approach is applicable for any filter requirement, and any other DSP module where the SQNR requirements may depend on the surroundings.

The dynamic channel filter is suggested to be a part of a larger dynamic system, where several modules share the same control interface. This will reduce the fraction of overhead due to additional control logic, and potentially lead to significant power reductions in the case where all the dynamic modules may operate in one of the low power modes.

9.1 Future work

The objectives for future work, discussed in the previous chapter, can be summarized as follows:

- Fine-tune the eligibility calculation in the AFGEC algorithm such that it provides an even higher percentage of accurate predictions.
- Adjust the area and power estimating tool-flow to support power gating by creating a default setup of power domains and switches according to the modes of performance.
- Implement the power gated design, *DynOrderMuxPG*, and run it through the updated tool-flow. Compare the results against the results from the theoretical analysis.
- Create a program that analyzes every possible combination of bitwidths (under certain constraints) for each of the noise sources in the design, in order to settle for an optimized solution in terms of SQNR, area and power dissipation.
- Find other IPs within the radio whose performance may depend on the circumstances, and perform similar studies to the one elaborated in this thesis.
- Find a source for calculating the quality of the radio link, and create the *link quality estimator* and the *mode selector*.

References

[1] A. Amara and P. Royannez. *Low-Power CMOS Circuits*. Taylor & Francis Group, 1st edition, 2006. ISBN 0-8493-9537-2.

[2] Bluetooth. *Specification of the Bluetooth System*, 1.1 edition, February 2001.

[3] A. P. Chandrakasan and R. W. Brodersen. Minimizing power consumption in digital CMOS circuits. *Proceedings of the IEEE*, 83(4):498–523, Apr. 1994.

[4] I. Diaz. *Algorith-Architecture Co-Design for Digital Front-Ends in Mobile Receivers*. PhD thesis, Lund University, April 2014. URL http://lup.lub.lu.se/record/4355293.

[5] Ericsson. *More than 50 billion connected devices (White Paper)*, February 2011.

[6] ETSI. *ETSI Technical Specification 136 213*, 8.8.0 edition, October 2009.

[7] W.-S. Gan and S. M. Kuo. *Embedded Signal Processing with the Micro Signal Architecture*. John Wiley & Sons, 1st edition, 2007. ISBN 9780471738411.

[8] M. Keating, D. Flynn, R. Aitken, A. Gibbons, and K. Shi. *Low Power Methodology Manual*. Springer, 2007. ISBN 9780387718194.

[9] F.-L. Luo. *Digital Front-End in Wireless Communications and Broadcasting: Circuits and Signal Processing*. Cambridge University Press, 1st edition, 2011. ISBN 978-1-107-00213-5.

[10] D. M. Pozar. *Microwave and RF Design of Wireless Systems*. John Wiley & Sons, Inc, 1st edition, 2011. ISBN 978-0-471-32282-5.

[11] J. G. Proakis and D. G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications*. Pearson Education, 4th edition, 2007. ISBN 0-13-187374-1.

[12] Synopsys. *Library Compiler Timing, Signal Integrity, and Power Modeling User Guide*, J-2014.09 edition, September 2014.

[13] Synopsys. *Using RMgen and Reference Methodology Scripts Application Note*, 2.10 edition, January 2015.

[14] J. N. Talstad. *Early Stage Power Estimation and Analysis on a Multi-Voltage Design*. Specialization project report, Norwegian University of Science and Technology, December 2014.

Appendix A

Automated area and power estimating tool-flow

A.1 Makefile for the automated tool-flow

```
SHELL = csh

runsim:
        cd sim/run/rtl; RUN_ALL --clean
        date > runsim

runsyn: runsim
        cd syn; make clean; make compile
        date > runsyn

runlay: runsyn
        cd lay; make clean; make outputs_cts
        date > runlay

runpow: runlay
        cd pow; make clean; make power_analysis
        date > runpow

allstatic:
        setenv FILE_LIST "IIRFilt46"; rm run*; make runpow;
        setenv FILE_LIST "IIRFilt21"; rm run*; make runpow;
        setenv FILE_LIST "IIRFilt29"; rm run*; make runpow;
        setenv FILE_LIST "IIRFilt30"; rm run*; make runpow;
        setenv FILE_LIST "IIRFilt120"; rm run*; make runpow;
```

```
        setenv FILE_LIST "IIRFilt296"; rm run*; make runpow;
        setenv FILE_LIST "IIRFiltQ12"; rm run*; make runpow;
        setenv FILE_LIST "IIRFiltQ13"; rm run*; make runpow;
        setenv FILE_LIST "IIRFiltQ14"; rm run*; make runpow;
        setenv FILE_LIST "IIRFiltQ15"; rm run*; make runpow;
        setenv FILE_LIST "IIRFiltQ16"; rm run*; make runpow;
        date > allstatic

alldynamic:
        setenv FILE_LIST "IIRFiltDynOrder"; rm run*; make runpow;
        setenv FILE_LIST "IIRFiltDynOrderMux"; rm run*; make runpow;
        setenv FILE_LIST "IIRFiltDynOrderReg"; rm run*; make runpow;
        setenv FILE_LIST "IIRFiltDynNoiseTrn8"; rm run*; make runpow;
        setenv FILE_LIST "IIRFiltDynNoiseRnd8"; rm run*; make runpow;
        setenv FILE_LIST "IIRFiltDynNoiseTrn12"; rm run*; make runpow;
        setenv FILE_LIST "IIRFiltDynNoiseRnd12"; rm run*; make runpow;
        date > alldynamic
```

A.2 IIRFilt testbench in SystemVerilog

```
`timescale 1ns / 1ns

module test_IIRFilt;

  parameter test_cycles = 500;
  parameter wl = 12;
  parameter SRATE = 16e6;
  parameter M_PI = 3.141592654;
  parameter TESTFREQ = 1e6;
  parameter AMP = 2000;

  logic                          arst;
  logic                          ck;
  logic [wl-1:0]         dataIn;
  logic signed [wl-1:0] dataOut,dataOutpre;
  logic [1:0]            ctrl;

  integer                i,j,k,error;
  integer                dout_handle,dout_handle1,dout_handle2;
  integer                ampmaxpre,ampminpre;
  logic                  log;

  real phaseaccu,tonefreq,tempval,att;

  logic start=0;

 //NOT DYNAMIC
   IIRFilt
    u_IIRFilt
      (
       .dataOut                 (dataOut),
```

```
        .arst                   (arst),
        .ck                     (ck),
        .dataIn                 (dataIn));

 //DYNAMIC
/*  IIRFilt
    u_IIRFilt
      (
       .dataOut                 (dataOut),
       .arst                    (arst),
       .ck                      (ck),
       .dataIn                  (dataIn),
           .ctrl                (ctrl));          */

  // Include single phase clock generator
  parameter
           CkPeriod      = 20,       // Clock period
           Setuptime     = 4,        // Input event generator toggle
               ref:ck
           StrobeDelay   = 4;        // Output event generator toggle
               ref:ck
  `include "/pro/nrf4360/lib/verilog/CK_GEN_1P.v"


   task delay_periodes;
     input [32:0] n;
     integer i;
     for (i=0; i<n; i=i+1)
       @DefineInputs;
   endtask


initial
begin
        dout_handle=$fopen("${FILE_LIST}_dump.mat"); //JOTA
        dout_handle1=$fopen("${FILE_LIST}_impresp.mat"); //JOTA

        for (i=3; i>-1; i=i-1)
        begin
                ctrl = i;
                error=0;
                arst = 0;
                dataIn=0;
                delay_periodes(1);
                arst = 1;
                delay_periodes(2);
                arst = 0;

                for (j=0; j<test_cycles; j=j+1)
                begin
                  @(posedge ck)
                          begin
```

```
                if(j==10)
                        dataIn = AMP;
                else
                        dataIn = 0;
                $fdisplay(dout_handle1,"%d",dataOut);
          end
end

arst = 0;
dataIn=0;
delay_periodes(1);
arst = 1;
delay_periodes(2);
arst = 0;

for (j=0; j<test_cycles; j=j+1)
begin
  @(posedge ck)
          begin
                dataIn = 0;
          end
end


for (k=1; k<=3; k=k+1)
begin
        arst = 0;
        dataIn=0;
        phaseaccu=0;
        tonefreq=TESTFREQ*k;
        delay_periodes(1);
        arst = 1;
        log = 0;
        delay_periodes(2);
        arst = 0;

        for (j=0; j<test_cycles; j=j+1)
          begin
                @(posedge ck)
                        begin
                          dataIn=int'($cos(phaseaccu)*
                              AMP);
                          $fdisplay(dout_handle,"%d",
                              dataOut);

                            if (j>100 && j<400)
                                    log = 1;
                            else
                                    log = 0;

                          //$fdisplay(dout_handle2,"%d
                              ",u_IIRFilt.
```

```
                                                dataOutRounded);
                                        end
                                end

                        // check the response
                        tempval=real'(2*AMP)/(real'(ampmaxpre)-real'(
                            ampminpre));
                        att=20*$log10(real'(tempval));
                        $display("Attenuation␣is␣%f␣dB␣@␣%f␣Hz",att,
                            tonefreq);
                end
        end

        $fclose(dout_handle);
        $fclose(dout_handle1);
        $stop;
end

    always_ff @(posedge ck or posedge arst)
      begin
        if(arst)
        begin
          phaseaccu=0;
        end
        else begin
          // increment phaseaccu
          if(phaseaccu > 2*M_PI)
            phaseaccu = phaseaccu - 2*M_PI + 2*M_PI*(real'(tonefreq))/
                SRATE;
          else
            phaseaccu = phaseaccu + 2*M_PI*(real'(tonefreq))/SRATE;
        end
      end

        always_ff @ (posedge ck or posedge arst)
                begin
                        if (arst)
                                begin
                                        ampmaxpre<=0;
                                        ampminpre<=0;
                                end
                        else
                                begin
                                        if(dataOut>ampmaxpre && log==1)
                                                begin
                                                        ampmaxpre<=
                                                            dataOut;
                                                end
                                        if(dataOut<ampminpre && log==1)
                                                begin
                                                        ampminpre<=
                                                            dataOut;
```

```
                                                    end
                                        end
                        end

endmodule
```

A.3 Makefile for synthesis

```
setup:
        mkdir -p logs
        mkdir -p reports
        mkdir -p results
        @date > $@

elaborate: setup
        dc_shell -f dc_scripts/dc_$@.tcl | tee logs/dc_$@.log
        @make -s summary ARG=$@
        @date > $@

compile: elaborate
        dc_shell -f dc_scripts/dc_$@.tcl | tee logs/dc_$@.log
        @make -s summary ARG=$@
        @date > $@

clean:
        rm -fr elaborate
        rm -fr compile
        rm -fr logs/*
        rm -fr reports/*
        rm -fr results/*
        rm -fr WORK/
```

A.4 Synthesis design constraints

```
create_clock -name "ck16M" -p 62.5 -w [list 0 31.25]  [get_port "ck"] \
             -comment "16␣MHz␣clock"
set_dont_touch_network "ck16M"
set_clock_transition 0.2 ck16M
set_clock_uncertainty -setup 0.20 -rise_from [get_clocks "ck16M"]
    -rise_to [get_clocks "ck16M"]
set_clock_uncertainty -setup 0.20 -fall_from [get_clocks "ck16M"]
    -fall_to [get_clocks "ck16M"]
set_clock_uncertainty -setup 0.80 -rise_from [get_clocks "ck16M"]
    -fall_to [get_clocks "ck16M"]
set_clock_uncertainty -setup 0.80 -fall_from [get_clocks "ck16M"]
    -rise_to [get_clocks "ck16M"]
set_clock_uncertainty -hold  0.10 [get_clocks "ck16M"]
set high_fanout_net_threshold 0
set_input_transition 5.0 [all_inputs]
set_load 0.05 [all_outputs]
set_max_transition 2.0 [find design "*"]
```

A.5 Makefile for layout

```
ICC_EXEC = icc_shell -64bit
LOGS_DIR     = logs_zrt
REPORTS_DIR = reports
RESULTS_DIR = results
PNA_OUTPUT_DIR = pna_output
DESIGN_LIB = ${DESIGN_NAME}_LIB

init_design_icc:
        mkdir -p $(REPORTS_DIR) $(RESULTS_DIR) $(LOGS_DIR)
        $(ICC_EXEC) $(OPTIONS) -f rm_icc_scripts/init_design_icc.tcl |
            tee -i $(LOGS_DIR)/init_design_icc.log
        date > init_design_icc

place_opt_icc: init_design_icc
        mkdir -p $(REPORTS_DIR) $(RESULTS_DIR) $(LOGS_DIR)
        $(ICC_EXEC) $(OPTIONS) -f rm_icc_scripts/place_opt_icc.tcl |tee
             -i $(LOGS_DIR)/place_opt_icc.log
        date > place_opt_icc

clock_opt_cts_icc: place_opt_icc
        mkdir -p $(REPORTS_DIR) $(RESULTS_DIR) $(LOGS_DIR)
        $(ICC_EXEC) $(OPTIONS) -f rm_icc_scripts/clock_opt_cts_icc.tcl
            |tee -i $(LOGS_DIR)/clock_opt_cts_icc.log
        date > clock_opt_cts_icc

clock_opt_psyn_icc: clock_opt_cts_icc
        mkdir -p $(REPORTS_DIR) $(RESULTS_DIR) $(LOGS_DIR)
        $(ICC_EXEC) $(OPTIONS) -f rm_icc_zrt_scripts/clock_opt_psyn_icc
            .tcl |tee -i $(LOGS_DIR)/clock_opt_psyn_icc.log
        date > clock_opt_psyn_icc

outputs_cts: clock_opt_psyn_icc
        mkdir -p $(REPORTS_DIR) $(RESULTS_DIR) $(LOGS_DIR)
        $(ICC_EXEC) $(OPTIONS) -f rm_icc_zrt_scripts/outputs_cts.tcl |
            tee -i $(LOGS_DIR)/outputs_cts.log
        date > outputs_cts

clean:
        rm  -f init_design_icc flat_dp dp init_design_icc_dp
            place_opt_icc clock_opt_cts_icc clock_opt_psyn_icc
            clock_opt_route_icc route_icc route_opt_icc chip_finish_icc
             metal_fill_icc signoff_drc_icc outputs_icc ic
        rm -rf $(DESIGN_LIB) $(LOGS_DIR) $(RESULTS_DIR)/*sbpf* $(
            RESULTS_DIR)/*.def $(RESULTS_DIR)/*pg*  $(REPORTS_DIR)/
            place* $(REPORTS_DIR)/clock* $(REPORTS_DIR)/route* $(
            REPORTS_DIR)/sign* $(REPORTS_DIR)/chip*  *_map\.* \
               net.acts *.attr .zr* Milkyway.cmd.*_*_*_* Milkyway.log.*
                   _*_*_* \.vers* port_mapping.* pna_output
        rm -f snapshot/* legalizer*/*
        rm -f command.log icc_output.txt
        rm -f outputs_cts
```

A.6 Makefile for power analysis

```
clean:
        rm -rf command.log
        rm -f power_analysis
        rm -rf reports/power_analysis_${DESIGN_NAME}_*
        rm -rf log/*

power_analysis:
        pt_shell -f power_analysis.tcl | tee -i log/power_analysis.log
        date > power_analysis
```

A.7 Power analysis script

```
set FILE_LIST [getenv "FILE_LIST"]
set DESIGN_NAME [getenv "DESIGN_NAME"]
set DESIGN_DIR      "/pri/jota/workspace/master/ip/IIRFilt"
set LAY_REPORT_DIR      "$DESIGN_DIR/lay/reports"
set SCORE_DIR           "$DESIGN_DIR/score"

sh rm -f $SCORE_DIR/${FILE_LIST}_score.csv

set READ_SDC       true
set READ_SAIF      false
set READ_VCD       true
set READ_TOGGLE    false
set REPORT_DETAILS true
set MODE_AVERAGED  true

set DESIGN_IS_READ      false
set DYNAMIC_RTL true

if {$DYNAMIC_RTL} {
        set scenarios {ctrl3 ctrl2 ctrl1 ctrl0 inactive}
} else {
        set scenarios {active inactive}
}

foreach my_power_scenario $scenarios {
     set POWER_SCENARIO  $my_power_scenario
     echo ": POWER_SCENARIO IS: $POWER_SCENARIO"


##
    ------------------------------------------------------------------------

## Design files and locations (modify to meet your flow)
```

```
##
    ------------------------------------------------------------------------


set POWER_REPORT_DIR       "$DESIGN_DIR/pow/reports/power_analysis_${
    FILE_LIST}_${POWER_SCENARIO}"
set POWER_SAIF_FILE        "$DESIGN_DIR/pow/${FILE_LIST}_${
    POWER_SCENARIO}.saif"
set POWER_ACTIVITY_FILE    "$POWER_REPORT_DIR/_power_activity.tcl"
set POWER_VCD_FILE         "/pri/jota/workspace/master/ip/$DESIGN_NAME/
    sim/run/rtl/IIRFilt.vcd"
set POWER_STRIP_PATH       "test_IIRFilt/u_IIRFilt"
set power_rail_output_file  $POWER_REPORT_DIR/
    poCalculatePower_VctFreeRptFile
set POWER_SDC_FILE         "/pri/jota/workspace/master/ip/IIRFilt/lay/
    results/$DESIGN_NAME.output.sdc"
set POWER_VERILOG_FILE     "/pri/jota/workspace/master/ip/IIRFilt/lay/
    results/$DESIGN_NAME.output.v"
set POWER_STAR_FILE        "/pri/jota/workspace/master/ip/IIRFilt/lay/
    results/$DESIGN_NAME.output.sbpf.max"

##
    ------------------------------------------------------------------------

## Parameters (dont_change)
##
    ------------------------------------------------------------------------

set power_enable_analysis true
set power_ccsp_use_zero_for_missing_leakage true
set power_enable_multi_rail_analysis true
if {$MODE_AVERAGED} {
     echo ":␣power_analysis_mode␣averaged"                1
     set power_analysis_mode averaged
 } else {
   echo ":␣power_analysis_mode␣time_based"
   set power_analysis_mode time_based
 }
set power_read_activity_ignore_case true

##
    ------------------------------------------------------------------------

## Paths and Libraries (modify to meet your flow)
##
    ------------------------------------------------------------------------

set VC_WORKSPACE [getenv VC_WORKSPACE]
set search_path { \
   /pro/lode4377/library/OPCOND \
   /cad/synopsys/ICCompiler/2013.03-SP5/libraries/syn \
   /cad/synopsys/ICCompiler/2013.03-SP5/dw/syn_ver \
```

```
   /cad/synopsys/ICCompiler/2013.03-SP5/dw/sim_ver \
   /n/library/artisan/tsmc018/sc/2004q3v1/synopsys_jota/20110526 \
   /n/library/artisan/tsmc018/sc/2004q3v1/synopsys_jota/20130220 \
   /n/library/nvlsi/tsmc/tsmc018/ICGLIB/1.3/synopsys/20131114 \
}


  set link_path {* \
     /n/library/artisan/tsmc018/sc/2004q3v1/synopsys_jota/20110526/
         ARTISAN018G_SC_1V4FF-25C_1V2.db \
     /n/library/artisan/tsmc018/sc/2004q3v1/synopsys_jota/20110526/
         ARTISAN018G_SC_1V15SS85C_1V2.db \
     /n/library/nvlsi/tsmc/tsmc018/ICGLIB/1.3/synopsys/20131114/
         ICGLIB_1V4FF-25C_1V2.db \
     /n/library/nvlsi/tsmc/tsmc018/ICGLIB/1.3/synopsys/20131114/
         ICGLIB_1V15SS85C_1V2.db \
  }


if { [file exists [which $POWER_REPORT_DIR]] }  {
     echo "Report-directory␣already␣exists:␣$POWER_REPORT_DIR"
   } else {
     echo "Create␣report-directory:␣$POWER_REPORT_DIR"
     sh mkdir $POWER_REPORT_DIR
   }

##
    ------------------------------------------------------------------------

## Read Design Data_ Verilog, UPF, Extraction (modify to meet your flow
    )
##
    ------------------------------------------------------------------------

if {!$DESIGN_IS_READ} {
        read_verilog    $POWER_VERILOG_FILE
        current_design  $DESIGN_NAME
        link
        #load_upf        $POWER_UPF_FILE
        read_parasitics $POWER_STAR_FILE
        set DESIGN_IS_READ      true
}

##
    ------------------------------------------------------------------------

## Operating Conditions TYP (modify to meet your flow)
##
    ------------------------------------------------------------------------
```

```
set_operating_conditions -analysis_type on_chip_variation LIB_1V4FF-25C
     -library ARTISAN018G_SC_1V4FF-25C_1V2


##
    ------------------------------------------------------------------------

## Activity  (modify to meet your flow)
##
    ------------------------------------------------------------------------


if {$READ_SDC} {
   echo "-------------------------------"
   echo ":Read␣sdc␣file"
   echo "-------------------------------"
   set sdc_write_unambiguous_names t

   read_sdc $POWER_SDC_FILE
   set_propagated_clock [all_clocks]
}

if {$READ_TOGGLE} {
   echo "-------------------------------"
   echo ":Read␣TOGGLE␣file"
   echo "-------------------------------"
   echo "POWER_SCENARIO␣is␣$POWER_SCENARIO"
   source /pri/jota/workspace/master/ip/IIRFilt/syn/results/${
       DESIGN_NAME}.mapped.SAIF.namemap
   set power_default_toggle_rate 0.0
   set_switching_activity -toggle_rate 0.5 -static_probability 0.2 -
       type [list registers] -hierarchy
   #set_switching_activity -toggle_rate 0.5 -static_probability 0.2 [
       get_pin u_FOS/Z01_reg_7_/RN]
   set_switching_activity -toggle_rate 0.5 -static_probability 0.0001 [
       get_port arst]
   update_power
}

if {$READ_SAIF} {
   echo "-------------------------------"
   echo ":Read␣SAIF␣file"
   echo "-------------------------------"
   #source /pri/jcs/project/ip/CORTEXM0/syn/results.21/CORTEXM0.mapped.
       SAIF.namemap
   read_saif -strip_path $POWER_STRIP_PATH $POWER_ACTIVITY_FILE
   set power_default_toggle_rate 0.00
   update_power
}

if {$READ_VCD} {
   echo "-------------------------------"
```

```
echo ":Read␣VCD␣file"
echo "------------------------------"
if {!$MODE_AVERAGED} {
    echo ":save␣waveform␣to␣be␣opened␣with␣nWave:␣$POWER_REPORT_DIR/
        powerWaves_$POWER_SCENARIO.RX"
    set_power_analysis_options -waveform_interval 620 -
        waveform_output $POWER_REPORT_DIR/powerWaves_$POWER_SCENARIO
        .RX
    echo ":Read␣VCD␣file,␣look␣at␣the␣activities␣with␣nWave"
    read_vcd  -format SystemVerilog  -strip_path $POWER_STRIP_PATH -
        time {12000 16000}  $POWER_VCD_FILE
    # Analyze your wave-data with %nwave,
    #                               read data:$POWER_REPORT_DIR/
        powerWaves_$POWER_SCENARIO.RX
    #                               read signal-list:
} else {
    echo ":Read␣VCD␣file"
    #write_activity_waveforms -vcd  $POWER_VCD_FILE -output ./power/
        $POWER_SCENARIO.vcd_waves -peak_window 6250 -interval 625 -
        hierarchical_levels 8


    #HERE
    source /pri/jota/workspace/master/ip/IIRFilt/syn/results/${
        DESIGN_NAME}.mapped.SAIF.namemap

        if {!$DYNAMIC_RTL} {
                if {$POWER_SCENARIO == "active"} {
                            read_vcd -rtl -format SystemVerilog -
                                strip_path $POWER_STRIP_PATH -time
                                {20000 26000 30000 36000 40000
                                46000}  $POWER_VCD_FILE         ;#
                                power_test*
                } elseif {$POWER_SCENARIO == "inactive"} {
                            read_vcd -rtl -format SystemVerilog -
                                strip_path $POWER_STRIP_PATH -time
                                {10000 16000}  $POWER_VCD_FILE
                                        ;# power_test*
                }
             } else {
                if {$POWER_SCENARIO == "ctrl3"} {
                            read_vcd -zero_delay -rtl -format
                                SystemVerilog -strip_path
                                $POWER_STRIP_PATH -time {20000
                                26000 30000 36000 40000 46000}
                                $POWER_VCD_FILE         ;#
                                power_test*
                } elseif {$POWER_SCENARIO == "ctrl2"} {
                            read_vcd -rtl -format SystemVerilog -
                                strip_path $POWER_STRIP_PATH -time
                                {70000 76000 80000 86000 90000
                                96000}  $POWER_VCD_FILE         ;#
```

```
                                power_test*
                    } elseif {$POWER_SCENARIO == "ctrl1"} {
                                read_vcd -rtl -format SystemVerilog -
                                    strip_path $POWER_STRIP_PATH -time
                                    {120000 126000 130000 136000 140000
                                     146000}  $POWER_VCD_FILE         ;
                                    # power_test*
                    } elseif {$POWER_SCENARIO == "ctrl0"} {
                                read_vcd -rtl -format SystemVerilog -
                                    strip_path $POWER_STRIP_PATH -time
                                    {170000 176000 180000 186000 190000
                                     196000}  $POWER_VCD_FILE         ;
                                    # power_test*
                    } elseif {$POWER_SCENARIO == "inactive"} {
                                read_vcd -rtl -format SystemVerilog -
                                    strip_path $POWER_STRIP_PATH -time
                                    {160000 166000}  $POWER_VCD_FILE
                                            ;# power_test*
                    }
            }
    }
    update_power
    set power_default_toggle_rate 0.00
    echo ":Write␣SAIF␣file"
    write_saif  $POWER_SAIF_FILE
}

set timing_save_pin_arrival_and_slack true

##
    ------------------------------------------------------------------------

## Annotate power
##
    ------------------------------------------------------------------------

update_timing
check_power
update_power

##
    ------------------------------------------------------------------------

## Standard Reports
##
    ------------------------------------------------------------------------


set power_clock_network_include_register_clock_pin_power true


if {$REPORT_DETAILS} {
```

```
foreach current_clock {ck16M \
                       } {
     echo ":Report␣power␣for:␣$current_clock"
     report_power -group clock_network -hierarchy -clocks
         $current_clock -cell_power -nosplit       > $POWER_REPORT_DIR/
         power_clk_$current_clock
     report_power -group clock_network -hierarchy -clocks
         $current_clock -cell_power -leaf -nosplit > $POWER_REPORT_DIR/
         power_clk_$current_clock.leaf
     }
}

set power_rail_output_file .poCalculatePower_VctFreeRptFile
report_switching_activity -list_not_annotated -include_only rtl -
    show_pin    > $POWER_REPORT_DIR/power_list_nonannotated_rtl.rpt
report_switching_activity -list_not_annotated     > $POWER_REPORT_DIR/
    power_list_nonannotated.rpt
report_switching_activity -list_annotated         > $POWER_REPORT_DIR/
    power_list_switching.rpt
report_switching_activity -list_annotated -include_only rtl -show_pin >
     $POWER_REPORT_DIR/power_list_switching2.rpt
report_power -verbose                             > $POWER_REPORT_DIR/
    power_summary.rpt
report_annotated_power -rails {VDD VSS} -list    > $POWER_REPORT_DIR/
    power_list_annotated.rpt
report_annotated_power -list_annotated    > $POWER_REPORT_DIR/
    power_list_annotated2.rpt

report_power -net_power  -leaf  -nosplit          > $POWER_REPORT_DIR/
    power_all_nets.rpt
report_power -cell_power -leaf  -nosplit          > $POWER_REPORT_DIR/
    power_all_cells.rpt
report_power -hierarchy         -nosplit          > $POWER_REPORT_DIR/
    power_hierarchy.rpt
report_power -cell_power -groups clock_network  -nosplit  >
    $POWER_REPORT_DIR/power_clock_network.rpt
report_power -cell_power -leaf -groups clock_network -nosplit >
    $POWER_REPORT_DIR/power_clock_network_leaf.rpt

sh echo "Name␣Internal␣Switching␣Leakage␣Total␣Percent" >> $SCORE_DIR/$
    {FILE_LIST}_score.csv
sh echo $POWER_SCENARIO >> $SCORE_DIR/${FILE_LIST}_score.csv
sh grep "IIRFilt␣" $POWER_REPORT_DIR/power_hierarchy.rpt >> $SCORE_DIR/
    ${FILE_LIST}_score.csv
}

sh echo $FILE_LIST >> $SCORE_DIR/${FILE_LIST}_score.csv
sh grep "Combinational␣Cell␣Count:" $LAY_REPORT_DIR/clock_opt_cts_icc.
    qor >> $SCORE_DIR/${FILE_LIST}_score.csv
sh grep "Sequential␣Cell␣Count:" $LAY_REPORT_DIR/clock_opt_cts_icc.qor
    >> $SCORE_DIR/${FILE_LIST}_score.csv
```

```
sh grep "CT␣Buf/Inv␣Cell␣Count:" $LAY_REPORT_DIR/clock_opt_cts_icc.qor
    >> $SCORE_DIR/${FILE_LIST}_score.csv
sh grep "ADDFXL_1V2␣" $LAY_REPORT_DIR/init_design_icc.sum >> $SCORE_DIR
    /${FILE_LIST}_score.csv
sh grep "Combinational␣Area:" $LAY_REPORT_DIR/clock_opt_cts_icc.qor >>
    $SCORE_DIR/${FILE_LIST}_score.csv
sh grep "Noncombinational␣Area:" $LAY_REPORT_DIR/clock_opt_cts_icc.qor
    >> $SCORE_DIR/${FILE_LIST}_score.csv
sh grep "Buf/Inv␣Area:" $LAY_REPORT_DIR/clock_opt_cts_icc.qor >>
    $SCORE_DIR/${FILE_LIST}_score.csv
sh grep "Cell␣Area:" $LAY_REPORT_DIR/clock_opt_cts_icc.qor >>
    $SCORE_DIR/${FILE_LIST}_score.csv


quit
```

A.8 Matlab script for visualizing score results

```
close all; clear all;

N = 6;

names = cell(1:N);

names{1} = 'IIRFilt46';
names{2} = 'IIRFilt21';
names{3} = 'IIRFilt29';
names{4} = 'IIRFilt30';
names{5} = 'IIRFilt120';
names{6} = 'IIRFilt296';

All = cell(N,1);
formatSpec = '%s';
for i=1:N
    A = textscan(fopen(['../../../score/' names{i} '_score.csv']),
        formatSpec,200);
    All{i}=A{1};
end

fclose all;

% Power: Internal Switching Leakage Total
% Cell count: Combinational Sequential CT_Buf/Inv Adders
% Cell area: Combinational Noncombinational Buf/Inv Total
activePow = zeros(N,4);
inactivePow = zeros(N,4);
cellCount = zeros(N,4);
cellArea = zeros(N,4);
for i = 1:N
    activePow(i,:) = [str2num(All{i}{9}) str2num(All{i}{10}) str2num(
        All{i}{11}) str2num(All{i}{12})];
```

```
    inactivePow(i,:) = [str2num(All{i}{22}) str2num(All{i}{23}) str2num
        (All{i}{24}) str2num(All{i}{25})];
    cellCount(i,:) = [str2num(All{i}{31}) str2num(All{i}{35}) str2num(
        All{i}{40}) str2num(All{i}{43})];
    cellArea(i,:) = [str2num(All{i}{50}) str2num(All{i}{53}) str2num(
        All{i}{56}) str2num(All{i}{59})];
end

figure
bar(activePow')
colormap summer
grid on
title('Average␣Power␣in␣Active␣Scenario')
ylabel('Average␣Power␣[W]')
set(gca,'XTickLabel',{'Internal', 'Switching', 'Leakage', 'Total'})
legend(names{1}, names{2}, names{3}, names{4}, names{5}, names{6}, '
    Location', 'northwest')

figure
bar(inactivePow')
colormap summer
grid on
title('Average␣Power␣in␣Inactive␣Scenario')
ylabel('Average␣Power␣[W]')
set(gca,'XTickLabel',{'Internal', 'Switching', 'Leakage', 'Total'})
legend(names{1}, names{2}, names{3}, names{4}, names{5}, names{6}, '
    Location', 'northwest')

figure
bar(cellCount')
colormap summer
grid on
title('Cell␣Count')
ylabel('Number␣of␣cells')
set(gca,'XTickLabel',{'Combinational', 'Sequential', 'CT␣Buf/Inv', '
    Adders'})
legend(names{1}, names{2}, names{3}, names{4}, names{5}, names{6}, '
    Location', 'northeast')

figure
bar(cellArea')
colormap summer
grid on
title('Cell␣Area')
ylabel('Area␣[\mum^2]')
set(gca,'XTickLabel',{'Combinational', 'Sequential', 'Buf/Inv', 'Total'
    })
legend(names{1}, names{2}, names{3}, names{4}, names{5}, names{6}, '
    Location', 'northwest')

impresp = cell(N,1);
for i=1:N
```

```
    impresp{i} = load(['../../run/rtl/' names{i} 'Unity_impresp.mat'],
        '-ascii');
end

figure
col = gray(N+3);
AMP = 2000;
for i=1:N
    f=(0:1:511)/1024*16e6;
    M=20*log10(abs(fft(impresp{i}(1:500)/AMP,1024)));
    plot(f,M(1:512), 'color', col(i,:))
    hold on
end

xlabel('Frequency [Hz]');
ylabel('Gain [dB]');
legend(names{1}, names{2}, names{3}, names{4}, names{5}, names{6}, '
    Location', 'northeast')
title('Frequency Response');
```

Appendix B

Automated filter generation and eligibility calculation

B.1 Matlab script for AFGEC

```
clear all; close all

%% Settings

Wn = 0.125;     %Normalized cutoff
order = 3;
sweep = 'freq';
freqStep = 0.001;
orderStep = 1;
T = 300;
numButter = 100;
numCheby = 100;
numEllip = 100;
plotTop = 6;
tolerance = 2^(-3);

%% Generate filters

A = cell(1,T);
B = cell(1,T);
Y = cell(1,T);
Yorig = cell(1,T);  %original
Yq = cell(1,T);     %quantized
```

```
Yqr = cell(1,T);    %reduced
YqFracbits = cell(1,T);
YqrFracbits = cell(1,T);
Ytype = cell(1,T);
Yfreq = cell(1,T);
Yorder = cell(1,T);
gain = zeros(1,T);
gainq = zeros(1,T);
gainqFracbits = zeros(1,T);

for i=1:T

    if i < numButter
        type = 'butter';
    elseif ((i > numButter) && (i < numButter+numCheby+1))
        type = 'cheby1';
    elseif (i > numButter+numCheby)
        type = 'ellip';
    end

    Ytype{i} = type;


    if strcmp(type,'butter')
        if strcmp(sweep,'order')
            order = i*orderStep;
            freq = Wn;
            [B{i} A{i}] = butter(order,freq);
        elseif strcmp(sweep,'freq')
            freq = Wn+freqStep*(i-1);
            [B{i} A{i}] = butter(order,freq);
        end


    elseif strcmp(type,'cheby1')
        if strcmp(sweep,'order')
            order = (i-numButter)*orderStep;
            freq = Wn;
            Rp = 1; %dB peak-to-peak passband ripple
            [B{i} A{i}] = cheby1(order,Rp,freq); %T=100
        elseif strcmp(sweep,'freq')
            Rp = 1; %dB peak-to-peak passband ripple
            freq = Wn+freqStep*(i-1-numButter);
            [B{i} A{i}] = cheby1(order,Rp,freq);
        end


    elseif strcmp(type,'ellip')
        if strcmp(sweep,'order')
            order = (i-numButter-numCheby)*orderStep;
            freq = Wn;
            Rp = 1; %dB peak-to-peak passband ripple
```

```
            Rs = 20; %dB of stopband attenuation
            [B{i} A{i}] = ellip(order,Rp,Rs,freq); %T=100
        elseif strcmp(sweep,'freq')
            Rp = 1; %dB peak-to-peak passband ripple
            Rs = 20; %dB of stopband attenuation
            freq = Wn+freqStep*(i-1-numButter-numCheby);
            [B{i} A{i}] = ellip(order,Rp,Rs,freq);
        end
    end

    Yorder{i} = order;
    Yfreq{i} = freq;


    % Convert to second-order-sections
    [Y{i} gain(i)] = tf2sos(B{i},A{i});

    % Sign invert for A coefficients
    [totSec coefPerSec] = size(Y{i});
    Yorig{i} = Y{i};
    for j=1:totSec
        Y{i}(j,5:6) = -Y{i}(j,5:6);
    end

end


%% Quantize

for i=1:T

    [totSec coefPerSec] = size(Y{i});
    for j=1:totSec
        for m=1:coefPerSec
            k = 0;
            err = 1000;
            while ((abs(err) > tolerance) || (m==6 && dec==-1))
                cur = Y{i}(j,m);
                fix = dec2fix(cur,k,k+2);
                dec = fix2dec(fix);
                err = cur-dec;
                Yq{i}(j,m) = dec;
                YqFracbits{i}(j,m) = k;
                if m==6
                    sumA = Yq{i}(j,4)-Yq{i}(j,5)-Yq{i}(j,6);
                    while sumA==0
                        YqFracbits{i}(j,5) = YqFracbits{i}(j,5) + 1;
                        YqFracbits{i}(j,6) = YqFracbits{i}(j,6) + 1;
                        Yq{i}(j,5) = fix2dec(dec2fix( Y{i}(j,5),
                            YqFracbits{i}(j,5) , YqFracbits{i}(j,5)+2 )
                            );
```

```
                        Yq{i}(j,6) = fix2dec(dec2fix( Y{i}(j,6),
                            YqFracbits{i}(j,6) , YqFracbits{i}(j,6)+2 )
                            );
                        sumA = Yq{i}(j,4)-Yq{i}(j,5)-Yq{i}(j,6);
                    end
                end
                k = k+1;
            end
        end
    end
end


%% Gather filter characteristics

totCoefs = zeros(1,T);
oneCoefs = zeros(1,T);
zeroCoefs = zeros(1,T);
remCoefs = zeros(1,T);
totBits = zeros(1,T);
singleOnes = zeros(1,T);
repeatedCoefs = zeros(1,T);
filtersSharingCoefs = zeros(1,T);

for i=1:T

        % Number of 1-coefficients, 0-coefficients, remaining
            coefficients
    totCoefs(i) = numel(Y{i});
    oneCoefs(i) = length(find(Yq{i}==1));
    zeroCoefs(i) = length(find(Yq{i}==0));
    Yqr{i} = Yq{i};
    Yqr{i}(Yq{i} == 1 | Yq{i} == 0) = [];
    YqrFracbits{i} = YqFracbits{i};
    YqrFracbits{i}(Yq{i} == 1 | Yq{i} == 0) = [];
    remCoefs(i) = numel(Yqr{i});

    % Number of bits per filter, and number of single-ones
    sum = 0;
    for j=1:numel(Yqr{i})
        k = YqrFracbits{i}(j);
        cur = Yqr{i}(j);
        bin = dec2fix(cur,k,k+2);
        bits = length(bin)-1;
        sum = sum + bits;

        numberOfOnes = length(strfind(bin,'1'));
        if numberOfOnes == 1
            singleOnes(i) = singleOnes(i) + 1;
        end
    end
    totBits(i) = sum;
```

```
end


%% Eligibility function

a = zeros(1,T);
b = zeros(1,T);
c = zeros(1,T);
e = zeros(1,T);

E = zeros(1,T);
aW = 1;
bW = 1;
cW = 1;
eW = 1;
for i=1:T
    a(i) = oneCoefs(i)/totCoefs(i);
    b(i) = totBits(i)/remCoefs(i);
    c(i) = singleOnes(i)/remCoefs(i);
    e(i) = zeroCoefs(i)/totCoefs(i);
    E(i) = aW*a(i) + bW*(1/b(i)) + cW*c(i) + eW*e(i);
end

[sortedE, sortedIndex] = sort(E, 'descend');


%% Convert best filters back and plot

figs = 0;
previous = [];
for i=1:T

    if ((isequal(Yq{sortedIndex(i)}, previous) == 0) && (figs < plotTop
        ))
        figure
        figs = figs+1;

        [totSec coefPerSec] = size(Yq{sortedIndex(i)});

        Asos = cell(1,totSec);
        Bsos = cell(1,totSec);

        col = summer(totSec+1);
        for j=1:totSec
            Bsos{j} = Yq{sortedIndex(i)}(j,1:3);
            Asos{j} = Yq{sortedIndex(i)}(j,4:6);
            Asos{j}(2:3) = -Asos{j}(2:3);

            [h w] = freqz(Bsos{j},Asos{j},1024,16000000);
            plot(w,20*log10(abs(h)),'color', col(j,:))
            hold on
```

```
        end

        Btemp = [1];
        Atemp = [1];
        for j=1:totSec
            Btemp = conv(Btemp, Bsos{j});
            Atemp = conv(Atemp, Asos{j});
        end

        Bout = Btemp; %remember gain
        Aout = Atemp;

        disp(['Filter implementation: ' num2str( sortedIndex(i) ) ', 
            Rank: ' num2str( i ) ])
        for j=1:numel(Yqr{sortedIndex(i)})
            k = YqrFracbits{sortedIndex(i)}(j);
            cur = Yqr{sortedIndex(i)}(j);
            disp(dec2fix(cur,k,k+2))
        end

        [h w] = freqz(Bout,Aout,1024,16000000);

        plot(w,20*log10(abs(h)),'color', col(totSec+1,:))
        xlabel('Frequency [Hz]')
        ylabel('Magnitude [dB]')
        title( ['Filter implementation: ' num2str( sortedIndex(i) ) ', 
            Type: ' Ytype{sortedIndex(i)} ', Order: ' num2str(Yorder{
            sortedIndex(i)}) ', Cutoff: ' num2str(Yfreq{sortedIndex(i)
            }*8) 'MHz, Rank: ' num2str( i ) ] )
        axis([0 8000000 -50 50])
        legend('FOS', 'SOS', 'FOS+SOS', 'Location', 'northeast')
        set(gca,'Color',[0.3 0.3 0.3]);
        grid on

        col2 = lines(plotTop);
        figure(100)
        hold on
        plot(w,360/(2*pi)*angle(h), 'color', col2(figs,:))
        xlabel('Frequency [Hz]')
        ylabel('Phase [degrees]')
        title( 'Phase response in passband' )
        axis([0 5500000 -200 0])
        legend('IIRFilt46', 'IIRFilt21', 'IIRFilt29', 'IIRFilt30', '
            IIRFilt120', 'IIRFilt296', 'Location', 'northeast')
        grid on

        previous = Yq{sortedIndex(i)};
    end
end


%% Plot characteristics
```

```
figure
subplot(411)
plot(zeroCoefs, '.-')
ylabel('zeroCoefs')
subplot(412)
plot(oneCoefs, '.-')
ylabel('oneCoefs')
subplot(413)
plot(singleOnes, '.-')
ylabel('singleOnes')
subplot(414)
plot(totBits, '.-')
ylabel('totBits')
xlabel('Filter␣implementation␣number')

figure
subplot(411)
plot(zeroCoefs./totCoefs, '.-')
ylabel('c_0(i)')
subplot(412)
plot(oneCoefs./totCoefs, '.-')
ylabel('c_1(i)')
subplot(413)
plot(singleOnes./remCoefs, '.-')
ylabel('c_2(i)')
subplot(414)
plot(remCoefs./totBits, '.-')
ylabel('1/c_3(i)')
xlabel('Filter␣implementation␣number')

figure
plot(E,'.-')
ylabel('Eligibility')
xlabel('Filter␣implementation␣number')
```

Appendix C

RTL implementation of winner candidates

C.1 SystemVerilog code for IIRFilt46

```
module IIRFilt #(parameter wl = 12)

(
  // Outputs
  output logic signed [wl-1:0] dataOut,
  // Inputs
  input logic arst,
  input logic ck,
  input logic signed[wl-1:0] dataIn
  );


        parameter               wlinout = 20;
        parameter               wlcoef = 4;
        parameter       wlint1 = wlinout+wlcoef;


        // Internal
        logic signed [wlcoef-1:0]               A11;
        logic signed [wlcoef-1:0]               A12;
        logic signed [wlcoef-1:0]               A22;

        logic signed [wlinout-1:0]              dataInProlong;
        logic signed [wlinout-1:0]              dataOutFOS;
```

```
        logic signed [wlinout-1:0]      dataInt;
        logic signed [wlinout-1:0]      dataOutSOS;

        logic signed [wl:0]                     dataOutRounded;

        FOS #(.wlinout(wlinout), .wlint1(wlint1), .wlcoef(wlcoef))
                u_FOS(
                        // Outputs
                        .dataOut        (dataOutFOS),
                        // Inputs
                        .arst           (arst),
                        .ck             (ck),
                        .dataIn         (dataInProlong),
                        .A11            (A11)
                        );


        SOS #(.wlinout(wlinout), .wlint1(wlint1), .wlcoef(wlcoef))
                u_SOS(
                        // Outputs
                        .dataOut        (dataOutSOS),
                        // Inputs
                        .arst           (arst),
                        .ck             (ck),
                        .dataIn         (dataInt),
                        .A12            (A12),
                        .A22            (A22)
                        );

  always_comb
    begin
                A11 = 4'b00_10;  //0.5
                A12 = 4'b01_01;  //1.25
                A22 = 4'b11_10;  //-0.5
                dataInProlong = {dataIn, {(wlinout-wl){1'b0}}};
                dataOutRounded = dataOutSOS[wlinout-1:wlinout-wl-1] +
                    {1'b1};
        end

  always_ff @(posedge ck or posedge arst)
    begin
      if(arst)
        begin
            dataOut  <= 0;
                    dataInt  <= 0;
        end
      else
        begin
                        dataInt <= dataOutFOS;
                        dataOut <= dataOutRounded[wl:1];
        end
    end
```

```
endmodule


module FOS #(parameter wl = 12, parameter wlinout = 12, parameter
    wlint1 = 16, parameter wlcoef = 4)
(
        // Outputs
        output logic signed [wlinout-1:0] dataOut,
        // Inputs
        input logic arst,
        input logic ck,
        input logic signed [wlinout-1:0] dataIn,
        input logic signed [wlcoef-1:0] A11
);

    logic signed [wlint1-1:0]          Z01;
    logic signed [wlint1-1:0]          Z01pre;
    logic signed [wlint1-1:0]          prod_z01_a11;

    logic signed [(wlinout-1)+(wlcoef-2):0]   dataInProlong;
    logic signed [wlint1:0]                dataOutpre;
    logic signed [wlinout:0]               dataOutScale;
    logic signed [wlinout+2:0]             dataOutRounded;

    multiplier1  #(.INTERNAL_REG_WIDTH(wlint1), .COEFF_WIDTH(wlcoef))
        u_mult1_0( .dataOut(prod_z01_a11),  .dataInA(Z01),  .dataInB (
            A11)); //A11

    always_comb
        begin
        dataInProlong = {dataIn, {(wlcoef-2){1'b0}}};
        Z01pre      = prod_z01_a11 + dataInProlong;
        dataOutpre    = Z01pre + Z01;
        dataOutScale = dataOutpre[wlint1:0] >> 3;
        dataOutRounded = {{2{dataOutScale[wlinout]}}, dataOutScale} +
            {1'b1};

        if ((dataOutRounded[wlinout+2] ^ dataOutRounded[wlinout]) || (
            dataOutRounded[wlinout+2] ^ dataOutRounded[wlinout+1]))
                dataOut = {dataOutRounded[wlinout+2], {wlinout-1{!
                    dataOutRounded[wlinout+2]}}};
        else
                begin
                        dataOut = dataOutRounded[wlinout:1];
                end

        end

    always_ff @ (posedge ck or posedge arst)
        begin
                if(arst)
```

```
                        begin
                                Z01 <= 0;
                        end
                else
                        begin
                                Z01 <= Z01pre;
                        end

        end
endmodule


module SOS #(parameter wl = 12, parameter wlinout = 12, parameter
    wlint1 = 16, parameter wlcoef = 4)
(
        // Outputs
        output logic signed [wlinout-1:0] dataOut,
        // Inputs
        input logic arst,
        input logic ck,
        input logic signed [wlinout-1:0] dataIn,
        input logic signed [wlcoef-1:0] A12,
        input logic signed [wlcoef-1:0] A22
);

    logic signed [wlint1:0]         Z11pre;
    logic signed [wlint1:0]         Z11;
    logic signed [wlint1+1:0]           Z11shifted;
    logic signed [wlint1:0]         Z12;
    logic signed [wlint1:0]           prod_z11_a12;
    logic signed [wlint1:0]           prod_z12_a22;
    logic signed [(wlinout-1)+(wlcoef-2):0]   dataInProlong;
    logic signed [wlint1+2:0]             dataOutpre;
    logic signed [wlinout:0]              dataOutScale;
    logic signed [wlinout+2:0]            dataOutRounded;

    multiplier1  #(.INTERNAL_REG_WIDTH(wlint1+1), .COEFF_WIDTH(wlcoef))
        u_mult2_0( .dataOut(prod_z11_a12),  .dataInA(Z11),  .dataInB (
            A12));

    multiplier1 #(.INTERNAL_REG_WIDTH(wlint1+1), .COEFF_WIDTH(wlcoef))
        u_mult2_1( .dataOut(prod_z12_a22),  .dataInA(Z12),  .dataInB (
            A22));


    always_comb
        begin
                dataInProlong = {dataIn, {(wlcoef-2){1'b0}}};
                Z11shifted      = Z11 << 1;
                Z11pre      = dataInProlong + prod_z11_a12 +
                    prod_z12_a22;
                dataOutpre      = Z11pre + Z11shifted + Z12;
```

```
                dataOutScale = dataOutpre[wlint1+2:0] >> 5; // 5 for 4-
                    bit coef, 6 for 5-bit coef
                dataOutRounded = {{2{dataOutScale[wlinout]}},
                    dataOutScale} + {1'b1};

                if ((dataOutRounded[wlinout+2] ^ dataOutRounded[wlinout
                    ]) || (dataOutRounded[wlinout+2] ^ dataOutRounded[
                    wlinout+1]))
                        dataOut = {dataOutRounded[wlinout+2], {wlinout
                            -1{!dataOutRounded[wlinout+2]}}};
                else
                        begin
                                dataOut = dataOutRounded[wlinout:1];
                        end
        end

    always_ff @(posedge ck or posedge arst)
        begin
                if(arst)
                        begin
                                Z11 <= 0;
                                Z12 <= 0;
                        end
                else
                        begin
                                Z11 <= Z11pre;
                                Z12 <= Z11;
                        end
        end
endmodule


module multiplier1 #(parameter INTERNAL_REG_WIDTH = 16, parameter
    COEFF_WIDTH = 4)
(
output logic signed [INTERNAL_REG_WIDTH-1:0] dataOut,
input logic signed [INTERNAL_REG_WIDTH-1:0] dataInA,
input logic signed [COEFF_WIDTH-1:0]  dataInB
);

   logic signed [INTERNAL_REG_WIDTH+COEFF_WIDTH-1:0] res;

   always_comb
     begin
        res = dataInA * dataInB;
                dataOut = res[((INTERNAL_REG_WIDTH-1)+(COEFF_WIDTH-2))
                    :(COEFF_WIDTH-2)];
     end

endmodule
```

Appendix D

Quantization level exploration

D.1 Matlab script for quantization noise analysis

```
Pq = zeros(1,20);
k=12;
d1 = 2^-15;
d2 = 2^-15;
d3 = 2^-11;%
d4 = 2^-16;
d5 = 2^-16;
d6 = 2^-16;
d7 = 2^-11;%
delta = [d1 d2 d3 d4 d5 d6 d7];
sigma2 = zeros(1,7);
for i=1:7
    if (i==3 || i==7)
        sigma2(i) = (delta(i)^2)/12; %Rounding
    else
        sigma2(i) = (delta(i)^2)/3; %Truncating
    end
end
Pq(k)=sum(sigma2);
disp(['Noise␣in␣new␣implementation␣is␣' num2str(sum(sigma2))])
x(k) = k;


for k=13:length(Pq)
    C = 4;
    B = k;
```

```
    int = C+B;
    d1 = 2^-(int-1);
    d2 = 2^-(int-1);
    d3 = 2^-(B-1);%
    d4 = 2^-(int);
    d5 = 2^-(int);
    d6 = 2^-(int);
    d7 = 2^-(B-1);%
    d8 = 2^-(11);
    delta = [d1 d2 d3 d4 d5 d6 d7 d8];
    sigma2 = zeros(1,length(delta));
    for i=1:length(sigma2)
        if (i==3 || i==7 || i==8)
            sigma2(i) = (delta(i)^2)/12; %Rounding
        else
            sigma2(i) = (delta(i)^2)/3; %Truncating
        end
    end
    Pq(k)=sum(sigma2);
    disp(['Noise in newer implementation is ' num2str(sum(sigma2))])
    x(k) = inout;
end

Px = 1;
SQNR = 10.*log10(Px./Pq);

figure
plot(x(12:20), Pq(12:20))
grid on
title('Total quantization noise')
ylabel('\sigma_e^2')
xlabel('Inter-module bitwidth, B')

figure
plot(x(12:20), SQNR(12:20))
grid on
title('Signal-to-quantization-noise ratio')
ylabel('SQNR [dB]')
xlabel('Inter-module bitwidth, B')
```

Appendix E

Dynamic RTL

E.1 SystemVerilog code for DynOrderMux

```
module IIRFilt #(parameter wl = 12)

(
  // Outputs
  output logic signed [wl-1:0] dataOut,
  // Inputs
  input logic arst,
  input logic ck,
  input logic signed[wl-1:0] dataIn,
  input logic [1:0] ctrl
  );


        parameter               wlinout = 15;
        parameter               wlcoef = 4;
        parameter       wlint1 = wlinout+wlcoef;

        // Internal
        logic ctrlFOS, ctrlSOS;
        logic signed [wlcoef-1:0]               A11;
        logic signed [wlcoef-1:0]               A12;
        logic signed [wlcoef-1:0]               A22;
        logic signed [wlinout-1:0]              dataInProlong;
        logic signed [wlinout-1:0]              dataOutFOS;
        logic signed [wlinout-1:0]      dataInt;
        logic signed [wlinout-1:0]      dataOutSOS;
```

```
        logic signed [wl:0]                     dataOutRounded;

        FOS #(.wlinout(wlinout), .wlint1(wlint1), .wlcoef(wlcoef))
                u_FOS(
                        // Outputs
                        .dataOut        (dataOutFOS),
                        // Inputs
                        .arst           (arst),
                        .ck             (ck),
                        .dataIn         (dataInProlong),
                        .A11            (A11),
                        .ctrl           (ctrlFOS)
                        );


        SOS #(.wlinout(wlinout), .wlint1(wlint1), .wlcoef(wlcoef))
                u_SOS(
                        // Outputs
                        .dataOut        (dataOutSOS),
                        // Inputs
                        .arst           (arst),
                        .ck             (ck),
                        .dataIn         (dataInt),
                        .A12            (A12),
                        .A22            (A22),
                        .ctrl   (ctrlSOS)
                        );

  always_comb
    begin
                ctrlFOS = ctrl[0];
                ctrlSOS = ctrl[1];
                A11 = 4'b00_10;  //0.5
                A12 = 4'b01_01;  //1.25
                A22 = 4'b11_10;  //-0.5
                dataInProlong = {dataIn, {(wlinout-wl){1'b0}}};
                dataInt = dataOutFOS;
                dataOutRounded = dataOutSOS[wlinout-1:wlinout-wl-1] +
                    {1'b1};
                dataOut = dataOutRounded[wl:1];
        end

endmodule

module FOS #(parameter wl = 12, parameter wlinout = 12, parameter
    wlint1 = 16, parameter wlcoef = 4)
(
        // Outputs
        output logic signed [wlinout-1:0] dataOut,
        // Inputs
        input logic arst,
        input logic ck,
```

```
        input logic signed [wlinout-1:0] dataIn,
        input logic signed [wlcoef-1:0] A11,
        input logic ctrl
);


logic signed [wlint1-1:0]         Z01;
logic signed [wlint1-1:0]         Z01pre;
logic signed [wlint1-1:0]         prod_z01_a11;
logic signed [(wlinout-1)+(wlcoef-2):0]   dataInProlong;
logic signed [wlint1:0]           dataOutpre;
logic signed [wlinout:0]          dataOutScale;
logic signed [wlinout+2:0]                dataOutRounded;
logic signed [wlinout-1:0]                      dataOutreg;
logic                    arstctrl, ck_g, obs, tm;

multiplier1  #(.INTERNAL_REG_WIDTH(wlint1), .COEFF_WIDTH(wlcoef))
        u_mult1_0( .dataOut(prod_z01_a11),  .dataInA(Z01),  .dataInB (
            A11)); //A11


`ifdef HINST_RTL
  always_latch
        if (!ck)
          obs = ctrl;

        assign ck_g = ck && (obs || tm);
`elsif HINST_TSMC_180_ARM_1V2
  TLATNCOAX2_1V2 u_SizeOnlyHinstTlatncoa(.ECK(ck_g), .CK(ck), .E(ctrl),
       .OBS(obs), .TM(tm));
`endif

always_comb
        begin
                tm = 0;

                if (ctrl == 1)
                        dataInProlong = {dataIn, {(wlcoef-2){1'b0}}};
                else
                        dataInProlong = '0;

                Z01pre      = prod_z01_a11 + dataInProlong;
                dataOutpre    = Z01pre + Z01;
                dataOutScale = dataOutpre[wlint1:0] >> 3;
                dataOutRounded = {{2{dataOutScale[wlinout]}},
                    dataOutScale} + {1'b1};

                arstctrl = (arst || !ctrl);

                if (ctrl==1)
                        dataOut = dataOutreg;
                else
```

```
                        dataOut = dataIn;

        end

always_ff @ (posedge ck_g or posedge arstctrl)
        begin
                if(arstctrl)
                        begin
                                Z01 <= 0;
                                dataOutreg <= 0;
                        end
                else
                        begin
                                Z01 <= Z01pre;

                                if ((dataOutRounded[wlinout+2] ^
                                    dataOutRounded[wlinout]) || (
                                    dataOutRounded[wlinout+2] ^
                                    dataOutRounded[wlinout+1]))
                                        dataOutreg <= {dataOutRounded[
                                            wlinout+2], {wlinout-1{!
                                            dataOutRounded[wlinout
                                            +2]}}};
                                else
                                        dataOutreg <= dataOutRounded[
                                            wlinout:1];
                        end

        end

endmodule

module SOS #(parameter wl = 12, parameter wlinout = 12, parameter
    wlint1 = 16, parameter wlcoef = 4)
(
        // Outputs
        output logic signed [wlinout-1:0] dataOut,
        // Inputs
        input logic arst,
        input logic ck,
        input logic signed [wlinout-1:0] dataIn,
        input logic signed [wlcoef-1:0] A12,
        input logic signed [wlcoef-1:0] A22,
        input logic ctrl
);

logic signed [wlint1:0]         Z11pre;
logic signed [wlint1:0]         Z11;
logic signed [wlint1+1:0]               Z11shifted;
logic signed [wlint1:0]         Z12;
logic signed [wlint1:0]           prod_z11_a12;
logic signed [wlint1:0]           prod_z12_a22;
```

```
logic signed [(wlinout-1)+(wlcoef-2):0]   dataInProlong;
logic signed [wlint1+2:0]         dataOutpre;
logic signed [wlinout:0]          dataOutScale;
logic signed [wlinout+2:0]                dataOutRounded;
logic signed [wlinout-1:0]                      dataOutreg;
logic           arstctrl, ck_g, obs, tm;

multiplier1  #(.INTERNAL_REG_WIDTH(wlint1+1), .COEFF_WIDTH(wlcoef))
        u_mult2_0( .dataOut(prod_z11_a12),  .dataInA(Z11),  .dataInB (
            A12));

multiplier1 #(.INTERNAL_REG_WIDTH(wlint1+1), .COEFF_WIDTH(wlcoef))
        u_mult2_1( .dataOut(prod_z12_a22),  .dataInA(Z12),  .dataInB (
            A22));


`ifdef HINST_RTL
  always_latch
        if (!ck)
          obs = ctrl;

        assign ck_g = ck && (obs || tm);
`elsif HINST_TSMC_180_ARM_1V2
  TLATNCOAX2_1V2 u_SizeOnlyHinstTlatncoa(.ECK(ck_g), .CK(ck), .E(ctrl),
       .OBS(obs), .TM(tm));
`endif

always_comb
        begin
                tm = 0;

                if (ctrl == 1)
                        dataInProlong = {dataIn, {(wlcoef-2){1'b0}}};
                else
                        dataInProlong = '0;

                Z11shifted      = Z11 << 1;
                Z11pre      = dataInProlong + prod_z11_a12 +
                    prod_z12_a22;
                dataOutpre      = Z11pre + Z11shifted + Z12;
                dataOutScale = dataOutpre[wlint1+2:0] >> 5;
                dataOutRounded = {{2{dataOutScale[wlinout]}},
                    dataOutScale} + {1'b1};
                arstctrl = (arst || !ctrl);

                if (ctrl == 1)
                        dataOut = dataOutreg;
                else
                        dataOut = dataIn;
        end
```

```
always_ff @(posedge ck_g or posedge arstctrl)
        begin
                if(arstctrl)
                        begin
                                Z11 <= 0;
                                Z12 <= 0;
                                dataOutreg <= 0;
                        end
                else
                        begin
                                Z11 <= Z11pre;
                                Z12 <= Z11;

                                if ((dataOutRounded[wlinout+2] ^
                                    dataOutRounded[wlinout]) || (
                                    dataOutRounded[wlinout+2] ^
                                    dataOutRounded[wlinout+1]))
                                        dataOutreg <= {dataOutRounded[
                                            wlinout+2], {wlinout-1{!
                                            dataOutRounded[wlinout
                                            +2]}}};
                                else
                                        dataOutreg <= dataOutRounded[
                                            wlinout:1];
                        end
        end

endmodule

module multiplier1 #(parameter INTERNAL_REG_WIDTH = 16, parameter
    COEFF_WIDTH = 4)
(
output logic signed [INTERNAL_REG_WIDTH-1:0] dataOut,
input logic signed [INTERNAL_REG_WIDTH-1:0] dataInA,
input logic signed [COEFF_WIDTH-1:0]  dataInB
);

   logic signed [INTERNAL_REG_WIDTH+COEFF_WIDTH-1:0] res;

   always_comb
     begin
        res = dataInA * dataInB;
                dataOut = res[((INTERNAL_REG_WIDTH-1)+(COEFF_WIDTH-2))
                    :(COEFF_WIDTH-2)];
     end

endmodule
```

E.2 SystemVerilog code for DynNoiseRnd12

```
module IIRFilt #(parameter wl = 12)

(
  // Outputs
  output logic signed [wl-1:0] dataOut,
  // Inputs
  input logic arst,
  input logic ck,
  input logic signed[wl-1:0] dataIn,
  input logic [1:0] ctrl
  );


        parameter               wlinout = 15;
        parameter               wlcoef = 4;
        parameter       wlint1 = wlinout+wlcoef;

        parameter               Llimit = 12;
        parameter               Rlimit = 6;


        // Internal
        logic gateLSBits, gateMiddle, gateMSBits;
        logic obs0, obs1, obs2;
        logic [2:0] ck_g, arstctrl;
        logic signed [wlcoef-1:0]               A11;
        logic signed [wlcoef-1:0]               A12;
        logic signed [wlcoef-1:0]               A22;
        logic signed [wlinout-1:0]              dataInProlong;
        logic signed [wlinout-1:0]              dataOutFOS;
        logic signed [wlinout-1:0]      dataInt;
        logic signed [wlinout-1:0]      dataOutSOS;
        logic signed [wl:0]                     dataOutRounded;

        `ifdef HINST_RTL
          always_latch
                if (!ck)
                  obs0 = !gateLSBits;

                assign ck_g[0] = ck && obs0;
        `elsif HINST_TSMC_180_ARM_1V2
          TLATNCOAX2_1V2 u_SizeOnlyHinstTlatncoa0(.ECK(ck_g[0]), .CK(ck
              ), .E(ctrl0), .OBS(obs0), .TM(1'b0));
        `endif

        `ifdef HINST_RTL
          always_latch
                if (!ck)
                  obs1 = !gateMiddle;

                assign ck_g[1] = ck && obs1;
```

```
        `elsif HINST_TSMC_180_ARM_1V2
          TLATNCOAX2_1V2 u_SizeOnlyHinstTlatncoa1(.ECK(ck_g[1]), .CK(ck
              ), .E(ctrl1), .OBS(obs1), .TM(1'b0));
        `endif

        `ifdef HINST_RTL
          always_latch
                if (!ck)
                  obs2 = !gateMSBits;

                assign ck_g[2] = ck && obs2;
        `elsif HINST_TSMC_180_ARM_1V2
          TLATNCOAX2_1V2 u_SizeOnlyHinstTlatncoa2(.ECK(ck_g[2]), .CK(ck
              ), .E(ctrl2), .OBS(obs2), .TM(1'b0));
        `endif


        FOS #(.wlinout(wlinout), .wlint1(wlint1), .wlcoef(wlcoef), .
            Llimit(Llimit), .Rlimit(Rlimit))
                u_FOS(
                        // Outputs
                        .dataOut        (dataOutFOS),
                        // Inputs
                        .arst           (arst),
                        .ck             (ck),
                        .dataIn         (dataInProlong),
                        .A11            (A11),
                        .arstctrl       (arstctrl),
                        .ck_g           (ck_g),
                        .ctrl           (ctrl)
                        );


        SOS #(.wlinout(wlinout), .wlint1(wlint1), .wlcoef(wlcoef), .
            Llimit(Llimit), .Rlimit(Rlimit))
                u_SOS(
                        // Outputs
                        .dataOut        (dataOutSOS),
                        // Inputs
                        .arst           (arst),
                        .ck             (ck),
                        .dataIn         (dataInt),
                        .A12            (A12),
                        .A22            (A22),
                        .arstctrl       (arstctrl),
                        .ck_g           (ck_g),
                        .ctrl           (ctrl)
                        );

  always_comb
    begin
```

```
                gateLSBits = !ctrl[0] || !ctrl[1];
                gateMiddle = !ctrl[1];
                gateMSBits = !ctrl[0] && !ctrl[1];

                arstctrl[0] = arst || gateLSBits;
                arstctrl[1] = arst || gateMiddle;
                arstctrl[2] = arst || gateMSBits;

                A11 = 4'b00_10;  //0.5
                A12 = 4'b01_01;  //1.25
                A22 = 4'b11_10;  //-0.5
                dataInProlong = {dataIn, {(wlinout-wl){1'b0}}};
                dataInt = dataOutFOS;
                dataOutRounded = dataOutSOS[wlinout-1:wlinout-wl-1] +
                    {1'b1};

                if (ctrl == 0)
                        dataOut = dataIn;
                else
                        dataOut = dataOutRounded[wl:1];
        end

endmodule

module FOS #(parameter wl = 12, parameter wlinout = 12, parameter
    wlint1 = 16, parameter wlcoef = 4, parameter Llimit = 16, parameter
     Rlimit = 8)
(
        // Outputs
        output logic signed [wlinout-1:0] dataOut,
        // Inputs
        input logic arst,
        input logic ck,
        input logic signed [wlinout-1:0] dataIn,
        input logic signed [wlcoef-1:0] A11,
        input logic [2:0] arstctrl,
        input logic [2:0] ck_g,
        input logic [1:0] ctrl
);

logic signed [wlint1-1:0]         Z01;
logic signed [wlint1-1:0]         Z01pre, Z01preRound;
logic signed [wlint1-1:0]         prod_z01_a11;
logic signed [(wlinout-1)+(wlcoef-2):0]   dataInProlong;
logic signed [wlint1:0]           dataOutpre;
logic signed [wlinout:0]          dataOutScale;
logic signed [wlinout+2:0]                dataOutRounded;
logic signed [wlinout-1:0]                      dataOutreg;

multiplier1  #(.INTERNAL_REG_WIDTH(wlint1), .COEFF_WIDTH(wlcoef))
        u_mult1_0( .dataOut(prod_z01_a11),  .dataInA(Z01),  .dataInB (
            A11)); //A11
```

```
always_comb
        begin
                dataInProlong = {dataIn, {(wlcoef-2){1'b0}}};
                Z01pre      = prod_z01_a11 + dataInProlong;

                if (ctrl==3)
                        begin
                        Z01preRound = Z01pre;
                        end
                else if (ctrl==2)
                        begin
                        //Z01preRound   = Z01pre + {{1'b1}, {(Rlimit-1)
                            {0'b0}}};
                        Z01preRound     = Z01pre + {{1'b1}, {5'b0}};
                        end
                else if (ctrl==1)
                        begin
                        //Z01preRound   = Z01pre + {{1'b1}, {(Llimit-1)
                            {0'b0}}};
                        Z01preRound     = Z01pre + {{1'b1}, {11'b0}};
                        end
                else if (ctrl==0)
                        begin
                        Z01preRound     = Z01pre;
                        end

                dataOutpre    = Z01pre + Z01;
                dataOutScale = dataOutpre[wlint1:0] >> 3;
                dataOutRounded = {{2{dataOutScale[wlinout]}},
                    dataOutScale} + {1'b1};

                                if ((dataOutRounded[wlinout+2] ^
                                    dataOutRounded[wlinout]) || (
                                    dataOutRounded[wlinout+2] ^
                                    dataOutRounded[wlinout+1]))
                                        dataOutreg <= {dataOutRounded[
                                            wlinout+2], {wlinout-1{!
                                            dataOutRounded[wlinout
                                            +2]}}};
                                else
                                        dataOutreg <= dataOutRounded[
                                            wlinout:1];
        end

always_ff @ (posedge ck_g[2] or posedge arstctrl[2])
        begin
                if (arstctrl[2])
                        begin
                        Z01[wlint1-1:Llimit] <= 0;
                        dataOut[wlinout-1:0] <= 0;
                        end
```

```
                else
                        begin
                        Z01[wlint1-1:Llimit] <= Z01preRound[wlint1-1:
                            Llimit];
                        dataOut[wlinout-1:0] <= dataOutreg[wlinout
                            -1:0];
                        end
        end

always_ff @ (posedge ck_g[1] or posedge arstctrl[1])
        begin
                if (arstctrl[1])
                        begin
                        Z01[Llimit-1:Rlimit] <= 0;
                        end
                else
                        begin
                        Z01[Llimit-1:Rlimit] <= Z01preRound[Llimit-1:
                            Rlimit];
                        end
        end

always_ff @ (posedge ck_g[0] or posedge arstctrl[0])
        begin
                if (arstctrl[0])
                        begin
                        Z01[Rlimit-1:0] <= 0;
                        end
                else
                        begin
                        Z01[Rlimit-1:0] <= Z01preRound[Rlimit-1:0];
                        end
        end

endmodule

module SOS #(parameter wl = 12, parameter wlinout = 12, parameter
    wlint1 = 16, parameter wlcoef = 4, parameter Llimit = 16, parameter
     Rlimit = 8)
(
        // Outputs
        output logic signed [wlinout-1:0] dataOut,
        // Inputs
        input logic arst,
        input logic ck,
        input logic signed [wlinout-1:0] dataIn,
        input logic signed [wlcoef-1:0] A12,
        input logic signed [wlcoef-1:0] A22,
        input logic [2:0] arstctrl,
        input logic [2:0] ck_g,
        input logic [1:0] ctrl
);
```

```
logic signed [wlint1:0]          Z11pre, Z11preRound;
logic signed [wlint1:0]          Z11;
logic signed [wlint1+1:0]                Z11shifted;
logic signed [wlint1:0]          Z12;
logic signed [wlint1:0]            prod_z11_a12;
logic signed [wlint1:0]            prod_z12_a22;
logic signed [(wlinout-1)+(wlcoef-2):0]   dataInProlong;
logic signed [wlint1+2:0]          dataOutpre;
logic signed [wlinout:0]           dataOutScale;
logic signed [wlinout+2:0]                dataOutRounded;
logic signed [wlinout-1:0]                      dataOutreg;


multiplier1  #(.INTERNAL_REG_WIDTH(wlint1+1), .COEFF_WIDTH(wlcoef))
        u_mult2_0( .dataOut(prod_z11_a12),  .dataInA(Z11),  .dataInB (
            A12));

multiplier1 #(.INTERNAL_REG_WIDTH(wlint1+1), .COEFF_WIDTH(wlcoef))
        u_mult2_1( .dataOut(prod_z12_a22),  .dataInA(Z12),  .dataInB (
            A22));

always_comb
        begin
                dataInProlong = {dataIn, {(wlcoef-2){1'b0}}};
                Z11shifted      = Z11 << 1;
                Z11pre      = dataInProlong + prod_z11_a12 +
                    prod_z12_a22;

                if (ctrl==3)
                        begin
                        Z11preRound = Z11pre;
                        end
                else if (ctrl==2)
                        begin
                        //Z11preRound   = Z11pre + {{1'b1}, {(Rlimit-1)
                            {0'b0}}};
                        Z11preRound     = Z11pre + {{1'b1}, {5'b0}};
                        end
                else if (ctrl==1)
                        begin
                        //Z11preRound   = Z11pre + {{1'b1}, {(Llimit-1)
                            {0'b0}}};
                        Z11preRound     = Z11pre + {{1'b1}, {11'b0}};
                        end
                else if (ctrl==0)
                        begin
                        Z11preRound     = Z11pre;
                        end

                dataOutpre      = Z11pre + Z11shifted + Z12;
                dataOutScale = dataOutpre[wlint1+2:0] >> 5;
```

```
                dataOutRounded = {{2{dataOutScale[wlinout]}},
                    dataOutScale} + {1'b1};

                if ((dataOutRounded[wlinout+2] ^ dataOutRounded[wlinout
                    ]) || (dataOutRounded[wlinout+2] ^ dataOutRounded[
                    wlinout+1]))
                        dataOutreg <= {dataOutRounded[wlinout+2], {
                            wlinout-1{!dataOutRounded[wlinout+2]}}};
                else
                        dataOutreg <= dataOutRounded[wlinout:1];

        end

always_ff @ (posedge ck_g[2] or posedge arstctrl[2])
        begin
                if (arstctrl[2])
                        begin
                        Z11[wlint1:Llimit] <= 0;
                        Z12[wlint1:Llimit] <= 0;
                        dataOut[wlinout-1:0] <= 0;
                        end
                else
                        begin
                        Z11[wlint1:Llimit] <= Z11preRound[wlint1:Llimit
                            ];
                        Z12[wlint1:Llimit] <= Z11[wlint1:Llimit];
                        dataOut[wlinout-1:0] <= dataOutreg[wlinout
                            -1:0];
                        end
        end

always_ff @ (posedge ck_g[1] or posedge arstctrl[1])
        begin
                if (arstctrl[1])
                        begin
                        Z11[Llimit-1:Rlimit] <= 0;
                        Z12[Llimit-1:Rlimit] <= 0;
                        end
                else
                        begin
                        Z11[Llimit-1:Rlimit] <= Z11preRound[Llimit-1:
                            Rlimit];
                        Z12[Llimit-1:Rlimit] <= Z11[Llimit-1:Rlimit];
                        end
        end

always_ff @ (posedge ck_g[0] or posedge arstctrl[0])
        begin
                if (arstctrl[0])
                        begin
                        Z11[Rlimit-1:0] <= 0;
                        Z12[Rlimit-1:0] <= 0;
```

```
                        end
                else
                        begin
                        Z11[Rlimit-1:0] <= Z11preRound[Rlimit-1:0];
                        Z12[Rlimit-1:0] <= Z11[Rlimit-1:0];
                        end
        end

endmodule

module multiplier1 #(parameter INTERNAL_REG_WIDTH = 16, parameter
    COEFF_WIDTH = 4)
(
output logic signed [INTERNAL_REG_WIDTH-1:0] dataOut,
input logic signed [INTERNAL_REG_WIDTH-1:0] dataInA,
input logic signed [COEFF_WIDTH-1:0]  dataInB
);

   logic signed [INTERNAL_REG_WIDTH+COEFF_WIDTH-1:0] res;

   always_comb
     begin
        res = dataInA * dataInB;
                dataOut = res[((INTERNAL_REG_WIDTH-1)+(COEFF_WIDTH-2))
                    :(COEFF_WIDTH-2)];
     end

endmodule
```

E.3 Clock and reset distribution of dynamic RTL implementations

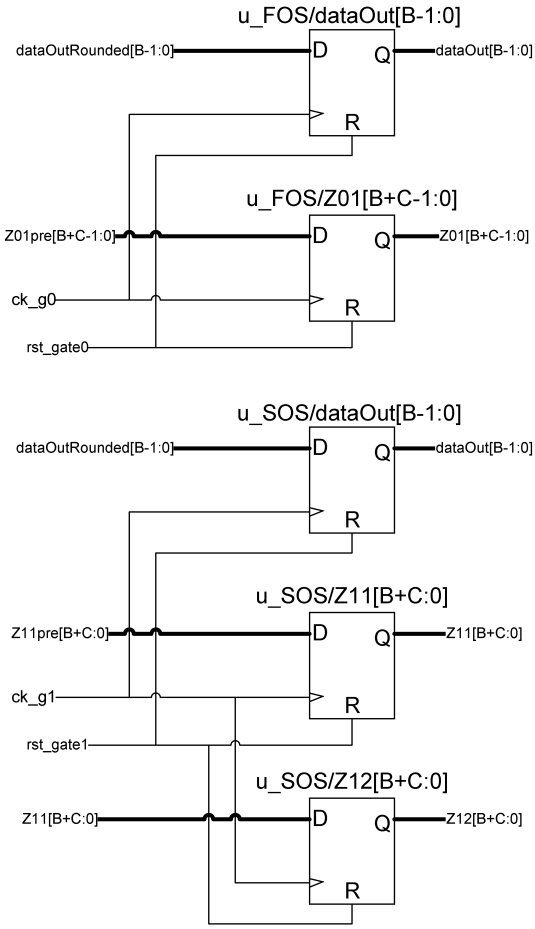


Figure E.1: Registers and clock signals in the *DynOrder* and *DynOrderMux* implementations

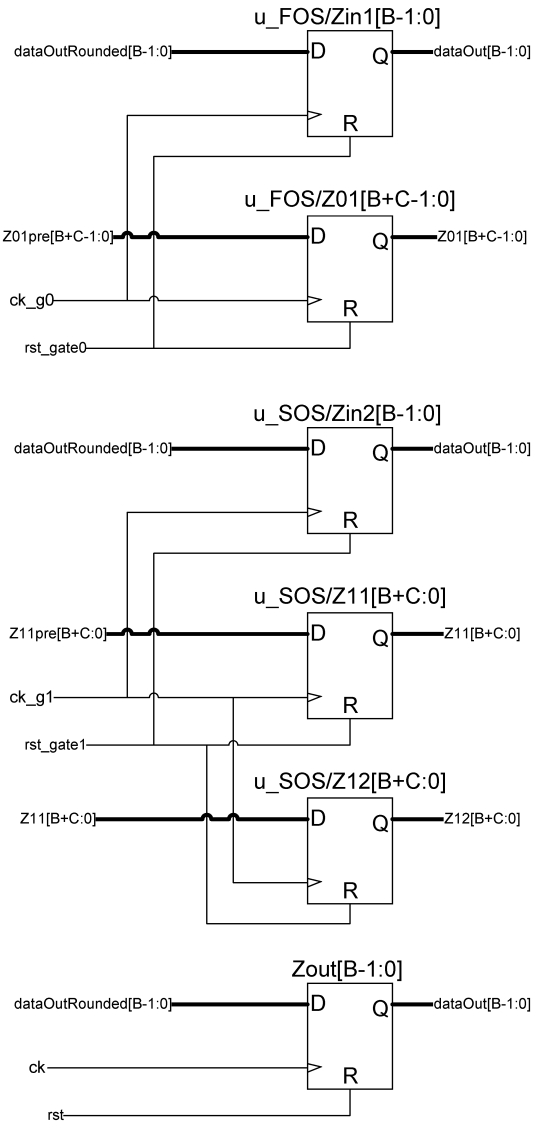


Figure E.2: Registers and clock signals in the *DynOrderReg* implementation

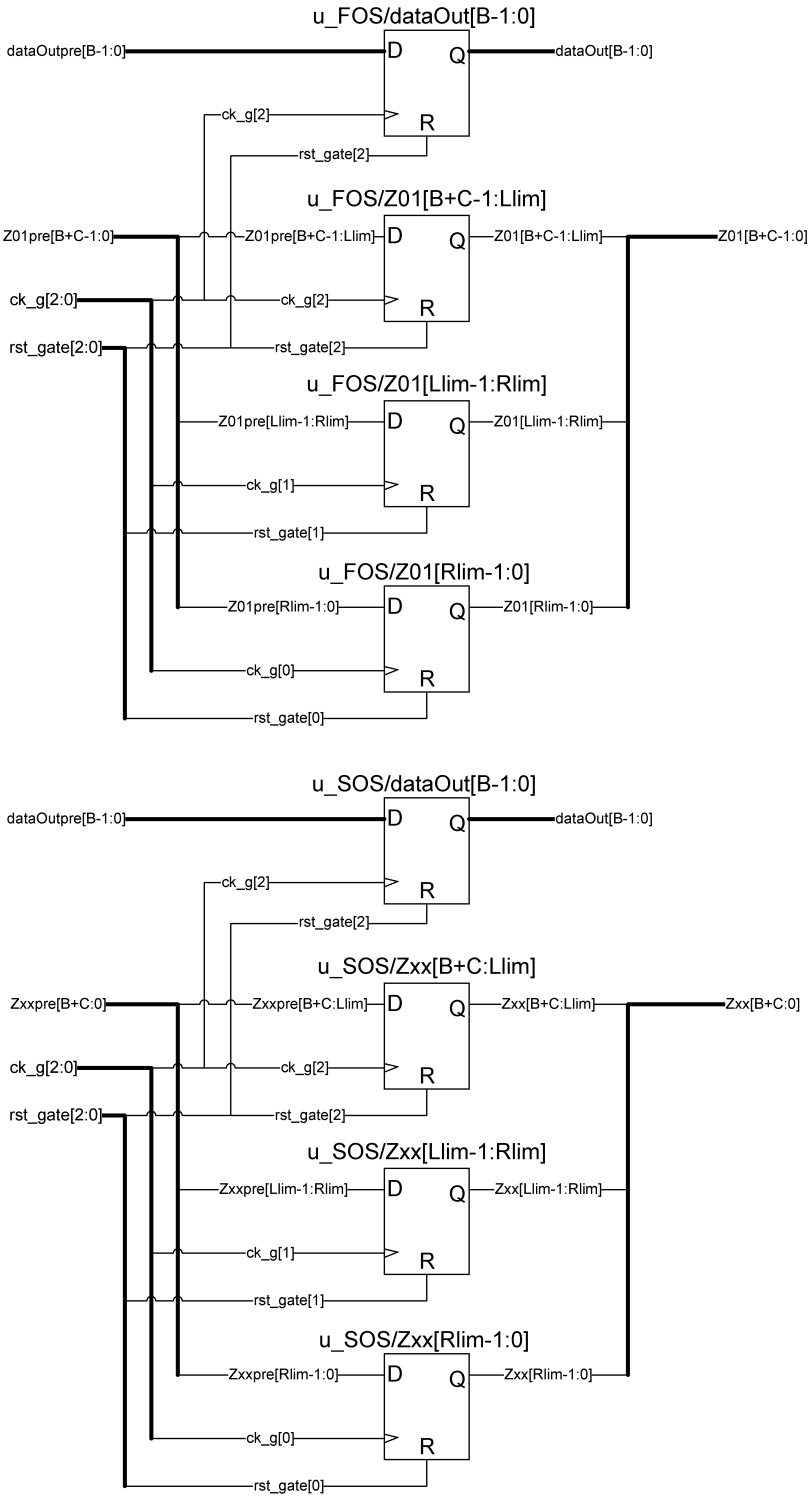


Figure E.3: Registers and clock signals in the dynamic quantization noise approach