UNIVERSITY OF OSLO
Department of Informatics

Suppression of $\Delta\Sigma$ DAC Quantisation Noise by Bandwidth Adaptation

Thesis in partial
fulfillment of the
requirements for the
degree of Cand. Scient.

Jørgen Andreas
Michaelsen

May 15, 2006

contents

1	Introduction	9
1.1	Related work	10
1.2	Thesis outline	11
2	Background	13
2.1	Lowpass oversampling $\Delta\Sigma$ digital to analogue conversion .	13
2.1.1	Interpolation filter	14
2.1.2	$\Delta\Sigma$ modulator	15
2.1.3	D/A conversion	19
2.1.4	Reconstruction filter	19
2.1.5	Performance	19
2.2	Tunable filter	22
2.2.1	Switched-capacitor basics	22
2.2.2	Switched-capacitor filter structures	24
2.2.3	Tunable switched-capacitor filters	25
2.3	Signal analysis	27
2.3.1	The short time Fourier transform	27
2.3.2	Wavelets	28
2.3.3	Subband coded signals	30
3	Theory	33
3.1	Implementation	33
3.1.1	Signal analysis	33
3.1.2	Tunable reconstruction filter	35
3.1.3	Assembling the system	37
3.2	Performance gain	37
4	Simulation	41
4.1	Simulation model implementation	41
4.2	Experimental Setup	42
4.3	Results obtained from a synthetic input signal	43
4.4	Results obtained from audio data input	47
4.5	Summary of measured results	48

5	Conclusion	53
5.1	Conclusion	53
5.2	Future work	54
A	Source code	55
A.1	Octave scripts	55
A.2	Simulator	57
A.2.1	Top-level	57
A.2.2	MPEG audio parser	61
A.2.3	$\Delta\Sigma$ modulator	67
A.2.4	Reconstruction filter	68

list of figures

1.1	Basic outline of the proposed noise reduction scheme	10
2.1	High-level overview of a $\Delta\Sigma$ DAC	13
2.2	Interpolation filter	14
2.3	Cascaded interpolation filter ($N = 4$)	15
2.4	Polyphase interpolation filter	16
2.5	$\Delta\Sigma$ modulator	16
2.6	Plot of noise shaping gain as a function of angular frequency	18
2.7	A basic switched-capacitor resistor approximation	23
2.8	The charge programming circuit from [1]	26
2.9	The Daubechies wavelet with three vanishing moments . . .	29
2.10	Wavelet analysis implemented by a filter bank	29
2.11	Subband encoder and decoder system	31
2.12	MPEG-1 audio decoding overview	32
3.1	Input signal analysis with decoupling to reconstruction filter	34
3.2	Wrapping a static bandwidth $\Delta\Sigma$ DAC	35
3.3	Full integration with the $\Delta\Sigma$ DAC	36
3.4	Relative SNR improvement as a function of bandwidth . . .	38
4.1	A time domain comparison of the 1 kHz output signal . . .	44
4.2	A frequency domain presentation of the first time window .	45
4.3	A time domain comparison of the 2 kHz output signal . . .	47
4.4	Interpolated output from the MPEG audio decoder	49
4.5	Unfiltered output from the $\Delta\Sigma$ modulator	49
4.6	Output reconstructed with a static cutoff frequency filter . .	50
4.7	Output reconstructed with a dynamic cutoff frequency filter	50
4.8	Interpolated output from the MPEG audio decoder	51
4.9	Unfiltered output from the $\Delta\Sigma$ modulator	51
4.10	Output reconstructed with a static cutoff frequency filter . .	52
4.11	Output reconstructed with a dynamic cutoff frequency filter	52
A.1	Frequency response of two reconstruction filters	72

This page is intentionally left blank

preface

PLEASE FIND ENCLOSED, a thesis in partial fulfillment of the requirements for the degree of Cand. Scient., respectfully submitted for your perusal.

I would like to take this opportunity to thank my supervisor, Dag T. Wisland, for his skillful guidance during the execution of this work. I would also like to thank my parents, Karin and Oddbjørn, for their kind support throughout, and Maja for encouraging me to finish. Thank you all for your patience. Thanks are also due to Viktor S. Wold Eide, Frank Eliassen, and Hans Ole Rafaelsen for giving me the opportunity to work on interesting and challenging projects at Simula Research Laboratory. You inspired me to finish this thesis. Erik Arne Øen took the time to read a draft version of this report. Thank you for your valuable feedback.

Whilst writing this report, I made extensive use of the following tools: GNU Emacs provided a convenient and efficient means of inputting the text and figures, subsequently typeset by pdf $\text{\LaTeX}$ with PGF, microtype, and a host of other packages extending its functionality. GNU Octave, a MATLAB®-like numerical programming language, was used for numerical calculations and for computing the data for some of the figures.

BØN, MAY 15, 2006

JØRGEN ANDREAS MICHAELSEN

This page is intentionally left blank

IN MODERN ELECTRONIC CIRCUITS, a lot of signal processing takes place in the digital domain. Digital circuits are preferable in many applications due to their robustness and accuracy. Digital systems generally require less effort to implement and are not susceptible to the drift or tuning problems commonly found in analogue circuitry. As the real world is inherently analogue, signal conversion is needed when interacting between these domains. By using high speed digital signal processing, the analogue boundary can be pushed further and the analogue domain signal processing made simpler and more robust. As state of the art in transducers, actuators, and digital signal processing progress, increasing demand is placed on the data converters. With portable electronics becoming increasingly popular, demands are also placed on the power consumption of circuits.

The kind of data converter that will be considered in this thesis can be described as a lowpass oversampling $\Delta\Sigma$ modulator based digital to analogue converter ($\Delta\Sigma$ DAC). It takes as input a sequence of words from a digital source, to produce a corresponding analogue voltage or current waveform at its output. This kind of data converter will be discussed more in-depth in the next chapter. A key point with the $\Delta\Sigma$ DAC is to move complexity from the analogue to the digital domain, reducing the stringent requirements on analogue circuitry found in conventional Nyquist-rate DACs. However, coarse quantisation comes at a price—it introduces substantial quantisation noise and tonality may occur. The $\Delta\Sigma$ modulator will shape most of this noise away from the output spectrum, but some noise will inevitably remain.

This thesis aims to explore whether it is feasible to dynamically alter the bandwidth of the $\Delta\Sigma$ DAC in order to enhance the overall performance. To achieve this, the input signal must be analysed to obtain frequency domain information, and this information must in turn be used to program the reconstruction filter with an appropriate transfer function. The reconstructed bandwidth will be dynamic with respect to properties of the input signal, and only spectral bands that contain signal energy will be reconstructed. Figure 1.1 on the following page shows a basic outline of

Figure 1.1 Basic outline of the proposed noise reduction scheme

the proposed quantisation noise reduction scheme. This approach poses two challenges which may incur overhead in the converter—the signal analysis (in the digital domain), and the adaptive signal reconstruction (in the analogue domain). Both of these challenges will be discussed further in chapter 3, and evaluated according to what level of performance gain they contribute compared to the complexity they incur. The viability of the approach depends on how efficiently the added complexity can be surmounted.

As the novelty of the approach discussed in this report is at the system level and not in the implementation of its individual components, emphasis is not placed on their implementation. Rather, published circuit topologies are relied upon. However, a simulation model is developed in the context of this work in order to gain insight into the performance improvement attributable to the system, as well as system level implementation issues.

1.1 Related work

A literature search was carried out to establish an overview of published works pertaining to adaptivity in $\Delta\Sigma$ modulator based data converters. A subset of the search results, those that appeared most relevant, were studied further. In a $\Delta\Sigma$ modulator based data converter system there are several dimensions along which online adaptation can take place in order to instigate performance improvements. However, in the scope of this report improvement in terms of signal to noise ratio (SNR) is investigated. The works considered in this section adaptively alters one or more parameters in the data converter system at run-time to better suit non-stationary signals.

In [2], a scheme for adaptively reducing the order of the noise shaping loop to avoid instability is outlined. The rationale is that an improvement in SNR can be made if a high order noise shaping loop is used when the input signal has a relatively low amplitude. If the amplitude of the input signal is large enough to cause a potential instability, the order is reduced.

Another scheme for improving SNR, in presence of a small amplitude

input signal, is presented in [3]. By amplifying weak input signals, the SNR is improved provided that the feedback signal in the $\Delta\Sigma$ modulator is compensated accordingly. The signal strength can be sensed directly at the input of the converter or by estimating the input signal strength from the $\Delta\Sigma$ modulator output. An adaptation algorithm is used to select appropriate gain for the pre-amplifier and for the feedback signal. Similar approaches are reported in [4] and [5].

As will be further discussed in the next chapter (cf. page 21), a random dither signal is frequently applied to the noise shaping loop of a $\Delta\Sigma$ modulator in order to avoid tonal noise components in the output signal. By applying the dither, the quantisation noise is made more white, resulting in a decorrelation of tonality. An inevitable side effect of this process is an increase in the noise floor, hence a reduced SNR performance of the converter. A dither capable of removing all tonal noise components may result in an unacceptable decrease in SNR. The author of [6] has made the observation that tonal behaviour is most prominent when the amplitude of the input signal is small, and suggests adapting the dither signal accordingly. The dither is modulated by the magnitude of the input signal so that full dither is applied when the instantaneous input signal is near DC, and no dither is applied when the input signal is at full dynamic range. Thereby a reduction in SNR performance is avoided when the dither is not needed.

The aforementioned adaptation schemes all adapt the converter system based on time domain properties of the input signal, or to some extent based on statistical measures on the time series. The authors of [7] describe a digital synthesiser capable of frequency, phase, and quadrature modulation. The synthesiser uses a programmable bandpass $\Delta\Sigma$ modulator and adjusts the notch frequency of the noise shaping function to suit the synthesised signal.

After studying available literature, it is apparent that dynamic run-time adaptation of parameters in a $\Delta\Sigma$ modulator based data converter system, based on frequency domain properties of the input signal, is not previously reported.

1.2 Thesis outline

The remainder of this report is structured as follows: Chapter 2 provides background on relevant topics. In chapter 3, the proposed approach is discussed. In chapter 4, a simulation model is devised. The system is simulated and results are presented. Finally, in chapter 5, a conclusion regarding the viability of this approach is given along with an outlook to further work.

This page is intentionally left blank

THIS CHAPTER provides background on topics relevant to the remainder of this report. The sections are structured according to figure 1.1 of the introduction. First, the lowpass oversampling $\Delta\Sigma$ modulator based digital to analogue converter ($\Delta\Sigma$ DAC) is explained in more detail. Then, suitable architectures for the tunable filter will be explored, and finally, suitable methods for signal analysis are considered.

2.1 Lowpass oversampling $\Delta\Sigma$ digital to analogue conversion

As previously stated, a key point of the $\Delta\Sigma$ DAC is to relax the requirements on analogue component matching, and instead rely on high-speed digital signal processing to attain the desired precision[8]. Figure 2.1 outlines the main components of a $\Delta\Sigma$ DAC. It is common to have few analogue output levels from the D/A converter; two output levels ensure linearity while avoiding matching problems. To facilitate this, the $\Delta\Sigma$ modulator reduces the signal resolution down to one bit—a process that introduces significant quantisation noise.* (i) The interpolation filter increases the sample rate (oversampling). The amount of quantisation noise energy remains constant, but is distributed over a larger area. The amount of noise energy in the band of interest is therefore reduced. (ii) The noise is filtered with a

*The band of interest is here defined as half the original sampling frequency.

Figure 2.1 High-level overview of a $\Delta\Sigma$ DAC

Figure 2.2 Interpolation filter

highpass filter, while the signal remains unchanged (noise shaping). The noise is thus molded away from the band of interest. After reconstructing the analogue output signal, only the band of interest remains. Hence, most of the undesired spectral components are rejected.

In the following subsections, the workings of each module in the the $\Delta\Sigma$ DAC will be discussed further. Finally, ideal estimates for noise performance are given.

2.1.1 Interpolation filter

The purpose of the interpolation filter is to oversample the input signal; to increase the sample-rate, f_s . The number of samples in $x[n]$ should be increased without altering its spectral contents to produce a sequence $\hat{x}[n]$.

This process can be implemented as shown in figure 2.2[9, ch. 10]. The first block, called an up-sampler, inserts $(N - 1)$ zero-valued samples between each input sample. N is often referred to as the oversampling ratio (OSR). The insertion of zero-valued samples causes the intermediate signal to contain unwanted spectral components, as it is no longer band-limited to its old Nyquist frequency. Since no information is to be added nor removed from the signal, $\hat{x}[n]$ should only contain spectral components up to half the original Nyquist-rate. A lowpass filter, $H(z)$, must therefore be designed to have a cut-off frequency at $\frac{\pi}{N}$ to avoid aliasing, and it must have a gain of N to account for the zero-valued samples. In the frequency domain, this process is equivalent to zero-padding the frequency spectrum. As the lowpass filter is a normal digital filter, several techniques are available for its design.

When implementing the structure in figure 2.2, additional considerations should be made—(i) the up-sampler will produce a sparse sequence, (ii) the passband will be narrow for high oversampling ratios, requiring high-order filters if FIR structures are used[10].

In order to conserve power, it is desirable to keep hardware that operates at the oversampled frequency to a minimum. If the overall desired transfer function of the interpolation filter is to be maintained, it is necessary to partition the filter. The work in [11] describes a general implementation where an interpolation filter is implemented as a cascade of filters. Filter complexity decreases further out in the cascade as the sampling frequency increases. Low complexity filters, such as the comb stages introduced in

Figure 2.3 Cascaded interpolation filter ($N = 4$)

[12], are used furthest out in the cascade where the operating speed is high. Earlier in the cascade, where frequencies are relatively low, precise filters are used. These early filters also correct for droop in the passband introduced by the high-speed filters. This cascading approach alleviates the filter order problem mentioned in the previous paragraph, because the passbands of filters further out in the cascade widen. This is illustrated in figure 2.3.

As for the sparse input problem, it can be handled by choosing a seemingly more complex filter structure known as a polyphase implementation; $H(z)$ is subsampled to produce N sub-filters (N still being the oversampling ratio). The polyphase implementation of figure 2.2 is depicted in figure 2.4 on the next page. These structures are shown to be equivalent in [9]. The impulse response for the sub-filters are given by: $h_p[n] = h[Nn + p]$, $p = 0 \dots (N - 1)$. The polyphase implementation can be combined with the cascading approach to produce an efficient interpolation filter implementation.

2.1.2 $\Delta\Sigma$ modulator

An outline of a first-order $\Delta\Sigma$ modulator is given in figure 2.5a on the following page. It is a feedback loop with a filter, $H(z)$, and a quantiser.

Figure 2.4 Polyphase interpolation filter

(a) $\Delta\Sigma$ modulator outline

(b) Loop filter, $H(z)$

(c) Quantiser

Figure 2.5 $\Delta\Sigma$ modulator

The $\Delta\Sigma$ modulator reduces the input signal resolution to facilitate the actual D/A conversion. This process is called quantisation—the quantiser maps several input values to each output value. The difference between the original and the quantised signal is called quantisation noise—a distortion introduced by the quantiser. The quantiser can be modelled as simply adding quantisation noise, $e[n]$, to the input signal. This is depicted in figure 2.5c.

$$e[n] = y[n] - v[n] \quad (2.1)$$

As the quantised signal contains less information, fewer bits are needed to represent it. When quantising a high resolution input signal down to two levels per sample (one bit), the quantisation noise can become excessive.

Oversampling contributes to reducing this noise, but it is often not practical to oversample enough to obtain the desired precision. Therefore, to reduce the quantisation noise further, the $\Delta\Sigma$ modulator shapes the noise away from the band of interest.

Noise shaping

As is evident from figure 2.5 on the preceding page, $H(z)$ acts on both the input signal, $x[n]$, and the quantisation error, $e[n]$. The system can be described in the z -domain as follows:

$$Y(z) = X(z) \frac{H(z)}{H(z) + 1} + E(z) \frac{1}{H(z) + 1} \quad (2.2)$$

From this, two transfer functions can be defined—a signal transfer function, $S(z)$, and a noise transfer function, $N(z)$.

$$S(z) = \frac{Y(z)}{X(z)} = \frac{H(z)}{H(z) + 1} \quad (2.3)$$

$$N(z) = \frac{Y(z)}{E(z)} = \frac{1}{H(z) + 1} \quad (2.4)$$

In a lowpass $\Delta\Sigma$ DAC, it is usually suitable to shift the quantisation noise to high frequencies, away from the band of interest. If $H(z)$ is defined as:

$$H(z) = \frac{1}{z - 1} \quad (2.5)$$

$S(z)$ and $N(z)$ becomes:

$$S(z) = z^{-1} \quad (2.6)$$

$$N(z) = 1 - z^{-1} \quad (2.7)$$

$S(z)$ is simply a delay and does not alter the frequency spectrum, while $N(z)$ has a highpass frequency characteristic. $N(z)$ will therefore suppress

Figure 2.6 Plot of noise shaping gain as a function of angular frequency

low-frequency noise and amplify high-frequency noise. Figure 2.6 shows a plot of $N(z)$. It should be noted that this z -domain model does not emphasise that $e[n]$ is deterministic and given by $x[n]$. A statistical description of the quantisation noise will be discussed in subsection 2.1.5.

The discussion so far has been limited to first-order modulators. Still, the loop filter, $H(z)$, is not required to be a first-order function. Using a loop filter of higher order provides an opportunity for more aggressive noise shaping. In figure 2.6, loop filters which give rise to noise transfer functions of the form $(1 - z^{-1})^n$ are shown. Any $H(z)$ that gives rise to a sound $N(z)$ and $S(z)$ is admissible, depending on the desired performance and application. Stability is a concern as $H(z)$ increases in order, and may limit the the performance that is feasible. Thus, a stable system calls for trade-offs to be made in the design[13]. Instead of relying on a single-stage implementation of a higher-order $\Delta\Sigma$ modulator, it is possible to make an arrangement with low-order $\Delta\Sigma$ modulators as building blocks. In an arrangement dubbed MASH[6], $\Delta\Sigma$ modulators are cascaded accompanied by error cancellation logic in order to eliminate the quantisation error from all but the last modulator. The input signal, $x[n]$, is used as input to the first modulator, while input to the remaining modulators is the quantisation error sequence, $e[n]$, from the preceding modulator. Output from all

modulators is used as input to the error cancellation logic which forms the output, $y[n]$. Unfortunately, the output from the MASH arrangement is inherently multibit, and thus incurs an increased complexity in the D/A module—either by requiring preprocessing of the multibit signal or by requiring several analogue output levels.

In summary, the $\Delta\Sigma$ modulator produces a single-bit representation of the input where most of the truncation error is contained in the high-frequency portion of the spectrum. The $\Delta\Sigma$ modulator does this by quantising the input signal, while shaping the resultant quantisation noise.

2.1.3 D/A conversion

The D/A converter is the interface between the digital and analogue domains. It takes as input a sequence of bits from the $\Delta\Sigma$ modulator and produces a corresponding output sequence of sampled analogue values. These analogue samples are represented by some physical quantity, such as a voltage potential or an electric current.

When the $\Delta\Sigma$ DAC is using a two-level quantiser, the D/A conversion is inherently linear. The spacing between the output levels does not matter, as long as the levels are stable. Several second-order effects may, however, corrupt the ideal linearity—power supply noise will make the output levels fluctuate, perhaps to a noticeable extent. Over-shoot and under-shoot, and rise-time and fall-time, may introduce distortion at the output, depending on the choice of reconstruction filter (cf. section 2.2 on page 22).

2.1.4 Reconstruction filter

The oversampled portion of the frequency spectrum is used by the $\Delta\Sigma$ DAC to hold quantisation noise. As discussed in subsection 2.1.2, a significant amount of quantisation noise is shifted from the band of interest into the oversampled spectrum. At the output of the $\Delta\Sigma$ DAC, the analogue signal is reconstructed by band limiting it to a frequency, f_b , the band of interest. For many applications this would be about half the original sampling frequency.

The reconstruction filter takes as input the sequence of analogue samples from the D/A converter, and outputs a continuous time signal, $y(t)$. The reconstruction filter may consist of several stages. Implementation alternatives for the reconstruction filter which are suitable for this application are discussed in section 2.2 on page 22.

2.1.5 Performance

The purpose of this subsection is to estimate the performance of a $\Delta\Sigma$ DAC in terms of signal to noise ratio (SNR).

When analysing the noise performance of the $\Delta\Sigma$ DAC, the non-linear behaviour of the quantiser presents a challenge. It is common to simplify the analysis by assuming the input, $x[n]$, to be busy and therefore seemingly uncorrelated with the quantisation noise, $e[n]$. In statistical terms, $e[n]$ is modelled as a zero-mean white noise wide-sense stationary (WSS) process. This enables linear analysis and simplifies estimating the power spectrum[14, 15, 16]. Furthermore, the quantiser is assumed to be uniform* with two output levels (one bit) and the $\Delta\Sigma$ DAC input to be a signed integer, b bits long. Using these assumptions, it is possible to quantify the worst-case quantisation error as:

$$-e_{max} \leq e[n] \leq e_{max}, \quad e_{max} = 2^{b-1} - 1 \quad (2.8)$$

The absolute value of $e[n]$, for all n , is limited to a quantity e_{max} . It is implied, from the assumptions above, that each of the possible values are equally likely.

For a zero-mean white-noise process, the power spectrum is equal to the variance of the process[17].

$$P_e(e^{j\omega}) = \sigma_e^2 = E\{e^2\} \propto \frac{1}{2e_{max} + 1} \sum_{k=-e_{max}}^{e_{max}} k^2 = \frac{4^b - 2^{b+1}}{12} \quad (2.9)$$

P_e is independent of ω and independent of the sampling frequency, f_s , so the amount of quantisation noise introduced by the $\Delta\Sigma$ modulator is constant and defined by b , the number of bits in the input word.

Quantisation noise undergoes noise shaping in the digital domain as discussed in subsection 2.1.2 on page 15. The resulting noise shaped power spectrum, $P_{e,ns}$, can be expressed as:

$$P_{e,ns}(e^{j\omega}) = \sigma_e^2 \left| N(e^{j\omega}) \right|^2 = \sigma_e^2 (1 - e^{-j\omega}) (1 - e^{j\omega}) \quad (2.10)$$

$$= 2\sigma_e^2 (\cos \omega - 1) \quad (2.11)$$

In the digital domain, the frequency variable is normalised such that 2π corresponds to f_s . When going from the digital to the analogue domain, the digital frequency variable ω is mapped to an analogue frequency f :

$$\omega = \frac{2\pi f}{f_s} \quad (2.12)$$

It must also be taken into account that only the band of interest, f_b , will pass through the analogue reconstruction filter, $R(s)$. For the purpose of estimating SNR performance, $R(s)$ is assumed to be an ideal brick-wall

*A uniform quantiser rounds an input sample to the closest output level

lowpass filter with cutoff at f_b . Thus, the total amount of noise power at the output of the converter is:

$$\begin{aligned}
 \sigma_{ey}^2 &= \frac{1}{f_s} \int_{-\frac{f_s}{2}}^{\frac{f_s}{2}} P_{e,ns} \left(e^{j\frac{2\pi f}{f_s}} \right) \cdot \left| R \left(e^{j\frac{2\pi f}{f_s}} \right) \right|^2 df \\
 &= \frac{1}{f_s} \int_{-\frac{f_b}{2}}^{\frac{f_b}{2}} P_{e,ns} \left(e^{j\frac{2\pi f}{f_s}} \right) df \\
 &= 2\sigma_e^2 \left(\frac{2f_b}{f_s} - \frac{\sin \left(\frac{2\pi f_b}{f_s} \right)}{\pi} \right) \\
 &\approx \sigma_e^2 \frac{\pi^2}{3} \left(\frac{2f_b}{f_s} \right)^3 = \sigma_e^2 \frac{\pi^2}{3} \text{OSR}^{-3}
 \end{aligned} \tag{2.13}$$

Because the signal transfer function, $S(z)$, does not alter the frequency spectrum, the signal power remains intact at the output. By using σ_x^2 to denote the total power in $x[n]$, and the definition of SNR (in dB), the following relationship arises:

$$\begin{aligned}
 \text{SNR} &= 10 \cdot \log_{10} \frac{\sigma_x^2}{\sigma_{ey}^2} \\
 &\approx 10 \cdot \log_{10} \frac{\sigma_x^2}{\sigma_e^2} - 10 \cdot \log_{10} \frac{\pi^2}{3} + 30 \cdot \log_{10} \text{OSR}
 \end{aligned} \tag{2.14}$$

As a rule of thumb, doubling the OSR improves SNR by 9 dB* in the case of a first order modulator.

Idle tones

As already mentioned, the quantisation noise sequence, $e[n]$, is a function of the input to the $\Delta\Sigma$ modulator and the noise shaping loop. Certain classes of input signals and low-order noise shaping loops give rise to concentrations of noise energy at deterministic frequencies[18, 19, 15]. This coloured noise is commonly referred to as idle tones, as it occurs when the converter is near idle.

Dithering is a common strategy employed to lessen the tonality of the quantisation noise[6]. The dither is a random noise signal applied to the noise shaping loop to decorrelate the tonality by whitening the quantisation noise at the cost of raising the noise floor.

The work in [20] presents a different strategy for lessening the impact of tonality—by placing the zeros of the loop filter, $H(z)$, slightly outside the unit circle, the system becomes chaotic. It was shown through simulation

* $30 \cdot \log_{10} 2 \approx 9$

that tonality is reduced. As with dithering, this approach raises the noise floor. The study reported in [21] concludes that chaos is less efficient than dithering.

2.2 Tunable filter

In a $\Delta\Sigma$ DAC, the purpose of the reconstruction filter is to reject the oversampled portion of the spectrum and only pass through the band of interest. In a lowpass $\Delta\Sigma$ DAC, this amounts to lowpass filtering the D/A converted $\Delta\Sigma$ bitstream. While it is possible to use any kind of analogue filter topology, sampled data filters are usually preferred. Like digital filters, sampled data filters process sampled input signals. However, the samples are not quantised, and they are stored as a physical quantity, usually a charge or a current, hence the precision of the samples are only limited by the performance of the signal processing components. When using sampled data techniques, it is thus possible to relax the requirements on the switching characteristics of the D/A converter compared to a continuous time filter, and to avoid the need for return-to-zero (RTZ) or similar output coding.

In this report, only switched-capacitor based filters, where samples are stored as charges in capacitors, are considered. Switched-current filters, where samples are stored as currents, have some salient features, but reported switched-capacitor structures are still able to attain higher performance in terms of accuracy and noise[22]. In the following subsection, the basic principles of switched-capacitor integrators are discussed. Because the scheme outlined in chapter 1 requires a tunable filter, structures with tunable transfer functions are needed. This is discussed in subsection 2.2.3 on page 25.

2.2.1 Switched-capacitor basics

Switched-capacitor refers to a technique for implementing resistor approximations using switches and capacitors. Switches can in turn be implemented using transistors, and thus, the technique lends itself well for use in integrated circuits when a large resistance is required. The switched-capacitor resistor approximations are commonly found in filtering applications, but are also used in, for example, gain stages and oscillators[23].

Figure 2.7 on the facing page shows a simplified switched-capacitor resistor approximation. Here, ϕ_1 and ϕ_2 are two non-overlapping phases of a clock signal controlling the switches S_1 and S_2 . The length of a clock cycle is denoted T , thus, ϕ_1 and ϕ_2 are each active slightly less than $T/2$ every clock cycle. The nodes V_1 and V_2 are connected to voltage sources

Figure 2.7 A basic switched-capacitor resistor approximation

with voltage potentials v_1 and v_2 respectively, slowly varying relative to T . Assuming an ideal capacitor, it is governed by the relationship

$$Q = C \cdot V \quad (2.15)$$

C is the capacitance in farads, Q is the charge in coulombs, and V is the voltage potential between signal ground and the switches. The voltage sources and switches are also assumed to be ideal, hence, the voltage sources are capable of supplying an infinite amount of current and no resistance is seen through the switches. After the first clock cycle, the initial charge present in the capacitor at $t = 0_-$ is lost, and the following analysis holds. At phase ϕ_1 , S_1 connects the capacitor to V_1 , while S_2 is open. This results in a charge of Q coulomb travelling from V_1 to charge (or discharge) the capacitor to v_1 volts, at which time equilibrium is reached and no more current flows. Referring to equation 2.15, the amount of charge transferred, Q , is thus given by $C(v_1 - v_2)$. The same is true for phase ϕ_2 , however, here S_2 is closed and S_1 is open, resulting in V_2 discharging (or charging) the capacitor to v_2 volts. The amount of charge in this phase is therefore equal to the amount of charge in the previous phase, Q . The tactic for seeing how this transfer of charges approximates a resistor is to look at the average current, i_{avg} , flowing from V_1 to V_2 . If node V_1 is at a higher potential than V_2 , the net result is that V_1 sources current and V_2 sinks the same amount of current. A charge Q is therefore transferred from V_1 to V_2 each clock cycle, resulting in an average current of $i_{\text{avg}} = QT^{-1} = C(v_1 - v_2)T^{-1}$. This is similar to the case of a true resistor where the current is given by $i = (v_1 - v_2)R^{-1}$ (R is the resistance in ohms). Here, the current is equal at all times, assuming a constant voltage, and need not be averaged. By comparing these two equations, the average resistance exhibited by the switched-capacitor structure can be expressed as

$$R_{\text{avg}} = \frac{T}{C} \quad (2.16)$$

In practical switched-capacitor circuits, none of the components making up the approximated resistor will be ideal. These non-idealities contribute to distortion and noise; some more than others. As the switches are usually implemented by MOSFET transistors, they will exhibit a finite non-zero resistance, R_{on} , when conducting. This resistance must be small enough so that the time constant formed by it and the capacitor leaves enough time for settling. The transistors used in the switches will release their channel charge when turned off. This signal dependent charge will contribute to the charge already stored in the capacitor and cause a voltage offset. The clock signals can therefore be said to feed through into the signal path*. The switches will also contribute thermal and $1/f$ noise, as well as junction leakage current. In many common configurations, the switch-capacitor is driven by an operational amplifier (op-amp) approximating a voltage source. The switched-capacitor circuit will therefore suffer a degradation in performance due to imperfections of the op-amp. Common non-idealities that need attention in switched-capacitor circuit design include offset voltage, gain bandwidth (GBW), slew rate, output resistance, and thermal and $1/f$ noise.

Various circuit implementation techniques are available for improving the performance of switched-capacitor circuits[24], including chopper modulation, which is used to escape $1/f$ noise as the input signal is modulated to a higher frequency before amplification, and demodulated afterwards. Correlated double sampling (CDS) is often used to compensate for voltage offset and finite gain in the op-amp, whereby low frequency noise and op-amp offset is sampled and then subtracted.

2.2.2 Switched-capacitor filter structures

As switched-capacitor based filters are operating in the z -domain, both FIR and IIR realisations are possible.

A switched-capacitor filter IIR structure can be obtained by simply replacing resistors with switched-capacitor resistor approximations in a continuous time prototype. Another strategy is to factor the transfer function into a cascade of second-order sections, and use specialised switched-capacitor biquads for their realisation. The final structure will then benefit from a performance gain inherited from the building blocks, as these are often tuned to work around switched-capacitor non-idealities.

As the absolute equivalent resistance of the switched-capacitor resistor approximation is only dependent on the clocking frequency and the capacitance value, accurate transfer functions can be realised as the accuracy of the poles and zeros depends on the ratios between capacitors, and not on

*If a manufacturing process without self-aligned gates is used, the clock will also be capacitively coupled to the source and drain of transistors because of overlap

absolute matching between capacitors and resistors. Even though precise absolute capacitance values can be difficult to achieve in integrated circuits, matching between capacitor devices in a monolithic circuit can be made very accurate if careful layout principles are adhered to[25]. The manufactured level of accuracy may be sufficient for many applications, thus avoiding the need for post-fabrication trimming.

It may be necessary to reduce the sampling rate prior to reconstructing the output signal if important criterions, such as power constraints, cannot be met at the oversampled output rate. Several switched-capacitor based decimation structures are reported[26, 27, 28].

2.2.3 Tunable switched-capacitor filters

The convenient tunability of switched-capacitor filters stems from the tunability of the resistance exhibited by the switched-capacitor resistor approximations. As already stated, the equivalent resistance is given by the capacitance value and the clock frequency. Thus, by altering the clock frequency or the capacitance value, the equivalent resistance changes. The clock frequency can be down-converted in the digital domain by a frequency divider. If it is not practical to alter the clock frequency, the capacitor can be replaced by a programmable capacitor array. Each capacitor in the capacitor array can be programmed to be grounded or left floating. Thus, the capacitance exhibited by the array will depend on which capacitors are active. In [29], a lowpass switched-capacitor filter with three selectable cutoff frequencies is presented. Programmability is achieved by digitally switching capacitors in or out.

A different approach was implored by the authors of [1]. Instead of using programmable capacitor banks to, in effect, alter the capacitance area, transfer function tunability is achieved by controlling capacitor charge. The authors propose to assemble higher-order filters from second-order FIR building blocks, and two slightly different building blocks are available for realising poles and zeros. Coefficient programmability is achieved by replacing one or more fixed capacitors in each FIR building block with a charge programming circuit. An illustrative version of the charge programming circuit is presented in figure 2.8 on the following page along with a timing pattern for the switches. The charge programming circuitry takes as input a voltage, v_e , and a sequence of bits, $\{b_0, b_1, \dots, b_{N-1}\}$. After the start-of-conversion (SOC) cycle, a charge, Q_b , is built up in C_b by first building a charge in C_a during the S cycle and then connecting C_a and C_b in parallel during the T cycle to force an equilibrium. Provided that the S cycle is long enough to allow settling, the charge in C_a builds up to $Q_a = C_a \cdot v_e$. Then, at the S cycle, C_a and C_b are connected in parallel to produce the equivalent capacitor $C_e = C_a + C_b$ with charge $Q_e = Q_a + Q_b$.

Figure 2.8 The charge programming circuit from [1]

Assuming $C_a = C_b$, the following relationship arise:

$$v_n = v_e \sum_{i=0}^n 2^{i-n-1} b_i \quad (2.17)$$

Where v_n is the voltage drop over C_e after input bit b_n . The output, v_s , is driven by an op-amp in an inverting integrator configuration. At the end-of-conversion (EOC) cycle, the open loop gain of the op-amp will drive C_b towards a short circuit between the true ground and the virtual ground at the negative input terminal of the op-amp, forcing the capacitor to release its charge, Q_b , to be subsequently trapped by C_c in the inverting integrator configuration and resulting in:

$$\begin{aligned} v_s &= -\frac{Q_b}{C_c} = -v_n \frac{C_b}{C_c}, \quad n = N - 1 \\ &= -v_e \frac{C_b}{C_c} \sum_{i=0}^{N-1} 2^{i-N-2} b_i \end{aligned} \quad (2.18)$$

The authors present circuit level simulations of a filter built from the FIR building blocks. The filter uses one FIR building block for poles and four FIR building blocks for zeroes, arriving at 8 zeroes and 2 poles, and is capable of filter coefficient programmability with a resolution of 8 bits. The circuit simulations demonstrate a good agreement between ideal and simulated frequency response.

2.3 Signal analysis

In the context of this work, the purpose of the signal analysis is to extract properties from the input signal in order to obtain programming information for the output signal reconstruction filter. This amounts to finding the spectral contents of the input signal, $x[n]$, as a function of time. In this section, background for the signal analysis is presented. Signal analysis in the context of this work is discussed further in section 3.1.1 on page 33.

2.3.1 The short time Fourier transform

The Fourier transform provides a frequency domain description of a signal and is a widely used and established method for signal analysis[30, 31].

An N -length sequence of samples, $x[n]$, can be expressed in terms of its transfer domain coefficients[9, ch. 3], $X[k]$, as:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{j\frac{2\pi kn}{N}}, \quad 0 \leq n \leq N-1 \quad (2.19)$$

Where

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi kn}{N}}, \quad 0 \leq k \leq N-1 \quad (2.20)$$

Equation 2.20 is usually referred to as the discrete Fourier transform (DFT), while 2.19 is referred to as the inverse discrete Fourier transform (IDFT). These transforms are usually implemented using efficient algorithms known as the fast Fourier transform (FFT) and the inverse fast Fourier transform (IFFT)[32].

The squared magnitude of the transform coefficients, $|X[k]|^2$, reveals how much power each frequency component contributes. However, they leave no reference to *when* they are occurring. In the context of this work, the spectral contents of the input needs to be tracked over time. The remainder of this subsection discusses the short time Fourier transform (STFT), which provides positioning in time for the transform coefficients. The localisation in time provided by the STFT is achieved by windowing the input signal. The window extracts a portion of the signal in time, thus, the frequency components reported by the transform must be occurring inside the window. The STFT results in a three-dimensional data set that can be visualised as a spectrogram plot. A common configuration is to use time as the x -axis, frequency as the y -axis, and power as a colour intensity, the z -axis.

The window used in the transform can be described in the time domain by its weights. Each weight is multiplied by its corresponding sample. The non-zero weights define which samples are to be extracted from the

sequence. By making the window narrower in the time dimension, the localisation property improves because fewer samples are extracted. However, this also results in fewer transform coefficients, and hence, a more coarse frequency resolution. This calls for an application-specific trade-off when designing the system. The DFT assumes a periodic function, with period equal to the length of the sequence. Thus, clipping an arbitrary signal in time may introduce distortion, which corrupts the analysis. To lessen the distortion, an artificial periodicity is introduced by choosing the weights so that they attenuate the beginning and end of the extracted sequence. In turn, this may require some overlap in the analysis to capture the entire signal in the transform domain. From the weights, several important performance metrics can be inferred. The work in [33] provides a comprehensive overview of windows in general, along with performance metrics and a comparison. The work in [34] presents a method for generating customised windows that, e.g., can be tuned to be more sensitive to certain frequencies as it provides the designer with some degrees of freedom.

2.3.2 Wavelets

The use of wavelet analysis and synthesis can be advantageous for certain applications. The analysis wavelet, which in turn makes up the basis, has localisation in both time and frequency. This is an important property of the wavelet transform, and in contrast to Fourier analysis where the frequency components are pure tones with an infinite extent in time—requiring the introduction of artificial periodicity by means of windowing if localisation in time is required (cf. subsection 2.3.1 on the preceding page).

Wavelet analysis decomposes the input signal into a linear combination of dilated and time shifted versions of the analysis wavelet, $\psi(t)$ —forming a wavelet basis, $\psi_{kn}(t)$, from the initial wavelet function. A signal, $x(t)$, can be expressed in terms of its wavelet transform coefficients, c_{kn} , and the wavelet basis as[35]:

$$x(t) = \sum_{k,n} c_{kn} \psi_{kn}(t) \quad (2.21)$$

where

$$\psi_{kn}(t) = 2^{\frac{k}{2}} \psi(2^k t - n) \quad (2.22)$$

provided that $x(t)$ and $\psi(t)$ meet certain requirements. n represents the time shift, while k defines the amount of dilation. c_{kn} can be computed as the inner product between $x(t)$ and $\psi_{kn}(t)$:

$$c_{kn} = \langle x(t), \psi_{kn}(t) \rangle \quad (2.23)$$

Equations 2.21 and 2.23 are not the most general form of the wavelet transform, but a discrete version that dyadically samples the analysis wavelet. Equation 2.21 *synthesises* the signal, $x(t)$, from its wavelet coefficients, c_{kn} ,

Figure 2.9 The Daubechies wavelet with three vanishing moments

Figure 2.10 Wavelet analysis implemented by a filter bank

while equation 2.23 analyses $x(t)$ to obtain c_{kn} . A salient feature of the wavelet transform for discrete signals is that its implementation is reduced to a recursive hierarchical quadrature mirror filter bank (QMF)[36], as depicted in figure 2.10. The analysis filter bank consists of the two filters $H_L(z)$ and $H_H(z)$, representing the lowpass and highpass parts respectively. Figures 2.9a and 2.9b shows a plot of the the DB6 wavelet with the impulse response for $H_H(z)$. From $h_H(z)$, the impulse response for each of the three other filters can be computed.

ECG signal compression

In compression applications, the wavelet basis is chosen so that relatively few wavelet coefficients accurately capture the signal, resulting in a compact representation. In the following, an electrocardiogram (ECG) signal compression scheme is used as an example.

An ECG trace is a recording of electrical activity in the heart, and is used for diagnostic purposes. Wavelets are used for a variety of tasks

in processing ECG signals[37]. Even though the signal is sampled at a relatively low frequency, compression is needed to make the total amount of data manageable; there may be a lengthy recording of each patient and it may be necessary to store data for several patients at once. The fidelity of the trace needs to survive the data reduction in order to retain a clinically acceptable quality. Percent root-mean-square difference (PRD) is often used as an objective metric of how much detail is retained in the synthesised signal (in equation 2.24, $x[n]$ denotes the original signal and $\hat{x}[n]$ is the decompressed approximation).

$$\text{PRD} = \sqrt{\frac{\sum (x[n] - \hat{x}[n])^2}{\sum x[n]^2}} \times 100\% \quad (2.24)$$

A straightforward scheme for compressing ECG traces using wavelets is presented in [38]. An ECG trace is partitioned into sequential blocks of equal length in the time dimension. Each block is then transformed using a multi-resolution wavelet analysis to obtain a hierarchy of coefficients. Then, a variant of the embedded zerotree wavelet algorithm (EZW)[39] is used to ensure that the most significant coefficients, and the most significant bits, are encoded first. Even if the bitstream is truncated to meet a given target bitrate, the most characteristic properties of the signal are transmitted, allowing for a faithful synthesis of the signal. Finally, a lossless compression is applied which exploits redundancy amongst the syntax elements to reduce the data rate further. This example will be used in section 3.1.1 on page 33 to illustrate how to efficiently obtain spectral information from the signal to be D/A converted.

2.3.3 Subband coded signals

When coding signals for storage or transmission, it is usually desirable to compress the signal to facilitate a reduced transmission time, or a reduced amount of required storage space. By using lossy compression, it is possible to trade signal accuracy for a reduced number of bits required to represent the signal.

Subband coding is a powerful and generic method for signal compression. The technique uses an analysis filter bank to partition the frequency spectrum, enabling separate quantisation at each subband according to some application specific criterion. A suitable encoding is applied before transmission or storage. At the receiver, the subband channels are decoded, and applied to an inverse filter bank (a reconstruction filter bank) in order to re-synthesise the signal. Figure 2.11 on the facing page shows an outline of a generic subband coding and decoding system[9, p. 800].

In the context of this work, the potential for exploiting syntax elements from subband coded signals in order to obtain spectral information is

Figure 2.11 Subband encoder and decoder system

investigated.

MPEG audio compression

While many compression systems use subband coding, MPEG audio[40] is used as an example in this report. Audio compression is a large and complex topic—here, only a brief overview will be given with emphasis on topics that can be usefully exploited in the context of this work.

The MPEG audio standards define how compliant bitstreams are to be decoded[41, 42]. With the encoder left as an exercise to the reader*, lucid innovations are permitted in future encoders while maintaining compatibility with existing decoders.

The audio parts of MPEG-1 and MPEG-2 are similar, however, MPEG-2 adds support for multichannel coding and low sampling frequencies. The standards define three separate layers with increasing coding compactness at the cost of increased codec complexity. Layer II and III are the most common in use. Layer II is used in digital versatile disk (DVD) and video compact disc (VCD) systems. A subset of Layer II is the audio compression used for digital audio broadcasts (DAB)[43]. Layer III is a very popular

*Outline of an example encoder is part of the standards

Figure 2.12 MPEG-1 audio decoding overview

format for the exchange of music on the Internet and for storing compressed audio on portable music players.

The compressed MPEG audio bitstream is a sequence of self-contained units, referred to as frames. Each frame contains some milliseconds of audio. Layer I exhibits the basic system common to all three layers. A layer I frame contains 12 samples for each of the 32 equally spaced subbands. Separate quantisation for each subband is possible, along with scale factors for the subband samples. Layer II is very similar but allows for grouping of samples, and requires the decoder to use more look-up tables when decoding. The number of subband samples is increased to 36. Layer III introduces finer-grained subbands by applying an MDCT transform[44] to each subband. In addition, layer III uses Huffman tables that allows fewer bits to be used to encode commonly occurring symbols, and thereby achieving a statistical coding gain.

The encoder will try to use as few bits as possible while maintaining an inaudible quantisation noise level for typical human listeners. By using a psychoacoustic model*, the encoder can determine the acceptable quantisation noise level and eliminate inaudible signal components. If an entire subband is idle or inaudible, it can be eliminated from the bitstream by not having any bits allocated.

This example will be used in section 3.1.1 on the next page to illustrate how to efficiently obtain spectral information from the signal to be D/A converted.

*A model describing signal masking, and audible sound pressure level as a function of frequency

AS STATED IN THE INTRODUCTION, the purpose of this report is to investigate whether making the bandwidth of the $\Delta\Sigma$ DAC dynamic with respect to properties of the input signal, yields a large enough performance gain to justify the complexity it incurs. In this chapter, strategies for implementing the building blocks of the system are discussed. Then, system level implementation issues are considered, and estimates of the performance gain attributable to the system are given.

3.1 Implementation

The implementation of the signal analysis and the tunable reconstruction filter needs to be as simple as possible in order to introduce the least amount of overhead compared to a static converter. Less overhead gives better performance in terms of complexity; thus a more viable approach.

First, considerations are made with respect to implementing the signal analysis circuitry, and how it can be simplified if the DAC is to some extent integrated with a signal decompression system. Then, some architectures for the tunable reconstruction filter are discussed.

3.1.1 Signal analysis

The purpose of the signal analysis in this application is to reveal the spectral contents of the input signal as a function of time. This information will in turn be used to tune the reconstruction filter so that only the spectral bands that contain energy will be reconstructed. The input to the analysis task is a digital signal, $x[n]$, a replica of the input to the $\Delta\Sigma$ DAC. The output is a list of digital control words for the tunable reconstruction filter. This process is conceptually depicted in figure 3.1 on the following page: Relevant properties are extracted from $x[n]$, and then mapped into control words, $c[n]$. The purpose of the control word mapping is to provide a layer

Figure 3.1 Input signal analysis with decoupling to reconstruction filter

of indirection, decoupling the semantics of the output of the analysis from the programming input of the reconstruction filter.

If, in a given application, some intrinsic properties of the input signal are known, an efficient and precise method for signal analysis can be put to use, for example, by using a model that relies on statistical properties of the input signal. In the context of this work, no such assumptions are made to allow for the broadest class of inputs. It is assumed that the task of the DAC is to convert the input signal as faithfully as possible without attenuating or amplifying frequency bands that contain signal energy.

In the previous chapter, two main opportunities for signal analysis were outlined: (i) by explicitly analysing the input signal, or (ii) by leveraging some existing information in the system to reveal the spectral contents. Frequency domain transforms with localisation in time, such as the STFT (cf. subsection 2.3.1 on page 27), are well suited for providing spectral information. The transform may require a dedicated signal processing subsystem, and thus it incurs an overhead. However, the analysis and its output can be tailored to the task at hand, and therefore provide a resolution and rate of the frequency domain data suited to the application at hand. Compressed domain signals, such as signals that are subband compressed (cf. subsection 2.3.3 on page 30) and wavelet compressed (cf. subsection 2.3.2 on page 28), seem like good candidates for classes of signals to be D/A converted where spectral information is available a priori from the point of view of the converter. By reusing spectral information encoded in the compressed signal, there is an opportunity for significantly reducing the complexity of the signal analysis compared to explicitly analysing the input signal. In the following, two examples of extracting spectral information from a compressed signal will be given.

MPEG audio compression

As discussed in subsection 2.3.3 on page 30, a typical encoder will try to use as few bits as possible in each subband, and not allocate bits to a subband that is idle. In the decoder, it is therefore possible to analyse the spectral contents of the signal by exploiting syntax elements in the compressed bitstream, namely the bit allocation information—causing the signal analysis performed by the encoder to be efficiently reused. Similar techniques have been used to eliminate unnecessary computation in the

Figure 3.2 Wrapping a static bandwidth $\Delta\Sigma$ DAC

decoder[45]. Other works address the problem of obtaining information, such as segmentation and classification information, directly from syntax elements in the compressed bitstream[46, 47, 48].

ECG signal compression

A wavelet-based scheme for compressing ECG signals was discussed on page 29 of the previous chapter. The compression achieved by this approach relies on the wavelet transform to concentrate much of the signal energy in relatively few coefficients. It was outlined how the wavelet transform could be implemented as a filter bank where, for each level, a QMF divides the spectrum into a lowpass component and a highpass component. Thus, by looking at which coefficients appear in the compressed bitstream, the spectral characteristics of the ideal synthesised signal can be inferred.

3.1.2 Tunable reconstruction filter

Section 2.2 on page 22 provided background on switched-capacitor filters—a technique that is frequently used and expected to be a viable technology platform for implementing the reconstruction filter.

The architectural requirements for the reconstruction filter will depend on the application at hand. While for some applications, a tunable filter bank of bandpass filters may be required, other applications could be expected to reap a substantial performance gain from a simple lowpass filter with a tunable cutoff frequency. In this subsection, architecture outlines for the reconstruction filter are discussed. To limit the complexity, only tunable lowpass filtering is considered.

In figure 3.2, a configuration is shown where a full static bandwidth $\Delta\Sigma$ DAC is utilised in the presented approach. In this configuration, a logical reconstruction filter is formed, comprised of the internal reconstruction filter of the DAC and the external tunable lowpass filter. As the DAC outputs a continuous time waveform, any filter deemed suitable can be put to use. In the illustration, a switched-capacitor filter is shown as it can be conveniently tuned by altering its clock frequency—here, the frequency

Figure 3.3 Full integration with the $\Delta\Sigma$ DAC

content information, provided externally, is mapped into an appropriate clock frequency for driving the switched-capacitors.

As the tunable filter is not fully integrated with the DAC, the resulting reconstruction filter is more complex than it possibly need be. Figure 3.3 shows two alternative approaches. Here, the filter takes as input the sampled analogue stream from the D/A directly. As this stream is output at a constant rate, the simple clock tuning approach presented earlier is no longer feasible. In figure 3.3a, an approach is presented whereby a programmable decimation filter adapts the sample rate to match the input of the reconstruction filter. As the D/A converts samples at the oversampled rate, it could be convenient for some applications to decimate the output stream before filtering[49]. However, this approach would require a programmable decimation rate, which could incur a level of complexity that outweighs the potential benefit. Note that if the decimation filter is implemented as a lowpass filter followed by a sample and hold circuit (S/H), the bandwidth of the lowpass filter can be fixed while the S/H is used to adapt the desired output sample rate. In figure 3.3b, another potential solution is outlined whereby the D/A output is optionally decimated at a fixed rate. Then, the result is processed by a switched-capacitor based filter with a tunable transfer function. By using a filter with a tunable transfer function, the input rate is constant and can be customised during design to match any desired input rate. However, circuitry for tuning the transfer function gives rise to overhead.

3.1.3 Assembling the system

Two properties define the ability of the system to track spectral content in the input signal over time: (i) The rate at which the signal analyser makes frequency content information available, and at what rate the reconstruction filter can be tuned. (ii) The resolution of the spectral information provided by the signal analyser, and the tuning capability of the reconstruction filter.

Some input signals may require using a programmable bandpass filter to effectively suppress quantisation noise when reconstructing the analogue output signal. For many applications however, a lowpass filter with a tunable cutoff frequency is more realistic in terms of complexity, while still making possible a substantial performance increase. Again, using MPEG audio as an example, it can be inferred from the bit allocation tables in the standard document[41] that the expected input signal will have most of its energy concentrated in the lower portion of the frequency spectrum.

Referring to figure 3.1 on page 34, the control word mapping is responsible for coordinating the signal analysis effort with tuning of the reconstruction filter. To avoid unnecessary computations and complexity, it is important that the capabilities of the reconstruction filter and the signal analysis are matched. The reconstruction filter does not need to allow for finer grained tuning than the signal analysis is able to provide. Similarly, if the analysis rate is greater than the programming rate capability of the filter, some of the information that has been obtained cannot be put to use. There obviously needs to be temporal correlation between the time-window in which the signal is analysed, and when the filter is programmed. It is to be expected that a substantial SNR penalty is incurred if spectral bands that contain signal energy are attenuated. Thus, the filter will have to be allowed a sufficient amount of time to settle on the programmed transfer function.

3.2 Performance gain

The basic premise underlying the presented approach is a promise of performance increase. In this section, the magnitude of the performance gain will be investigated by using the theoretical relations developed in subsection 2.1.5 on page 19. While it is possible that a similar approach could be put to use to enhance other aspects of performance, such as power consumption, only performance in terms of SNR is considered in this report.

By referring to equations 2.13 and 2.14, it is apparent that SNR can be improved by increasing the OSR—namely, the SNR will increase by 9 dB if the OSR is doubled, in the case of a first-order noise shaping loop. The OSR was defined in equation 2.13 to be proportional to the ratio

Figure 3.4 Relative SNR improvement as a function of bandwidth, $f_s = 384$ kHz

between the bandwidth of the oversampled spectrum and the bandwidth of the reconstruction filter. Thus, by reconstructing only the frequency bands that contain energy, and attenuating the frequency bands that only hold quantisation noise, the SNR will increase accordingly. However, it is important that frequency bands that contain signal energy are not attenuated. The improvement in SNR is plotted in figure 3.4. Equation 2.13, before the approximation, is used to calculate the theoretical noise power at the output of the converter, σ_{ey}^2 . For illustration purposes, an original sampling frequency of 48 kHz with eight times oversampling is assumed, arriving at $f_s = 384$ kHz. f_b is then swept from 750 Hz up to the full bandwidth, 24 kHz, to simulate the various bandwidth needs of the signal. The improvement in SNR is computed relative to the amount of quantisation noise at full bandwidth.

Which portions of the spectrum that contain signal energy will rely on statistical properties of the input signal—which in turn implies that the presented approach is not able to provide a guaranteed performance increase. However, if the expected class of input signals adheres to statistical relations in terms of frequency distribution, it will be possible to derive a statistical model for the expected performance increase. Again, using MPEG audio as an example, the compression scheme itself relies on the full spectrum not being occupied at all times, or at least not occupied by audible components that will have to be encoded. So in this case, the same statistical properties relied upon to achieve compression will contribute to an increase in SNR compared to a static-bandwidth DAC.

By making the assumption that idle tones are equally likely to appear anywhere in the output spectrum, the dynamic adjustment of the reconstructed bandwidth will also to some extent contribute to reducing tonality in the output noise.

This page is intentionally left blank

IN ORDER TO QUANTIFY THE PERFORMANCE GAIN experimentally and to gain practical insight into implementation issues, a simulation model was developed. In this chapter, an outline is given of the simulation model which is subsequently used in experiments. The experimental setup is discussed and results are presented.

4.1 Simulation model implementation

In this section, the implementation of the simulation model is outlined along with design considerations and a rationale for selecting the simulation framework.

The digital components of the simulation model were implemented using SystemC[50]. Implemented as a framework in C++, SystemC allows for a smooth transition from a conceptual design through to a full custom implementation. The implemented model can start off at a high level of abstraction emphasising architectural decisions, and may be gradually refined as the implementation proceeds to arrive at a cycle and bit accurate model suitable for synthesis by automated tools. The same testbench can drive the simulation through all stages of development, using a predefined set of stimuli and comparing the response to an accordant golden reference. Thus, avoiding an error-prone manual translation of high-level models into a register transfer level (RTL) description. An initiative is underway to introduce analogue and mixed signal (AMS) modelling capabilities to the SystemC framework[51]. In time, SystemC-AMS will enable seamless simulation of mixed signal systems. However, in its current state, SystemC-AMS is still lacking essential functionality. The work in [52] describes a separate effort aimed at implementing an analogue modelling framework for SystemC. Their implementation was however not yet publicly available when the simulation model was implemented.

The simulation model developed for use in this work was implemented at the behavioural level[53], where the hardware is described algorithmi-

cally. The model simulates an MPEG audio decoder, whose output is interpolated and then $\Delta\Sigma$ modulated. The $\Delta\Sigma$ bitstream is subsequently filtered to obtain a sampled analogue output waveform*. I.e. an MPEG audio bitstream is D/A converted by a $\Delta\Sigma$ DAC. A decoder capable of decoding a subset of MPEG Audio Layer II bitstreams was implemented—frames with a monaural signal at 48 kHz sampling rate without CRC codes are supported. The decoder was instrumented with the proposed method for obtaining spectral information. Optimisations described in [54] and [55] were applied to the subband filtering component. A multistage polyphase lowpass FIR filter was implemented to apply eight times interpolation to the output of the MPEG decoder (cf. subsection 2.1.1 on page 14). A first-order $\Delta\Sigma$ modulator was implemented to quantise and noise shape the interpolated signal (cf. subsection 2.1.2). An output buffer was implemented to ensure a smooth play-out of the single bit signal from the $\Delta\Sigma$ modulator. These modules form the digital components of the $\Delta\Sigma$ DAC. Due to concerns outlined above, an ad-hoc approach was implored for simulating the sampled data analogue reconstruction filter. The work in [1] was selected as a platform for the tunable filter due to its convenient programmability (cf. subsection 2.2.3). As the output rate from the D/A in this case is 384 kHz, the charge programming circuitry needs to be clocked at close to 10 MHz. As the output rate from the D/A and the clocking frequency for the charge programming circuit is relatively low, a decimation filter was not deemed necessary. As close agreement between predicted and simulated performance was shown in [1], a straightforward z-domain simulation of the lowpass IIR filter was deemed to be sufficient. As in [1], the filter has eight zeros and two poles and the coefficients were obtained using the approach described in [56], using software kindly provided by its author. The reconstruction filter has selectable cutoff at 750 Hz, 1500 Hz, 3000 Hz, 6000 Hz, 12000 Hz, and 20250 Hz. These cutoff frequencies correspond to subband indices 0, 1, 3, 7, 15, and 26 in the MPEG audio bitstream. The frequency response of two of the filters is plotted in figure A.2.4 on page 72 of the appendix. All transfer functions were designed to have a passband ripple of 1 dB and a stopband attenuation of 50 dB. Selected source code files are printed in section A.2 on page 57. Full simulation model source code is available[†].

4.2 Experimental Setup

Two experiments were designed in order to indagate the performance gain experimentally by means of simulation. The first experiment uses a

*A continuous time post filter is not part of the simulation model

[†]<http://heim.ifi.uio.no/~jorgenam/hovedfag/model/>

synthetic input signal, and real audio data is used in the last experiment. The TwoLAME encoder* was used to encode the input vectors as compliant MPEG Audio Layer II bitstreams. A low bitrate was used to deliberately starve the encoder. If the encoder is allowed a too generous bit budget, dummy data will be encoded in order to fill the target bitrate. Thus, the bit allocation information will not convey accurate spectral information. Additionally, in a realistic audio compression scenario, the bitrate would be set as low as possible. The default psychoacoustic model in TwoLAME, which is based on Psychoacoustic Model I described in Annex D of [41], was used.

For each experiment, the output of the interpolation filter and the output of the $\Delta\Sigma$ modulator was stored and used as reference for the reconstructed signal. Two reconstructed outputs were stored for comparison; one output where the cutoff of the reconstruction filter was adjusted dynamically according to the proposed approach, and another output reconstructed by a filter with a fixed cutoff frequency. In addition to the output signals, the programming input of the reconstruction filter and the frequency domain information obtained from the MPEG decoder was stored to aid in subsequent interpretation of the results. Four separate runs of the simulator were needed in each experiment to obtain all signals. This is not a problem as the simulation model is deterministic. After running the simulation, results from both experiments were analysed offline using a STFT, and the resulting data set was visualised as a spectrogram. As discussed in 2.3.1 on page 27, time is on the x -axis, frequency is on the y -axis, and on the z -axis, the colour intensity represents the signal power in dB. Only a fraction of the oversampled spectrum is shown to emphasise the interesting details.

All simulator output data from both experiments are available[†].

4.3 Results obtained from a synthetic input signal

The input vector in this experiment was a synthetic signal with a duration of five seconds encoded at 32 kbit/s. The signal contains a single tone alternating between 1 kHz and 2 kHz in one-second intervals. To avoid an overload situation in the decoder, which would result in clipping and distortion, the synthetic signal has a time domain amplitude of 0.8. Whereas the average power of a full scale sine wave is -3 dB, the average power of this signal is -4.95 dB, or -1.95 dB relative to full scale (FS)[‡].

The decoded and interpolated signal is visualised as a spectrogram in figure 4.4. In this figure, the magenta-coloured line depicts the spectral information obtained from the decoder. Figure 4.5 depicts the output from

*<http://www.twolame.org/>

[†]<http://heim.ifi.uio.no/~jorgenam/hovedfag/results/>

[‡] $-3.01 + 20 \log_{10} 0.8 = -4.95 \text{ dB}$

Figure 4.1 A time domain comparison of the 1 kHz output signal reconstructed with a dynamic and static cutoff frequency filter

the $\Delta\Sigma$ modulator which in turn will serve as input to the reconstruction filter. It can be seen from this figure that the noise introduced by the quantiser is shaped away from the signal of interest, as expected (cf. figure 2.6 on page 18). These spectrograms are included for reference.

Figures 4.6 and 4.7 present the reconstructed output from the simulated DAC as a spectrogram along with the cutoff frequency of the reconstruction filter shown as a magenta-coloured line. In figure 4.6, the output was reconstructed with a static cutoff filter, while figure 4.7 presents the results of dynamically altering the cutoff frequency to suit the frequency content of the input signal. A time domain comparison of one period of the reconstructed output signals is given in figure 4.1. The passband ripple and phase shift introduced by the reconstruction filters are clearly visible (cf. figure A.2.4 on page 72).

As the input signal in this experiment is stationary inside the five one-second time windows, it is possible to measure the SNR in each window. As in [57], the measurements were obtained from the power spectrum (cf. section A.1). In this experiment, noise was considered to be everything except the input tone and DC; thus, distortion is included in the noise energy measurement. This is referred to as signal to noise and distortion ratio (SINAD) in [57]. Power spectra for the first time window is plotted in figure 4.2. By comparing figures 4.2 and A.2.4, the influence of the reconstruction filter is evident. Noise in the oversampled portion of the spectrum is attenuated by about 50 dB, and the input tone is amplified by about 0.5 dB in 4.2c and -0.5 dB in 4.2b. This modulation of the input tone stems from the 1 dB passband ripple of the reconstruction filter.

SNR figures were obtained from both the dynamic and static output sequences. Finally, the time windows with equal cutoff frequency were

(a) $\Delta\Sigma$ modulator output

(b) output reconstructed with the static cutoff filter (20250 Hz)

(c) output reconstructed with the dynamic cutoff filter (1500 Hz)

Figure 4.2 A frequency domain presentation of the first time window

Table 4.1 Summary of measured results

		SNR (dB)	
		Average	Variance
Static	1000 Hz	21.89	$1.69 \cdot 10^{-5}$
	2000 Hz	21.74	$4.65 \cdot 10^{-4}$
Dynamic	1000 Hz	47.11	$5.55 \cdot 10^{-8}$
	2000 Hz	44.53	$6.22 \cdot 10^{-4}$
Improvement	1000 Hz	25.22	$1.74 \cdot 10^{-5}$
	2000 Hz	22.79	$2.16 \cdot 10^{-3}$

averaged to obtain the results presented in table 4.1. Ideally, all SNR measurements in similar time windows would be equal. This is not to be expected from practical measurements, but the underlying assumption is confirmed by the low variance, σ^2 .

In the following, the measured SNR improvement is compared to the SNR improvement predicted from theory. We cannot expect the input vector in this experiment to produce a white quantisation noise sequence, as it contains only a single tone of relatively low frequency[19]. Thus, equation 2.9 on page 20 cannot be used to obtain an accurate estimate of the quantisation noise power. However, equation 2.14 is still useful for predicting the relative performance increase, since quantisation noise power is a common factor that cancels out.

$$\begin{aligned}
\text{SNR}_{\text{diff}} &= \text{SNR}_{\text{dynamic}} - \text{SNR}_{\text{static}} = 10 \cdot \log_{10} \frac{\sigma_{ey:\text{static}}^2}{\sigma_{ey:\text{dynamic}}^2} \\
&= 10 \cdot \log_{10} \frac{f_s \sin(2\pi f_{b:\text{static}} f_s^{-1}) - 2\pi f_{b:\text{static}}}{f_s \sin(2\pi f_{b:\text{dynamic}} f_s^{-1}) - 2\pi f_{b:\text{dynamic}}} \quad (4.1) \\
&\approx 30 \cdot \log_{10} \frac{f_{b:\text{static}}}{f_{b:\text{dynamic}}}
\end{aligned}$$

The above equation predicts a performance increase of 33.9 dB for time windows where a cutoff frequency of 1500 Hz was selected, and 24.9 dB for time windows reconstructed with a 3000 Hz cutoff filter. Although the measured performance increase is lower than expected from theoretical estimates, there is still a worthwhile performance improvement. The lower performance increase stems from violations of ideal theoretical assumptions; the assumption of a white quantisation noise sequence and a brick wall

Figure 4.3 A time domain comparison of the 2 kHz output signal reconstructed with a dynamic and static cutoff frequency filter

reconstruction filter.

4.4 Results obtained from audio data input

In this experiment, an input vector with five seconds of audio data encoded at 48 kbit/s was used. As in the previous experiment, the spectrograms depicting the interpolated output from the decoder and the $\Delta\Sigma$ modulated bitstream are provided for reference, and plotted in figures 4.8 and 4.9.

In order to accommodate the desired bitrate, the audio encoder has bandlimited the signal to 6 kHz, as evident by the magenta-coloured line in figure 4.8 depicting the frequency domain information obtained from the decoder. Thus, the reconstruction filter is given an opportunity to suppress a substantial amount of quantisation noise that would otherwise have been present in the output signal. As evident by comparing figures 4.10 and 4.11, there is a significant suppression of quantisation noise in the output signal reconstructed with the dynamically adapted reconstruction filter. Figure 4.4

shows a time domain comparison of a portion of the reconstructed signal. In both plots, the dotted line represents the input to the $\Delta\Sigma$ modulator, and is provided for reference. The noise suppression is clearly visible in the time domain as well.

In this experiment, the SNR improvement was not easily quantifiable due to the non-stationary nature of the input signal.

4.5 Summary of measured results

In the preceding sections, two experiments were presented to experimentally verify and substantiate the promised performance increase. In the first experiment, the synthetic input signal was designed to be stationary inside one-second time windows. SNR figures were obtained, which demonstrated a performance increase as expected. The measured SNR figures were also compared to theoretical estimates. In the second experiment, a performance increase was demonstrated with real audio data, but in this case no SNR figures were obtained. However, a substantial suppression of quantisation noise is evident from the visualised results.

Figure 4.4 Interpolated output from the MPEG audio decoder

Figure 4.5 Unfiltered output from the $\Delta\Sigma$ modulator

Figure 4.6 Output reconstructed with a static cutoff frequency filter

Figure 4.7 Output reconstructed with a dynamic cutoff frequency filter

Figure 4.8 Interpolated output from the MPEG audio decoder

Figure 4.9 Unfiltered output from the $\Delta\Sigma$ modulator

Figure 4.10 Output reconstructed with a static cutoff frequency filter

Figure 4.11 Output reconstructed with a dynamic cutoff frequency filter

IN THE PRECEDING CHAPTERS, a novel approach for adapting the bandwidth of a $\Delta\Sigma$ DAC was presented along with relevant background material. The goal of this chapter is to conclude on the viability of this approach based on the theory and the simulation results presented in the preceding chapters. An outlook to further work is also given.

5.1 Conclusion

In this report, a scheme for dynamically adapting the bandwidth of a $\Delta\Sigma$ DAC was presented. The approach allows for a further suppression of quantisation noise if the input signal is non-stationary. The resulting performance increase was illustrated experimentally by running a simulation model of the presented approach. As was shown in the previous chapter, the simulated performance improvement was lower than what could be expected based on theoretical predictions, but still a substantial improvement in SNR was demonstrated. However, it is not possible to guarantee a performance increase in the most general case, as it depends on the variability of frequency band usage as a function of time of the the signal to be converted. However, for certain classes of signals, such as certain compressed signals, a statistical performance increase can be expected, as the necessary assumptions are already made by the underlying compression scheme.

As established in the introduction, the viability of the presented approach is to a large extent determined by the complexity it incurs. Although the increase in complexity attributable to the programmable reconstruction filter is to some extent invariant of the input signal, the selectable transfer functions must be designed to efficiently suppress quantisation noise. Thus, different classes of input signals will dictate different requirements to the reconstruction filter. The overhead from gathering spectral information depends to a large extent on the type of input signal to be converted and whether existing system components can be leveraged to provide this

information.

In summary, the presented approach can be expected to be viable if the frequency band utilisation of the input signal is varying as a function of time to an extent where it is not economical in some sense to allow the $\Delta\Sigma$ DAC to unconditionally operate at its full bandwidth, and it is convenient to obtain frequency domain information, for example by integrating the converter with a decoder where frequency domain information is available a priori to the system.

5.2 Future work

It is imperative to determine to what extent the adaptation introduces distortion when the coefficients in the reconstruction filter change, and to investigate possible countermeasures.

A full custom prototype implementation would be interesting. Even though the implementation itself is a practical task without much scientific merit, it would allow for more realistic measurements to be made. It would open up for a more accurate assessment of the approach, because power consumption can be measured and the actual increase in complexity can be quantified. It is believed that the digital simulation modules are synthesisable with only minor adjustments. However, some mixed signal design remains to be done.

In order to broaden the applicability of the approach, it is of interest to identify other classes of input signals and compression schemes that lends itself well to the extraction of frequency domain data.

Investigating whether it is practical to adaptively alter the noise shaping loop, $H(z)$, is an interesting topic. If $H(z)$ is dynamically altered to match the reconstruction filter by shaping the noise to bands that are attenuated, there is an opportunity for further performance increase. It is also worth investigating to what extent it is possible to leverage the added infrastructure to accommodate the adaptive pre-amplification scheme discussed in section 1.1 on page 10, to increase the performance further without adding significant complexity.

Identifying to what extent there are possibilities for run-time adaptation of other dimensions, for example to yield an improvement in power consumption, is another possible future direction.

Whether the presented approach can be gainfully applied to analogue to digital converters (ADCs) is another aspect of the presented approach that needs further investigation. The spectral information would have to be obtained from the analogue domain with low overhead. To what extent it is feasible to estimate the frequency content in the analogue input signal from the digital output of the $\Delta\Sigma$ modulator should also be looked into, as this would reduce the complexity in the analogue domain.

source code

A.1 Octave scripts

```
1  function [f,X] = fftpowerspec( x, fs, N, win )

    % [f,X] = fftpowerspec -- Returns a single sided power spectrum
    % representation of the time sequence x, with sampling frequency fs, using
    % an N length discrete Fourier transform.
6  %
    % To plot result (in dB, clipped to -120 dB):
    % X_dB = max( 10 * log10(X), -120 );
    % plot( f, X_dB );
    %
11 % Author: Jorgen Andreas Michaelsen <jorgenam@ifi.uio.no>
    % Based on: http://www.mathworks.com/support/tech-notes/1700/1702.html

    if nargin < 3
16       N = length( x );
    end

    if nargin < 4
        win = blackmanharris( length(x) );
21    end

    if rem(N,2) == 1
        error( 'x_length_must_be_even' )
        return;
26    end

    Nhalf = (N+1) / 2;

    % window the signal to introduce artificial periodicity
31    x = x .* win';

    % do the actual Fourier transform using the efficient fft
    X = fft( x, N );

36    % signal is real so only first part is interesting, i.e. unique
```

```

X = X(1:Nhalf);

% make magnitude spectrum
X = abs( X );

41 % fft algorithm omits compensating for number of samples
X = X / length(x);

% window will introduce some loss, typically a few dB
46 X = X / mean(win);

% make the power spectrum
X = X .^ 2;

51 % scale since we only use one side of the spectrum,
% everything except dc and nyquist needs doubling
X(2:end-1) = X(2:end-1) * 2;

% make index to frequency map
56 f = (0:Nhalf-1) * fs / N;

function [A,pp,pf] = outputspectrogram( aufile, imgfile )

% outputspectrogram -- Make spectrogram of simulation result
4 %
% Author: Jorgen Andreas Michaelsen <jorgenam@ifi.uio.no>

[x,fs] = auload( aufile );
9 N = fs / 100; % make input tones fall dead center in a fft bin

% slice up signal in time domain and analyse each slice
% in the frequency domain
14 A = []; % the spectrogram picture
pp = []; % peak power in each slice
pf = []; % frequency of peak

19 for pos = 1:(N*0.25):(length(x)-N+1)

    slice = x(pos:pos+N-1)';
    [f,X] = fftpowerspec( slice, fs, N ); % convert to power spectrum

24 % funny and useful information
    [peak,pos] = max( X );
    pp = [pp peak];
    pf = [pf f(pos)];

29 % append this slice to the spectrogram
    A = [A X'];

end

```



```

34  if( nargin == 2 )
        A = A(1:(N/8),:);                                % only a fraction of spectrum
        A = max( -100, 10*log10(A) );                    % make dB (clipped)
        A = flipud( 64 - fix( A * 63 / (-100) ) );      % flip frequency scale
        imwrite( imgfile, A, jet );                      % write to image file
39  end

function fftpowerspecanalyse( f, X )

3  % fftpowerspecanalyse -- Analyse the power spectrum to obtain performance
% metrics. X is the spectral power as a function of fft bin index, and f
% is a mapping between fft bin index and frequency (in Hz).
%
% Author: Jorgen Andreas Michaelsen <jorgenam@ifi.uio.no>
8  % Based on: http://www.maxim-ic.com/appnotes.cfm/appnote\_number/729
%              http://www.maxim-ic.com/appnotes.cfm/appnote\_number/728

        span = 20; % to make a vicinity of input tone, and dc

13  % find input tone (ignoring DC)
        [peakval, peakpos] = max( X(span:end) );
        peakpos = peakpos + span - 1;
        f_in = f(peakpos)
18  P_in = peakval

        % find noise (everything else, since we count distortion as noise)
        P_noi = sum( X(span:peakpos-span) );
        P_noi = sum( X(peakpos+span:end) ) + P_noi

23  % performance metric
        disp( 'signal_to_noise_and_distortion_ratio:' );
        10 * log10( P_in / P_noi )

```

A.2 Simulator

Not all simulator module source code files are printed in this section due to space constraints. Source code for all modules are available from <http://heim.ifi.uio.no/~jorgenam/hovedfag/model/>

A.2.1 Top-level

simulator.cpp

```

1  #include <stdio.h>
   #include <string.h>
   #include <signal.h>
   #include <sys/types.h>

6  #include "modules/leds/leds.h"
   #include "modules/testbench/input.h"
   #include "modules/testbench/output.h"
   #include "modules/testbench/mpgout.h"

```

```

#include "modules/testbench/dsmout.h"
11 #include "modules/mpeg/mpgparse.h"
#include "modules/mpeg/mpgfilter.h"
#include "modules/interp/interp.h"
#include "modules/dsmmod/dsmmod.h"
#include "modules/fifo/fifo.h"
16 #include "modules/recon/recon.h"

void sighandler( int signal )
{
21   sc_stop();
}

int sc_main( int argc , char **argv )
{
26   bool          mpegoutput = false;
   bool          dsmmodoutput = false;
   bool          dynamic = true;
   char          *input_file = NULL;
   char          *output_file = NULL;
31   char          *trace_file = NULL;
   char          *freq_log_file = NULL;
   sc_trace_file *tf = NULL;

36   signal( SIGINT, sighandler );
   signal( SIGPIPE, sighandler );
   cerr << endl;

   for( int i = 1; i < argc; i++ )
41     if( !strcmp("--trace=",argv[i],8) )
       trace_file = argv[i] + 8;
     else if( !strcmp("--input=",argv[i],8) )
       input_file = argv[i] + 8;
     else if( !strcmp("--output=",argv[i],9) )
46       output_file = argv[i] + 9;
     else if( !strcmp("--freq-log=",argv[i],11) )
       freq_log_file = argv[i] + 11;
     else if( !strcmp("--mpeg-output",argv[i]) )
       mpegoutput = true;
51     else if( !strcmp("--dsmmod-output",argv[i]) )
       dsmmodoutput = true;
     else if( !strcmp("--static",argv[i]) )
       dynamic = false;
     else
56       cerr << "simulator:_" << argv[i] << "_not_understood" << endl;

   if( input_file != NULL )
     cerr << "simulator:_input_" << input_file << endl;
   if( output_file != NULL )
61     cerr << "simulator:_output_" << output_file << endl;
   if( freq_log_file != NULL )
     cerr << "simulator:_freq_log_" << freq_log_file << endl;
   if( trace_file != NULL )
     cerr << "simulator:_trace_" << trace_file << endl;
66   if( mpegoutput )
     cerr << "simulator:_mpeg_output_only" << endl;
   if( dsmmodoutput )
     cerr << "simulator:_dsmmod_output_only" << endl;
   cerr << endl;

71   sc_set_default_time_unit( 1, SC_PS );

   if( trace_file != NULL )
     tf = sc_create_vcd_trace_file( trace_file );
76   sc_clock          clock( "CLK", 434030, 0.5 );
   sc_signal<bool>    reset;

   if( tf != NULL )
81   {
     sc_trace( tf, clock.signal(), "clock" );
     sc_trace( tf, reset, "reset" );
   }

86   sc_signal<bool> mpeg_error;
   sc_signal<bool> sync_error;
   sc_signal<bool> fifo_error;

   if( tf != NULL )
91   {
     sc_trace( tf, mpeg_error, "mpeg_error" );

     leds leds1( "leds" );

```

```

    leds1.CLK( clock );
    leds1.RST( reset );
96    leds1.errSYNTAX( mpeg_error );
    leds1.errBUFFER( fifo_error );
    leds1.errSYNC( sync_error );

    sc_signal<sc_uint<8> > bitstreambyte;
101    sc_signal<bool> ready_input;
    sc_signal<sc_uint<8> > request_input;

    if( tf != NULL )
    {
106        sc_trace( tf, bitstreambyte, "bitstreambyte" );
        sc_trace( tf, ready_input, "ready_input" );
        sc_trace( tf, request_input, "request_input" );
    }

111    input input1( "input_source" );
    input1.CLK( clock );
    input1.RST( reset );
    input1.outREQ( request_input );
    input1.outRDY( ready_input );
116    input1.out( bitstreambyte );
    input1.set_file( input_file );

    sc_signal<sc_int<17> > subbandsample;
    sc_signal<bool> frame_request;
121    sc_signal<bool> subband_request;
    sc_signal<bool> subband_ready;
    sc_signal<sc_uint<2> > sampling_frequency;
    sc_signal<sc_uint<5> > max_frequency;

126    if( tf != NULL )
    {
        sc_trace( tf, subbandsample, "subbandsample" );
        sc_trace( tf, frame_request, "frame_request" );
        sc_trace( tf, subband_request, "subband_request" );
131        sc_trace( tf, subband_ready, "subband_ready" );
    }

    mpgaudparse parser( "mpeg_parser" );
    parser.CLK( clock );
136    parser.RST( reset );
    parser.inRDY( ready_input );
    parser.inREQ( request_input );
    parser.in( bitstreambyte );
    parser.outSAMPFREQ( sampling_frequency );
141    parser.outMAXFREQ( max_frequency );
    parser.outFRAMEREQ( frame_request );
    parser.outSBREQ( subband_request );
    parser.outRDY( subband_ready );
    parser.out( subbandsample );
146    parser.errSYNTAX( mpeg_error );
    parser.errSYNC( sync_error );

    sc_signal<sc_int<17> > audiosample;
    sc_signal<bool> valid_audiosample;
151    sc_signal<bool> ack_audiosample;

    mpgaudfilter subbandfilter( "mpeg_filter" );
    subbandfilter.CLK( clock );
    subbandfilter.RST( reset );
156    subbandfilter.inFRAMEREQ( frame_request );
    subbandfilter.inSBREQ( subband_request );
    subbandfilter.inRDY( subband_ready );
    subbandfilter.in( subbandsample );
    subbandfilter.outRDY( valid_audiosample );
161    subbandfilter.outACK( ack_audiosample );
    subbandfilter.out( audiosample );

    sc_signal<sc_int<17> > upsamplesample;
    sc_signal<bool> valid_upsamplesample;
166    sc_signal<bool> ack_upsamplesample;

    interp interp1( "interpolation" );
    interp1.CLK( clock );
    interp1.RST( reset );
171    interp1.in( audiosample );
    interp1.inRDY( valid_audiosample );
    interp1.inACK( ack_audiosample );
    interp1.out( upsamplesample );
    interp1.outRDY( valid_upsamplesample );
176    interp1.outACK( ack_upsamplesample );

```

```

    if( mpegoutput )
    {
        sc_signal<bool> valid_adaptersample;
181      sc_signal<float> adaptersample;

        mpgout mpgout1( "mpeg_adapter" );
        mpgout1.CLK( clock );
        mpgout1.RST( reset );
186      mpgout1.in( upsamplesample );
        mpgout1.inRDY( valid_upsamplesample );
        mpgout1.inACK( ack_upsamplesample );
        mpgout1.out( adaptersample );
        mpgout1.outRDY( valid_adaptersample );

191      output output1( "output_sink" );
        output1.inRST( reset );
        output1.inRDY( valid_adaptersample );
        output1.inFREQPROG( max_frequency );
196      output1.in( adaptersample );
        output1.fixed_point_output();
        output1.set_file( output_file );
        output1.set_log_file( freq_log_file );

201      reset = 1;
        sc_start( clock, 6 );
        reset = 0;
        sc_start( clock, -1 );
    }
206  else if( dsmodoutput )
    {
        sc_signal<bool> dsmodsample;
        sc_signal<bool> valid_dsmodsample;
        sc_signal<bool> ack_dsmodsample;

211      dsmod dsmod1( "ds_modulator" );
        dsmod1.CLK( clock );
        dsmod1.RST( reset );
        dsmod1.in( upsamplesample );
        dsmod1.inRDY( valid_upsamplesample );
216      dsmod1.inACK( ack_upsamplesample );
        dsmod1.out( dsmodsample );
        dsmod1.outRDY( valid_dsmodsample );
        dsmod1.outACK( ack_dsmodsample );

221      sc_signal<bool> valid_adaptersample;
        sc_signal<float> adaptersample;

        dsmout dsmout1( "dsm_adapter" );
226      dsmout1.CLK( clock );
        dsmout1.RST( reset );
        dsmout1.in( dsmodsample );
        dsmout1.inRDY( valid_dsmodsample );
        dsmout1.inACK( ack_dsmodsample );
231      dsmout1.out( adaptersample );
        dsmout1.outRDY( valid_adaptersample );

        output output1( "output_sink" );
        output1.inRST( reset );
236      output1.inRDY( valid_adaptersample );
        output1.inFREQPROG( max_frequency );
        output1.in( adaptersample );
        output1.fixed_point_output();
        output1.set_file( output_file );
241      output1.set_log_file( freq_log_file );

        reset = 1;
        sc_start( clock, 6 );
        reset = 0;
246      sc_start( clock, -1 );
    }
  else
  {
        sc_signal<bool> dsmodsample;
        sc_signal<bool> valid_dsmodsample;
251      sc_signal<bool> ack_dsmodsample;

        dsmod dsmod1( "ds_modulator" );
        dsmod1.CLK( clock );
256      dsmod1.RST( reset );
        dsmod1.in( upsamplesample );
        dsmod1.inRDY( valid_upsamplesample );
        dsmod1.inACK( ack_upsamplesample );
        dsmod1.out( dsmodsample );
261      dsmod1.outRDY( valid_dsmodsample );

```

```

    dsmod1.outACK( ack_dsmodsample );

    sc_signal<bool>      fifosample;
    sc_signal<bool>      clock_fifosample;
266    sc_signal<sc_uint<5> > prog_frequency;

    fifo fifol( "playout_fifo" );
    fifol.CLK( clock );
    fifol.RST( reset );
271    fifol.in( dsmodsample );
    fifol.inRDY( valid_dsmodsample );
    fifol.inACK( ack_dsmodsample );
    fifol.inMAXFREQ( max_frequency );
    fifol.inSAMPFREQ( sampling_frequency );
276    fifol.out( fifosample );
    fifol.outCLK( clock_fifosample );
    fifol.outMAXFREQ( prog_frequency );
    fifol.errBUFFER( fifo_error );

281    sc_signal<float> outsample;

    recon recon1( "reconstruction" );
    recon1.inRST( reset );
    recon1.inCLK( clock_fifosample );
286    recon1.inFREQPROG( prog_frequency );
    recon1.in( fifosample );
    recon1.out( outsample );
    if( dynamic )
        recon1.set_dynamic();
291    else
        recon1.set_static();

    output output1( "output_sink" );
    output1.inRST( reset );
296    output1.inRDY( clock_fifosample );
    output1.inFREQPROG( prog_frequency );
    output1.in( outsample );
    output1.floating_point_output();
    output1.set_file( output_file );
301    output1.set_log_file( freq_log_file );

    reset = 1;
    sc_start( clock, 6 );
    reset = 0;
306    sc_start( clock, -1 );
}

cerr << "simulation_ends_at_" << (sc_time_stamp().to_seconds()*1000);
cerr << "_ms" << endl;
311    return 0;
}

```

A.2.2 MPEG audio parser

mpgaudparse.h

```

2    #define CACHE_SIZE    256                // words

SC_MODULE( mpgaudparse )
{
7    sc_in<bool>      RST;

    sc_out<sc_uint<8> > inREQ;
    sc_in<bool>      inRDY;
    sc_in<sc_uint<8> > in;

12    sc_out<sc_uint<5> > outMAXFREQ;
    sc_out<sc_uint<2> > outSAMPFREQ;
    sc_in<bool>      outFRAMEFREQ;
    sc_in<bool>      outSBREQ;
    sc_out<bool>      outRDY;
17    sc_out<sc_int<17> > out;

    sc_out<bool>      errSYNTAX;
    sc_out<bool>      errSYNC;

22    sc_in_clk      CLK;

    sc_uint<32>      cache[CACHE_SIZE];

```

```

        sc_uint<8>      cache_rd_pos;
        sc_uint<8>      cache_wr_pos;
27      sc_uint<32>      cache_word;
        sc_uint<6>      cache_wordpos;

        SC_CTOR( mpgaudparse )
32      {
        SC_CTHREAD( run, CLK.pos() );
        watching( RST.delayed() == true );
        }

37      void run( void );
        void write_sample( sc_int<17> );
        sc_uint<16> read_sample_code( sc_uint<5> );
        sc_uint<16> get_next_word( sc_uint<5> );
        bool get_next_bit( void );
42      void refresh_cache( void );
        void mpgaudparse::fill_cache( void );
        sc_int<17> invq( sc_uint<16>, sc_uint<4>, sc_uint<5> );
    };

```

mpgaudparse.cpp

```

#include <stdio.h>
#include <systemc.h>

#include "mpgaudparse.h"
5  #include "mpgyscale.h"
    #include "mpginvq.h"
    #include "mpgalloc.h"
    #include "mpggrouper.h"

10 #define MPEG_ERROR { errSYNTAX.write( true ); wait(); continue; }

#define SYNC_PATTERN 0xFFFFF
#define SBLIMIT 32

15 bool mpgaudparse::get_next_bit( void )
{
    if( cache_wordpos == 0 )
        refresh_cache();
20     cache_wordpos--;

    return cache_word & (1<<cache_wordpos) ? 1 : 0;
}

25 sc_uint<16> mpgaudparse::read_sample_code( sc_uint<5> num_bits )
{
    sc_uint<16> sample_code = 0; // 16 bit
30     if( cache_wordpos < num_bits )
    {
        num_bits -= cache_wordpos;
        sample_code = get_next_word( cache_wordpos ) << num_bits;
35         refresh_cache();
        sample_code |= get_next_word( num_bits );
    }
    else
        sample_code = get_next_word( num_bits );
40     return sample_code;
}

45 sc_uint<16> mpgaudparse::get_next_word( sc_uint<5> num_bits )
{
    sc_uint<32> mask = (1<<num_bits) - 1;

    cache_wordpos -= num_bits;
50     mask <= cache_wordpos;
    mask &= cache_word;
    mask >= cache_wordpos;
    mask &= 0xFFFF;

55     return mask;
}

```

```

60 void mpgaudparse::refresh_cache( void )
{
    cache_word = cache[cache_rd_pos];

    if( cache_rd_pos == CACHE_SIZE - 1 )
        cache_rd_pos = 0;
65     else
        cache_rd_pos++;

    cache_wordpos = 32;
}

70 void mpgaudparse::fill_cache( void )
{
    unsigned char req_size = 0;

75     if( cache_rd_pos == cache_wr_pos )
        // this only happens on reset
        req_size = CACHE_SIZE - 1;
    else if( cache_rd_pos < cache_wr_pos )
80         req_size = CACHE_SIZE - 1 - cache_wr_pos + cache_rd_pos;
    else /* if( cache_rd_pos > cache_wr_pos ) */
        req_size = cache_rd_pos - cache_wr_pos - 1;

    inREQ.write( req_size );
85     wait_until( inRDY.delayed() == true );

    inREQ.write( 0 );

    for( int i = 0; i <= req_size; i++ )
90     {
        cache[cache_wr_pos] = 0;

        cache[cache_wr_pos] |= (in.read() << 24) & 0xFF000000;
        wait();
95         cache[cache_wr_pos] |= (in.read() << 16) & 0x00FF0000;
        wait();
        cache[cache_wr_pos] |= (in.read() << 8) & 0x0000FF00;
        wait();
100        cache[cache_wr_pos] |= in.read() & 0x000000FF;
        wait();

        if( cache_wr_pos == CACHE_SIZE - 1 )
            cache_wr_pos = 0;
        else
105            cache_wr_pos++;
    }

    wait();
}

110 sc_int<17> mpgaudparse::invq( sc_uint<16> qsample, sc_uint<4> msb,
                             sc_uint<5> index )
{
115     sc_int<33> value;

    value = (qsample & ((1<<msb)-1)) << (16-msb);

120     if( !(qsample & (1<<msb)) )
        value -= 1 << 16;

    value += invq_D[index];
    value *= invq_C[index];
125     value /= ((1<<16)-1);

    return value;
}

130 void mpgaudparse::run( void )
{
    unsigned int framecount;
    unsigned int table_select = 0;
135     bool protection_bit;
    sc_uint<4> bitrate;
    sc_uint<5> max_frequency;
    sc_uint<2> sampling_frequency;

140     sc_uint<5> allocation[SBLIMIT]; // 2..4 bits
    sc_uint<6> scale_factor[SBLIMIT][3];

```

```

        sc_int<17> frame_sample[SBLIMIT][3];

145     framecount = 0;
        cache_rd_pos = 0;
        cache_wr_pos = 0;
        cache_wordpos = 0;

150     out.write( 0 );
        outRDY.write( false );
        inREQ.write( 0 );
        errSYNC.write( false );

155     wait();

        // Frame loop
160     while( 1 )
        {
            errSYNTAX.write( false );

165             fill_cache();

            // We decode MPEG-1 layer II, so start of frame contains
            // 14 one-bits (sync + id + first bit of layer).

170             for( int i = 0; i < 14; i++ )
            {
                if( get_next_bit() == 0 )
                {
                    errSYNC.write( true );
175                     i = -1;
                }

                wait();
            }

180             wait();

            errSYNC.write( false );

185             // Expect a zero-bit next (layer II), otherwise find next frame

            if( get_next_bit() != 0 )
                MPEG_ERROR

190             // protection_bit

            protection_bit = !get_next_bit();

195             // bitrate, we support 32 kbps up to 192 kbps

            bitrate = get_next_bit() << 3;
            bitrate |= get_next_bit() << 2;
            bitrate |= get_next_bit() << 1;
            bitrate |= get_next_bit();

200             if( bitrate == 1 || bitrate == 2 )
                table_select = 1;
            else if( bitrate >= 3 && bitrate <= 10 )
                table_select = 0;
205             else
                MPEG_ERROR

            // sampling frequency, we only support 48 kHz

210             sampling_frequency = get_next_bit() << 1;
            sampling_frequency |= get_next_bit();

            if( sampling_frequency != 1 )
215                 MPEG_ERROR

            outSAMPFREQ.write( sampling_frequency );

            // ignore padding and private

220             for( int i = 0; i < 2; i++ )
                get_next_bit();

            // mode must be mono

225             if( get_next_bit() != 1 || get_next_bit() != 1 )

```



```

    MPEG_ERROR

    // ignore mode extension, copyright and original
230    for( int i = 0; i < 4; i++ )
        get_next_bit();

    // we require "no emphasis"
235    if( get_next_bit() != 0 || get_next_bit() != 0 )
        MPEG_ERROR

    // skip crc (if any)
240    if( protection_bit )
    {
        for( int i = 0; i < 16; i++ )
            get_next_bit();
245        wait();
    }

    // we are now ready for frame data
250    // read bit allocation information

    max_frequency = 0;
    outMAXFREQ.write( 0 );
255    for( int sb = 0; sb < SBLIMIT; sb++ )
    {
        sc_uint<4> alloc = 0;
        switch( quant_alloc_table[table_select][sb][0] )
        {
260            case 4:
                alloc |= get_next_bit() << 3;
            case 3:
                alloc |= get_next_bit() << 2;
            case 2:
265                alloc |= get_next_bit() << 1;
                alloc |= get_next_bit();
        }

        if( alloc == 0 )
270            allocation[sb] = 0;
        else
        {
            allocation[sb] = quant_alloc_table[table_select][sb][alloc];
            max_frequency = sb;
275            outMAXFREQ.write( sb );
        }

        wait();
    }
280    wait();

    cerr << "mpggaudparse:_" << outMAXFREQ.read() << endl;
285    // read scalefactor grouping information

    for( int sb = 0; sb < SBLIMIT; sb++ )
    {
290        if( allocation[sb] != 0 )
        {
            scale_factor[sb][2] = get_next_bit() << 1;
            scale_factor[sb][2] |= get_next_bit();
        }
295        wait();
    }

    wait();
300    // read actual scale factors

    for( int sb = 0; sb < SBLIMIT; sb++ )
305    {
        if( allocation[sb] != 0 )
        {
            scale_factor[sb][0] = read_sample_code( 6 );
310            switch( scale_factor[sb][2] )

```

```

    {
    case 0:
        // (one) (two) (three)
        scale_factor[sb][1] = read_sample_code( 6 );
315         scale_factor[sb][2] = read_sample_code( 6 );
        break;

    case 1:
        // (one two) (three)
        scale_factor[sb][1] = scale_factor[sb][0];
320         scale_factor[sb][2] = read_sample_code( 6 );
        break;

    case 2:
        // (one two three)
        scale_factor[sb][1] = scale_factor[sb][0];
325         scale_factor[sb][2] = scale_factor[sb][0];
        break;

    case 3:
        // (one) (two three)
        scale_factor[sb][1] = read_sample_code( 6 );
        scale_factor[sb][2] = scale_factor[sb][1];
330         break;
    }
335 }

    wait();
}

340 wait();

// output frame sync and frequency info
345 wait_until( outFRAMEREQ.delayed() == true );

    outRDY.write( true );
    out.write( SYNC_PATTERN );
350 wait();
    outRDY.write( false );

// read quantised sample codes, dequantise and scale
355 for( int gr = 0; gr < 12; gr++ )
    {
        for( int sb = 0; sb < SBLIMIT; sb++ )
        {
360             int alloc_index = allocation[sb];
            if( alloc_index != 0 )
            {
                sc_uint<16> qsample;

365                 if( alloc_index < 4 ) // if grouping
                {
                    sc_uint<4> nlevels =
                        quant_grouped_table[alloc_index][0];

370                     sc_uint<10> c =
                        read_sample_code( quant_grouped_table[alloc_index][1] );

                    for( int s = 0; s < 3; s++ ) // unroll
                    { // degroup
375                         qsample = c % nlevels;
                        c /= nlevels;

                        frame_sample[sb][s] =
380                             invq( qsample, alloc_index, alloc_index - 1 );
                    }
                }
            }
            else // no grouping, read three samples
            {
385                 for( int s = 0; s < 3; s++ ) // unroll
                {
                    qsample = read_sample_code( alloc_index - 1 );

                    frame_sample[sb][s] =
390                         invq( qsample, alloc_index - 2, alloc_index - 1 );
                }
            }

// denormalise
for( int s = 0; s < 3; s++ ) // unroll

```

```

395         {
            sc_int<42> product;

            product = frame_sample[sb][s] *
                scale_factor_table[scale_factor[sb][gr>>2]];

400         product >>= 23;

            // if neccessary, clip
            if( product > ((1<<16)-1) )
                product = (1<<16) - 1;
405         else if( product < -(1<<16) )
            product = -(1<<16);

            frame_sample[sb][s] = product;
410         }
        }
    else
        for( int s = 0; s < 3; s++ ) // unroll
            frame_sample[sb][s] = 0;
415    }

    // output samples (to be read by filterbank)

420    for( int s = 0; s < 3; s++ ) // unroll
    {
        wait_until( outSBREQ.delayed() == true );
        outRDY.write( true );

425        for( int sb = 0; sb < SBLIMIT; sb++ )
        {
            out.write( frame_sample[sb][s] );
            wait();
            outRDY.write( false );
430        }

        wait();
    }
}
435 }
}

```

A.2.3 $\Delta\Sigma$ modulator

dsmod.h

```

SC_MODULE( dsmod )
{
    sc_in<bool>      RST;

4    sc_out<bool>    inACK;
    sc_in<bool>      inRDY;
    sc_in<sc_int<17>> in;

9    sc_in<bool>    outACK;
    sc_out<bool>    outRDY;
    sc_out<bool>    out;

    sc_in_clk       CLK;

14

    SC_CTOR( dsmod )
    {
        SC_CTHREAD( run, CLK.pos() );
19        watching( RST.delayed() == true );
    }

    void run( void );
};

```

dsmod.cpp

```

#include <systemc.h>
2  #include "dsmod.h"

void dsmod::run( void )
{
7    sc_int<32> u;

```

```

    sc_int<32> v;
    sc_int<17> x;
    bool y;
    sc_uint<32> o;

12

    v = 0;
    y = false;

17    out.write( y );
    outRDY.write( false );
    inACK.write( false );
    wait();

22    while( 1 )
    {
        // wait until interpolation filter is ready
        // to burst data

27        inACK.write( true );
        wait_until( inRDY.delayed() == true );

32        // read and compute

        o = 0;

        for( unsigned int i = 0; i < 32; i++ )
37        {
            x = in.read();

            if( y )
                u = x - 32767;
42            else
                u = x + 32768;

            v = u + v;

47            y = 0;
            if( v >= 0 )
            {
                y |= 1;
                o |= 1 << i;
52            }

            wait();

57            inACK.write( false );
        }

        // write

62        outRDY.write( true );
        out.write( (o & 1) != 0 );
        wait_until( outACK.delayed() == true );
        outRDY.write( false );

67        for( unsigned int i = 1; i < 32; i++ )
        {
            out.write( (o & (1<<i)) != 0 );
            wait();
72        }
    }
}

```

A.2.4 Reconstruction filter

filterdef.h

```

1 // *****
//
// 750 Hz:
// [b,a] = iirremez( 8, 2, 1, 1/32/8, 0.035+1/32/8 );
// The stopband attenuation is 50.024823
6 //
// 1500 Hz:
// [b,a] = iirremez( 8, 2, 1, 2/32/8, 0.065+2/32/8 );
// The stopband attenuation is 50.495655
//

```

```

11 // 3000 Hz:
// [b,a] = iirremez( 8, 2, 1, 4/32/8, 0.1+4/32/8 );
// The stopband attenuation is 50.088281
//
// 6000 Hz:
16 // [b,a] = iirremez( 8, 2, 1, 8/32/8, 0.145+8/32/8 );
// The stopband attenuation is 50.569357
//
// 12000 Hz:
// [b,a] = iirremez( 8, 2, 1, 16/32/8, 0.185+16/32/8 );
21 // The stopband attenuation is 50.706724
//
// 20250 Hz: (because sbmax is 27 for the streams we decode)
// [b,a] = iirremez( 8, 2, 1, 27/32/8, 0.2+27/32/8 );
// The stopband attenuation is 50.530362
26 //
// *****

31 const double a[6][N] =
{
{ 1.00000000000000e+00, -1.98828051035226e+00, 9.88423710082639e-01 },
{ 1.00000000000000e+00, -1.97307964507895e+00, 9.73733396241935e-01 },
{ 1.00000000000000e+00, -1.94581629875807e+00, 9.48410232837472e-01 },
36 { 1.00000000000000e+00, -1.89029384083126e+00, 9.00547778618750e-01 },
{ 1.00000000000000e+00, -1.77876514486171e+00, 8.18906519700735e-01 },
{ 1.00000000000000e+00, -1.63233798856964e+00, 7.43344054934946e-01 }
};

41 const double b[6][M] =
{
{ 1.79096577699688e-03, -3.10533649328450e-03, 1.37599093960560e-03,
4.12406373687460e-06, 3.78330794301216e-06, 3.97629867362051e-06,
46 1.37595758093198e-03, -3.10515271483066e-03, 1.79088048886445e-03 },

{ 1.92657804346049e-03, -2.82507355539670e-03, 1.15881235059790e-03,
3.21418411545018e-05, 3.22765521365358e-05, 3.21246037557058e-05,
1.15878541806046e-03, -2.82501166268633e-03, 1.92654719105031e-03 },
51 { 2.42218838018840e-03, -2.62579609088878e-03, 1.14170817229617e-03,
1.89949528960408e-04, 1.92579998132317e-04, 1.90139029412703e-04,
1.14147563452980e-03, -2.62557242308833e-03, 2.42207165370041e-03 },

{ 3.09437959409010e-03, -1.43369842877068e-03, 1.59416188984010e-03,
1.04649196754133e-03, 1.07770452224338e-03, 1.04659867717741e-03,
56 1.59393671497599e-03, -1.43348986709300e-03, 3.09426200330691e-03 },

{ 4.59313739737402e-03, 1.68310066593505e-03, 4.78048733019410e-03,
5.16803093221080e-03, 5.44537485246289e-03, 5.16812040045895e-03,
61 4.78020008924796e-03, 1.68329093110759e-03, 4.59292409775701e-03 },

{ 6.82953391841229e-03, 7.50600047854121e-03, 1.31862384813037e-02,
1.61816211866498e-02, 1.73914693545687e-02, 1.61816889084217e-02,
66 1.31855252324219e-02, 7.50611033116081e-03, 6.82896777694246e-03 }
};

```

recon.h

```

#define M 9
#define N 3

3 SC_MODULE( recon )
{
    sc_in<sc_uint<5> > inFREQPROG;
    sc_in<bool> inCLK;
    sc_in<bool> inRST;
    sc_in<bool> in;

    sc_out<float> out;

13 unsigned int select;
    unsigned int freq;
    unsigned int t;
    bool dynamic;

18 double x[M];
    double y[N];

23 SC_CTOR( recon )

```

```

        {
            SC_METHOD( entry );
            sensitive << inCLK.pos() << inRST.pos();
        }
28
    void entry( void );
    void set_dynamic( void );
    void set_static( void );
};

```

recon.cpp

```

#include <systemc.h>

3  #include "recon.h"
   #include "filterdef.h"

#define XIDX(i) ((i+t)%M)
8  #define YIDX(i) ((i+t)%N)

#define X(i)  x[XIDX(i)]
#define Y(i)  y[YIDX(i)]

13
void recon::set_static( void )
{
    dynamic = false;
}

18
void recon::set_dynamic( void )
{
    dynamic = true;
}

23
void recon::entry( void )
{
    if( inRST.read() )
    {
28        select = 5;
        freq = 0;
        t = 0;

        for( int i = 0; i < N; i++ )
33            y[i] = 0;

        for( int i = 0; i < M; i++ )
            x[i] = 0;

38        return;
    }

    // update time, then input

43    t++;
    if( t == 9 )
        t = 0;

48    if( in.read() )
        X(0) = 1.0;
    else
        X(0) = -1.0;

53    if( dynamic && inFREQPROG.read() != freq )
    {
        freq = inFREQPROG.read();

        if( freq <= 0 )
58            select = 0;
        else if( freq <= 1 )
            select = 1;
        else if( freq <= 3 )
            select = 2;
63        else if( freq <= 7 )
            select = 3;
        else if( freq <= 15 )
            select = 4;
        else
68            select = 5;
    }
}

```

```
Y(0) = 0;

73 // compute new output
    for( int k = 0; k < M; k++ )
        Y(0) += b[select][k] * X(M-k);
78
    for( int k = 1; k < N; k++ )
        Y(0) -= a[select][k] * Y(N-k);

83 Y(0) /= a[select][0];

    out.write( Y(0) );
}
```

(a) magnitude response, cutoff at 20250 Hz

(b) phase response, cutoff at 20250 Hz

(c) magnitude response, cutoff at 1500 Hz

(d) phase response, cutoff at 1500 Hz

Figure A.1 Frequency response of two reconstruction filters

bibliography

- [1] Joarez B. Monteiro, Antonia Petaglia, and Carlos A. Leme. Capacitor Charge Control Technique Applied to Digitally Programmable IIR Switched-Capacitor Filter. In *15th Symposium on Integrated Circuits and Systems Design*, pages 68–73. IEEE Computer Society, September 2002.
- [2] Chris Dunn and Mark Sandler. Adaptive sigma-delta modulation for use in DACs. *Electronics Letters*, 32(10):867–868, May 1996.
- [3] M. C. Ramesh and K. S. Chao. Sigma Delta Analog to Digital Converters with Adaptive Quantization. In *Proceedings of the 40th Midwest Symposium on Circuits and Systems*, 1997, volume 1, pages 22–25. IEEE, August 1997.
- [4] Clemens M. Zierhofer. Adaptive Sigma-Delta Modulation With One-Bit Quantization. *IEEE Transactions on Circuits and Systems—II: Analog and Digital Signal Processing*, 47(5):408–415, May 2000.
- [5] Mansour A. Aldajani and Ali H. Sayed. Stability and Performance Analysis of an Adaptive Sigma-Delta Modulator. *IEEE Transactions on Circuits and Systems—II: Analog and Digital Signal Processing*, 48(3):233–243, March 2001.
- [6] Steven R. Norsworthy. Quantization Errors and Dithering in $\Delta\Sigma$ Modulators. In Steven R. Norsworthy, Richard Schreier, and Gabor C. Temes, editors, *Delta-Sigma Data Converters: Theory, Design, and Simulation*, chapter 3, pages 75–140. IEEE Press, 1997.
- [7] Jouko Vankka, Jonne Lindeberg, and Kari Halonen. A Direct Digital Synthesizer with Tunable Delta Sigma Modulator. *Analog Integrated Circuits and Signal Processing*, 38(1):7–15, January 2004.
- [8] J. C. Candy. An Overview of Basic Concepts. In Steven R. Norsworthy, Richard Schreier, and Gabor C. Temes, editors, *Delta-Sigma Data Converters: Theory, Design, and Simulation*, chapter 1, pages 1–43. IEEE Press, 1997.

- [9] Sanjit K. Mitra. *Digital Signal Processing*. McGraw-Hill, second edition, 2001.
- [10] Yrjö Neuvo, Dong Cheng-Yu, and Sanjit K. Mitra. Interpolated Finite Impulse Response Filters. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 32(3):536–570, June 1984.
- [11] Brian P. Brandt and Bruce A. Wooley. A Low-Power, Area-Efficient Digital Filter for Decimation and Interpolation. *IEEE Journal of Solid-State Circuits*, 29(6):679–687, June 1994.
- [12] Eugene B. Hogenauer. An Economical Class of Digital Filters for Decimation and Interpolation. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 29(2):155–162, 1981.
- [13] Robert W. Adams and Richard Schreier. Stability Theory for $\Delta\Sigma$ Modulators. In Steven R. Norsworthy, Richard Schreier, and Gabor C. Temes, editors, *Delta-Sigma Data Converters: Theory, Design, and Simulation*, chapter 4, pages 141–164. IEEE Press, 1997.
- [14] Pervez M. Aziz, Henrik V. Sorensen, and Jan van der Spiegel. An Overview of Sigma-Delta Converters. *IEEE Signal Processing Magazine*, 13(1):61–84, January 1996.
- [15] Mücahit Kozak and İzzet Kale. *Oversampled Delta-Sigma Modulators*. Kluwer Academic Publishers, 2003.
- [16] Anekal B. Sripad and Donald L. Snyder. A Necessary and Sufficient Condition for Quantization Errors to be Uniform and White. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 25(5):442–448, October 1977.
- [17] Monson H. Hayes. *Statistical Digital Signal Processing and Modeling*. John Wiley & Sons, Inc., 1996.
- [18] Robert M. Gray. Spectral Analysis of Quantization Noise in a Single-Loop Sigma-Delta Modulator with DC Input. *IEEE Transactions on Communications*, 37(6):588–599, June 1989.
- [19] Robert M. Gray, Wu Chou, and Ping W. Wong. Quantization Noise in Single-Loop Sigma-Delta Modulation with Sinusoidal Inputs. *IEEE Transactions on Communications*, 37(9):956–968, September 1989.
- [20] Richard Schreier. On the Use of Chaos to Reduce Idle-Channel Tones in Delta-Sigma Modulators. *IEEE Transactions on Circuits and Systems—I: Fundamental Theory and Applications*, 41(8):539–547, August 1994.

- [21] Chris Dunn and Mark Sandler. Linearising Sigma-Delta Modulators using Dither and Chaos. In *1995 IEEE International Symposium on Circuits and Systems*, volume 1, pages 625–628. IEEE, 1995.
- [22] J. B. Hughes and A. Worapishet. Switched Capacitors or Switched Current—Which Will Succeed? In Chris Toumazou, George S. Moschytz, and Barrie Gilbert, editors, *Trade-Offs in Analog Circuit Design*, chapter 11. Springer, first edition, October 2002.
- [23] Roubik Gregorian and Gabor C. Temes. *Analog MOS Integrated Circuits for Signal Processing*. Wiley, 1986.
- [24] Christian C. Enz and Gabor C. Temes. Circuit Techniques for Reducing the Effects of Op-Amp Imperfections. *Proceedings of the IEEE*, 84(11):1584–1614, November 1996.
- [25] Alan Hastings. *The Art of Analog Layout*. Prentice Hall, 2001.
- [26] Paul J. Hurst and James E. C. Brown. Finite Impulse Response Switched-Capacitor Decimation Filters for the DSM D/A Interface. In *IEEE International Symposium on Circuits and Systems*, volume 3, pages 1688–1691. IEEE, May 1989.
- [27] Mikael Gustavsson and Nianxiong Nick Tan. Decimating SC filter for high speed Sigma-Delta D/A converters. In *Proceedings of the 2004 International Symposium on Circuits and Systems*, volume 1, pages 657–660. IEEE, May 2004.
- [28] A. Petraglia, J.S. Pereira, and F.A.P. Barúqui. Low-sensitivity IIR switched-capacitor decimation filters using delay lines with staggered T/H stages. *IEE Proceedings – Circuits, Devices and Systems*, 153(2):193–198, April 2006.
- [29] Un-Ku Moon. CMOS High-Frequency Switched-Capacitor Filters for Telecommunication Applications. *IEEE Journal of Solid-State Circuits*, 35(2):212–220, February 2000.
- [30] Franklin F. Kuo. *Network Analysis and Synthesis*, chapter 3. John Wiley & Sons, Inc., second edition, 1966.
- [31] W. Kenneth Jenkins. Fourier Series, Fourier Transforms, and the DFT. In Wai-Kai Chen, editor, *The Circuits and Filters Handbook*, volume 28 of *Electrical Engineering Handbook*, chapter 4. CRC Press, second edition, 2003.
- [32] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965.

- [33] Frederic J. Harris. On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform. *Proceedings of the IEEE*, 66(1):51–83, January 1978.
- [34] Magdy T. Hanna. Windows with Rapidly Decaying Sidelobes and Steerable Sidelobe Dips. *IEEE Transactions on Signal Processing*, 42(5):2037–2044, August 1994.
- [35] P. P. Vaidyanathan and Igor Djokovic. Wavelet Transforms. In Wai-Kai Chen, editor, *The Circuits and Filters Handbook*, volume 28 of *Electrical Engineering Handbook*, chapter 6. CRC Press, second edition, 2003.
- [36] Gilbert Strang and Truong Nguyen. *Wavelets and Filter Banks*. Wellesley-Cambridge Press, 1996.
- [37] Paul S. Addison. Wavelet transforms and the ECG: a review. *Physiological Measurement*, 26(5):R155–R199, October 2005.
- [38] Michael L. Hilton. Wavelet and Wavelet Packet Compression of Electrocardiograms. *IEEE Transactions on Biomedical Engineering*, 44(5):394–402, May 1997.
- [39] Jerome M. Shapiro. Embedded Image Coding Using Zerotrees of Wavelet Coefficients. *IEEE Transactions on Signal Processing*, 41(12):3445–3462, December 1993.
- [40] Peter Noll. MPEG Digital Audio Coding. *IEEE Signal Processing Magazine*, September 1997.
- [41] International Organization for Standardization. *ISO/IEC 11172-3:1993 Information technology—Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s—Part 3: Audio*. International Organization for Standardization, 1998.
- [42] International Organization for Standardization. *ISO/IEC 13818-3:1998 Information technology—Generic coding of moving pictures and associated audio information—Part 3: Audio*. International Organization for Standardization, 1998.
- [43] European Broadcasting Union. *ETS 300 401 Radio broadcasting systems; Digital Audio Broadcasting (DAB) to mobile, portable and fixed receivers*, chapter 7. European Telecommunications Standards Institute, second edition, May 1997.
- [44] John P. Princen and Alan Bernard Bradley. Analysis/Synthesis Filter Bank Design Based on Time Domain Aliasing Cancellation. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 34(5):1153–1161, October 1986.

- [45] Patrick De Smet, Filip Rooms, Hiệp Quang Luong, and Wilfried Philips. Do not zero-pute: an efficient homespun MPEG-audio layer II decoding and optimization strategy. In *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, pages 376–379. ACM Press, 2004.
- [46] George Tzanetakis and Perry Cook. Sound analysis using MPEG compressed audio. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 761–764. IEEE, June 2000.
- [47] Yasuyuki Nakajima, Yang Lu, Masaru Sugano, Akio Yoneyama, Hiromasa Yangihara, and Akira Kurematsu. A fast audio classification from MPEG coded data. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 6, pages 3005–3008. IEEE, March 1999.
- [48] Roman Jarina, Noel O'Connor, Noel Murphy, and Seán Marlow. An Experiment in Audio Classification from Compressed Data. In *IWSSIP'04—International Workshop on Systems, Signals and Image Processing*, Poznan, Poland, September 2004.
- [49] Mike Rebeschini and Paul F. Ferguson Jr. Analog Circuit Design for $\Delta\Sigma$ DACs. In Steven R. Norsworthy, Richard Schreier, and Gabor C. Temes, editors, *Delta-Sigma Data Converters: Theory, Design, and Simulation*, chapter 12, pages 380–405. IEEE Press, 1997.
- [50] Thorsten Grötter, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, May 2002.
- [51] Alain Vachoux, Christoph Grimm, and Karsten Einwich. Towards analog and mixed-signal soc design with systemc-ams. In *Proceedings of the Second IEEE International Workshop on Electronic Design, Test and Applications*, pages 97–102. IEEE Computer Society, January 2004.
- [52] Johnny Bjørnsen, Thomas E. Bonnerud, and Trond Ytterdal. Behavioral Modeling and Simulation of Mixed-Signal System-on-a-Chip using SystemC. *Analog Integrated Circuits and Signal Processing*, 34(1):25–38, January 2003.
- [53] Synopsys. *CoCentric™ SystemC Compiler Behavioural Modeling Guide*. Synopsys, March 2001.
- [54] Patrick De Smet and Ignace Bruyland. Optimized Recursive Subband Synthesis Windowing for Implementing Efficient MPEG Audio Decoders. *IEEE Signal Processing Letters*, 10(10):303–306, October 2003.
- [55] Konstantinos Konstantinides. Fast Subband Filtering in MPEG Audio Coding. *IEEE Signal Processing Letters*, 1(2):26–28, February 1994.

- [56] Leland B. Jackson. An Improved Martinez/Parks Algorithm for IIR Design with Unequal Numbers of Poles and Zeros. *IEEE Transactions on Signal Processing*, 42(5):1234–1238, May 1994.
- [57] Maxim Integrated Products. Dynamic Testing of High-Speed ADCs, Part 2. Application Note 729, January 2001. http://www.maxim-ic.com/appnotes.cfm/appnote_number/729.