TÉCNICO LISBOA

Asynchronous Audio Sample Rate Converter

Pedro Miguel Portela Teixeira

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor(s): Prof. José João Henriques Teixeira de Sousa

Examination Committee

Chairperson: Prof. Francisco André Corrêa Alegria
Supervisor: Prof. José João Henriques Teixeira de Sousa
Member of the Committee: Prof. Gonçalo Nuno Gomes Tavares

September 2021

Declaration

I declare that this document is an original work of my own authorship and that it fullfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

This thesis is the result of two years of work and five years of studies in Instituto Superior Técnico. As such, I would like to thank those who helped me in this journey.

From the two years of development of the thesis, I thank Pedro Miranda and João Lopes for helping me testing the work performed. I also thank Valter Mario for helping me with the initial development of the core during my internship at IObundle. Special thanks to my advisor, Professor José Teixeira de Sousa, for proposing the subject of the thesis to me, reviewing it, and keeping me productive, even while stressing due to the pandemic.

From the five years of studies in Instituto Superior Técnico, I want to acknowledge my colleagues who helped me with my studies and ensured my success, as well as lifting my spirits with engineering jokes: Daniel Pestana, Renato Dias, José Gomes, Gil Serrano, João Pinto, Miguel Cardoso, Afonso Luis, Simão Eusébio and Thomas Berry.

Lastly, I want to give special thanks to my parents, Carlos Teixeira and Sónia Teixeira, as well as my brother, João Teixeira, for giving me support, allowing me to achieve my objectives.

Resumo

Esta dissertação propõe o projeto de um novo conversor de frequências de amostragem assíncrono multi-canal de 24 bits ou menos, com os seguintes objetivos: (1) taxa de distorção harmónica e ruído total (THD+N) de $-130\,\mathrm{dB}$ ou menos, (2) razões de conversão suportadas entre 1:24 e 24:1; centenas de canais de audio; (3) tempo de sincronização inferior a $200\,\mathrm{ms}$, (4) variação da fase entre entrada e saída inferior a uma amostra de saída após o reset, e (5) consumo de recursos por canal semelhante ao CWda52, ou menor.

Todos os objetivos foram cumpridos, e centenas de testes foram aplicados ao novo projeto, que obtém uma THD+N de um mínimo de $-143\,\mathrm{dB}$ até um máximo de $-136\,\mathrm{dB}$ para um total de 121 testes, com uma média de aproximadamente $-140\,\mathrm{dB}$. O novo projeto suporta razões de conversão de 1:24 até 24:1. A melhor solução alternativa é o chip SRC4194 da Texas Instruments, que suporta entre 1:16 e 16:1. O projeto também suporta dezenas ou centenas de canais, utilizando multiplexagem por divisão de tempo; as outras soluções tipicamente suportam oito canais, e o CWda52 precisa de replicar a sua unidade stereo para suportar mais canais. Consequentemente, a utilização de recursos por canal é extremamente competitiva. O tempo de sincronização é $20\,\mathrm{ms}$, que é uma ordem de grandeza inferior a qualquer outra alternativa. A variação da fase após reset, uma característica importante omitida em soluções alternativas, é inferior a uma amostra de saída.

Palavras-chave: Conversor de Frequências de Amostragem, Processamento Digital de Sinal, FPGA, Filtro Digital

Abstract

This thesis proposes a new design of a new 24-bit or less multi-channel Asynchronous Sample Rate Converter, with the following specification: (1) total harmonic distortion plus noise ratio (THD+N) of $-130\,\mathrm{dB}$ or less; (2) supported conversion ratios from 1:24 to 24:1; hundreds of audio channels; (3) synchronisation time of less than $200\,\mathrm{ms}$; (4) variation of the phase between input and output of less than one output sample after a reset; (5) resource consumption per channel similar to the CWda52, or less.

All objectives have been met, and hundreds of tests have been applied to the new design, which achieves a THD+N from a minimum of $-143\,\mathrm{dB}$ to a maximum of $-136\,\mathrm{dB}$ for a total of 121 tests, with an average of approximately $-140\,\mathrm{dB}$. The new design supports conversion ratios range from 1:24 to 24:1. The best alternative solution is the Texas Instruments SRC4194 chip, which supports a range from to 1:16 to 16:1. It can support tens or hundreds of channels using time division multiplexing; the other solutions typically support eight channels, and the CWda52, the main competitor to this approach, needs to replicate its stereo unit to handle more channels. Consequently, the resource usage per channel of this approach is extremely competitive. The synchronisation time is $20\,\mathrm{ms}$, which is one order of magnitude lower than any other alternative. The variation of the phase after reset, an important characteristic omitted in the alternative solutions, is less than one output sample.

Keywords:Asynchronous Sample Rate Converter, Digital Signal Processing, FPGA, Digital Filter

Contents

Acknowledgments v
Resumo vii
Abstract ix
List of Tables xv
List of Figures xvii
List of Acronyms xix

1 Introduction **1**
1.1 Topic Overview 1
1.2 Motivation 2
1.3 Objectives 2
1.4 Thesis Outline 3

2 Asynchronous Sample Rate Converter Theory **5**
2.1 Sample Rate Converter's Structure 5
2.2 Interpolation/Decimation Filter As A Fractional Delay Filter 7

3 Previous Work **11**
3.1 Implementation of the Interpolation/Decimation Filter 11
3.1.1 Direct Filtering Using a Multiply-Accumulate Unit (MACC) 11
3.1.2 Cascaded Integrator Comb Filters (CIC) 12
3.1.3 Approximation By Piece-wise Quadratic Function 13
3.1.4 Farrow Structure 15
3.2 Implementation Of The Sample Rate Conversion Ratio Estimator 16
3.2.1 Period Measurement And Averaging 17
3.2.2 Digital Phase Locked Loop As Frequency Tracker 18

4 Proposed Design **21**
4.1 Data Memory 23
4.2 Ratio Estimator 24
4.2.1 Initial Approximation - Period Measurement and Averaging 25
4.2.2 Frequency Tracker 26

4.2.3 Ratio Estimator Control Unit . . . 28
4.3 Resampler . . . 32
4.3.1 Address Generator . . . 32
4.3.1.1 Filter Setup . . . 33
4.3.1.2 Input Sample Address Generator . . . 34
4.3.1.3 Filter Coefficients Address Generator . . . 35
4.3.2 Coefficient Memory . . . 36
4.3.3 Multiply-Accumulator . . . 37
4.4 Pipeline Registers . . . 37

5 Testing System **39**
5.1 Signal Generation and Analysis . . . 39
5.1.1 Parameter Adjustment . . . 39
5.1.2 Filter Coefficients Generation . . . 41
5.1.3 Input Signal Generation . . . 42
5.1.4 Output Signal Analysis . . . 43
5.1.5 Effect Of Resetting On The Group Delay . . . 45
5.2 IOB-SoC Hardware Implementation . . . 46
5.2.1 Peripherals . . . 47
5.2.1.1 UART . . . 48
5.2.1.2 Ethernet . . . 48
5.2.1.3 ASRC's Wrapper . . . 49
5.2.2 Firmware . . . 53

6 Results **55**
6.1 Fast Fourier Transform Setup . . . 55
6.2 Total Harmonic Distortion Plus Noise . . . 58
6.3 Frequency Response . . . 61
6.4 Phase Difference After Reset . . . 61
6.5 Linearity Of The ASRC . . . 63
6.6 FPGA Resource Usage . . . 64
6.7 ASIC Resource Usage . . . 66
6.8 Multi-Channel Support and Limitations . . . 67

7 Conclusions **69**
7.1 Achievements . . . 70
7.2 Future Work . . . 71

Bibliography **73**

A Full Test Results **A.1**

A.1 THD+N A.1

A.2 Group Delay After Reset A.3

A.3 THD+N With Varying Input Frequency A.6

A.4 Frequency Response A.8

A.5 THD+N With Varying Input Magnitude A.10

A.6 Magnitude With Varying Input Magnitude A.12

List of Tables

4.1 Interface signals. 22
4.2 Synthesis Parameters 22
4.3 Conversion ratio estimator FSM output signals per state. 31

5.1 IOB-SoC memory map 47
5.2 CPU native slave interface signals 48
5.3 ASRC testing system's synthesis parameters. 50
5.4 Software accessible registers for the ASRC. 51
5.5 Software accessible registers for the input and output buffers. 52
5.6 Software accessible registers for the DMA. 52

6.1 Commonly used sample rates in audio applications and their particular uses 58
6.2 Total harmonic distortion+noise ratio for some conversions 58
6.3 Group delay of some conversions. 62
6.4 Results of linear regression of some conversions 64
6.5 Resource utilization on a XCKU040 (hierarchical representation) 65
6.6 Resource utilization on a XC7A35 (left) and Cyclone V GT (right) 65
6.7 ASIC resource usage for the ASRC 66
6.8 ASIC resource usage for the CWda52 with two audio channels 66

A.1 Total harmonic distortion+noise ratio for every tested conversion A.3
A.2 Group delay for every tested conversion A.5

List of Figures

2.1 Analog interpretation of a sample rate converter . . . 5
2.2 Classic digital sample rate conversion algorithm by factor L/M . . . 6
2.3 Graphical representation of $h[n]$ for $\tau_d = 0$ . . . 8
2.4 Illustration of upsampling (3:4) used to determine the output sample at n = 4.5 . . . 9
2.5 Illustration of downsampling (4:3) used to determine the output sample at n = 8 . . . 10

3.1 MACC filter block diagram. . . . 12
3.2 Example of cascaded integrator comb filter structure, used as an interpolation filter . . . 13
3.3 Example of a piece-wise kernel filter structure . . . 14
3.4 Example of optimized kernel filter structure . . . 15
3.5 Example of farrow structure, optimized for variable fractional delay . . . 16
3.6 Period measurement and ρ computation using a PLL. . . . 17
3.7 Period measurement and ρ computation using a divider. . . . 17
3.8 DPLL phase detector block diagram. . . . 19
3.9 DPLL loop filter and VCO block diagram. . . . 19

4.1 Symbol and interface diagram. . . . 21
4.2 Asynchronous sample rate converter top block diagram. . . . 23
4.3 Data memory module block diagram. . . . 24
4.4 Ratio Estimator module block diagram. . . . 25
4.5 Frequency Tracker module block diagram. . . . 27
4.6 Ratio Estimator FSM state diagram. . . . 30
4.7 Resampler block diagram. . . . 32
4.8 Address generator block diagram. . . . 33
4.9 Output sample address, h_step and α computation submodule block diagram. . . . 34
4.10 Input sample address computation submodule block diagram. . . . 35
4.11 Coefficient address computation submodule block diagram. . . . 36
4.12 Coefficient memory submodule block diagram. . . . 36
4.13 MACC block diagram. . . . 37

5.1 IOB-SoC block diagram. . . . 46
5.2 Ethernet data frame format. . . . 48

5.3 ASRC's wrapper block diagram. 50
5.4 ASRC testing firmware flowchart. 54

6.1 Fast-Fourier transform of upsampled signals. 56
6.2 Fast-Fourier transform of downsampled signals. 57
6.3 THD+N of the ASRC's output for fixed conversions and varying input frequency. 59
6.4 THD+N of the ASRC output for fixed conversions while varying the input magnitude. . . . 60
6.5 Frequency response of the ASRC for a few conversions. 61
6.6 Output signal before (blue) and after (red) reset for a conversion from 48 kHz to 32 kHz. . . 62
6.7 Magnitude of the ASRC's output for fixed conversions and variable input magnitude. . . . 63

A.1 THD+N of the ASRC's output for fixed conversions and varying input frequency (Full results - 1/2). A.6
A.2 THD+N of the ASRC's output for fixed conversions and varying input frequency (Full results - 2/2). A.7
A.3 Frequency response (Full results - 1/2). A.8
A.4 Frequency response (Full results - 2/2). A.9
A.5 THD+N of the ASRC's output for fixed conversions and varying input magnitude (Full results - 1/2). A.10
A.6 THD+N of the ASRC's output for fixed conversions and varying input magnitude (Full results - 2/2). A.11
A.7 Magnitude of the ASRC's output for fixed conversions and varying input magnitude (Full results - 1/2). A.12
A.8 Magnitude of the ASRC's output for fixed conversions and varying input magnitude (Full results - 2/2). A.13

List of Acronyms

AES Audio Engineering Society

ASIC Application Specific Integrated Circuits

ASRC Asynchronous Sample Rate Converter

CDC Clock Domain Crossing

CIC Cascaded Integrator Comb

CPU Central Processing Unit

CRC Cyclic Redundancy Check

DDR Double Data Rate

DMA Direct Memory Access

DPLL Digital Phase Locked Loop

DSP Digital Signal Processor

FIFO First In, First Out

FIR Finite Impulse Response

FPGA Field Programmable Gate Array

FSM Finite State Machine

HDL Hardware Description Language

IC Integrated Circuit

IP Intellectual Property

LPF Low Pass Filter

LUT Lookup Table

MACC Multiply-Accumulate

MIG Memory Interface Generator

MMCM Mixed-Mode Clock Manager

PLL Phase Locked Loop

RAM Random Access Memory

ROM Read-Only Memory

RTL Register Transfer Level

SFD Start Frame Delimiter

SRAM Static Random Access Memory

SSRC Synchronous Sample Rate Converter

SoC System-on-a-Chip

TDM Time Division Multiplexed

THDN Total Harmonic Distortion Plus Noise

UART Universal Asynchronous Receiver-Transmitter

USB Universal Serial Bus

VCO Voltage Controlled Oscillator

Chapter 1

Introduction

1.1 Topic Overview

In 1977, with the growing popularity of audio interfaces, the Audio Engineering Society (AES) founded the AES Digital Audio Standards Committee, creating some of the most used audio standards to this day. One of the most popular standards, the AES3 digital audio interface is still used in current audio equipment, like microphones and speakers with XLR connectors.

Since then, there has been a significant rise in AES standards, many of them demanding the conversion of an audio signal's sample rate. One of the classic examples is the conversion from CD quality with a sampling rate $F_s = 44.1\,\text{kHz}$ to DVD quality with $F_s = 48\,\text{kHz}$. Consequently, there is a great demand for sample rate converters, both in software algorithms and hardware designs.

To answer this need, some integrated circuit manufacturers developed multiple sample rate converters with varied specifications. The AD1896 [1], for instance, is an asynchronous sample rate converter made by Analog Devices, in 2003. It supports a stereo (2 channel) audio signal and converts its sampling rate from 7.75:1 to 1:8 ratios, with a Total Harmonic Distortion Plus Noise (THD+N) ratio of around $-125\,\text{dB}$. Another example is the SRC4194 [2], manufactured by Texas Instruments since 2004, supporting up to 4 channel inputs and a wider sampling rate ratio, ranging from 16:1 to 1:16 ratios, with a THD+N of around $-140\,\text{dB}$.

The implementation as Intellectual Property (IP) is quintessential for the development of embedded audio systems, as it allows one to synthesise a new core with different configurations, fulfilling different specifications, in a faster and cheaper way. Furthermore, one can implement the IP directly into the system, without the need to develop a new core specifically for the system or to externally connect a discrete core to the system.

The first IP (IP) was developed by Coreworks, an IP design company, and called the CWda52 [3]. It supports up to 8 channels and converts sampling rates between $8\,\text{kHz}$ and $192\,\text{kHz}$, with ratios from 7:1 to 1:7. The THD+N of the converted signal is around $-120\,\text{dB}$. The core uses around 700 Slices when implemented on a *Xilinx Kintex-7* Field Programmable Gate Array (FPGA) board as a stereo converter. This IP has some limitations that can be improved: Firstly, the configurable nature of the IP should allow

it to support more channels at the cost of more resources if needed. The THD+N of the output can be further reduced to make the core more competitive with its Integrated Circuit (IC) counterparts. The same reason justifies the improvement of the limit imposed on the ratios. Finally, as the core replicates itself for each pair of channels, the resource usage per channel can be further reduced.

This work started in a summer internship at IObundle Lda, another IP design company, and has been carried out in a double academic and industrial context since then. At the beginning of the work, a preliminary document describing the ASRC, an Octave model and a non-functional Verilog were available. These materials made it possible to learn the sample rate conversion algorithm and have a starting point.

The resampler was fixed, and its THD+N was reduced to near the theoretical limit of -146dB for 24-bit audio. The resampler was also used to develop a Synchronous Sample Rate Converter (SSRC) core in parallel and outside the scope of this work. Then the conversion ratio estimator module was developed from scratch, which involves very accurate digital signal processing in a 3-clock domain implementation, requiring synchroniser circuits and a comprehensive test environment.

1.2 Motivation

While there are currently hardware implementations of asynchronous sample rate converters, most of them are implemented as ICs, while IP implementations are few and have limited specifications compared to their IC counterparts. This context makes it difficult for companies that manufacture multi-media devices to integrate a sample rate converter IP.

The overall growth of the market for embedded audio systems, with many requiring sample rate conversion, motivates the development of a better sample rate converter IP core. The only existing IP implementation of an ASRC, the CWda52, shows several limitations, so it makes sense to develop an improved IP core, competing with both the existing IP core and the IC counterparts.

1.3 Objectives

The main objective of this work is to design an asynchronous sample rate converter (ASRC) IP core using the Verilog hardware description language. The core should meet the following specifications:

- Support for up to 24-bit samples.
- Conversion from and to any sample rate, in the range between $8\,\text{kHz}$ and $192\,\text{kHz}$.
- Capability of converting multiple channels, limited by the operation clock frequencies.
- Output THD+N equal or lower than $-130\,\text{dB}$ (for 24-bit samples).
- Synchronization time equal or lower than $200\,\text{ms}$.
- Variation of phase between input and output after a reset of less than one output sample.

- Resource consumption per channel of the same order as the CWda52, or lower.

1.4 Thesis Outline

This document is composed of 6 more chapters. The second chapter explains the sample rate conversion algorithm. The third chapter presents some sample rate converter designs, along with their advantages and disadvantages. In the fourth chapter, the hardware design of the core is described, including architecture, interfaces and sub-modules. The fourth chapter presents the process of testing and integrating the ASRC into a system-on-a-chip (SoC). The fifth chapter shows and analyses the results obtained from the tests performed. In the sixth and final chapter, some conclusions of the work performed are drawn, as the final product is compared to its IC counterparts, and some possible future work is proposed.

Chapter 2

Asynchronous Sample Rate Converter Theory

An ideal Asynchronous Sample Rate Converter (ASRC) is able to convert the sample rate of an input audio signal to a desired output sample rate without any loss of signal quality. As opposed to a synchronous sample rate converter, the ASRC also needs to measure the value of the input and output sample rates continuously to compute the conversion ratio. The ideal ASRC converts a discrete time input signal $x[n]$, sampled at a rate F_{s1}, to a continuous time signal $x(t)$, using a reconstruction filter. The signal is then filtered by an anti-aliasing filter which ensures that its output $y(t)$ has no components which would violate Nyquist's law. Signal $y(t)$ is then converted to a discrete time signal $y[m]$, sampled at the desired output rate F_{s2} [4].

A block diagram of this model is shown in Fig. 2.1. Note that the reconstruction filter and anti-aliasing filters can be combined in a single Low Pass Filter (LPF).

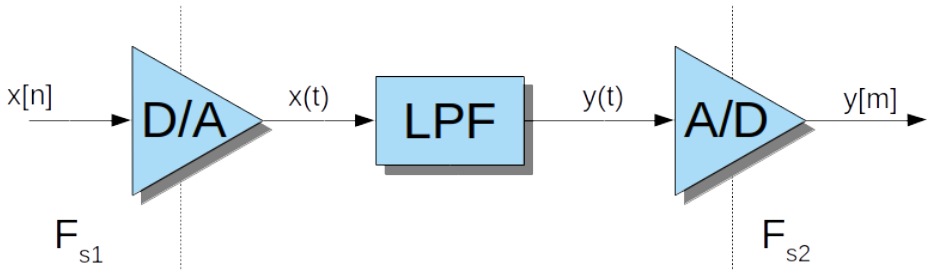


Figure 2.1: Analog interpretation of a sample rate converter

2.1 Sample Rate Converter's Structure

One of the challenges of creating a purely digital solution is the design of the LPF digital filter, which must be both accurate and efficient. The classical approach to the problem consists in upsampling the

signal by a factor L (interpolation), doing the processing at the frequency $L \times F_{s1}$, and downsampling the result by a factor M (decimation). This is a means to emulate a discrete to continuous and continuous to discrete signal conversion [5]. A simple block design of the algorithm is shown in Fig. 2.2.

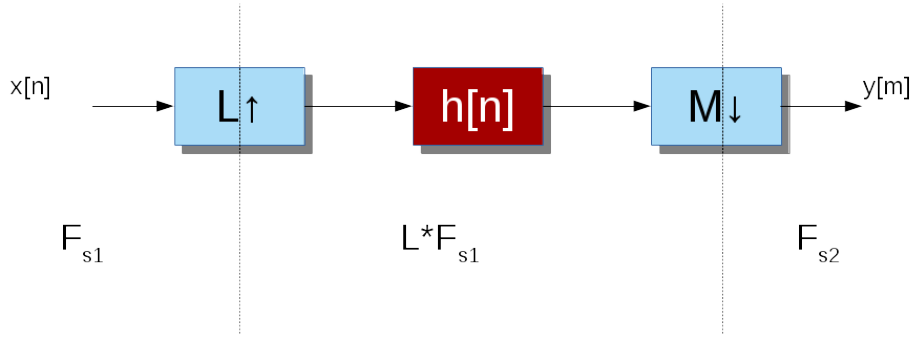


Figure 2.2: Classic digital sample rate conversion algorithm by factor L/M

In practice, an input signal $x[n]$, sampled at frequency F_{s1} is upsampled by insertion of $L-1$ samples with value zero between two consecutive input samples. The resultant signal is then inserted into a low pass filter $h[n]$, which will interpolate the inserted samples and avoid both aliasing, caused by downsampling, and imaging, caused by upsampling. Finally, the output of the filter is downsampled by taking only every M-th sample for the output signal $y[m]$. The relationship between L and M is such that

$$F_{s2} = \frac{L}{M} F_{s1}. \tag{2.1}$$

To avoid the use of multiple frequencies in the computations, one defines a normalized frequency Ω, given by

$$\Omega = \frac{2\pi f}{L f_{s1}} [rad]. \tag{2.2}$$

Regarding the filter $h[n]$, its cutoff frequency depends on the relationship described in Equation (2.1). In the upsampling case ($L \geq M$), one needs to remove the resultant spectral images, using a filter with a normalized cutoff frequency ($\Omega_c \leq 0.5\pi/L$). In the case of downsampling ($L < M$), one needs to filter signal frequencies that would cause aliasing, leading to a filter with a normalized cutoff frequency ($\Omega_c \leq 0.5\pi/M$). By combining these two conditions, the filter's cutoff frequency is given by

$$\Omega_c = min(\frac{\pi}{2L}, \frac{\pi}{2M}) [rad]. \tag{2.3}$$

Note that the normalization considered is in relation to the upsampled frequency:

In the current state-of-the-art, this algorithm is the basis of most synchronous and asynchronous sample rate converters.

For conversions which use small values of L and M the computational effort is modest. However, if the conversion involves sampling rates with a small difference, the factors L and M will increase significantly. As a consequence the computational cost of the conversion increase drastically, only to have

most of the computed samples discarded. Note that, in this case, a small normalised cutoff frequency must be used for the filter.

Furthermore, one needs to design the filter to both remove aliasing and interpolate the $L-1$ inserted samples. This is why the design of the filter $h[n]$ is the main challenge of this architecture. The solution presented in this thesis is based on the use of a fractional delay filter.

Additionally, for asynchronous sample rate converters, the filter is not only time-varying, but also varies with the sample rate ratio. This means that one needs to define a structure that computes the ratio and adapts the filter. For synchronous sample rate converters, the ratio stays constant. I this case the filter is predictable, and it is possible to trade off storage space for computation time, by pre-computing a finite set of filters [6].

2.2 Interpolation/Decimation Filter As A Fractional Delay Filter

An efficient way of obtaining an interpolated value of a sample is to consider that the output samples needed correspond to the input samples, delayed or advanced by a certain value. Considering a discrete-time signal $y[n]$, obtained by delaying a signal $x[n]$,

$$y[n] = x[n - \tau_d] = x[n] * h_d[n], \tag{2.4}$$

where τ_d is the normalized delay. For continuous signals, a time shift in the frequency domain can be expressed as a product between an input $X(e^{j\omega})$ and a filter $H(e^{j\omega})$,

$$Y(e^{j\omega}) = H(e^{j\omega})X(e^{j\omega}), \tag{2.5}$$

where

$$H(e^{j\omega}) = e^{-j\omega\tau_d}. \tag{2.6}$$

By analysis of Equation (2.6), it is possible to note that a delay filter is an all-pass filter with unitary gain and linear phase. However this is only the case for a continuous time domain $x(t)$ signal. For the digital signal $x[n]$, which needs to be reconstructed and aliasing-free, a low-pass filter with a cutoff frequency defined by Equation (2.3) is needed. Since both the reconstruction and anti-aliasing filters are linear systems, they can be combined together in a single filter whose frequency response $H_d(e^{j\Omega})$ is the product of the frequency responses of the two filters:

$$H_d(e^{j\Omega}) = \begin{cases} 1, |\Omega| < \Omega_c \\ 0, otherwise \end{cases}. \tag{2.7}$$

To obtain the impulse response of filter $h[n]$, defined in Equation (2.4), the inverse discrete-time Fourier transform (IDTFT) of $H_d(e^{j\omega})$ is performed.

$$h[n] = IDTFT(H_d(e^{j\omega})) = \frac{1}{2\pi}\int_{-\pi}^{\pi} e^{-j\Omega\tau_d} e^{j\Omega n} d\Omega. \tag{2.8}$$

Since the integrated function is non-zero only in the interval $[-\Omega_c, \Omega_c]$ and $e^{-j\Omega\tau_d}e^{j\Omega n} = e^{j\Omega(n-\tau_d)}$, the integral can be solved yielding

$$h[n] = \frac{sin(\Omega_c(n-\tau_d))}{\pi(n-\tau_d)}. \tag{2.9}$$

The impulse response of the filter $h[n]$ is then defined by Equation (2.10), which is the normalized *sinc* function.

$$h[n] = \frac{\Omega_c}{\pi} sinc\left[\frac{\Omega_c}{\pi}(n-\tau_d)\right]. \tag{2.10}$$

Fig. 2.3 represents the function for $\tau_d = 0$. Note that a variation of τ_d is equivalent to a translation of the figure in the n (discrete time) axis. Furthermore, the cutoff frequency of the filer Ω_c varies with the sample rate conversion ratio. By direct observation of the figure, it is possible to note that for integer values of τ_d, $h[n] = 0$ for all samples except for $n = \tau_d$ where $h[n] = 1$. This is the expected filter response for an integer delay. On the other hand, for a fractional value of τ_d, $h[n]$ is non-zero for all samples. For the ASRC algorithm, $0 \leq \tau_d \leq 1$, since the objective is to use the fractional delay filter as a way to obtain an interpolated sample using a finite number of input neighbor samples separated by an unitary normalized delay.

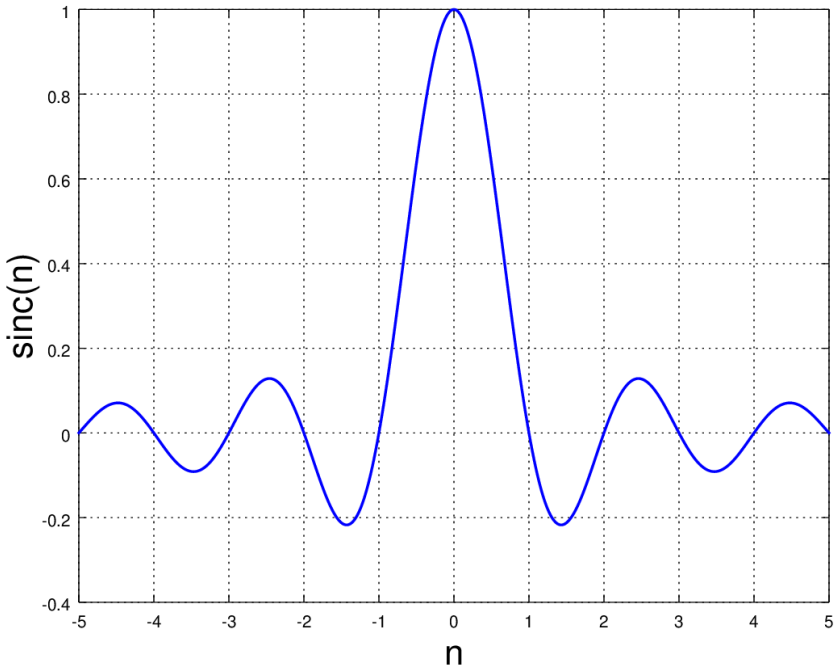


Figure 2.3: Graphical representation of $h[n]$ for $\tau_d = 0$

As this function is infinite and non-causal, an approximation must be done while retaining enough low-pass filtering capability to ensure quality, as explained in Section 2.1. This problem can be solved by

applying a window to the ideal filter $h[n]$. The choice of the format of the window and bandwidth have an influence not only on the quality of ASRC's output signal, but also on the complexity of the computations done.

Using the truncation of the impulse response as an approximation technique, the resultant filter is a section of the sinc function which contains a certain number of zeroes. To exemplify, a filter with six zeroes is considered in Fig. 2.4, and Fig. 2.5.

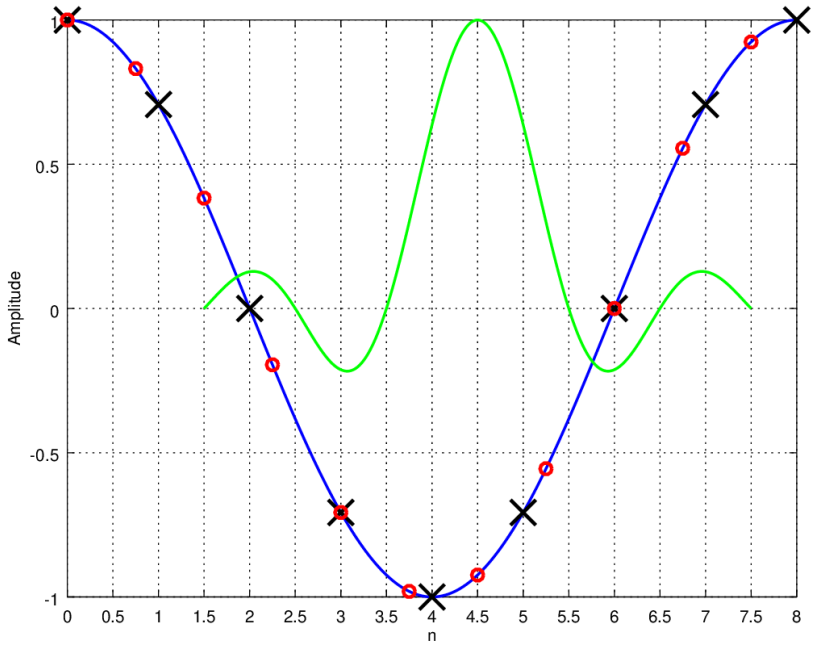


Figure 2.4: Illustration of upsampling (3:4) used to determine the output sample at n = 4.5

In Fig. 2.4, the digital input signal, whose samples are represented by crosses, is plotted along its continuous time representation. To upsample the signal by a factor of 4/3, the output samples, represented by circles, need to be computed. To interpolate these signals, the filter approximation is applied, centering the window at the desired output sample, multiplying each input sample by the corresponding filter value, and accumulate all obtained values. By analysis of this illustration, it is possible to note that a larger upsampling factor will decrease the distance between the output samples. According to Equation (2.3), the normalized cutoff frequency of the filter is $\frac{4}{3}/L$. The filter is always the same in the upsampling case and its number of accumulations is the same as the number of zeroes in the truncated sinc function.

Fig. 2.5 is analogous to Fig. 2.4, representing downsampling by a factor of 4/3 instead. In this case, according to Equation (2.3), the cutoff frequency of the filter is given by $\frac{3}{4}/M$. The number of accumulations is now M times the number of zeroes in the truncated sinc function.

Regarding the approximation of the filter itself, there is a great variety of techniques used to obtain a fractional delay filter which minimizes the discretization error, ranging from applying a window to the

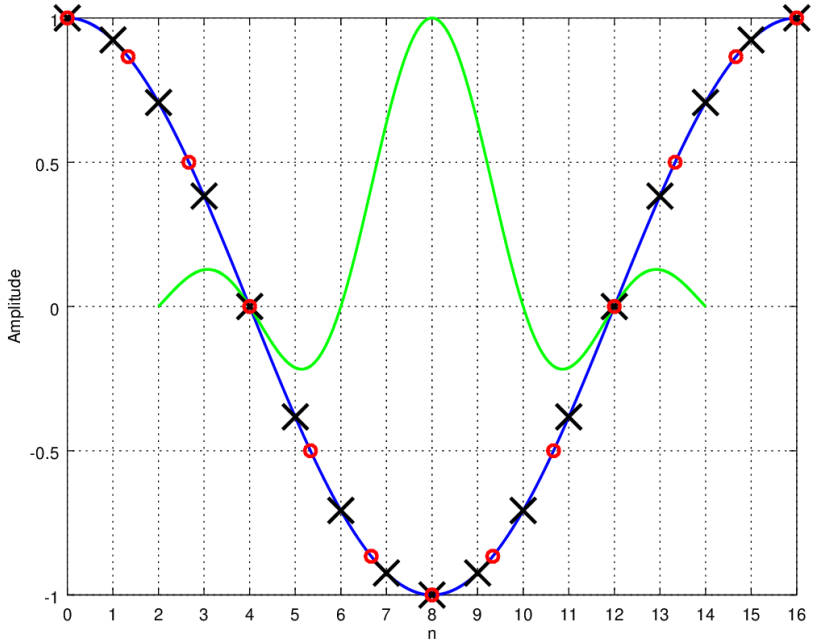


Figure 2.5: Illustration of downsampling (4:3) used to determine the output sample at n = 8

ideal filter to applying Lagrangian interpolation to compute an intermediate coefficient from a set of known filter samples. These and other methods are explained in detail in [7–10].

Chapter 3

Previous Work

With the knowledge that the sample rate converter can be implemented as a filter, as was explained in Chapter 2, multiple distinct implementations can be performed, leading to different results regarding output fidelity and resources usage. In this chapter, some designs proposed in previous works are shown and analyzed, regarding the implementation of the filter, as well as the implementation of the sample rate conversion ratio estimator.

3.1 Implementation of the Interpolation/Decimation Filter

3.1.1 Direct Filtering Using a Multiply-Accumulate Unit (MACC)

As was studied in Section 2.2, the output of the sample rate converter can be obtained by applying a low-pass filter to the input signal. As such, the most direct approach is to perform the direct computation of the discrete convolution between the filter and the samples, using two memories for the coefficients and samples, and a MACC unit for the computations [11]. A block diagram of the MACC unit is shown in Fig. 3.1.

In this implementation, an accumulator is used to generate the phase of the desired output samples, by accumulating the sample rate conversion ratio. The integer part of the accumulated value will correspond to the previous input sample that is closest in phase to the current output sample. The fractional part is the distance between the phase of the desired output sample and the closest input sample, and is used to select the starting point of the filter. Graphically, this selection is represented as centering the sinc to the closest input sample (as exemplified in Fig.2.4 and 2.5). It can also be interpreted as a polyphase filter [4].

After determining the starting values, both the coefficient and sample addresses decrement in equivalent values, obtaining the previous samples and respective coefficients. The output sample $y[n]$, is obtained by

$$y[n] = \sum_{i=0}^{N} a_i x[n-i], \tag{3.1}$$

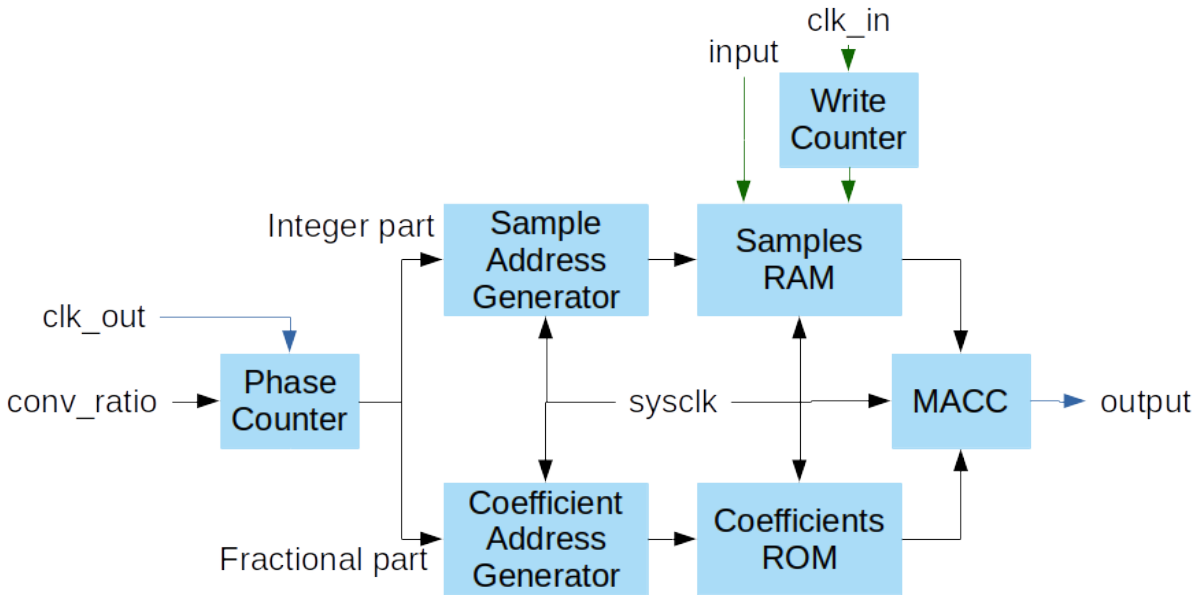


Figure 3.1: MACC filter block diagram.

where a_i are the coefficients of the filter and $x[n-i]$ is the input sample at phase $n-i$.

It is important to note that the filter implemented by this method has a finite impulse response (FIR), which has a non-recursive structure. This is ideal, as FIR filters guarantee a linear phase, which is desired for audio applications.

This design has the advantage of being the easiest to implement, as well as being the one that uses the least amount of logic in comparison to the other implementations presented in this chapter, requiring only one MACC unit for the computations. However, it requires a large amount of filter coefficients, which, in hardware, results in high memory usage, an undesirable result. The problem can be solved by interpolating the filter's coefficients, leading to the use of some logic to reduce the amount of memory used, as some intermediate values do not need to be stored in it.

To guarantee that the sample rate converter supports multiple channels, the structure of the converter should be able to support the computation of multiple output samples. This can be done at the cost of hardware area, by having one output computation unit per channel, or at the cost of computation time, by having one unit computing multiple outputs sequentially.

3.1.2 Cascaded Integrator Comb Filters (CIC)

As was explained in Section 2.1, sample rate conversion can be interpreted as signal upsampling with interpolation, followed by downsampling with decimation. Furthermore, the algorithm can be implemented using FIR filters as shown in the previous section. One of the designs commonly used for this purpose is a Cascaded Integrator Comb filter (CIC) [12]. The filter, originally designed by Hogenauer [13], is based on the implementation of simple integrators and differentiators. CIC filters are obtained by combining adders and delay registers, which consumes a reduced amount of memory and has the great advantage of dispensing with multipliers, which can consume a great amount of hardware resources. Due to the fact that the CIC decimator has a symmetric structure in comparison to a

CIC interpolator, the combination of the two components leads to a highly efficient implementation in application specific integrated circuits (ASIC) and FPGA's. An interpolation filter using the CIC structure can be seen in Fig. 3.2. A decimation filter would have the same components, with the difference that the comb filters and integrator stages would swap positions.

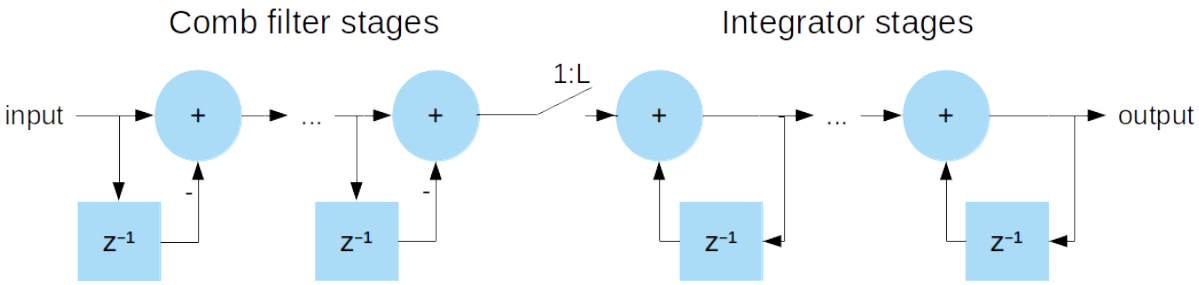


Figure 3.2: Example of cascaded integrator comb filter structure, used as an interpolation filter

While for one CIC structure, the frequency response does not fulfill the requirements, this problem can be solved by cascading multiple interpolation and integration units [12], increasing the attenuation in the filter's stop-band. The implementation is elegant but has limited configurability, due to the fact that no coefficients are used, making it hard to adapt the filter to variations in the sample rates. Furthermore, this type of filter expects integer factors, which leads to the problem of making it able to convert fractional sample rate relations. Finally, this type of filter needs an oversampled signal to properly filter it, making it unusable for a sample rate converter without the use of another filter to obtain more samples.

3.1.3 Approximation By Piece-wise Quadratic Function

As explained in Section 2.2, the impulse response of the filter is a *sinc* function, as expressed by Equation (2.10). In practice, the ideal filter impossible implement, due to its infinite number of taps and requirement to use future samples (non-causality).

The solution presented so far is to truncate and approximate the coefficients and delay the response to make it causal. Another way to address this problem is to split the sinc function into a piece-wise function, and approximate each piece by a quadratic function, leading to an interpolation filter which can be easily implemented [14, 15]. The approximation of the piece-wise sinc function $h(x)$ into quadratic functions can be expressed as

$$h(t) = \begin{cases} a_{1,1}t^2 + b_{1,1}t + c_{1,1}, \left(0 \leq |t| \leq \frac{1}{N}\right) \\ \vdots \\ a_{1,n}t^2 + b_{1,n}t + c_{1,n}, \left(\frac{n-1}{N} \leq |t| \leq 1\right) \\ \vdots \\ a_{s,n}t^2 + b_{s,n}t + c_{s,n}, \left(s - 1 + \frac{n-1}{N} \leq |t| \leq s - 1 + \frac{n}{N}\right) \\ \vdots \\ a_{S,n}t^2 + b_{S,n}t + c_{S,n}, \left(S - 1 + \frac{n-1}{N} \leq |t| \leq S\right) \end{cases}, \tag{3.2}$$

where N is the number of quadratic functions used to represent a piece-wise function, and S is the number of piece-wise functions of the kernel. This technique leads to the implementation presented in Fig. 3.3.

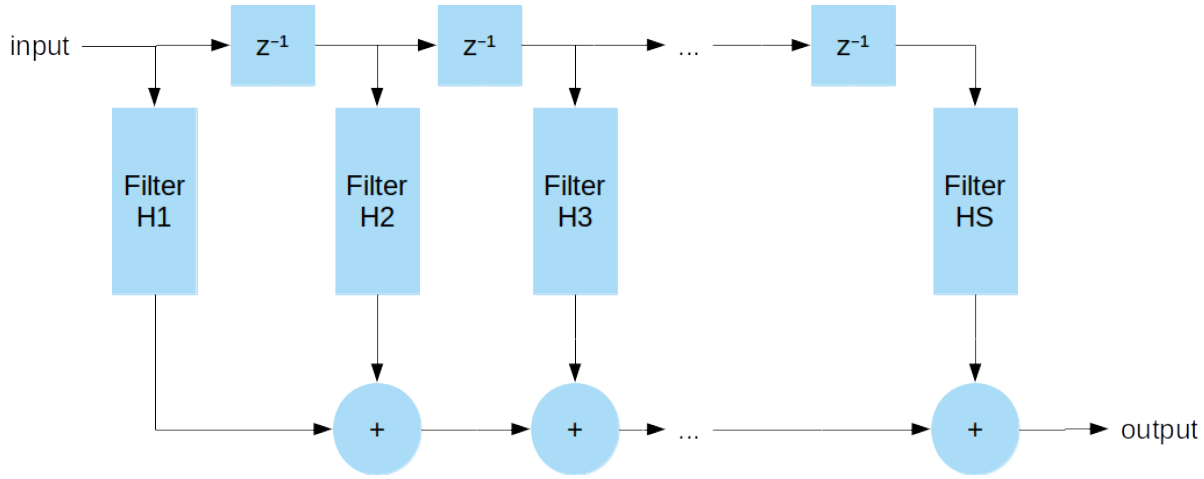


Figure 3.3: Example of a piece-wise kernel filter structure

The design can be further optimized, by considering that from a certain section onward, the polynomial remains the same, changing only in scale by a determined set of factors e_j. The block diagram of the structure is presented in Fig. 3.4. This leads to a structure where the polynomials used remain the same, while the factors e_j change with the sample rate conversion ratio. There are, however, some restrictions that need to be considered, as further explained in [14].

The main advantage of this design is the ability to change sample rate ratio without need to reconfigure the filter, leading to a kernel which not only is able to implement the function, but also does not need to change over time. However, as one needs to define a high number of pieces to obtain large attenuation in the stop-band (to acceptably attenuate distortion and noise), the design requires a large number of quadratic functions, which significantly increases the computational cost. Apart from the structure itself, there is still the high cost of computing the coefficients e_j, which turns even more problematic with applications for which the sample rate conversion ratio varies in time.

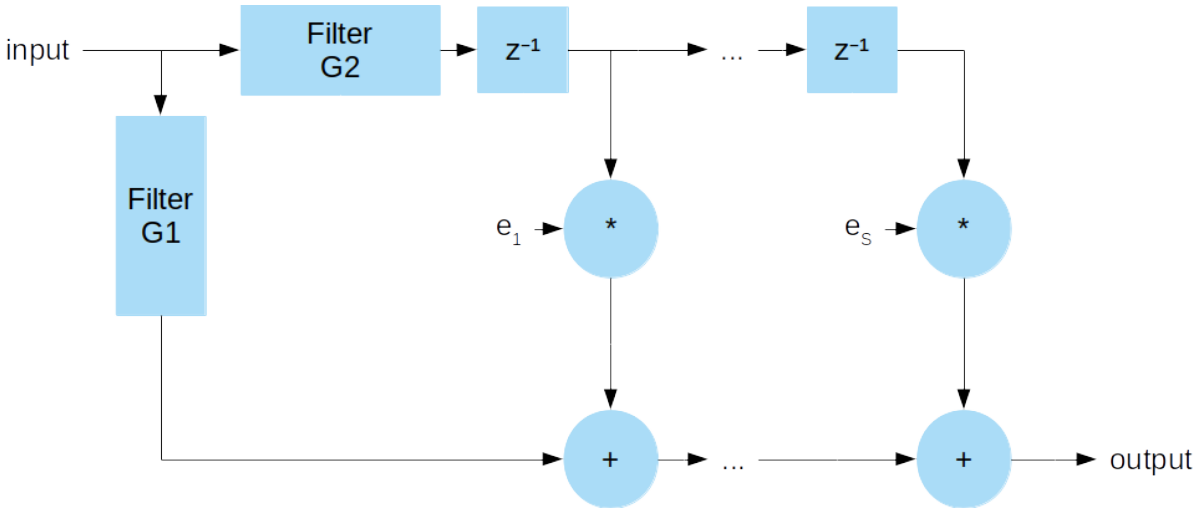


Figure 3.4: Example of optimized kernel filter structure

3.1.4 Farrow Structure

The Farrow Structure is one of the most common implementations of fractional delay filters [16]. The structure is based on the assumption that each sample of the filter impulse response $h[n]$ can be approximated by a polynomial of order q, which depends on the fractional delay d [17]:

$$h[n] = \sum_{m=0}^{q} c_m[n] d^m. \tag{3.3}$$

Similar to the structure presented in Section 3.1.3, a Farrow structure requires the implementation of a set of q FIR filters, which becomes a hardware-intensive design. Furthermore, a change in the fractional delay forces the change of all coefficients, leading to an increase of the computational cost. Over the years, this structure has been subjected to many changes and optimizations, with the objective to develop filters for different applications. For the specific application this thesis is concerned, the structure is optimized to design filters with a variable fractional delay. One possible optimization, further explained in [18], makes it possible to implement a low area Farrow structure, with the great advantage of having fixed coefficients and a single parameter μ that corresponds to the fractional component of the output instant, when normalized to the input frequency. The block design of the structure is presented in Fig. 3.5, where the value of μ can be computed by

$$\mu = \frac{kT_{out} - mT_{in}}{T_{in}}, \tag{3.4}$$

where T_{in} and T_{out} are the input and output clock periods, respectively, m is the largest integer for which $mT_{in} \leq kT_{out}$, and $k \in \{0, 1, 2, ...\}$, is a value which increments after the computation of one output. The transfer function of each FIR sub-filter, $C_m(z)$ is expressed by

$$C_m(z) = \sum_{k=0}^{N-1} c_m \left(k - \frac{N}{2} \right) z^{-k}. \tag{3.5}$$

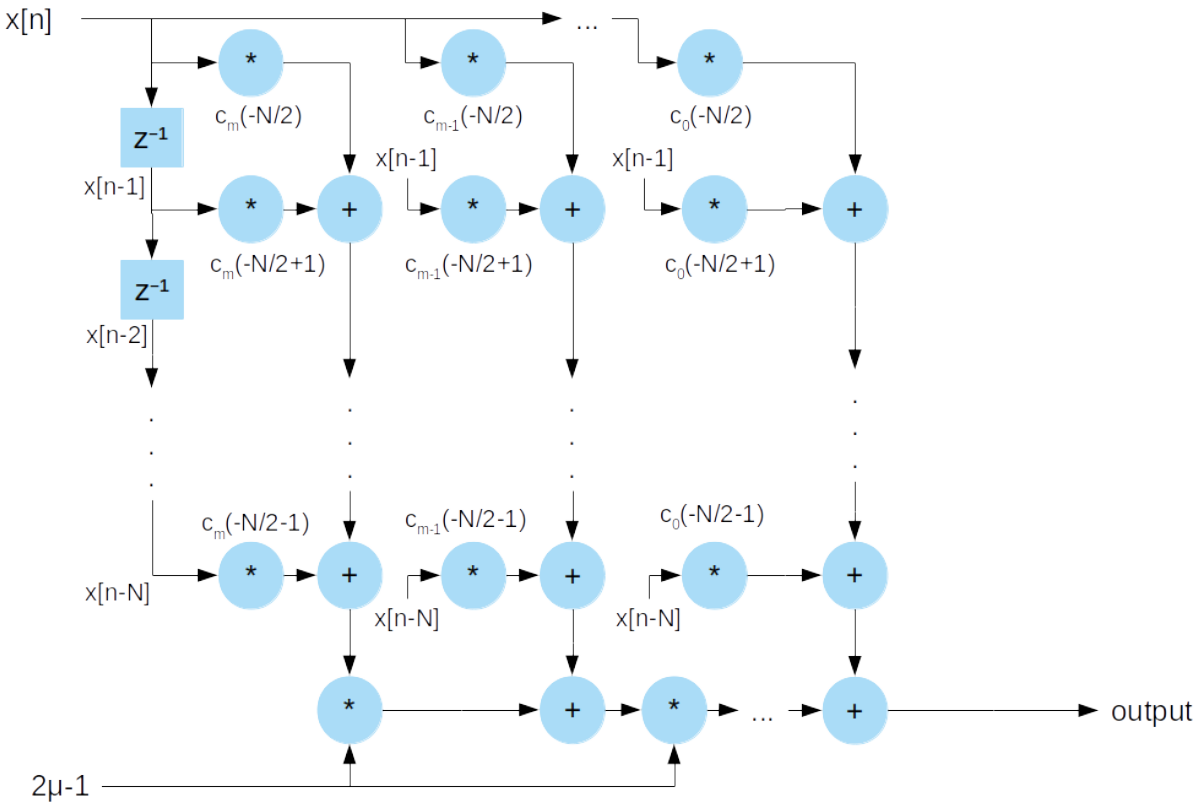


Figure 3.5: Example of farrow structure, optimized for variable fractional delay

While the structure presented in Fig. 3.5 is optimized for interpolation, there are some modifications, explained in detail in [18] which apply to decimation. This structure has the disadvantage of having to include all sub-filters to compute all coefficients c_n, it is one of the easiest methods to allow variations on sample rate conversion ratios, and occupies a low area, which is mainly taken by the delay elements.

3.2 Implementation Of The Sample Rate Conversion Ratio Estimator

With the analysis done in Chapter 2, and the implementation of one of the filters described in Section 3.1, a synchronous sample rate converter (SSRC) can be implemented. To implement its asynchronous counterpart, the sample rate conversion ratio needs to be computed.

The sample rate conversion ratio can be expressed by

$$\rho = \frac{F_{sOut}}{F_{sIn}}, \tag{3.6}$$

where ρ is the sample rate conversion ratio, and F_{sIn} and F_{sOut} are the respective sample rates of the input and output signals.

3.2.1 Period Measurement And Averaging

By analysis of Equation (3.6), it is possible to determine the value of ρ by direct measurement of the sample rate clock periods, T_{sIn} and T_{sOut}, as

$$\rho = \frac{T_{sIn}}{T_{sOut}}. \tag{3.7}$$

An analysis and implementation of two different implementations of this method were done in [19].

The first implementation consists on using a phase locked loop (PLL) to multiply the input sample rate clock's frequency by a factor of 2^{L-k}, where L is the desired number of precision bits of ρ, and k is a parameter related to the number of measurements averaged. The output of the PLL is then used to clock a counter that will count the output sample rate clock's period. The average value of the counter's output is the inverse of ρ. A block diagram of this implementation is represented in Fig. 3.6.

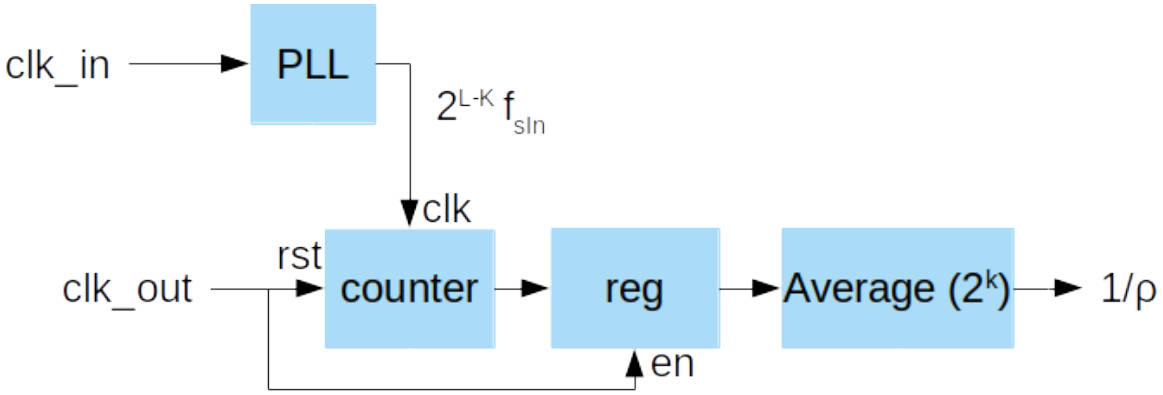


Figure 3.6: Period measurement and ρ computation using a PLL.

The other implementation consists on using two counters clocked at a higher speed than F_{sIn} and F_{sOut}, which will count the sample rate clock's periods. The outputs of these counters can be averaged for improved precision, and then are divided in accordance to Equation (3.7). The block diagram is represented in Fig. 3.7.

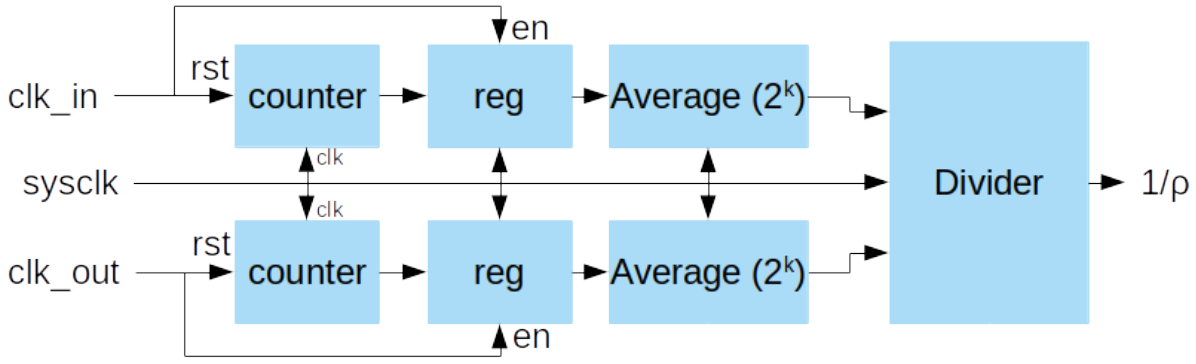


Figure 3.7: Period measurement and ρ computation using a divider.

While the first implementation requires the implementation of the PLL, which may require additional hardware, the second implementation uses an independent clock, as well as the hardware cost of the

additional counter and average block. In both implementations, the precision of the computed ρ depends on the frequency of the clock used for the counters, as well as the amount of averages done.

Additionally, this implementation has no error tracking mechanism, leading to a drift of the group delay of the converter, due small errors in the computed value of ρ. An additional module to avoid the drift needs to be implemented. This mechanism can be implemented as a frequency tracker, that stabilizes the group delay of the converter through a feedback loop.

3.2.2 Digital Phase Locked Loop As Frequency Tracker

To compute the frequency ratio of the sample rate clocks, a digital phase locked loop (DPLL) can be used. This loop uses the value of the inverse of ρ to control the phase of the output in a feedback loop, allowing it to track the input's phase. Studies of this method were made in [19, 20]. Analog Devices' ASRC [1] also uses this loop.

Similarly to a conventional PLL, the DPLL can be split into three blocks: a phase detector, a loop filter and a voltage controlled oscillator (VCO).

The phase detector is used to compute the difference of the output and input phases. In the digital sense, it can be done with phase counters and a subtractor. Through Equation (3.6), it is possible to obtain

$$F_{sOut} = \frac{F_{sIn}}{\rho}. \tag{3.8}$$

Since the phase can be seen as the frequency's primitive, and assuming that the initial phase of both input and output is zero, then the input and output's phase, ϕ_{in} and ϕ_{out} can be expressed as

$$\begin{cases} \phi_{in} = F_{sIn} \times n \times \Delta t \\ \phi_{out} = F_{sOut} \times n \times \Delta t \end{cases}, \tag{3.9}$$

where n is the discrete time passed since the beginning of counters' accumulations. By combining Equation (3.8) with Equation (3.9), the input and output phases can be related, yielding

$$\phi_{out} = \frac{1}{\rho} \times \phi_{in}. \tag{3.10}$$

Due to the fact that the counters are only updated at the sample rate clocks' frequencies, they need to be sampled at the same frequency. To allow this, the input's phase is sampled at the output's frequency.

The block diagram of the phase detector is shown in Fig.3.8.

For the loop filter, a low-pass filter needs to be implemented. The filter has the objective to attenuate the effect of the phase detector's jitter. However, a higher attenuation leads to a slower tracking of the DPLL. An example of a filter is a lead-lag section filter, as explained in [19].

The VCO receives the filtered phase difference to compute an output frequency. In the digital domain, this can be achieved with an integrator block.

The block diagram of the loop filter and VCO are shown in Fig. 3.9.

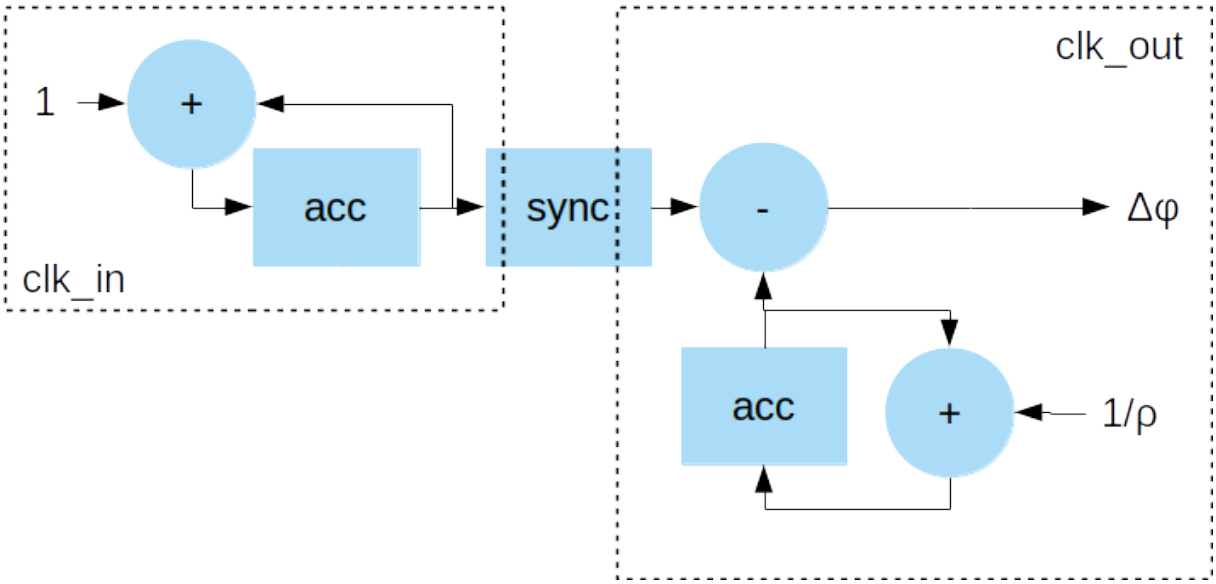


Figure 3.8: DPLL phase detector block diagram.

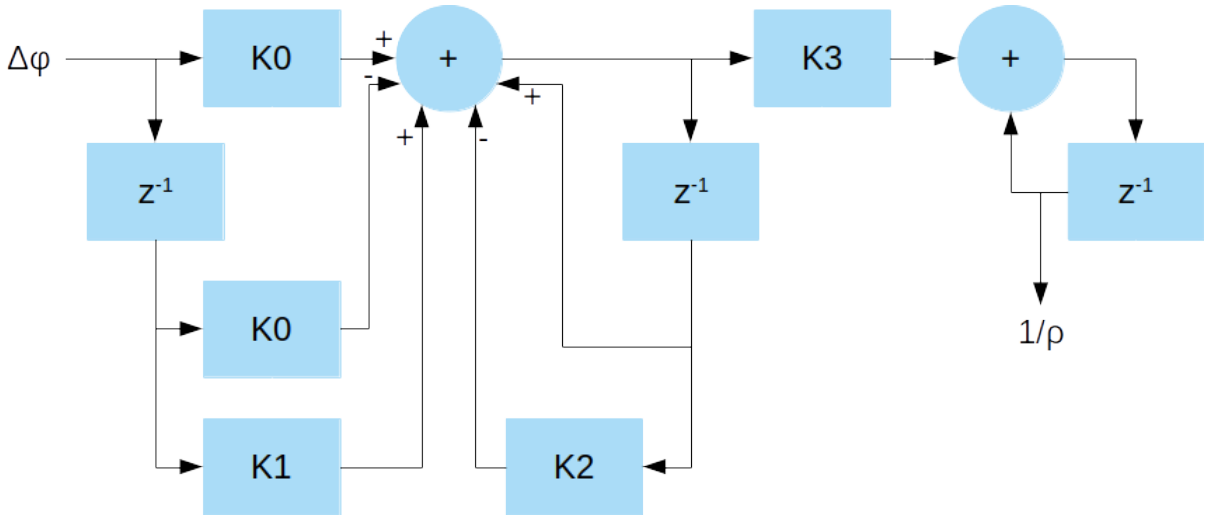


Figure 3.9: DPLL loop filter and VCO block diagram.

To improve the DPLL's dynamic behavior, the filter's bandwidth can be controlled, by allowing a higher bandwidth for a faster tracking and gradually lowering it for an improved precision and noise rejection. An in-lock detector that uses the phase difference as input can be used.

This implementation has the advantage of not requiring a fast clock to work, as the computations are done at the output's sample rate clock. However, it also leads to the need to determine the correct parameters (K0, K1, K2 and K3) of the filter, as well as the additional hardware used to implement the filter. In FPGA implementations, it may also lead to timing problems, as pipelining leads to latency in the computations, which impacts the DPLL's performance.

Chapter 4

Proposed Design

The symbol and interface signals of the sample rate converter design are presented in Fig. 4.1. The data input and output signals are $audio\_in$ and $audio\_out$, respectively, and the sample rate clocks are $audio\_in\_wclk$ and $audio\_out\_wclk$, respectively. The definition of all interface signals is presented in Table 4.1.

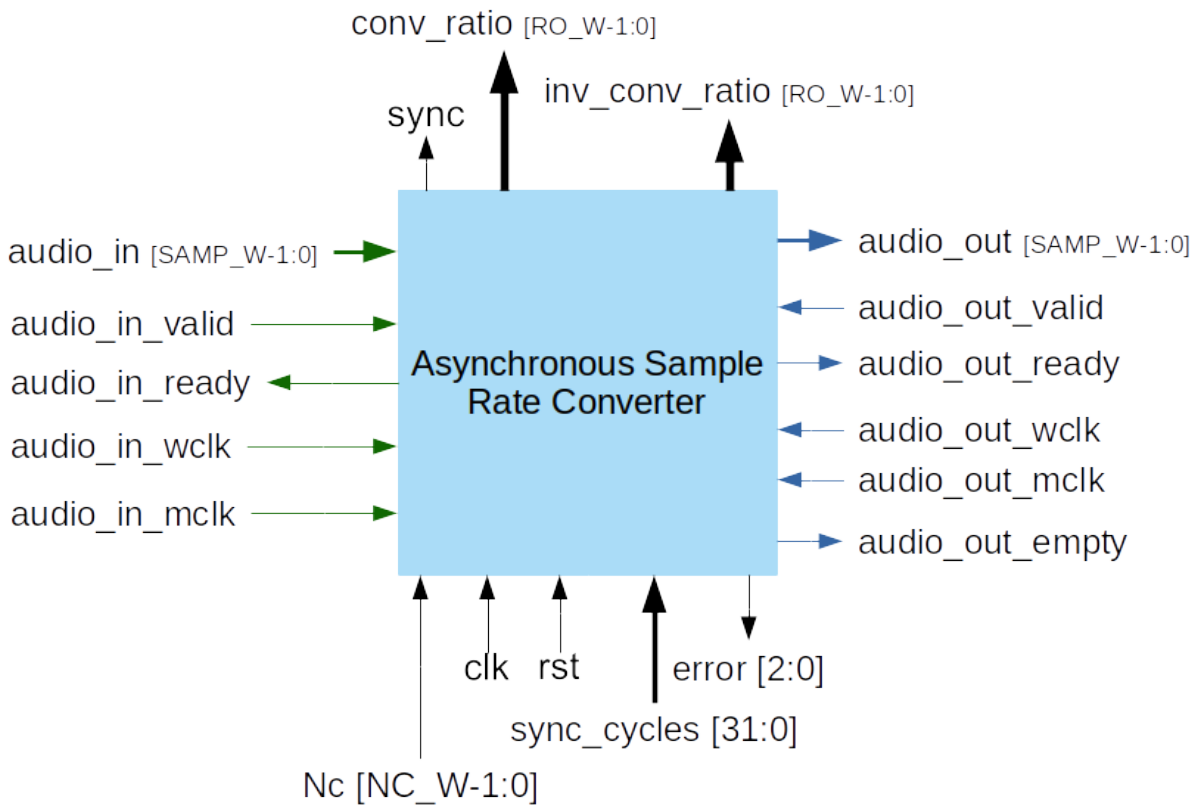


Figure 4.1: Symbol and interface diagram.

As a core designed for FPGA and ASIC implementations, the ASRC can be configured pre-synthesis to match the intended application. The synthesis parameters are presented in Table 4.2.

Name	Direction	Width	Description
clk	input	1	System clock signal.
rst	input	1	Reset signal active high (1).
Nc	input	NC_W	Number of channels to convert must be power of 2.
sync_cycles	input	32	Number of system clock cycles to wait for synchronization.
sync	output	1	Active high (1) when the ASRC has computed conversion ratio.
conv_ratio	output	RO_W	Sample rate conversion ratio.
inv_conv_ratio	output	RO_W	Inverse of sample rate conversion ratio.
audio_in	input	SAMP_W	Input audio stream.
audio_in_valid	input	1	Input audio valid signal.
audio_in_ready	output	1	Input audio ready signal.
audio_in_mclk	input	1	Input audio master clock (frequency should be $N \times f\_s$ where f_s is the input sample rate and N is an integer greater than the number of channels).
audio_in_wclk	input	1	Input audio word clock (frequency should be equal to input sample rate).
audio_out	output	SAMP_W	Output audio stream.
audio_out_ready	output	1	Active high (1) if the ASRC has a valid output sample buffered.
audio_out_wclk	input	1	Output audio word clock (frequency should be equal to input sample rate).
audio_out_mclk	input	1	Output audio master clock (frequency should be $N \times f\_s$ where f_s is the output sample rate and N is an integer greater than the number of channels).
audio_out_valid	input	1	Output audio valid signal.
audio_out_empty	output	1	Set low (0) if there are output samples to retrieve.
error	output	3	Error bits (2) Audio out buffer— (1) PTR_DIFF— (0) NChannels

Table 4.1: Interface signals.

Parameter	Default Value	Description
SAMP_W	24	Audio sample width
NC_W	8	Number of channels signal width
RO_W	35	Sample rate conversion ratio width
SAMP_BUF_W	10	Data memory size (log_2)

Table 4.2: Synthesis Parameters

The ASRC is divided in three main modules: the Input Data Memory, the Ratio Estimator and the Resampler. Additionally, the ASRC includes a positive edge detector circuit, implemented with a register, to generate the $start$ signal, used to start the computation of a new output sample, as well as an asynchronous FIFO, used to temporarily store the output samples to be read, and to allow their crossing from the system clock to the output audio master clock. A block diagram of the ASRC is presented in Fig. 4.2.

As the core works in three different clock domains, all figures present the signals in three different colors, with green arrows indicating signals in the $audio\_in\_mclk$ domain, blue arrows for signals in the $audio\_out\_mclk$ domain, and black arrows for signals in the clk domain. This is important, as wherever there is a clock domain crossing, one needs to implement a suitable synchronizer circuit.

A synchronizer module performs the crossing of the $audio\_in\_wclk$ and $audio\_out\_wclk$ signals to the system clock domain. Since the $wclk$ signals are single wires, and the transitions are from a slower to a faster clock domain, the synchronizer module is composed by two registers, clocked by clk.

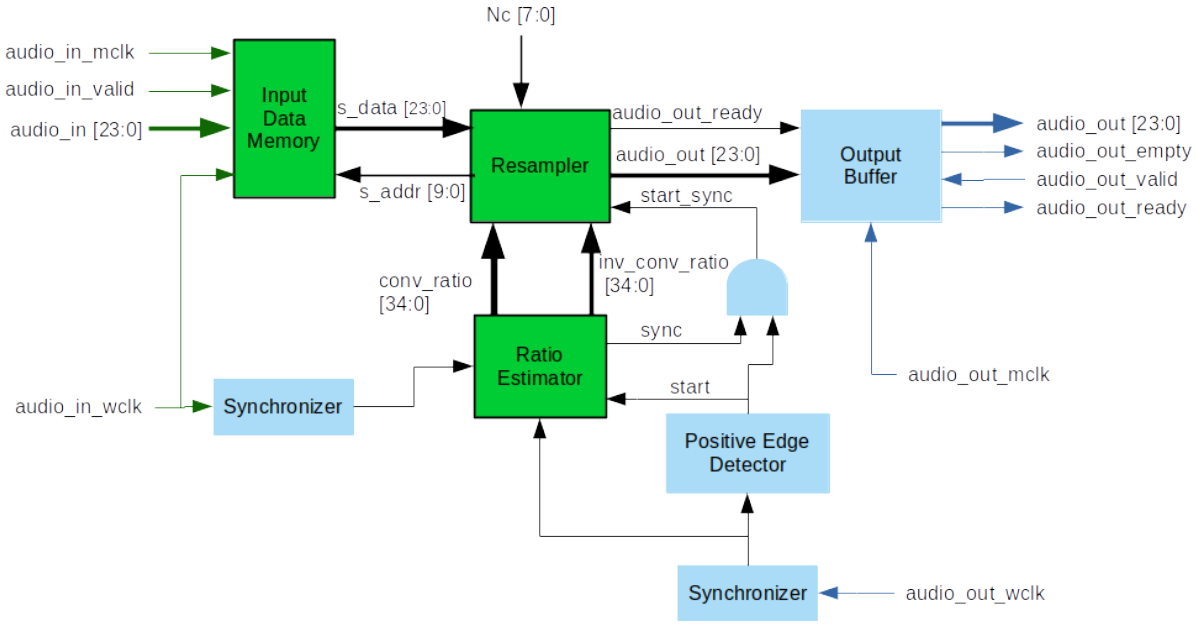


Figure 4.2: Asynchronous sample rate converter top block diagram.

4.1 Data Memory

To store the input samples, a data memory is used. The data memory is a dual port Random Access Memory (RAM) capable of working at two different clock domains. A block diagram of the data memory and its write port address counter is presented in Fig. 4.3.

In the input clock domain, the samples $audio\_in$ are stored in the position given by $audio\_in\_waddr$ at the rate of $audio\_in\_wclk$. The write address counter $audio\_in\_waddr$ is incremented at the $audio\_in\_wclk$ rate, whenever $audio\_in\_valid$ is active. In the system clock domain, the samples are read by accessing the position given by s_addr. The read samples are used to perform the filtering. For Nc Time Division

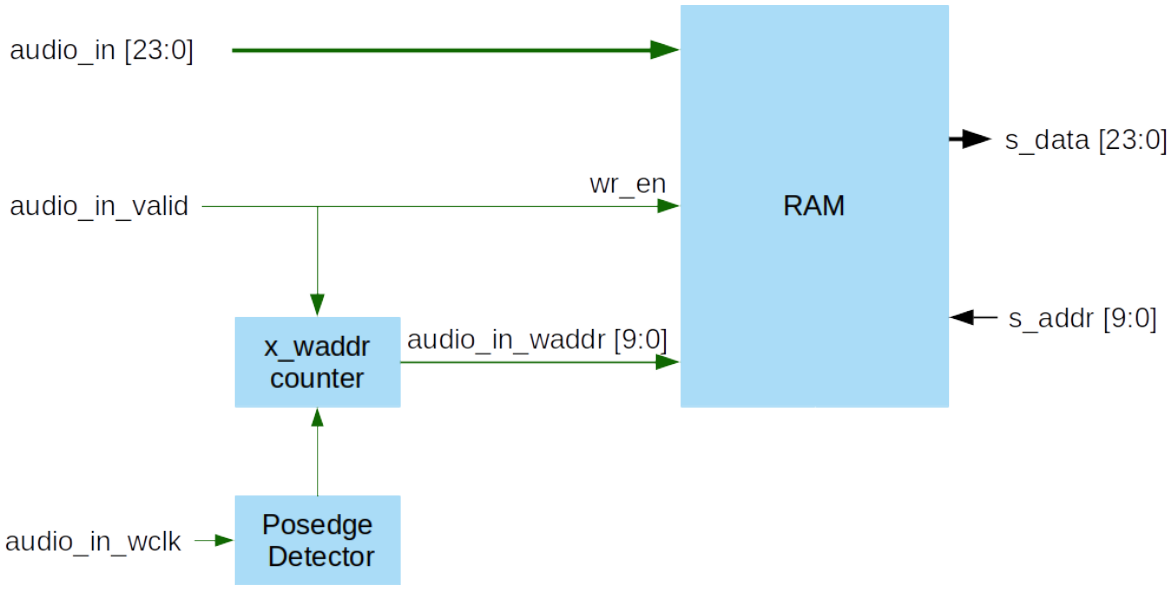


Figure 4.3: Data memory module block diagram.

Multiplexed (TDM) channels, to obtain the next sample of a certain channel, the address $s\_addr[n+1]$ is given by

$$s\_addr[n+1] = s\_addr[n] + Nc. \tag{4.1}$$

Since the accumulators do not stop or reset during the conversion, they wrap around creating a circular memory. When there is a stream of input samples, the newer ones will overwrite the older ones which will not be used anymore in the filter. On average, the write and read pointers increment at the same rate, which is guaranteed by the Ratio Estimator block.

Since the data memory is a true dual port memory, the synchronizer to cross the clock domains unnecessary, as the memory already serves that purpose.

4.2 Ratio Estimator

The sample rate converter is asynchronous, so the frequency of the input and output clocks can change over time. Furthermore, their frequencies can also take any value in the supported range. The frequencies of the data clocks are unknown by the core, but their ratio, ρ, must be computed. The ratio estimator computes an estimated value of the sample rate conversion ratio, $\hat{\rho}$.

As was studied in section 3.2, there are two possible implementations for the ratio estimator. In this thesis, a period measurement and averaging implementation, described in subsection 3.2.1, is used. The system clock is used as the reference clock, and two counters and a divider obtain the approximate sample rate conversion ratio ($\hat{\rho}$), as shown in Fig. 3.7. Additionally, a frequency tracker is added to ensure that the error in the approximation $\hat{\rho^{-1}}$ of ρ^{-1} does not lead to a deviation of the write and read data memory pointers.

4.2.1 Initial Approximation - Period Measurement and Averaging

The block diagram of the period meter is shown in Fig. 4.4.

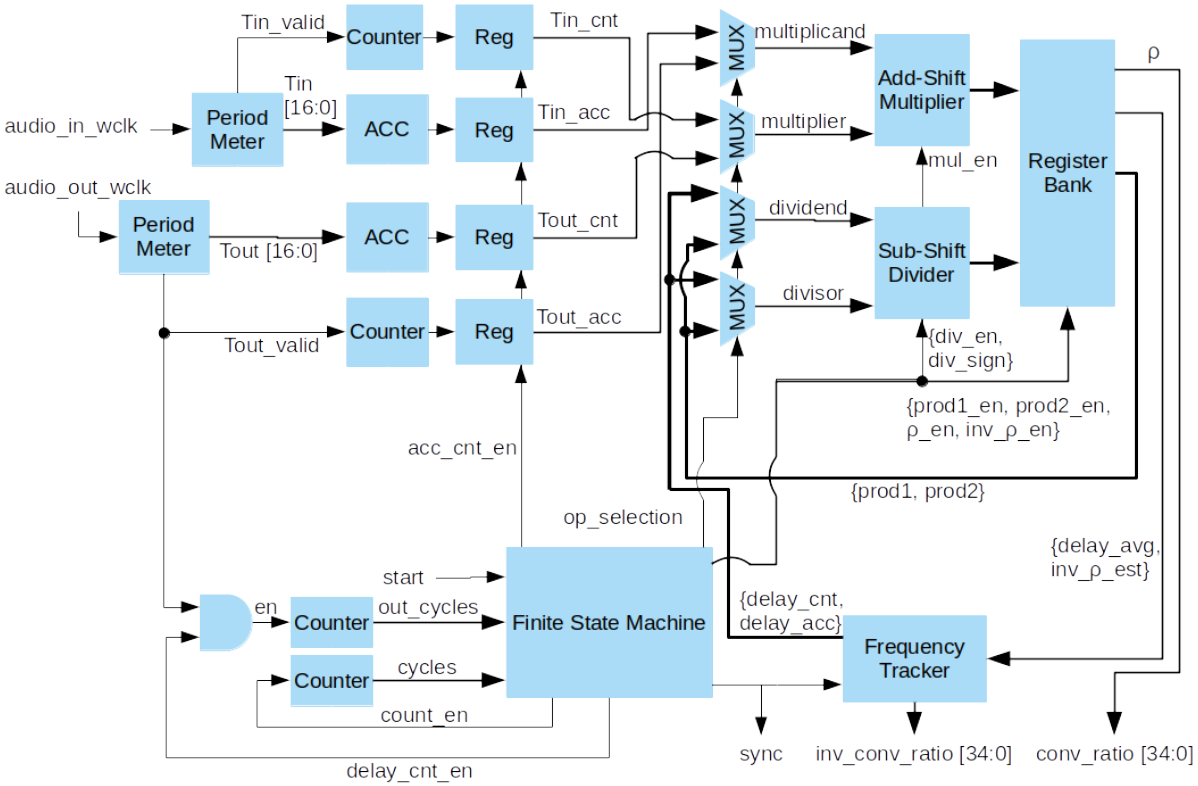


Figure 4.4: Ratio Estimator module block diagram.

The Period Meter block uses a counter to obtain the values, $\hat{Tin}$ and $\hat{Tout}$ of the data clocks periods measured in system clock cycles. Since the counter is incremented at the system clock rate f_{sys_clk}, approximate values of the data clock periods $Tin_{seconds}$ and $Tout_{seconds}$ are given by

$$\begin{cases} Tin_{seconds} = \frac{\hat{Tin}}{f_{sys\_clk}} \\ Tout_{seconds} = \frac{\hat{Tout}}{f_{sys\_clk}} \end{cases}. \tag{4.2}$$

The ratio ρ between the input and output clock frequencies f_{in} and f_{out} is defined by

$$\rho = \frac{f_{out}}{f_{in}}. \tag{4.3}$$

Combining Equation (4.2) and (4.3), an approximate value of ρ can be computed by

$$\hat{\rho} = \frac{\hat{Tin}}{\hat{Tout}}. \tag{4.4}$$

The value of $\hat{\rho}$ can be computed in hardware without knowing any clock frequencies. However, since the counter is only able to count integer values, the periods of the input and output clocks are imprecise. For the same clock frequency the values $\hat{Tin}$ and $\hat{Tout}$ can vary each time they are measured, with an absolute error of 1 time unit (clock period). It should be noted that the resolution of the counter increases

with the frequency of the system clock: a higher system clock frequency leads to a lower error in $\hat{Tin}$ and $\hat{Tout}$.

To improve the precision of this approximation, the measured periods $\hat{Tin}$ and $\hat{Tout}$ are accumulated (Tin_acc and $Tout\_acc$) and counted (Tin_cnt and $Tout\_cnt$), to compute an average value. The operation is expressed as

$$T\_avg = \frac{T\_acc}{T\_cnt}. \tag{4.5}$$

The combination of the Equations (4.4) and (4.5) leads to the expression used by the design:

$$\begin{cases} \rho = \frac{Tin\_acc \times Tout\_cnt}{Tin\_cnt \times Tout\_acc} = \frac{prod1}{prod2} \\ \rho^{-1} = \frac{Tout\_acc \times Tin\_cnt}{Tout\_cnt \times Tin\_acc} = \frac{prod2}{prod1} \end{cases}. \tag{4.6}$$

One needs to perform a significant number of accumulations to obtain an accurate average; the accumulations are performed for a certain synchronization time. A counter, clocked at the system clock, is used to count this synchronization time. The number of synchronization cycles counted is given by the user through the interface of the core.

After the measurements are done and the average is computed, the resulting values of $\hat{\rho}$ and $\hat{\rho^{-1}}$ are valid and accurate enough to allow the ASRC to start the conversion.

4.2.2 Frequency Tracker

The approximate values, $\hat{\rho}$ and $\hat{\rho^{-1}}$, obtained are accurate enough to fulfill the specification. However, the inaccuracy of $\hat{\rho^{-1}}$ leads to a drift of the group delay of the converter, which may lead to simultaneous read and write operations at the same address of the data memory. The Frequency Tracker block is used to correct the error by using the approximate value $\hat{\rho^{-1}_{est}}$, computed by the hardware described in subsection 4.2.1, as an initial approximation. The frequency tracker design is a simplification of the digital PLL described in subsection 3.2.2, as the loop filter is implemented by averaging multiple phases, obtained by the phase detector. The block diagram of the frequency tracker is shown in Fig. 4.5.

The frequency tracker uses the phases normalized to the input sample rate. As such, on every tick of the input word clock, the input phase accumulator adds '1.0'. Meanwhile, the output phase accumulator will add $\hat{\rho^{-1}}$ on every tick of the output word clock. Note that, since the output sample rate is ρ times faster than the input counterpart, the output phase accumulator will in average accumulate $\hat{\rho}$ times the amount of input phase accumulations. During certain amount of time Δt, the accumulated phases ϕ_{in} and ϕ_{out} are given by

$$\begin{cases} \phi_{in} = 2\pi F_{sIn} \times \Delta t \\ \phi_{out} = \hat{\rho^{-1}} \times 2\pi F_{sOut} \times \Delta t \end{cases}, \tag{4.7}$$

where F_{sIn} and F_{sOut} are the sample rates of the input and output, respectively. The normalization of Equation (4.7) to the input sample rate, and the application of Equation (3.8) yield

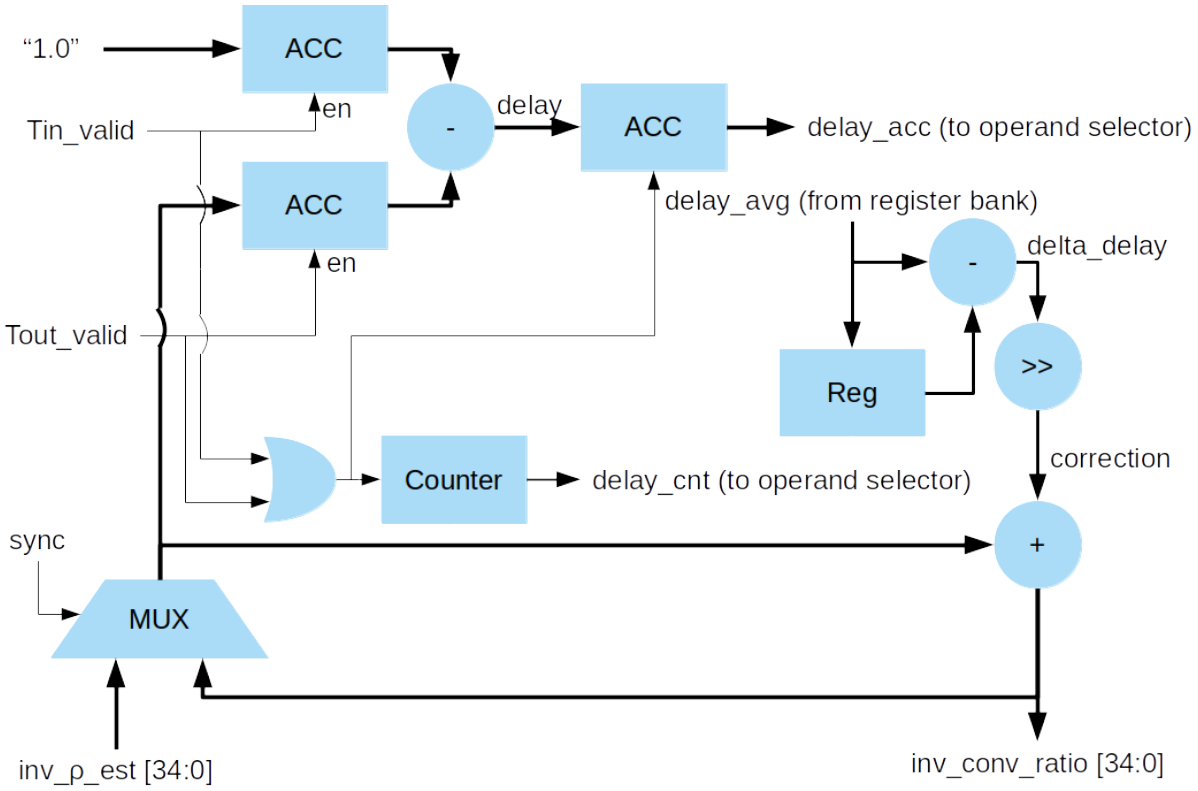


Figure 4.5: Frequency Tracker module block diagram.

$$\begin{cases} \frac{\phi_{in}}{2\pi F_{sIn}} = \Delta t \\ \frac{\phi_{out}}{2\pi F_{sIn}} = \rho \times \hat{\rho^{-1}} \times \Delta t \end{cases} . \tag{4.8}$$

By subtracting the two expressions of Equation (4.8) one obtains the variation of the delay between the input and output phases, given by

$$\frac{\phi_{in} - \phi_{out}}{2\pi F_{sIn}} = \Delta t - \rho \times \hat{\rho^{-1}} \times \Delta t. \tag{4.9}$$

To adapt to the discrete time domain, and normalize the variation of delay to a single variable ΔD, Equation (4.9) can be rewritten as

$$\Delta D = \frac{(\phi_{in} - \phi_{out})}{2\pi} = \rho^{-1} \times N_{out} - \hat{\rho^{-1}} \times N_{out}, \tag{4.10}$$

where N_{out} is the number of output accumulations done, given by

$$N_{out} = F_{sOut} \times \Delta t = \rho \times F_{sIn} \times \Delta t. \tag{4.11}$$

If the value of $\hat{\rho^{-1}}$ is accurate, being equal to ρ^{-1}, the value of ΔD will be equal to zero. Otherwise, there is a drift of the input and output phases, which will influence the value of ΔD. As such, the value of ΔD can be used to apply corrections to the value of $\hat{\rho^{-1}}$, to ensure that ΔD converges to zero. The correction needed to apply, C is given by

$$C = \rho^{-1} - \hat{\rho^{-1}}, \tag{4.12}$$

By combining Equations (4.10) and (4.12), the expression used by the module to determine the correction applied to $\hat{\rho^{-1}}$ is obtained, being

$$C = \frac{\Delta D}{N_{out}}. \tag{4.13}$$

To obtain the value of ΔD in hardware, two values of the delay are measured sequentially, as the difference between them is the value of ΔD used. The delay itself is the difference between the accumulated input and output phases. To obtain more precision, the delay is measured for 4096 output clock cycles ($N_{out} = 4096$). Additionally, to ensure that the corrections of $\hat{\rho^{-1}}$ occur as smoothly as possible, and to reject clock jitter, the value of the correction applied is attenuated, as a strong variation of $\hat{\rho^{-1}}$ negatively impacts the audio fidelity, increasing the THD+N.

4.2.3 Ratio Estimator Control Unit

To save on the silicon area used, only one multiplier and one divider are used, and shared by both the Ratio Estimator and the Frequency Tracker. The divider and the multiplier are implemented using a serial shift-subtract and shift-add units, respectively, for minimum size and power consumption, as performance is not important here.

The inputs of the multiplier and divider, as well as the accumulators and counters, are controlled by a finite state machine (FSM). The Frequency Tracker block is also controlled by the FSM, and also uses the divider to compute $\hat{\rho}$. The state transition diagram of the FSM is shown in Fig. 4.6.

The FSM has the following states:

- State 0: Initial state - The FSM resets the counters and accumulators.
- State 1: Measure periods - The FSM keeps the enables of the period meters active. The state does not change until the system clock cycle counter reaches the value input by the user.
- State 2: Compute ρ dividend ($prod1$) - The FSM selects Tin_acc as multiplicand and $Tout\_cnt$ as multiplier, and enables the multiplier.
- State 3: Wait for multiplication - The FSM keeps the multiplier enabled. The state does not change until the multiplier is ready for a new multiplication. During the state transition, the FSM enables the register for $prod1$, storing the output of the multiplier in it.
- State 4: Compute ρ divisor ($prod2$) - The FSM selects $Tout\_acc$ as multiplicand and Tin_cnt as multiplier, and enables the multiplier.
- State 5: Wait for multiplication - Similar to State 3. Instead of enabling the register for $prod1$ during the state transition, the FSM enables the register for $prod2$.

- State 6: Compute ρ - The FSM selects $prod1$ as dividend and $prod2$ as divisor, enables the divider and keeps the divider sign signal disabled (unsigned division).
- State 7: Wait for division - The FSM keeps the divider enabled. The state does not change until the divider is ready for a new division. During state transition, the FSM enables the register for ρ.
- State 8: Compute inv_ρ - The FSM selects $prod2$ as dividend and $prod1$ as divisor, enables the divider and keeps the divider sign signal disabled (unsigned division).
- State 9: Wait for division - Similar to State 7. Instead of enabling the register for ρ, the FSM enables the register for inv_ρ during state transition.
- State 10: Measure delay - The FSM keeps the delay counter and accumulator enabled. The state is kept while the output period counter counts up to 4096. Note that from this state onward, the ASRC is ready to start the conversion. As such, the FSM toggles the $sync$ signal on.
- State 11: Compute average delay - The FSM selects $delay\_acc$ as dividend and $delay\_cnt$ as divisor, and enables the divider with sign active (signed division).
- State 12: Wait for division - Similar to State 7 and 9. During state transition, the FSM enables the register for $delay\_avg$, and resets the delay counter and accumulator. This state transitions to state 10. Note that the frequency tracker module uses the value of $delay\_avg$ to periodically adjust $\hat{\rho^{-1}}$

The outputs of the FSM per state are shown in Table 4.3.

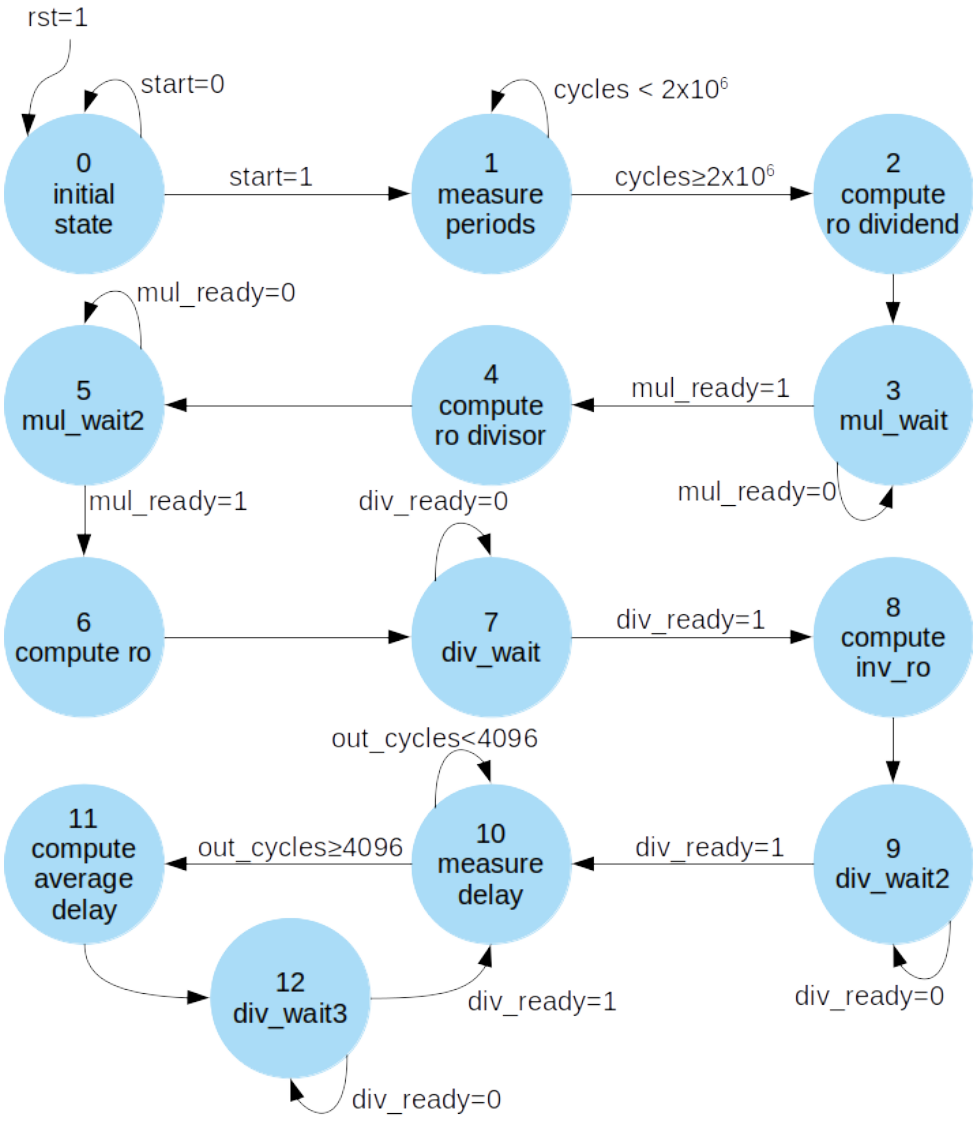


Figure 4.6: Ratio Estimator FSM state diagram.

State ID	Output Signals	Notes
0	count_rst = 1	
	delay_cnt_rst = 1	
	sync_nxt = 0	
1	count_en = 1	
2	multiplicand = Tin_acc	
	multiplier = Tout_cnt	
	mul_en = 1	
3	mul_en = 1	Only if mul_ready=0.
	prod1_en = 1	Only if mul_ready=1. Multiplier output is stored in register prod1.
4	multiplicand = Tout_acc	
	multiplier = Tin_cnt	
	mul_en = 1	
5	mul_en = 1	Only if mul_ready=0.
	prod2_en = 1	Only if mul_ready=1. Multiplier output is stored in register prod2.
6	dividend = prod1	
	divisor = prod2	
	div_en = 1	
	div_sign = 0	
7	div_en = 1	Only if div_ready=0.
	ρ_en = 1	Only if div_ready=1. Divider output is stored in register ρ.
8	dividend = prod2	
	divisor = prod1	
	div_en = 1	
	div_sign = 0	
9	div_en = 1	Only if div_ready=0.
	inv_ρ_en = 1	Only if div_ready=1. Divider output is stored in register inv_ρ. ASRC is ready to start the conversion.
	sync_nxt = 1	
	count_rst = 1	
10	count_en = 1	
	delay_cnt_en = 1	
11	dividend = delay_acc	
	divisor = delay_cnt	
	div_en = 1	
	div_sign = 1	
12	div_en = 1	Only if div_ready=0.
	delay_avg_en = 1	Only if div_ready=1. Divider output is stored in register delay_avg.
	delay_cnt_rst = 1	
	count_rst = 1	

Table 4.3: Conversion ratio estimator FSM output signals per state.

4.3 Resampler

The resampler is the main module of the sample rate converter. The resampler is implemented with a direct filter that uses a MACC unit, presented in subsection 3.1.1.

The resampler is split into three submodules: the *address generator*, the *coefficient memory* and the *multiply-accumulator* submodule, which are explained in the next subsections. A block diagram of the resampler is presented in Fig. 4.7.

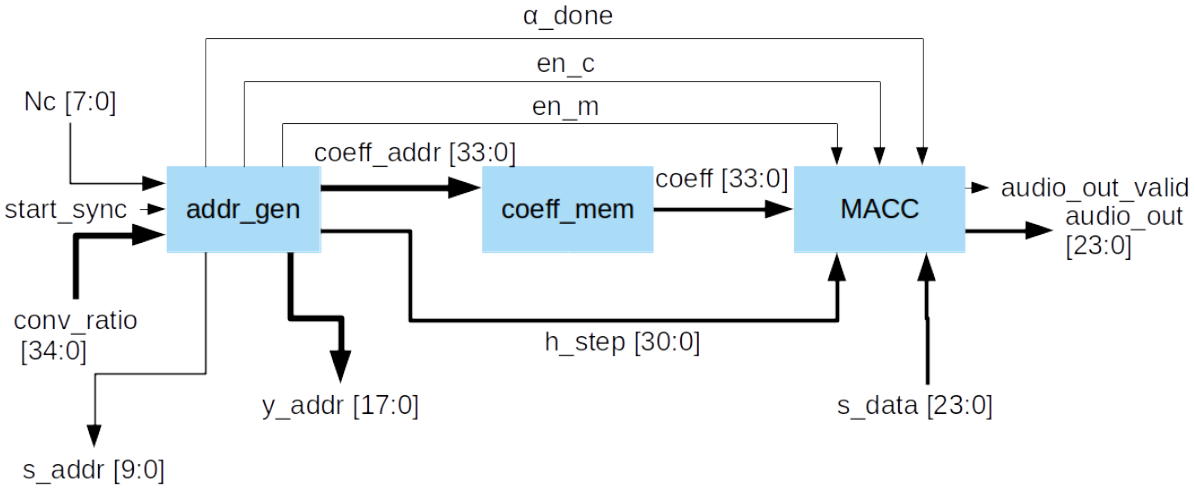


Figure 4.7: Resampler block diagram.

4.3.1 Address Generator

The address generator computes the addresses of the input samples needed to compute the output sample. Its block diagram is presented in Fig. 4.8. As explained in Section 2.2 and subsection 3.1.1, there are two main steps needed to compute an output sample: (1) to center the filter at the time of output sample to be computed; (2) to acess the input samples, compute the correspondent filter coefficients, and perform the filter convolution operation.

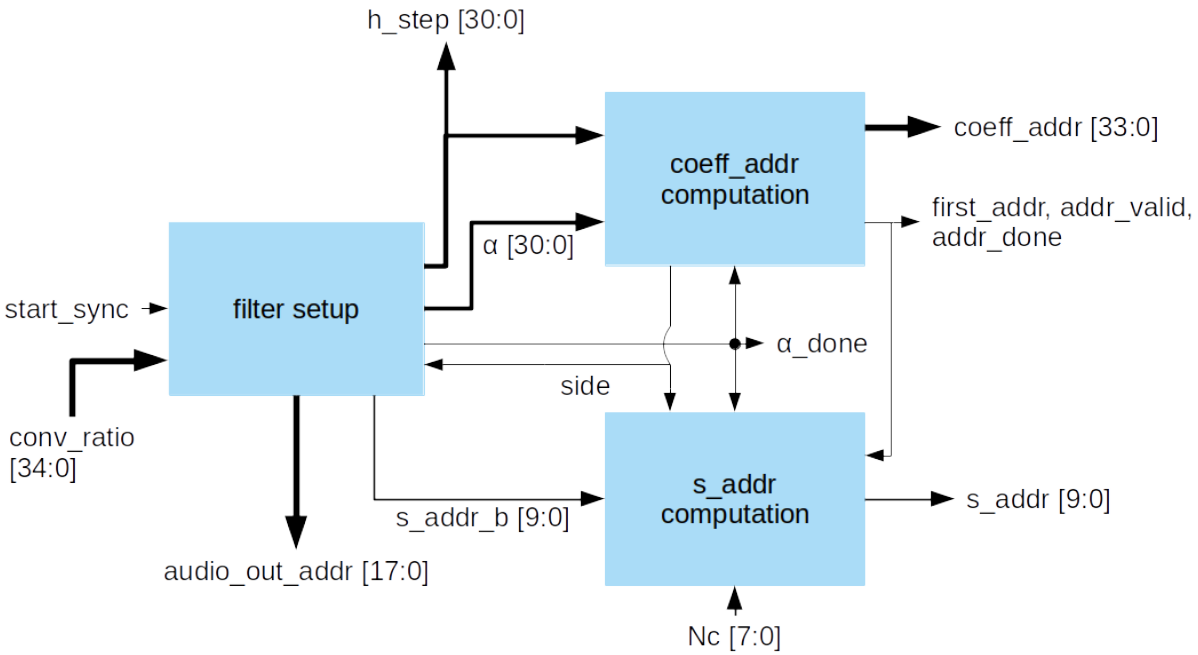


Figure 4.8: Address generator block diagram.

4.3.1.1 Filter Setup

The filter setup submodule performs the computation of three parameters: the time of the current output sample to be computed, $audio\_out\_addr$, the distance between $audio\_out\_addr$ and the two adjacent input samples, α and $1-\alpha$, and the parameter h_step, defined to impose the filter's cutoff frequency and normalize the output.

The submodule computes the time of the current output sample, $audio\_out\_addr$, through the use of an accumulator that increments $\hat{\rho^{-1}}$ at the output sample rate. While the filter setup submodule uses the fractional part of $audio\_out\_addr$ for further computations, the input sample address generator submodule uses its integer part, as is explained in Subsection 4.3.1.2.

As can be seen in the examples illustrated in 2.4 and 2.5, the filter needs to be centered at the current output sample, and the adjacent input samples need to be multiplied with the corresponding filter coefficients. The distance α allows one to obtain the address of the first input sample posterior to the current output sample, leading to the first address of the coefficient to be used for the computation of the output sample. Since the filter expands both before and after the current output sample, one needs to compute the address of the last input sample prior to the current output sample as well. The submodule computes α by assuming that there is an input sample $audio\_in[n]$ for all integer values of n. As such, the fractional part of $audio\_out\_addr$ is the value of α. Consequently, the distance between the current output sample and the last input sample prior to it is equal to $1-\alpha$. The submodule controls the selection of α or $1-\alpha$ through a multiplexer, using the current side of the filter as the selection bit.

To adapt the filter shape, ensuring that it has a cutoff frequency defined by Equation (2.3), a parameter h_step is defined by normalizing the cutoff frequecy to the input sample clock frequency:

$$h\_step = min(0.875, \rho). \tag{4.14}$$

Note that for an upsampling conversion, the value of h_step does not follow Equation (2.3), where it should be 1. Instead it is set to 0.875 to prevent operating near the Nyquist frequency with the finite selectivity filter used in practice. This change allows the converter to properly convert frequency components close to the input Nyquist frequency, $F_{in}/2$.

The block diagram of the filter setup submodule is presented in Fig. 4.9.

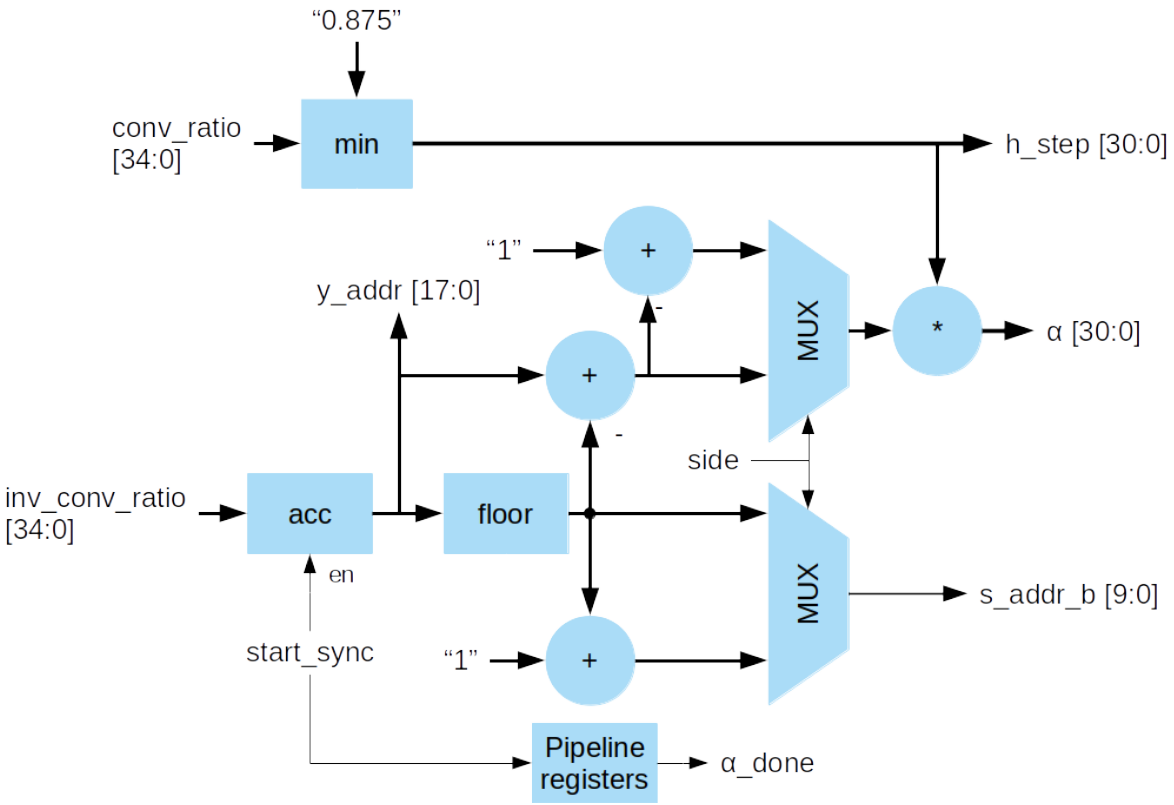


Figure 4.9: Output sample address, h_step and α computation submodule block diagram.

4.3.1.2 Input Sample Address Generator

To get the input samples' addresses, a simple counter submodule (*s_addr computation*) is used. The block diagram of the submodule is presented in Fig. 4.10. The integer part of $audio\_out\_addr$ is used as the base address of the input samples. Knowing that the distance between two consecutive input samples in the same channel is given by Equation (4.1), it suffices to accumulate this distance from the offset of the current channel to get the input sample addresses that come after (or to the right of) the output sample instant.

After the right side is completed, the left side is computed by negative accumulations from the channel offset, getting the addresses of all input samples before the output sample instant. After both sides are done, the channel offset is incremented and the process is repeated for the next channel. This

means that channels are computed one at a time, using a single multiply-accumulate unit to compute the output samples for all channels.

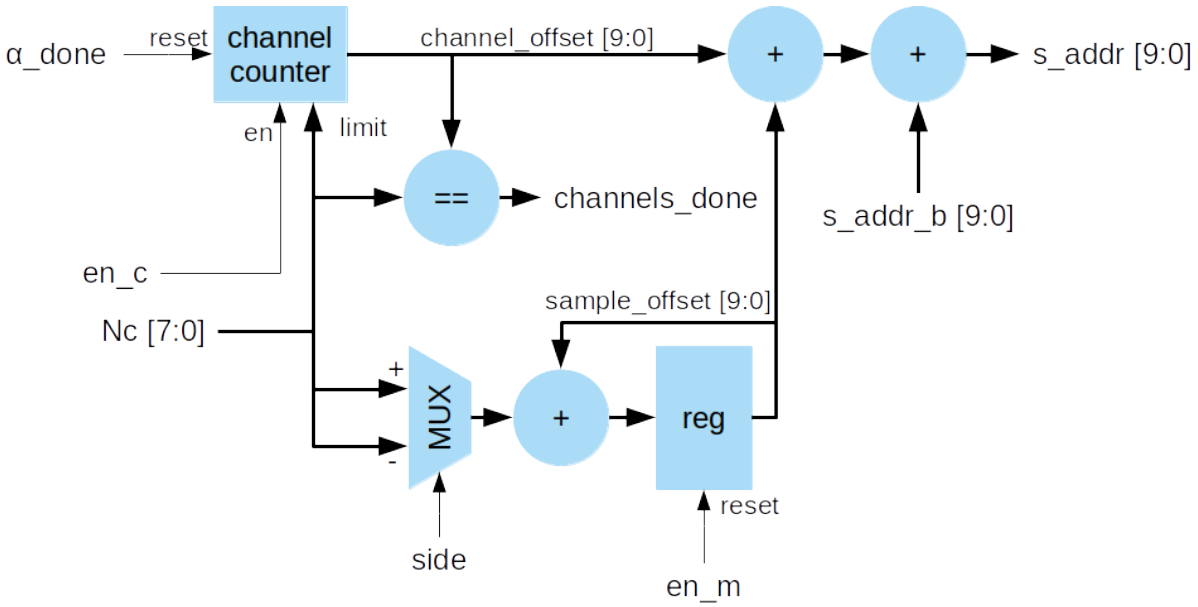


Figure 4.10: Input sample address computation submodule block diagram.

4.3.1.3 Filter Coefficients Address Generator

To get the coefficients' addresses, another accumulator is used. Its block diagram is presented in Fig. 4.11. The value of α is used as the initial coefficient address for the right-hand side samples, aligning the first input sample on this side to its filter coefficient. Since an increment of one input sample corresponds to a increment of h_step in the coefficient function argument, h_step is used as the accumulation value. These accumulations will be done until the filter is no longer defined, which happens when the accumulator reaches or overcomes the final address of the coefficients memory. After the right-hand side is done, the accumulations resumes on the left-hand side, with the complement $1 - \alpha$ as the initial coefficient address and the same increment of h_step. Considering the value of h_step and the fact that the coefficient memory contains a *sinc* function with 32 zeroes, the number of coefficients to be used, N_{coeffs}, and consequently the order of the resampling filter, is given by

$$N_{coeffs} = \frac{32}{min(0.875, \rho)}. \tag{4.15}$$

The coefficient address generator also produces three flags, $first\_addr$, $addr\_valid$ and $addr\_gen\_done$. The $first\_addr$ flag indicates the first coefficient address is valid, so the accumulator can load its initial value. The $addr\_valid$ flag signals the succeeding coefficients, so the accumulator is enabled for each of them. The $addr\_gen\_done$ flag indicates that all coefficients have been generated for the current channel sample, so one can validate the output sample present in the MACC, and proceed to do the next channel.

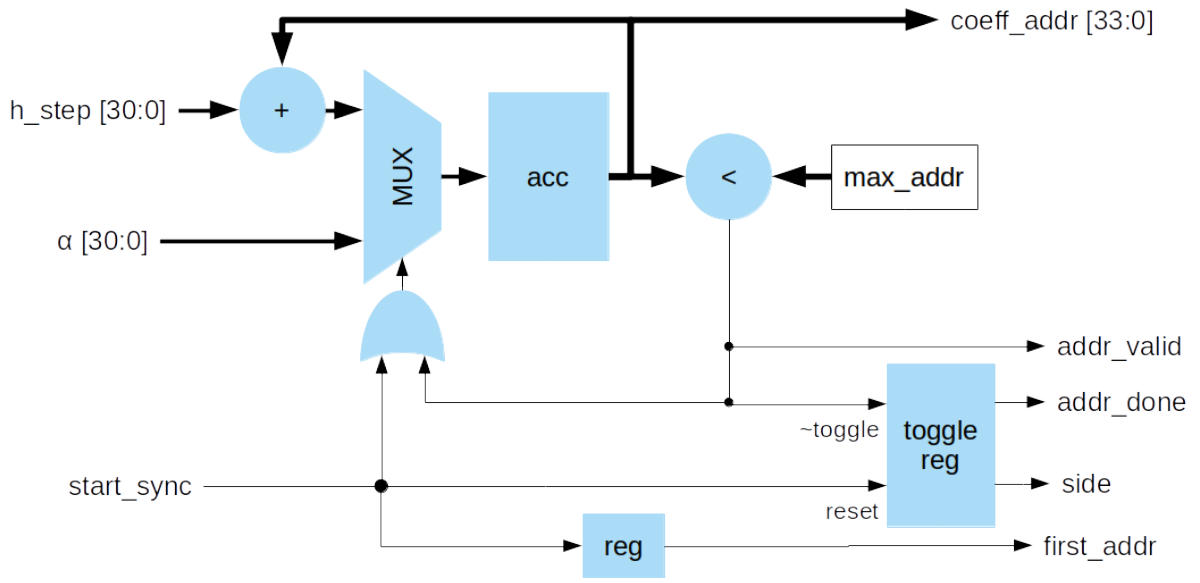


Figure 4.11: Coefficient address computation submodule block diagram.

4.3.2 Coefficient Memory

To get the filter coefficients, given by a *sinc* function, one uses a lookup table implemented with a Read-Only Memory (ROM). The block diagram of the Coefficient Memory module is presented in Fig. 4.12. If all the needed coefficients were stored in the ROM, it would need to be very large because of the required precision. To solve this problem, a linear interpolator is used. When a coefficient $h[i+\Delta]$ is needed, $h[i]$ and $h[i+1]$ exist in the lookup table, and Δ is a fractional positive distance from i, the coefficient is obtained by

$$h[i+\Delta] = h[i] + \Delta(h[i+1] - h[i]). \tag{4.16}$$

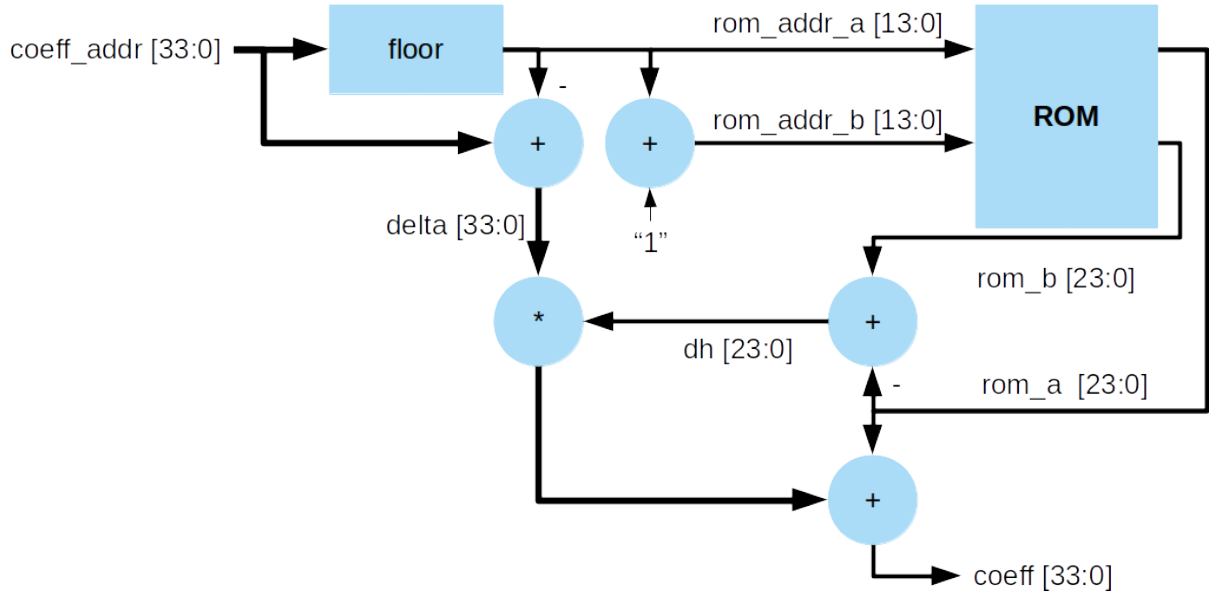


Figure 4.12: Coefficient memory submodule block diagram.

4.3.3 Multiply-Accumulator

This module implements the multiply-accumulate function as its name indicates. A block diagram of the module is presented in Fig. 4.13. Its input is the product between the coefficient and the input sample as defined by Equation (3.1). Its initial value is set by directly loading (not accumulating) the first product.

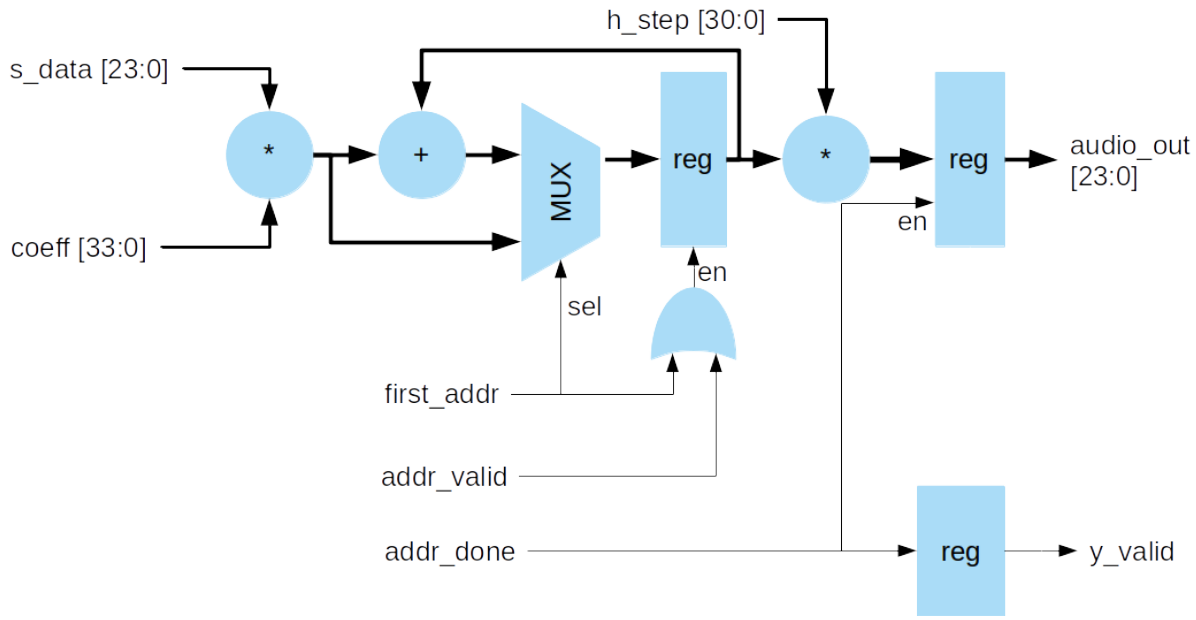


Figure 4.13: MACC block diagram.

The final register, which produces the output sample $audio\_out$, is enabled only when all accumulations have finished. This procedure ensures that $audio\_out$ never has an intermediate value. The output is updated at the output sample rate but still lives in the system clock domain. The output sample is written an output FIFO, which can be read in the $audio\_out\_wclk$ clock domain. Note that the output sample is multiplied by h_step, as the filter is normalized for unit gain in the pass-band (Equation (2.3).

4.4 Pipeline Registers

The ASRC implementation on a low cost FPGA at a system clock frequency of at least 100 MHz requires pipeline registers. These registers break critical paths, but are not shown in the diagrams for simplicity. A total of 9 pipeline registers are added: 3 in the address generator module, 4 in the coefficients ROM module, and 2 in a MACC module.

Chapter 5

Testing System

To test the results and performance of the ASRC, a system needs to be developed. This chapter describes the system used for this purpose. Section 5.1 presents the scripts developed for the input signal generation and output signal analysis, as well as the script used to generate the filter's coefficients. Section 5.2 describes the implementation in a System-on-a-Chip (SoC) environment, which enables FPGA testing.

5.1 Signal Generation and Analysis

To test the hardware developed, one needs to generate the input samples, as well as the filter's coefficients. Furthermore, with the knowledge of the input signal used, the output should be analyzed. For this matter, three *GNU Octave* scripts are used: one to generate the filter's coefficients, one to generate the input signal, and one to analyze the output signal. The scripts need to receive some parameters that need to be previously configured and adjusted, to allow an accurate analysis. Furthermore, an extra script is used to test the effect of resetting the core on the group delay.

5.1.1 Parameter Adjustment

For the generation of the input samples and analysis of the output samples, the scripts need to receive as parameters the sampling rates and the frequency of the test signal. In simulation, these parameters can be inserted by the user directly, as the ideal case can be tested. However, in FPGA implementations, the real frequency of the sampling rate clocks is limited to the physical clocking capabilities of the board.

In the case of the work performed for this thesis, the board uses two possible sampling rate master clocks, with periods of $42\,\mathrm{ns}$ and $91\,\mathrm{ns}$, as is explained in section 5.2.

To approximate the sampling rate of the generated and analyzed signals to the real frequencies of the word clocks, the frequencies are recomputed, by selecting the master clocks, f_mclk0 or f_mclk1, that allow for the real frequencies, fin_fpga and $fout\_fpga$ closest to the desired sample rates, fin and

$fout$. Afterwards, the real frequency is computed by reducing the clock frequency by an integer division value. A snippet of the *Octave* code of this adjustment is shown in Listing 5.1.

```
% real FIN
if (mod(Fin, 8000) > mod(Fin, 11025)) % check which mclk to use
  % use MCLK1
  DIV_IN = round(f_mclk1/Fin);
  fin_fpga = f_mclk1/DIV_IN;
else
  % use MCLK0
  DIV_IN = round(f_mclk0/Fin);
  fin_fpga = f_mclk0/DIV_IN;
endif

% real FOUT
if (mod(Fout, 8000) > mod(Fout, 11025)) % check which mclk to use
  % use MCLK1
  DIV_OUT = round(f_mclk1/Fout);
  fout_fpga = f_mclk1/DIV_OUT;
else
  % use MCLK0
  DIV_OUT = round(f_mclk0/Fout);
  fout_fpga = f_mclk0/DIV_OUT;
  endif
```

Listing 5.1: Sample rate adjustment script

To increase the precision of the analysis, it is ideal if the frequency of the test signal, $Ftest$, is a divisor of the output sampling rate. Due to the adjustment of both sampling rates to the real frequencies of the word clocks, the value of $Ftest$ is also adjusted, as

$$Ftest\_adjusted = fout\_fpga \times round(\frac{Ftest}{fout\_fpga}). \tag{5.1}$$

To allow an accurate analysis, the number of samples should be high enough to allow a proper frequency resolution of the fast-Fourier transform, as well as guarantee that the analysis has enough samples to attenuate the negative effects of windowing the output, as is explained in subsection 5.1.4. Furthermore, the number of samples should ensure that the analyzed signal is an integer number of periods, to avoid spectral scattering. Finally, the input signal should have some extra samples to ensure that there are enough to account for the ASRC's settling time, as well as to ensure that there are enough valid output samples, as the initial output samples are invalid, since they are computed using whatever is in the memory before an input sample is written. As such, the script computes the number of periods

to analyze. $PERIOD\_ANALYSIS$, as well as the number of input samples to be generated, TD. Listing 5.2 shows an An *Octave* snippet of the code that performs this computation. Note that the listing defines a minimum number of 120 signal periods and 11000 samples to analyze. Furthermore, for multi-channel tests, each channel changes the frequency of the input signal, and the minimum number of samples applies to the worst case.

```
1 PERIOD_ANALYSIS = 120; %number of periods to analyze
2 while((PERIOD_ANALYSIS*Fout/Ftest) < 11000)
3    PERIOD_ANALYSIS = PERIOD_ANALYSIS+1;
4 endwhile
5
6 Ftest_delta = 3*Ftest_param/200 # change in FTest per channel
7 TD = floor(fin_fpga/(Ftest-Ftest_delta*(NChannels-1))*(
     PERIOD_ANALYSIS+50)+2^(SAMP_BUF_W)/NChannels); %test duration in
     input samples
```

Listing 5.2: Sample rate adjustment script

5.1.2 Filter Coefficients Generation

To generate the coefficients, the script receives three values: $filter\_nzeros$, $filter\_nfrac$, and H_bits, which give the number of zeroes of the sinc function, the number of address bits to address the filter's memory, and the number of bits to quantify the value of the coefficients, respectively. A snippet of the *Octave* code of the script is shown in Listing 5.3.

```
1  function filter_tab = getCoeffROM(filter_nzeros, filter_nfrac,
      H_bits)
2     filter_t_res = 1/2**filter_nfrac;
3     filter_t = 0: filter_t_res : filter_nzeros/2 -  filter_t_res; %
         normalized positive time axis
4     window = kaiser(2*length(filter_t)-1, 14.4)';
5     window = window(length(filter_t): 2*length(filter_t)-1); %take
         half window
6     filter_tab = (2^(H_bits-1)-1)*sinc(filter_t) .* window; %build
         one sided impulse response filter table
7     filter_tab = do_2s_comp(round(filter_tab), H_bits);
8     return
9   endfunction
```

Listing 5.3: Coefficients generation script

With these values, the script generates an ideal sinc function and truncates it using window function. Due to its configurability, and to the fact that it is already included in *Octave*'s *Signal* package, a *Kaiser* window is used. The value chosen for the parameter (14.4) is the one that led to the fulfillment of the specification for most tests, as can be seen in chapter 6.

Regarding the values of the received parameters, for the tests done in this thesis, $filter\_nzeroes$, $filter\_nfrac$ and H_bits have are equal to 32, 10 and 24, respectively.

Since the result is a symmetric function, only half the window values are taken to fill the coefficient memory.

5.1.3 Input Signal Generation

To generate the input samples, the script generates a sinusoidal signal and samples it at the input sample rate. To do this, the script needs the sinusoid's frequency and magnitude $Ftest$ and $magtest$, the duration of the signal in input samples TD, the number of bits of each sample H_bits, the input signal's sample rate Fin, and the number of channels $Nchannels$. With these values, the script generates a vector with all input samples. Regarding the amplitude of the input signal, it is slightly below unity to prevent overflows during processing. An example script is outlined in Listing 5.4.

```
1  function x = genInput(Ftest, magtest, TD, H_bits, Fin, Nchannels)
2      %compute test duration in seconds
3      td = TD/Fin;
4      %compute input wave x
5      t_in = 0: 1/Fin : td-1/Fin; %input time axis
6      % create final vector for all audio channels
7      x = zeros(1,NChannels*length(t_in));
8      for ch =1:NChannels
9        % create test wave with MagTest dBFS input gain for each
             channel
10       Ftest_ch = fout_fpga/round(fout_fpga/(Ftest_param-Ftest_delta
             *(ch-1)));
11       x_ch = sin( 2*pi*Ftest_ch*t_in + pi/(4*ch)) * 10^(MagTest/20)
             ;
12
13       % interleave audio channels
14       x(ch:NChannels:end) = x_ch;
15     endfor
16
17     %quantize x to 1Q23
18     x = round( x*2^(H_bits-1) );
19     %do 2s complement
```

```
20        x = do_2s_comp(x, H_bits);
21        return
22    endfunction
```

Listing 5.4: Input generation script

5.1.4 Output Signal Analysis

To test the output signal, a script which computes its spectrum and the total harmonic distortion plus noise ratio is run. The script receives as inputs the array of output samples y, obtained by reading a file produced by the simulation, the sampling rate Fs, the magnitude of the original signal $MagTest$, and the frequency of the original signal f.

To prevent spectrum artifacts, an integer number of periods is extracted from y and used by the script. Additionally, a Blackman-Harris window is applied to the signal to analyze, leading to a further reduction of a spread spectrum. It is, however, important to note that, if the number of frequency bins of the spectrum (that is related to the number of samples) is not enough, the window's frequency response will show its effects, affecting negatively the results of the analysis. As such, it is important to guarantee that there are enough bins to ensure an accurate analysis.

To obtain the signal's spectrum, a fast Fourier transform is used. Afterwards, the bin with the highest power is found, and treated as the signal's frequency bin. Then, the total harmonic distortion plus noise ratio is computed by summing the power value of every bin which is not deemed to belong to the original sinusoidal signal. The script allows a small tolerance around the signal bin's frequency, to compensate for the difficulty to avoid scattering when computing spectra. The bins of the tolerance area are considered signal bins. An example script is presented in Listing 5.5.

```
1   function thdn = getTHDN(y, Fs, MagTest, f)
2
3     Nypts = length(y);
4     f_axis = 0: Fs/Nypts :Fs/2-Fs/Nypts;
5
6     % compute bandwidth surrounding the fundamental
7     sig_bin_width = 8;
8
9     % GET TONE MAG
10    YM=abs(fft(y))/Nypts;
11
12    [maxVal, tone_idx] = max(YM(1:Nypts/2));
13
14    signalBins = tone_idx-floor(sig_bin_width/2):tone_idx+floor(
          sig_bin_width/2);
```

```
15      signalBins = signalBins(signalBins>0); %remove bins lower than 0
16      signalBins = signalBins(signalBins<=Nypts/2); %remove bins over
            FFTpoints/2
17      s = norm(YM(signalBins));
18
19      mag = 20*log10(2*s); % multiplied by 2 to consider negative half
            of FFT
20
21      % GET THDN
22      %eliminate spectral leakage by windowing
23      window = blackmanharris(Nypts);
24      yw = y.* window;
25
26      Y=abs(fft(yw))/Nypts;
27
28      [maxVal, tone_idx] = max(Y(1:Nypts/2));
29
30      signalBins = tone_idx-floor(sig_bin_width/2):tone_idx+floor(
            sig_bin_width/2);
31      signalBins = signalBins(signalBins>0); %remove bins lower than 0
32      signalBins = signalBins(signalBins<=Nypts/2); %remove bins over
            FFTpoints/2
33      s = norm(Y(signalBins));
34      noiseBins = 1: Nypts/2;
35      noiseBins(signalBins) = []; %excludes signal bins
36      noiseBins(1: floor(sig_bin_width/2)) = []; %excludes DC
37      n = norm(2*Y(noiseBins));
38      thdn = 20 * log10(n);
39
40      return
41  endfunction
```

Listing 5.5: Output analysis script

It is important to notice that, in this case, both the THD+N and the signal's frequency should be analyzed. In the context of the work made for this thesis, the specification consists on having a THD+N smaller than $-130\,\text{dB}$, and a signal frequency equal to $Ftest$, with a precision of around a unit of Hertz.

5.1.5 Effect Of Resetting On The Group Delay

To ensure that the group delay of the output audio stream suffers close to no change, two similar simulations of the ASRC are run. To simulate the hardware, Cadence's *NCsim* simulator is used. The use of the simulator ensures that the simulations do not have any difference regarding clock phases or other signals that may vary on different runs of the same test.

In the first run, the simulation behaves similarly to the tests performed in the FPGA, with the difference that the samples are not transferred through an Ethernet module. The second run is similar to the first one, with the difference that the reset signal of the ASRC is activated on a pulse, in the middle of the conversion. This causes the synchronization of the sample rate conversion ratio to be rerun, leading to a possible difference in the difference between the data write and read pointers, consequently changing the group delay of the ASRC.

The difference of the group delay of the two tests is then compared, by extracting the final periods of the output signals, are computing the phase between them. The computation of the approximate phase between two signals, ϕ, is done through the method explained in [21], being expressed as

$$\phi = \arccos \frac{\overrightarrow{y_1} \cdot \overrightarrow{y_2}}{|y_1||y_2|}, \tag{5.2}$$

where y_1 and y_2 are vectors containing each sample as an element. It is important to note that the approximate is only accurate if both signals are sinusoidal, and the number of samples is large enough.

A snippet of the *Octave* script used to compute the phase between the output signal of the first test, y_full, and the output signal of the second test, y_rst is shown in Listing 5.6.

```
1  % OBTAIN SAMPLES AFTER RESET
2  last_zero_idx = 0;
3  if size(y_rst) < 3
4     printf("Sample Files are too small\n");
5     exit();
6  endif
7  for idx = 3:size(y_rst)
8    if((y_rst(idx-2)==y_rst(idx-1)) && (y_rst(idx-1)==y_rst(idx)))
9      last_zero_idx = idx;
10   endif
11 endfor
12
13 last_zero_idx = last_zero_idx+ceil(2^(SAMP_BUF_W)*(Fout_fpga/
       Fin_fpga));
14
15 %% get waves from last_zero_idx
16 y_full_after_rst = y_full(last_zero_idx+1:end);
17 y_rst_after_rst = y_rst(last_zero_idx+1:end);
```

```
18
19  %% find phase difference
20
21  %% approximate method:
22  %% Good approximation for long vectors size >> 1
23  dot_product = dot(y_full_after_rst, y_rst_after_rst);
24  norm_product = (norm(y_full_after_rst)*norm(y_rst_after_rst));
25  phase_shift_rad = acos(dot_product/norm_product);
26
27  diff_seconds = phase_shift_rad/(2*pi*Ftest); % seconds
28
29  out_samples_diff = diff_seconds*Fout_fpga; % SAMPLE = s/(s/SAMPLE)
```

Listing 5.6: Sample rate adjustment script

5.2 IOB-SoC Hardware Implementation

IOB-SoC [22], an open-source system-on-a-chip (SoC) platform, tests the implementation of the ASRC in FPGA,. The block diagram of the SoC is shown in Fig. 5.1.

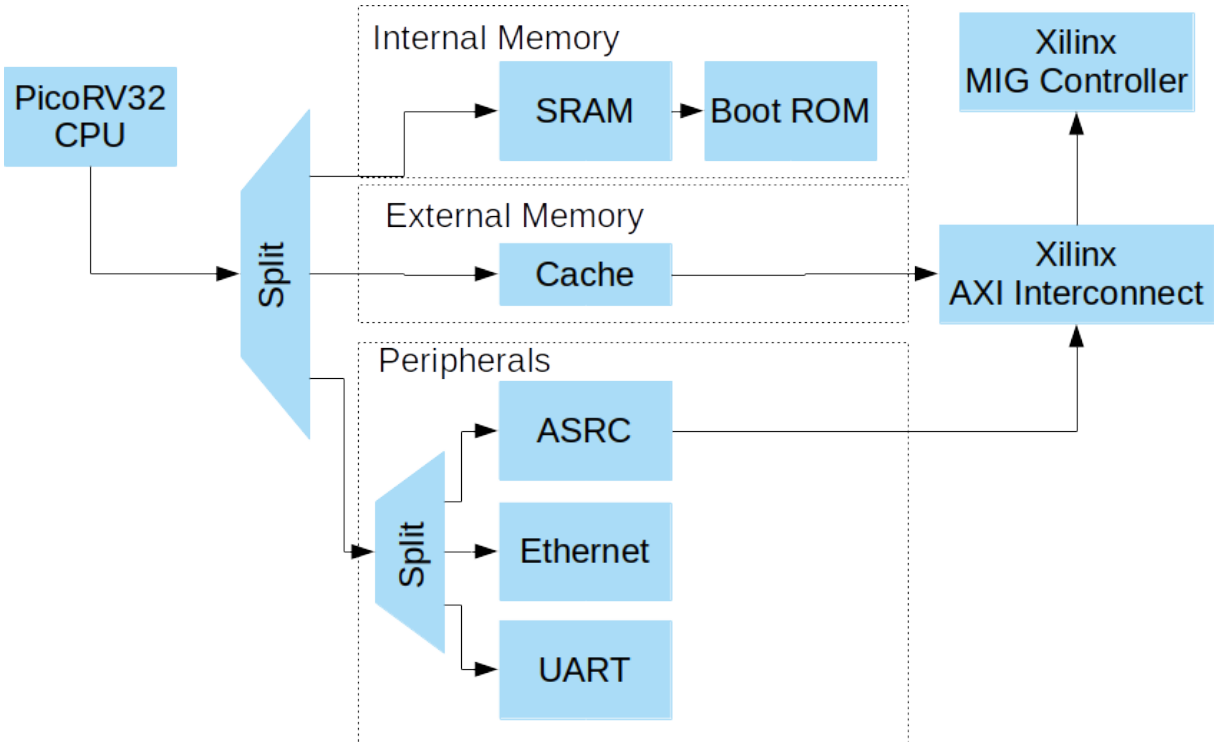


Figure 5.1: IOB-SoC block diagram.

To control the system, the PicoRV32 [23], a RISC-V soft processor is included. This central processing unit (CPU) includes integer, multiply and atomic instruction set extensions, and a RISC-V GNU compiler toolchain is used to compile C programs to run on it [24]. The CPU is connected to an internal static random access memory (SRAM), where the instructions and local variables are stored, as well as a boot read-only memory (ROM) which the CPU initially accesses to receive the program through its universal asynchronous receiver-transmitter (UART) module, and store it in the SRAM.

Additionally, an external Double Data Rate (DDR) memory is connected to a cache module, that is accessible by the CPU. The CPU is also connected to three more peripherals: the ASRC, an Ethernet module, and a UART module. The description of the Ethernet and UART modules is done in sections 5.2.1.2 and 5.2.1.1, respectively. The description of the cache can be found in [25].

The memory map of the memories and peripherals is shown in Table 5.1. The first address is used as a base, used to select the memory or peripheral. The addition of an offset allows the selection of a specific register.

Memory/Peripheral	**Address**
Internal Memory	0x0000 0000 - 0x1FFF FFFF
Boot Controller	0x2000 0000 - 0x3FFF FFFF
UART	0x4000 0000 - 0x4FFF FFFF
ASRC	0x5000 0000 - 0x5FFF FFFF
Ethernet	0x6000 0000 - 0x6FFF FFFF
Cache/External Memory	0x8000 0000 - 0xFFFF FFFF

Table 5.1: IOB-SoC memory map

To generate the sample rate clocks, a *Xilinx*'s Memory Interface Generator (MIG) generates two more independent clocks. Due to hardware limitations imposed by the MIG, the clocks have periods of $42\,\text{ns}$ and $91\,\text{ns}$. The first clock, $audio\_mclk\_0$, with a period of $42\,\text{ns}$, is mostly used for frequencies that are multiple of $8\,\text{kHz}$, as it frequency is approximately equal to $24\,\text{MHz}$. As for the other clock, $audio\_mclk\_1$, its period is approximately equal to $11\,\text{MHz}$, and it is mostly used for frequencies that are multiple of $11.025\,\text{kHz}$. It is important to note that the scripts adapt the frequencies to an approximate value, due to the limitations imposed by the period of the clocks.

5.2.1 Peripherals

The connection to the peripherals and memory is done through a native master-slave interface, where the CPU is always the master. The signals used for the interface, and their descriptions, are shown in Table 5.2.

Name	Direction	Width	Description
valid	input	1	Native CPU interface valid signal
address	input	ADDR_W	Native CPU interface address signal
wdata	input	WDATA_W	Native CPU interface data write signal
wstrb	input	DATA_W/8	Native CPU interface write strobe signal
rdata	output	DATA_W	Native CPU interface read data signal
ready	output	1	Native CPU interface ready signal

Table 5.2: CPU native slave interface signals

5.2.1.1 UART

The UART module is used to transfer the firmware to the SoC during the boot execution, as well as allow the display of debug messages sent by the SoC to the computer. The connection between the module and host computer is done through an universal serial bus (USB) interface.

To allow a proper communication with the computer, a baud rate compatible to the USB interface is configured. The baud rate is the frequency of bits transferred per second. In the case of the work performed for this thesis, the baud rate used is equal to 115200. It is important to note that the baud rate leads to transfers with a frequency in the same order as the sample rates. As such, this module should not be used during the ASRC's execution, as it will allocate the CPU for too much time.

To control the module on the side of the SoC, C drivers are used, working in a similar way to *print* and *scan* instructions from the *stdio.h* C library. On the side of the computer, a C program is used to send the firmware, as well as read data from the SoC and print it to the screen.

5.2.1.2 Ethernet

The Ethernet module is used to transfer the input and output audio samples from the computer to the SoC, as well as the input and output sample rates to be configured. Considering that an audio file has a size that ranges from dozens of kB to units of MB, the use of the Ethernet module instead of the UART is essential, as a UART transfer would bottleneck the program's execution.

The Ethernet module used works on raw sockets, without the use of an internet protocol. The data frame's structure is shown in Fig. 5.2.

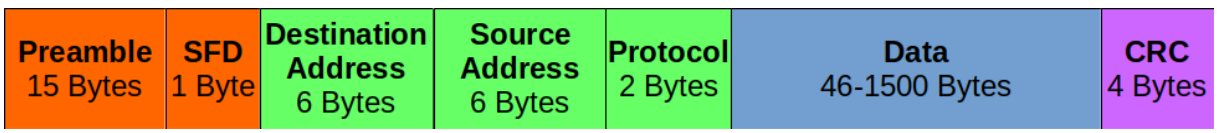


Figure 5.2: Ethernet data frame format.

The preamble is composed by a word of 15 bytes with alternating zeroes and ones. The start frame delimiter (SFD) section should have the value equal to 0xD5. These two sections make the frame's preamble.

The destination and source addresses used correspond to the MAC addresses of the FPGA and the host computer. The protocol section indicates the Ethernet protocol used. In the case of this module,

the value of this section is equal to 0x8000, which corresponds to an IPv4 datagram protocol. These three sections correspond to the frame's header.

The data section contains the data that the SoC or the host computer need to send. The size of the data section is variable, and needs to be configured in both the FPGA and the host computer's side, in order to be the same.

To validate the the results received, the cyclic redundancy check (CRC) section of the data frame is checked, and a message is printed if the CRC is incorrect. The value of the CRC is obtained through a checksum of the data sent, and a transmission error is detected if the computed value at the receiver's side does not correspond to the value sent through the Ethernet module.

On the side of the SoC, C drivers are used to control the module, allowing the easy transfer of multiple data frames. On the side of the computer, a python script is used.

5.2.1.3 ASRC's Wrapper

To adapt the ASRC's interface to the native interface of the SoC, a wrapper is added. The wrapper not only converts the SoC's native interface to the ASRC's interface, but also adds some control elements to allow adaptability to different configurations. As such, the wrapper can be split into four blocks:

- software accessible registers: to control the wrapper and the ASRC, the CPU directly accesses a bank of registers, that are mapped in the SoC's memory, being therefore accessible by the firmware. This bank also includes some read-only registers, used to check the ASRC's status. The list of software accessible registers, and their details, is shown in Tables 5.4, 5.5 and 5.6. The synthesis parameters used to define the sizes of some of the registers are shown in Table 5.3.

- a clock selector and divider: to allow multiple combinations of input and output sampling rates, two clocks, $audio\_mclk\_0$ and $audio\_mclk\_1$ are generated in the MIG, and used as inputs of the wrapper. Two clock selectors, controlled by the value of two software accessible registers, are used to determine which of the two clocks shall be used as the input and output master clocks, $audio\_in\_mclk$ and $audio\_out\_mclk$. Additionally, to generate the input and output word clocks, $audio\_in\_wclk$ and $audio\_out\_wclk$, two clock dividers are used, with the division values being determined by the value of two software accessible registers. These dividers are achieved with a counter that counts at every cycle of the respective audio master clock, and resets whenever the division value is reached, as well as a register that compares if the counted value reached half of its target. The result of the division is a word clock with a duty cycle of 50% (with an error of half master clock cycle).

- an input and an output buffer: to temporarily store the input and output samples, as well as guarantee that they are written at the right timings, two asynchronous first in first out (FIFO) blocks are used, one for the input and one for the output. These FIFOs not only guarantee that the input and output sample signals are properly synchronized from the system's clock to the audio master clocks, but also allow the CPU to process other instructions while there are still samples in the input buffer and slots in the output buffer. To control the write of the input in the ASRC, a counter

is used. The counter counts at every input audio master clock cycle, as long as the counted value is less than the number of channels configured in the corresponding software accessible register. While the counted value is less than the number of channels, the wrapper keeps $audio\_in\_valid$ active, allowing the samples to be written at every positive edge of $audio\_in\_mclk$.

- a direct memory access (DMA) block: to write or read a burst of samples, allowing a faster data transfer, an AXI DMA connects the input and output buffers to the MIG. This allows a transfer of approximately one sample per system clock cycle. The execution and configuration of the DMA is done through the value of six software accessible registers.

The block diagram of the wrapper is shown in Fig. 5.3.

Parameter	Default Value	Description
SAMP_W	24	Audio sample width
FIFO_IN_ADDR_W	7	Input audio FIFO size (log2)
FIFO_OUT_ADDR_W	7	Output audio FIFO size (log2)
DMA_ADDR_W	30	Addressable memory space (log2)

Table 5.3: ASRC testing system's synthesis parameters.

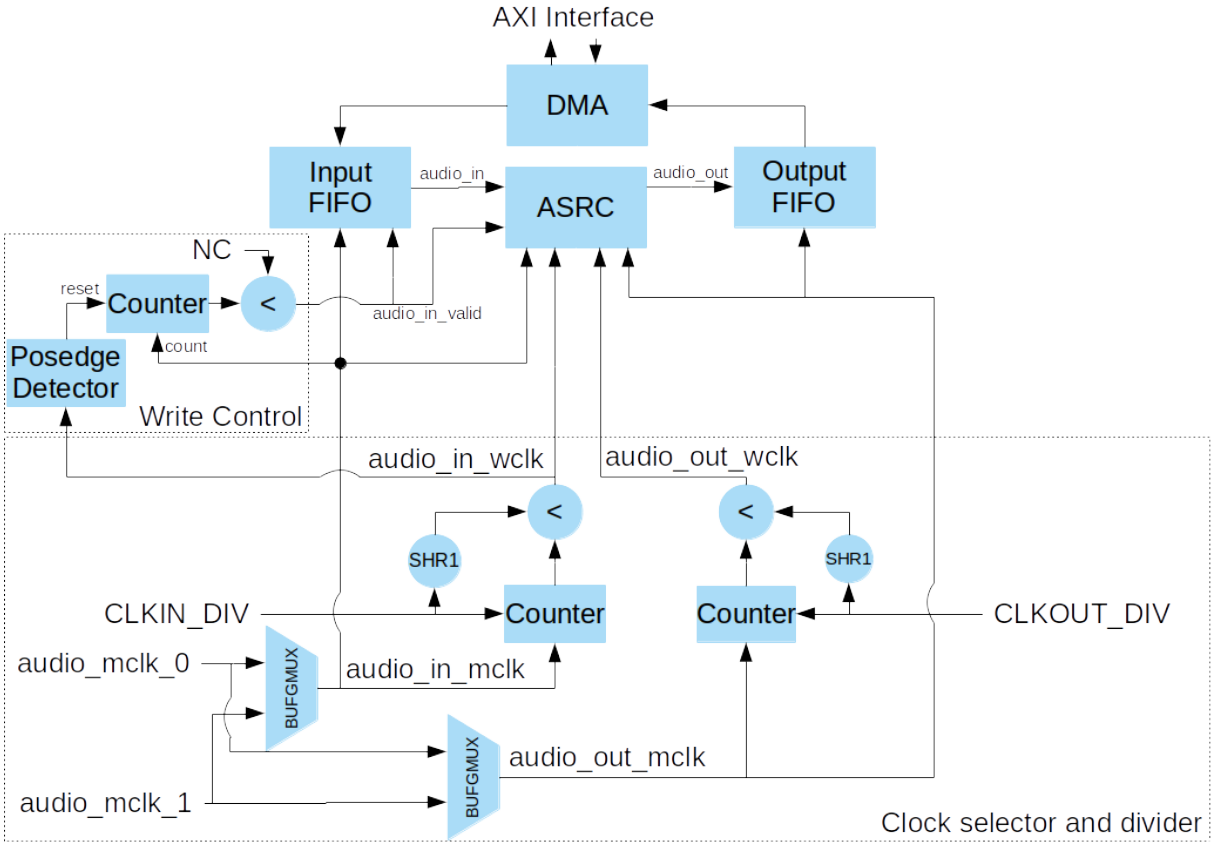


Figure 5.3: ASRC's wrapper block diagram.

Name	R/W	Address	Bits	Initial Value	Description
ASRC_NC	W	0x00	7:0	1	Number of data channels.
ASRC_SOFT_RESET	W	0x04	0:0	0	Resets the sample rate converter.
ASRC_DATA_IN	W	0x08	SAMP_W-1:0	0	Data to input to sample rate converter.
ASRC_WR	W	0x0c	0:0	0	Writes an input sample to the sample rate converter.
ASRC_RD	W	0x10	0:0	0	Reads an output sample from the sample rate converter.
ASRC_DATA_OUT	R	0x14	SAMP_W-1:0	0	Output data from sample rate converter.
ASRC_ERROR	R	0x18	2:0	0	Error bits (2) Audio out buffer— (1) PTR_DIFF— (0) NChannels
ASRC_CLKIN_SEL	W	0x1c	0:0	0	Select mclk_0 (0) or mclk_1 (1) as input audio clock.
ASRC_CLKOUT_SEL	W	0x20	0:0	1	Select mclk_0 (0) or mclk_1 (1) as output audio clock.
ASRC_CLKIN_DIV	W	0x24	12:0	500	Number of input audio clock periods per wclk_in period minus 1 maximum value is 4095.
ASRC_CLKOUT_DIV	W	0x28	12:0	1000	Number of output audio clock periods per wclk_out period minus 1 maximum value is 4095.
SYNC_CYCLES	W	0x2c	31:0	0	Number of system clock cycles to wait for synchronization
RO_METER_SYNC	R	0x30	0:0	0	ASRC has determined the conversion ratio and is ready to convert (1) or not (0)
RO_METER_RO_LOW	R	0x34	31:0	0	Sample rate conversion ratio (least significant bits)
RO_METER_RO_HIGH	R	0x38	2:0	0	Sample rate conversion ratio (most significant bits)
RO_METER_RO_INV_LOW	R	0x3c	31:0	0	Inverse of sample rate conversion ratio (least significant bits)
RO_METER_RO_INV_HIGH	R	0x40	2:0	0	Inverse of sample rate conversion ratio (most significant bits)

Table 5.4: Software accessible registers for the ASRC.

Name	R/W	Address	Bits	Initial Value	Description
INFIFO_FULL	R	0x44	0:0	0	Input sample fifo full flag.
INFIFO_EMPTY	R	0x48	0:0	0	Input sample fifo empty flag.
INFIFO_LEVEL	R	0x4c	FIFO_IN_ADDR_W-1:0	0	Number of Input samples in fifo.
OUTFIFO_FULL	R	0x50	0:0	0	Output sample fifo full flag.
OUTFIFO_EMPTY	R	0x54	0:0	0	Output sample fifo empty flag.
OUTFIFO_LEVEL	R	0x58	FIFO_OUT_ADDR_W-1:0	0	Number of output samples in fifo.

Table 5.5: Software accessible registers for the input and output buffers.

Name	R/W	Address	Bits	Initial Value	Description
INDMA_ADDR	W	0x5c	DMA_ADDR_W-1:0	0	Memory address for memory to sample rate converter transfers.
INDMA_LEN	W	0x60	7:0	0	Burst length for memory to sample rate converter transfers.
INDMA_RUN	W	0x64	0:0	0	Starts the DMA for memory to sample rate converter transfers.
INDMA_READY	R	0x68	0:0	0	DMA is ready to start an input sample RUN (1) or not (0).
OUTDMA_ADDR	W	0x6c	DMA_ADDR_W-1:0	0	Memory address for sample rate converter to memory transfers.
OUTDMA_LEN	W	0x70	7:0	0	Burst length for sample rate converter to memory transfers.
OUTDMA_RUN	W	0x74	0:0	0	Starts the DMA for sample rate converter to memory transfers.
OUTDMA_READY	R	0x78	0:0	0	DMA is ready to start an output sample RUN (1) or not (0).
OUTFIFO_SWITCH	W	0x7c	0:0	1	Disable or enable getting samples from ASRC
PTR_DIFF_SWITCH	W	0x80	0:0	1	Disable or enable sending samples to data_mem (stops x_addr)

Table 5.6: Software accessible registers for the DMA.

5.2.2 Firmware

To allow the execution of multiple tests of the ASRC, a testing firmware is developed. The firmware allows the CPU to configure the ASRC, and send and receive audio samples from it.

To configure the ratio estimator module, a synchronization time of $20\,\mathrm{ms}$ is chosen. Assuming a system clock with a frequency equal to $100\,\mathrm{MHz}$, the internal counter of the module needs to count until $\frac{20 \times 10^{-3}}{10 \times 10^{-9}} = 2 \times 10^{6}$.

A flowchart of the firmware developed is shown in Fig. 5.4.

Initially, the CPU waits for the host computer to send the input and output sampling rates used for the test. With this information, it configures the ASRC's clock selection and divisors, used by the core's wrapper. Afterwards, it receives the input samples, and stores them in the external memory. After receiving every sample and finishing every configuration, the ASRC is reset, while the configurations are kept the same.

After a reset, the ASRC starts the computation of the conversion ratio. As such, to avoid wasting input samples, the CPU waits for the computation to finish, as the ASRC does not convert while the computation is being performed. After the computation is finished, the CPU reads and stores the value of the conversion ratio computed, as well as its inverse.

When the ASRC starts the conversion, the CPU ensures that it is always properly working, by keeping at least half of the input buffer full with valid input samples, and half of the output buffer empty. For both cases, the CPU checks the number of samples in the FIFO. The DMA is commanded to transfer as many input samples as needed to fill the input buffer, if the buffer is at less than half of its capacity. Similarly, the DMA is also commanded to transfer as many output samples from the output buffer to the external memory as valid samples in the buffer, if it is at more than half of its capacity.

After every input sample is sent, and the input buffer is empty, the DMA is commanded to transfer all remaining samples in the output buffer to the external memory, and the conversion is finished. As such, every output sample in the external memory is sent to the computer, using the Ethernet module.

As a debug measure, the value of the conversion ratio, and its inverse, obtained before, are printed, by using the UART to send the debug message to the computer, where it is displayed.

Regarding the generation of the input samples, and analysis of the output samples, these processes are performed in the host computer, by using the *Octave* scripts shown in Section 5.1.

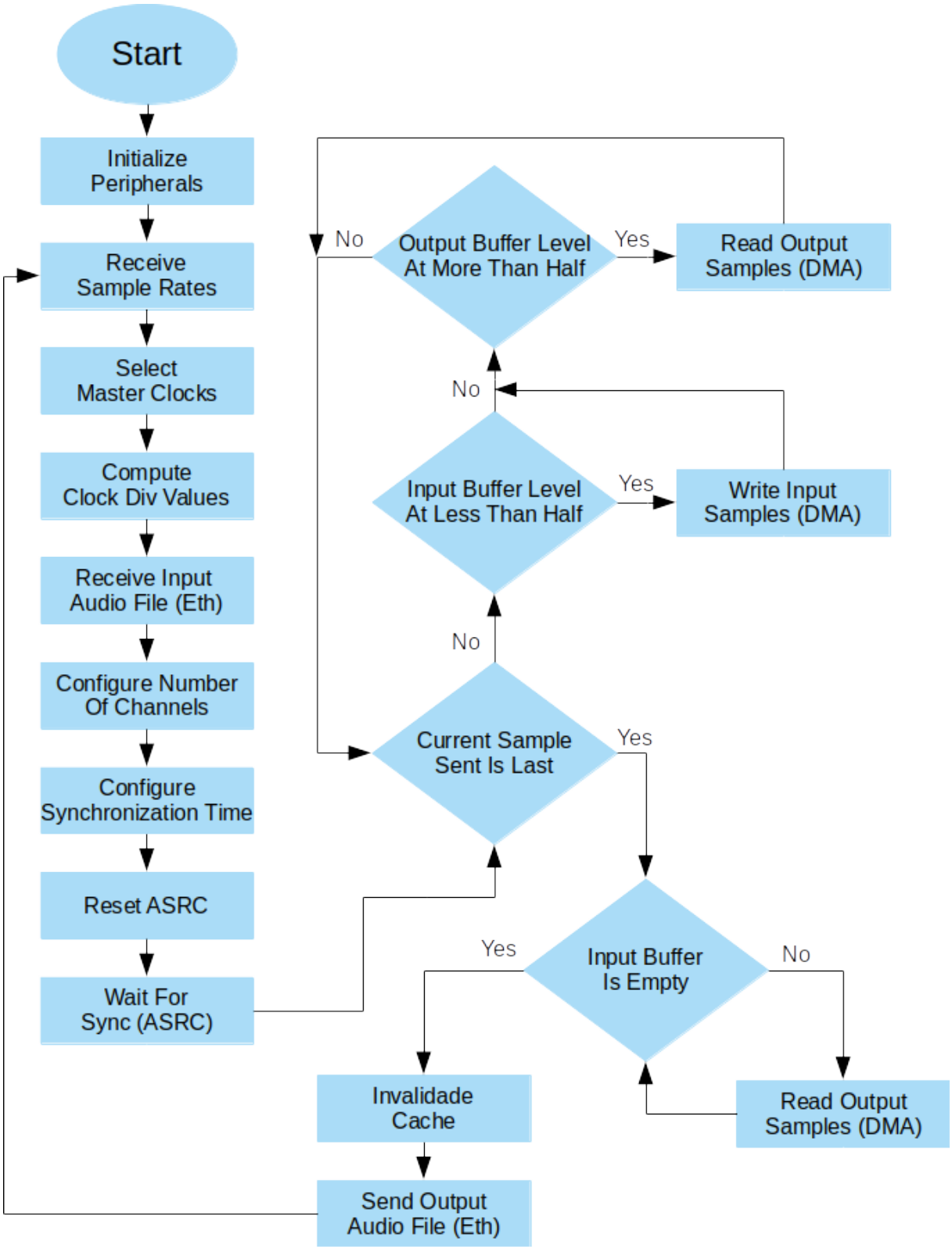


Figure 5.4: ASRC testing firmware flowchart.

Chapter 6

Results

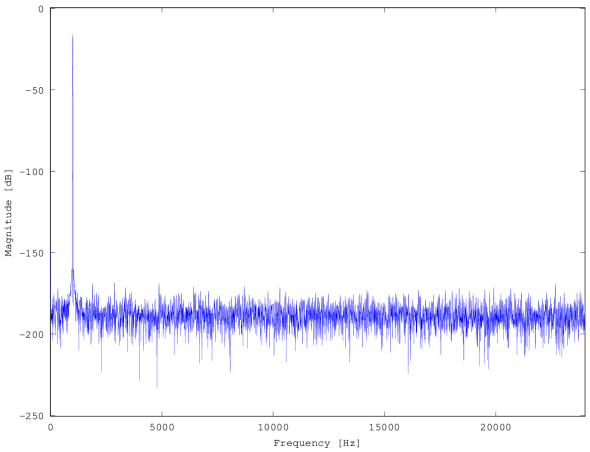
To validate the Asynchronous Sample Rate Converter design, a rigorous testing suite has been applied to it, based on similar tests that can be found in [1], [2] and [3]. This chapter presents the parameters used in each test and the results obtained. All tests have been applied to an FPGA implementation of the design, embedded in the SoC described in Section 5.2, and run on a Kintex Ultrascale FPGA (XCKU040-FBVA676-1-C) device.

6.1 Fast Fourier Transform Setup

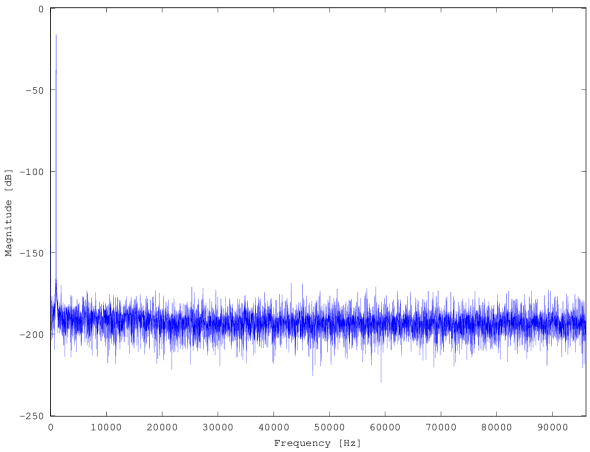
Before presenting the experimental results, it is important to explain that the Fast Fourier Transform method has been used to analyse the output signal spectrum. The output signal is sent to the PC by the SoC running on the FPGA via Ethernet. Then the signal is analysed using the Octave software package.

The plots of the FFT of the output signal for some conversions are shown in Fig. 6.1 and 6.2 as examples. The input used in all conversions is a sine wave with a magnitude of $-1\,\mathrm{dBFS}$ and a frequency of $1\,\mathrm{kHz}$. The FFT is computed after windowing the output, to avoid the effect of spread spectrum. It is also important to note that only half of the FFT is being shown, as it has a symmetric counterpart from $-F_{sout}/2$ to 0 Hz.

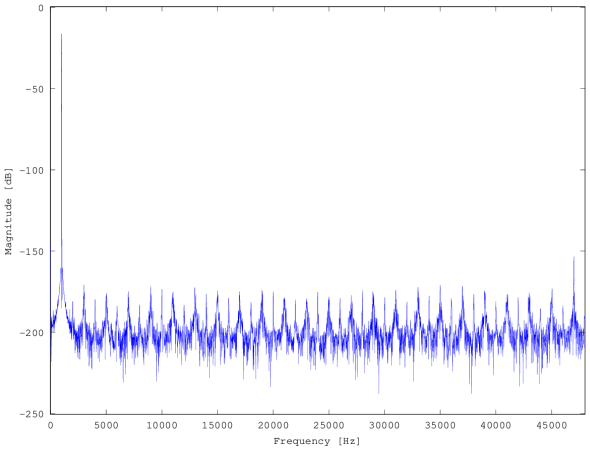
Note that the magnitude of the signal bin is lower than the expected value. This is due to the window applied to the FFT, used to eliminate spectral leakage. This window is, however, not applied when obtaining the output signal's magnitude, as can be seen in Listing 5.5.



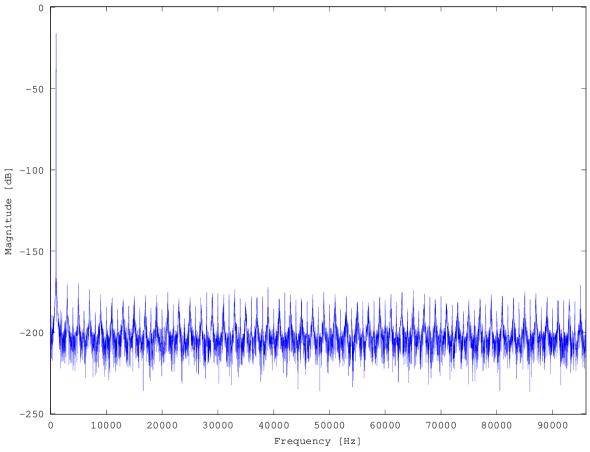
(a) From 44.1 kHz to 48 kHz.



(b) From 44.1 kHz to 192 kHz.

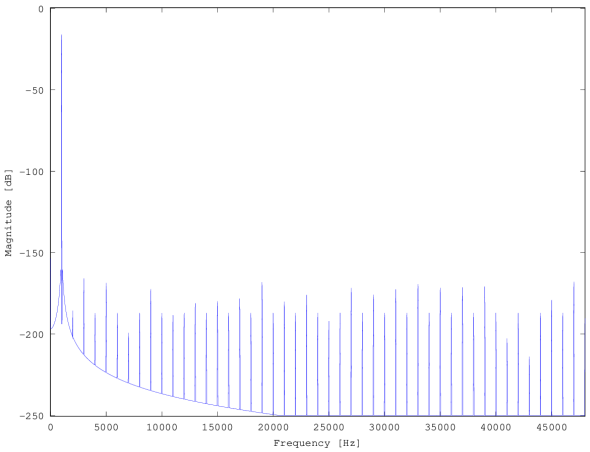


(c) From 48 kHz to 96 kHz.

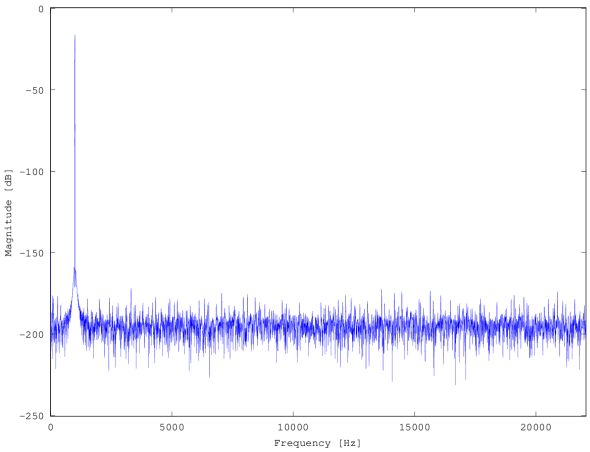


(d) From 96 kHz to 192 kHz.

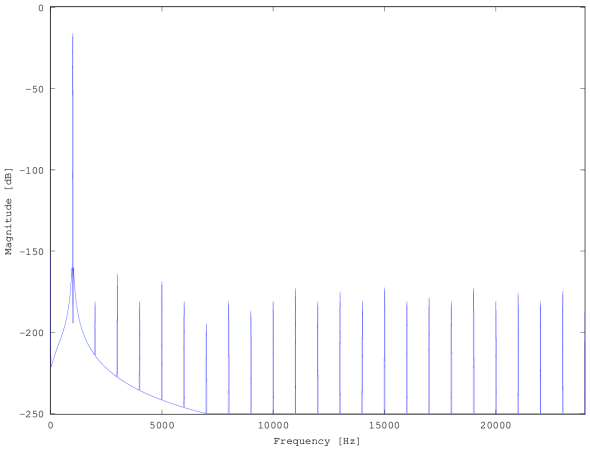
Figure 6.1: Fast-Fourier transform of upsampled signals.



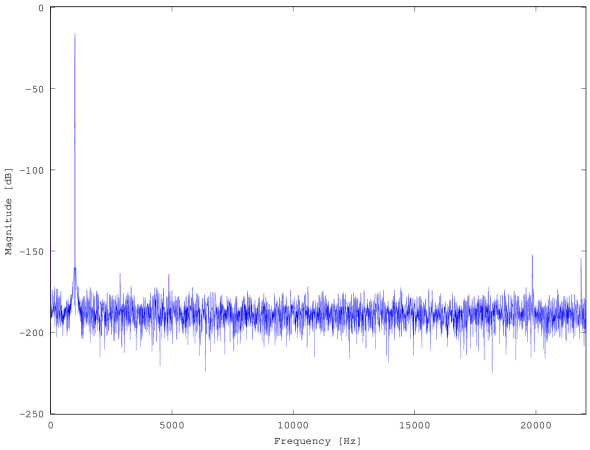
(a) From 192 kHz to 96 kHz.



(b) From 192 kHz to 44.1 kHz.



(c) From 96 kHz to 48 kHz.



(d) From 48 kHz to 44.1 kHz.

Figure 6.2: Fast-Fourier transform of downsampled signals.

6.2 Total Harmonic Distortion Plus Noise

To compute the Total Harmonic Distortion Plus Noise (THD+N), the script presented in Subsection 5.1.4 is used multiple times, for multiple sampling rates, as well as multiple signal frequencies.

The script runs once for every combination of input and output sampling rates shown in Table 6.1. The input signal has a magnitude of $-1\,\text{dBFS}$ and frequency of approximately $1\,\text{kHz}$. A subset of the tests is shown in Table 6.2. The remaining results are shown in Section A.1.

Table 6.1: Commonly used sample rates in audio applications and their particular uses

Sample Rate	Use
8.000 kHz	Telephones/walkie-talkies.
11.025 kHz	Lower-quality PCM, MPEG audio and analizers of subwoofer bandpasses.
16.000 kHz	VoIP and VVoIP applications.
22.050 kHz	Lower-quality PCM, MPEG audio and low frequency energy analyzers.
32.000 kHz	miniDV camcorders, and some video tapes. Also used for high-quality wireless microphones.
44.100 kHz	Audio CDs, most used with MPEG-1 audio (VCD, MP3) covers the audible bandwidth (up to 20 kHz).
48.000 kHz	Professional digital video equipment and consumer video formats, like digital TV, DVD and films.
88.200 kHz	Some professional recording equipment that targets CD, such as mixers or equalizers.
96.000 kHz	High definition DVD and blu-ray audio tracks.
176.400 kHz	High definition CD recorders and other applications targeting CD.
192.000 kHz	Professional video equipment targeting high definition DVD and blu-ray audio tracks.

Input Sampling Rate [Hz]	Output Sampling Rate [Hz]	THD+N [dB]
8000	177242	-139.706173
11022	96006	-138.768504
44132	48003	-138.784562
177242	192012	-141.533573
192012	11022	-141.887817
87912	8000	-141.829588
48003	32002	-136.135586
11022	8000	-138.289522

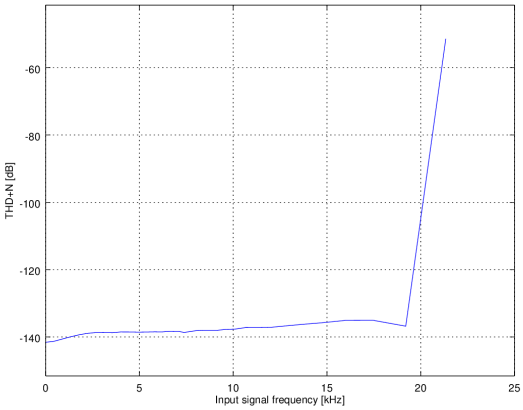
Table 6.2: Total harmonic distortion+noise ratio for some conversions

By direct analysis of the results shown in 6.2 and A.1, it is possible to conclude that the specification of a THD+N equal to $-130\,\text{dB}$ or less is fulfilled for the most common sampling rate conversions.

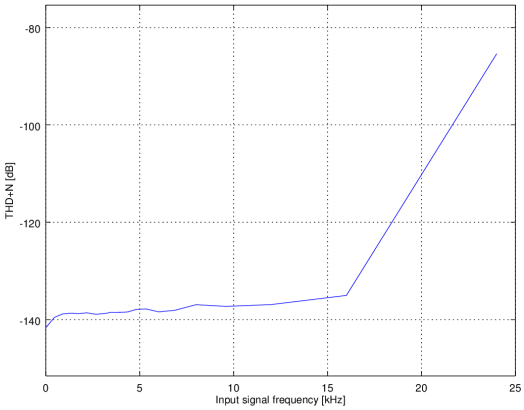
Fig. 6.3 shows a subset of the THD+N results when varying the input signal's frequency. This test is done for 16 conversions, involving combinations of $44.1\,\text{kHz}$, $48\,\text{kHz}$, $96\,\text{kHz}$ and $192\,\text{kHz}$, with an input signal with magnitude of $-1\,\text{dBFS}$. The remaining results are shown in Section A.3.

To keep the results consistent with the ones shown in [1] and [2], the frequency responses are measured in a range from $20\,\text{Hz}$ to the Nyquist frequency of either the input or the output sampling

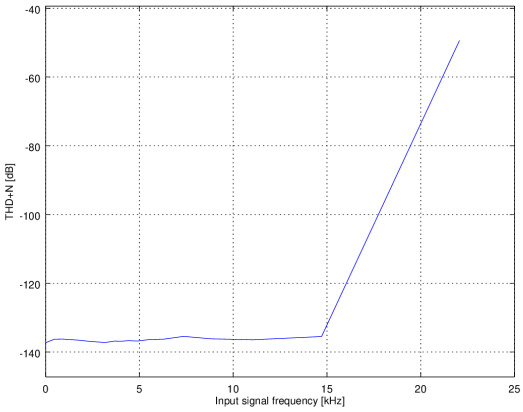
rates, whichever is the lowest. However, it is important to note that the ASRC is designed for audio applications, and thus only the audible range (20 Hz to 20 kHz) is relevant for analysis.



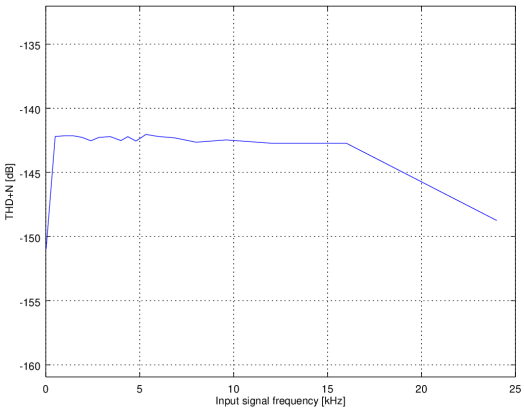
(a) From 44.1 kHz to 192 kHz.



(b) From 44.1 kHz to 48 kHz.



(c) From 48 kHz to 44.1 kHz.



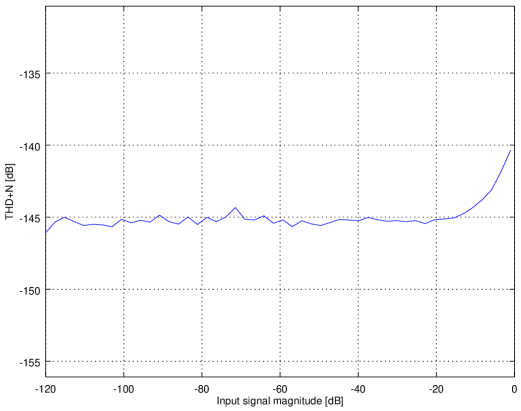
(d) From 96 kHz to 48 kHz.

Figure 6.3: THD+N of the ASRC's output for fixed conversions and varying input frequency.

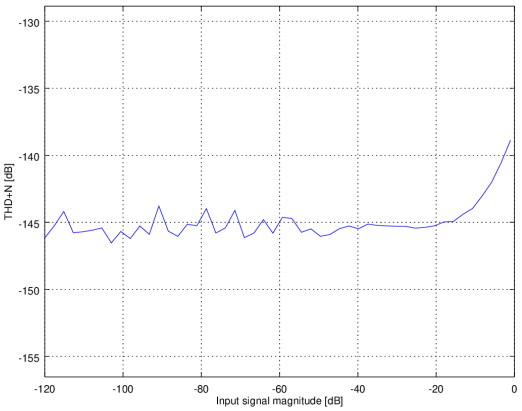
These results show that, for the case of upsampling, the THD+N increases for signals with frequencies close to the Nyquist frequency ($F \approx F_{in}/2$). This is caused by aliasing in the filtered output, as the low-pass filter is not selective enough to completely remove the distortion caused by aliasing. The issue is partially solved by reducing the filter's cutoff frequency, as explained in Section 4.3.1.1.

The THD+N for a subset of conversions of a signal with a frequency of 1 kHz and varying magnitude, between -120 dBFS and -1 dBFS, is shown in Fig. 6.4. The full test runs for the same 16 conversions used in the previous test. The remaining results are shown in Section A.5.

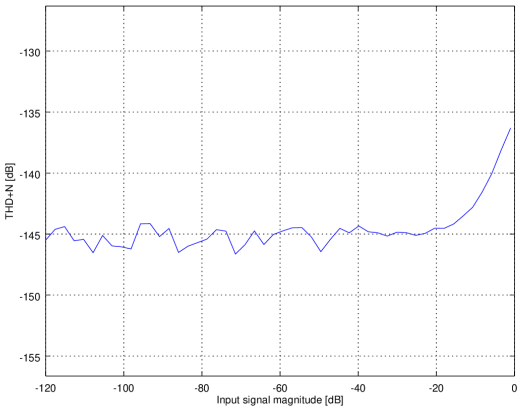
One can note that the increase of the input magnitude leads to an increased THD+N. This is due to two main factors: firstly, an increase of the signal amplitude leads to the increase of the harmonics' amplitude as well. Secondly, due to the variation of the sample rate conversion ratio estimated by the ASRC, caused by the corrections applied by the frequency tracker, as explained in Subsection 4.2.2, some of the energy of the input signal is spread to adjacent frequencies. It is, however, possible to note that the increase of the input signal magnitude does not lead to a THD+N higher than -130 dB.



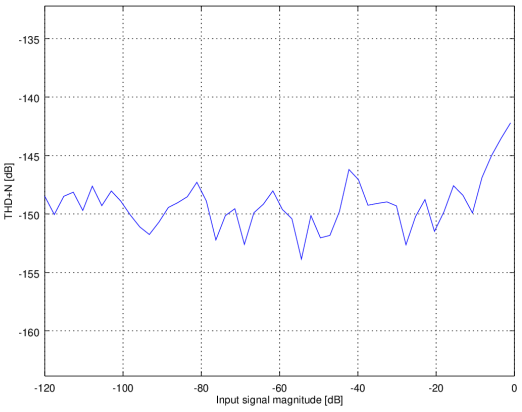
(a) From 44.1 kHz to 192 kHz.



(b) From 44.1 kHz to 48 kHz.



(c) From 48 kHz to 44.1 kHz.



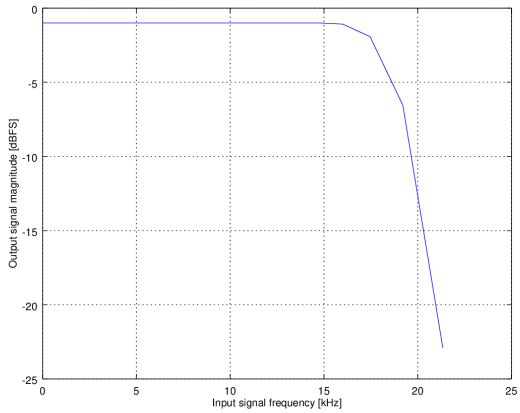
(d) From 96 kHz to 48 kHz.

Figure 6.4: THD+N of the ASRC output for fixed conversions while varying the input magnitude.

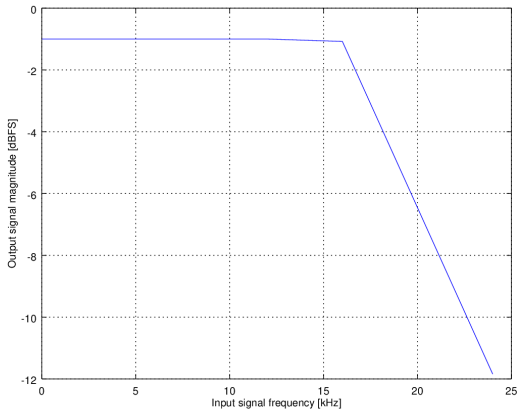
6.3 Frequency Response

The frequency responses of multiple conversions use an input signal with a magnitude of -1 dBFS. Similar to the tests performed in the previous section, this test is performed for 16 conversions (combinations of 44.1 kHz, 48 kHz, 96 kHz and 192 kHz). A subset of 4 conversions is shown in Fig. 6.5. The remaining results are shown in Section A.3.

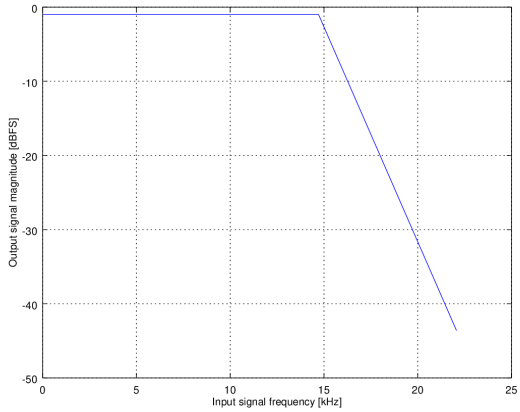
Similarly to the results for the test presented by Fig. 6.3, the frequency of the input varies between 20 Hz to the Nyquist frequency the input or the output sampling rates.



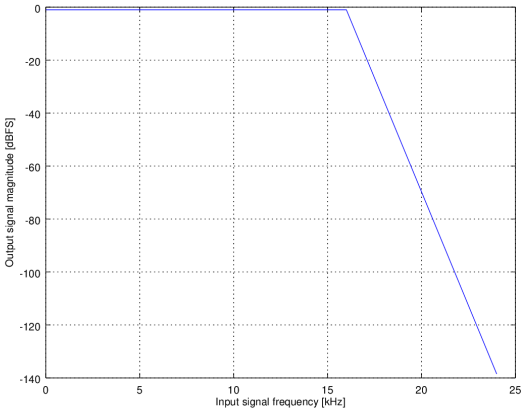
(a) From 44.1 kHz to 192 kHz.



(b) From 44.1 kHz to 48 kHz.



(c) From 48 kHz to 44.1 kHz.



(d) From 96 kHz to 48 kHz.

Figure 6.5: Frequency response of the ASRC for a few conversions.

6.4 Phase Difference After Reset

The results of the phase difference after reset are obtained using the method explained in Section 5.1.5. This test is performed for every combination of input and output sample rates contained in Table 6.1, with the exception of conversion from 8 kHz to sample rates on the higher range, due to limitations of the test script. Table 6.3 shows a subset of the results obtained. Section A.2 contains the remaining set of results.

The difference in the output signal before and after reset for a single conversion is illustrated in Fig. 6.6.

Input Sample Rate [Hz]	Output Sample Rate [Hz]	Phase Shift (# Output Samples)	Phase Shift (microseconds)
11022	96006	0.329150	3.428430
44132	48003	0.001540	0.032090
176366	192012	0.090451	0.471067
192012	11022	0.009258	0.839947
88183	8000	0.044237	5.529248
48003	32002	0.333216	10.41233
11022	8000	0.044391	5.548544

Table 6.3: Group delay of some conversions.

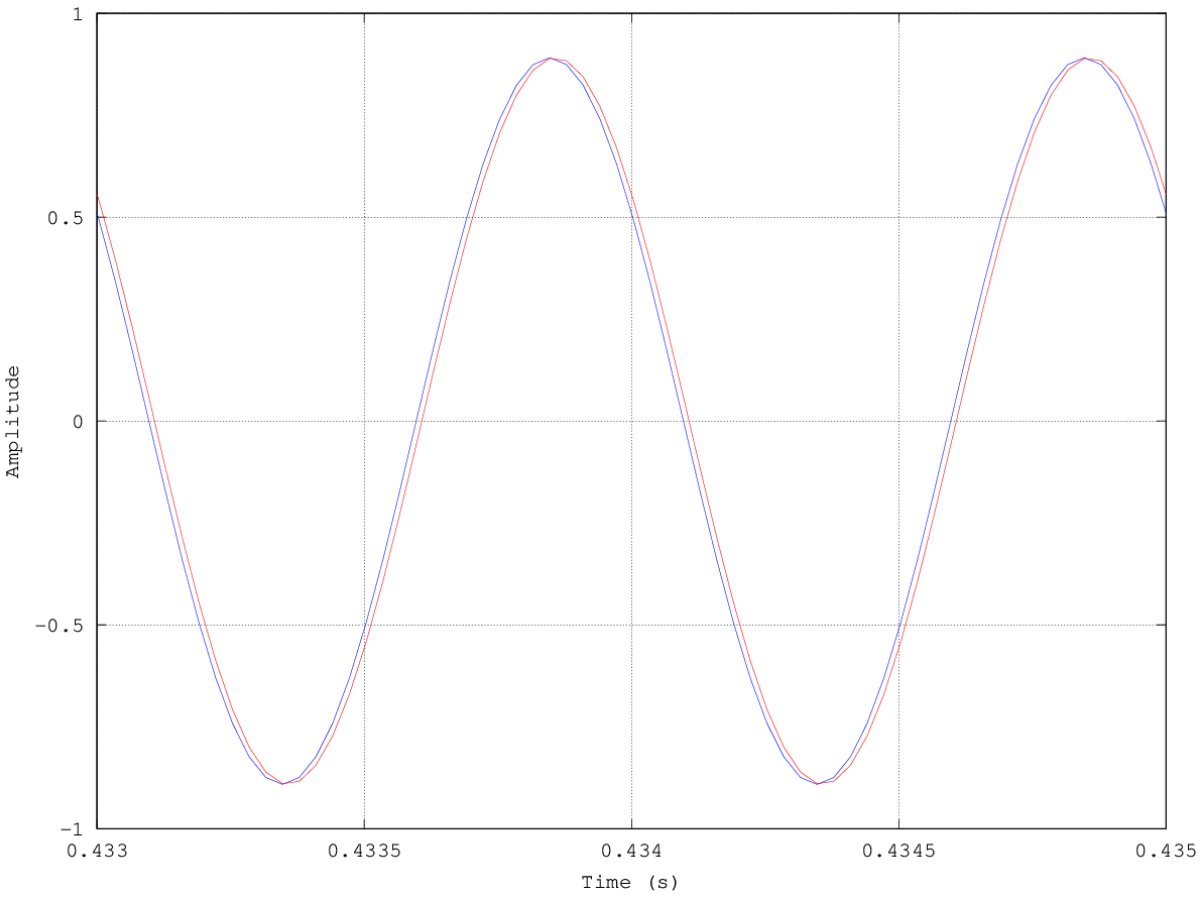


Figure 6.6: Output signal before (blue) and after (red) reset for a conversion from 48 kHz to 32 kHz.

From the results in Table 6.3, one concludes that the synchronization process of the ASRC may lead to a small change in the group delay when repeated. This result is expected, as the ratio estimator module, described in Section 4.2, uses frequency tracking to avoid frequency drift, instead of phase tracking. The corrections use the variation of the delay ΔD instead of the delay D itself.

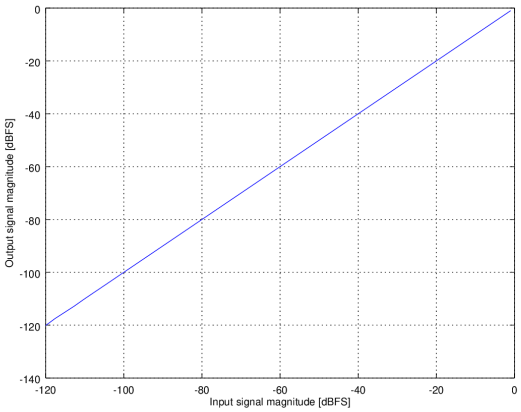
Although there is a small variation of the group delay with each run, the displacement of the output signals for every test performed is less than one output sample. This is because the overall group delay is dictated by the filter delay and only affected by the system sampling instant within the signal's sample

period. As such, the difference does not negatively impact the performance of the ASRC, as for most audio applications such a small phase difference is not audible.

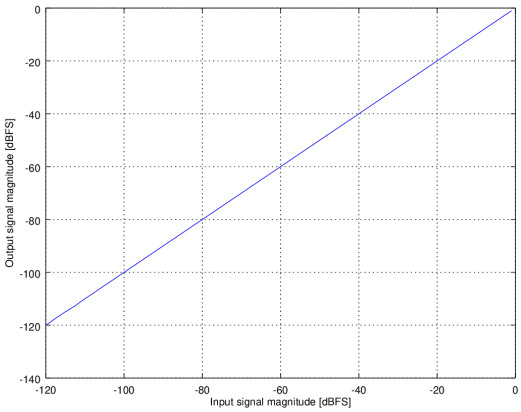
6.5 Linearity Of The ASRC

Linearity is a property of a function or system. A system is considered linear if the relationship between the output and the input can be graphically represented by a straight line. For the ASRC, one expects that the converter is linear, with the amplitude of the output signal being equal to the amplitude of the input signal.

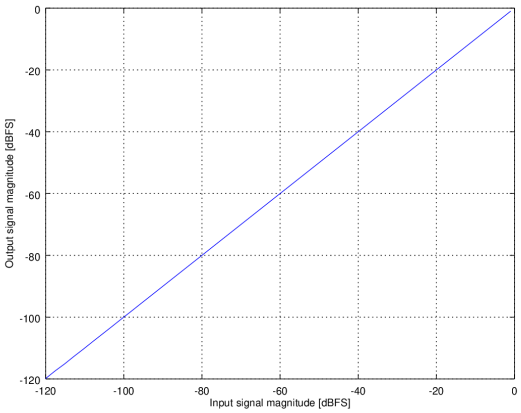
To test the linearity of the ASRC, multiple conversions are performed, with a $1\,\text{kHz}$ input signal, and varying magnitude, between $-120\,\text{kHz}$ and $-1\,\text{kHz}$. The output magnitude is then compared to the corresponding input magnitude. This test is performed for the same 16 conversions as the ones performed in the test described in Section 6.3. The results for a subset of the conversions are shown in Fig. 6.7. The remaining results are shown in Section A.6.



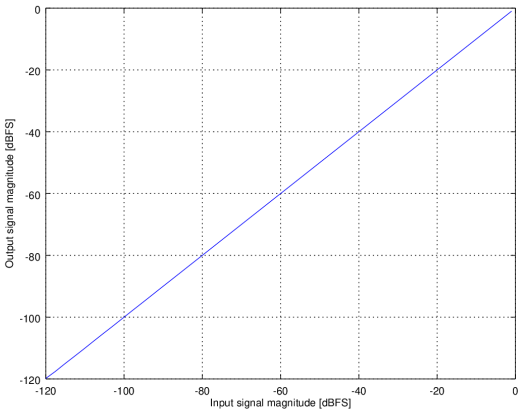
(a) From 44.1 kHz to 192 kHz.



(b) From 44.1 kHz to 48 kHz.



(c) From 48 kHz to 44.1 kHz.



(d) From 96 kHz to 48 kHz.

Figure 6.7: Magnitude of the ASRC's output for fixed conversions and variable input magnitude.

The above plots shows that, at least visually, ASRC appears to be linear. To further analyze the results, one performs a linear regression over the results, by considering that the magnitudes of the output, Y is related to the magnitudes of the input, X as

$$Y = X \times \beta, \tag{6.1}$$

where β is a regression parameter, expected to be approximate to 1. The coefficient of determination, R^2 validates the linear regression, as a value close to 100% indicates that the linear expression define in Equation (6.1) is accurate. Note that the use of this statistical method, or any other finer validation technique, was not found in other similar works. The values of β and R^2 obtained for the tests performed are shown in Table 6.4.

Input Sample Rate [Hz]	**Output Sample Rate [Hz]**	β	R^2
192012	192012	1.000022	99.99993%
192012	44132	0.999999	99.99996%
192012	48003	1.000078	99.99997%
192012	96006	1.000026	99.99997%
44132	192012	1.000058	99.99994%
44132	44132	1.000042	99.99999%
44132	48003	1.000071	99.99993%
44132	96006	1.000057	99.99994%
48003	192012	0.999945	99.99990%
48003	44132	0.999785	99.99993%
48003	48003	0.999979	99.99975%
48003	96006	0.999995	99.99991%
96006	192012	0.999981	99.99992%
96006	44132	0.999971	99.99990%
96006	48003	0.999930	99.99995%
96006	96006	0.999936	99.99992%

Table 6.4: Results of linear regression of some conversions

For the results shown in Table 6.4, one can conclude that the filter is linear for the range of input magnitude tested, as the value of R^2 is close to 100%. Furthermore, as the value of β is close to 1, one can also conclude that the converter has an approximate gain of 1. This is close to ideal, as it means that the output signal is approximately the same as the input signal.

6.6 FPGA Resource Usage

The detailed resource usage of the ASRC on the Kintex Ultrascale FPGA (XCKU040-FBVA676-1-C) is shown in Table 6.5. The resource usage of the ASRC on the Basys3 board (XC7A35TCPG236-1) and on the Cyclone-V-GT-DK board (5CGTFD9E5F35C7) are shown in Tables 6.6.

As can be seen in Table 6.5, more than half of the lookup tables (LUTs) are used by the ratio estimator (ro_meter) module. In it, the divider module is the most resource-hungry, as it needs to compute averages that require many bits to account for the accumulated periods, number of accumulations and of fractional bits of the conversion ratio.

Instance Name	LUTs	Flip-Flops	RAMB36	RAMB18	DSP48
asrc_hardcore	1654	1540	13	1	9
asrc_hardcore (without submodules)	36	26	0	0	0
audio_out_fifo	29	46	0	1	0
data_mem	10	32	1	0	0
data_mem (without submodules)	10	32	0	0	0
mem	0	0	1	0	0
resampler	585	501	12	0	9
resampler (without submodules)	4	99	0	0	0
addr_gen	234	128	0	0	2
addr_gen (without submodules)	1	1	0	0	0
alpha	107	48	0	0	2
coeff_addr	73	38	0	0	0
s_addr	53	41	0	0	0
coef_rom	170	147	12	0	1
coef_rom (without submodules)	170	147	0	0	1
rom	0	0	12	0	0
macc	177	127	0	0	6
ro_meter	994	935	0	0	0
ro_meter (without submodules)	454	593	0	0	0
Tin_meter	19	37	0	0	0
Tout_meter	19	37	0	0	0
div	445	198	0	0	0
mul	57	70	0	0	0

Table 6.5: Resource utilization on a XCKU040 (hierarchical representation)

Resource	Used
LUTs	1677
Registers	1540
DSPs	9
BRAM	13

Resource	Used
ALM	1,083
FF	1745
DSP	8
BRAM blocks	52
BRAM bits	419,328

Table 6.6: Resource utilization on a XC7A35 (left) and Cyclone V GT (right)

It is also important to note that, although the resampler module only uses approximately 600 LUTs, it also uses 9 digital signal processing blocks (DSPs) for its multipliers. In an ASIC implementation, the multipliers would occupy a significant silicon area, most likely higher than the silicon area taken by the ratio estimator module.

Regarding the usage of memory blocks, the coefficient ROM is the element that uses the most, as the module stores 16384 (2^{14}) samples, with 24 bits per sample, stored. In FPGA, ROMs are typically implemented with pre-initialised block RAMs, but in an ASIC implementation there are specific ROM hard macros that are one order of magnitude compared to the RAM hard macros, allowing for a cheaper implementation.

6.7 ASIC Resource Usage

By using Cadence's IC synthesis software, *Encounter RC Compiler*, one can estimate the cell and area usage of the ASRC for an ASIC implementation. Table 6.7 presents the resource usage of the core, with a UMC130 nm technology node.

Table 6.7: ASIC resource usage for the ASRC

Resource	Used
Cell count	17510
Area (mm^2)	0.212

To compare this result with the Coreworks CWda52 core [3], implemented in the $TSMC$65 nm and $TSMC$45 nm nodes, one needs to convert the area using geometric scaling. Table 6.8 shows the estimated area usage of the CWda52 with two audio channels, if implemented in the UMC130 nm technology.

Table 6.8: ASIC resource usage for the CWda52 with two audio channels

Resource	Used
Cell count	9220
Area (μm^2)	0.164

The area of the proposed core of is about 30% higher than the CWda52's. However, one can note that, while the area of the proposed core does not depend on the number of channels, the area the CWda52 replicates most of its logic for each pair of channels. This leads to a lower and lower area usage per channel than the CWda52, as the number of channels increases.

Regarding memory size, Coreworks reports the usage of a 28 kB ROM and a 768 B RAM per channel, plus a fixed 48-byte RAM segment. The present ASRC contains a 48 kB ROM, and 2.25 kB of RAM per channel. Note that the RAM usage also depends on the conversion ratio, as explained in Section 6.8. The limited downsampling conversion ratio of 7:1 for the CWda52 compared to the 24:1 ratio of the prpoposed core explains the difference in the needed RAM.

6.8 Multi-Channel Support and Limitations

As defined in the core's target specification, it should support multiple channels. With the design presented in this chapter, there are two components that lead to the limitation of the number of channels: the size of the data memory, presented in Section 4.1, and the number of clock cycles available to compute a sample for every channel.

Considering that the filter is a *sinc* function with 32 zeroes, as explained in Subsection 5.1.2, and the coefficients are iterated in increments of h_step, as shown in Subsection 4.3.1.3, one can obtain the total amount of coefficients used for the conversion of an output sample, N_{coeffs}:

$$N_{coeffs} = \frac{32}{h\_step} = \frac{32}{min(0.875, \rho)}. \tag{6.2}$$

The case that leads to the biggest value for N_{coeffs} is a conversion from 192 kHz to 8 kHz, where

$$N_{coeffs} = \frac{32}{\frac{8}{192}} = 768. \tag{6.3}$$

Since one needs to have one input sample for each coefficient, for each channel, the data memory should have a capacity equal or greater than $N_{coeffs} \times N_{channels}$ input samples. As such, the number of channels is limited by the capacity of the data memory. For the results present in this chapter, the data memory has a capacity of 1024 input samples, as can be seen in Table 4.2. That size can, however, be changed through the synthesis of the core with a different synthesis parameter.

Furthermore, as the MACC module performs a single multiplication per system clock cycle, as shown in Subsection 4.3.3, the output samples can only be computed if the system clock cycle is $(N_{coeffs} + 10) \times N_{channels}$ times faster than the output sample rate. The 10 extra clock cycles refer to the 9 cycles needed to allow the internal signals to pass through the pipeline registers, mentioned in Section 4.4, as well as an extra cycle needed to change the side of the filter.

Chapter 7

Conclusions

An ASRC is a rather complex circuit, and only major semiconductor players such as Cyrrus Logic, Analog Devices and Texas Instruments have Integrated Circuit (IC) solutions available in the market. To the best of the author's knowledge, the only existing ASRC IPs in the market, the CWda5x family, is made available by the IP design company Coreworks, SA. As was explained in Section 1.1, the CWda5x cores have a higher (worse) THD+N than the IC solutions, as well as a more limited range of conversion ratios.

This thesis proposes a new design of an audio Asynchronous Sample Rate Converter (ASRC), implements it and presents the experimental results. The circuit is described in Verilog, simulated, and prototyped in FPGA. The ASRC is designed as a multi-channel sample rate converter Intellectual Property (IP) module for integration into System-on-Chip. The imposed specifications have the purpose of making it competitive to any IC or IP solutions, with the CWda52 IP core, the AD1896 and the SRC4194 chips being the leading competitors.

The specifications include: support for up to 24-bit samples, sampled in the range between $8\,\mathrm{kHz}$ and $192\,\mathrm{kHz}$, multiple (hundreds) of channels, THD+N of $-130\,\mathrm{dB}$ or less, a synchronisation time lower than $200\,\mathrm{ms}$, variation of phase between input and output of less than one output sample after a reset, and a hardware resource consumption similar to the CWda52.

The ASRC developed in this thesis can be split into two main blocks: the resampler and the conversion ratio estimator. The resampler uses a finite impulse response filter with a variable number of coefficients dynamically computed. It uses a ROM and an interpolator circuit to determine the coefficients for the multiply-accumulate module. The design uses very few hardware resources, as the arithmetic computations are done sequentially.

The conversion ratio estimator uses counters to determine the conversion ratio by asynchronously measuring the periods of the sample rate clocks. After a sufficiently high number of measurements, the ratio between the output and input ample rates can be accurately estimated, enabling the start of the conversion. After that, a frequency tracker performs a fine adjustment of the initial approximation, keeping the ASRC synchronised to both the input and output clocks. The computation is done in a third

and high-speed system clock domain. This unique architecture was not found in any other previous work.

7.1 Achievements

The final design fulfils every specification imposed in Section 1.3.

The bit-width of the samples can be specified as a synthesis parameter. Although the specification imposes the upper bound of 24 bits, the IP can support any bit-width, though tests for more than 24 bits are not performed.

Similarly to the CWda5x IP core family and the IC chips, the present core supports any sample rate between $8\,\mathrm{kHz}$ and $192\,\mathrm{kHz}$. The present design has no limit regarding the conversion ratio; the supported sample rates define the conversion ratio limits. Hence, the supported range is from 24:1 to 1:24 ratios. This range is broader than the ranges of any other solution, as the maximum previously found by the author is 16:1 to 1:16 [2].

The core supports multiple channels. The maximum number of channels supported depends on the input and output sample rates, data memory size, and system clock's frequency. While one can change the size of the data memory by changing the synthesis parameters, the frequency of the system clock may be limited by its critical path length. For example, the core can support up to 8 channels for the worst conversion ratio: 192:8 kHz for a 100 MHz system clock For most standard conversion rates, the core support at least 16 channels. The solution is thus the most competitive ones in terms of the number of supported channels.

The output's THD+Nis lower than $-136\,\mathrm{dB}$ for all the conversions in the supported range; a total of 121 conversions have been run. For most conversions, the THD+N is around $-140\,\mathrm{dB}$. For the best case, the THD+N is $-143\,\mathrm{dB}$. These figures are a direct improvement over the CWda52 and lead to a sound fidelity similar to the IC counterparts, as their THD+N reaches values around $-140\,\mathrm{dB}$ [2].

Thanks to the unique architecture of the conversion ratio estimator, the synchronisation time of the core is only $20\,\mathrm{ms}$. This low synchronisation time improves the IC solutions, as the synchronisation time is one order of magnitude lower. This architecture also presents no issues regarding the variation of the phase after a reset; it is lower than one output sample for the 121 tested conversions.

For two channels, the FPGA implementation of core uses around 60% of the total amount of LUTs used by the CWda52, while the DSP usage is double. However, for more than two channels, the resource usage of the CWda52 increases as it replicates itself for each pair of channels. The core proposed in this thesis has no hardware overhead with the number of channels, which makes the hardware resource usage per channel of the proposed core lower and lower than the CWda52's as the number of channels increases. The area usage of the ASIC implementation confirms these results.

By changing the synthesis parameters, ASRCs with different specifications can be produced, trading off its high-end features for less hardware resource usage.

Once the conversion ratio is estimated, the computation of the output samples may proceed synchronously. Synchronous sample rate converters find many applications in digital audio processing to

operate on stored signals for which the sample rate is known. Consequently, synchronous sample rate converter IPs can easily be implemented using just the resampler part of this work and adding the necessary system integration circuitry.

The asynchronous nature of the algorithm dictates the need to work with multiple clock domains, which is as complex as it can get in terms of *digital* hardware design. In this work, three different clock domains are used: the input, output and processing clock domains. Typical synchronisation structures have been developed as needed to avoid errors caused by metastability or data misses.

7.2 Future Work

The design of the ASRC can be further improved regarding the hardware resources usage. More testing in different FPGA or even an ASIC test chip is helpful for further validating its specifications. More interfaces can be developed for easy integration of the core in a variety of other systems. Other interfaces can be developed as wrappers for the core to simplify the integration of the ASRC in other systems. A host of audio interfaces could be supported to make the core even more flexible, such as the several $I^2S - TDM$ interface [26], flavours used by the industry.

The multiply-accumulate unit of the resampler can be optimised, as the multiplication of the input sample and coefficient uses a multiplier unit (which uses DSPs), while the scaling of the output sample by h_step uses a different multiplier unit, despite the fact it is only needed after the multiply accumulate series. The insertion of a control unit to allow the unit to share the same multiplier for both computations leads to a reduction of DSP usage, at the cost of very few LUTs, and a latency overhead of 1 system clock cycle per output sample.

While the ASRC is tested in most common sample rates used by audio applications, both in simulation and FPGA, it is essential to note that the same Mixed-Mode Clock Manager (MMCM) unit generates the audio clocks used in the FPGA. The MMCM can generate non-synchronised clocks related by a rational factor but has limits for the possible ranges. To further test the developed core, using external clocks without any frequency constraints provides more freedom.

Additionally, to further validate the results, the output should be analysed using a professional audio analyser, such as the equipment made available by the company*Audio Precision*, even if there are no reasons to believe that the presented results are wrong. The analysis of the results produced by the developed core is obtained by running an Octave script on the produced data. This script already existed when the work started but has been refined in the course of this work. Moreover, its results have been compared to an available open-source audio measurement solution [27], and no significant differences have been detected.

Bibliography

[1] Analog Devices, *192 kHz Stereo Asynchronous Sample Rate Converter*, March 2003. Rev. A.

[2] Burr-Brown Products from Texas Instruments, *4-Channel, Asynchronous Sample Rate Converter*, June 2004. Rev. B.

[3] coreworks, *Multi-Channel Audio Sample Rate Converters*, June 2016.

[4] R. Crochiere and L. Rabiner, *Multirate Digital Signal Processing*. Prentice-Hall Signal Processing Series: Advanced monographs, Prentice-Hall, 1983.

[5] S. K. Mitra and J. F. Kaiser, eds., *Handbook for Digital Signal Processing*. New York, NY, USA: John Wiley & Sons, Inc., 1st ed., 1993.

[6] P. Beckman and T. Stilson, "An efficient asynchronous sampling-rate conversion algorithm for multi-channel audio applications," in *AES Convention Papers Forum*, no. 6553, October 2005.

[7] P. J. Kootsookos and R. C. Williamson, "Fir approximation of fractional sample delay systems," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 43, pp. 269–271, March 1996.

[8] C. Tseng and S. Lee, "Design of fir fractional delay filter based on maximum signal-to-noise ratio criterion," in *2013 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference*, pp. 1–8, Oct 2013.

[9] V. Valimaki and T. I. Laakso, "Principles of fractional delay filters," in *2000 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.00CH37100)*, vol. 6, pp. 3870–3873 vol.6, June 2000.

[10] A. Yardin, G. D. Cain, and A. Lavergne, "Performance of fractional-delay filters using optimal offset windows," in *1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 3, pp. 2233–2236 vol.3, April 1997.

[11] R. Adams and T. Kwan, "A stereo asynchronous digital sample-rate converter for digital audio," *IEEE Journal of Solid-State Circuits*, vol. 29, no. 4, pp. 481–488, 1994.

[12] S. CHARANJIT, M. Patterh, and S. Sharma, "Efficient implementation of sample rate converter," *International Journal of Advanced Computer Sciences and Applications*, vol. 1, 01 2011.

[13] E. Hogenauer, "An economical class of digital filters for decimation and interpolation," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 29, pp. 155 – 162, 05 1981.

[14] Y. Mori and N. Aikawa, "Kernel using piecewise nth polynomials for rate converter," in *Proceedings of the 6th Nordic Signal Processing Symposium, 2004. NORSIG 2004.*, pp. 57–60, June 2004.

[15] N. Aikawa and Y. Mori, "Kernel with block structure for sampling rate converter," in *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03).*, vol. 6, pp. VI–269, April 2003.

[16] C. W. Farrow, "A continuously variable digital delay element," in *1988., IEEE International Symposium on Circuits and Systems*, pp. 2641–2645 vol.3, June 1988.

[17] M. Blok and P. Drozda, "Variable ratio sample rate conversion based on fractional delay filter," 2015.

[18] D. Babic, J. Vesma, T. Saramaki, and M. Renfors, "Implementation of the transposed farrow structure," in *2002 IEEE International Symposium on Circuits and Systems. Proceedings (Cat. No.02CH37353)*, vol. 4, pp. IV–IV, May 2002.

[19] F. Rothacher, "Sample rate conversion: algorithms and vlsi implementation," 1995.

[20] E. Stikvoort, *Some subjects in digital audio : noise shaping, sample-rate conversion, dynamic range compression and testing*. PhD thesis, Department of Electrical Engineering, 1992. Proefschrift.

[21] Fat32, "Calculating the phase shift between two signals based on samples." `https://dsp.stackexchange.com/questions/41291/calculating-the-phase-shift-between-two-signals-based-on-samples`, 2017.

[22] IObundle Lda, "IOb-SoC." `https://github.com/IObundle/iob-soc`, 2020.

[23] C. Wolf and et. al., "PicoRV32 - A Size-Optimized RISC-V CPU." `https://github.com/cliffordwolf/picorv32`, 2019.

[24] K. Cheng and et. al., "RISC-V GNU Compiler Toolchain." `https://github.com/riscv/riscv-gnu-toolchain`, 2020.

[25] J. Roque, "Development Environment for a RISC-V Processor: Cache," Master's thesis, Instituto Superior Técnico, Jan 2021.

[26] IObundle Lda, "IOb-I2S-TDM." `https://www.iobundle.com/products`, 2020.

[27] Endolith and et. al., "Waveform analyzer." `https://github.com/endolith/waveform_analysis`, 2020.

Appendix A

Full Test Results

In this appendix, the results for all tests performed on the proposed ASRC IP core are shown for the sake of completeness.

A.1 THD+N

The THD+N obtained for every tested conversion is shown in Table A.1.

Input Sample Rate [Hz]	Output Sample Rate [Hz]	THD+N [dB]
8000	8000	-141.011684
8000	11022	-138.530518
8000	16001	-140.882912
8000	22066	-138.432015
8000	32002	-140.133722
8000	44132	-138.528623
8000	48003	-140.290626
8000	87912	-137.875565
8000	96006	-140.063857
8000	177242	-139.706173
8000	192012	-138.885908
11022	8000	-138.289522
11022	11022	-140.811012
11022	16001	-138.620632
11022	22066	-137.925631
11022	32002	-138.579419
11022	44132	-138.519941
11022	48003	-138.593798
11022	87912	-138.648082
11022	96006	-138.768504
11022	177242	-139.926637
11022	192012	-139.921249
16001	8000	-142.202396
16001	11022	-137.411226
16001	16001	-141.446781
16001	22066	-138.627725
16001	32002	-140.904698
16001	44132	-138.789158
16001	48003	-139.32498
16001	87912	-138.795488

16001	96006	-138.019154
16001	177242	-140.635133
16001	192012	-139.947812
22066	8000	-140.682308
22066	11022	-139.360062
22066	16001	-139.354148
22066	22066	-140.268077
22066	32002	-138.87156
22066	44132	-140.611269
22066	48003	-138.944743
22066	87912	-138.686315
22066	96006	-139.002938
22066	177242	-140.482407
22066	192012	-140.519239
32002	8000	-142.417186
32002	11022	-140.762483
32002	16001	-142.260718
32002	22066	-139.343116
32002	32002	-139.731319
32002	44132	-138.751445
32002	48003	-140.05235
32002	87912	-138.754049
32002	96006	-140.265518
32002	177242	-140.367458
32002	192012	-140.210348
44132	8000	-141.27054
44132	11022	-141.068312
44132	16001	-140.61727
44132	22066	-141.856592
44132	32002	-139.146076
44132	44132	-140.765805
44132	48003	-138.784562
44132	87912	-138.266626
44132	96006	-138.990988
44132	177242	-140.213909
44132	192012	-140.443879
48003	8000	-142.327267
48003	11022	-138.57661
48003	16001	-142.051965
48003	22066	-140.38474
48003	32002	-136.135586
48003	44132	-136.310933
48003	48003	-140.718072
48003	87912	-138.831915
48003	96006	-139.44882
48003	177242	-140.635922
48003	192012	-138.529765
87912	8000	-141.829588
87912	11022	-141.733057
87912	16001	-141.617312
87912	22066	-141.092989
87912	32002	-140.654618
87912	44132	-140.19514
87912	48003	-138.989243
87912	87912	-140.919474
87912	96006	-139.067869
87912	177242	-140.541631

87912	192012	-140.540154
96006	8000	-142.193299
96006	11022	-141.879077
96006	16001	-141.96826
96006	22066	-141.256779
96006	32002	-142.246329
96006	44132	-140.381846
96006	48003	-142.215722
96006	87912	-136.920476
96006	96006	-141.359662
96006	177242	-140.703636
96006	192012	-143.8303
177242	8000	-142.019908
177242	11022	-141.987755
177242	16001	-141.829816
177242	22066	-141.327332
177242	32002	-141.197921
177242	44132	-140.944601
177242	48003	-141.180059
177242	87912	-139.869287
177242	96006	-139.900566
177242	177242	-142.642202
177242	192012	-141.533573
192012	8000	-142.391516
192012	11022	-141.887817
192012	16001	-142.335637
192012	22066	-141.694367
192012	32002	-142.211564
192012	44132	-141.334853
192012	48003	-142.267555
192012	87912	-138.063176
192012	96006	-142.203783
192012	177242	-138.536784
192012	192012	-143.796734

Table A.1: Total harmonic distortion+noise ratio for every tested conversion

A.2 Group Delay After Reset

The group delay obtained for every tested conversion is shown in Table A.2.

Input Sample Rate [Hz]	**Output Sample Rate [Hz]**	**Phase Shift (# Output Samples)**	**Phase Shift (microseconds)**
8000	8000	0.000002	0.0002282628
8000	11022	0.009835	0.8922735
8000	16001	0.001794	0.1121198
8000	22066	0.526498	23.85985
8000	32002	0.006160	0.1924749
8000	44132	0.643224	14.57482
8000	48003	0.008208	0.1709873
11022	8000	0.044391	5.548544
11022	11022	0.000002	0.0001584548
11022	16001	0.450584	28.15971

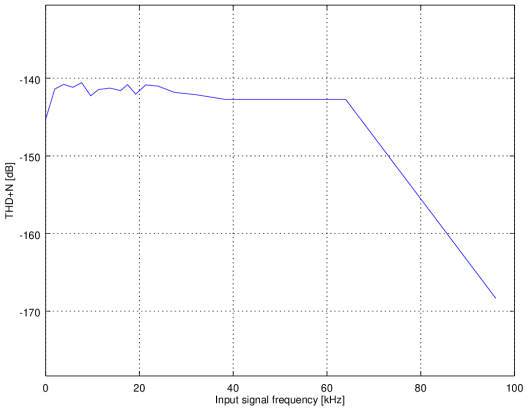
11022	22066	0.000460	0.02084268
11022	32002	0.713158	22.28478
11022	44132	0.000556	0.01259015
11022	48003	0.177509	3.697871
11022	88183	0.064500	0.7314325
11022	96006	0.329150	3.42843
11022	176366	0.269239	1.526586
11022	192012	0.339561	1.768435
16001	8000	0.000000	3.303036e-05
16001	11022	0.515777	46.79489
16001	16001	0.000002	0.0001274857
16001	22066	0.113512	5.144147
16001	32002	0.000997	0.0311462
16001	44132	0.313876	7.112115
16001	48003	0.001173	0.02444472
16001	88183	0.312138	3.53965
16001	96006	0.001623	0.01690017
16001	176366	0.616469	3.495379
16001	192012	0.002853	0.01485741
22066	8000	0.099363	12.4196
22066	11022	0.005251	0.4764416
22066	16001	0.208738	13.04529
22066	22066	0.000002	9.454921e-05
22066	32002	0.707478	22.10728
22066	44132	0.005814	0.1317463
22066	48003	0.630368	13.13183
22066	88183	0.112613	1.277033
22066	96006	0.133732	1.392957
22066	176366	0.418037	2.370269
22066	192012	0.032669	0.170138
32002	8000	0.000000	4.010472e-05
32002	11022	0.083320	7.559399
32002	16001	0.000001	8.111579e-05
32002	22066	0.063100	2.859553
32002	32002	0.000002	7.592318e-05
32002	44132	0.052207	1.182966
32002	48003	0.000540	0.01125465
32002	88183	0.077713	0.8812601
32002	96006	0.000755	0.007867538
32002	176366	0.155831	0.8835641
32002	192012	0.001328	0.006915732
44132	8000	0.081321	10.16448
44132	11022	0.002865	0.2599766
44132	16001	0.000487	0.0304303
44132	22066	0.000213	0.009635942
44132	32002	0.012563	0.3925651
44132	44132	0.000003	6.436396e-05
44132	48003	0.001540	0.03209046
44132	88183	0.355219	4.028188
44132	96006	0.384749	4.007546
44132	176366	0.436397	2.474373
44132	192012	0.231672	1.206549
48003	8000	0.000000	2.383269e-05
48003	11022	0.061109	5.544201
48003	16001	0.000001	6.640037e-05
48003	22066	0.129037	5.847688
48003	32002	0.333216	10.41233

48003	44132	0.792722	17.96228
48003	48003	0.000002	4.940345e-05
48003	88183	0.003298	0.03739722
48003	96006	0.000748	0.007794552
48003	176366	0.000061	0.0003451026
48003	192012	0.001523	0.00793401
88183	8000	0.022164	2.770314
88183	11022	0.051249	4.649652
88183	16001	0.055153	3.446863
88183	22066	0.143591	6.507247
88183	32002	0.027542	0.8606251
88183	44132	0.084036	1.904169
88183	48003	0.333071	6.938534
88183	88183	0.000004	4.496655e-05
88183	96006	0.419096	4.365301
88183	176366	0.005708	0.03236453
88183	192012	0.001460	0.007605842
96006	8000	0.000001	6.799052e-05
96006	11022	0.024072	2.183948
96006	16001	0.000001	7.034833e-05
96006	22066	0.045812	2.076096
96006	32002	0.000002	6.459731e-05
96006	44132	0.009311	0.2109765
96006	48003	0.000002	3.786889e-05
96006	88183	0.003540	0.04013956
96006	96006	0.000004	4.549214e-05
96006	176366	0.008088	0.04586147
96006	192012	0.010162	0.05292369
176366	8000	0.020439	2.554766
176366	11022	0.018455	1.674331
176366	16001	0.033931	2.12053
176366	22066	0.019670	0.8913893
176366	32002	0.124957	3.904657
176366	44132	0.151731	3.43808
176366	48003	0.146333	3.0484
176366	88183	0.496731	5.63293
176366	96006	0.041693	0.4342786
176366	176366	0.000012	6.89182e-05
176366	192012	0.165392	0.8613627
192012	8000	0.000000	0.0
192012	11022	0.009258	0.8399472
192012	16001	0.000000	2.914076e-05
192012	22066	0.015760	0.7141931
192012	32002	0.000001	4.2554e-05
192012	44132	0.200233	4.537069
192012	48003	0.000001	2.543088e-05
192012	88183	0.075704	0.8584818
192012	96006	0.500115	5.209197
192012	176366	0.842821	4.778796
192012	192012	0.000013	6.54621e-05

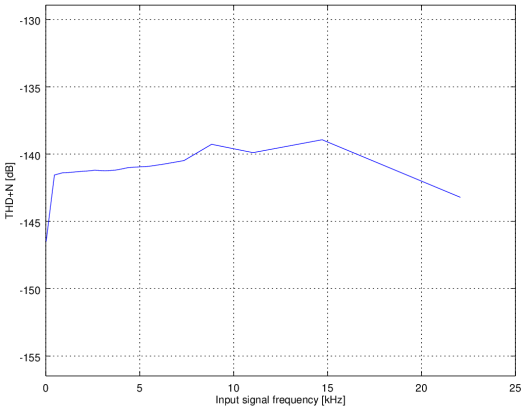
Table A.2: Group delay for every tested conversion

A.3 THD+N With Varying Input Frequency

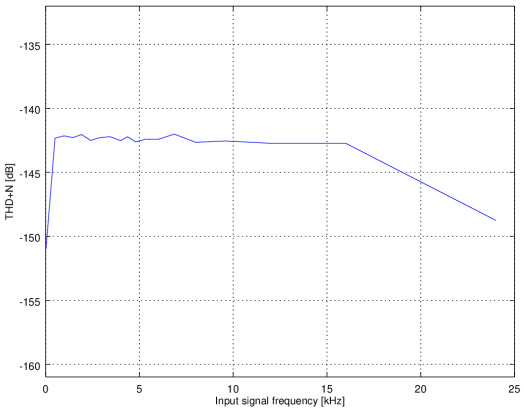
The THD+N for varying input frequency obtained for every tested conversion is shown in Fig. A.1 and Fig. A.2.



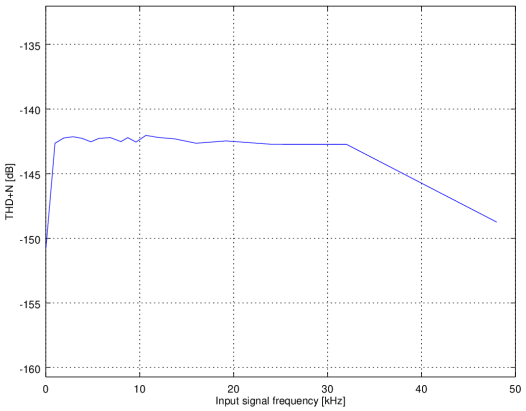
(a) From 192.0 kHz to 192.0 kHz.



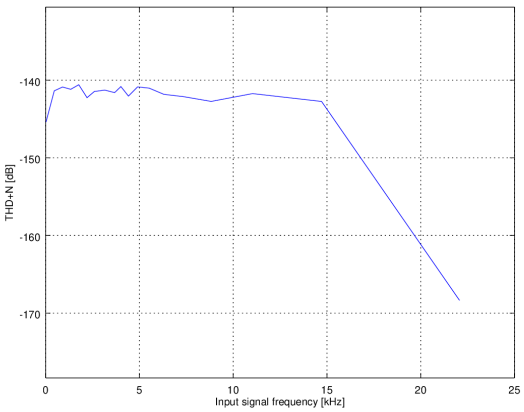
(b) From 192.0 kHz to 44.1 kHz.



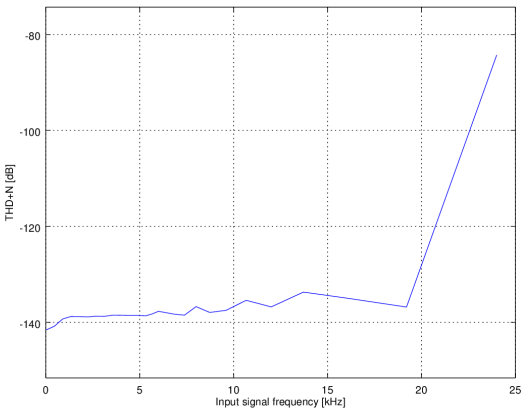
(c) From 192.0 kHz to 48.0 kHz.



(d) From 192.0 kHz to 96.0 kHz.

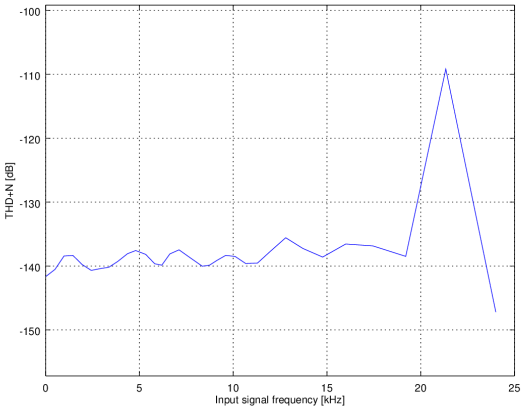


(e) From 44.1 kHz to 44.1 kHz.

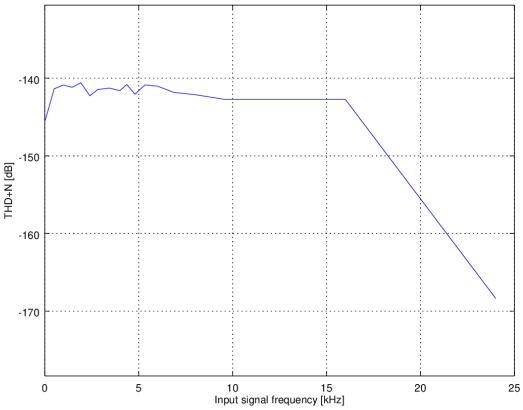


(f) From 44.1 kHz to 96.0 kHz.

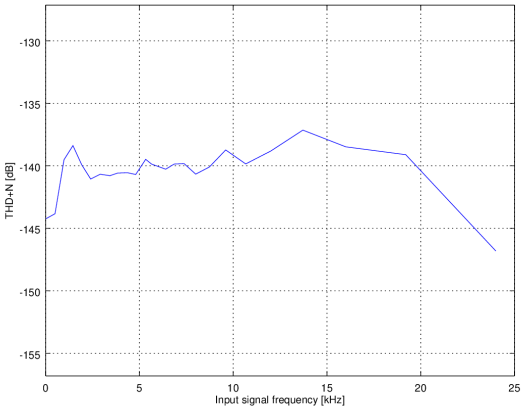
Figure A.1: THD+N of the ASRC's output for fixed conversions and varying input frequency (Full results - 1/2).



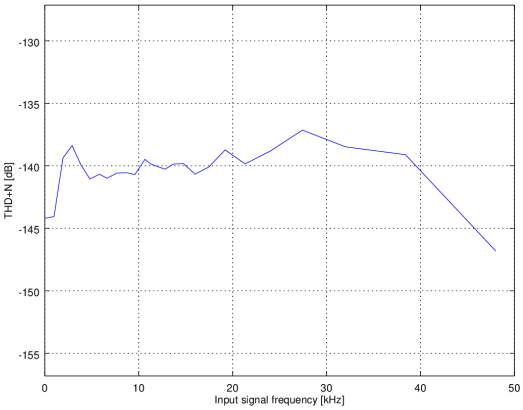
(a) From 48.0 kHz to 192.0 kHz.



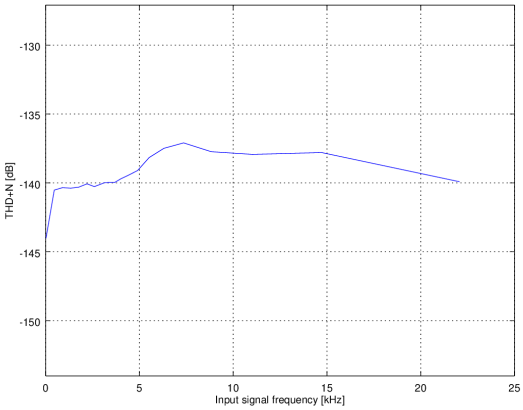
(b) From 48.0 kHz to 48.0 kHz.



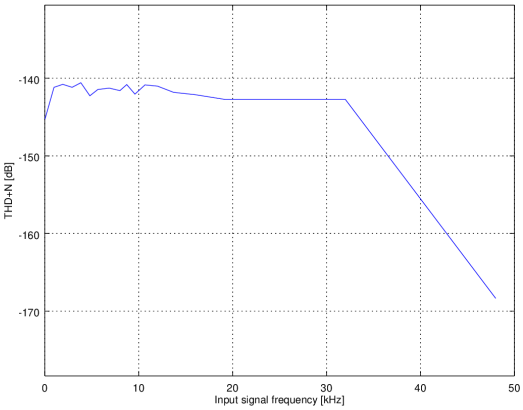
(c) From 48.0 kHz to 96.0 kHz.



(d) From 96.0 kHz to 192.0 kHz.



(e) From 96.0 kHz to 44.1 kHz.

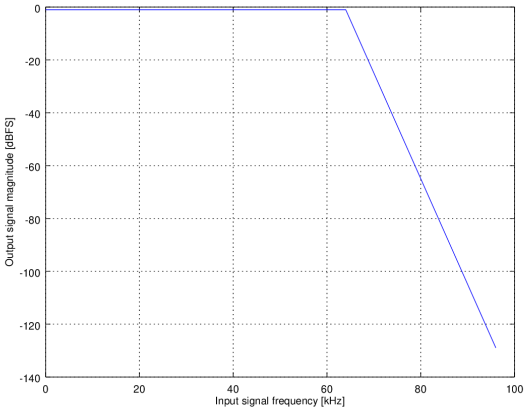


(f) From 96.0 kHz to 96.0 kHz.

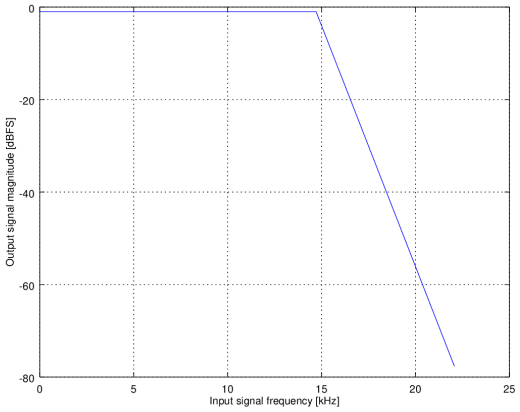
Figure A.2: THD+N of the ASRC's output for fixed conversions and varying input frequency (Full results - 2/2).

A.4 Frequency Response

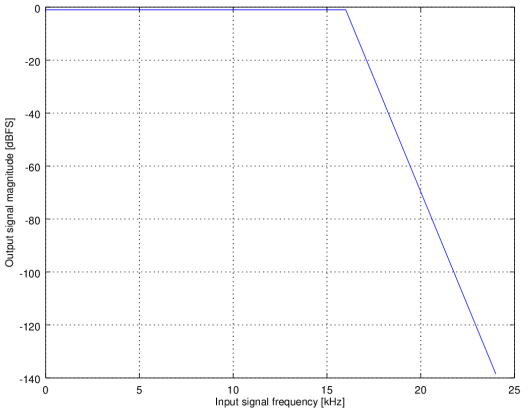
The frequency response obtained for every tested conversion is shown in Fig. A.3 and Fig. A.4.



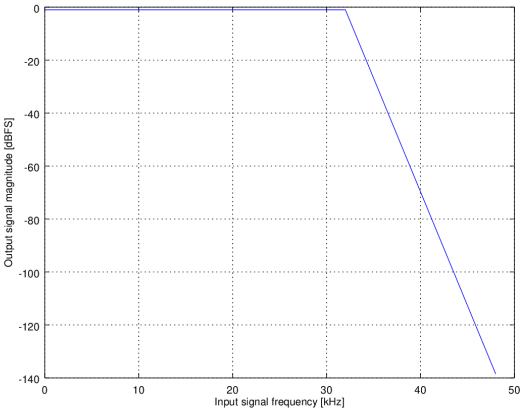
(a) From 192.0 kHz to 192.0 kHz.



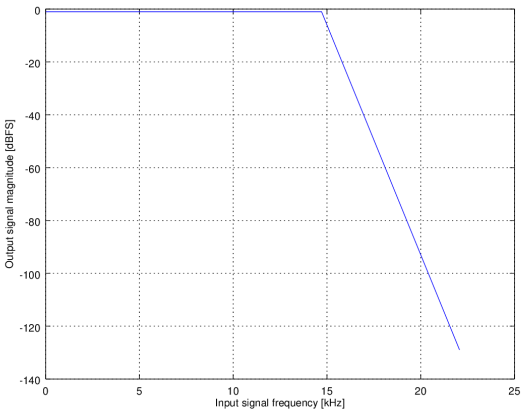
(b) From 192.0 kHz to 44.1 kHz.



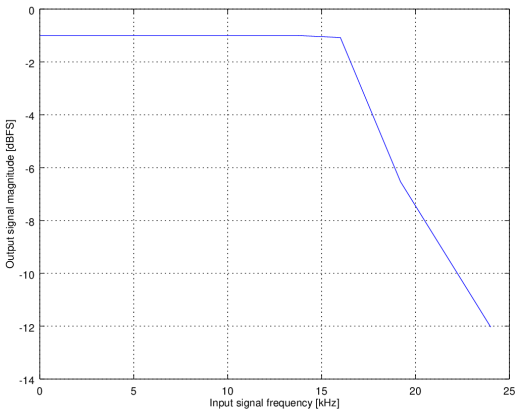
(c) From 192.0 kHz to 48.0 kHz.



(d) From 192.0 kHz to 96.0 kHz.

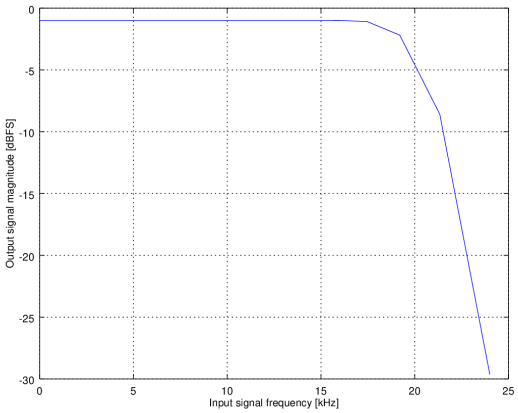


(e) From 44.1 kHz to 44.1 kHz.

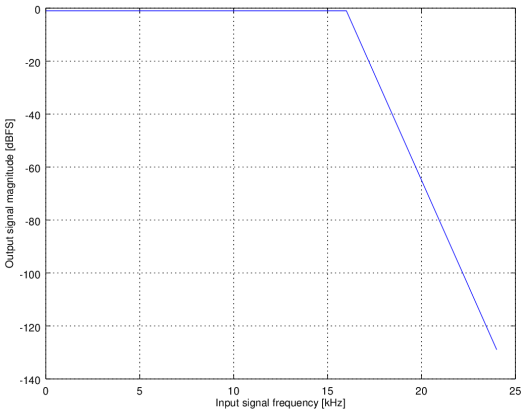


(f) From 44.1 kHz to 96.0 kHz.

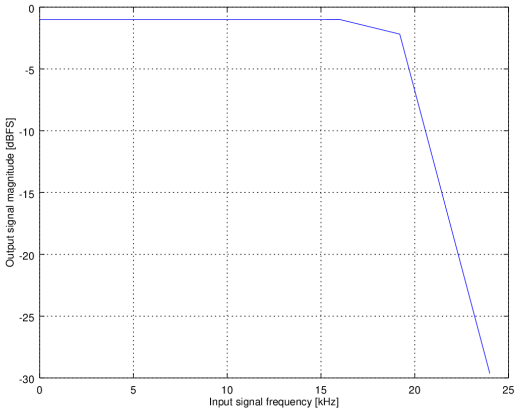
Figure A.3: Frequency response (Full results - 1/2).



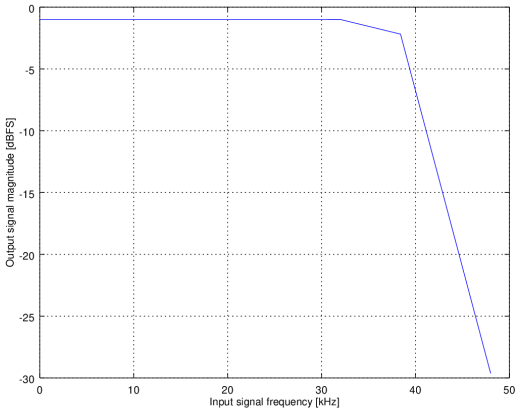
(a) From 48.0 kHz to 192.0 kHz.



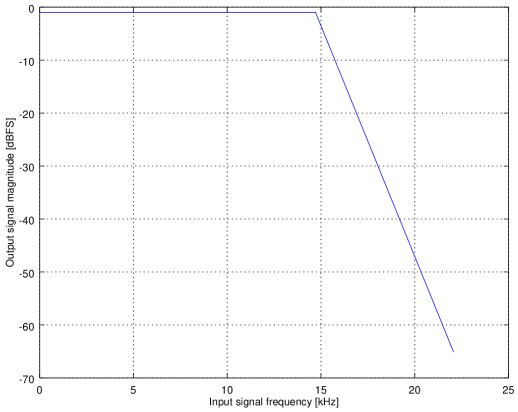
(b) From 48.0 kHz to 48.0 kHz.



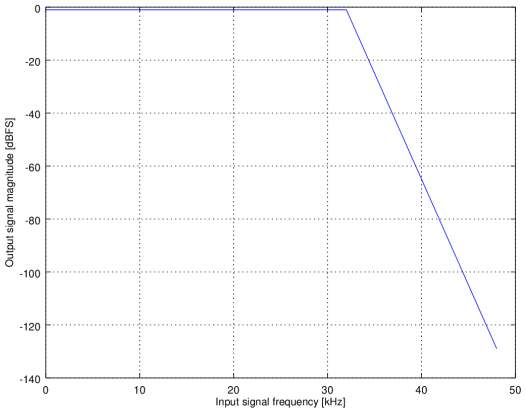
(c) From 48.0 kHz to 96.0 kHz.



(d) From 96.0 kHz to 192.0 kHz.



(e) From 96.0 kHz to 44.1 kHz.

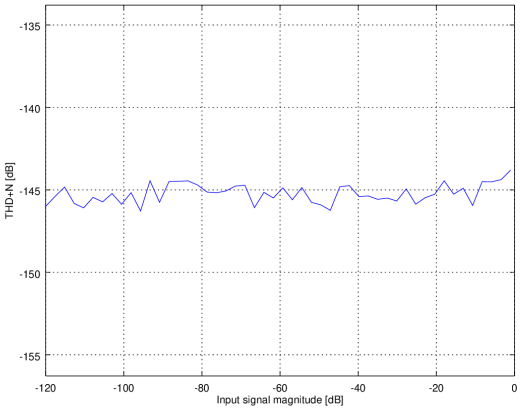


(f) From 96.0 kHz to 96.0 kHz.

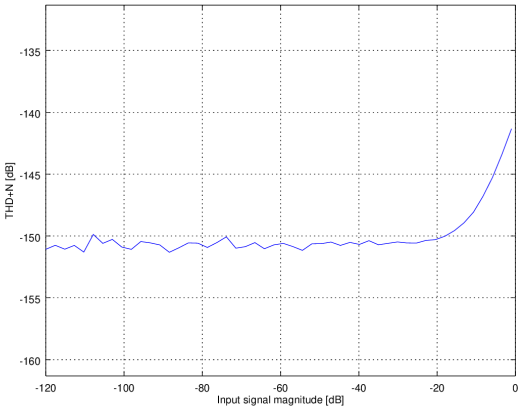
Figure A.4: Frequency response (Full results - 2/2).

A.5 THD+N With Varying Input Magnitude

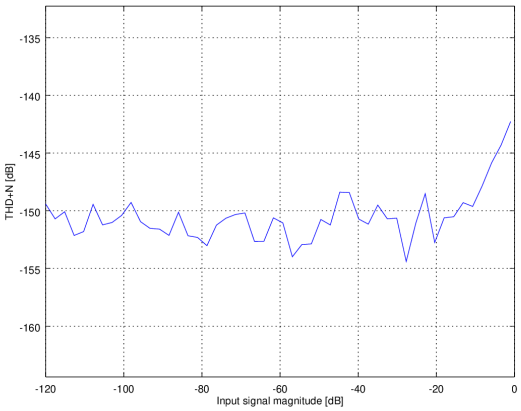
The THD+N for varying input magnitude obtained for every tested conversion is shown in Fig. A.5 and Fig. A.6.



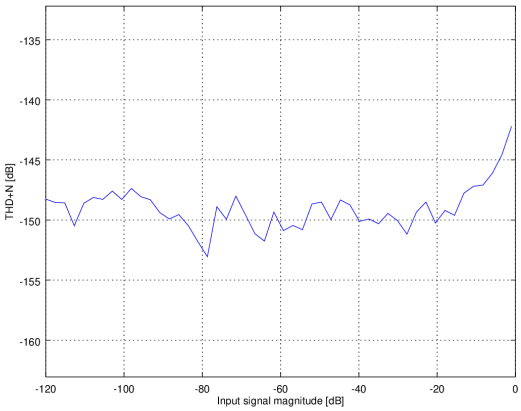
(a) From 192.0 kHz to 192.0 kHz.



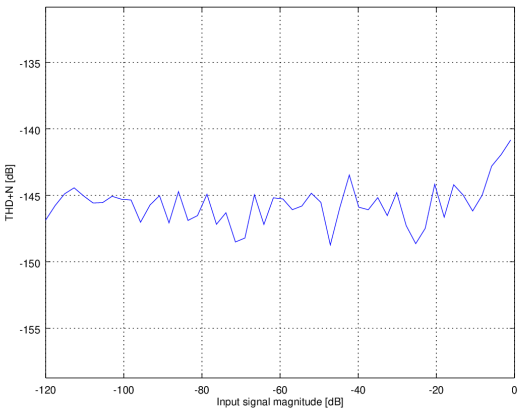
(b) From 192.0 kHz to 44.1 kHz.



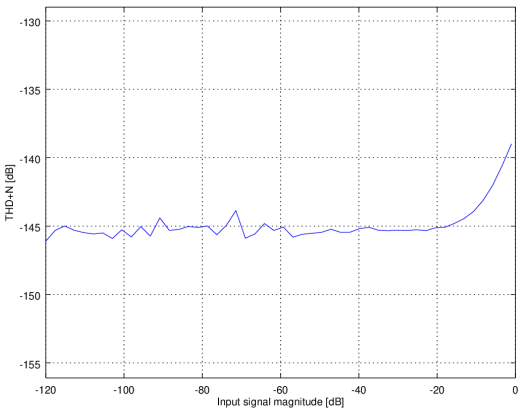
(c) From 192.0 kHz to 48.0 kHz.



(d) From 192.0 kHz to 96.0 kHz.

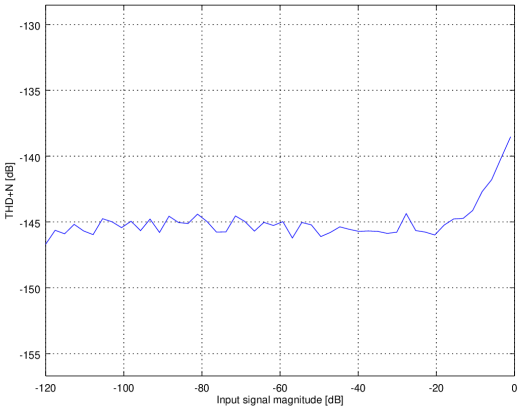


(e) From 44.1 kHz to 44.1 kHz.

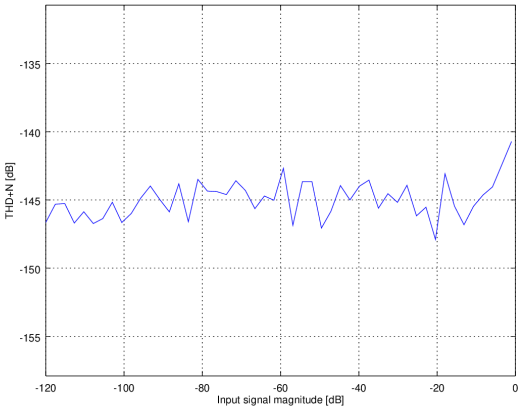


(f) From 44.1 kHz to 96.0 kHz.

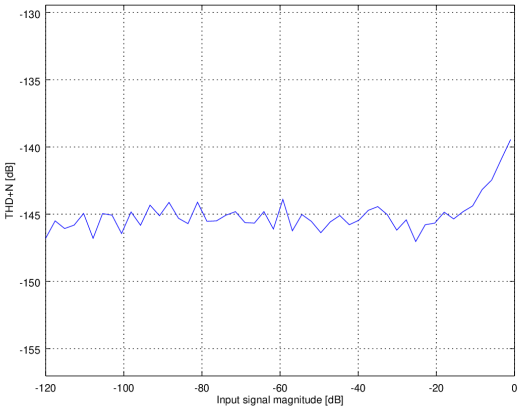
Figure A.5: THD+N of the ASRC's output for fixed conversions and varying input magnitude (Full results - 1/2).



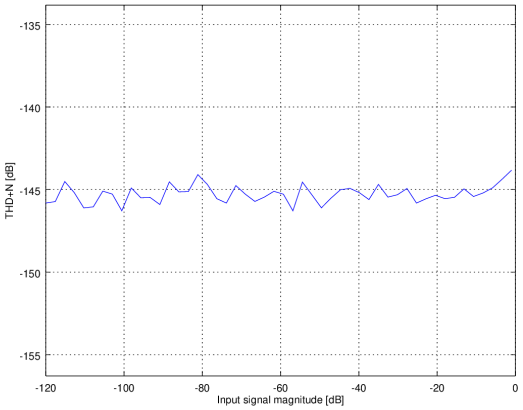
(a) From 48.0 kHz to 192.0 kHz.



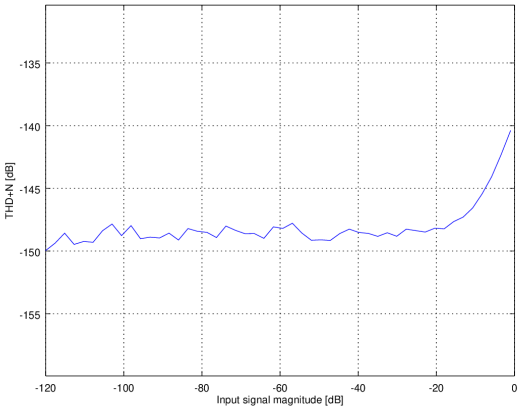
(b) From 48.0 kHz to 48.0 kHz.



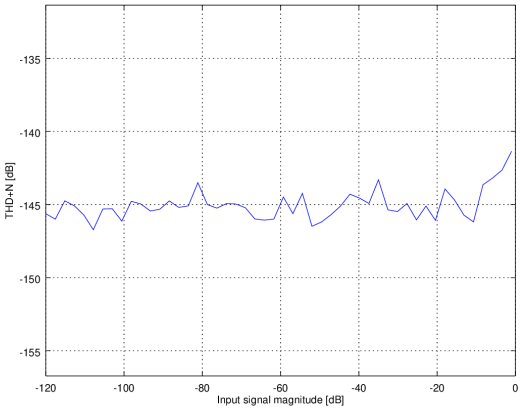
(c) From 48.0 kHz to 96.0 kHz.



(d) From 96.0 kHz to 192.0 kHz.



(e) From 96.0 kHz to 44.1 kHz.

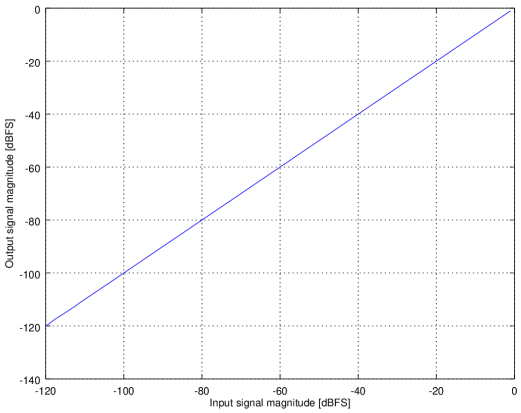


(f) From 96.0 kHz to 96.0 kHz.

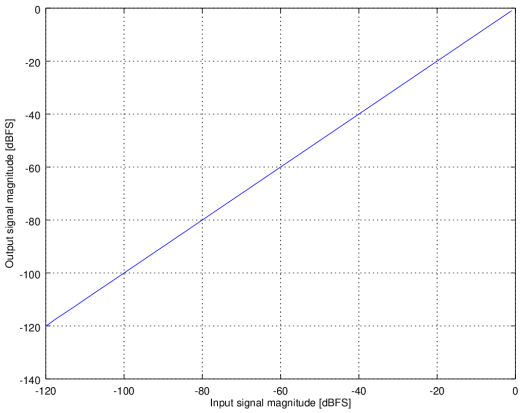
Figure A.6: THD+N of the ASRC's output for fixed conversions and varying input magnitude (Full results - 2/2).

A.6 Magnitude With Varying Input Magnitude

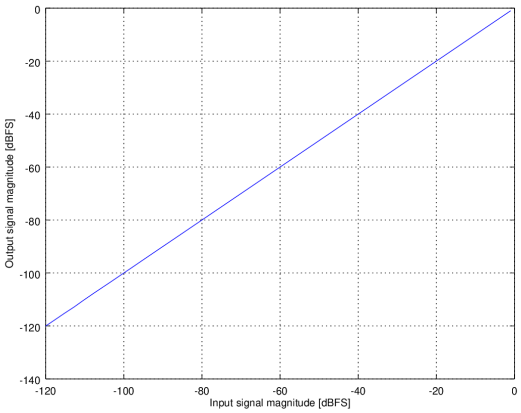
The output magnitude for varying input magnitude obtained for every tested conversion is shown in Fig. A.7 and Fig. A.8.



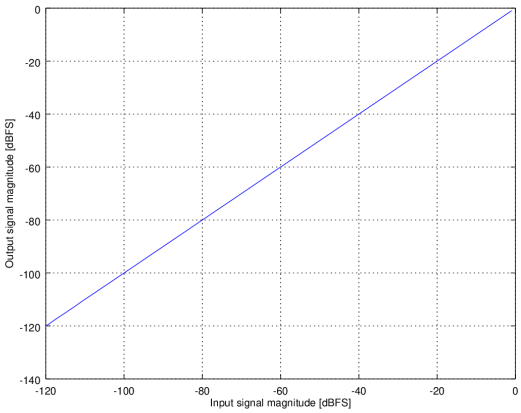
(a) From 192.0 kHz to 192.0 kHz.



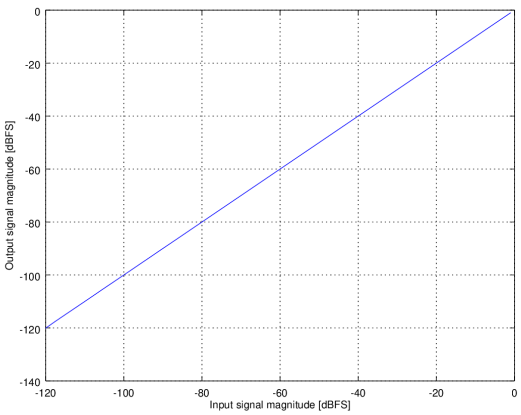
(b) From 192.0 kHz to 44.1 kHz.



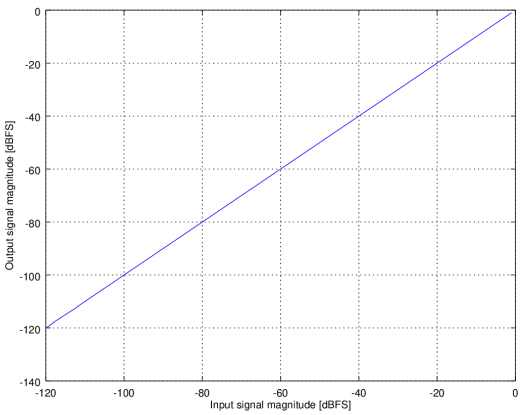
(c) From 192.0 kHz to 48.0 kHz.



(d) From 192.0 kHz to 96.0 kHz.

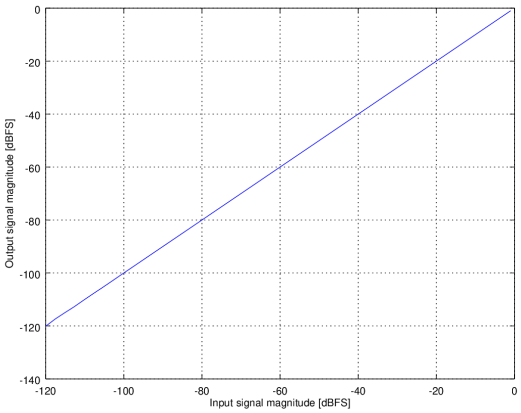


(e) From 44.1 kHz to 44.1 kHz.

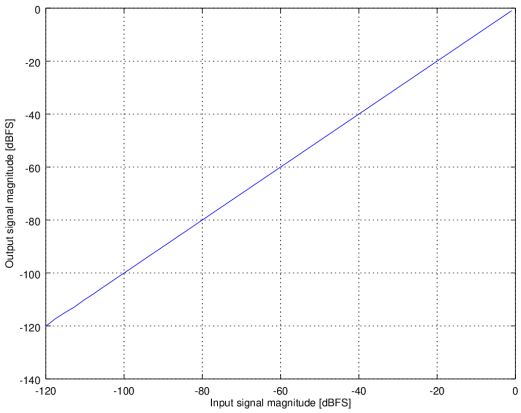


(f) From 44.1 kHz to 96.0 kHz.

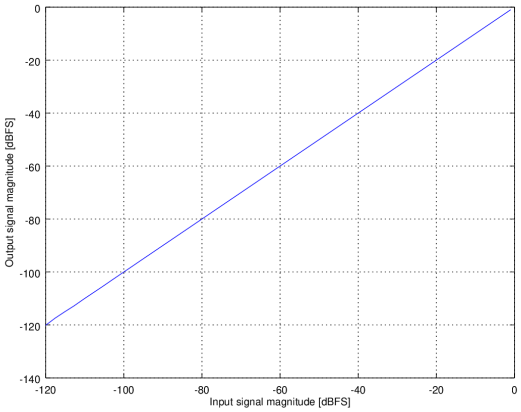
Figure A.7: Magnitude of the ASRC's output for fixed conversions and varying input magnitude (Full results - 1/2).



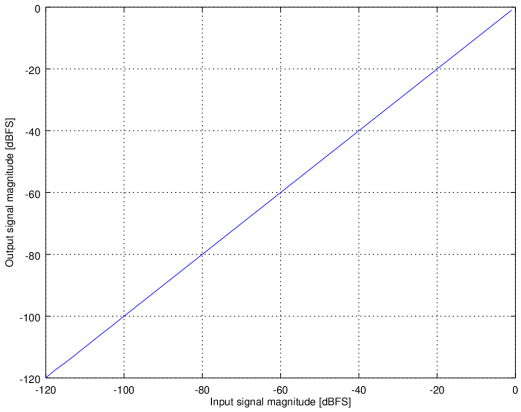
(a) From 48.0 kHz to 192.0 kHz.



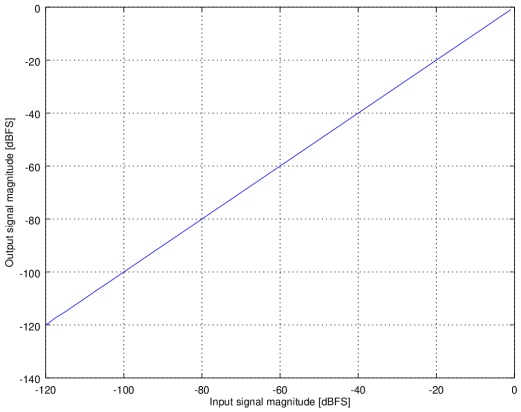
(b) From 48.0 kHz to 48.0 kHz.



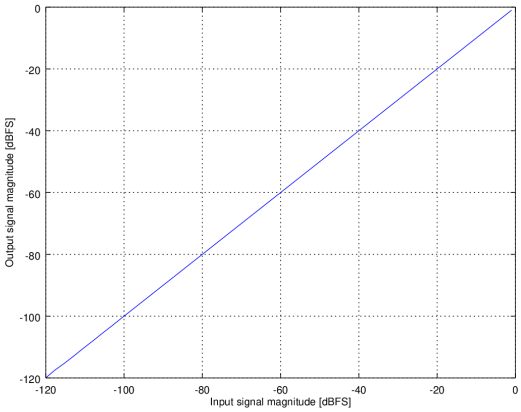
(c) From 48.0 kHz to 96.0 kHz.



(d) From 96.0 kHz to 192.0 kHz.



(e) From 96.0 kHz to 44.1 kHz.



(f) From 96.0 kHz to 96.0 kHz.

Figure A.8: Magnitude of the ASRC's output for fixed conversions and varying input magnitude (Full results - 2/2).