Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Aktive Geräuschkompensation mit einer FPGA-basierten Signalverarbeitungsplattform

Jörn Matthies

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Aktive Geräuschkompensation mit einer FPGA-basierten Signalverarbeitungsplattform

Jörn Matthies

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. B. Schwarz
Zweitgutachter: Prof. Dr. rer. nat. W. Fohl

Abgegeben am 15. Februar 2008

Jörn Matthies

Thema der Masterarbeit

Aktive Geräuschkompensation mit einer FPGA-basierten Signalverarbeitungsplattform

Stichworte

Active Noise Cancellation, Echtzeit Audiosignalverarbeitung, Adaptive Feedforward System, Sequentielles FIR-Filter, LMS-Adaptionsalgorithmus, Xilinx-FPGA-Implementierung, Codec-Interface

Kurzzusammenfassung

In dieser Masterarbeit ist ein System zur aktiven Geräuschkompensation entwickelt worden. Das System identifiziert Störgeräusche und kompensiert diese durch generierte Gegensignale, sodass Geräusche ausgelöscht werden, wie sie durch Gerätelüfter oder durch Vibrationen oder schnelle Fahrt im Inneren eines KFZ entstehen.
Um ein Störsignal eindeutig zu identifizieren ist der Aufbau als Adaptive Feedforward System konzipiert worden, welches sich durch ein Referenzmikrofon zur Aufnahme des Störsignals an seiner Quelle auszeichnet. Die Signalwege des gesamten Feedforward Systems sind identifiziert und analysiert worden, um ein realistisches Modell in Simulink zu erstellen. Aus dem Modell ist eine Hardware-Struktur für das adaptive Filter konzipiert worden, die durch eine HW-Modellierungssprache beschrieben und auf einem Spartan-3-FPGA implementiert ist, welches als digitale Signalverarbeitungseinheit auf einer FPGA-Audio-Codec Plattform dient.
Diese Struktur besteht aus einem sequentiellen FIR-Filter, dessen Koeffizienten durch den LMS-Algorithmus adaptiert werden. Das Filter ist sequentiell aufgebaut worden, um mit einem geringen HW-Aufwand eine hohe Filterordnung zu realisieren. Die durch diesen Aufbau ermöglichte, maximale Filterordnung beträgt $N_{max} = 436$.
Das entwickelte System kompensiert Frequenzen von $200 - 4000\ Hz$. Die Kompensation findet im Ohr des Empfängers statt, wobei das Kompensationssignal über einen Kopfhörer wiedergegeben wird. In den durchgeführten Funktionstests wurde bei Störsignalen mit einer konstanten Frequenz eine Dämpfung von bis zu $-23\ dB$ erreicht. Bei Signalen, die mehrere Frequenzanteile und Rauschen enthielten, lag die Dämpfung bei $-8\ dB$.

Jörn Matthies

Title of the paper

Active Noise Cancellation with a FPGA based signal processing platform

Keywords

Active Noise Cancellation, Realtime audio signal processing, Adaptive Feedforward System, Sequential FIR-Filter, LMS adaption algorithm, Xilinx-FPGA implementation, Codec-Interface

Abstract

With this thesis a system for Active Noise Cancellation has been developed. The system is capable of identifying interfering noises and to compensate this by creating a countercurrent sound. By this means interfering noises can be eliminated, generated by e. g. cooling fans of various devices or vibrations and high speed driving inside a car.

The compensation occurs in the ear of the listener, whereas the compensating signal is introduced via headphones. The signal processing unit consists of a FPGA-Audio-Codec-platform for creating the compensating signal. The signal paths of the entire feedforward system were identified and analyzed in order to establish a realistic model with Simulink. Based on this model, a hardware structure for an adaptive filter has been conceived, which is described by a HW modelling language and implemented on the FPGA.

This structure consists of a sequential FIR-filter, whose coefficients are adapted by the LMS-algorithm. The filter is designed sequential in order to implement a high-order filter with low HW-costs. The filter order is limited by the HW-structure, it´s absolute value is N = 437.

The system developed is capable of compensating frequencies in the range of 200 to 4000 Hz. Functional tests have proven an attenuation of up to - 23 dB of the interfering noise with constant frequency. The attenuation of signals with various frequencies and undefined noise was found at - 8 dB.

Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mich bei der Durchführung und Erstellung dieser Arbeit tatkräftig unterstützt haben. Mein Dank gilt Prof. Dr.-Ing. Bernd Schwarz und Prof. Dr. rer. nat. Wolfgang Fohl.

Für die Unterstützung bei messtechnischen Fragen danke ich Prof. Dr. Ulrich Sauvagerd

Für die Zusammenarbeit während meines Studium bedanke ich mich bei Ingmar Gründel, Olaf Christ und Ning Liu.

Für die Korrektur und Durchsicht dieser Arbeit danke ich Denis Schettler und John Alberts.

Für die finanzielle und moralische Unterstützung während meines Studiums danke ich meinen Eltern Karl-Heinz und Birgit Matthies, meinen Großeltern Ottomar und Ursula Foerster und meinem Bruder Dirk Matthies.

Inhaltsverzeichnis

1 Einleitung **9**

2 Anwendungen adaptiver Filter **13**
2.1 System-Identifikation . . . 13
2.2 System-Inversion . . . 14
2.3 Signal-Prädiktion . . . 15
2.4 Aktive Geräuschkompensation . . . 15
2.4.1 Adaptive Feedback System . . . 17
2.4.2 Adaptive Feedforward System . . . 17

3 Adaptive Filterung in einem Adaptive Feedforward System **19**
3.1 Das adaptive FIR-Filter . . . 19
3.2 Lineare optimale Filterung . . . 21
3.2.1 Linear MSE Abschätzung . . . 21
3.2.2 Berechnung des *Wiener-Filters* durch Minimierung der Fehlerfunktion *J(w)* . . . 21
3.3 Der LMS-Adaptionsalgorithmus . . . 22
3.3.1 Herleitung des LMS-Algorithmus aus dem Newton- und Gradienten-Verfahren . . . 23
3.3.2 Konvergenz des LMS-Algorithmus . . . 24
3.3.3 Normierung des LMS-Algorithmus . . . 28

4 Modellierung und Simulation des Feedforward Systems mit dem LMS-Algorithmus **29**
4.1 Übertragungs-Eigenschaften der verwendeten Hardwarekomponenten im Sekundärpfad . . . 30
4.2 Simulink-Modell . . . 31
4.3 Simulationsergebnisse . . . 34

5 HW-Modellierung des adaptiven Filters für eine Xilinx-FPGA-Plattform **40**
5.1 Entwurfmethodik und Implementierungstechnologien . . . 40
5.2 Systemübersicht . . . 43
5.3 Das AC'97 Codec-Interface . . . 45
5.4 Das adaptive sequentielle FIR-Filter . . . 48
5.4.1 Multiplizierer-Akkumulatoreinheit . . . 52
5.4.2 Block-RAM für die Abtastwerte des Referenzsignals *XN* . . . 53
5.4.3 LMS-Adaptionseinheit . . . 56
5.4.3.1 Adaption der Filterkoeffizienten . . . 56
5.4.3.2 Dual Port Block-RAM für die Koeffizienten . . . 57
5.4.4 Zustandsautomat zur Steuerung des adaptiven Filters . . . 58

6 Syntheseergebnisse und Timing Simulationen des implementierten Systems **62**
6.1 Syntheseergebnisse . . . 62
6.2 Timing Simulation . . . 64
6.2.1 Codec-Interface . . . 64
6.2.2 Adaptives FIR-Filter . . . 67

7 Messtechnische Analyse des Hardware-Aufbaus **70**
7.1 Monofrequente Störsignale 70
7.2 Ventilations-Störsignal 73
7.3 Motor-Störsignal 74

8 Erweiterungen des Systems **76**

9 Zusammenfassung **80**

Abbildungsverzeichnis **81**

Tabellenverzeichnis **87**

Literatur **88**

A Signalverzeichnis **90**

B Messtechnische Ergebnisse der Untersuchung des Sekundärpfades **91**

C Ergebnisse der Signallaufzeitmessungen **94**

D Matlab Code **96**
D.1 Level-2 S-Function Adaptives FIR-Filter 96
D.2 Level-2 S-Function LMS-Adaptionsalgorithmus 97

E VHDL-Quellcode **99**
E.1 Entity CODEC_INTERFACE 99
E.2 Adaptives FIR-Filter VHDL Code 105
E.2.1 Entity LMS_FIR 105
E.2.2 Entity RAM_S_257_18 107
E.2.3 Entity MAC_LMS 109
E.2.4 Entity MAC_FIR 110
E.3 Entity SYSTEM 111
E.4 Testbench VHDL Code 113

F Die Spartan-3 LC Entwicklungsplattform **116**

G Der Audio-Codec UCB 1400 **118**
G.1 Das AC ’97 Interface 118
G.1.1 SDATA_OUT Frame 119
G.1.2 SDATA_IN Frame 121
G.2 Konfigurationsregister des Audio-Codecs 122

H Prozesse des VHDL-Codec-Interfaces **124**
H.1 Serieller Datenempfang des FPGAs vom Audio-Codec 124
H.2 Serielles Senden des FPGAs zum Audio-Codec 125
H.3 Paralleles Laden der Ausgangsschieberegister 125
H.4 Erzeugung des Signals zur Synchronisation zwischen FPGA und Audio-Codec 126
H.5 Bestimmung der Position innerhalb des seriellen Telegramms zur Steuerung der Sende- und Empfangsregister 128
H.6 Steuerung der Eingangsschieberegister 128

H.7 Steuerung der Ausgangsschieberegister 132
H.8 Anlaufphase des Codec-Interfaces 134

I Bilder zur Entwicklungsplattform und zum AV-Board 136

J Tabellen zur Pinbelegung der Konnektoren des Spartan-3-Boards, AV-Boards und P160 Moduls 139

K Tabellen zur Definition der Konfigurationsregister-Bits 142

L UCF-Datei des Spartan 3 FPGA 144

Abkürzungsverzeichnis 145

Index 146

1 Einleitung

Das Ziel dieser Masterarbeit ist die Entwicklung eines *Active Noise Cancellation* (ANC) Systems zur aktiven Kompensation eines Störgeräusches.

Geräuschkompensation ist sowohl im industriellen Bereich an Arbeitsplätzen als auch im privaten Bereich in der Wohnung, im Auto oder bei Kommunikationsmitteln ein an Bedeutung zunehmendes Thema. Passive Techniken zur Geräuschdämpfung, die aus schalldämpfenden oder absorbierenden Materialien bestehen, sind oft nur praktikabel, wenn es um die Kompensation von hochfrequenten Geräuschen geht. Die passive Geräuschkompensation von niederfrequenten Geräuschen ist zwar technisch realisierbar, jedoch in den meisten Fällen sehr platzaufwendig und mit hohen Kosten verbunden [8].

An dieser Stelle schafft die aktive Geräuschkompensation Abhilfe, bei der dem störenden Geräusch ein Gegengeräusch überlagert wird, wodurch sich beide gegenseitig aufheben. Ein weiterer Vorteil von ANC-Systemen ist, dass sie auf Veränderungen des Störsignals reagieren und bei Bedarf auch für den mobilen Einsatz konzipiert werden können. Im industriellen Bereich findet die aktive Geräuschauslöschung beispielsweise im Bereich der Kopfhörerentwicklung [20] und des Fahrzeugbaus [7] Anwendung. Auch im Flugzeugbau wird an der Entwicklung von ANC-Systemen gearbeitet, um den Geräuschpegel von Flugzeugtriebwerken zu reduzieren [27].

Das in dieser Arbeit konzipierte System stellt einen Prototyp zur Kompensation von Motorgeräuschen im Innenraum eines Fahrzeuges dar. Das HAWKS Racing Team des Departments Fahrzeugtechnik der Hochschule für Angewandte Wissenschaften Hamburg hat einen Rennwagen entwickelt, mit dem es sich an der Formula Student beteiligt. Ein in den Wagen integriertes ANC-System könnte die, hauptsächlich durch den Motor verursachte, Geräuschbelastung des Fahrers reduzieren.

Ein Schwerpunkt dieser Arbeit ist die Modellierung eines Feedforward-ANC-Systems auf Basis einer vollständigen Systemanalyse, die alle Laufzeitpfade der vorhandenen Signale berücksichtigt. Der zweite Schwerpunkt liegt auf der parallelen Implementierung der Signalverarbeitungskomponenten mit Nutzung aller FPGA-Ressourcen.

Das ANC-System, welches im Rahmen dieser Arbeit entwickelt worden ist, soll ein durch eine bestimmte Quelle erzeugtes Störsignal $s(t)$ für das menschliche Gehör kompensieren. Das Gesamtsystem besteht im wesentlichen aus vier Komponenten (vgl. Abb. 1):

- Das *AKG C 3000 B* Kondensatormikrofon nimmt das Störsignal $s(t)$ an der Quelle auf und stellt es dem ANC-System als Referenzsignal $x(t)$ zur Verfügung
- Als Fehlermikrofon dient ein Inohr-Kondensatormikrofon *Soundman OKM-II professional* mit Kugelcharakteristik, welches die Überlagerung $e(t)$ aus Primär- $d(t)$ und Kompensationssignal $y(t)$ aufnimmt
- Das *Spartan -3* FPGA *XC3S400* (vgl. Anh. F) ist die digitale Signalverarbeitungseinheit des Systems und erzeugt das Kompensationssignal $x(n)$ durch ein adaptives Filter
- Der *Sennheiser HD 600* ist ein offener Kopfhörer, über den das Kompensationssignal $y(t)$ wiedergegeben wird

Das Störsignal $s(t)$ wird über ein Mikrofon direkt an seiner Entstehungsquelle aufgenommen und dient dem System als Referenzsignal $x(t)$. Auf dem Weg zum Ohr des Empfängers wird das Störsignal $s(t)$ mit dem Nutzsignal $n(t)$ überlagert. Das Nutzsignal $n(t)$ setzt sich aus allen Geräuschen zusammen, die nicht dem Störgeräusch $s(t)$ entsprechen. Durch die Überlagerung entsteht das Primärsignal $d(t)$ welches am Ohr des Empfängers ankommt (vgl. Abb. 1).

Auf dem FPGA ist ein adaptives Filter implementiert, welches als Eingangssignal das digi-

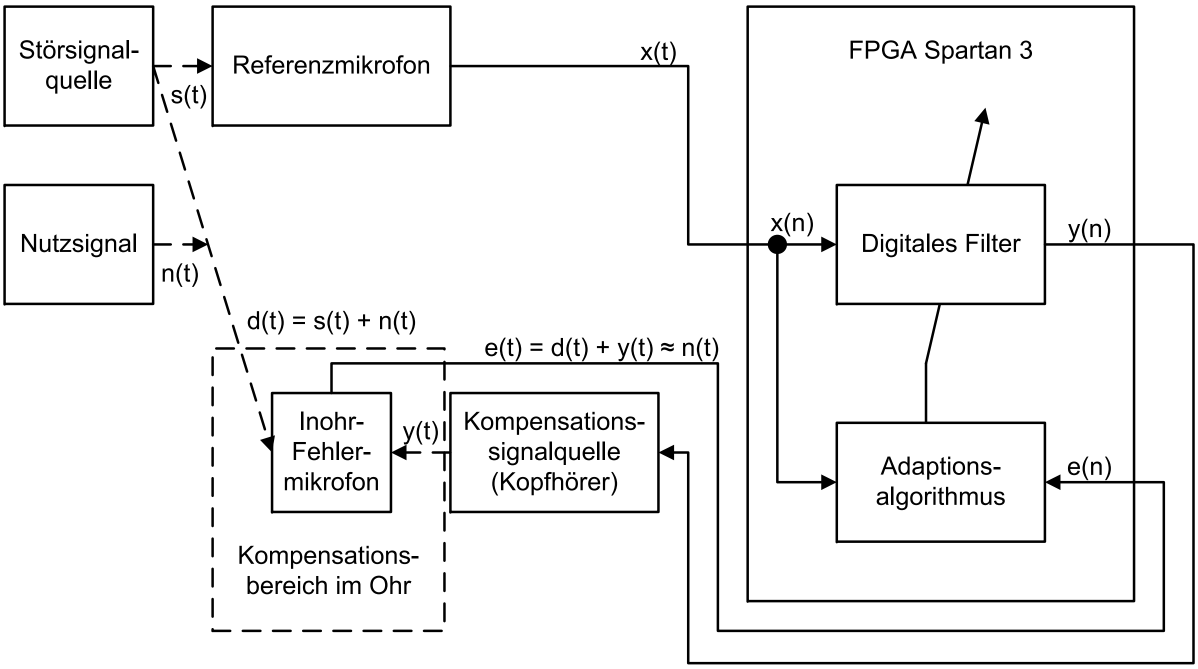

Abb. 1: ANC System mit den physikalisch/analogen Signalen: Störsignal $s(t)$, Nutzsignal $n(t)$, Primärsignal $d(t)$, Referenzsignal $x(t)$, Kompensationssignal $y(t)$ und Fehlersignal $e(t)$, sowie den digitalisierten Signalen $x(n)$, $y(n)$ $e(n)$ und einem FPGA als digitale Signalverarbeitungseinheit

talisierte Referenzsignal $x(n)$ besitzt und aus diesem Signal das Kompensationssignal $y(n)$ erzeugt. Dieses wird in ein analoges Signal $y(t)$ umgewandelt, über einen Kopfhörer wiedergegeben und direkt am Ohr des Empfängers mit dem Primärsignal $d(t)$ überlagert. Aus dieser Überlagerung des Primärsignals $d(t)$ mit dem Kompensationssignal $y(t)$ entsteht das Fehlersignal $e(t)$, das vom Inohr-Mikrofon aufgenommen wird und bei erfolgreicher Kompensation des Störsignals $s(t)$ dem Nutzsignal $n(t)$ entspricht. Anhand des digitalisierten Fehlersignals $e(n)$ wird das Filter auf dem FPGA adaptiert.

Das beim Empfänger ankommende Störsignal $s(t)$ variiert, entweder durch Veränderungen an der Quelle, durch räumliche Änderungen zwischen Quelle und Empfänger oder durch eine Positionsänderung des Empfängers. Die Variationen können sowohl den Frequenzbereich, als auch die Phasenlage oder den Leistungspegel des Signals betreffen. Um auf diese Variationen zu reagieren, wird das digitale Filter adaptiv gestaltet.

Zusätzlich zu den oben genannten Komponenten verfügt das ANC-System über zwei zusätzliche HW-Elemente:

- Der *Philips* Stereo-Audio-Codec *UCB 1400* (vgl. Anh. G) ist mit dem FPGA verbunden und übernimmt die AD- und DA-Wandlung der Audiosignale
- Über das *Motu* Audio-Interface *828 mkII* werden die Pegel der beiden Mikrofon-Signale (Referenzsignal $x(t)$ und Fehlersignal $e(t)$) eingestellt. Außerdem werden sie zu einem Stereo-Signal zusammengefasst, um sie an den Stereo-Eingang des Audio-Codecs anzuschließen

Durch die Verwendung eines Stereo-Codecs ist die dieses ANC-System für die Kompensation auf einem Kanal ausgelegt, da es für die Kompensation auf einem zweiten Kanal ein zusätzliches Fehlersignal $e(t)$ und somit drei Eingangssignale, bestehend aus einem Referenzsignal $x(t)$ und zwei Fehlersignalen $e(t)$, benötigen würde.

Die Entwicklungsziele bestehen darin, zunächst die gegebenen Systemeigenschaften, wie Gruppenlaufzeiten und Frequenzgänge der Signalpfade zu identifizieren und in einem Modell zusammenzufassen. Anschließend ist das adaptive Filter dem Modell entsprechend zu realisieren. Dabei sollen die sequentielle Filterung und der Algorithmus zur Adaption der Koeffizienten des Filters parallel umgesetzt werden.

Das adaptive Filter besteht aus zwei Komponenten, einem sequentiellen FIR-Filter und dem LMS-Algorithmus zur Adaption der Koeffizienten. Um diesen zweiteiligen Algorithmus parallel implementieren zu können, wird ein FPGA verwendet, welches über parallele Ressourcen, wie Multiplizierer, Akkumulatoren und integrierte Speicher verfügt. Die Parallelisierung der während der adaptiven Filterung auf dem FPGA ablaufen Prozessen wird in dieser Arbeit untersucht.

Zusammenfassend ergeben sich folgende Besonderheiten des ANC-Systems:

- Realisierung eines Kopfhörer-ANC-Systems als Adaptive Feedforward System mit einem Referenzmikrofon zur Aufnahme des Störsignals $s(t)$ an seiner Quelle
- FPGA-Audio-Codec Plattform als Basis der adaptiven Filterung
- Umsetzung des adaptiven Filters als sequentielles FIR-Filter mit parallel zur Filterung ablaufendem LMS-Adaptionsalgorithmus

Im Entwicklungsprozess des Systems zur aktiven Geräuschkompensation wurden folgende Software- und Mess-Werkzeuge verwendet:

- *Matlab 7.1/ Simulink 6.3* zur Modellierung und Simulation des Gesamtsystems
- *ModelSim SE 6.2c* zur VHDL-Implementierung und Simulation des Codec-Interfaces und adaptiven FIR-Filters
- *Xilinx ISE 9.1i* zur Synthese des in VHDL modellierten Systems und Programmierung des FPGAs

Diese Dokumentation ist neben dieser Einleitung in acht weitere Kapitel und einen Anhang gegliedert. Dabei werden zunächst die theoretischen Hintergründe der aktiven Geräuschkompensation behandelt. Anschließend wird die Modellierung des Gesamtsystems und speziell des adaptiven sequentiellen FIR-Filters beschrieben. Danach werden die Ergebnisse der Echzeit-Funktionstests präsentiert bevor ein Ausblick auf Erweiterungen des ANC-Systems gegeben wird. Abschließend erfolgt eine Zusammenfassung der Ergebnisse.

In Kapitel 2 wird eine Übersicht zu den Anwendungsbereichen der adaptiven Filterung gegeben, wobei der Schwerpunkt auf dem Bereich der Störsignalkompensation liegt. Dabei wird zwischen den Systemarten *Adaptive Feedback* und *Adaptive Feedforward* unterschieden.

Die theoretischen Grundlagen der adaptiven Filterung werden in Kapitel 3 behandelt, was die Erläuterung des FIR-Filters, sowie die Herleitung und Konvergenzkriterien des LMS-Adaptionsalgorithmus beinhaltet.

Kapitel 4 beschreibt die Modellierung des Feedforward-ANC-Systems. Dazu werden Laufzeiten der akustischen Signale und der Signalverarbeitungsstrecken des Testszenarios analysiert, was zu der Erstellung eines vollständigen System-Modells in Simulink führt.

Es folgt die Beschreibung des in VHDL modellierten Signalverarbeitungssystems in Kapitel 5. Dieses System besteht aus dem Codec-Interface und dem adaptiven Filter und ist auf einem Xilinx Spartan-3 FPGA implementiert. Die einzelnen Komponenten werden mit Blockschaltbildern und VHDL-Simulationsergebnissen erläutert.

In Kapitel 6 werden die Ergebnisse der Synthese und Timingsimulation des Systems vorgestellt und diskutiert. Hier werden die Funktionalität des VHDL-Modells überprüft und spezielle Timing-Eigenschaften verdeutlicht.

Eine Erläuterung unterschiedlicher Testfälle und Analyse deren Ergebnisse folgt in Kapitel 7. Bei den verwendeten Testsignalen handelt es sich um monfrequente und breitbandige Störsignale, wodurch der geeignete Anwendungsbereich des entwickelten ANC-Systems nachvollzogen wird.

Der Ausblick auf Weiterentwicklungen des Systems erfolgt in Kapitel 8 und beinhaltet Maßnahmen, die zu einer höheren Flexibilität des Systems führen sollen und sowohl Änderungen des Aufbaus, als auch der Implementierung bedeuten.

Abschließend folgt in Kapitel 9 eine Zusammenfassung der erzielten Ergebnisse, in der die Funktionalität, Analyse- und Entwurfsmethodik, sowie Mess- und Implementierungsergebnisse des entwickelten ANC-Systems dargestellt werden.

2 Anwendungen adaptiver Filter

Das Konzept der adaptiven Filterung wird im Folgenden mit Anwendungsbeispielen vorgestellt. Diese Anwendungen werden in vier Klassen eingeteilt (vgl. Tab. 1).

Anwendungsklasse	Beispiele
System-Identifikation	Echokompensation Adaptive Regelung Modellierung von Übertragungsstrecken
System-Inversion	Adaptive Entzerrung Blinde Entfaltung
Signal-Prognose	LPC-Analyse Veränderungsdetektion Kompensation von Hochfrequenzstörungen
Multisensor-Störsignal-Kompensation	Aktive Geräuschkompensation

Tabelle 1: Klassifikation von Anwendungen der adaptiven Filterung [10]

2.1 System-Identifikation

Das Ziel einer Anwendung der Klasse System-Identifikation besteht darin, ein unbekanntes LTI-System zu identifizieren. Das adaptive Filter hat dabei die Aufgabe, die unbekannte Übertragungsfunktion $G_s(z)$ des System zu modellieren. Bei erfolgreicher Adaption entspricht die Übertragungsfunktion $G_a(z)$ des adaptiven Filters der Übertragungsfunktion $G_s(z)$ des unbekannten Systems, sodass diese durch die Koeffizienten des Filters beschrieben wird (vgl. Abb. 2).

Dazu wird das Eingangssignal $x(n)$ des Systems mit unbekannter Übertragungsfunktion $G_s(z)$ simultan als Eingangssignal für das adaptive Filter $G_a(z)$ genutzt. Das Ausgangssignal $y(n)$ des unbekannten Systems wird als Ausgangssignal des Gesamtsystems betrachtet. Das Ausgangssignal $\widehat{y}(n)$ des Filters soll sich diesem Signal annähern, wodurch das Fehlersignal $e(n)$ minimiert wird. Um dieses Kriterium zu erreichen werden die Koeffizienten des Filters durch das Fehlersignal $e(n)$ adaptiert.

Angenommen die Ordnung des adaptiven Filters $G_a(z)$ würde der Ordnung des unbekannten Systems $G_s(z)$ entsprechen und das Ausgangssignal $y(n)$ wäre frei von zusätzlichen Signalen, dann würden die Koeffizienten des Filters durch die Minimierung des Fehlers $e(n)$ exakt den Koeffizienten entsprechen, durch die das unbekannte System $G_s(z)$ beschrieben wird. In praktischen Anwendungen ist das Ausgangssignal $y(n)$ jedoch oft überlagert mit zusätzlichen Signalanteilen $i(n)$, die unkorreliert zum Eingangssignal $x(n)$ sind, wodurch das Filter die Übertragungsfunktion $G_s(z)$ des unbekannten System nur näherungsweise erreicht [15].

Eine typische Anwendung der Systemidentifikation ist die adaptive Echokompensation in Telefonsystemen. Insbesondere bei Satellitenübertragungen kann es durch akustische Gegebenheiten oder störende Eigenschaften der Harware zu Reflexionen kommen, die um bis zu $500\ ms$ verzögert bei den Gesprächsteilnehmern ankommen. Dies hat zur Folge, dass man ein Echo seiner eigenen Stimme hört, wodurch die Qualität des Gespräches stark beeinträchtigt wird. Die Aufgabe des adaptiven Filters besteht darin, den Echoanteil im Übertragungssignal zu identifizieren. Dieser Anteil wird dann vom Nutzanteil des Signals subtrahiert [16].

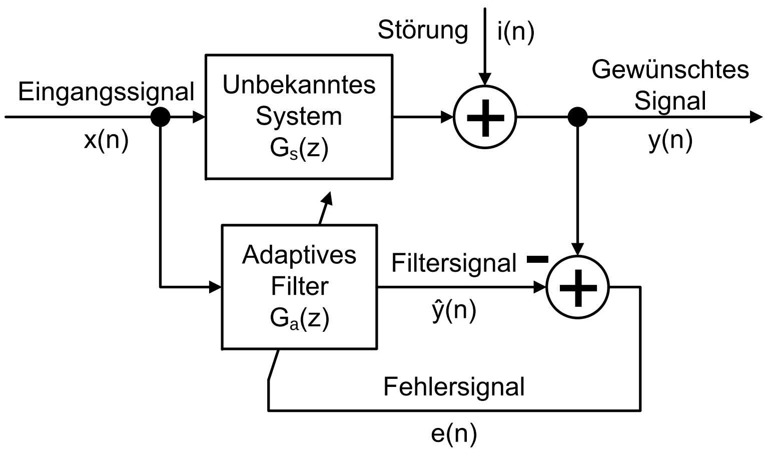

Abb. 2: System Identifikation: Modellierung des unbekannten Systems $G_s(z)$ durch das adaptive Filter $G_a(z)$ [10]

2.2 System-Inversion

Bei einer System-Inversion oder auch inversen Modellierung hat das adaptive Filter die Aufgabe, die inverse Übertragungsfunktion eines unbekannten Systems $G_s(z)$ nachzubilden (vgl. Abb. 3). Das Ausgangssignal $x(n)$ des unbekannten Systems $G_s(z)$ wird zum Eingangssignal des adaptiven Filters $G_a(z)$, dessen Ausgangssignal $\widehat{y}(n)$ sich dem verzögerten Eingangssignal $u(n)$ des Systems annähren soll. Die Verstärkung von $G_s(z) * G_a(z)$ wird für alle Frequenzen 1. Lediglich die resultierende Phasendrehung soll der Verzögerung $z^{-\Delta}$ entsprechen.

Auch in diesem System kann die Annäherung durch die Überlagerung des Filter-Eingangssignals $x(n)$ mit zusätzlichen Signalanteilen $i(n)$ erschwert werden. Die Verzögerung $z^{-\Delta}$ des System-Eingangssignals $u(n)$ setzt sich aus der Signallaufzeit von System und Filter zusammen [16].

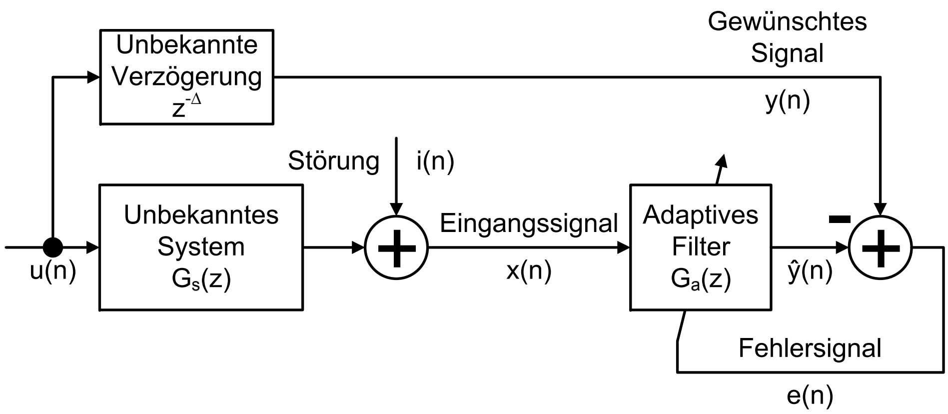

Abb. 3: System Inversion: Modellierung der inversen Übertragungsfunktion des unbekannten Systems $G_s(z)$ [10]

Ein Anwendungsbeispiel für die inverse Modellierung ist die adaptive Entzerrung bei der Datenübertragung über nichtideale bandbeschränkte Kanäle, wie zum Beispiel die Telefonleitung [16]. Verzerrungen des Übertragungssignals, die durch die Übertragungsfunktion $G_s(z)$ des unbekannten Übertragungskanals entstehen, werden durch die inverse Modellierung dieser Übertragungsfunktion kompensiert.

2.3 Signal-Prädiktion

Das Ziel bei der Signal-Prädiktion ist die Berechnung des Wertes $x(n_0)$ eines stochastischen Signals aus einer Reihe von aufeinanderfolgen Signalwerten $x(n), n_1 \leq n \leq n_2$. Dabei gibt es drei unterschiedliche Arten von Prädiktion:

- $n_0 > n_2$: Berechnung eines Signalwertes aus vergangenen Werten
- $n_0 < n_1$: Berechnung eines vorhergehenden Signalwertes aus nachfolgenden Werten
- $n_1 < n_0 < n_2$: Berechnung eines Signalwertes zwischen zwei anderen Werten, wobei der Wert bei $n = n_0$ nicht zur Berechnung genutzt wird

Die am häufigsten verwendete Prädiktion ist die lineare Prädiktion $n_0 > n_2$ (vgl. Abb. 4) [10].

In diesem Fall soll der zukünftige Verlauf des Eingangssignals $x(n)$ aus dem bisherigen Verlauf vorhergesagt werden. Das Eingangssignal $x(n)$ entspricht dem gewünschten Signal und wird daher zur Berechnung des Fehlersignals $e(n)$ verwendet. Am Eingang des adaptiven Filters liegen vergangene Werte von $x(n)$ an. Durch diesen Systemaufbau erhält man am Ausgang des adaptiven Filters die Vorhersage des Eingangssignals $\widehat{x}(n)$. Je nach Anwendung ist man entweder an der Signal-Vorhersage $\widehat{x}(n)$, am Prädiktionsfehler $e(n)$ oder an den Filterkoeffizienten interessiert, die das Signal beschreiben.

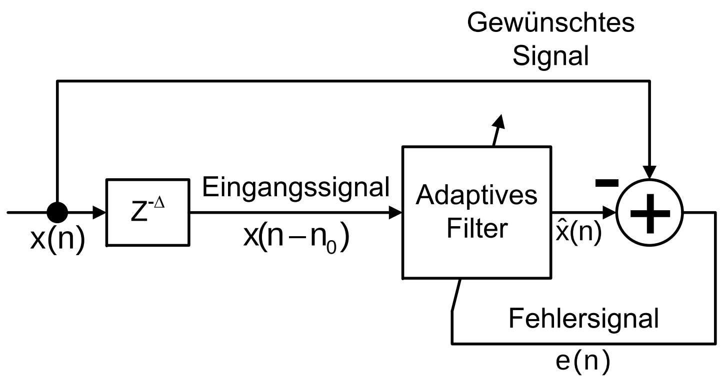

Abb. 4: Signal Prädiktion: Vorhersage des Eingangssignals $x(n)$ [10]

Ein typisches Beispiel für die Anwendung der linearen Prädiktion ist die *Linear Predictive Coding* (LPC)-Analyse [21] [26] von Sprachsignalen. Dabei soll der aktuelle Abtastwert eines Sprachsignals aus vergangenen Werten vorhergesagt werden. Auf diese Weise kann das Signal durch einen reduzierten Datensatz, den Filterkoeffizienten des Prädiktorfilters charakterisiert werden. Diese Modellierung des Sprachsignals durch einen Satz von Parametern kommt in unterschiedlichen Anwendungen zum Tragen. Bei der automatischen Spracherkennung und der Sprechverifizierung geschieht die Erkennung beispielsweise nicht direkt anhand des Sprachsignals, sondern mittels des durch die Prädiktion extrahierten Parametersatzes.

Eine weitere Anwendung ist die Übertragung und damit verbundene Kodierung von Sprachsignalen. Mit der Modellierung des Signals durch die Parameter lässt sich eine deutliche Datenreduktion erzielen [16]. Diese Technologie wird im Mobilfunkbereich genutzt und ermöglicht, dass bei gleicher Bandbreite der Übertragungskanäle mehr Teilnehmer kommunizieren können [21].

2.4 Aktive Geräuschkompensation

Das Ziel in dieser Klasse von Anwendungen ist es, ein Primärsignal $d(n)$ von unerwünschten Störungen zu befreien (vgl. Abb. 5). Das Primärsignal $d(n)$ besteht aus der Überlagerung des gewünschten Nutzsignals $n(n)$ mit dem Störsignal $s(n)$. Die Eingangssignale

$x_1(n) \ldots x_M(n)$ $(M \geq 1)$ des adaptiven Filters sind Referenzen für das Störsignal $s(n)$. Das adaptive Filter hat die Aufgabe, aus diesen Referenzen den Störanteil $s(n)$ im Primärsignal $d(n)$ zu modellieren. Diese Näherung $y(n)$ des Störsignals $s(n)$ wird dem Primärsignal $d(n)$ überlagert, wodurch das Fehlersignal $e(n)$ entsteht, welches im Idealfall dem gewünschten Nutzsignal $n(n)$ entspricht.

Die Referenzsignale $x_1(n) \ldots x_M(n)$ werden über Sensoren aufgenommen, bei denen das Nutzsignal $n(n)$ nicht oder nur so schwach ankommt, dass es ignoriert werden kann. Durch die Messung der Korrelation zwischen Primärsignal $d(n)$ und Referenzsignalen $x_1(n) \ldots x_M(n)$ wird die Näherung $y(n)$ des Störanteils $s(n)$ im Primärsignal $d(n)$ berechnet [10].

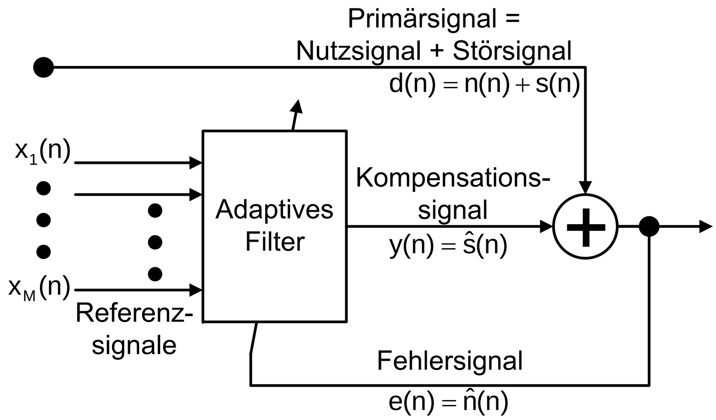

Abb. 5: Multisensor Störsignal Kompensation: Kompensation eines Störsignals im Primärsignal [10]

Die bekannteste Anwendung ist die aktive Geräuschkompensation oder auch *Active Noise Cancellation* (ANC). Die Grundidee der aktiven Geräuschkompensation ist die Erzeugung eines zum Störgeräusch $s(n)$ um 180° phasenverschobenen Gegengeräusches $y(n)$, das die gleiche Frequenz und die gleiche Amplitude wie das Störgeräusch $s(n)$ besitzt. Durch die Überlagerung löschen sich beide Geräusche gegenseitig aus (vgl. Abb. 6) [10].

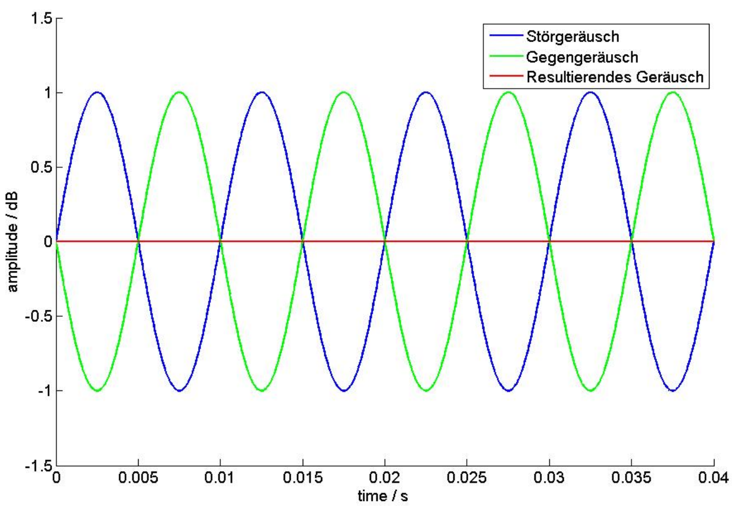

Abb. 6: Auslöschung einer Sinusschwingung

2.4.1 Adaptive Feedback System

Es gibt zwei grundsätzlich unterschiedliche Arten von ANC-Systemen. Eines davon ist das *Adaptive Feedback System* (vgl. Abb. 7).

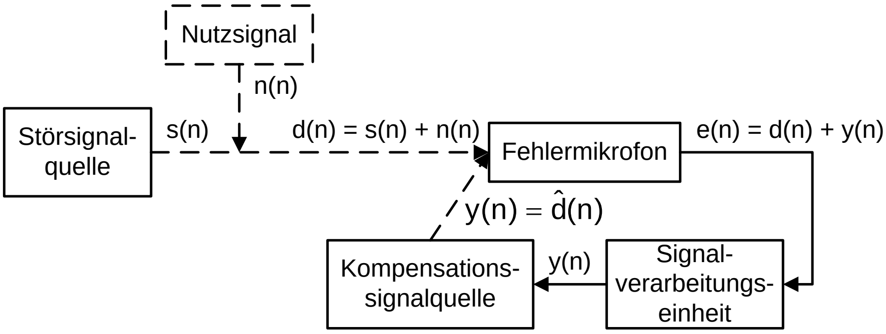

Abb. 7: Adaptive Feedback System

Das System besteht aus einer Störsignalquelle, einem Fehlermikrofon, einer Signalverarbeitungseinheit und einer Kompensationssignalquelle. Das Fehlermikrofon nimmt das Signal $e(n)$ auf, das aus der Überlagerung von Primärsignal $d(n)$ und Kompensationssignal $y(n)$ resultiert. Dieses Signal $e(n)$ soll minimiert werden. Dazu erzeugt die Signalverarbeitungseinheit ein Kompensationssignal $y(n)$, das dem Primärsignal $d(n)$ angeglichen, und um 180° phasenverschoben wird [8]. Das System versucht daher das komplette Signal $d(n)$ auszulöschen, das aus dem Störsignal $s(n)$ besteht, jedoch auch einen Nutzsignalanteil $n(n)$ enthalten kann.

2.4.2 Adaptive Feedforward System

Die zweite Variante ist das *Adaptive Feedforward System*. Dieses System verfügt, zusätzlich zu den Komponenten eines Feedback Systems, über ein Referenzmikrofon (vgl. Abb. 8).

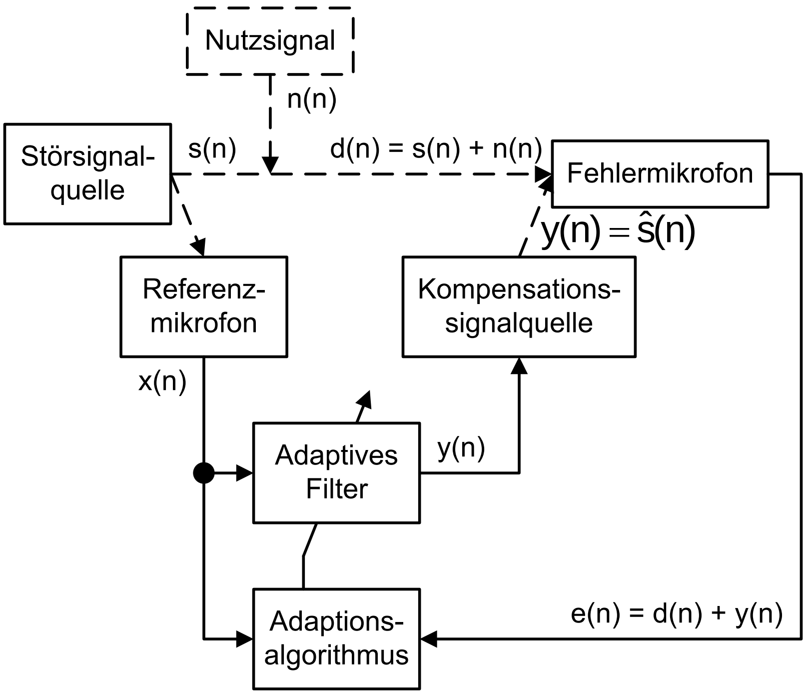

Abb. 8: Adaptive Feedforward System

Dieses Referenzmikrofon befindet sich direkt an der Störsignalquelle und stellt dem System das Störsignal $s(n)$ an der Quelle als Referenzsignal $x(n)$ zur Verfügung. Am Punkt des Empfängers wird das Kompensationssignal $y(n)$ dem Primärsignal $d(n)$ überlagert und das Signal, das durch die Überlagerung entsteht, wird dem System als Fehlersignal $e(n)$ zugeführt.

Das adaptive Filter hat dabei die Aufgabe, die Übertragungsstrecke des Störsignals $s(n)$ vom Referenzaufnahmepunkt bis zum Fehlersignalaufnahmepunkt zu modellieren und gleichzeitig für eine Phasenverschiebung um 180° zu sorgen. Um dies zu erreichen werden die Koeffizienten des Filters dahingehend adaptiert, dass ein minimales Fehlersignal $e(n)$ entsteht.

Wenn das Störsignal $s(n)$ mit einem Nutzsignal $n(n)$ überlagert ist, welches nicht ausgelöscht werden soll, ist dieses System in der Lage nur das Störsignal $s(n)$ zu entfernen, da das Störsignal $s(n)$ an seinem Entstehungspunkt bekannt ist. Das Ziel ist auch bei diesem System das Fehlersignal $e(n)$ zu minimieren. Im Gegensatz zum Feedback System wird das Primärsignal $d(n)$ jedoch nicht vollständig ausgelöscht, sondern durch Überlagerung mit dem Kompensationssignal $y(n)$ dem Nutzsignal $n(n)$ angeglichen.

Bei dem im Rahmen dieser Arbeit entwickelten System handelt es sich um ein Feedforward System, da es über ein Referenzsignal und ein Fehlersignal verfügt. Beide Signale werden über jeweils ein Mikrofon aufgenommen.

3 Adaptive Filterung in einem Adaptive Feedforward System

Ein adaptives Filtersystem besteht im wesentlichen aus zwei Komponenten, dem digigtalen Filter und dem Adaptionsalgorithmus (vgl. Abb. 9).

Bei dem gezeigten Sytsem handelt es sich um ein adaptives Filter zur aktiven Geräuschkompensation in einem Adaptive Feedforward System. Es bekommt zwei Signale zur Verfügung gestellt, das Fehlersignal $e(n)$ und das Störsignal $s(n)$ an der Entstehungsquelle, welches das Referenzsignal $x(n)$ darstellt. Das Primärsignal $d(n)$ setzt sich aus der Überlagerung von Nutzsignal $n(n)$ und Störsignal $s(n)$ zusammen. Die Aufgabe des Systems ist, das beim Empfänger ankommende Primärsignal $d(n)$ vom Störsignal $s(n)$ zu befreien.

Das digitale Filter soll dabei die unbekannte Übertragungsstrecke des Störsignals $s(n)$ von seiner Quelle bis zum Empfänger modellieren, so dass das Ausgangssignal $y(n)$ des Filters der Näherung des Sörsignals $s(n)$ beim Empfänger entspricht. Zur Minimierung des Fehlersignals $e(n)$ werden die Koeffizienten $w(n)$ des Filters durch den Adaptionsalgorithmus angepasst. In der Theorie wird dabei das Fehlersignal $e(n)$ durch die Subtraktion des Kompensationssignals $y(n)$ vom Primärsignal $d(n)$ berechnet.

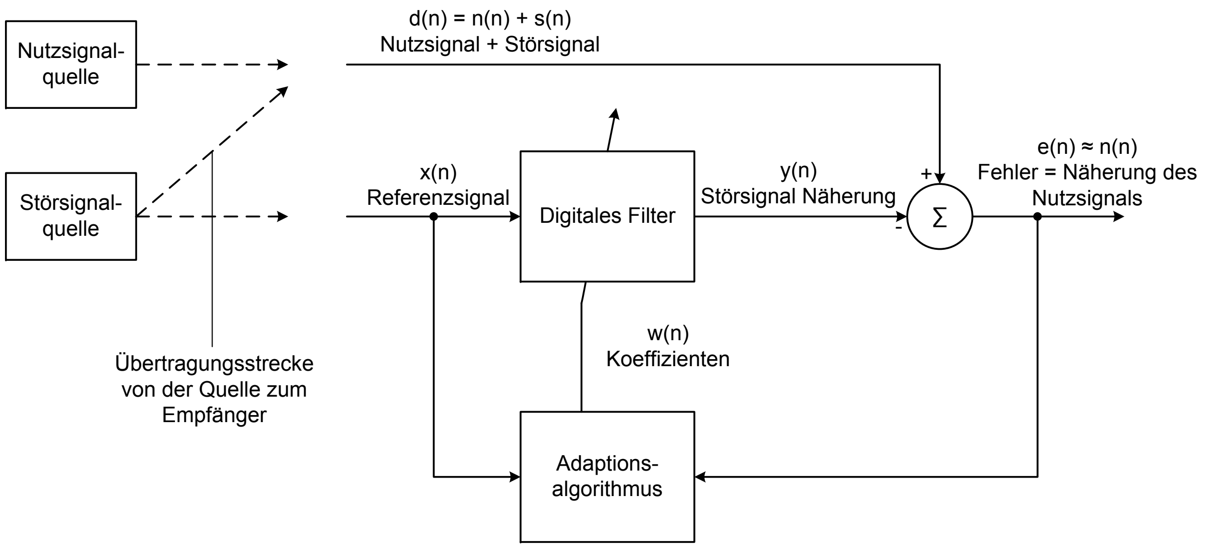

Abb. 9: Adaptives Filter in einem Adaptive Feedforward ANC-System

In diesem Kapitel werden zunächst die Vorteile einer FIR-Filterstruktur für die Filterkomponente erläutert. Anschließend wird die Adaptionsmethode der Filterkoeffizieten beschrieben, bevor auf die Leistungskriterien der Adaptionsmethode und ihre Einflussgrößen eingegangen wird.

3.1 Das adaptive FIR-Filter

Adaptive Filter können nach einer nichtrekursiven FIR-, oder einer rekursiven IIR-Struktur aufgebaut sein. Das FIR-Filter besitzt eine endliche Impulsantwort, die durch einen Koeffizientesatz $c(0), c(1), ..., c(M)$ beschrieben wird. Anhand der z-Transformierten der Impulsantwort

$$H(z) = \sum_{n=0}^{M} c(n) z^{-n} = c(0) \frac{\sum_{n=0}^{M} \frac{c(n)}{c(0)} z^{M-n}}{z^M} = c(0) \frac{\prod_{i=1}^{M} (z - z_i)}{z^M} \tag{1}$$

ist zu erkennen, dass die Übertragungsfunktion $H(z)$ eines FIR-Filters mit der Ordnung M, über M reelle oder konjugiert komplexe Nullstellen und M Vielfach-Pole im Ursprung der komplexen z-Ebene verfügt. Da alle Pole im Ursprung liegen, ist das FIR-Filter *absolut stabil*.

Diese absolute Stabilität, die einfache mathematische Handhabung und Implementierbarkeit des FIR-Filters, sowie die Möglichkeit, auf einfache Weise ein Filter mit linearer Phase zu entwerfen, führen dazu, dass die meisten adaptiven Filter auf einer FIR-Struktur basieren [16].

Ein FIR-Filter berechnet seine Ausgangswerte durch die Faltung der Eingangswerte $x(n)$ mit den Filterkoeffizienten $w(n)$. In einem ANC System wird das Gegensignal $y(n)$ daher berechnet durch Gleichung 2

$$y(n) = \sum_{i=0}^{N-1} w_{i+1}(n)x(n-i) \tag{2}$$

mit

$$w_i = c(i-1) \qquad\qquad i = 1, ..., N \tag{3}$$

wobei das Kompensationssignal $y(n)$ der Annäherung des Störsignals $s(n)$ beim Empfänger entspricht, das Eingangssignal $x(n)$ entspricht dem Störsignal $s(n)$ an seiner Quelle und $w(n)$ sind die Koeffizienten oder auch Gewichte des Filters mit der Ordnung $N-1$ (vgl. Abb. 10). In vektorieller Form

$$y(n) = \mathbf{w}^T(n)\mathbf{x}(n) \tag{4}$$

ist $\mathbf{w}^T(n)$ der transponierte Koeffizienten-Vektor und $\mathbf{x}(n)$ der Eingangssignal-Vektor [10]. Diese Schreibweise wird in den folgenden Gleichungen verwendet, um die Gleichungen übersichtlicher zu gestalten.

In Abbildung 10 ist zu erkennen, dass aus der Subtraktion des Filter-Ausgangssignals $y(n)$ vom gewünschten Signal $d(n)$ das Fehlersignal $e(n)$ berechnet wird, aus dem wiederum ein Gütemaß für den Algorithmus zur Adaption der Filterkoeffizienten $w(n)$ gebildet wird. In den folgenden Kapiteln wird die Ermittlung dieses Gütemaßes und des zugehörigen Adaptionsalgorithmus erläutert.

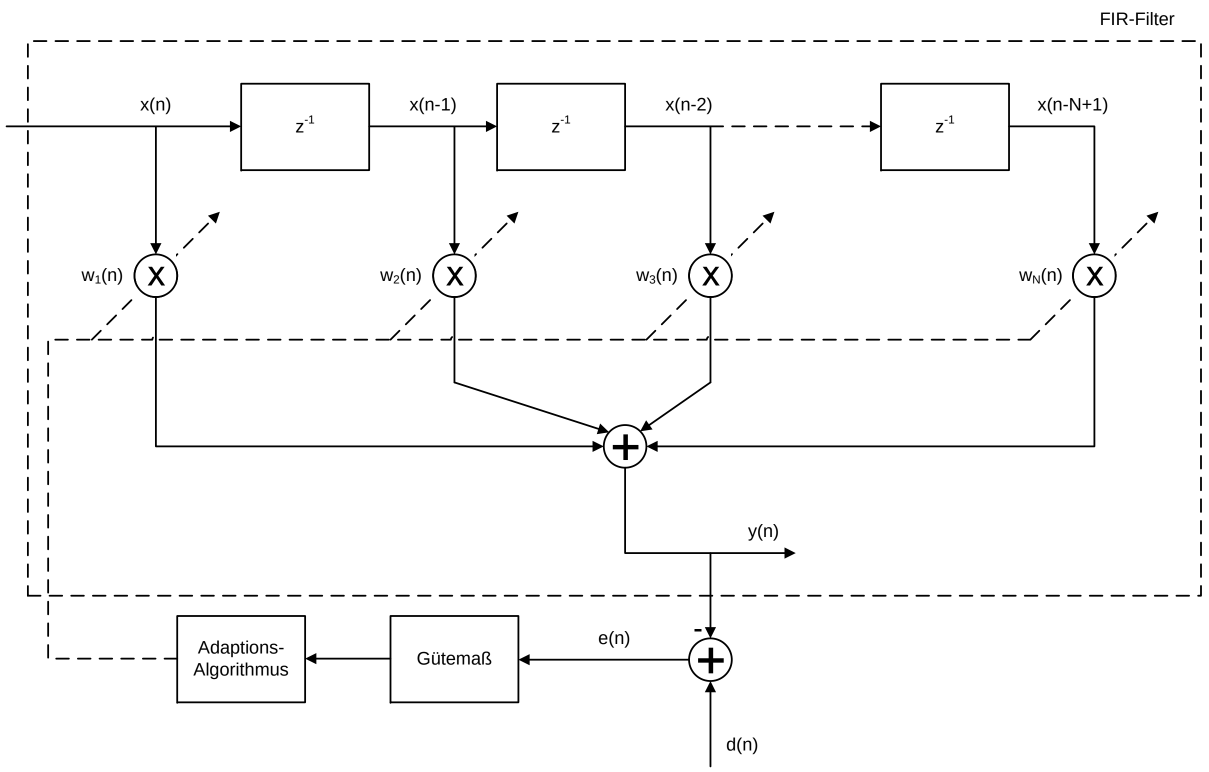

Abb. 10: Adaptives FIR-Filter in Direktform mit der Ordnung $N-1$

3.2 Lineare optimale Filterung

Adaptive Filter beruhen auf dem gleichen Optimalitätskriterium wie zeitinvariante, lineare optimale Filter. Diese sind optimal im Sinne der Minimierung des sogenannten mittleren quadratischen Fehler oder auch *Mean Square Error* (MSE) [10] [16].

Die Abhängigkeit des MSE von den Filterkoeffizienten $\mathbf{w}$ wird durch die sogenannte Fehlerfunktion $J(\mathbf{w})$ beschrieben. Dies ist eine quadratische Funktion, durch die eine mathematisch einfache Bestimmung der Filterkoeffizienten $\mathbf{w}$ ermöglicht wird, da sie in den meisten Fällen ein eindeutiges globales Minimum besitzt. Das Minimum kann durch die Berechnung des Gradienten $\nabla\mathbf{w}\{J(\mathbf{w})\}$ und dessen Gleichsetzung mit dem Nullvektor ermittelt werden. Ein solches optimales Filter wird *Wiener-Filter* genannt. Da das *Wiener-Filter* zeitinvariant ist, wird beim Koeffizientevektor $\mathbf{w}$ der diskrete Zeitindex n vorläufig weggelassen [16].

Für die im Folgenden beschriebene Herleitung des Adaptionsalgorithmus wird das FIR-basierte *Wiener-Filter* zu Grunde gelegt.

3.2.1 Linear MSE Abschätzung

Das Fehlersignal $e(n)$ beschreibt die Abweichung des Filterausgangs $y(n)$ vom Primärsignal $d(n)$ und berechnet sich durch Gleichung 5

$$e(n) = d(n) - y(n) \tag{5}$$

Durch Einsetzen von Gleichung 4 in Gleichung 5 folgt Gleichung 6

$$e(n) = d(n) - \mathbf{w}^T\mathbf{x}(n) \tag{6}$$

Der quadrierte Fehler $e^2(n)$ ergibt sich daher aus Gleichung 7

$$e^2(n) = d^2(n) - 2d(n)\mathbf{x}^T(n)\mathbf{w} + \mathbf{w}^T\mathbf{x}(n)\mathbf{x}^T(n)\mathbf{w} \tag{7}$$

Der mittlere quadratische Fehler J ergibt sich aus den Erwartungswerten der einzelnen Terme aus Gleichung 7

$$\begin{aligned} J &= E[e^2(n)] \\ &= E[d^2(n)] - 2E[d(n)\mathbf{x}^T(n)\mathbf{w}] + E[\mathbf{w}^T\mathbf{x}(n)\mathbf{x}^T(n)\mathbf{w}] \\ &= \sigma^2 + 2\mathbf{p}^T\mathbf{w} + \mathbf{w}^T\mathbf{R}\mathbf{w} \end{aligned} \tag{8}$$

wobei $E[\cdots]$ den Erwartungswert symbolisiert, σ^2 die Varianz des Primärsignals $d(n)$, $\mathbf{p}$ den Kreuzkorrelationsvektor der Länge N und $\mathbf{R}$ die $N \times N$ Autokorrelationsmatrix [5].

Gleichung 8 gilt unter den Bedingungen, dass das Eingangssignal $x(n)$ und das Primärsignal $d(n)$ gemeinsam schwach stationär sind, und das Primärsignal $d(n)$ reell und mittelwertfrei ist [16]. Diese Bedingungen werden zur weiteren Berechnung vorausgesetzt. Da unter diesen Umständen der mittlere quadratische Fehler J nur vom Gewichtsvektor $\mathbf{w}$ abhängt, kann dieser als quadratische Funktion $J(\mathbf{w})$ des Gewichtsvektors des FIR-Filters betrachtet werden.

$$J(\mathbf{w}) = \sigma^2 + 2\mathbf{p}^T\mathbf{w} + \mathbf{w}^T\mathbf{R}\mathbf{w} \tag{9}$$

Im nächsten Kapitel wird die Minimierung dieser Funktion beschrieben.

3.2.2 Berechnung des *Wiener-Filters* durch Minimierung der Fehlerfunktion *J(w)*

Die Fehlerfunktion $J(\mathbf{w})$ lässt sich in Abhängigkeit der Koeffizienten $\mathbf{w}$ als Fehlerfläche im $N+1$-dimensionalen Raum visualisieren. Auf dieser Fläche existiert ein absolutes Minimum bei $\mathbf{w}_o$, dem optimalen Gewichtsvektor.

Bei einem FIR-Filter 1. Ordnung wird die Fehlerfläche im 3-dimensionalen Raum dargestellt (vgl. Abb. 11).

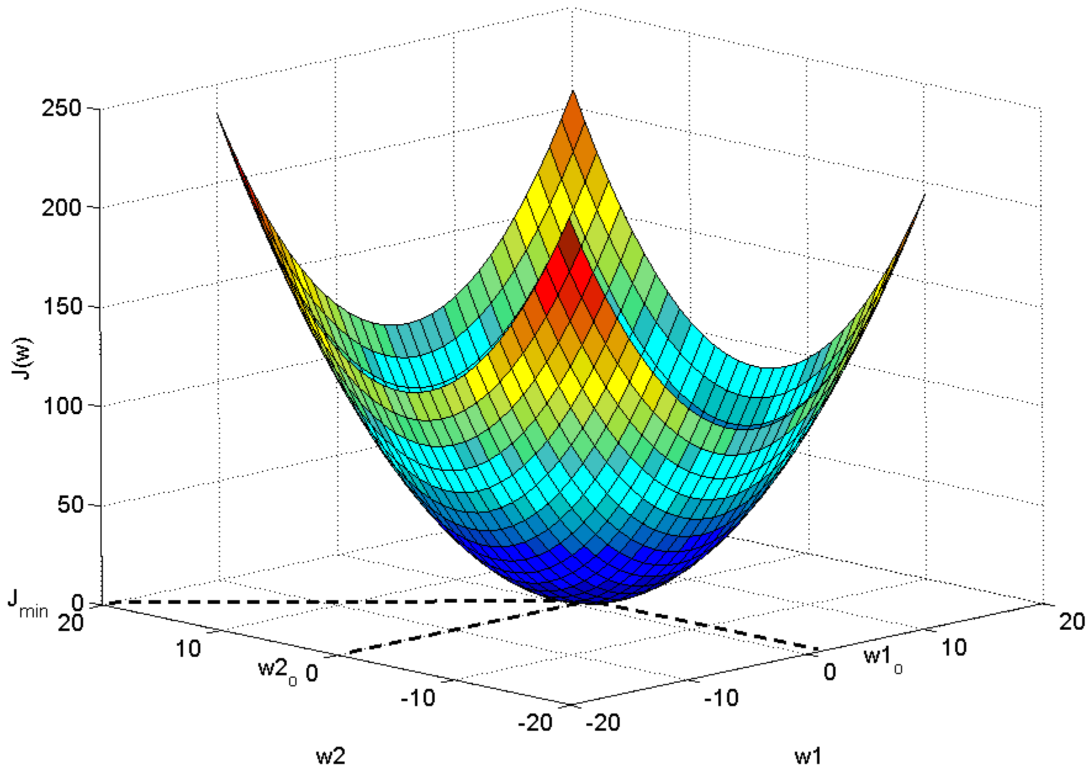

Abb. 11: Fehlerfläche bei N = 2 Koeffizienten

Um die sogenannte *Wiener-Lösung* $\mathbf{w}_o$ für die Koeffizienten zu finden, wird das Minimum J_{min} der Fehlerfunktion gesucht. Die Funktion $J(\mathbf{w})$ nimmt ihr Minimum an, wenn ihr Gradient den Wert 0 hat.

$$\nabla\mathbf{w}\ \{J(\mathbf{w})\}|_{\mathbf{w}=\mathbf{w}_o} = \mathbf{0} \tag{10}$$

Der Gradient $\nabla\mathbf{w}\{J(\mathbf{w})\}$ der Fehlerfunktion errechnet sich durch

$$\nabla\mathbf{w}\ \{J(\mathbf{w})\}|_{\mathbf{w}=\mathbf{w}_o} = \frac{\delta J(\mathbf{w})}{\delta\mathbf{w}} = -2\mathbf{p} + 2\mathbf{R}\mathbf{w} \tag{11}$$

Setzt man Gleichung 11 in Gleichung 10 ein und stellt nach $\mathbf{w}$ um, erhält man die sogenannte *Wiener-Hopf-Gleichung*

$$\mathbf{w}_o = \mathbf{R}^{-1}\mathbf{p} \tag{12}$$

Durch diese lassen sich die optimalen Koeffizienten $\mathbf{w}_o$ für das *Wiener-Filter* bestimmen, mit denen man den kleinsten mittleren quadratischen Fehler J_{min}, oder auch *Least Mean Square Error* (LMS) erhält [15].

3.3 Der LMS-Adaptionsalgorithmus

Die optimalen Filterkoeffizienten $\mathbf{w}_o$ im Sinne der Minimierung des MSE werden durch die *Wiener-Hopf-Gleichung* (Gleichung 12) berechnet. Diese Gleichung hat jedoch Nachteile beim Einsatz in einem praktischen System, denn sie benötigt die Autokorrelationsmatrix $\mathbf{R}$ und den Kreuzkorrelationsvektor $\mathbf{p}$, die beide nicht von vornherein bekannt sind. Außerdem muss die Matrix $\mathbf{R}$ invertiert werden, was sehr viel Rechenzeit verlangt. In einer Umgebung, in der sich die Statistik der Eingangssignale $x(n)$ und $d(n)$ ändert, wodurch sich $\mathbf{R}$ und $\mathbf{p}$ ändern und dadurch die Fehlerfläche im Raum wandern würde, müsste $\mathbf{w}_o$ permanent neu berechnet werden [16][5].

Für Echtzeit-Anwendungen werden daher Algorithmen verwendet, die die optimalen Koeffizienten bei jedem Samplingschritt neu berechnen, ohne **R** und **p** direkt zu berechnen oder eine Matrix-Inversion durchführen zu müssen.

Ein solcher Algorithmus ist der LMS-Adaptionsalgorithmus. Für die Herleitung des Algorithmus wird zunächst vom sogenannten *Newton-Verfahren* ausgegangen und durch schrittweise Vereinfachung erhält man über das *Gradienten-Verfahren* den LMS-Algorithmus [16].

Im Anschluss an die Herleitung wird auf die Konvergenzeigenschaften des Algorithmus eingegangen. Dabei werden die Faktoren beschrieben, von denen die Konvergenz zur *Wiener-Lösung*, die Konvergenzzeit und die Stabilität des Filters abhängen.

3.3.1 Herleitung des LMS-Algorithmus aus dem Newton- und Gradienten-Verfahren

Sowohl beim *Newton-*, als auch beim *Gradienten-Verfahren* basiert die Adaption des Koeffizientenvektors auf dem Gradienten $\nabla \mathbf{w}\{J(\mathbf{w})\}$ der Fehlerfunktion (Gleichung 11), wodurch mit **R** und **p** die Bekanntheit der Statistik der Eingangssignale $x(n)$ und $d(n)$ vorausgesetzt wird. Um dies zu vermeiden, wird beim LMS-Algorithmus lediglich eine Schätzung des Gradienten $\nabla \mathbf{w}\{J(\mathbf{w})\}$ verwendet. Um den Zusammenhang zwischen der Methode des steilsten Abstieges oder auch *Method Of Steepest Descent* und dem LMS-Algorithmus zu verdeutlichen, wird zunächst das *Newton-Verfahren* beschrieben, bei dem die Nullstelle des Gradienten $\nabla \mathbf{w}\{J(\mathbf{w})\}$ gesucht wird, um das Minimum auf der Fehlerfläche zu finden [16].

Der Gradient $\nabla \mathbf{w}\{J(\mathbf{w})\}$ der Fehlerfunktion ist

$$\nabla \mathbf{w}\{J(\mathbf{w})\} = -2\mathbf{p} + 2\mathbf{R}\mathbf{w} = 2(\mathbf{R}\mathbf{w} - \mathbf{p}) \tag{13}$$

mit Gleichung 12

$$\nabla \mathbf{w}\{J(\mathbf{w})\} = 2\mathbf{R}(\mathbf{w} - \mathbf{w}_o) \tag{14}$$

und aufgelöst nach $\mathbf{w}_o$ erhält man

$$\mathbf{w}_o = \mathbf{w} - \frac{1}{2}\mathbf{R}^{-1}\nabla \mathbf{w}\{J(\mathbf{w})\} \tag{15}$$

Diese Gleichung führt in einem einzelnen Schritt von einem beliebigen Startvektor **w** ausgehend zum optimalen Koeffizientenvektor $\mathbf{w}_o$. Um einen iterativen Algorithmus zu erhalten, wird der Schrittweitenfaktor $\mu > 0$ eingeführt, der, wie später gezeigt wird, die Konvergenz und Stabilität des Algorithmus reguliert. Zu jedem Zeitpunkt n wird eine Iteration ausgeführt (Gl. 16)

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \frac{\mu}{2}\mathbf{R}^{-1}\nabla \mathbf{w}\ \{J(\mathbf{w})\}|_{\mathbf{w}=\mathbf{w}(n)} \tag{16}$$

Dies wird als *Newton-Verfahren* bezeichnet [10][16].

Wie aus Gleichung 16 hervorgeht, wird für die Adaption durch das *Newton-Verfahren* weiterhin die inverse Autokorrelationsmatrix $\mathbf{R}^{-1}$ benötigt. Die Autokorrelationsmatrix **R** kann jedoch positiv semidefinit und damit nicht invertierbar sein. Des Weiteren ist die Invertierung einer Matrix bei grossen Dimensionen $N \times N$ sehr rechenaufwendig.

Es wird daher ein Algorithmus gesucht, der ohne $\mathbf{R}^{-1}$ auskommt. Dieser Algorithmus soll, ausgehend von einem Startkoeffizientenvektor $\mathbf{w}(0)$, in mehreren Schritten $\mathbf{w}(1), \mathbf{w}(2), \ldots$ das Minimum der Fehlerfunktion J_{min} suchen und sich dadurch mit jedem Schritt dem optimalen Koeffizientenvektor $\mathbf{w}_o$ annähern.

Dazu wird bei jedem Schritt die Ableitung der Fehlerfunktion $J(\mathbf{w})$ nach **w** berechnet. Ist die

Ableitung positiv, steigt die Fehlerfunktion $J(\mathbf{w})$ an dieser Stelle und **w** muss im nächsten Schritt abnehmen. Im umgekehrten Fall muss **w** zunehmen. Dies wird erreicht durch

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \frac{\mu}{2} \left. \frac{\delta J(\mathbf{w})}{\delta \mathbf{w}} \right|_{\mathbf{w}=\mathbf{w}(n)} \tag{17}$$

Wie in Gleichung 11 erkennbar, entspricht die Ableitung von $J(\mathbf{w})$ nach **w** dem Gradienten $\nabla\mathbf{w}\{J(\mathbf{w})\}$ der Fehlerfunktion und Gleichung 17 lässt sich schreiben als

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \frac{\mu}{2} \nabla\mathbf{w} \left.\{J(\mathbf{w})\}\right|_{\mathbf{w}=\mathbf{w}(n)} \tag{18}$$

Dies wird als *Gradienten-Verfahren* oder auch *Steepest Descent Algorithm* (SDA) bezeichnet [16].

Auf diesem Algorithmus basiert der LMS-Adaptionsalgorithmus, bei dem der Gradient von $J = e^2(n)$ als Näherung des Gradienten von $J = E[e^2(n)]$ betrachtet wird. Der Gradient $\nabla\mathbf{w}\{J(\mathbf{w})\}$ kann beschrieben werden durch Gleichung 19

$$\nabla\mathbf{w}\{J(\mathbf{w})\} = \left[\frac{\delta E[e^2(n)]}{\delta w_1}, \frac{\delta E[e^2(n)]}{\delta w_2}, \ldots, \frac{\delta E[e^2(n)]}{\delta w_N}\right]^T \tag{19}$$

Die Näherung des Gradienten ergibt sich daher aus Gleichung 20

$$\begin{aligned} \widehat{\nabla}\mathbf{w}\{J(\mathbf{w})\} &= \left[\frac{\delta e^2(n)}{\delta w_1}, \frac{\delta e^2(n)}{\delta w_2}, \ldots, \frac{\delta e^2(n)}{\delta w_N}\right]^T \\ &= 2e(n)\left[\frac{\delta e(n)}{\delta w_1}, \frac{\delta e(n)}{\delta w_2}, \ldots, \frac{\delta e(n)}{\delta w_N}\right]^T \end{aligned} \tag{20}$$

Mit Gleichung 6, welche die Berechnung des Fehlers $e(n)$ beschreibt, folgt daraus

$$\widehat{\nabla}\mathbf{w}\{J(\mathbf{w})\} = -2e(n)\frac{\delta e(n)}{\delta \mathbf{w}} = -2e(n)\mathbf{x}(n) \tag{21}$$

Wenn der Gradient $\nabla\mathbf{w} \left.\{J(\mathbf{w})\}\right|_{\mathbf{w}=\mathbf{w}(n)}$ in Gleichung 18 durch diese Näherung ersetzt wird, erhält man die Adaptionsgleichung des LMS-Algorithmus [15]

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \frac{\mu}{2}\widehat{\nabla}\mathbf{w} \left.\{J(\mathbf{w})\}\right|_{\mathbf{w}=\mathbf{w}(n)} = \mathbf{w}(n) + \mu e(n)\mathbf{x}(n) \tag{22}$$

3.3.2 Konvergenz des LMS-Algorithmus

Die durch das *Newton-Verfahren*, den *SDA*- und den LMS-Algorithmus berechneten Filterkoeffizieten **w** konvergieren nach einer bestimmten Zeit zu den optimalen Koeffizienten $\mathbf{w}_o$ der *Wiener-Lösung*. Das *Newton-Verfahren* erreicht das Optimum auf direktem Weg, der *SDA-Algorithmus* findet es auf einem kleinen Umweg und der LMS-Algorithmus bewegt sich im Mittel um die Lösung des *SDA-Algorithmus* und somit nach einer bestimmten Anzahl an Schritten auch um das Optimum (vgl. Abb. 12). Die Konvergenz der Adaptionsalgorithmen hängt von dem Schrittweitenfaktor μ ab. Um eine Grenze für diese Schrittweite μ zu finden, werden zunächst die Konvergenzeigenschaften des SDA betrachtet. Die Konvergenzbedingungen dieses Algorithmus lassen sich im Mittel direkt auf den LMS-Algorithmus übertragen [16].

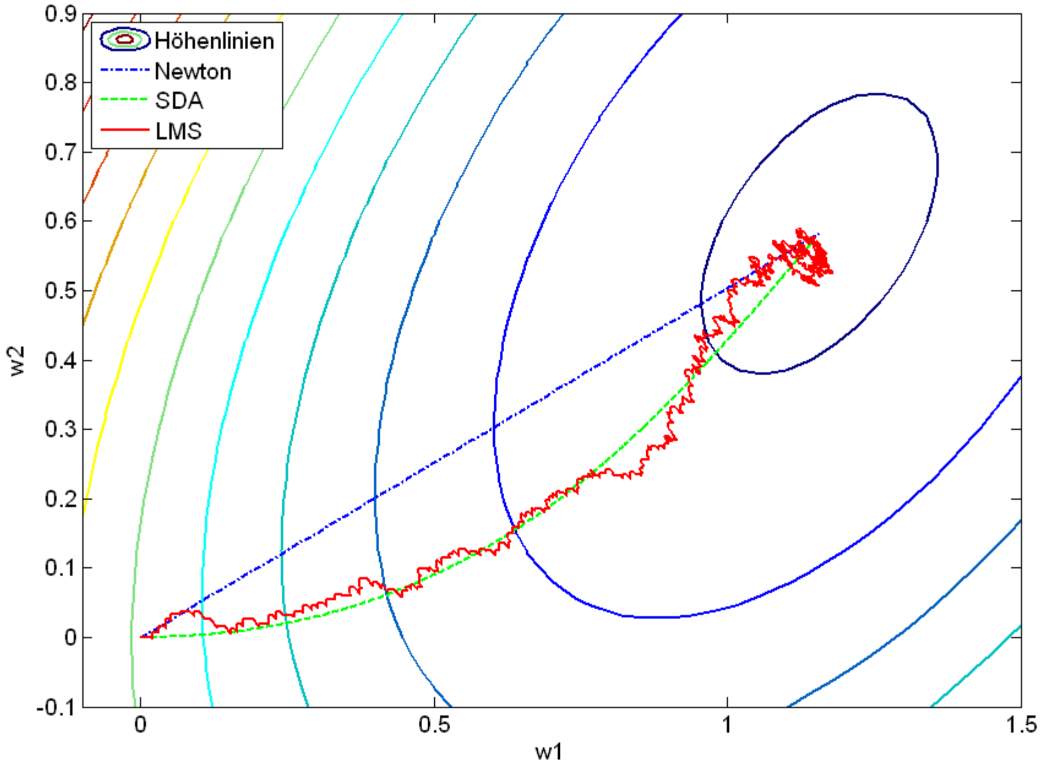

Abb. 12: Rekursionsverlauf des Newton-, SDA- und LMS-Algorithmus auf den Höhenlinien der Fehlerfläche

Der Gradient der Fehlerfunktion $J(\mathbf{w})$ bei der n-ten Iteration wird berechnet durch

$$\nabla \mathbf{w} \left\{J(\mathbf{w})\right\}|_{\mathbf{w}=\mathbf{w}(n)} = -2\mathbf{p} + 2\mathbf{R}\mathbf{w}(n) \tag{23}$$

Eingesetzt in die Gleichung des SDA (18) erhält man

$$\begin{aligned} \mathbf{w}(n+1) &= \mathbf{w}(n) - \mu(-\mathbf{p} + \mathbf{R}\mathbf{w}(n)) \\ &= \mathbf{w}(n) + \mu\mathbf{p} - \mu\mathbf{R}\mathbf{w}(n) \end{aligned} \tag{24}$$

Durch Einsetzen der Einheitsmatrix $\mathbf{I}$ lässt sich Gleichung 24 umschreiben

$$\mathbf{w}(n+1) = (\mathbf{I} - \mu\mathbf{R})\mathbf{w}(n) + \mu\mathbf{p} \tag{25}$$

Mit einem Startvektor $\mathbf{w}(0) = \mathbf{0}$ folgt für jeden weiteren Schritt

$$\begin{aligned} \mathbf{w}(1) &= (\mathbf{I} - \mu\mathbf{R})\mathbf{w}(0) + \mu\mathbf{p} = \mu\mathbf{p} \\ \mathbf{w}(2) &= \mu(\mathbf{I} - \mu\mathbf{R})\mathbf{p} + \mu\mathbf{p} \\ \mathbf{w}(3) &= (\mathbf{I} - \mu\mathbf{R})\mathbf{w}(2) + \mu\mathbf{p} \\ &= (\mathbf{I} - \mu\mathbf{R})[\mu(\mathbf{I} - \mu\mathbf{R})\mathbf{p} + \mu\mathbf{p}] + \mu\mathbf{p} \\ &= \mu(\mathbf{I} - \mu\mathbf{R})^2\mathbf{p} + \mu(\mathbf{I} - \mu\mathbf{R})\mathbf{p} + \mu\mathbf{p} \\ &= \mu[(\mathbf{I} - \mu\mathbf{R})^2 + (\mathbf{I} - \mu\mathbf{R}) + \mathbf{I}]\mathbf{p} \\ \mathbf{w}(n) &= \mu[(\mathbf{I} - \mu\mathbf{R})^{n-1} + (\mathbf{I} - \mu\mathbf{R})^{n-2} + \ldots + \mathbf{I}]\mathbf{p} \\ &= \mu\left[\sum_{m=0}^{n-1}(\mathbf{I} - \mu\mathbf{R})^m\right]\mathbf{p} \end{aligned} \tag{26}$$

Mit der Annahme, dass $(\mathbf{I} - \mu\mathbf{R})$ kleiner als eins ist, in dem Sinne, dass

$$\|(\mathbf{I} - \mu\mathbf{R})\mathbf{v}\| < \|\mathbf{v}\| \tag{27}$$

lässt sich Gleichung 26 umformen nach

$$\begin{aligned} \mathbf{w}(n) &= \mu\left([\mathbf{I} - (\mathbf{I} - \mu\mathbf{R})^n][\underbrace{\mathbf{I} - (\mathbf{I} - \mu\mathbf{R})}_{\mu\mathbf{R}}]^{-1}\right)\mathbf{p} \\ &= [\mathbf{I} - (\mathbf{I} - \mu\mathbf{R})^n]\mathbf{R}^{-1}\mathbf{p} \end{aligned} \tag{28}$$

Durch die Annahme in Gleichung 27 folgt

$$\lim_{n\to\infty}(\mathbf{I} - \mu\mathbf{R})^n = \mathbf{0} \tag{29}$$

und mit Gleichung 28 und 29

$$\lim_{n\to\infty}\mathbf{w}(n) = \mathbf{R}^{-1}\mathbf{p} = \mathbf{w}_o \tag{30}$$

Durch Gleichung 30 wird bewiesen, dass die Rekursionsformel des SDA unter der Annahme 27 zum optimalen Koeffizientenvektor $\mathbf{w}_o$ konvergiert.

Um durch das Konvergenzverhalten auf eine Grenze für den Schrittweitenfaktor μ schliessen zu können, wird die Eigenvektortransformation angewandt

$$\mathbf{w}(n) = \mathbf{Q}\mathbf{w}'(n) \tag{31}$$

wobei $\mathbf{Q}$ die $N \times N$ Matrix der Eigenvektoren von $\mathbf{R}$ ist. Mit den Beziehungen

$$\mathbf{R} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^H = \sum_{i=1}^{N}\lambda_i\mathbf{q}_i\mathbf{q}_i^H \tag{32}$$

$$\mathbf{\Lambda} = \mathbf{Q}^H\mathbf{R}\mathbf{Q} \tag{33}$$

$$\mathbf{p} = \mathbf{Q}\mathbf{p}' \tag{34}$$

$$\mathbf{Q}^H\mathbf{Q} = \mathbf{I} \tag{35}$$

lässt sich die entkoppelte Form des SDA aufstellen

$$\mathbf{w}'(n+1) = (\mathbf{I} - \mu\mathbf{\Lambda})\mathbf{w}'(n) + \mu\mathbf{p}' \tag{36}$$

Da $\mathbf{I}$ und $\mathbf{\Lambda}$ Diagonaltmatrizen sind, können die entsprechenden N Gleichungen entkoppelt und dadurch einzeln betrachtet werden

$$w_i'(n+1) = (1 - \mu\lambda_i)w_i'(n) + \mu p_i' \tag{37}$$

Analog zur Gleichung 25 muss für Gleichung 37 die Bedingung

$$|1 - \mu\lambda_i| < 1 \tag{38}$$

erfüllt sein, damit jeder entkoppelte Koeffizient $w_i'(n)$ zu dem entsprechenden optimalen Koeffizienten w_{i_o}' konvergiert [16].

Daraus folgt für die Begrenzung des Schrittweitenfaktors μ

$$0 < \mu < \frac{2}{\lambda_{max}} = \mu_{max} \tag{39}$$

In der Praxis muss für μ jedoch ein deutlich kleinerer Wert gewählt werden, um eine zuverlässige Konvergenz zu erreichen. Die Simulation zeigt, dass bei einer Schrittweite von $\mu = \frac{2}{\lambda_{max}}$ der Algorithmus das Optimum nicht erreicht (vgl. Abb. 13).

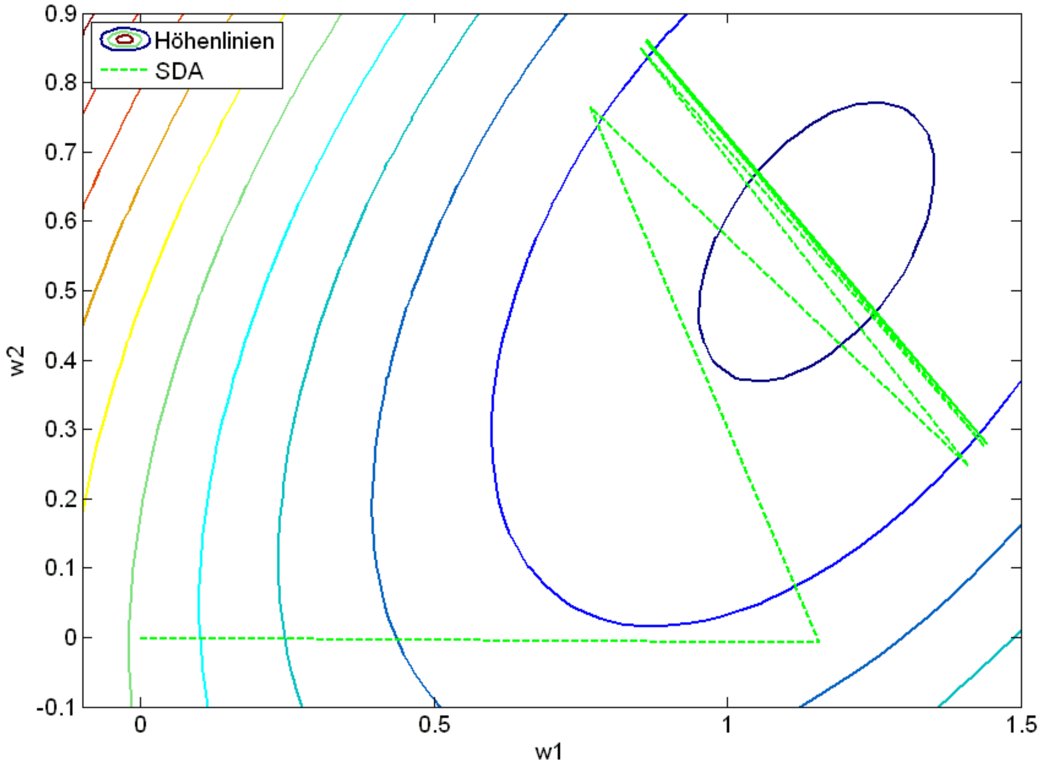

Abb. 13: Konvergenzverhalten des SDA-Algorithmus bei einer Schrittweite $\mu = \frac{2}{\lambda_{max}}$

Um diese Instabilität des Algorithmus zu verhindern, muss eine konservativere Grenze für μ gewählt werden. Dazu wird folgende Abschätzung betrachtet

$$\lambda_{max} \leq \sum_i \lambda_i = \sum_i (Diagonalelemente\ von\ \mathbf{\Lambda}) = Sp(\mathbf{\Lambda}) \tag{40}$$

Dabei wird $Sp(\mathbf{\Lambda})$ als Spur der Matrix $\mathbf{\Lambda}$ bezeichnet.
Da $Sp(\mathbf{AB}) = Sp(\mathbf{BA})$, gilt

$$\begin{aligned} Sp(\mathbf{\Lambda}) &= Sp(\mathbf{Q}^H\mathbf{RQ}) \\ &= Sp(\mathbf{R}\underbrace{\mathbf{QQ}^H}_{\mathbf{I}}) = Sp(\mathbf{R}) \\ &= \sum (Diagonalelemente\ von\ \mathbf{R}) \end{aligned} \tag{41}$$

mit Gleichung 40

$$\lambda_{max} \leq Sp(\mathbf{R}) = \sum (Diagonalelemente\ von\ \mathbf{R}) \tag{42}$$

Da jeder Diagonalwert $r(0)$ der Matrix $\mathbf{R}$ der mittleren Leistung $E[|x(n)|^2]$ des Eingangssignals $x(n)$ entspricht, lässt sich Gleichung 42 auch schreiben als

$$\lambda_{max} \leq Sp(\mathbf{R}) = N \cdot E[|x(n)|^2] = N \cdot (mittlere\ Eingangsleistung) \tag{43}$$

Daraus ergibt sich mit Gleichung 39 eine konservative Obergrenze für den Schrittweitenfaktor μ

$$0 < \mu < \mu_{max2} = \frac{2}{Sp(\mathbf{R})} \tag{44}$$

Für FIR-basierte adaptive Filter gilt

$$0 < \mu < \mu_{max2} = \frac{2}{N \cdot (mittlere\ Eingangsleistung)} \tag{45}$$

Diese Obergrenze gilt sowohl für den *SDA*- als auch im Mittel für den LMS-Algorithmus [16].

3.3.3 Normierung des LMS-Algorithmus

Um die Konvergenz der, durch den LMS-Algorithmus adaptierten, Filterkoeffizienten **w** zum Optimum zu gewährleisten, darf der Schrittenweitenfaktor μ die Obergrenze (Gl. 45)

$$0 < \mu < \frac{2}{N \cdot (mittlere\ Eingangsleistung)} \tag{46}$$

nicht überschreiten. In Systemen in denen die mittlere Eingangsleistung des Referenzsignals $x(n)$ variiert, muss die Schrittweite μ daher adaptiv gestaltet werden. Dabei wird die mittlere Eingangsleistung durch das Skalarprodukt

$$\mathbf{x}(n)^t\mathbf{x}(n) = \sum_{i=0}^{N-1} x^2(n-i) = x^2(n) + x^2(n-1) + \ldots + x^2(n-N+1) \tag{47}$$

$$\approx N \cdot E[x^2(n)] = N \cdot (mittlereEingangsleistung) \tag{48}$$

geschätzt. Damit gilt für die Schrittweite

$$\mu(n) = \frac{\beta}{\gamma + \mathbf{x}^t(n)\mathbf{x}(n)} \qquad 0 < \beta < 2 \tag{49}$$

wobei γ eine kleine positive Konstante ist, die verhindert das $\mu(n)$ zu groß wird, falls die Schätzung der mittleren Eingangsleistung einen sehr kleinen Wert ergibt. Die Adaption der Filterkoeffizienten erfolgt beim normierten LMS-Algorithmus somit durch

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \frac{\beta}{\gamma + \mathbf{x}^t(n)\mathbf{x}(n)} e(n)\mathbf{x}(n) \qquad 0 < \beta < 2 \tag{50}$$

Aus (Gl. 47) folgt die rekursive Gleichung zur Schätzung der mittleren Eingangsleistung

$$\mathbf{x}^t(n)\mathbf{x}(n) = \mathbf{x}^t(n-1)\mathbf{x}(n-1) + x^2(n) - x^2(n-N+1) \tag{51}$$

Dadurch wird der hohe Rechenaufwand von N Multiplikationen und N Additionen für die Berechnung von $\mathbf{x}^t(n)\mathbf{x}(n)$ bei jedem Adaptionsschritt auf zwei Multiplikationen, eine Addition und eine Subtraktion verringert [16][10].

4 Modellierung und Simulation des Feedforward Systems mit dem LMS-Algorithmus

Das Feedforward ANC-System setzt sich aus mehreren Hardware-Komponenten zusammen (vgl. Abb. 14). Durch diese Komponenten und ihre Verknüpfung miteinander entstehen für jedes System spezifische Signalübertragungseigenschaften, die messtechnisch ermittlet werden und in die Modellierung des Gesamtsystems eingehen. Hierzu wird zunächst der konzeptionelle Aufbau des Systems mit den verwendeten HW-Komponenten beschrieben.

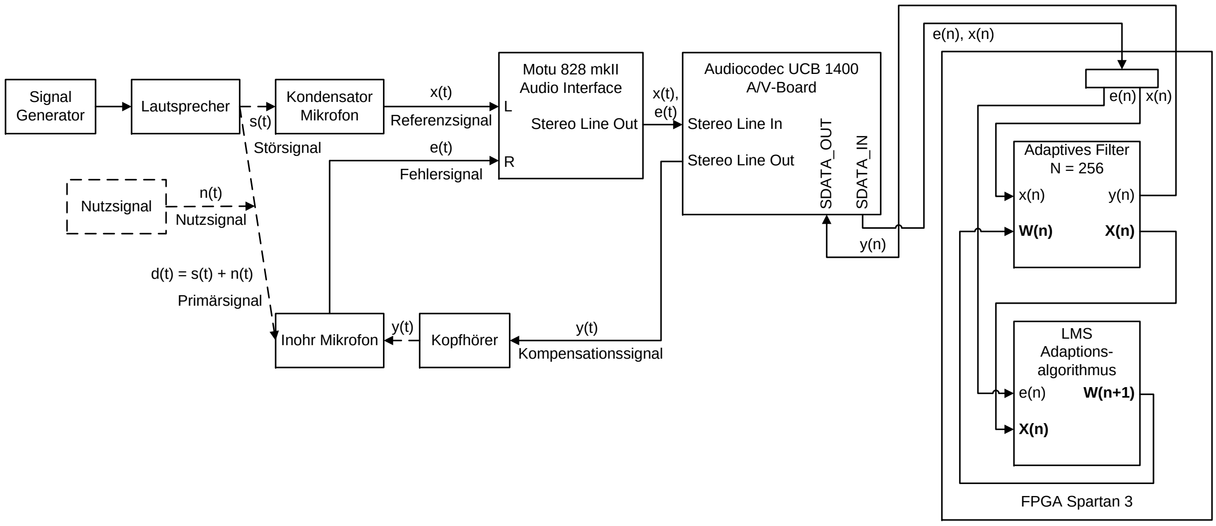

Abb. 14: Aufbau des Feedforward ANC-Systems mit allen verwendeten HW-Komponenten: Lautsprecher, Kondensatormikrofon, Audio-Interface, Audio-Codec, FPGA, Kopfhörer und Inohr-Mikrofon. Signalbezeichnungen mit dem Index t stellen analoge Signale dar, während der Index n ein digitales Signal bezeichnet.

Folgende Signale sind in dem System vorhanden:

- Störsignal $s(t)$; Wiedergabe über einen Lautsprecher
- Referenzsignal $x(t)$; vom Kondensatormikrofon aufgenommenes Störsignal $s(t)$
- Nutzsignal $n(t)$; Signalanteile in der Umgebung des Empfängers, die nicht zum Störsignal gehören
- Primärsignal $d(t)$; Überlagerung von Störsignal $s(t)$ und Nutzsignal $n(t)$ beim Empfänger
- Kompensationssignal $y(t)$; vom adaptiven Filter erzeugtes Signal zur Kompensation des Störsignals $s(t)$; wird über einen Kopfhörer wiedergegeben
- Fehlersignal $e(t)$; Überlagerung von Primärsignal $d(t)$ und Kompensationssignal $y(t)$; Aufnahme erfolgt durch ein Inohr-Mikrofon, das sich zwischen Kopfhörer und Trommelfell einer menschlichen Person befindet

Das Referenzsignal $x(t)$ und das Fehlersignal $e(t)$ werden an die Line-Eingänge eines Audio-Interfaces gelegt, um die beiden Monosignale zu einem Stereosignal zusammenzufassen, das an den Stereo-Line-Eingang des Audio-Codecs angelegt wird. Über das Audio-Interface werden außerdem die Pegel der aufgenommenen Signale eingestellt, welche diese am Ausgang des Interfaces besitzen.

Der Audio-Codec digitalisiert die analogen Eingangssignale $x(t)$ und $e(t)$ und überträgt sie über eine serielle Datenleitung zum FPGA, wo sie dem adaptiven Filter zur Verfügung gestellt werden. Das Filter erzeugt das Kompensationssignal $y(n)$, welches über eine zweite serielle

Datenleitung vom FPGA zum Codec übertragen wird. Der Codec wandelt das digitale Kompensationssignal $y(n)$ in ein analoges Signal $y(t)$ um, so dass es über den Kopfhörer, der an den Stereo-Line-Ausgang des Codecs angeschlossen ist, wiedergegeben wird (vgl. Abb. 14).

4.1 Übertragungs-Eigenschaften der verwendeten Hardwarekomponenten im Sekundärpfad

Einen störenden Einfluss auf die Funktion des System hat der sogenannte Sekundärpfad $S(z)$. Dieser stellt den Rückkopplungspfad des Kompensations- $y(t)$ und Fehlersignals $e(t)$ vom Ausgang des digitalen Filters bis zum Eingang des Fehlersignals $e(n)$ in den Adaptionsalgorithmus dar (vgl. Abb. 15). Die Übertragungsstrecke $S(z)$ dieses Pfades muss im Eingangspfad des Referenzsignals $x(n)$ zum Adaptionsalgorithmus nachgebildet werden, um ein stabiles und konvergierendes adaptives Filter zu erhalten. In Kombination mit dem LMS-Algorithmus kann $S(z)$ durch ein zusätzliches Filter mit der Übertragungsfunktion $\widehat{S}(z)$ modelliert werden. Dieser Algorithmus wird als *Filtered-x* oder auch *FxLMS*-Algorithmus bezeichnet [3] [9] [24].

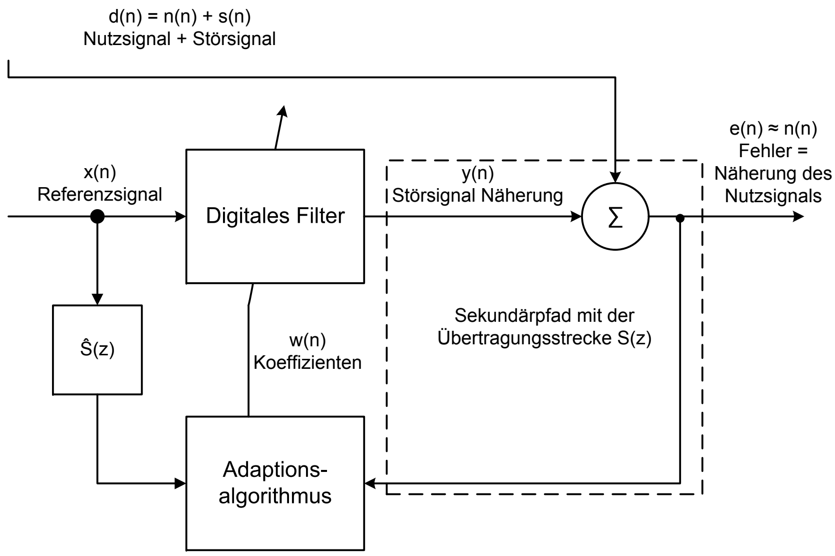

Abb. 15: Adaptives Filter mit einem zusätzlichen Filter $\widehat{S}(z)$ im Eingangspfad des Referenzsignals $x(n)$ zum Adaptionsalgorithmus, der die Übertragungsstrecke $S(z)$ des Sekundärpfades modelliert

Der Sekundärpfad in dem entwickelten ANC-System besteht aus den HW-Komponenten:

- Audio-Codec UCB1400
- Kopfhörer Sennheiser HD 600
- Inohr-Mikrofon Soundman OKM-II professional

Die Einflüsse dieses Pfades werden untersucht, da das entwickelte ANC-System eine physikalische Kompensation des Störsignals $s(t)$ durchführt. Im Gegensatz zu dem in [4] vorgestellten Modell, bei dem das Fehlersignal $e(n)$ durch die interne Subtraktion des digitalen Kompensationssignals $y(n)$ vom digitalen Primärsignal $d(n)$ entsteht, findet in dem in dieser Arbeit beschriebenen System eine akkustische Überlagerung der Signale $y(t)$ und $d(t)$ statt. Dies erfordert die Berücksichtigung sämtlicher, durch zusätzliche Hardware verursachten Signallaufzeiten, die bei einer digitalen Kompensation wie in [4] nicht vorhanden sind.

Die Eigenschaften des Sekundärpfades $S(z)$ wurden durch Messungen des Frequenzgangs, der Phase und der Gruppenlaufzeit des Audio-Codecs und der Kombination aus Audio-Codec, Kopfhörer und Inohr-Mikrofon ermittelt (vgl. Anhang B). Diese Messungen haben ergeben, dass sowohl Frequenzgang als auch Gruppenlaufzeit des Sekundärpfades $S(z)$ bis zu einer Frequenz von $4\ kHz$ annähernd konstant sind. Für eine Kompensation von Geräuschen unterhalb von $4\ kHz$ wird daher für die Modellierung lediglich die Gruppenlaufzeit berücksichtigt und der Frequenzgang des Sekundärpfades $S(z)$ als konstant angesehen.

Bei der aktiven Geräuschkompensation muss das Kompensationssignal $y(t)$ am Kompensationspunkt, der sich in diesem System am Inohr-Mikrofon befindet, eine Phasenverschiebung von 180° zum Störsignal $s(t)$ besitzen. Den größten Störeinfluss auf die Funktionalität des Systems haben daher die durch die einzelnen HW-Komponenten verursachten Signallaufzeiten.

Beispielsweise entspricht die 180° Phasenverschiebung, bei einem Störsignal $s(t)$ mit einer Freqeunz von $400\ Hz$, einer Verschiebung des Kompensationssignals $y(t)$ zum Störsignal von $1,25\ ms$. Wenn das Kompensationssignal $y(t)$ jedoch durch zusätzliche Signallaufzeiten verzögert wird, ist es nicht mehr um 180° phasenverschoben und kann daher das Störsignal $s(t)$ nicht kompensieren.

Zur Ermittlung der Signallaufzeiten des Primärsignals $d(t)$, des Referenzsignals $x(t)$, des Kompensationssignals $y(t)$ und des Fehlersignals $e(t)$, sind vier Zeitmessungen durchgeführt worden (vgl. Abb. 16).

Die Signallaufzeiten wurden durch eine Messkombination aus Signalgenerator und Oszilloskop gemessen. Dabei diente ein Rechtecksignal als Referenz und wurde direkt am Ausgang des Signalgenerators mit einem Oszilloskop gemessen. Der zweite Messpunkt befand sich am Ende eines Signallaufweges. Anhand dieser beiden Messpunkte konnte der zeitliche Abstand der Signale zueinander gemessen werden. Dieser Abstand entspricht der Laufzeit, die ein Signal für den Weg vom Start- bis zum Enpunkt benötigt (vgl. Anhang C Abb. 82 - 85).

Die in Abbildung 16 dargestellten Signalwege und Messergebnisse entsprechen folgenden Signallaufzeiten:

- $t1 = 296\ \mu s$: Referenzsignallaufzeit von der Störquelle bis zum Eingang des Audio-Codecs
- $t2 = 1230\ \mu s$: Audio-Codec Gruppenlaufzeit vom Eingang des Codecs bis zum Ausgang des Codecs
- $t3 = 1330\ \mu s$: Signalkreislaufzeit des im Fehlersignal $e(t)$ enthaltenen Kompensationssignals $y(t)$ vom Eingang des Audio-Codecs bis zurück zum Eingang des Audio-Codecs
- $t4 = 3200\ \mu s$: Signallaufzeit des im Fehlersignal $e(t)$ enthaltenen Störsignals $s(t)$ von der Störquelle bis zum Eingang des Audio-Codecs

Im folgenden Kapitel wird erläutert, wie die einzelnen Signallaufzeiten in der Modellierung des Systems berücksichtigt werden.

4.2 Simulink-Modell

In diesem Abschnitt wird die Modellierung des zuvor dargestellten Aufbaus in Simulink beschrieben. Durch die Simulation des Modells lässt sich feststellen, ob die gemessenen Laufzeiten zu einem unerwünschten Verhalten des Systems führen. Anschließend wird eine gegebenenfalls erforderliche Korrektur des Systems modelliert und in der Simulation überprüft.

Für die Modellierung des Systems wird das Referenz- $x(n)$[1] und das Primärsignal $d(n)$ aufgezeichnet. Die Aufzeichnung erfolgt in dem, im vorhergehenden Kapitel beschriebenen, Aufbau über das *Motu* Audio-Interface. Das Modell besteht aus diesen beiden Signalen, dem adaptiven Filter und der, durch eine Addition nachempfundenen, Überlagerung des Primär- $d(n)$ und Kompensationssignals $y(n)$ am Inohr-Mikrofon. Die Membranen des Kopfhörers, der das

[1] Da es sich in diesem Abschnitt um Simulationssignale handelt, werden diese durchgängig mit n indiziiert

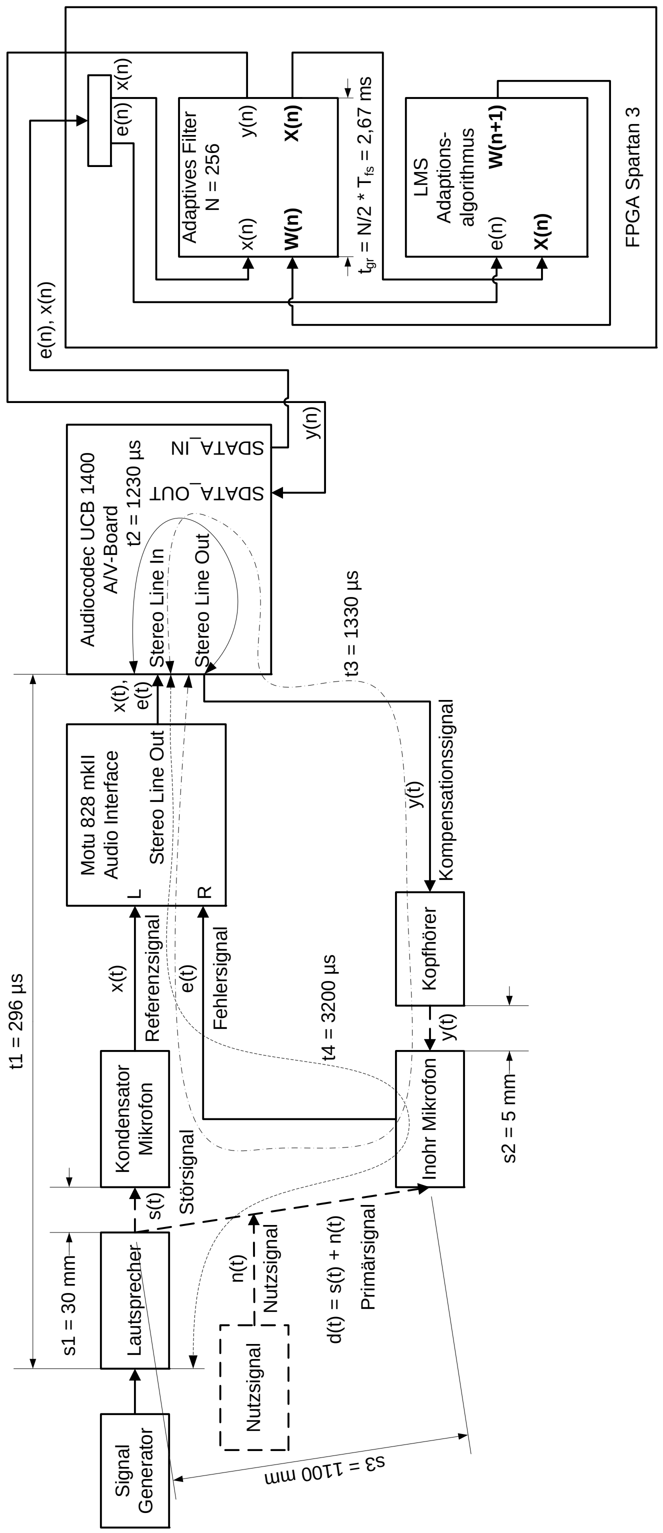

Abb. 16: ANC-System-Aufbau mit den gemessenen Signallaufzeiten: Störsignal $s(t)$ vom Ausgang des Signalgenerators, über Lautsprecher und Kondensatormikrofon, bis zum Ausgang des Audio-Interfaces ($t1$); Gruppenlaufzeit des Audio-Codecs ($t2$); Signalkreislaufzeit des Kompensationssignals $y(t)$ vom Eingang des Audio-Codecs, zum Ausgang des Codecs, über den Kopfhörer, das Inohr-Mikrofon und das Audio-Interface, zurück zum Eingang des Audio-Codecs ($t3$); Störsignal $s(t)$ vom Ausgang des Signalgenerators, über Lautsprecher und Inohr-Mikrofon, bis zum Ausgang des Audio-Interfaces ($t4$)

Kompensationssignal $y(n)$ wiedergibt, und des Inohrmikrofons besitzen eine gleichsinnige Polung, weshalb die Überlagerung als Addition modelliert wird. In der Theorie errechnet sich das Fehlersignal $e(n)$ allerdings aus der Subtraktion des Kompensationssignals $y(n)$ vom Primärsignal $d(n)$ (vgl. Kap. 3.2.1 Gl. 5). Die Subtraktion wird durch die 180° Phasenverschiebung des Kompensationssignals $y(n)$ erreicht, weshalb eine Invertierung des Ausgangssignal $y(n)$ in das adaptive Filter integriert ist (vgl. Anh. D Code 1).

Die Signallaufzeiten werden als Verzögerungsblöcke modelliert. Da die Aufnahme der Eingangssignale am Audio-Interface erfolgt, sind die Laufzeiten $t1$ und $t4$ bereits in den aufgezeichneten Signalen enthalten und müssen daher nicht modelliert werden. Zur Darstellung der Simulationsergebnisse werden Primär- $d(n)$, Referenz- $x(n)$, Kompensations- $y(n)$ und Fehlersignal $e(n)$, sowie Filterkoeffizienten in Workspace-Variablen gespeichert.

Ein Verzögerungsblock mit einer Übertragungsfunktion z^{-30} entspricht der Verzögerung eines Signals um 30 Abtastwerte. Die Simulation wird mit einer Abtastrate $f_s = 48\ kHz$ durchgeführt. Um die ermittelten Signallaufzeiten zu modellieren, werden diese daher in verzögerte Abtastwerte umgerechnet.

Umrechnungsbeispiel: $N = t2/T_{f_s} = 1230\ \mu s/20,83\ \mu s = 59,04 =$ *Anzahl der zu verzögernden Abtastwerte*

Die Signallaufzeit $t2 = 1230\ \mu s$ würde daher als Verzögerungsblock mit der Übertragungsfunktion z^{-59} modelliert werden. Die gemessenen Signallaufzeiten $t2$ und $t3$ entsprechen folgenden Übertragungsfunktionen:

- $t2 = 1230\ \mu s \equiv z^{-59}$
- $t3 = 1330\ \mu s \equiv z^{-64}$

Die Gruppenlaufzeit des Audio-Codecs $t2$ wird aufgeteilt in z^{-30} für den Eingangs- und z^{-29} für den Ausgangspfad. Die gemessene Signalkreislaufzeit des Kompensationssignals $y(n)$ $t3$ setzt sich daher zusammen aus der Verzögerung von z^{-29} im Ausgangspfad des Codecs und der Verzögerung von z^{-35} durch die Übertragung des Signals per Kopfhörer und Inohr-Mikrofon, sowie durch den Eingangspfad des Codecs (vgl. Abb. 17).

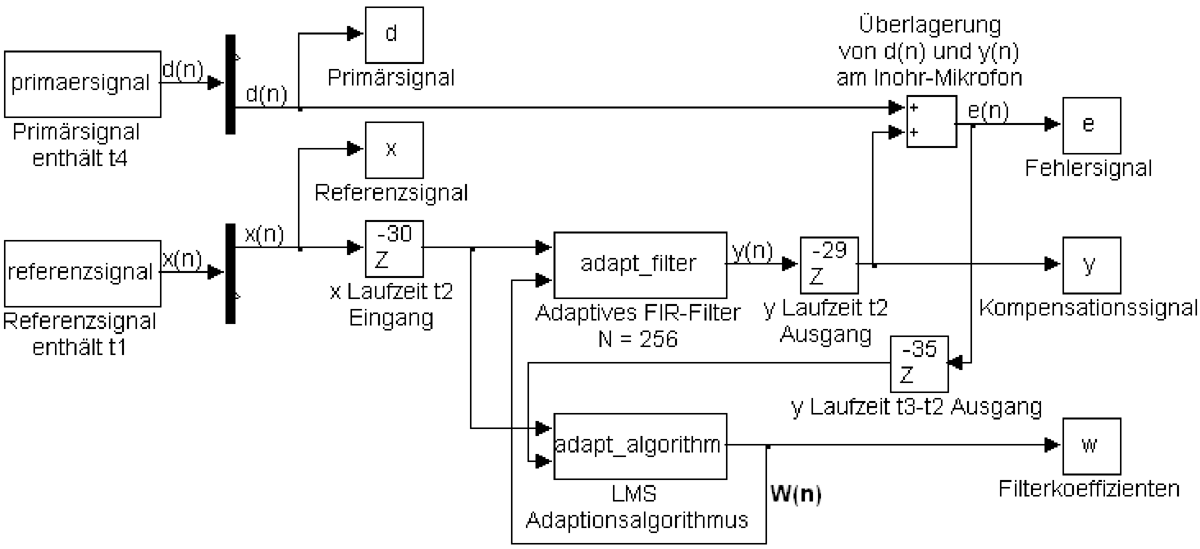

Abb. 17: Simulink-Modell des Feedforward ANC-Systems mit den am Audio-Interface aufgenommenen Primär- und Referenzsignalen, sowie den ermittelten Signallaufzeiten

Dabei gilt die Annahme, dass sich die Gruppenlaufzeit des Codecs zu annähernd gleichen Teilen auf Eingangs- und Ausgangspfad aufteilt. Sollte dies nicht der Realität entsprechen, würde sich die Verzögerung des Referenzsignals $x(n)$ im Eingangspfad und das Verhältnis der Verzögerungen des Kompensationssignals $y(n)$ im Ausgangspfad zu dem des Fehlersignals $e(n)$ im Eingangspfad ändern.

Das System funktioniert jedoch unabhängig von der Verzögerung des Referenzsignals $x(n)$ im Eingang. Entscheidend für die Stabilität des Systems ist lediglich die Gesamtverzögerung des Kompensationssignals $y(n)$ im Sekundärpfad, die sowohl Eingangs- als auch Ausgangsverzögerung des Codecs enthält.

Die Übertragunstrecke $S(z)$ des Sekundärpfades setzt sich in diesem System zusammen aus dem Übertragungsverhalten des Audiocodecs, des Kopfhörers, des Inohr-Mikrofons und der Strecke zwischen Kopfhörer und Inohr-Mikrofon. Um den zusätzlichen Rechenaufwand durch die Implementierung eines weiteren Filters zu vermeiden, wird lediglich eine, der Signallaufzeit des Sekundärpfades $t3$ entsprechende, Verzögerung des Referenzsignals $x(n)$ in den Eingangspfad des LMS-Algorithmus integriert (vgl. Abb. 18), dabei gilt die Annahme $S(z) = z^{-64}$.

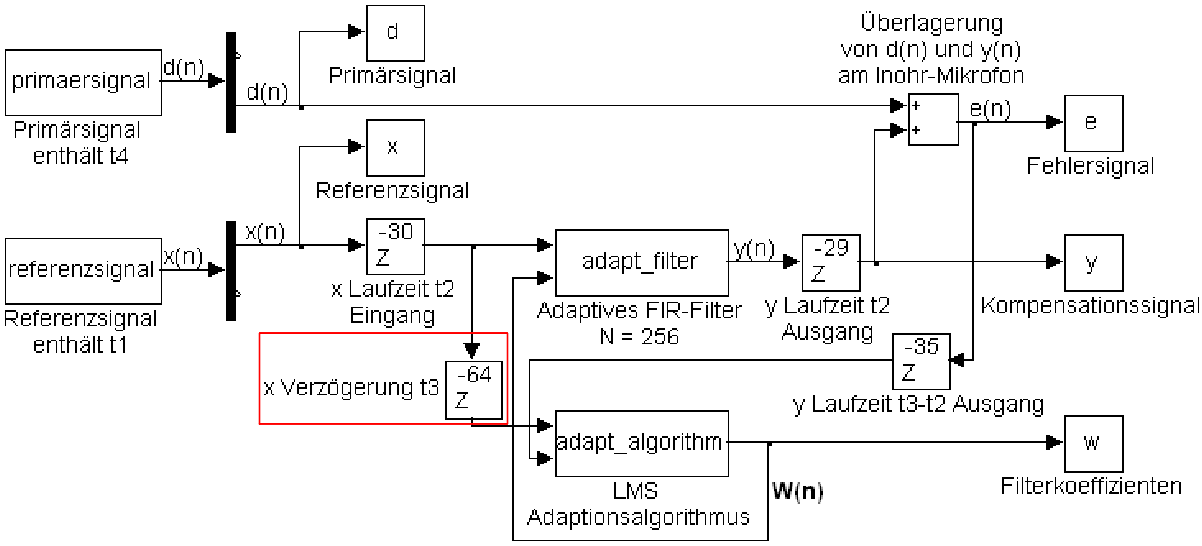

Abb. 18: Simulink-Modell mit der Verzögerung des Referenzsignals im Eingangspfad des Adaptionsalgorithmus

Die Blöcke für das adaptive FIR-Filter und den LMS-Adaptionsalgorithmus bestehen aus sogenannten *Level-2 S-Functions* [1][11]. Dies sind Matlab-Funktionen, die in das Simulink-Modell integriert werden. Auf diese Weise wurden die Algorithmen für das FIR-Filter (vgl. Anhang D.1) und den LMS-Algorithmus (vgl. Anhang D.2) implementiert. Über den Block *adapt_algorithm* lässt sich der Schrittweitenfaktor μ als Parameter einstellen.

4.3 Simulationsergebnisse

In diesem Abschnitt werden die Ergebnisse der Simulationen des entwickelten Simulink System-Modells präsentiert und erläutert. Die Simulation zeigt, dass die Modellierung des Sekundärpfades $S(z)$ für die Stabilität des Systems unerlässlich ist. Für die Simulation wird ein Sinuston als Störsignal $s(n)$ erzeugt und das Referenz- $x(n)$ sowie das Primärsignal $d(n)$, wie in Kapitel 4.2 beschrieben, aufgenommen. Folgende Parameter wurden gewählt:

- Frequenz des Störsignals $s(n)$ $f_{st} = 400\ Hz$
- Ordnung des FIR-Filters $N = 256$
- Schrittweitenfaktor des Adaptionsalgorithmus $\mu = 1/4096$
- Simulationszeit $t_{sim} = 30\ ms$

Die Filterordnung wurde mit $N = 256$ der Ordnung für die Implementierung des Filters auf dem FPGA entsprechend gewählt. Der Schrittweitenfaktor μ wurde mit $1/4096$ sehr klein gewählt, da er während der Adaption konstant ist und sich nicht an die Eingangsleistung des Referenzsignals $x(n)$ anpassen kann. Um die Konvergenz des Algorithmus trotzdem sicherzustellen, ist die Wahl der Schrittweite μ konservativ ausgefallen.

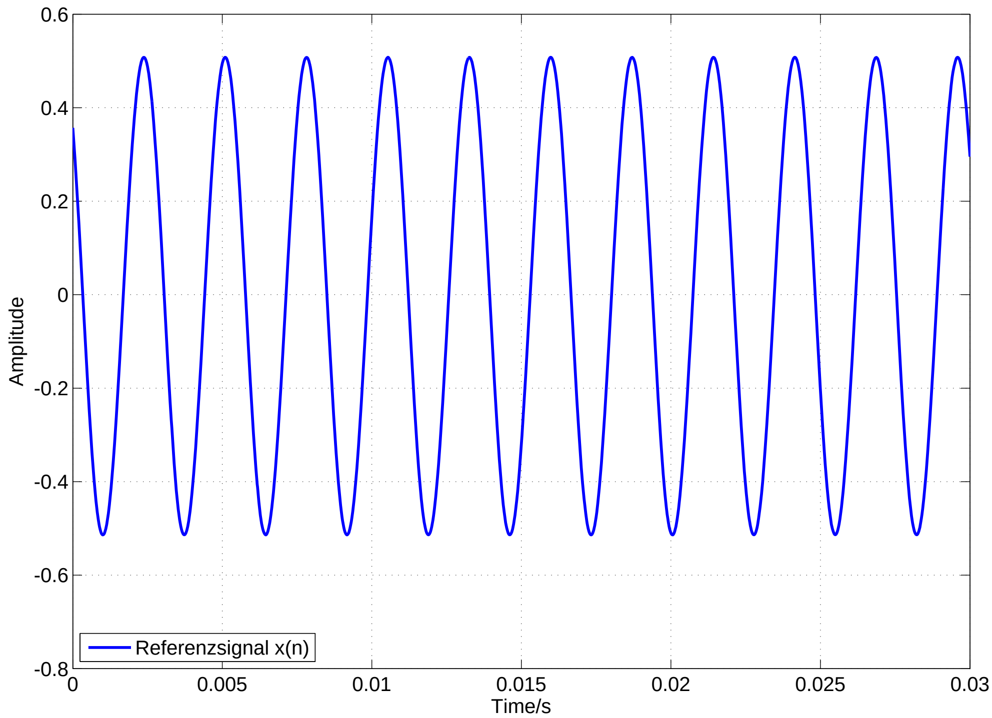

Abb. 19: Referenzsignal $x(n)$*,* $f = 400\ Hz$

Zunächst wurde eine Simulation ohne die Modellierung der Verzögerungszeiten durchgeführt, um die Funktion des Modells grundsätzlich zu überprüfen. Die Signalkurven des Primärsignals $d(n)$ und des Kompensationssignals $y(n)$ zeigen, dass das adaptive Filter aus dem Referenzsignal $x(n)$ (vgl. Abb. 19) ein zu $s(n)$ um 180° phasenverschobenes Gegensignal $y(n)$ erzeugt hat (vgl. Abb. 20).

Die Adaption der Filterkoeffizienten beginnt direkt nach dem Start der Simulation und nach 30 ms haben sie sich an das Optimum angenähert (vgl. Abb. 22), wodurch eine deutliche Abschwächung des Fehlersignals $e(n)$ erreicht wird (vgl. Abb. 21). Das bedeutet eine geringe Adaptionszeit trotz der kleinen Schrittweite $\mu = 1/4096$.

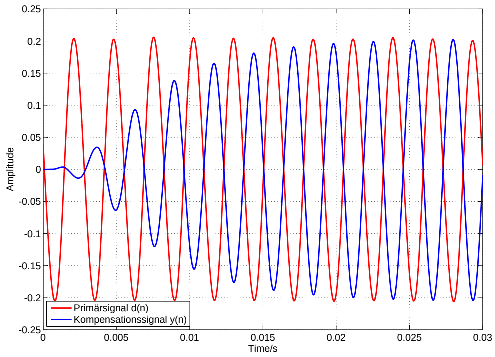

Abb. 20: Primärsignal $d(n)$ aufgenommen am Inohr-Mikrofon und Kompensationssignal $y(n)$ erzeugt vom adaptiven Filter

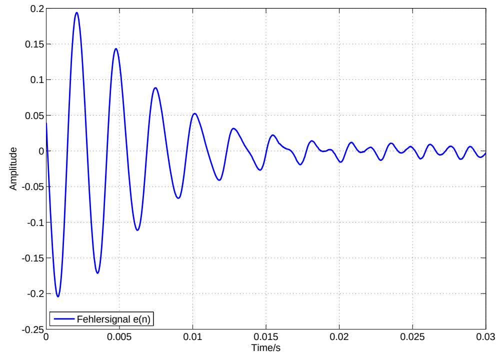

Abb. 21: Fehlersignal $e(n)$ nach der Überlagerung von $d(n)$ mit $y(n)$

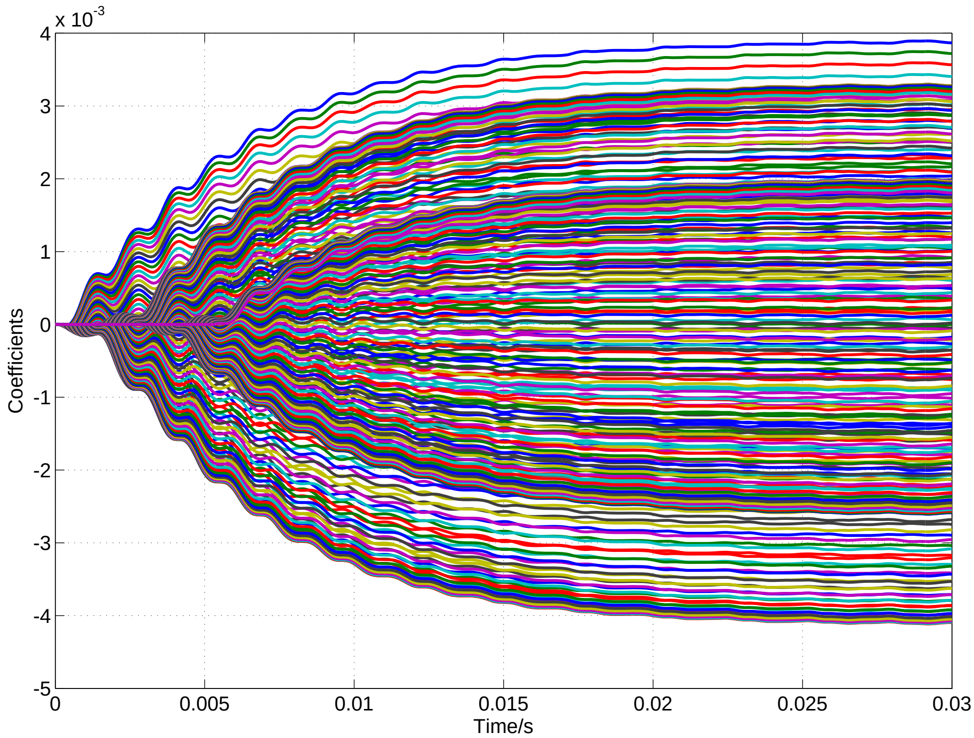

Abb. 22: Adaptierte Filterkoeffizienten $w_0(n) - w_{256}(n)$ des FIR-Filters

Die nächste Simulation wurde mit den durch Verzögerungsblöcke modellierten Signallaufzeiten durchgeführt. Die Filterkoeffizienten erreichen keinen stationären Wert, sondern werden exponentiell größer (vgl. Abb. 25). Dies hat zur Folge, dass auch das Kompensationssignal $y(n)$ und das Fehlersignal $e(n)$ exponentiell ansteigen (vgl. Abb. 23 und 24).

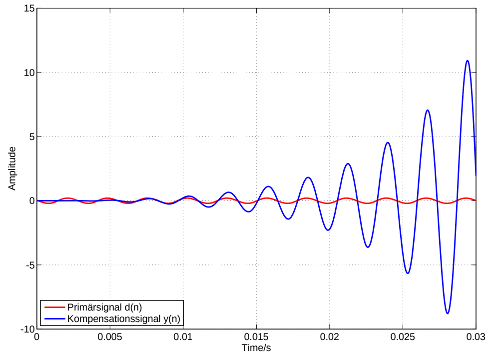

Abb. 23: Primärsignal $d(n)$ und exponentiell wachsendes Kompensationssignal $y(n)$

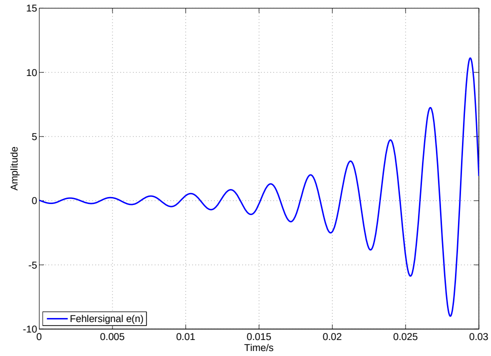

Abb. 24: Exponentiell wachsendes Fehlersignal $e(n)$ nach der Überlagerung von $d(n)$ mit $y(n)$

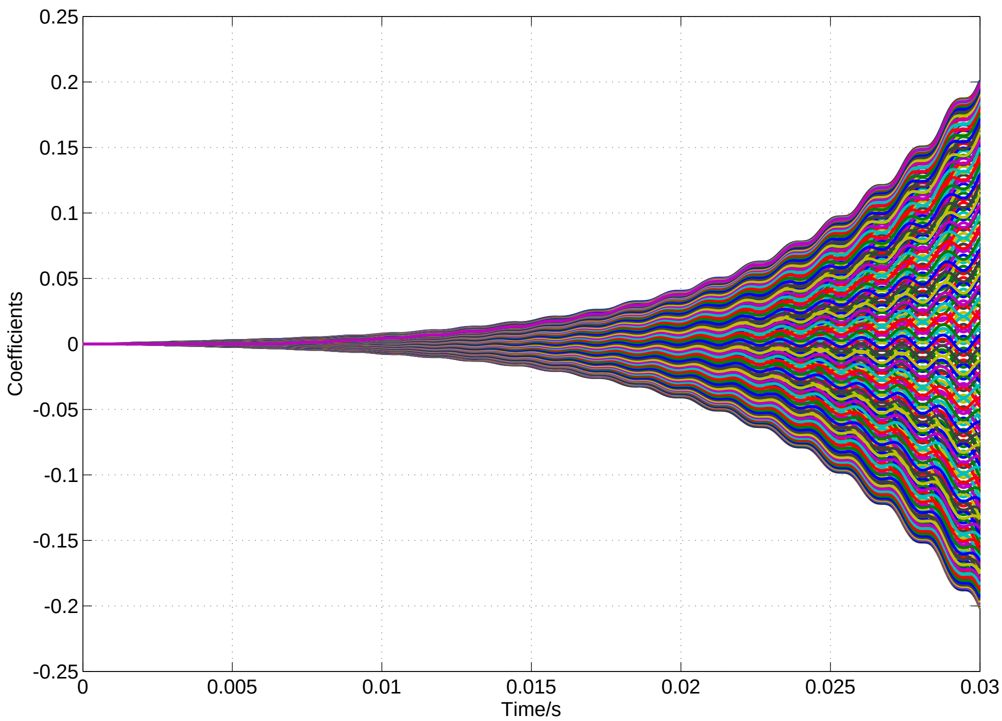

Abb. 25: Exponentiell wachsende Filterkoeffizienten $w_0(n) - w_{256}(n)$

Diese Instabilität wird durch die Verzögerung von $x(n)$ im Eingangspfad zum LMS-Algorithmus beseitigt. Die Adaption der Filterkoeffizienten beginnt dabei leicht verzögert, wodurch auch das Kompensationssignal $y(n)$ am Anfang etwas verzögert wird. Die Annäherung der Koeffizienten an das Optimum, die vollständige Anpassung des Kompensationssignal $y(n)$ an das Störsignal $s(n)$ und die dadurch erreichte Minimierung des Fehlersignals $e(n)$ erfolgt jedoch auch in diesem System innerhalb von 30 ms (vgl. Abb. 26 und 27 und 28).

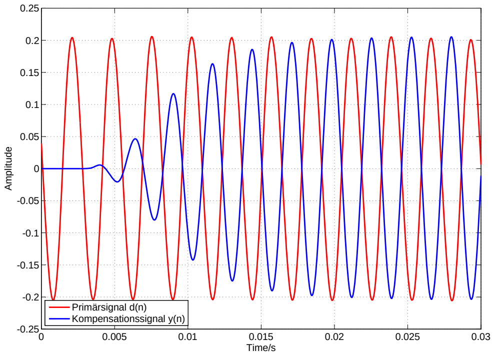

Abb. 26: Primärsignal $d(n)$ und Kompensationssignal $y(n)$ nach der Simulation mit verzögertem $x(n)$

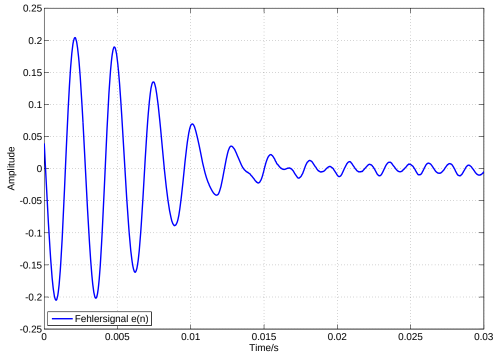

Abb. 27: Fehlersignal $e(n)$ nach der Überlagerung von $d(n)$ mit $y(n)$ und verzögertem $x(n)$

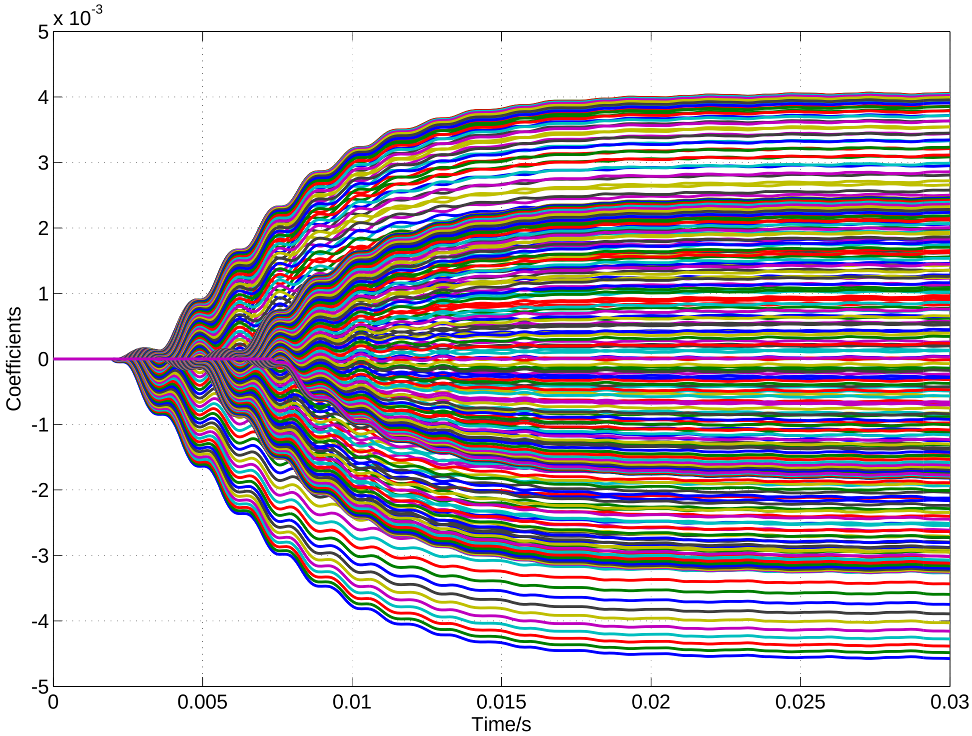

Abb. 28: Adaptierte Filterkoeffizienten $w_0(n) - w_{256}(n)$ nach der Simulation mit verzögertem $x(n)$

Folgende Erkenntnisse wurden durch die Modellierung und Simulation Systemaufbaus getroffen:

- Die Laufzeit des Referenzsignals $x(n)$ im Eingangspfad des adaptiven Filters hat keinen Einfluss auf die Funktion des Systems
- Die Gruppenlaufzeit des Sekundärpfades $S(z)$, bestehend aus Audio-Codec, Kopfhörer und Inohrmirkofon, führt zu einer unerwünschten Phasenverschiebung des Kompensationssignals $y(n)$, wodurch eine erfolgreiche Kompensation verhindert wird
- Im Frequenzbereich unterhalb von $4\ kHz$ kann der Frequenzgang des Sekundärpfades als konstant betrachtet werden, und es gilt $S(z) = z^{-64}$

- Durch die Verzögerung um die Sekundärpfad-Gruppenlaufzeit des Referenzsignals $x(n)$ im Eingangspfad des Adaptionsalgorithmus wird die unerwünschte Phasenverschiebung des Kompensationssignals $y(n)$ korrigiert, was zu einer erfolgreichen Kompensation führt
- Die Verzögerung des Referenzsignals $x(n)$ im Eingangspfad des Adaptionsalgorithmus muss bei der Implementierung des adaptiven Filters auf dem FPGA berücksichtigt werden

5 HW-Modellierung des adaptiven Filters für eine Xilinx-FPGA-Plattform

In diesem Kapitel wird der Aufbau des, auf dem Spartan-3 FPGA (vgl. Anhang F) implementierten, Systems zur adaptiven Filterung beschrieben. Zunächst wir ein Überblick zu Entwurfsmethodiken und Implementierungstechnologien gegeben bevor das Gesamtsystem bestehend aus den Komponenten

- *CODEC_INTERFACE*: Schnittstelle des FPGAs zum Audio-Codec
- *LMS_FIR*: adaptives sequentielles FIR-Filter mit LMS-Adaptionsalgorithmus

erläutert wird.

5.1 Entwurfmethodik und Implementierungstechnologien

Zur Realisierung des Codec-Interfaces und des adaptiven FIR-Filters auf dem FPGA werden diese als synthesefähige Register Transfer Level (RTL)-Modelle entworfen. Dazu müssen bestimmte Syntheserichtlinien eingehalten werden, die sich auf einfachster Ebene in drei Gruppen klassifizieren lassen:

- Kombinatorische Logik (inkl. Multiplexer, Tri-State-Treiber etc.)
- Latches (d.h. taktzustandsgesteuerte Speicher)
- Taktflankengesteuerte Flipflops, Zähler etc.

Charakteristisch für ein RTL-Modell, welches den VHDL-Syntesewerkzeugen zu Grunde liegt, ist die Trennung von Registern und kombinatorischen Logikstufen, die jeweils das Eingangssignal der nachfolgenden Registerstufe definieren (vgl. Abb. 29). Der Datenpfad wird somit als Pipeline aufgefasst. Kombinatorische und getaktete Prozesse sollen, einem reinen RTL-Entwurfsstil entsprechend, streng von einander getrennt werden.

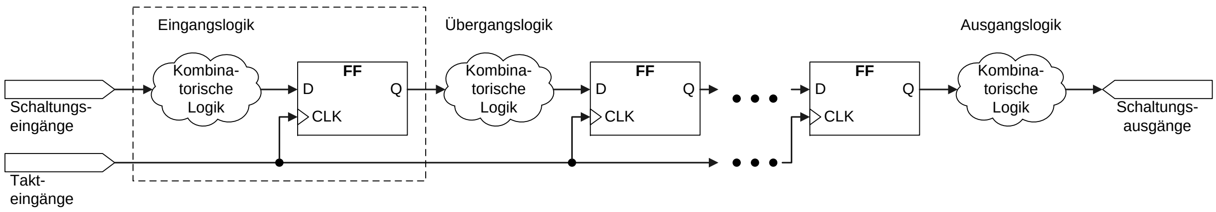

Abb. 29: Register Transfer Level-Modell [19]

In der Regel sind Synthesewerkzeuge jedoch in der Lage, kombinatorische Logik am Eingang von Registerstufen in einem getakteten Prozess zu indetifizieren und als gemeinsamen Prozess korrekt zu synthetisieren. Ein solcher VHDL-Entwurf erhöht die Übersichtlichkeit des Codes und wird beispielsweise beim Entwurf von gesteuerten Zählern verwendet (vgl. Anh. E.1 Code S. 102).

Kombinatorische Logik am Ausgang von Registerstufen muss hingegen immer als eigener Prozess oder als nebenläufige Anweisung modelliert werden [19].

Beide Komponenten, das Codec-Interface und das adaptive Filter, werden als Prozessorelement mit einem Datenpfad und einem Steuerpfad entworfen (vgl. Abb. 30). Dieses hat sich als Modell für die Strukturierung digitaler Systeme mit umfangreichen Datenmanipulationen bewährt [19].

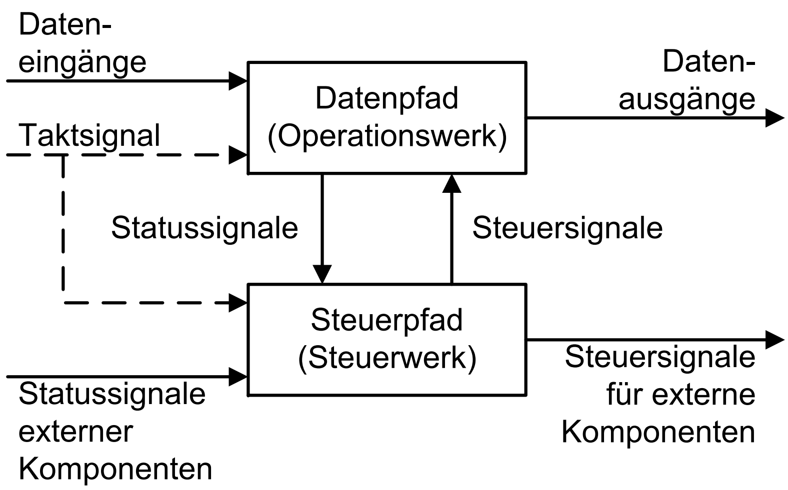

Abb. 30: Partitionierung eines digitalen Systems in Daten- und Steuerpfad. Die beiden Teilsysteme kommunizieren über Steuer- bzw. Statussignale. Zusätzlich wird das Verhalten des Steuerwerks von Statussignalen externer Komponenten beeinflusst und es werden Steuersignale für externe Komponenten erzeugt [19]

- Im Datenpfad befinden sich alle Arithmetikkomponenten zur Daten- bzw. Signalverarbeitung, Speicher und Multiplexer. Dies sind kombinatorische und getaktete Schaltungselemente, deren Funktion durch Steuersignale kontrolliert wird und die neben den Datenpfadausgangssignalen auch Statussignale erzeugen [19]. Im Entwurf des Codec-Interfaces und des adaptiven Filters sind dies (vgl. Abb. 38):
  - Schieberegister für serielle Kommunikation
  - Block-RAM on Chip Daten- und Koeffizientenspeicher
  - Adresszähler
  - MAC-Unit zur Bildung des Filterausgangs
  - Adaptionseinheit (Multiplizierer und Addierer) zur Berechnung der Koeffizienten
- Der Steuerpfad besteht in der Regel aus einem Zustandsautomaten, zu dessen Eingängen die Statussignale des Datenpfades gehören und der die Steuersignale für die Datenpfadkomponenten erzeugt [19]. Dieser wird im Codec-Interface durch zwei Prozesse dargestellt:
  - Prozess zur Kontrolle der Eingangsregister
  - Prozess zur Kontrolle der Ausgangsregister

  Beim adaptiven Filter bildet ein Zustandsautomat
  - Finite state machine (FSM)

  den Steuerpfad.

Sowohl im Codec-Interface als auch im adaptiven Filter werden die Funktionselemente auf nebenläufige Prozesse abgebildet. Die Organisation dieser Parallelität führt zum Entwurf von Pipelinestrukturen, deren Steuerung die FSM übernimmt.

Die Ziel-Hardware des Systems zur adaptiven Filterung ist ein Sparten-3 FPGA (vgl. Anh. F). Die im Folgenden vorgestellte Modellierung ist jedoch auf andere Xilinx-FPGAs portierbar. Die Xilinx-FPGA spezifischen Module sind:

- On Chip Block-RAMs
- Embedded 18x18-Bit Multiplizierer

Die Block-RAMs werden für diese Implementierung mit dem Xilinx CORE Generator erzeugt. Sie verhalten sich taktsynchron und werden über ein Adressierungs- und ein Schreibssignal gesteuert. Zur Speicherung des am Eingang *DI* anliegenden Datums wird parallel zur Speicheradresse das Schreibsignal *WE* gesetzt. Mit der nächsten steigenden Taktflanke erfolgt die Speicherung.

Beim Lesen einer Speicherzelle wird die gewünschte Adresse angelegt, wonach mit der nächsten Taktflanke das Datum am Ausgang *DO* zur Verfügung steht. Diese durch die Taktsynchronisation entstehenden Latenzen, müssen beim Entwurf des Steuerautomaten berücksichtigt werden.

Sämtliche Vektorlängen im adaptiven Filter ergeben sich aus der Verwendung der embedded 18x18-Bit Multiplizierer. Diese sind direkt mit den Block-RAMs verbuden und ermöglichen dadurch schnelle Multiplikationen aufgrund von kurzen Signallaufzeiten [25]. Die Integerarithmetik wird dabei mit dem sogenannten Q-Format und Guard-Bits zum Schutz vor Überläufen der Zweierkomplement-Darstellung realisiert.

Für Fractional-Zahlen mit der Wortlänge $B + 1$ und einem Binärpunkt (•) im Q-Format gilt ein Bitstring mit der Form:

$$x_{Bin} = b_B \bullet b_{B-1} \ldots b_1 \; b_0 \tag{52}$$

mit $b \in (0, 1)$ und einem Vorzeichenbit $b_B = 1$ für $x_{Dez} < 0$.

Dieses QB-Format repräsentiert Zahlen im Intervall $-1 \leq x_{Dez} \leq 1$ mit:

$$x_{Dez} = -b_B + \sum_{i=1}^{B} b_{B-i} \cdot 2^{-i} \tag{53}$$

Diese Fractional-Darstellung im Q-Format hat sich für die Eingangs- und Ausgangssignale, Koeffizienten sowie Rechengrößen in digitalen Systemen bewährt, da Multiplikationsergebnisse im Bereich $-1 \leq x_{Dez} \leq 1$ begrenzt bleiben und betragsmäßig in Richtung des LSBs $b_0 = 2^{-B}$ streben. Die Q-Format Betrachtungsweise hat jedoch nur der Anwender. Der VHDL-Simulator und das Synthesewerkzeug betrachten die Vektoren als Bitstring für eine standard Integerdarstellung. Der Anwender muss daher konsistent skalierte Vektorbreiten sicherstellen [19].

Sowohl bei der Addition als auch bei der Multiplikation von Zahlen im Q-Format enthält die Summe bzw. das Produkt sogenannte Guard-Bits als Überlaufschutz der Zweierkomplement-Arithmetik.

Eine Addition zweier QB-Größen A und C, die jeweils den positiven Maximalwert $1 - 2^{-B}$ annehmen können, liefert maximal den Wert $2 - 2^{-B+1}$, der aufgrund des Integeranteils nicht im QB-Format allein darstellbar ist. Der Ergebnisvektor SUM ist daher um eine Bitstelle, dem Guard-Bit, breiter zu wählen, die den Integeranteil aufnimmt. Da auf beiden Seiten einer Zuweisung die Signale gleiche Vektorbreiten aufweisen müssen, wird bei den Summanden A und C eine vorzeichenrichtige Erweiterung auf die Breite des Summenvektors durchgeführt.

Beispiel einer Additon zweier Summanden im Q17-Format:

$$SUM[sign.16:0] = SUM[sign, guard.15:0] \tag{54}$$

$$= A[sign, sign.15:0] + C[sign, sign.15:0] \tag{55}$$

Die Anzahl der einzurichtenden Guard-Bits bei n Summanden, die alle im maximalen Wertebereich liegen, ergibt sich aus $\lceil 3.32 log(n)$ [19].

Bei einer Multiplikation zweier Multiplikanden QB_1 und QB_2 entsteht ein QB_3-Ergebnis, in dem $B_3 = B_2 + B_1$ Bits rechts vom impliziten Binärpunkt stehen. Der gesamte Vektor enthält

$B_3 + 2$ Bits, da eine Q-Format-Multiplikation zwei Vorzeichen-Bits liefert, von denen das linke ein „echtes“ Vorzeichen ist und das rechte Bit vor dem Binärpunkt als Guard-Bit genutzt wird.

Beispiel einer Multiplikation mit zwei Q17-Multiplikanden:

$$MUL[sign, sign.33:0] = MUL[sign, guard.33:0] \quad (56)$$
$$= SAMP[sign.16:0] * COEF[sign.16:0] \quad (57)$$

Ein Überlaufschutz der Zweierkomplement-Arithmetik lässt sich für interne Signale allein mit Guard-Bits korrekt realisieren. Bei einem adaptiven Filter der Ordnung N, dessen Koeffizienten variabel sind, muss für jedes der $N+1$ Produkte, zur Bildung des Filterausgangssignals *Y* (vgl. Kap. 3.1 Gl. 2), der Maximalwert 1 angenommen werden. Der Summenvektor *Y* wird daher um $\lceil ld(N+1) - 1$ erweitert, da die Breite eines Produktvektors bereits ein Guard-Bit enthält [19].

5.2 Systemübersicht

Das *CODEC_INTERFACE* sorgt für den synchronen seriellen Datenaustausch zwischen FPGA und Audio-Codec. Es stellt die empfangenen Audiodaten dem adaptiven Filter in einem Register zur Verarbeitung bereit. Diese setzen sich aus einem Abtastwert des Referenzsignals *XN*[2] und einem Abtastwert des Fehlersignals *EN* zusammen. Das Senden und Empfangen der seriellen Daten zwischen FPGA und Codec findet parallel statt und erfolgt über die Leitungen *SDATA_IN* und *SDATA_OUT* (vgl. Abb. 31).

Die Schnittstellen des adaptiven Filters *LMS_FIR* sind kompatibel zu denen des Codec-Interfaces, wobei das Referenzsignal *XN* über den linken und das Fehlersignal *EN* über den rechten Audiokanal vom Codec zum FPGA übertragen wird. Das Kompensationssignal *YN* wird über den rechten Audiokanal vom FPGA zum Codec übertragen, da in dieser Anwendung das Störsignal im rechten Ohr kompensiert werden soll.

Das Codec-Interface übernimmt die Rolle eines AC'97[3] Controllers (vgl. Anhang G.1), der das durch die Signale *BIT_CLK*, *SYNC*, *N_RESET_OUT*, *SDATA_IN* und *SDATA_OUT* gekennzeichnete AC97 Inteface nutzt, um den Audio-Codec anzusteuern. Dazu ist das Spartan-3 Entwicklungsboard über ein P160-Modul mit einer Audio/Video-Plattform verbunden.

Codec-Interface und adaptives FIR-Filter sind auf dem FPGA zu einem System zusammengefasst. Das Codec-Inteface wird über das Taktsignal *BIT_CLK* getaktet und sendet bzw. empfängt Datenframes (vgl. Abb. 33) über die seriellen Datenleitungen, deren Länge durch die Periode des *SYNC*-Signals festgelegt ist [17] [12]. Die in einem solchen Frame enthaltenen Audiodaten werden in ein Register geschoben und darüber einem Modul, in diesem Fall dem adaptiven FIR-Filter, zur weiteren Verarbeitung bereitgestellt. Das Ausgangsdatum YN des Filters wird vom Codec-Interface parallel in ein Register eingelesen und seriell zum Codec gesendet.

Das adaptive FIR-Filter wird über das Taktsignal *CLK* getaktet und erhält mit jedem Ready-Impuls *RD* jeweils ein neues Einganssample des Referenzsignals *XN* und des Fehlersignals *EN*. Aus diesen Daten erzeugt das Filter das Kompensationssignal *YN*. Nach der Erzeugung eines neuen Kompensationssignalwertes YN erzeugt das Filter einen Impuls *FILT_RDY*, der dem Codec-Interface ein neues Ausgangs-Sample signalisiert.

[2] In diesem Kapitel werden die Signale mit den in der Impelementierung verwendeten Namen bezeichnet. $XN \equiv x(n)$, $EN \equiv e(n)$, $YN \equiv y(n)$

[3] Audio Codec '97: von Intel Architecture Labs im Jahr 1997 entwickelter Audio-Standard

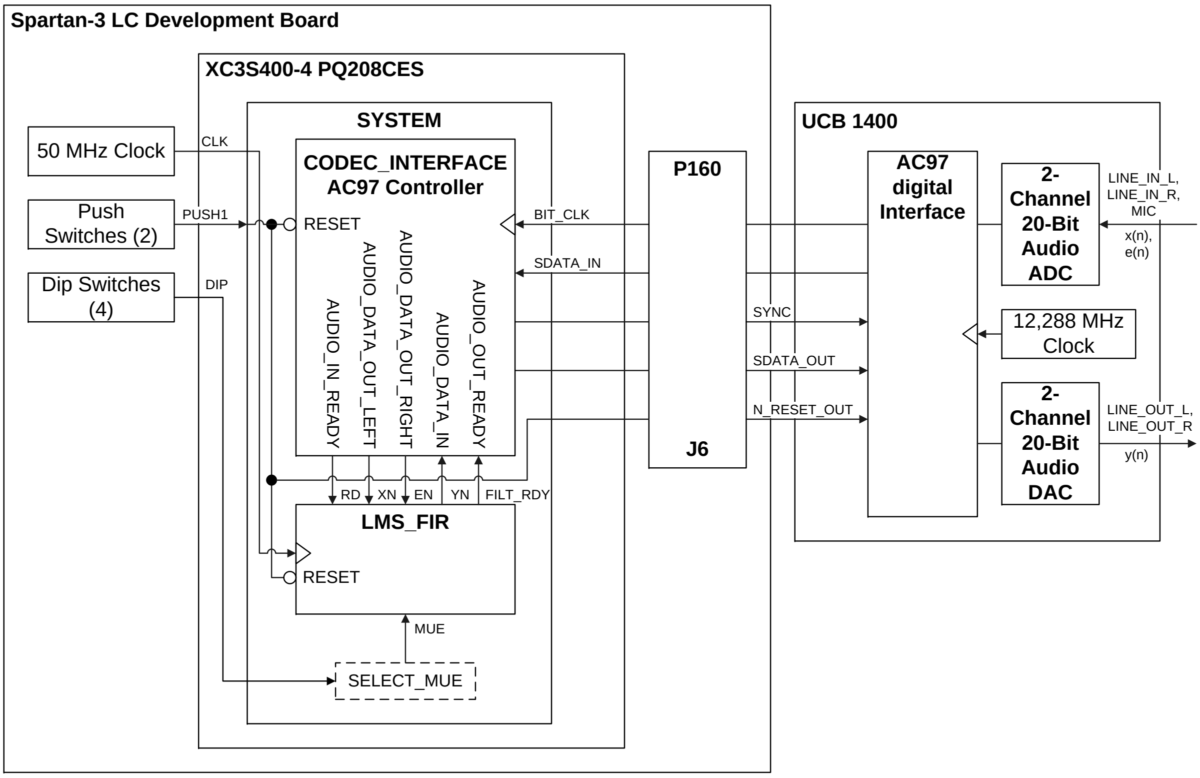

Abb. 31: FPGA Systemaufbau bestehend aus Codec-Interface und adaptivem FIR-Filter. $CLK = 50\ MHz$, $BIT\_CLK = 12,288\ MHz$, *Abtastfrequenz* $f_s = 48\ kHz$

Das durch VHDL beschriebene HW-Modell des Systems ist in mehrere Komponenten (`component`) unterteilt, welche hierarchisch aufgebaut sind und durch `entity`/`architecture`-Paare [19] beschrieben werden. In der `architecture` der Top-`entity` *System* werden die Komponenten *CODEC_INTERFACE* und *LMS_FIR* zusammengefasst. Während das Codec-Interface in keine weiteren Komponenten aufgeteilt ist, enthält die Haupt-`entity` des adaptiven Filters drei Unter-Komponenten (vgl. Abb. 32).

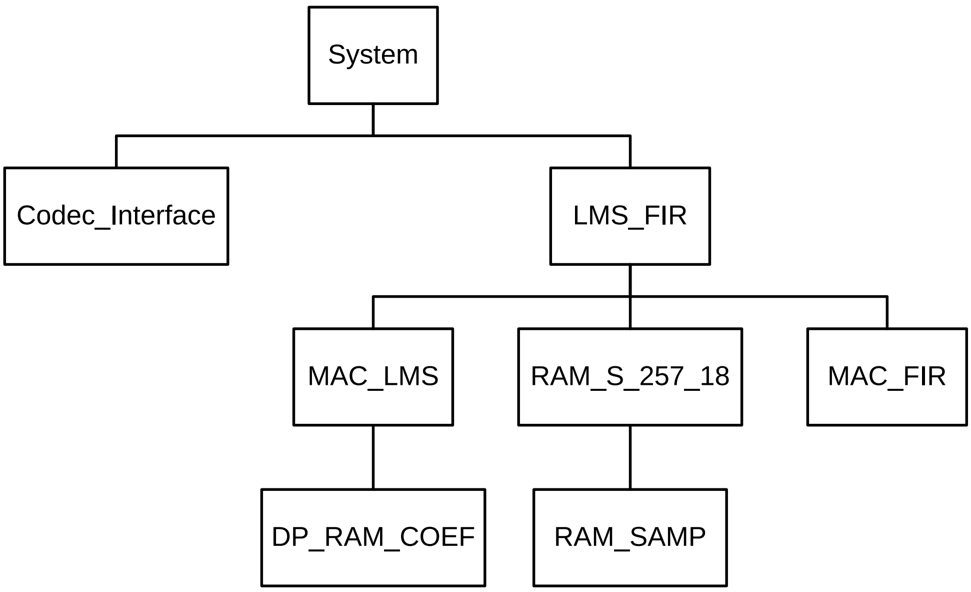

Abb. 32: Hierarchischer Baum der Komponenten des gesamten VHDL-System-Modells

Die Komponente *MAC_LMS* hat die Aufgabe die Filterkoeffizienten zu adaptieren und enthält daher den Speicher für die Koeffizienten *DP_RAM_COEF*. Der Speicher für die eingehenden

Abtastwerte des Referenzsignals *RAM_SAMP* wird in der Komponente *RAM_S_257_18* instanziiert. Dadurch stehen die Samples sowohl dem Adaptionsalgorithmus als auch der Komponente *MAC_FIR* zur Verfügung. Diese besteht aus einer Multiplizierer-Akkumulatoreinheit, in der die Koeffizienten mit den Abtastwerten multipliziert und die Ergebnisse akkumuliert werden.

In den folgenden Abschnitten wird zunächst die Struktur und Funktionsweise des Codec-Interfaces beschrieben. Anschließend wird das adaptive Filter mit allen zugehörigen Komponenten erläutert.

5.3 Das AC'97 Codec-Interface

Der auf dem AV-Board vorhandene Audio-Codec (vgl. Anhang G) wandelt analoge Audiosignale in digitale Signale um und umgekehrt. Dabei wird ein analoges Signal mit einer Frequenz von 48 kHz abgetastet. Die Abtastwerte werden als 20-Bit Binärzahl dargestellt und sollen seriell zum FPGA und wieder zurück übertragen werden. Dies geschieht innerhalb eines Datenframes, der eine Länge von 256 Bits hat und synchron zum *SYNC*-Signal gesendet wird (vgl. Abb. 33). Das Senden und Empfangen der Daten über die seriellen Leitungen *SDATA_OUT* und *SDATA_IN* geschieht parallel, wobei auf der steigenden *BIT_CLK*-Flanke gesendet, und auf der fallenden Flanke empfangen wird (vgl. Anhang G.1). Das Codec-Interface hat die Aufgabe den seriellen Datenaustausch zwischen Audio-Codec und FPGA, und die Verarbeitung der digitalen Audiodaten durch das sequentielle adaptive FIR-Filter zu synchronisieren.

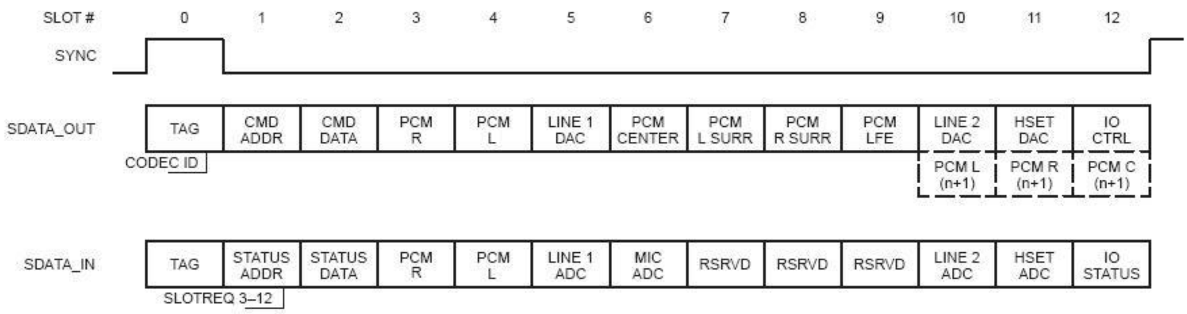

Abb. 33: Bidirektionaler Audio-Frame [17]

Der Aufbau des Interfaces folgt dem Konzept des Synchron-Systems [23]. Dieses ist aufgeteilt in einen Datenpfad und einen Steuerpfad. Der Datenpfad besteht aus den ladbaren Schieberegistern:

- *TAG_REG_IN*: Register für einen eingehenden TAG
- *ADR_REG_IN*: Register für eine eingehende Konfigurationsregister-Adresse
- *DATA_REG_IN*: Register für ein eingehendes Status-Datum eines Konfigurationsregisters
- *AUDIO_REG_IN*: Register für eingehende Audiodaten
- *TAG_REG_OUT*: Register für einen ausgehenden TAG
- *ADR_REG_OUT*: Register für eine ausgehende Konfigurationsregister-Adresse
- *DATA_REG_OUT*: Register für ein ausgehendes Konfigurations-Datum
- *AUDIO_REG_OUT*: Register für ausgehende Audiodaten

Die Register werden in den Prozessen *SHIFT_DATA_IN*, *LOAD_OUT_REGISTERS* und *SHIFT_DATA_OUT* zusammengefasst. Für jeden relevanten Datenslot eines Frames existiert ein seperates Register, so dass die Daten eines Slots getrennt von den Daten eines anderen verarbeitet werden können.

Der Steurpfad setzt sich aus folgenden Prozessen zusammen:

- *CONTROL_IN_REGISTERS*: Kontrolliert die Freigaben der Eingangsregister des Datenüfades
- *CONTROL_OUT_REGISTERS*: Kontrolliert die Freigaben der Ausgangsregister des Datenpfades

Die Freigaben für die Register werden aus dem *SYNC*-Signal, den Slot-Zählerständen *SLOT_COUNT* und *SLOT_BIT_COUNT* sowie den Daten des TAG-Slots dekodiert. Auf diese Weise wird erreicht, dass die Daten eines bestimmten Slots immer in das zugehörige Register geschoben bzw. aus dem Register rausgeschoben werden. Synchronisiert wird das System über das *BIT_CLK* Taktsignal (vgl. Abb. 34).

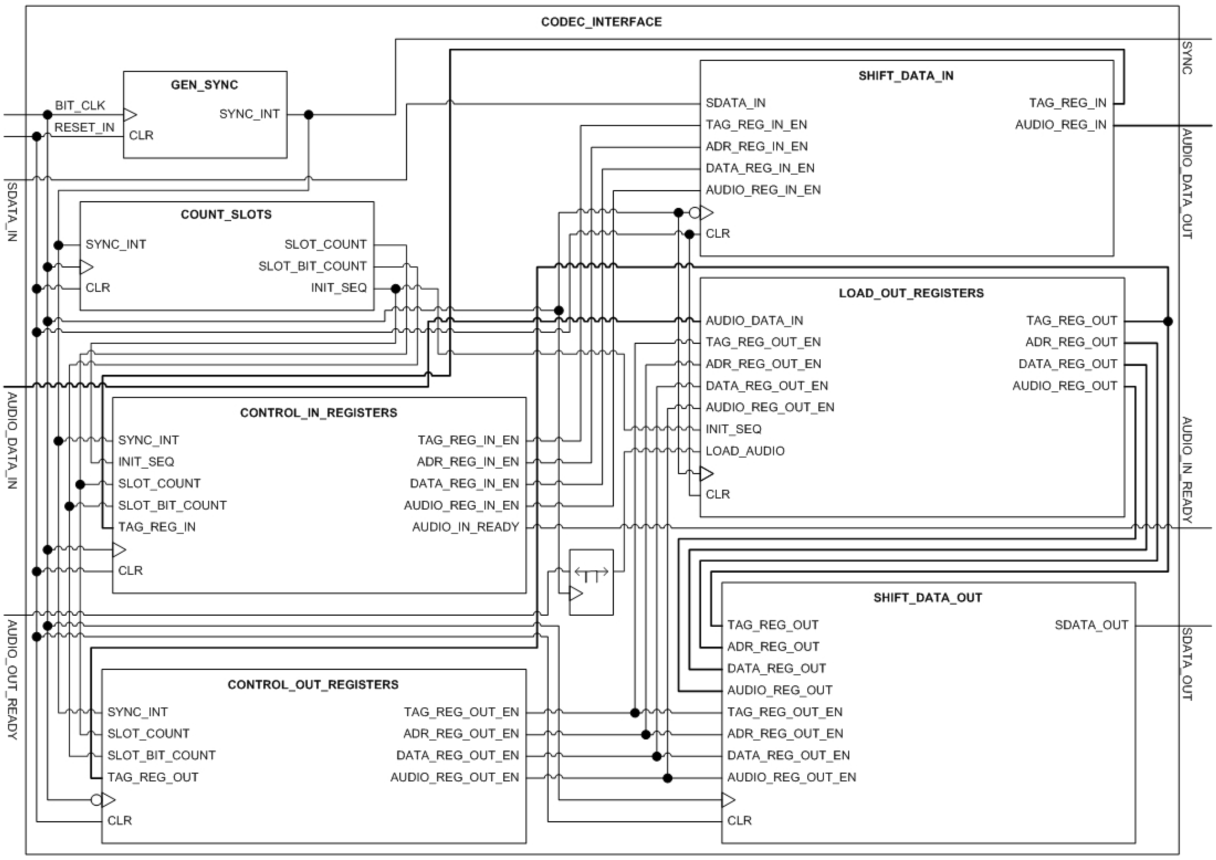

Abb. 34: Komponenten des Codec Interface. FPGA-Schnittstelle zum Audio-Codec. Datenpfad: SHIFT_DATA_IN, LOAD_OUT_REGISTERS und SHIFT_DATA_OUT. Steuerpfad: CONTROL_IN_REGISTERS und CONTROL_OUT_REGISTERS.

Das Codec-Interface erhält vom Audio-Codec das *BIT_CLK* Taktsignal mit einer Frequenz von $12,288\ MHz$. Dieses Taktsignal synchronisiert den Datenaustausch zwischen FPGA und Audio-Codec. Auf der steigenden Taktflanke werden Daten gesendet und auf der fallenden Flanke empfangen (vgl. Anhang G.1). Die Daten werden in Form von 256-Bit Frames über die seriellen Datenleitung *SDATA_IN* und *SDATA_OUT* ausgetauscht. Der Beginn eines neuen Datenframes wird durch die steigende Flanke des *SYNC*-Signals festgelegt. Dieses Signal muss vom Codec-Interface erzeugt werden und hat eine Periodenlänge, die genau der Länge eines Datenframes entspricht.

Das Interface empfängt Daten vom Codec über die serielle *SDATA_IN*-Leitung. Diese Daten werden in, den unterschiedlichen Slots zugeordnete, Eingangs-Register geschoben. Dies geschieht auf der fallenden *BIT_CLK*-Flanke im *SHIFT_DATA_IN*-Prozess. Parallel dazu werden auf der steigenden *BIT_CLK*-Flanke Daten aus den Ausgangsschieberegistern auf die serielle

SDATA_OUT-Leitung geschoben und zum Codec gesendet. Welches Ausgangsregister mit der *SDATA_OUT*-Leitung verbunden ist, wird im *SHIFT_DATA_OUT*-Prozess festgelegt. Geladen werden die Ausgangsregister im *LOAD_OUT_REGISTERS*-Prozess, wo das *TAG_REG_OUT*-Register, das *ADR_REG_OUT*-Register und das *DATA_REG_OUT*-Register während der Anlaufphase mit Werten geladen werden, um den Audio-Codec zu konfigurieren. Danach wird im Normalbetrieb nur noch das *AUDIO_REG_OUT*-Register mit neuen Ausgangswerten geladen. Bei allen anderen Ausgangsregistern wird der serielle Ausgang in einer Rückkopplung auf den seriellen Eingang zurückgeführt, so dass nach einem kompletten Schiebezyklus wieder die Ausgangswerte in den Registern stehen, ohne dass diese neu geladen werden müssen (vgl. Abb. 97).

Die Prozesse *GEN_SNYC* und *COUNT_SLOTS* sind für das Timing des Interfaces verantwortlich. Eine Periode des *SYNC*-Signals markiert dabei einen Frame. Der *SLOT_COUNT*-Zähler zählt die Slots innerhalb eines Frames und der *SLOT_BIT_COUNT*-Zähler zählt die Bits innerhalb eines Slots (vgl. Abb. 35). Durch diese drei Signale lässt sich die exakte Position innerhalb eines Datenframes bestimmen. Das *INIT_SEQ*-Signal dient der Bestimmung der Anlaufphase. Es zählt die ersten fünf Frames nach einem Systemstart oder einem Reset, die zur Konfiguration des Audio-Codecs benötigt werden (vgl. Abb. 36).

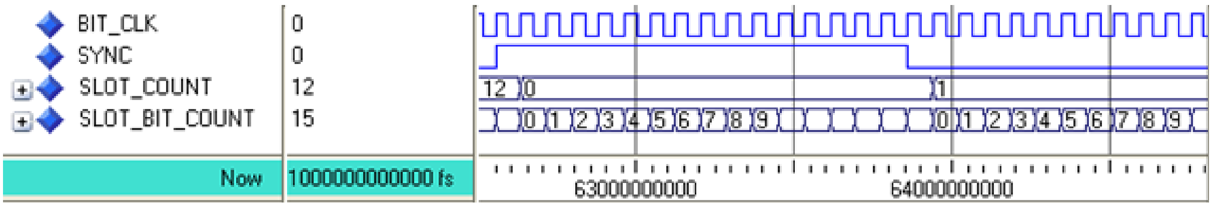

Abb. 35: VHDL-Simulation des SLOT_COUNT-Zählers und des SLOT_BIT_COUNT-Zählers. Beginn eines Datenframes mit $SYNC = 1$. $f_{BIT\_CLK} = 12,288\ MHz$

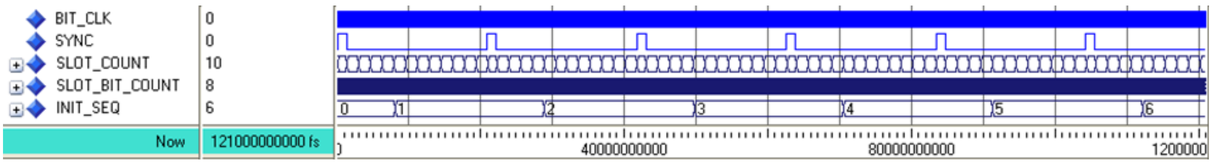

Abb. 36: VHDL-Simualtion des INIT_SEQ-Zählers. Fünf Datenframes zur Konfiguration des Audio-Codecs. Bei $INIT\_SEQ = 6$ ist die Anlaufphase beendet.

Anhand dieser Timing-Signale werden die Schiebe- und Ladervorgänge der unterschiedlichen Ein- und Ausgangsregister gesteuert. Die Prozesse *CONTROL_IN_REGISTERS* und *CONTROL_OUT_REGISTERS* lesen die Zählerstände und setzen das Enable-Signal des zu dem jeweiligen Slot gehörenden Ein- bzw. Ausgangsregisters (vgl. Abb. 37). Diese Signale aktivieren die Schiebvorgänge im *SHIFT_DATA_IN*- und *LOAD_OUT_REGISTERS*-Prozess, so dass die Register zum richtigen Zeitpunkt Daten empfangen bzw. senden. Das *INIT_SEQ*-Signal sorgt dafür, dass die Ausgangsregister während der Anlaufphase im *LOAD_OUT_REGISTERS*-Prozess zum richitgen Zeitpunkt mit den Codec-Konfigurationsdaten (vgl. Anh. G.2) geladen werden. Der *CONTROL_IN_REGISTERS*-Prozess erzeugt nach dem Empfang von Audiodaten zusätzlich einen *AUDIO_IN_READY*-Impuls, der als Ready-Impuls für das adaptive FIR-Filter dient. Durch diesen Impuls wird der Filterungszyklus des neuen Audiosampels angestoßen.

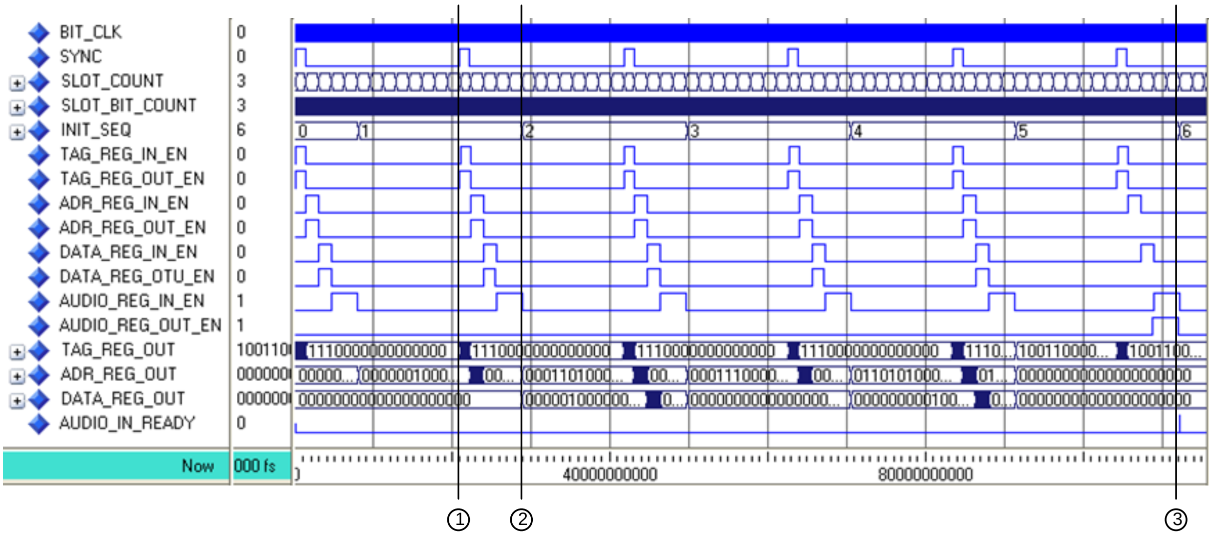

Abb. 37: VHDL-Simulation des Codec-Interfaces. Aktivierung der Ein- und Ausgangsregister (Marker 1); Paralleles Laden der Ausgangsregister ADR_REG_OUT und DATA_REG_OUT (Marker 2); Erzeugung des Ready-Impulses AUDIO_IN_READY (Marker 3)

Die Prozesse aus denen sich das Codec-Interface zusammensetzt werden in dem Projektbericht [12] und im Anhang H ausführlich beschrieben.

5.4 Das adaptive sequentielle FIR-Filter

Das adaptive Filter besitzt eine sequentielle FIR-Struktur und wird anhand des LMS-Kriteriums adaptiert (vgl. Kap. 3). Es hat die Aufgabe durch die Filterung des Referenzsignals *XN* ein Kompensationssignal *YN* zu erzeugen, welches das bei einem Empfänger ankommende Störsignal $s(n)$ kompensiert (vgl. Kap. 2.4). Die Koeffizienten werden dazu durch den LMS-Adaptionsalgorithmus mit dem Produkt aus Referenzsignal *XN*, Fehlersignal *EN* und Schrittweitenfaktor *MUE* fortlaufend angepasst (vgl. Kap. 3.3.1 Gl 22).

Folgende Eigenschaften werden in der Implementierung des adaptiven Filters berücksichtigt:

- Sequentielle Struktur mit einer MAC-Einheit zur Bildung des Filterausgangssignals *YN*, um bei geringem Ressourcenverbrauch eine hohe Filterordnung $N = 256$ zu realisieren, und die Nutzung der embedded 18x18-Bit Multiplizierer des Spartan-3 FPGAs für sämtliche Multiplikationen zu ermöglichen.
- Dual-Port-Block-RAM zur Speicherung der Koeffizienten, da das Zurückschreiben eines adaptierten Koeffizienten einen zusätzlichen Takt erfordert. Die Adressierung zum Lesen und Schreiben läuft dabei versetzt ab, da die Schreib-Adresse noch konstant anliegen muss, während die Leseadresse bereits inkrementiert wird.
- Das Referenzsignal *XN* wird, wie in Kapitel 4.2 dargestellt, im Eingangspfad der Adaptionseinheit *MAC_LMS* um die Gruppenlaufzeit des Sekundärpfades verzögert, um die unerwünschte Phasenverschiebung des Kompensationssignals *YN* zu korrigieren.

Das Filter wird mit einer Frequenz $f_{CLK} = 50\ MHz$ getaktet, sodass bei einer Audioabtastrate $f_s = 48\ kHz$ dem adaptiven Filter 878 *CLK*-Taktzyklen für die Aktualisierung des Filterausgangs *YN* und die Koeffizientenadaption zur Verfügung stehen.

$$f_{BIT\_CLK} = 12{,}288\ MHz \qquad T_{BIT\_CLK} = 81{,}38\ ns \tag{58}$$

$$f_{CLK} = 50\ MHz \qquad T_{CLK} = 20\ ns \tag{59}$$

$$f_S = 48\ kHz \qquad T_S = 20{,}83\ \mu s \tag{60}$$

Aus den Periodenlängen T_{BIT_CLK}, T_{CLK} und T_S lässt sich berechnen, wie viele Takte $N_{CLK/S}$

in einer Abtastperiode zur Verfügung stehen (Gl. 61) und die Anzahl N_{CLK/BIT_CLK} der Filter-Takte pro Interface-Taktperiode (Gl. 62).

$$N_{CLK/S} = \frac{T_S}{T_{CLK}} = 1041,67 \tag{61}$$

$$N_{CLK/BIT\_CLK} = \frac{T_{BIT\_CLK}}{T_{CLK}} = 4,069 \tag{62}$$

Zum Empfangen und Senden zweier Audio-Samples werden 40 *BIT_CLK*-Perioden benötigt. Daher berechnet sich die Zahl $N_{CLK/Filt}$ der für einen Filterungszyklus zur Verfügung stehenden Takte durch Gleichung 63.

$$N_{CLK/Filt} = N_{CLK/S} - 40N_{CLK/BIT\_CLK} = 878,906 \tag{63}$$

Da die Filterung des Referenzsignals *XN* und die Adaption der Koeffizienten als parallele Prozesse implementiert sind, steht die Anzahl $N_{CLK/Filt} = 878$ an Takten für beide Vorgänge zu Verfügung. Die Anzahl M der für einen sequentiellen Filter- bzw. Adaptionszyklus benötigten Takte beträgt 517 (vgl. Kap. 6.2.2 Gl. 69), wodurch sich das Filter mit einer einzigen Multiplizier-Akkumulatoreinheit (MAC-Unit) realisieren lässt.

Bei der parallelen Implementierung eines FIR-Filters in Direktform (vgl. Kap. 3.1 Abb. 10) mit der Ordnung N sinkt die Anzahl der für die Filterung benötigten Takte, es entsteht jedoch ein wesentlich größerer Ressorucenbedarf von maximal N Multiplizierern und N Addierern. Dieser lässt sich zwar durch eine Linear-Phasen-Struktur [19], in der die Symmetrie der Koeffizienten ausgenutzt wird, auf $N/2 + 1$ reduzieren, verhindert jedoch bei einem Filter der Ordnung $N = 256$ die aussschließliche Nutzung der 18x18-Bit embedded Multiplizierer, da auf der Ziel-Hardware nur 16 dieser Multiplzierer zur Verfügung stehen (vgl. Anh. F).

Zusätzlich zum erhöhten Ressourcenbedarf nimmt die Signallaufzeit von den Registerstufen $x(n-k)$ bis zum Ausgang $y(n)$ (vgl. Kap. 3.1 Abb. 10) durch die mit der Filterordnung N wachsende Addiererkette zu, was zur Folge hat, dass die maximale Taktfrequenz f_{CLK} sinkt.

Außerdem können bei sequentieller Arbeitsweise des Filters FPGA-interne RAMs zur Speicherung der Abtastwerte und Koeffizienten verwendet werden, wodurch gegenüber einer parallelen Lösung mit einer Registerkette, D-Flip-Flops und Verdrahtungsressourcen eingespart werden [19].

In den folgenden Abschnitten wird das als Prozessorelement mit einem Daten- und einem Steuerpfad [19][23] implementierte Filter mit der Ordnung $N = 256$ beschrieben (vgl. Abb. 38).

Das adaptive Filter setzt sich insgesamt aus sechs `entities` zusammen (vgl. Abb. 39). Dabei enthält die Haupt-`entity` *LMS_FIR* den Mealy-Automaten *FSM_LMS_FIR*, der die Komponenten der Datenpfad-`entities` *MAC_LMS*, *RAM_S_257_18* und *MAC_FIR* steuert.

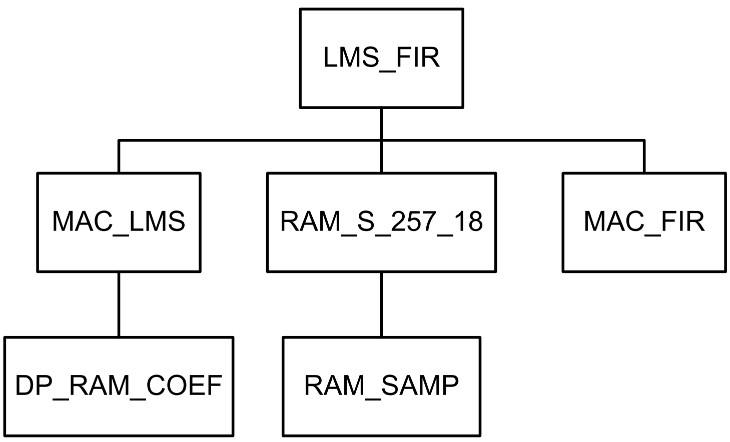

Abb. 39: `entity`-Baum des VHDL-Filter-Modells

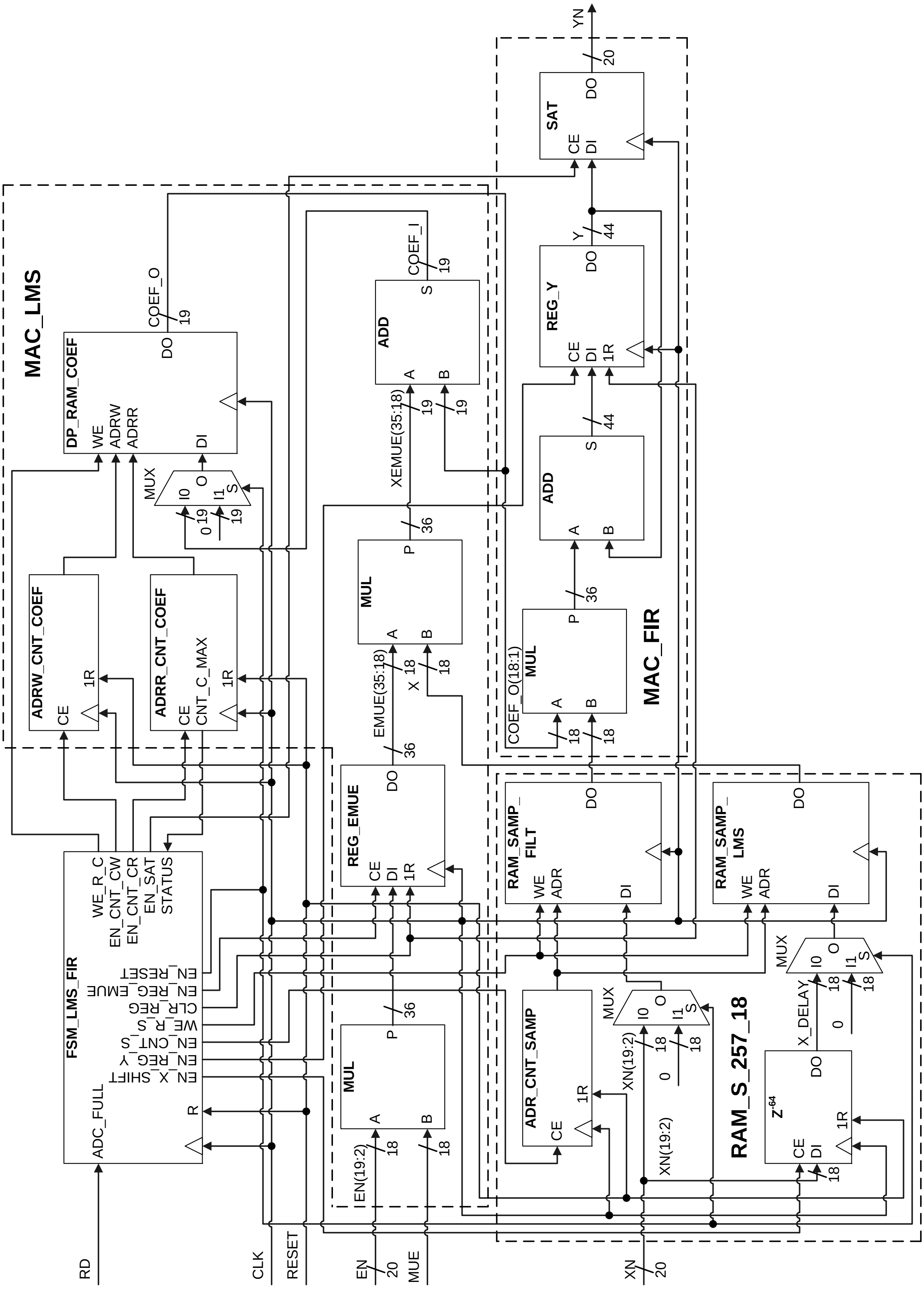

Abb. 38: Sequentielles adaptives FIR-Filter LMS_FIR mit den Datenpfadkomponenten MAC-Einheit MAC_FIR, LMS-Adaptionseinheit MAC_LMS und Abtastwerte-RAM RAM_S_257_18 sowie dem Steuerpfad, der aus dem Zustandsautomaten FSM_LMS_FIR und den zugehörigen Steuersignalen besteht. Taktsignal CLK mit $f_{CLK} = 50\ MHz$; $N = 256$; (vgl. Code S. 105)

- In der `entity` *MAC_FIR* werden in jedem Abtastzyklus $L = N + 1$ Produkte $COEF * SAMP\_FILT$ aus einem Abtastwert/Koeffizient-Paar gebildet und deren Ergebnisse akkumuliert (vgl. Kap. 3.1 Gl. 2). Durch das Sättigungsmodul *SAT* wird das Akkumulator-Ergebnis *Y* überlauffrei auf den 20-Bit Vektor *YN* begrenzt, der an das Codec-Interface übergeben wird. Mit der Sättigung wird auch das Vorzeichen des Kompensationssignalwertes *YN* umgekehrt, um die zur Kompensation benötigte Phasenverschiebung des Signals um 180° zu erreichen (vgl. Kap. 3.1 Abb. 10).
- Die `entity` *RAM_S_257_18* enthält zwei 257x18-Bit RAM-Module *RAM_SAMP_FILT* und *RAM_SAMP_LMS* für die Abtastwerte des Referenzsignals *XN*. Diese werden im 18-Bit Format gespeichert, da für die Multiplikationen sowohl bei der Filterung als auch bei der Adaption embedded 18x18-Bit Multiplizierer des FPGAs genutzt werden. Der Speicher *RAM_SAMP_FILT* stellt die Abtastwerte $SAMP\_FILT$ für die Filterung bereit, während der Speicher *RAM_SAMP_LMS* die verzögerten Abtastwerte $SAMP\_LMS$(vgl. Kapitel 4.2 Abb. 18) für die Adaption der Koeffizienten enthält. Beide Speicher sind als Ringpuffer konzipiert und werden über den Zähler *ADR_CNT_SAMP* adressiert.
- Die Adaption der Filterkoeffizienten $COEF$ findet in der `entity` *MAC_LMS* parallel zur Filterung statt. Dabei wird pro Abtastzyklus einmal das Produkt *EMUE* aus Fehlersignal *EN* und Schrittweitenfaktor *MUE* gebildet, welches als Faktor in die L Multiplikationen mit den Abtastwerten $SAMP\_LMS$ des Referenzsignals XN eingeht (vgl. Kap. 3.3.1 Gl. 22). Diese Abtastwerte werden um die in Kapitel 4.1 ermittelte Gruppenlaufzeit des Sekundärpfades verzögert, um eine Phasenkorrektur des Referenzsignals *YN* zu erzeugen (vgl. Kap. 4.2). Jeder Koeffizient wird durch die Addition mit dem zugehörigen Produkt $XEMUE$ aus Schrittweite *MUE*, Fehlersignal *EN* und verzögertem Referenzsignal *XN* in jeder Abtastperiode aktualisiert. Dieses Produkt *XEMUE* wird parallel zum Produkt aus einem Koeffizienten-Abtastwert-Paar $COEF * SAMP\_FILT$ gebildet, weshalb das parallele Lesen von Koeffizient $COEF$ und Sample $SAMP\_FILT$/$SAMP\_LMS$ erforderlich ist. Die Adaption erfordert zusätzlich das Zurcükschreiben des aktualisierten Koeffizienten $COEF\_I$ und damit einen zusätzlichen Takt, in dem die Schreibadresse ADR_CW konstant bleibt. Die Leseadresse ADR_CR muss jedoch zur Bildung neuer Produkte aus Koeffizient $COEF$ und Sample $SAMP\_FILT$ inkrementiert werden. Daher wird der Koeffizientenspeicher mit einem 257x19-Bit `Dual-Port-RAM` *DP_RAM_COEF* realisiert, der durch zwei getrennte Zähler *ADRR_CNT_COEF* zum Lesen, und *ADRW_CNT_COEF* zum Schreiben adressiert wird. Diese Zähler werden dazu jeweils um einen Takt versetzt inkrementiert.
- In diesem System sind zwei parallele Pipelinestrukturen vorhanden. Eine dreistufige Pipeline zur Filterung setzt sich aus folgenden getakteten Komponenten zusammen:
  - 1. Stufe: Adresszähler *ADR_CNT_SAMP* und *ADRR_CNT_COEF*
  - 2. Stufe: Speicher *RAM_SAMP_FILT* und *DP_RAM_COEF*
  - 3. Stufe: Akkumulatorregister *REG_Y*

  Die zweite Pipeline zur Adaption besitzt vier Stufen und setzt sich wie folgt zusammen:
  - 1. Stufe: Adresszähler *ADR_CNT_SAMP* und *ADRR_CNT_COEF*
  - 2. Stufe: Speicher *RAM_SAMP_LMS* und *DP_RAM_COEF*
  - 3. Stufe: Adresszähler *ADRW_CNT_COEF*
  - 4. Stufe: Speicher *DP_RAM_COEF*

  Beide Pipelines werden durch den Mealy-Automaten *FSM_LMS_FIR* kontrolliert, wobei der Start durch einen verkürzten Ready-Impuls *ADC_FULL* initialisiert wird. Es folgt die Übernahme eines neuen Abtastwertes *XN* in die RAMs und die Bildung des Produktes *EMUE* aus Fehlersignalsample *EN* und Schrittweitenfaktor *MUE*. Anschließend laufen Filterung und Adaption parallel ab, bis beide Vorgänge nach der Bildung von L Produkten bzw. Summen und L Schreibvorgängen ins Koeffizienten-RAM abgeschlossen sind. Danach wird die Sättigung des Akkumulatorergebnisses Y und die Speicherung des letzten adaptierten Koeffizienten $COEF\_I$ vorgenommen, bevor der Automat das Akkumulationsregister *REG_Y* löscht und auf den nächsten *ADC_FULL*-Impuls wartet.

Bei einem externen Reset des Systems werden neben den Adresszählern ADR_CNT_SAMP, $ADRR\_CNT\_COEF$ und $ADRW\_CNT\_COEF$ sowie den Registern REG_Y und REG_EMUE auch die Speicher DP_RAM_COEF, RAM_SAMP_FILT und RAM_SAMP_LMS zurückgesetzt. Dazu wird an den Eingang eines Speichers ein Nullvektor angelegt und durch L Schreibbefehle an jeder Adresse abgelegt.

Dieser Reset muss durchgeführt werden, da die Speicher ansonsten veraltete Werte enthalten, die nicht der aktuellen Situation nach dem Reset entsprechen. Sowohl die Filterung als auch die Adaption startet ausgehend von einem definierten Zustand (vgl. Kap. 3.3) und darf daher nicht auf Koeffizienten und Referenzsignalwerte einer vorhergehenden Adaption zurückgreifen.

5.4.1 Multiplizierer-Akkumulatoreinheit

Der Filterungsvorgang in der `entity` *MAC_FIR* besteht aus L Multiplikationen und L Additionen. Für die Multiplikation wird ein embedded 18x18-Bit Multiplizierer [25] verwendet, woraus sich die Vektorlängen für die zu speichernden Abtastwerte $SAMP\_FILT$/$SAMP\_LMS$ und die Koeffizienten $COEF$ ergeben. Beide liegen als 18-Bit Werte im Q-Format [19] am Multiplizierer an. Der Sample-Wert *SAMP_FILT* des Referenzsignals XN wird im Q17-Format betrachtet mit einem Vorzeichen-Bit. Der Koeffizient *COEF* wird im Q15-Format, das sich aus dem Adaptionsvorgang ergibt (Kapitel 5.4.3), mit einem Vorzeichen-Bit und 2 Guard-Bits dargestellt. Durch die Multiplikation entsteht ein 36-Bit Produkt im Q32-Format mit einem Vorzeichen- und 3 Guard-Bits (Gleichnug 64).

$$\begin{aligned} P[35:0] &= P[sign, guard, guard, guard.31:0] \\ &= COEF[sign, guard, guard.14:0] * SAMP\_FILT[sign.16:0] \end{aligned} \tag{64}$$

Die Anzahl an Guard-Bits, die für das Akkumulationsergebnis benötigt wird, errechnet sich aus $\lceil ld(L)$ [19]. Damit im Worst-Case der Akkumulation von $L = 257$ 36-Bit Produkten, die alle den Wert $1 - 1\ LSB$ haben, kein Überlauf auftritt, werden daher 9 zusätzliche Guard-Bits benötigt. Da das Ergebnis eines Produktes bereits ein Guard-Bit enthält, werden die Summanden *Y* und *P* um 8 Bits erweitert und liefern ein 44-Bit Ergebnis im Q32-Format mit einem Vorzeichen-Bit und 11 Guard-Bits (Gleichung 65).

$$\begin{aligned} S[43:0] &= S[sign, guard, \ldots, guard.31:0] \\ &= Y[sign, guard, \ldots, guard.31:0] \\ &\quad +P[sign, \ldots, sign, guard, guard, guard.31:0] \end{aligned} \tag{65}$$

Bei der Addition muss ein Summand auf der rechten Seite die gleiche Vektorbreite aufweisen wie das Ergebnis auf der linken Seite. Dies ist durch die Rückkopplung des 44-Bit Akkumulationsregisters erfüllt. Der zweite Summand wird durch den Simulationscompiler ModelSim und das Synthesewerkzeug ISE implizit, vorzeichenrichtig erweitert [19] (vgl. Abb. 40).

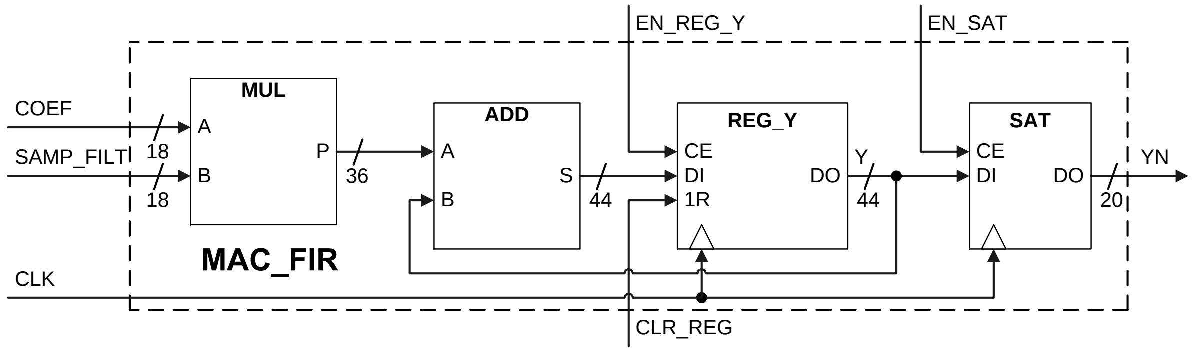

Abb. 40: Multiplizierer-Akkumulatoreinheit: 18x18-Bit Multiplizierer MUL, 42-Bit Addierer ADD mit $\lceil ld(L) \rceil = 9$ *Guard-Bits, 42-Bit Akkutmulationsregister REG_Y, Sättigungsmodul SAT (vgl. Code S. 110).* $f_{CLK} = 50\ MHz$

Nach der Akkumulation von L Produkten wird das Sättigungsmodul SAT aktiviert wodurch das 44-Bit Akkumulationsergebnis *Y* auf den 20-Bit Ergebnisvektor *YN* im Q19-Format reduziert wird (vgl. Abb. 41). Gleichzeitig erfolgt die Bildung des Zweierkomplements, um das Vorzeichen des Ausgangssignals YN umzukehren. Dies geschieht zur Erzeugung des Fehlersignals *EN*, welches nicht intern berechnet wird, sondern durch die Überlagerung des Kompensationssignals *YN* mit dem dem Primärsignal *DN* am Inohr-Mikrofon gebildet wird und daher eine Phasendrehung des Kompensationssignals YN um 180° erfordert. Anschließend wird der Inhalt des Akkumulationsregisters *Y* durch das *CLR_REG*-Signal gelöscht, damit der nächste Berechnungszyklus keinen Summen-Offset aus dem vorherigen Zyklus erhält [19].

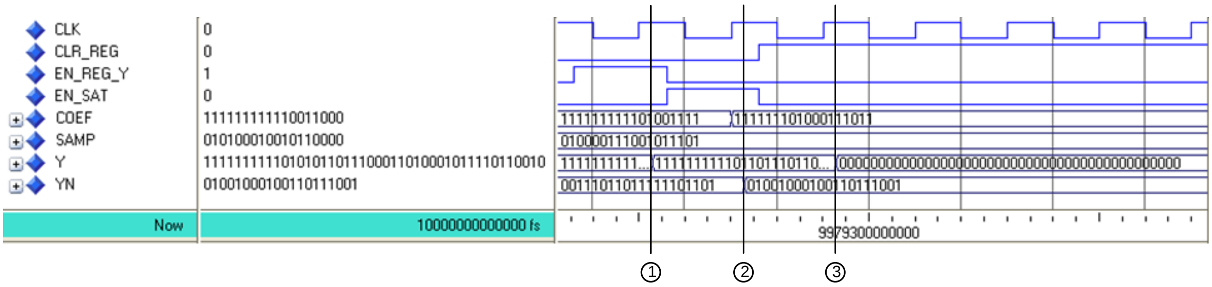

Abb. 41: VHDL-Simulation der MAC-Einheit. Abschluss der Akkumulation (Marker 1); Sättigung des Ergebnisses und Bildung des Zweierkomplements (Marker 2); sowie Rücksetzen des Akku-Registers (Marker 3)

5.4.2 Block-RAM für die Abtastwerte des Referenzsignals *XN*

Die Speicherung der Abtastwerte $SAMP\_FILT$ des Referenzsignals *XN* für die MAC-Einheit erfolgt in einem als Ringpuffer realisierten Block-RAM. Dazu wird der Speicher nach einem System-Reset mit jedem *ADC_FULL*-Impuls von der Adresse $A_S(0)$ an mit Samples gefüllt. Nach $L = N + 1$ Schreibvorgängen ist der Ringpuffer gefüllt und der jeweils älteste Abtastwert wird mit einem neue Abtastwert überschrieben (vgl. Abb. 42). In der Zeit zwischen zwei Schreibvorgängen werden in der MAC-Einheit L Produkte $x(n-i) * c_i$ gebildet und akkumuliert, wobei c_0 immer mit dem aktuellsten Abtastwert $x(n)$ und c_N mit dem ältesten Abtastwert $x(n-N)$ ein Faktoren-Paar bildet.

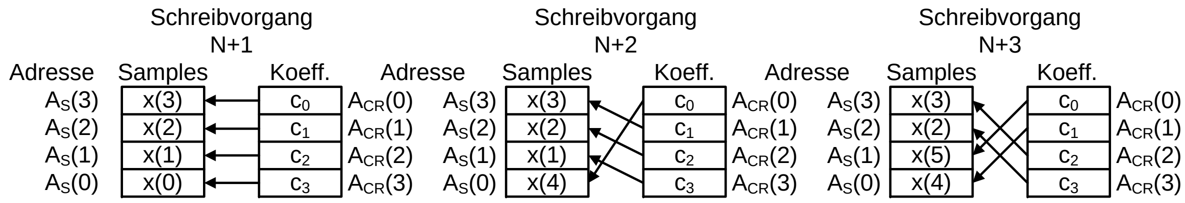

Abb. 42: Beispielsequenz für die Kombination der Koeffizienten mit den im Ringpuffer sukzessive gespeicherten Abtastwerten. Filterordnung $N = 3$ [19]

Zur Bildung eines Faktoren-Paares, werden die Adresszähler des Sample- und Koeffizienten-RAMs durch den Automaten *FSM_LMS_FIR* gesteuert und besitzen folgende Zählweisen:

- Der Zähler *ADR_CNT_SAMP* (vgl. Code S. 107 Z. 37-48) steht zu Beginn des Zyklus auf der letzten Schreibposition, die auch der ersten Leseposition entspricht, und wird dann dekrementiert (vgl. Abb. 43). Nach N Dekrementen, in denen die Abtastwerte $x(n-1)$ bis $x(n-N)$ adressiert und gelesen wurden, erfolgt die Fixierung der letzten Lese-Adresse, da dies auch die Schreibposition für den nächsten Abtastwert ist (vgl. Abb. 44). Wenn der Zähler die Untergrenze $A_S(0)$ im Laufe einer Sequenz erreicht, findet ein Überlauf auf den höchsten Adresswert $A_S(256)$ statt.
- Die Lese-Adresse *ADR_CR* des Dual-Port-RAM für die Koeffizienten wird parallel durch einen Inkrementer *ADRR_CNT_COEF* (vgl. Code S. 109 Z. 58-69) adressiert (vgl. Abb. 43). Dieser zählt in jedem Zyklus von $A_{CR}(0)$ bis $A_{CR}(256)$, um die Koeffizienten von c_0 bis c_{256} zu lesen. Am Ende der Sequenz führt er einen Überlauf auf die Adresse $A_{CR}(0)$ des Koeffizienten c_0 durch, der jeweils im nächsten Zyklus mit dem aktuellsten Abtastwert multipliziert wird (vgl. Abb. 44). Die um einen Takt versetzte Inkrementierung der Lese- und Schreibadresse des Koeffizienten-RAMs erfordert die gepulsten Enable-Signale für die Adress-Zähler.

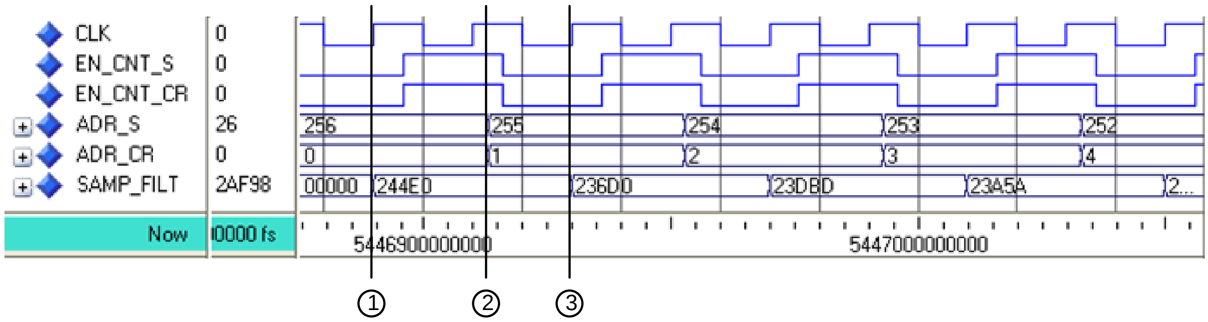

Abb. 43: VHDL-Simulation der RAM-Einheit beim Start der Adressierungssequenz. Adresszählerstände nach $L = 257$ Schreibvorgängen, Datum der letzten Schreib-Adresse $ADR\_S = 256$ erscheint am Ausgang(Marker 1); Dekrementierung der Sample-Adresse ADR_S und Inkrementierung der Koeffizienten-Adresse ADR_CR (Marker 2); Datum der Adresse $ADR\_S = 255$ erscheint am Ausgang (Marker 3)

Das von der Struktur mit dem *RAM_SAMP_FILT* identische Speicher-Modul der Abtastwerte $SAMP\_LMS$ für die LMS-Adaption *RAM_SAMP_LMS* wird parallel durch den selben Zähler adressiert. Die für den Algorithmus erforderliche Verzögerung des Referenzsignals *XN* (vgl. Kap. 4.2) wird durch eine, vor den Eingang des Speichers geschaltete, 64x18-Bit Registerkette X_DELAY realisiert (vgl. Abb. 45). Diese Kette muss jedes Sample XN durchlaufen, um die in Kapitel 4.1 ermittelte Verzögerung zu erhalten. Dazu wird das Signal *EN_X_SHIFT* zu Beginn eines Zyklus parallel zur Schreibfreigabe *WE_R_S* gesetzt, um einen neuen Abtastwert XN in die Kette X_DELAY aufzunehmen und jedes vorhandene Sample um eine Position weiterzuschieben (vgl. Abb. 46).

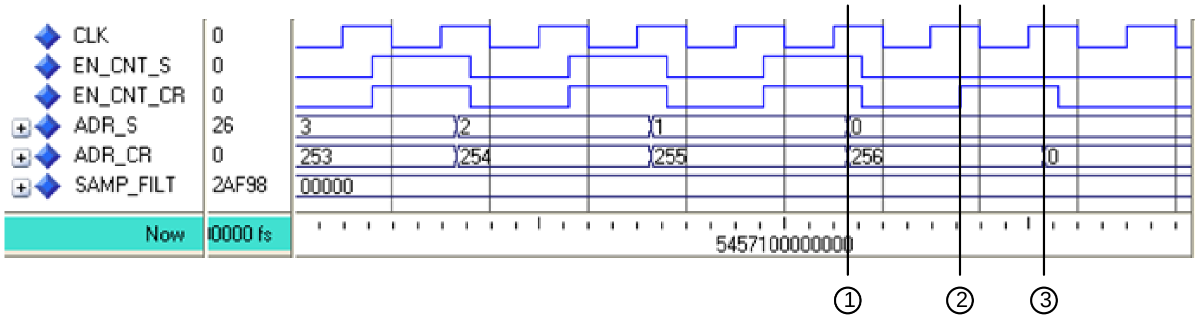

Abb. 44: VHDL-Simulation der RAM-Einheit beim Abschluss der Adressierungssequenz. Adressierung des letzten Faktorenpaares $x(0)$ und c_{256} (Marker 1); Enable-Signal für den Zähler des Koeffizientenspeichers wird gesetzt, um den Überlauf auf 0 durchzuführen (Marker 2); Sample-Adresse $A(0)$ wird beibehalten zur Speicherung des nächsten Samples, Überlauf der Koeffizienten-Adresse auf 0 (Marker 3)

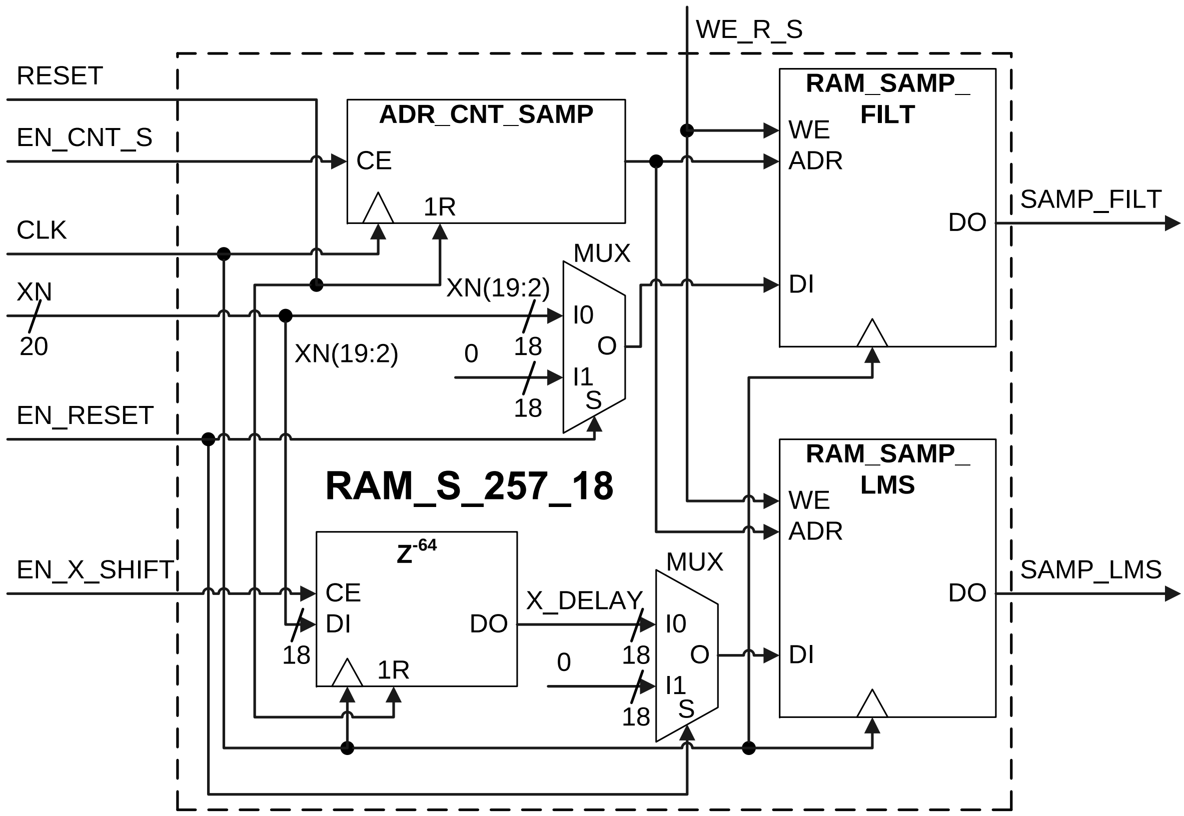

Abb. 45: RAM-Einheit für die Abtastsamples des Referenzsignals bestehend aus zwei 257x18-Bit Block-RAMs RAM_SAMP_FILT und RAM_SAMP_LMS, zwei Multiplexern MUX zum Umschalten des Eingangsdatums bei einem Reset, einem Adresszähler ADR_CNT_SAMP und einem z^{-64} Verzögerungsglied, das sich aus einer 64x18-Bit Registerkette X_DELAY zusammensetzt. $f_{CLK} = 50\ MHz$

Zum Rücksetzen der Speicher RAM_SAMP_FILT und RAM_SAMP_LMS in einen definierten Ausgangszustand im Falle eines System-Reset, wird an jede Speicheradresse ADR_S eine 0 geschrieben. Dazu werden über zwei Multiplexer, die durch das Signal *EN_RESET* gesteuert werden, Nullvektoren an die Eingänge der RAMs gelegt (vgl. Abb. 45). Durch die Adressierung jeder Speicherposition und gleichzeitige Aktivierung des Schreibsignals *WE_R_S* wird der Inhalt beider Speicher gelöscht.

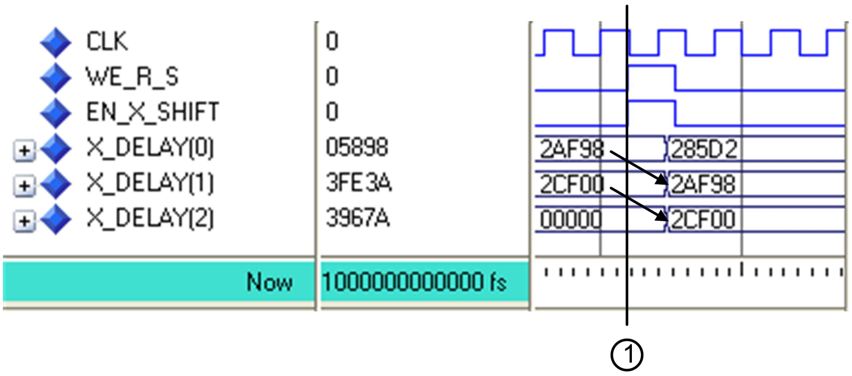

Abb. 46: VHDL-Simulation des Schiebevorgangs in der Verzögerungskette X_DELAY. Enable-Signal EN_X_SHIFT zum Schieben wird gesetzt (Marker 1); Die Abtastwerte in X_DELAY(0) und X_DELAY(1) werden um jeweils eine Position weitergeschoben; Ein neues Sample wird in das Register X_DELAY(0) geschrieben

5.4.3 LMS-Adaptionseinheit

In der `entity` *MAC_LMS* werden die Koeffizienten $COEF$ des Filters adaptiert und in einem Dual-Port-RAM DP_RAM_COEF gespeichert. Die Adaption eines Koeffizienten $COEF$ erfolgt durch seine Addition mit dem Produkt $XEMUE$ aus Fehlersignal *EN*, Schrittweitenfaktor *MUE* und dem zugehörigen Referenzsignalwert *SAMP_LMS* (vgl. Kap. 3.3.1 Gl. 22) (vgl. Abb. 47). Das Referenzsignal *SAMP_LMS* entspricht dem zuvor in der RAM-Einheit *RAM_S_257_18* verzögerten Signal *X_DELAY* (vgl. Kap. 4.2)(vgl. Abb. 45). Das Referenzsignal *XN* wird im Eingangspfad der Adaption verzögert, um die Gruppenlaufzeit des Sekundärpfades (vgl. Kap. 4.2 Abb. 15) und die damit verbundene unerwünschte Phasenverschiebung des Kompensationssignals *YN* am Inohrmirkofon auszugleichen.

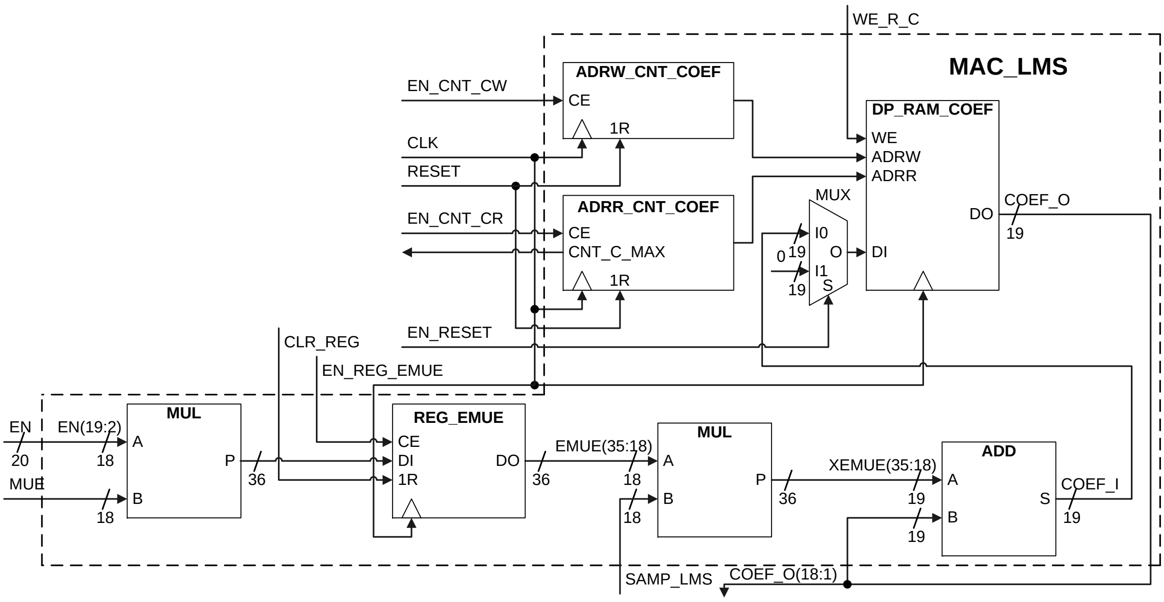

Abb. 47: LMS-Adaptionseinheit MAC_LMS: zwei 18x18-Bit Multiplizierer MUL, 18-Bit Addierer ADD mit $n = 2$ Summanden und $\lceil 3,32 log(n) = 1$ Guard-Bit, 36-Bit Multiplikationsregister REG_EMUE, 257x19-Bit Dual-Port-Block-RAM DP_RAM_COEF, Multiplexer MUX zum Umschalten des Eingangsdatums bei einem Reset, zwei Adresszähler ADRW_CNT_COEFfür die Schreibadresse und ADRR_CNT_COEF für die Leseadresse. $f_{CLK} = 50\ MHz$

5.4.3.1 Adaption der Filterkoeffizienten

Die Adaption der Filterkoeffizienten $COEF$ erfolgt parallel zur Filterung des Referenzsignals *XN*. Die Koeffizienten werden dabei aus dem RAM DP_RAM_COEF gelesen, um für den sequentiellen Filterzyklus und die Adaption zur Verfügung zu stehen. Das Rückschreiben des adaptierten Koeffizienten $COEF\_I$ geschieht um einen Takt versetzt zum Lesen, was eine Realisierung des Koeffizientenspeichers DP_RAM_COEF als Dual-Port-RAM erfordert, wobei ein Port zum Lesen und der andere zum Schreiben genutzt wird.

Nach dem Empfang eines neuen Fehlersignal-Samples *EN* durch das Codec-Interface wird zunächst das Produkt $EMUE$ aus Fehlersignal EN und Schrittweitenfaktor MUE gebildet. Da auch in dieser `entity` die embedded 18x18-Bit Multiplizierer genutzt werden, sind die Faktoren *EN* und *MUE* im Q17-Format zu betrachten. Daraus ergibt sich ein 36-Bit Produkt im Q34-Format mit einem Vorzeichen- und einem Guard-Bit (Gl. 66).

$$P[35:0] = EMUE[sign, guard.33:0] = EN[sign.16:0] * MUE[sign.16:0] \tag{66}$$

Dieses Produkt $EMUE$ bleibt bis zum Empfang des nächsten Fehlersignal-Samples EN konstant und wird daher für den Adaptionsvorgang im Register *REG_EMUE* gespeichert.

In einem Durchlauf werden $L = 257$ Koeffizienten adaptiert. Dazu sind auch in dieser `entity` L Multiplikationen und L Additionen durchzuführen. Das Produkt *EMUE* wird dabei mit dem, zum jeweiligen Koeffizienten $COEF$ gehörenden, Abtastwert $SAMP\_LMS$ des verzögerten Referenzsignals *XN* multipliziert (vgl. Kap. 4.2). Dazu wird es, durch Abschneiden der niederwertigen Bits auf 18 Bit gekürzt und im Q16-Format mit einem Vorzeichen- und einem Guard-Bit betrachtet. Durch die Multiplikation mit dem 18-Bit Faktor *X*, der im Q17-Format mit einem Vorzeichen-Bit betrachtet wird, entsteht ein 36-Bit Produkt im Q33-Format mit einem Vorzeichen- und 2 Guard-Bits (Gleichung 67).

$$\begin{aligned} P[35:0] &= P[sign, guard, guard.32:0] \\ &= EMUE[sign, guard.15:0] * X[sign.16:0] \end{aligned} \tag{67}$$

Dieses Produkt $XEMUE$ wird mit dem zugehörigen Koeffizienten $COEF$ addiert. Die Addition wird im Q16-Format durchgeführt. Dazu wird das Produkt *P* auf 19 Bits gekürzt und mit dem aktuellen Koeffizienten $COEF$ zu einem neuen Koeffizienten $COEF\_I$ mit einem Vorzeichen- und 2 Guard-Bits akkumuliert (Gleichung 68). Die Kürzung der Vektorlänge auf 19 Bits erfolgt an dieser Stelle, da ein Koeffizient $COEF$ in der MAC-Einheit *MAC_FIR* zur Multiplikation mit einem Sample $SAMP\_FILT$ durch einen embedded 18x18-Bit Multiplizierer im 18-Bit-Format vorliegen muss. Zur Bildung der Summe S wird ein zusätzliches Guard-Bit eingerichtet um einen Überlauf zu Verhindern. Durch die Speicherung der Koeffizienten als 19-Bit Vektoren werden, im Gegensatz zu einer Speicherung als 36-Bit Vektoren, zusätzlich Block-RAM Ressorucen gespart.

$$\begin{aligned} S[18:0] &= S[sign, guard, guard.15:0] \\ &= XEMUE[sign, guard, guard.15:0] \\ &\quad + COEF\_O[sign, guard, guard.15:0] \end{aligned} \tag{68}$$

5.4.3.2 Dual Port Block-RAM für die Koeffizienten

Bei der Adaption eines Koeffizienten $COEF$ wird der aktuelle Koeffizient $COEF\_O$ an der Adresse ADR_CR des Speichers DP_RAM_COEF gelesen, und nach der Addition mit dem Produkt *XEMUE* an die Adresse ADR_CW als neuer Koeffizient $COEF\_I$ zurückgeschrieben. Parallel zur Adaption wird der am Ausgang anliegende Koeffizient $COEF\_O$ in der `entity` *MAC_FIR* mit einem Abtastwert $SAMP\_FILT$ multipliziert. Sowohl Filterung als auch Adaption benötigen dazu einen Adressierungsvorlauf des Koeffizienten $COEF\_O$, damit dieser parallel zum Sample $SAMP\_FILT/SAMP\_LMS$ des Referenzsignals *XN* anliegt. Die Adaption erfordert zusätzlich das Zurückschreiben des Koeffizienten $COEF\_I$, sodass die Schreibadresse ADR_CW noch konstant bleibt, während die Leseadresse ADR_CR bereits inkrementiert wird.

Daher ist der Koeffizientenspeicher *DP_RAM_COEF* als Dual-Port-RAM mit einem Lese- und einem Schreib-Port realisiert worden. Beide Ports werden durch seperate Zähler $ADRW\_CNT\_COEF$ und $ADDR\_CNT\_COEF$ adressiert, die genau um eine Taktperiode versetzt inkrementiert werden (vgl. Abb. 48). Der Stand des Zählers *ADRR_CNT_COEF* für die Lese-Adresse wird von einem Komparator kontrolliert, der bei dem Maximalwert $N = 256$ den Abschluss eines Adressierungszyklus durch das Statusbit *CNT_C_MAX* signalisiert.

Die Adressierung des Dual-Port-RAM *DP_RAM_COEF* besitzt folgenden Ablauf:

- Datum $COEF\_O$ der aktuellen Lese-Adresse ADR_CR erscheint am Ausgang und wird zur Filterung des Referenzsignals XN genutzt sowie parallel zu einem neuen Koeffizienten $COEF\_I$ adaptiert (vgl. Abb. 48 Marker 1)
- Die Lese-Adresse wird inkrementiert, damit beim nächsten Takt ein neues Datum zur Verarbeitung bereit steht, während die Schreib-Adresse ADR_CW zum Zurückschreiben des adaptierten Koeffizienten $COEF\_I$ konstant bleibt(vgl. Abb. 48 Marker 2)
- Der adaptierte Koeffizient $COEF\_I$ wird an die aktuelle Schreib-Adresse ADR_CW des RAMs geschrieben und diese anschließend inkrementiert (vgl. Abb. 48 Marker 3)

Dieses FSM/Timer-Konzept wird zur Kopplung des Datenpfades mit dem Steuerpfad wird verwendet, da der Timer-Grenzwert *CNT_C_MAX* in zwei Zuständen geprüft wird. Der Komparator muss so nicht mehrfach in der FSM realisiert werden, sondern wird direkt in der Timer-Entity *ADRR_CNT_COEF* plaziert und spart dadurch Logik-Ressourcen ein [19].

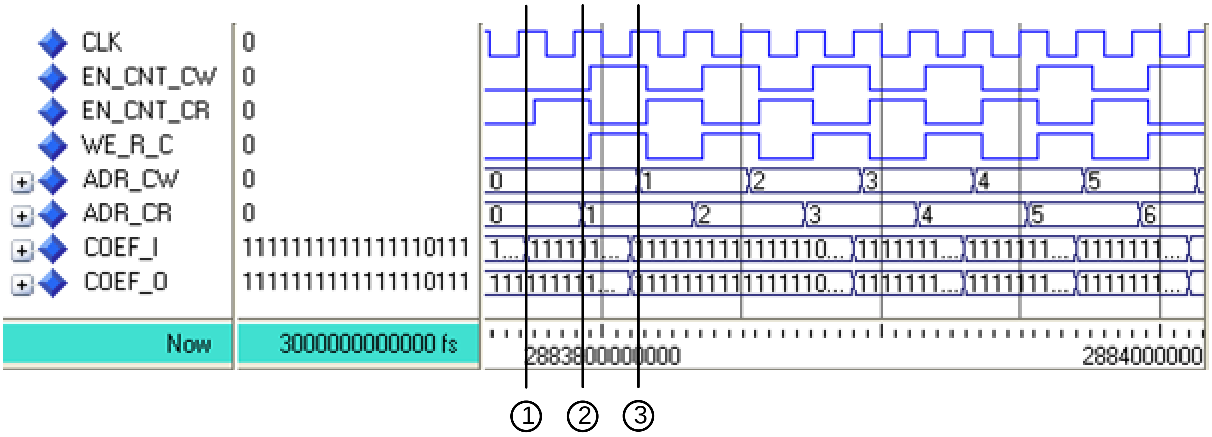

Abb. 48: VHDL-Simulation der MAC_LMS-Einheit beim Adressieren des Dual-Port-RAM DP_RAM_COEF

5.4.4 Zustandsautomat zur Steuerung des adaptiven Filters

Der Zustandsautomat FSM_LMS_FIR koordiniert folgende Vorgänge im Datenpfad ((vgl. Abb. 38):

- Die Speicherung eines neuen Abtastwertes XN des Referenzsignals und den gleichzeitigen Schiebevorgang in der Registerkette X_DELAY
- Die Bildung des Produktes $EMUE$ aus Fehlersignal-Sample EN und Schrittweitenfaktor MUE
- Die anschließend im Wechsel ablaufenden Lese- und Schreibvorgänge der Koeffizienten $COEF$ / $COEF\_I$ und Abtastwerte $SAMP\_FILT$ / $SAMP\_LMS$

- Die Freigabe des Akkumulationsregisters REG_Y in der `entity` *MAC_FIR*
- Die Übernahme des Akkumulationsergebnisses Y in das Ausgangsregister YN des Sättigungsmoduls SAT

Außerdem leitet der Automat bei einem Reset eine Sequenz zur Rücksetzung der internen Speicherblöcke ein.

Durch die in Kapitel 5.4.2 und 5.4.3.2 erläuterte Zählerstrategie ist durch die gewählten Zählrichtungen und Wrap-Arounds nur noch ein geeignetes Timing für die Zählfreigaben erforderlich. Die Sequenz des Automaten wird durch die Statusbits *ADC_FULL* und *CNT_C_MAX* kontrolliert (vgl. Abb. 49).

Der hier vorgestellte Automat besitzt eine Mealy-Charakterisitk, da die Steuersignale *WE_R_S* und *EN_X_SHIFT* unter direktem Einfluss des Statusbits *ADC_FULL* stehen und somit das wesentliche Kennzeichen eines Mealy-Automaten, die direkte Abhängigkeit zwischen Eingangs- und Ausgangssignalen, gegeben ist. Die Steuerausgänge, die nur durch den jeweiligen Zustand bestimmt werden, weisen einer Moore-Charakteristik auf [19]. Die Steuersignale, die nicht durch den aktuellen Zustand oder in Abhängigkeit eines Statusbits gesetzt werden, erhalten die Defaultzuweisung '0' (vgl. Abb. 49).

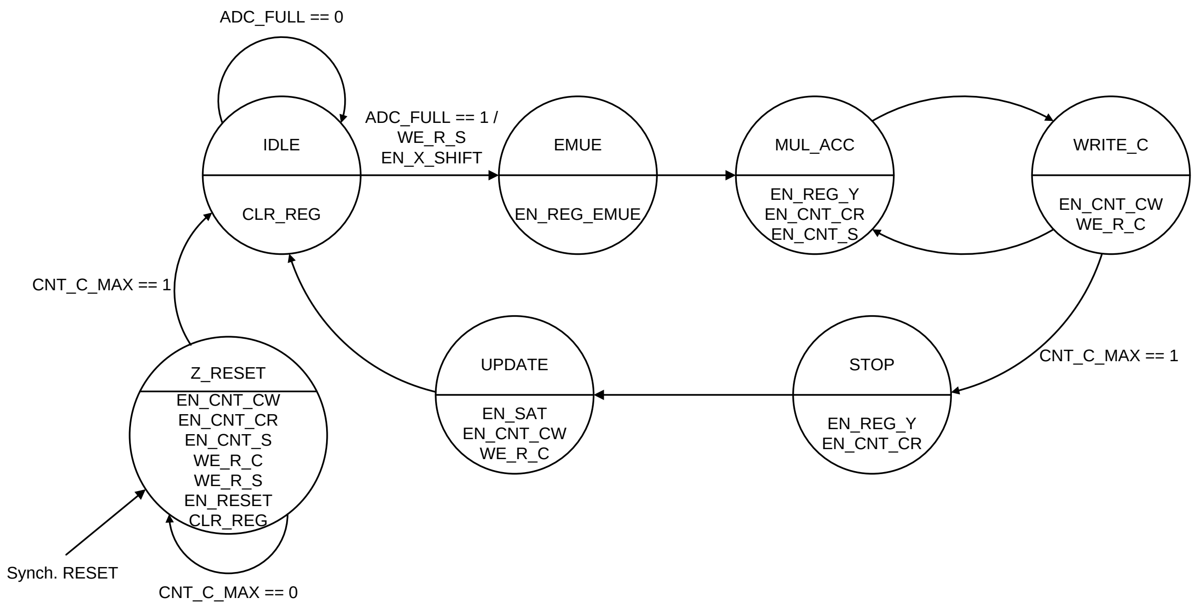

Abb. 49: Zustandsdiagramm der Sequenzsteuerung des adaptiven FIR-Filters mit MAC-Einheit und LMS-Adaption; Mealy-FSM mit Moore-Ausgängen [19]

Die Zustände des Automaten werden in folgender Sequenz durchlaufen:

- **IDLE**: Zur Löschung der Ergebnisse aus dem vorherigen Berechnungszyklus, werden das Akkumulationsregister *REG_Y* und das Register *REG_EMUE* zur Speicherung des Produktes $EN*MUE$ durch das Signal *CLR_REG* zurückgesetzt. Durch einen *ADC_FULL*-Impuls erfolgt die Schreibfreigabe *WE_R_S* zur Speicherung eines neuen Samples XN, sowie die Schiebefreigabe für die Registerkette *X_DELAY*.
- **EMUE**: Das Produkt $EMUE$ aus Schrittweitenfaktor *MUE* und Fehlersignal-Sample *EN* wird berechnet und im Register *REG_EMUE* abgelegt, dessen Freigabe durch das Signal *EN_REG_EMUE* erfolgt.
- **MUL_ACC / WRITE_C**: In diesen beiden Zuständen laufen zwei Pipeline-Vorgänge parallel ab. Eine dreistufige Pipeline für den Filterungszyklus ergibt sich aus folgenden Vorgängen (vgl. Abb. 50):

- Aktualisierung der Adressen *ADR_S*$(n-(i+1))$ und *ADR_CR*$(i+1)$
- An den Speicherausgängen erscheint ein Wertepaar bestehend aus *SAMP_FILT*$(n-i)$ und *COEF*(i)
- Ein Produkt aus *SAMP_FILT*$(n-(i-1)) *$ *COEF*$(i-1)$ wird akkumuliert und im Register *REG_Y* gespeichert, dessen Freigabe durch das Signal *EN_REG_Y* erfolgt

Eine vierstufige Pipeline zur Adaption der Koeffizienten setzt sich aus folgenden Schritten zusammen (vgl. Abb. 50):

- Aktualisierung der Adressen *ADR_S*$(n-(i+1))$ und *ADR_CR*$(i+1)$
- An den Speicherausgängen erscheint ein Wertepaar bestehend aus *SAMP_LMS*$(n-i)$ und *COEF*(i)
- Aktualisierung der Adresse *ADR_CW*$(i+1)$
- Akkumulation eines Produktes aus *SAMP_LMS*$(n-(i-1)) *$ *EMUE*(n) auf den aktuellen Koeffizienten *COEF*(i) und Speicherung des adaptierten Koeffizienten $COEF\_I$ an der Adresse *ADR_CW*(i)

Das Statusbit *CNT_C_MAX* signalisiert, dass die höchste Lese-Adresse $i = N$ am Koeffizientenspeicher DP_RAM_COEF anliegt und beide Pipelines werden in den folgenden Zuständen deaktiviert.

- **STOP**: Es erfolgt die letzte Multiplikation *SAMP_FILT*$(n-N) *$ *COEF*(N) und Akkumulation des Ergebnisses (*EN_REG_Y*). Anschließend führt die die Lese-Adresse ADR_CR des Koeffizientenspeichers beim Übergang in den nächsten Zustand einen Wrap-Around auf den Anfangswert durch (*EN_CNT_CR*) während die Sample-Adresse *ADR_S* für den nächsten Schreibvorgang unverändert bleibt.
- **UPDATE**: Das Akkumulationsergebnis *Y* durchläuft die Sättigungsfunktion und bleibt während des folgenden Zyklus im Ausgangsregister *YN* gespeichert (*EN_SAT*). Der letzte adaptierte Koeffizient *COEF_I*(N) wird gespeichert (*WE_R_C*) und die Schreibadresse ADR_CW führt beim Übergang in den nächsten Zustand ebenfalls einen Wrap-Around auf den Anfangswert durch (*EN_CNT_CW*)

Wenn durch den Benutzer ein *RESET* ausgelöst wird, wechselt der Automat in den Zustand **Z_RESET**, unabhängig davon in welchem Zustand er sich gerade befindet. In diesem Zustand werden die Register *REG_Y* und *REG_EMUE* (*CLR_REG*) sowie alle Block-RAMs gelöscht. Dazu werden die Multiplexer vor den Speichereingängen umgeschaltet und an jeden Eingang ein Nullvektor gelegt (*EN_RESET*). Es werden die Schreibfreigaben *WE_R_C* und *WE_R_s* gesetzt und der gesamte Adressbereich der Speicher durchlaufen (*EN_CNT_CW*, *EN_CNT_CR*, *EN_CNT_S*). Das Ende des Adressierungszyklus wird durch *CNT_C_MAX* signalisiert. Da dieses Statusbit in Abhängigkeit des Koeffizienten-Lesezählers $ADRR\_CNT\_COEF$ gesetzt wird, muss auch dieser aktiviert werden. Nach dem Abschluss des Zyklus stehen alle Adressen wieder auf ihrem Ausgangswert und der Automat wechselt in den Zustand *IDLE*.

Dieser Reset-Vorgang der Speicher wird parallel zur Konfiguration des Audio-Codecs durchgeführt, die eine Anlaufphase von fünf Abtastperioden erfordert. Dadurch ensteht durch den Adressierungszyklus keine zusätzliche Verzögerung nach einem Reset (vgl. Abb. 51).

Die übergeordnete `entity` *LMS_FIR* enthält die FSM und die Synchronisation des Ready-Impulses als nebenläufige Prozesse, sowie die Instanziierungen der Datenpfad-Komponenten (vgl. Code S. 105). Die Moore- und bedingten Mealy-Ausgänge werden im FSM-Ausgangsschaltnetz des Prozesses *UE_AUS_SN* sequentiell aufgelistet. Durch die vollständige Ansteuerung der Ausgänge in den Default- und Moore-Zuweisungen sind keine `else`-Pfade erforderlich, wodurch unerwünschte Speichereffekte durch Latches vermieden werden [19].

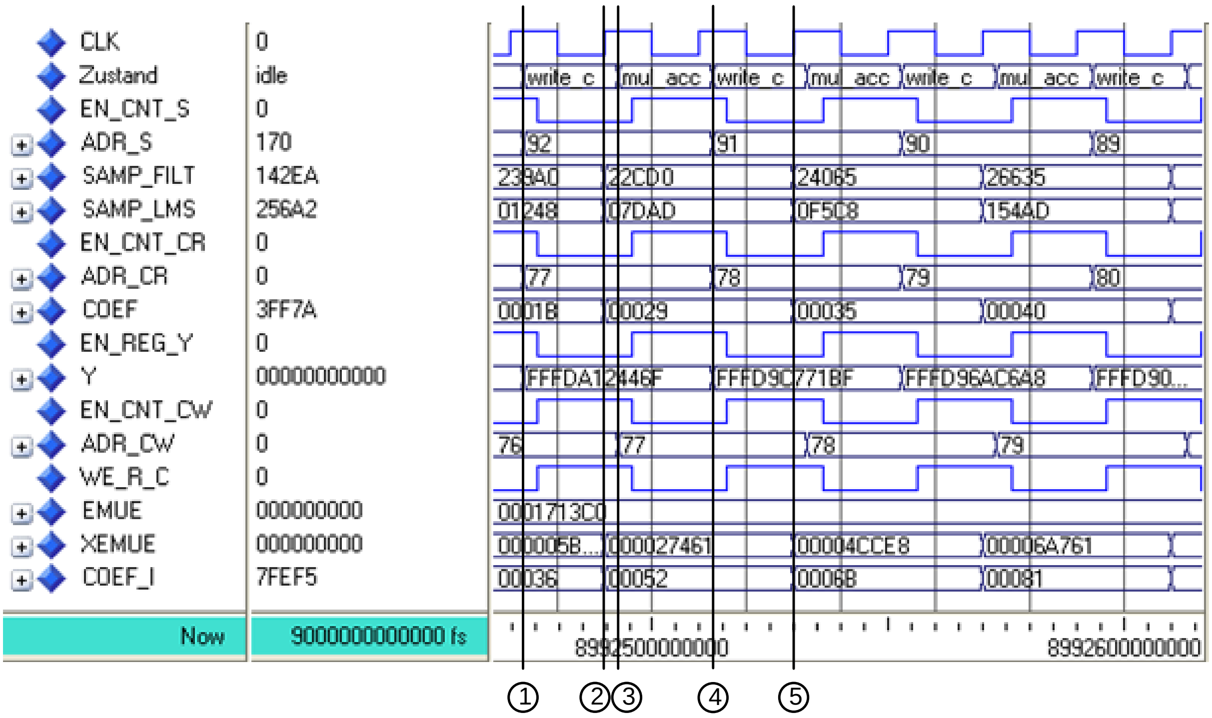

Abb. 50: VHDL-Simulation der FSM_LMS_FIR beim Durchlaufen der Pipelinestufen. Aktualisierung der Lese-Adressen ADR_S und ADR_CR (Marker 1); An den Speicherausgängen SAMP_FILT, SAMP_LMS und COEF erscheinen neue Werte (Marker 2); Aktualisierung der Schreibadresse ADR_CW (Marker 3); Akkumulation und Speicherung in REG_Y (Marker 4); Speicherung des adaptierten Koeffizienten $COEF\_I$ an der Adresse ADR_CW (Marker 5)

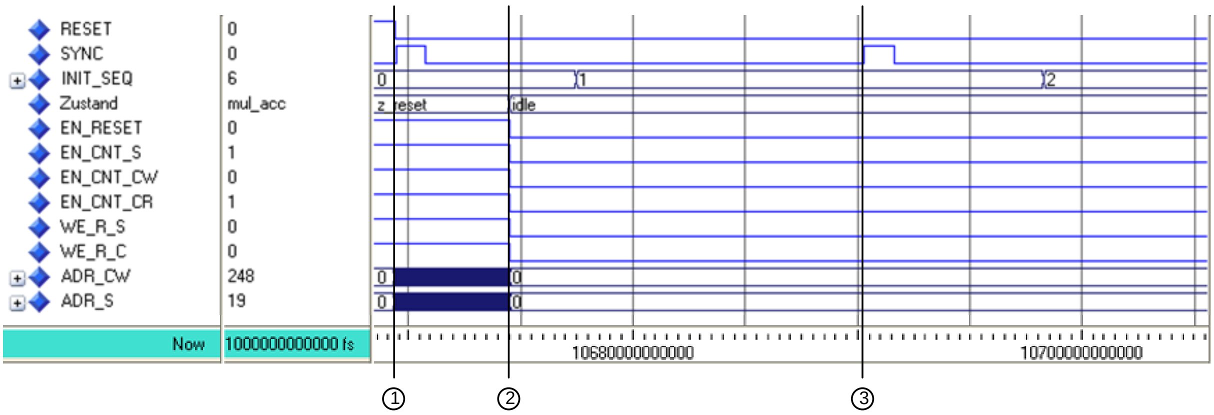

Abb. 51: VHDL-Simulation der Anlaufphase nach einem System-Reset. Das Reset-Signal wechselt auf den Low-Pegel und die Anlaufphase startet mit dem Adressierungszyklus und der ersten Abtastperiode, markiert durch das SYNC-Signal (Marker 1); der Adressierungszyklus ist abgeschlossen und der Automat geht in den Zustand IDLE über (Marker 2); die erste von fünf Abtastperioden der Anlaufphase des Codec-Interfaces ist beendet (Marker 3)

6 Syntheseergebnisse und Timing Simulationen des implementierten Systems

In diesem Abschnitt werden zunächst die Syntheseergebnisse des Werkzeugs ISE 9.1.03i dargestellt und erläutert. Dazu wurde das gesamte System bestehend aus Codec-Interface und adaptiven FIR-Filter synthetisiert. Es wird deutlich, dass durch die strukturierte Entwicklung und die Einhaltung eines synthesegerechten Codierstils die in den Komponenten des Systems enthaltenen Objekte als Macros erkannt und entsprechend umgesetzt werden.

Anschließend wird durch Timing Simulationen, die mit dem Simulationscompiler ModelSim durchgeführt wurden, die Funktionalität sowohl des Interfaces, als auch des Filters gezeigt und verifiziert.

6.1 Syntheseergebnisse

Das Syntheseergebnis zeigt, wie die zuvor beschriebenen Strukturen des Codec-Interfaces und des adaptiven FIR-Filters umgesetzt werden. Der Synthesereport listet die in den Komponenten enthaltenen Objekte auf.

Eingangs- und Ausgangsregister des Codec-Interfaces:

```
    Found 20-bit register for signal <ADR\_REG\_IN>.
    Found 20-bit register for signal <ADR\_REG\_OUT>.
    Found 40-bit register for signal <AUDIO\_REG\_IN>.
    Found 40-bit register for signal <AUDIO\_REG\_OUT>.
    Found 20-bit register for signal <DATA\_REG\_IN>.
    Found 20-bit register for signal <DATA\_REG\_OUT>.
    Found 16-bit register for signal <TAG\_REG\_IN>.
    Found 16-bit register for signal <TAG\_REG\_OUT>.
```

FSM des adaptiven Filters:

```
    Found finite state machine <FSM_0> for signal <ZUSTAND>.\\
    -------------------------------------------
    | States             | 7                    |
    | Transitions        | 10                   |
    | Inputs             | 2                    |
    | Outputs            | 9                    |
    | Clock              | CLK (rising_edge)    |
    | Reset              | RESET (positive)     |
    | Reset type         | synchronous          |
    | Reset State        | z_reset              |
    | Power Up State     | z_reset              |
    | Encoding           | automatic            |
    | Implementation     | LUT                  |
    -------------------------------------------
```

Macro Statistics des Gesamtsystems:

```
Macro Statistics
# Multipliers              : 3
 18x18-bit multiplier      : 3
# Adders/Subtractors       : 3
 19-bit adder              : 1
```

```
 20-bit adder               : 1
 5-bit adder                : 1
# Counters                  : 7
 3-bit up counter           : 1
 4-bit up counter           : 1
 8-bit up counter           : 1
 9-bit down counter         : 1
 9-bit up counter           : 2
# Accumulators              : 1
 44-bit up accumulator      : 1
# Registers                 : 106
 16-bit register            : 1
 18-bit register            : 64
 20-bit register            : 5
 36-bit register            : 1
 40-bit register            : 2
 5-bit register             : 1
```

Die FSM wird mit sieben Zuständen und zehn Übergängen gemäß Abg. 49 erzeugt. Für die Multiplikationen in *MAC_FIR* und *MAC_LMS* werden drei 18x18-Bit-Multiplizierer synthetisiert und für die zur Adaption eines Filterkoeffizienten $COEF$ notwendige Addition ein 19-Bit-Addierer. Die Bildung des Zweierkomplements des Ausgangssignal *YN*, zur 180° Phasendrehung erfordert einen zusätzlichen 20-Bit-Addierer. Die Adresszähler der Speicher werden als ein 9-Bit-down-counter (*ADR_CNT_SAMP*) und zwei 9-Bit-up-counter (*ADRW_CNT_COEF* und *ADRR_CNT_COEF*) synthetisiert. Die Registerkette zur Verzögerung des Referenzsignals *XN* um 64 Samples besteht aus der entsprechenden Anzahl an 18-Bit-Registern.

Die Eingangs- und Ausgangsregister des Codec-Interfaces werden erkannt und mit der passenden Datenbreite erzeugt. Zusätzlich wird ein 8-Bit-up-counter für den *BIT_CLK*-Zähler aus dem *GEN_SYNC*-Prozess, ein 3-Bit-up-counter für den Zähler *INIT_SEQ* und ein 4-Bit-up-counter für den Slot-Zähler *SLOT_COUNT* generiert. Auch der 44-Bit-Akkumulator der MAC-Unit wird korrekt synthetisiert.

Lediglich der Zähler *SLOT_BIT_COUNT* für die Bits eines Datenslots wird nicht als solcher erkannt. Dies liegt daran, dass es im *COUNT_SLOTS*-Prozess unterschiedliche Bedingungen für die Rücksetzung des Zählers gibt, die durch `if`- und `else`-Pfade verzweigt sind. Der Zähler *SLOT_BIT_COUNT* setzt sich daher aus einem 5-Bit-Register, einem 5-Bit-Addierer und einem 5-Bit-Komparator zusammen.

Auslastung der FPGA-Ressourcen:

```
Device utilization summary:
---------------------------

Selected Device : 3s400pq208-5

 Number of Slices:               897  out of   3584    25%
 Number of Slice Flip Flops:    1493  out of   7168    20%
 Number of 4 input LUTs:         357  out of   7168     4%
 Number of IOs:                   27
 Number of bonded IOBs:           27  out of    141    19%
 Number of BRAMs:                  3  out of     16    18%
 Number of MULT18X18s:             3  out of     16    18%
 Number of GCLKs:                  2  out of      8    25%
```

Anhand der Zusammenfassung der genutzten FPGA-Ressourcen lässt sich erkennen, dass die Auslastungsgrenzen der einzelnen Komponenten zu maximal 25% erreicht werden. Die 18x18-Bit-Multiplizierer werden als sogenannte *Dedicated Multipliers* [25] synthetisiert, die zu den Ressorucen der Ziel-Hardware gehören.

Tabelle 2 zeigt die maximale Verzögerung t_{Delay} des längsten Datenpfades des Systems und die sich daraus ergebende maximale Taktfrequenz f_{MAX}.

t_{Delay}/ns (f_{max}/MHz)	Delay-Pfad
12,587 (79,446) 9,671 ns Logik, 2,916 ns Verdr.	REG_EMUE → DP_RAM_COEF

Tabelle 2: Maximaler Verzögerungspfad des Systems. $N = 256$, *FSM mit Gray-Code, Spartan 3 FPGA XC3S400-5 mit je 8064 LUTs und D-FFs*

Der längste Datenpfad verläuft vom Ausgang des 36-Bit-Registers *REG_EMUE* über den 18x18-Bit-Multiplizierer und den 19-Bit-Adddierer bis zum Eingang *DI* des Koeffizienten RAMs DP_RAM_COEF (vgl. Abb. 38). Bisher wurde das sequentielle Filter mit einer Frequenz von $50\ MHz$ getaktet. Der doppelte Takt von $100\ MHz$ liesse sich mit diesem Aufbau nicht realisieren, da durch die Verzögerung des beschriebenen Pfades lediglich eine maximale Taktfrequenz von $79,446\ MHz$ möglich ist (vgl. Tabelle 2). Durch das Einfügen eines zusätzlichen Registers hinter dem Multiplizierer könnte der Datenpfad jedoch verkürzt und das adaptive Filter mit der doppelten Taktfrequenz getaktet werden. Dadurch würde die Anzahl der Takte, die dem Filter innerhalb einer Abtastperiode zur Verfügung stehen, erhöht, wodurch eine höhere Filterordnung N realisiert werden könnte.

6.2 Timing Simulation

Die Timing-Simulation wird mit einer Testbench durchgeführt, welche die Taktsignale CLK und BIT_CLK für das System sowie die Daten simuliert, die vom Codec zum FPGA gesendet werden. Es wird ein $50\ MHz$ *CLK*-Signal simuliert, welches als Taktsignal für das adaptive FIR-Filter dient. Für die Taktung des Codec-Interfaces wird das *BIT_CLK*-Signal mit einer Frequenz von $12,288\ MHz$ simuliert (vgl. Code S. 113).

Für das Referenzsignal *XN* und das Primärsignal *DN* werden zwei ROMs in der Testbench instanziiert, die mit einer Periode des jeweiligen Signals gefüllt sind. Es werden die gleichen Signale, mit der Frequenz des Störsignals $f_{st} = 400\ Hz$, wie in der Simulink Simulation verwendet. Die Werte der Signale, die eine komplette Periode beschreiben, werden durch ein Matlab Script in eine COE-Datei geschrieben, die beim Erzeugen eines ROMs durch den Xilinx CORE Generator als Initialisierungs-File verwendet wird. Der Speicher wird in der Testbench durch einen Zähler adressiert, der nach dem Erreichen der letzten Adresse wieder mit der Startadresse beginnt. Dadurch kann das periodische Signal für eine unbegrenzte Simulationszeit erzeugt werden.

Um sowohl Filter als auch Interface zu Beginn in einen definierten Zustand zu versetzen, wird für die Dauer einer Periode des *CLK*-Signals ein Reset erzeugt. Der Reset-Impuls muss diese Länge haben, da die Komponenten des Filters synchron zurückgesetzt werden.

6.2.1 Codec-Interface

Nach dem Reset wird abhängig von einem *BIT_CLK*-Zähler die serielle Datenübertragung vom Audio-Codec zum FPGA simuliert. Dazu wird abhängig vom Zählerstand ein Schieberegister entweder mit einem Abtastwert *XN* des Referenz- oder *DN* des Primärsignals geladen. Die

Daten werden anschließend auf die serielle Eingangsleitung $SDATA\_IN$ des Interfaces geschoben.

Während das *SYNC*-Signal 1 ist, wird der $TAG = 9800_{Hex}$ übertragen. Dieser Wert simuliert das gesetzte *Codec-Ready*-Bit und zwei gesetzte *Valid-Slot-Bits* für die beiden Audio-Slots. In der *SYNC*-Low-Phase werden die Audiodaten in Slot 3 und Slot 4 übertragen.

Das Senden und Empfangen eines TAGs erfolgt parallel, wobei das *TAG_REG_OUT*-Ausgangsregister auf der ersten negativen Taktflanke während des gesetzten *SYNC*-Signals freigegeben wird, damit auf der folgenden positiven Taktflanke das erste Bit des ersten Slots auf die Ausgangsleitung *SDATA_OUT* geschoben werden kann, welches vom Codec auf der fallenden Flanke empfangen wird.

Das Eingangsregister *TAG_REG_IN* wird auf der steigenden *BIT_CLK*-Flanke freigegeben, um mit der folgenden fallenden Flanke das erste empfangene Bit aufzunehmen. Durch die Rückkopplung des *TAG_REG_OUT*-Registerausgangs auf den Registereingang wird der ausgehende 16-Bit Wert direkt in das Register zurückgeschoben, so dass das Register nach Abschluss des Sendens bereits wieder mit dem richtigen Wert geladen ist. Nach 16 Takten ist das Senden und Empfangen des TAGs beendet, das Sende- und Empfangsregister werden nacheinander deaktiviert und das *SYNC*-Signal wird auf '0' gesetzt (vgl. Abb. 52).

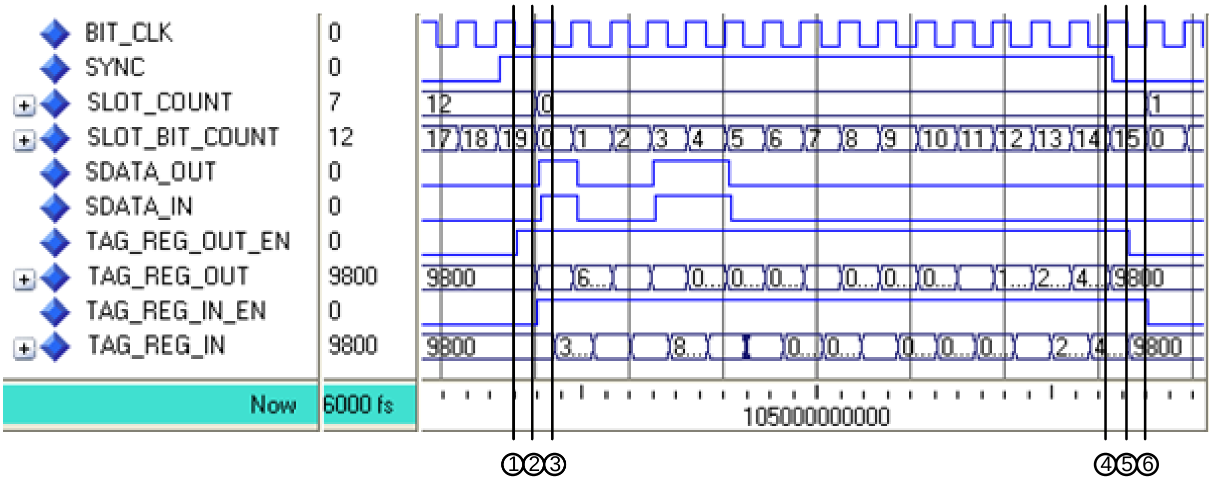

Abb. 52: Timing Simulation des Codec-Interfaces, beim parallelen Senden und Empangen eines TAGs. Freigabe des Senderegisters TAG_REG_OUT (Marker 1); erstes Bit wird auf die Ausgangsleitung SDATA_OUT geschoben und das Empangsregister TAG_REG_IN wird aktiviert (Marker 2); das erste Bit wird von der Eingangsleitung $SDATA\_IN$ gelesen und in das Eingangsregister TAG_REG_IN geschoben (Marker 3); das letzte Bit des ausgehenden TAGs wird gesendet (Marker 4); das letzte Bit des eingehenden TAGs wird empfangen und das Ausgangsregister TAG_REG_OUT wird deaktiviert (Marker 5); das Empfangsregister TAG_REG_IN wird deaktiviert (Marker 6)

Das Senden und Empfangen der Daten in den verbleibenden Slots wird durch die zugehörigen Sende- und Empfangsregister auf die gleiche Weise durchgeführt. Im Normalbetrieb werden zwischen FPGA und Audio-Codec lediglich Audio-Daten ausgetauscht. Dies geschieht in den Slots 3 und 4 eines Datenframes.

Die eingehenden Audiodaten werden in das 40-Bit Empfangsregister *AUDIO_REG_IN* geschoben, wobei als erstes die Daten des rechten Audiokanals und anschließend die Daten des linken Audiokanals empfangen bzw. gesendet werden. Am Ende des vierten Slots ist daher in den oberen 20 Bits ein Sample des rechten Kanals und in den unteren 20 Bits ein Sample des linken Kanals enthalten.

Gesendet wird in dieser Anwendung nur auf dem rechten Audiokanal. Daher werden nur 20

Bits des 40-Bit Senderegisters *AUDIO_REG_OUT* verwendet und in Slot 3 übertragen. Ein abgeschlossener Übertragungsvorgang wird vom Codec-Interface zu Beginn des fünften Slots durch einen *AUDIO_IN_READY*-Impuls signalisiert. Dieser meldet dem adaptiven Filter den Empfang neuer Abtastwerte und startet dadurch die Filterungs- und Adaptionssequenz (vgl. Abb. 53).

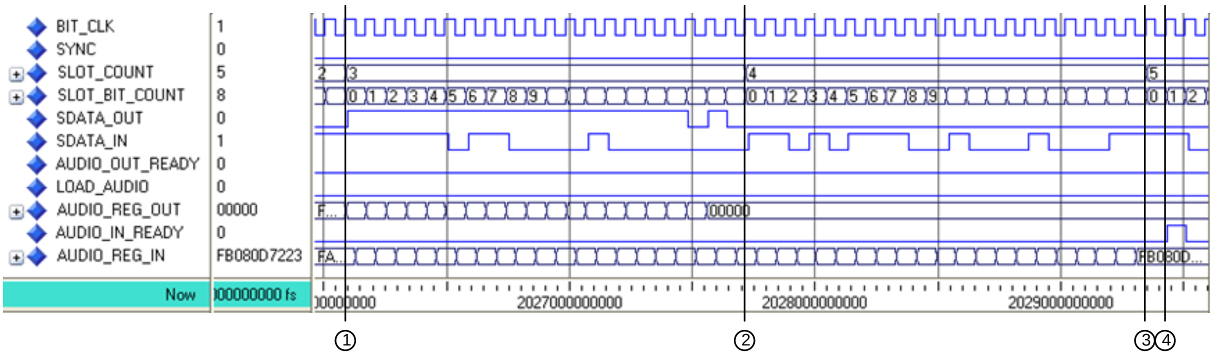

Abb. 53: Timing Simulation des Codec-Interfaces beim Senden und Empfangen von Audiodaten. Beginn der Übertragung eines 20-Bit Abtastwertes des rechten Audiokanals in Slot 3, Empfang über SDATA_IN, Senden über SDATA_OUT (Marker 1); Empfang des zweiten Abtastwertes (linker Kanal) in Slot 4, linker Sendekanal wird nicht genutzt (Marker 2); Ende der Audioübertragung (Marker 3); Erzeugung des Ready-Impulses für das Filter (Marker 4)

Das Senderegister $AUDIO\_REG\_OUT$ für Audiodaten wird beim Empfang eines Ready-Impulses $AUDIO\_OUT\_READY$ vom Filter *BIT_CLK*-synchron mit einem Sample YN des gefilterten Signals geladen. Dazu wird der Impuls $AUDIO\_OUT\_READY$ des Filters, der die Länge eine $CLK = 50\ MHz$ Periode besitzt, auf eine *BIT_CLK*-Periode verlängert (vgl. Abb. 54). Das vom Filter erhaltene Sample YN wird anschließend mit dem nächsten Datenframe zum Codec übertragen.

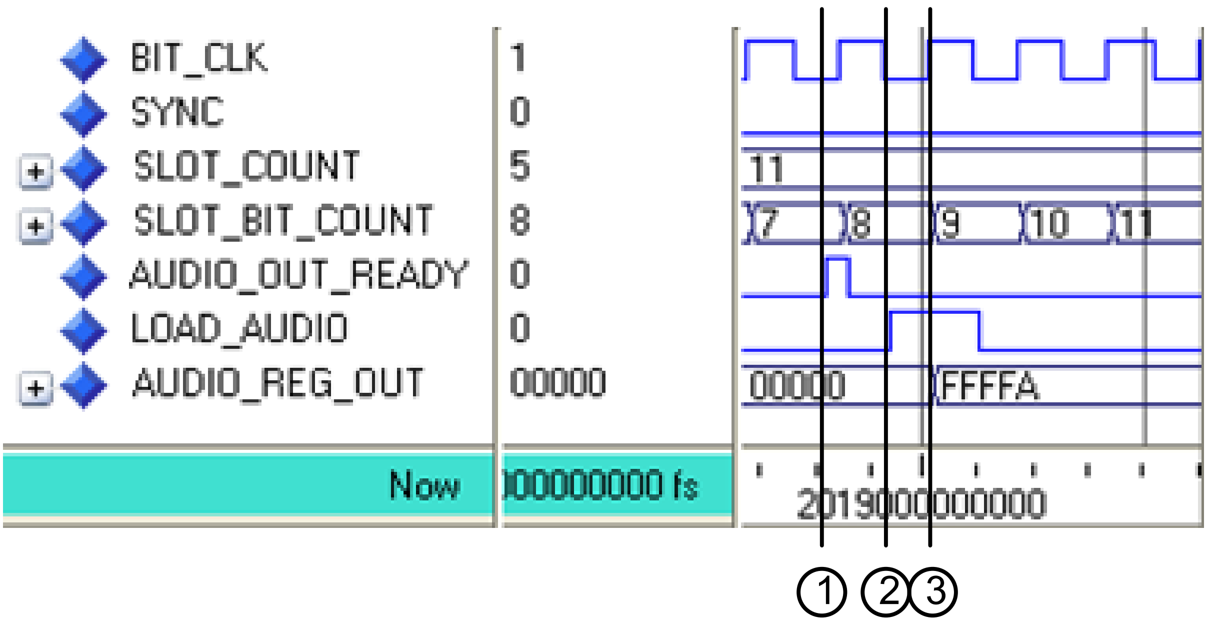

Abb. 54: Timing Simulation des Codec-Interfaces beim Laden des Audio-Ausgangregisters $AUDIO\_REG\_OUT$ mit einem gefilterten Abtastwert YN. Empfang des Ready-Impulses $AUDIO\_OUT\_READY$ vom Filter (Marker 1); Verlängerung des Impulses $AUDIO\_OUT\_READY$ auf eine BIT_CLK-Periode (Marker 2); Laden des Audio-Ausgangregisters $AUDIO\_REG\_OUT$ mit dem vom Filter erhaltenen 20-Bit Sample YN (Marker 3)

6.2.2 Adaptives FIR-Filter

Für die Timing Simulation des Filters müssen die Laufzeiten des Referenz- *XN*, Kompensations- *YN* und Fehlersignals *EN* modelliert werden (vgl. Kap. 4.2). Dies geschieht durch die Verzögerung der Signale mit Hilfe von Registerketten in der Testbench. Das Referenzsignal XN wird um 30, das Kompensationssignal YN um 29 und das Fehlersignal EN um 35 Samples verschoben.

Die Überlagerung von Primärsignal *DN* und Kompensationssignal *YN* wird durch eine Addition modelliert, woraus sich das Fehlersignal *EN* ergibt. Dieses wird zusammen mit dem Referenzsignal *XN* seriell zum Codec-Interface gesendet und dadurch dem Filter zur Verfügung gestellt (vgl. Code S. 113).

Die Timing Simulation des synthetisierten Systems zeigt das Referenzsignal *XN*, welches als Eingangssignal in das Filter eingeht, das Primärsignal *DN* am Überlagerungspunkt, das Kompensationssignal *YN* am Überlagerungspunkt, sowie das aus der Überlagerung resultierende Fehlersignal *EN*. Genau wie in der Simulink Simulation besitzt der Einschwingvorgang des Kompensationssignals *YN* eine Dauer von 15 ms und das Fehlersignal EN wird innerhalb von 30 ms minimiert (vgl. Abb. 55).

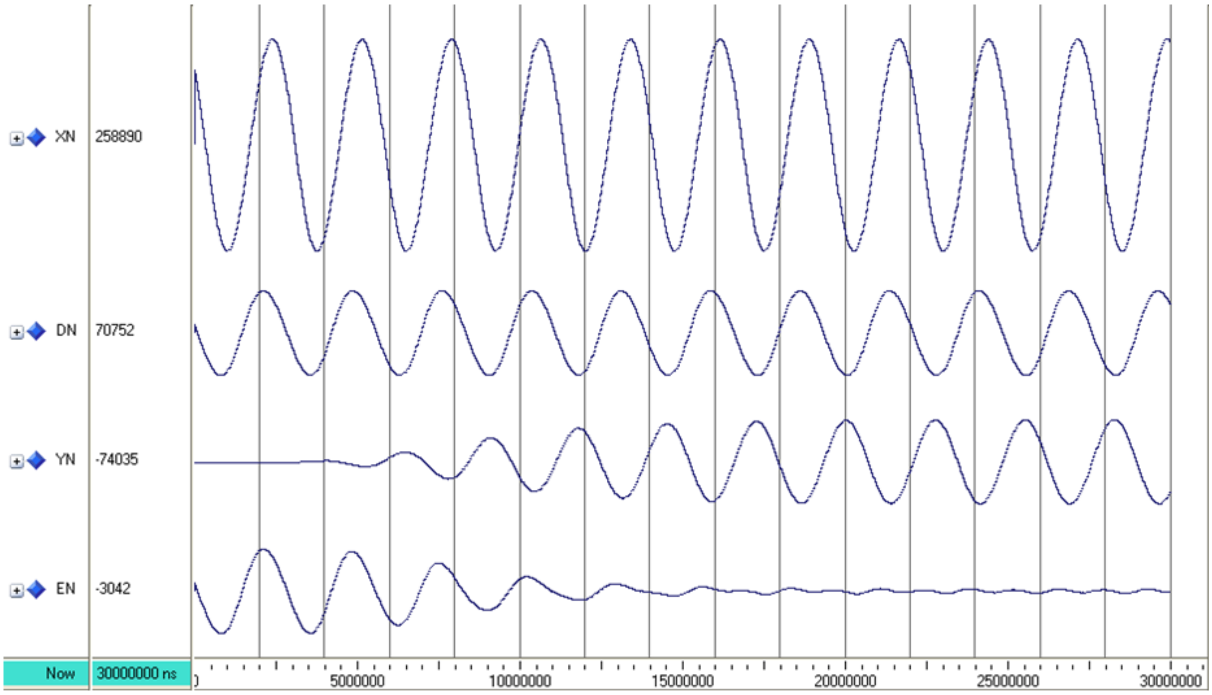

Abb. 55: Timing Simulation des Kompensationsprozesses des adatpiven FIR Filters. $N = 256$, $f_s = 48\ kHz$, $f_{st} = 400\ Hz$, $\mu = 1/512$

In der Timing Simulation wird im Vergleich zur Simulation mit Simulink eine um den Faktor 8 größere Schrittweite MUE verwendet. Dieser Faktor ergibt sich aus der Anzahl der reduzierten Bits des Fehlersignals EN. Bei der Berechnung des Fehlersignals EN aus einem 20-Bit Wert des Primärsignals DN und einem 20-Bit Wert des Kompensationssignals YN, entsteht ein 21-Bit Wert, um einen Überlauf zu verhindern. Dieser wird zur Übertragung an das Codec-Interface, durch das Abschneiden eines Bits, auf 20-Bit gekürzt, was einer Division durch 2 entspricht. Der 20-Bit Wert des Fehlersignals EN wird zur Multiplikation mit dem Schrittweitenfaktor *MUE* auf 18-Bit gekürzt, wodurch eine erneute Division durch 4 entsteht. Daher wird der Abtastwert des Fehlersignals EN schon vor der Multiplikation mit *MUE* durch 8 dividiert und *MUE* kann von $1/4096$ auf $1/512$ vergrößert werden.

Aus den Verzögerungen des Referenzsignals XN im Eingangspfad (z^{-30}) und zum Adaptionsalgorithmus (z^{-64}), der Verzögerung des Kompensationssignals YN im Ausgangspfad (z^{-29}) (vgl. Kap. 4.2 Abb. 18), sowie einer Abtastperiode für den Adaptionsvorgang mit dem ersten verzögerten Abtastwert (z^{-1}) und einer Abtastperiode für den Anschließenden Filterungszyklus (z^{-1}), ergibt sich die Anlaufzeit des Kompensationssignals YN, bis es am Überlagerungspunkt zur Verfügung steht. Dabei errechnen sich die Zeiten aus der Anzahl der verzögerten Samples multipliziert mit der Dauer einer Abtastperiode, woraus sich eine Gesamtzeit von $2604,17\ \mu s$ ergibt (vgl. Abb. 56).

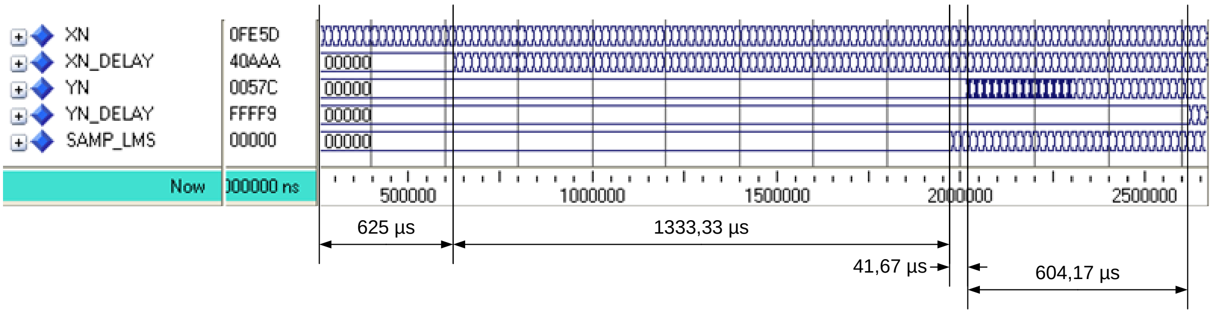

Abb. 56: Timing Simulation der Laufzeiten des Referenz- XN und Kompensationssignals YN. Verzögerung des Referenzsignals XN im Eingangspfad $625\ \mu s$; Verzögerung des Referenzsignals XN zum Adaptionsalgorithmus $1333,33\ \mu s$; Verzögerung bis zum ersten Sample des Kompensationssignals YN am Filterausgang $41,67\ \mu s$; Verzögerung des Kompensationssignal YN im Ausgangspfad $604,17\ \mu s$

Für die Adaption eines Filterkoeffizienten $COEF$ werden zwei Takte benötigt. Mit dem ersten Takt wird jeweils ein Paar aus Koeffizient $COEF$ und Abtastwert $SAMP\_FILT$ bzw. $SAMP\_LMS$ gelesen und gleichzeitig der aktuelle Koeffizient $COEF$ zu einem neuen $COEF_I$ adaptiert (vgl. Abb. 57 Marker 1). Mit der nächsten positiven Taktflanke erfolgt die Inkrementierung der Leseadressen ADR_CR und ADR_S wohingegen die Schreibadresse ADR_CW des Koeffizienten erhalten bleibt, da der zuvor adaptierte Koeffizient $COEF\_I$ an diese Adresse zurückgeschrieben wird. Gleichzeitig wird das Akkumulationsregister Y aktualisiert (vgl. Abb. 57 Marker 2). Mit dem darauffolgenden Takt wird erneut gelesen und die Schreibadresse ADR_CW inkrementiert 57 Marker 3).

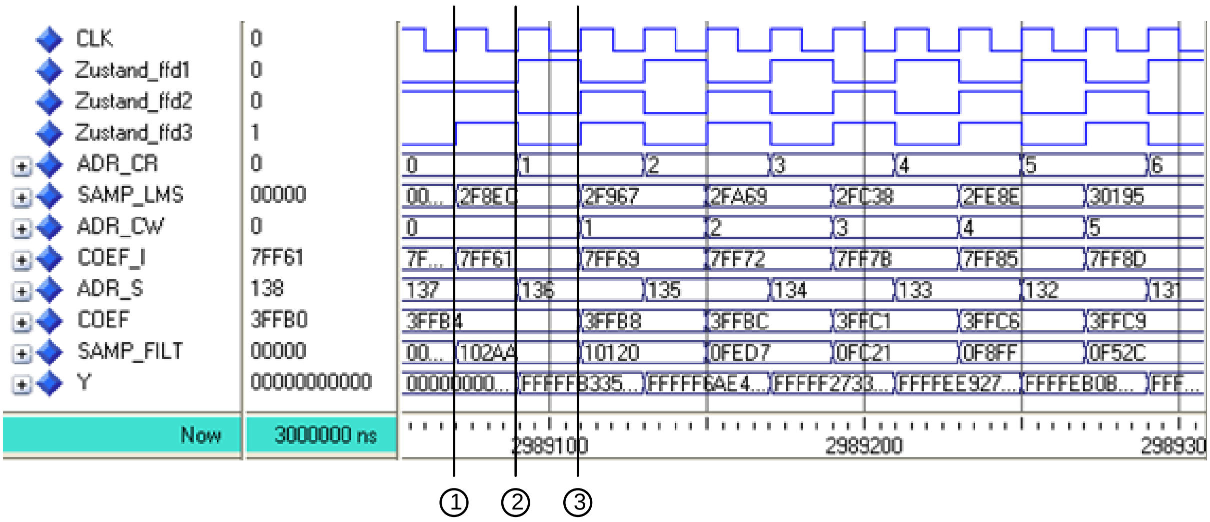

Abb. 57: Timing Simulation der Adressierung des Dual-Port-RAMs. $f_{CLK} = 50\ MHz$, $Zustand = ffd1\ ffd2\ ffd3\ MUL\_ACC = 011\ WRITE\_C = 100$

Die Anzahl M der für einen Zyklus des adaptiven Filters benötigten Takte ergibt sich daher aus

$$M = 2N + 5 \tag{69}$$

mit 5 Takten für die zusätzlichen Zustandsübergänge der FSM. Aus den insgesamt 878 Takten, die dem Filter zur Verfügung stehen (vgl. Kap. 5.4), ergibt sich eine maximale Filterordnung $N_{max} = 436$.

7 Messtechnische Analyse des Hardware-Aufbaus

Für die in diesem Kapitel durchgeführten Echtzeittests des entwickelten Systems, wird der in Kapitel 4 beschriebene Aufbau verwendet, wobei das Kunstkopf-Messsystem *HMS II.5* von *HEAD acoustics* zur Erfassung der tatsächlichen Kompensation des Störsignals $s(t)$ im Ohr eines Menschen eingesetzt wird. Dazu erfolgt die Positionierung des Fehlermikrofons und des Kopfhörers im bzw. am Ohr des Kunstkopfes.

In den folgenden Abschnitten werden zunächst Ergebnisse der Tests mit monofrequenten Störsignalen dargestellt und erläutert. Anschließend erfolgt die Analysierung von Testergebnissen mit breitbandigen Störsignalen. Um die Dämpfung eines Störsignals $s(t)$ messen zu können, wird das Primärsignal $d(t)$, welches das Störsignal $s(t)$ enthält zuerst durch das Mikrofon des Kunstkopfes aufgenommen, während das ANC-System deaktiviert ist. Danach wird das System eingeschaltet und das Signal $d(t) \equiv e(t)$ an der gleichen Position unter identischen Bedingungen erneut aufgenommen.

Da die Aufnahme beider Signale über das Mikrofon im Kunstkopf erfolgt, werden sie im weiteren Verlauf einheitlich als Primärsignal $d(t)$ bezeichnet, wobei das ANC-System das anteilig enthaltene Störsignal $s(t)$ kompensiert. Unterschieden wird dann zwischen Signal $d(t)$ mit deaktivierter oder $d(t) \equiv e(t)$ mit aktivierter Kompensation.

Die Messungen werden mit Sinussignalen ab $200\ Hz$ durchgeführt, da im RAM des Filters mit der gewählten Ordnung von $N = 256$ und bei einer Abtastfrequenz von $f_s = 48\ kHz$ eine vollständige Periode des Signals erst ab einer Frequenz von $f = 187\ Hz$ gespeichert werden kann. Daher kann bei niedrigeren Frequenzen eine Kompensation nicht gewährleistet werden.

Die mit Sinustönen durchgeführten Tests zeigen, dass sich das System bei einer Frequenz von $4000\ Hz$ an eine Obergrenze annähert, da bei dieser Frequenz die erreichte Dämpfung mit $-10\ dB$ um die Hälfte geringer ist als bei den niedrigeren Frequenzen. Bei einer Frequenz von $5000\ Hz$ wird bereits keine Dämpfung sondern ein Verstärkung des Primärignals $d(t) \equiv e(t)$ erzeugt. Dieses Verhalten steht in direktem Zusammenhang mit dem Frequenzgang des Sekundärpfades (vgl. Kap. 4.1), der zwischen 4 und 5 kHz deutlich abfällt (vgl. Anh. B Abb. 79), wodurch das System ein Kompensationssignal $y(t)$ mit einer unangemessen hohen Leistung erzeugt.

Bei Tests mit breitbandigen Signalen, die mehrere Frequenzen unterschiedlicher Amplitude enthalten, wird eine Dämpfung des Gesamtsignals von $-8\ dB$ erreicht. Als Testsignale werden ein Ventilations- und ein Motorengeräusch verwendet. Aufgrund der negativen Einflüsse des Sekundärpfades, der Filterordnung von $N = 256$ und der konstanten Schrittweite μ fällt die Reduzierung dieser Störsignale $s(t)$, im Vergleich zu periodischen Signalen mit einer konstanten Frequenz und Amplitude, geringer aus.

Die Sinusstörsignale $s(t)$ werden am PC generiert und über eine Kombination bestehend aus einem *Yamaha AX-397* Verstärker und einem *Canton Plus XL.2* Lautsprecher wiedergegeben.

7.1 Monofrequente Störsignale

Störsignal mit der Frequenz $f = 200\ Hz$:

Die zeitliche Darstellung der beim Empfänger ankommenden Signale $d(t)$ und $d(t) \equiv e(t)$ zeigt, dass nach dem, durch das Aufheben des Resets verursachten, Peak bei $t = 1\ s$ die Kompensation des Störsignals $s(t)$, wie zuvor in der Simulation gezeigt, im Millisekunden-Bereich erfolgt (vgl. Abb. 58). Nach weiteren $1,3\ s$ ist die Kompensation stabil und der Pegel des Primärsignals $d(t) \equiv e(t)$ mit aktivierter Kompensation um den Faktor 10 abgeschwächt.

In der Darstellung der beiden Signale im Frequenzbereich, lässt sich der genaue Wert der Abschwächung ermitteln, der bei der Frequenz $f = 200\ Hz$ $-21 dB$ beträgt (vgl. Abb. 59).

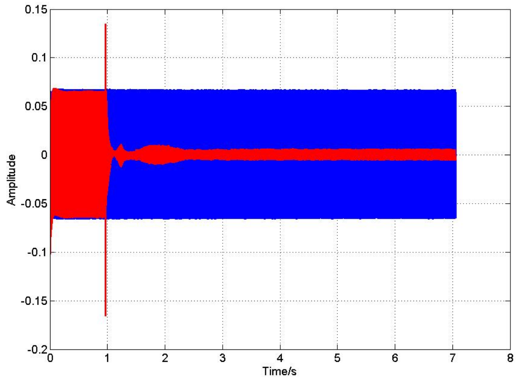

Abb. 58: $f_{st} = 200\ Hz$*: Zeitlicher Verlauf der Signale. Primärsignal* $d(t)$ *ohne Kompensation (blau); Primärsignal* $d(t) \equiv e(t)$ *mit Kompensation (rot)*

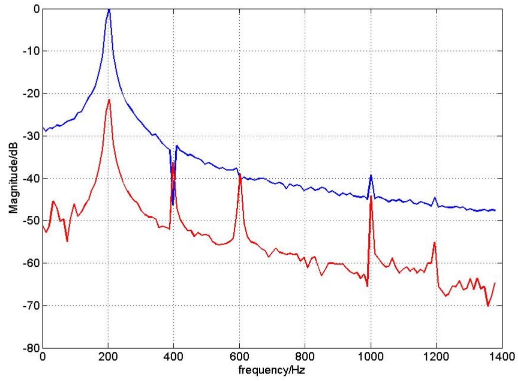

Abb. 59: $f_{st} = 200\ Hz$*: Signale im Frequenzbereich. Primärsignal* $d(t)$ *ohne Kompensation (blau); Primärsignal* $d(t) \equiv e(t)$ *mit Kompensation (rot)*

Der Schrittweitenfaktor μ ist durch einen Dip-Schalter auf dem FPGA-Board in Zweierpotenzen von $1/2^1$ bis $1/2^{16}$ einstellbar. Für diesen Test wurde zunächst der kleinste Faktor gewählt und anschließend schrittweise erhöht, bis das in diesem Testfall größtmögliche $\mu = 1/2^{12}$ erreicht wurde, was zu den dargestellten Testergebnissen geführt hat. Ein größere Schrittweite μ führte zu einem Übersteuern des Kompensationssignals $y(t)$, verursacht durch die Verletzung des Konvergenzkriteriums (vgl. Kap. 3.3.2 Gl. 45).

Störsignal mit der Frequenz $f = 400\ Hz$:

Bei diesem Test ist die Anlaufphase der Kompensation nach dem Reset mit $250\ ms$ länger als bei dem $200\ Hz$ Störsignal. Dies liegt daran, dass der gleiche Schrittweitenfaktor $\mu = 1/4096$ gewählt wurde, die Gesamtintensität und damit die mittlere Eingangsleistung des Störsignals $s(t)$ jedoch geringer ist als bei dem vorangegangenen Test. Eine stabile Kompensation wird innerhalb einer Sekunde erreicht (vgl. Abb. 60).

Die Frequenzbereichdarstellung zeigt wie die zeitliche Darstellung eine deutliche Abschwächung des Primärsignals $d(t) \equiv e(t)$ bei der Frequenz $f = 400\ Hz$ um $-23\ dB$.

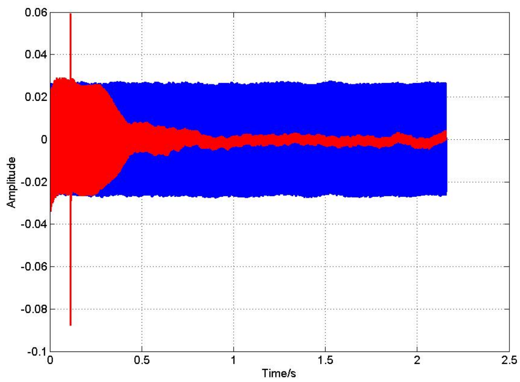

Abb. 60: $f_{st} = 400\ Hz$*: Zeitlicher Verlauf der Signale. Primärsignal* $d(t)$ *ohne Kompensation (blau); Primärsignal* $d(t) \equiv e(t)$ *mit Kompensation (rot)*

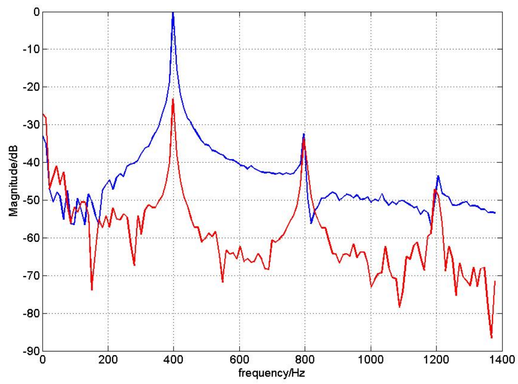

Abb. 61: $f_{st} = 400\ Hz$*: Signale im Frequenzbereich. Primärsignal* $d(t)$ *ohne Kompensation (blau); Primärsignal* $d(t) \equiv e(t)$ *mit Kompensation (rot)*

Störsignal mit der Frequenz $f = 1500\ Hz$:

Für diesen Test wurde die Schrittweite auf $\mu = 1/1024$ vergrößert. Der zeitliche Verlauf zeigt eine kontinuierliche Minimierung des Primärsignals $d(t) \equiv e(t)$, die nach $1,1\ s$ stationär ist (vgl. Abb. 62). Auch bei dieser Frequenz wird eine Dämpfung der Störfrequenz von $-23\ dB$ erreicht (vgl. Abb. 63).

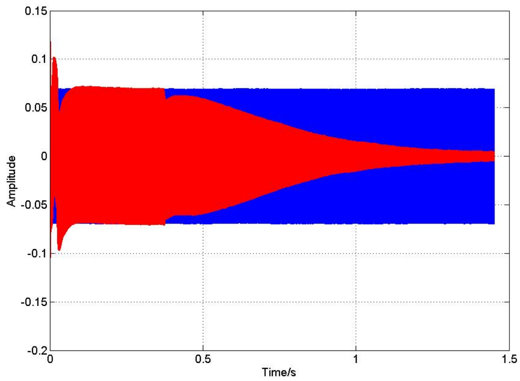

Abb. 62: $f_{st} = 1500\ Hz$: Zeitlicher Verlauf der Signale. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot)

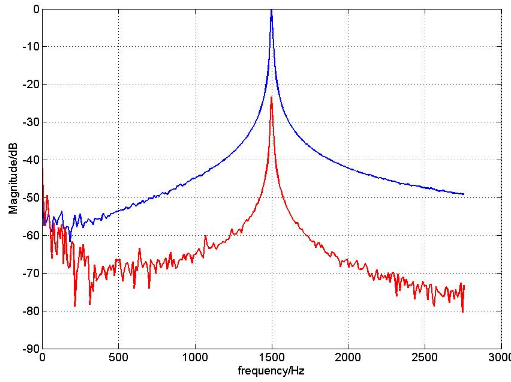

Abb. 63: $f_{st} = 1500\ Hz$: Signale im Frequenzbereich. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot)

Störsignal mit der Frequenz $f = 4000\ Hz$:

Bei einer Frequenz von $4000\ Hz$ erfährt das Fehlersignal $e(t)$, welches für die Adaption der Filterkoeffizienten verwendet wird, eine Abschwächung durch den Sekundärpfad (vgl. Anh. B Abb. 79). Da das System dieses bereits gedämpfte Fehlersignal $e(t)$ minimiert, erhält das Kompensationssignal $y(t)$ eine niedrige Amplitude, die das tatsächliche Restsignal $d(t) \equiv e(t)$ lediglich auf die Hälfte, der zuvor bei niedrigeren Frequenzen erreichten Leistung, reduziert (vgl. Abb. 64).

Durch diesen Einfluss des Sekundärpfades, wird bei diesem Test eine Dämpfung des Primärsignals $d(t) \equiv e(t)$ um $-11\ dB$ erreicht (vgl. Abb. 65).

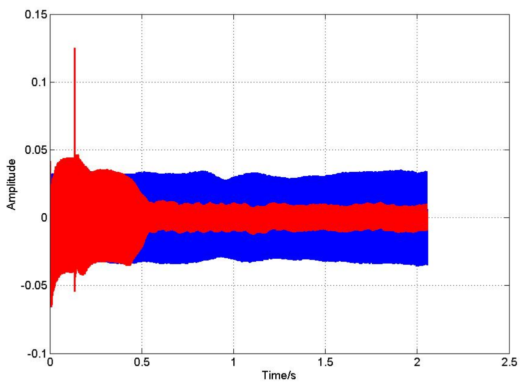

Abb. 64: $f_{st} = 4000\ Hz$: Zeitlicher Verlauf der Signale. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot)

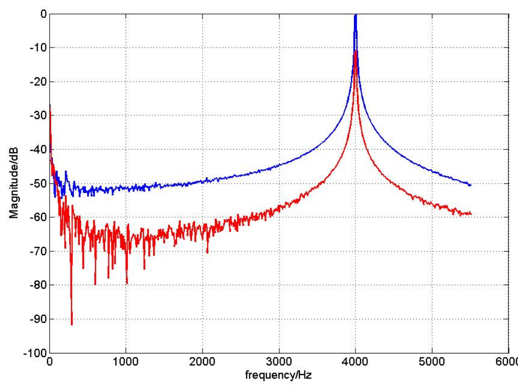

Abb. 65: $f_{st} = 4000\ Hz$: Signale im Frequenzbereich. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot)

Störsignal mit der Frequenz $f = 5000\ Hz$:

Der Frequenzgang des Sekundärpfades fällt bei $5\ kHz$ weiter ab, wodurch das übertragene Fehlersignal $e(t)$ um mehr als $-10\ dB$ gedämpft wird. Da im Sekundärpfad sowohl das Störsignal $s(t)$ als auch das rückgekoppelte Kompensationssignal $y(t)$ diese Dämpfung erfahren, erzeugt das System in diesem Fall ein Kompensationssignal $y(t)$, dessen Amplitude größer ist, als die des tatsächlichen Störsignals $s(t)$, wodurch das Primärsignal $d(t) \equiv e(t)$ nicht reduziert sondern verstärkt wird (vgl. Abb. 66).

Die durch das Filtersignal $y(n)$ verursachte Verstärkung beträgt in diesem Test $5\ dB$ (vgl. Abb. 67).

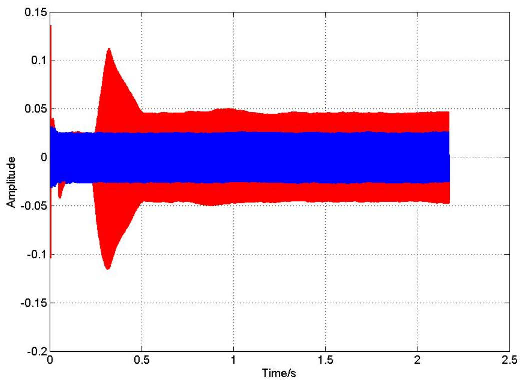

Abb. 66: $f_{st} = 5000\ Hz$: Zeitlicher Verlauf der Signale. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot)

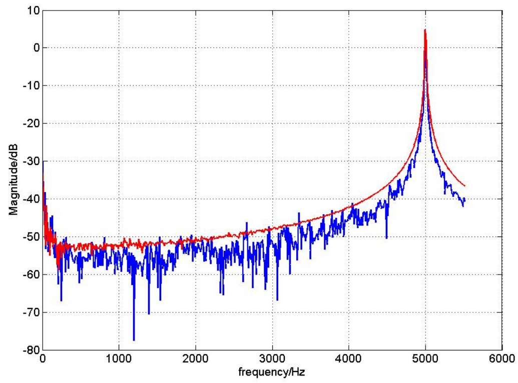

Abb. 67: $f_{st} = 5000\ Hz$: Signale im Frequenzbereich. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot)

7.2 Ventilations-Störsignal

Das für diesen Test verwendete Ventilationsgeräusch stammt von einem *Tektronix TDS544A* Farboszilloskop. Das Geräusch wurde aufgezeichnet und ebenfalls über den *Yamaha* Verstärker und den *Canton* Lautsprecher wiedergegeben. Dieses Störsignal $s(t)$ enthält mehrere unterschiedliche Frequenzanteile mit unterschiedlichen Amplituden. Den Hauptanteil des Signals machen Frequenzen im Bereich von $190 - 250\ Hz$ aus (vgl. Abb. 69).

Der Hauptanteil des Primärsignals $d(t) \equiv e(t)$ mit aktivierter Kompensation liegt mit einer Frequenz von $f = 130\ Hz$ in einem Bereich, der aufgrund der Filterordnung $N = 256$ nicht kompensiert werden kann. Die leistungsstärksten Frequenzen des Störsignals $s(t)$ werden jedoch reduziert, wodurch das Primäsignal $d(t) \equiv e(t)$ eine Dämpfung von $-8\ dB$ erfährt und damit eine deutlich niedrigere Amplitude als das Primärsignal $d(t)$ mit deaktivierter Kompensation besitzt (vgl. Abb. 69).

Die im Vergleich zu Sinussignalen geringere Dämpfung in anderen Frequenzbereichen ist auf den nichtlinearen Frequenzgang und die ebenfalls nicht lineare Gruppenlaufzeit des Sekundärpfades (vgl. Anh. B) zurückzuführen, da die einzelnen Signalanteile in diesem Pfad eine zusätzliche Dämpfung erhalten oder unterschiedlich verzögert werden, wodurch das ANC-System nicht optimal reagiert.

Bei diesem Test wurde ein Schrittweitenfaktor $\mu = 1/128$ gewählt, damit das adaptive Filter eine Annäherung an die optimalen Koeffizienten, trotz der Frequenzvariationen des Störsignals $s(t)$, erreicht. Dies führt zunächst zu einer Übersteuerung des Kompensationssignals $y(t)$ nach dem Aufheben des Resets, auf die das Filter jedoch reagieren. Da das Störsignal $s(t)$ eine ge-

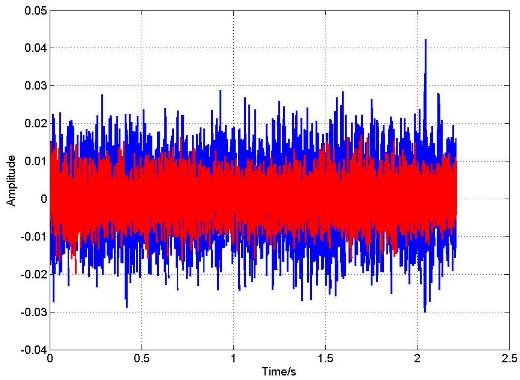

Abb. 68: Ventilations-Störsignal: Zeitlicher Verlauf der Signale. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot)

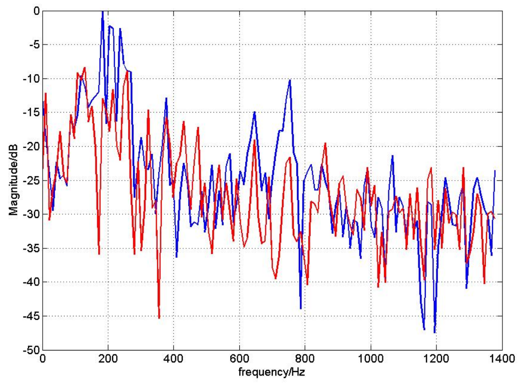

Abb. 69: Ventilations-Störsignal: Signale im Frequenzbereich. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot)

ringe Eingangsleistung besitzt, ensteht eine Adaptionszeit von $7\ s$ (vgl Abb. 70). Ein kleinerer Schrittweitenfaktor μ führte dazu, dass das System langsamer reagierte und dadurch keine Kompensation erreicht wurde.

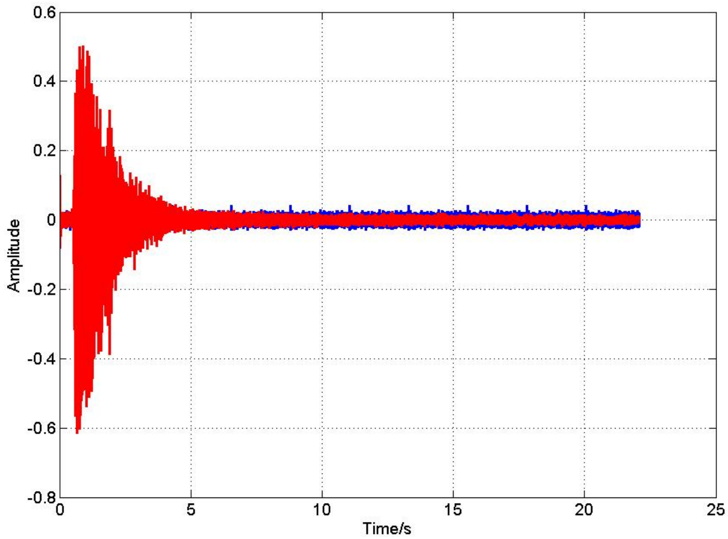

Abb. 70: Anlauf der Kompensation des Ventilationssignals nach einem System-Reset. $\mu = 1/128$

7.3 Motor-Störsignal

Für diesen Test wurde das Motorengeräusch eines *BMW 325i* mit einem *Eisenmann* Sportschalldämpfer bei geringer Drehzahl verwendet. Die Frequenzanteile mit der höchsten Leistung liegen bei $215\ Hz$, $441\ Hz$ und $474\ Hz$. Bei diesen Frequenzen wird durch das ANC-System eine Dämpfungen von $-18\ dB$, $-22\ dB$ und $-31\ dB$ erreicht (vgl. Abb. 72).

Der Hauptanteil des Primärsignal $d(t) \equiv e(t)$ mit aktivierter Kompensation liegt bei der Frequenz $f = 323\ Hz$. Da dieser Frequenzanteil im Primärsignal $d(t)$ ohne Kompensation mit einer geringen Leistung vorhanden ist, die durch den Sekundärpfad zusätzlich reduziert wird, entsteht bei dieser Frequenz eine Verstärkung von $4\ dB$. Insgesamt wird das Primärsignal

$d(t) \equiv e(t)$ jedoch um $-8\ dB$ gedämpft und weisst daher eine niedrigere Amplitude als ohne Kompensation auf (vgl. Abb. 71).

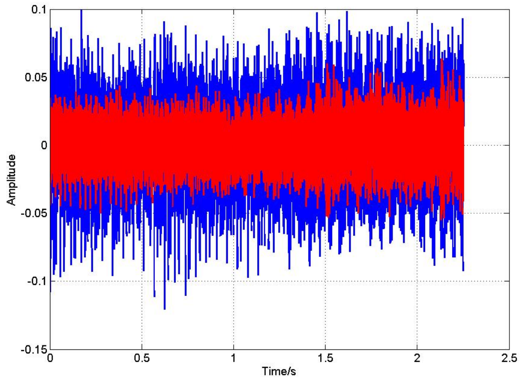

Abb. 71: Motor-Störsignal: Zeitlicher Verlauf der Signale. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot)

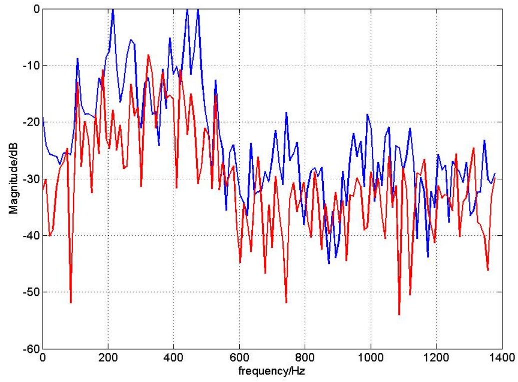

Abb. 72: Motor-Störsignal: Signale im Frequenzbereich. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot)

Bei diesem Test wurde ebenfalls ein Schrittweitenfaktor $\mu = 1/128$ gewählt. Die Übersteuerung des Kompensationssignals $y(t)$ wird hier innerhalb einer Sekunde korrigiert, da durch eine höhere mittlere Eingansleistung des Störsignals $s(t)$, im Vergleich zum Ventilationsgeräusch, die Adaptionszeit auf $1\ s$ verkürzt wird (vgl. Abb. 73).

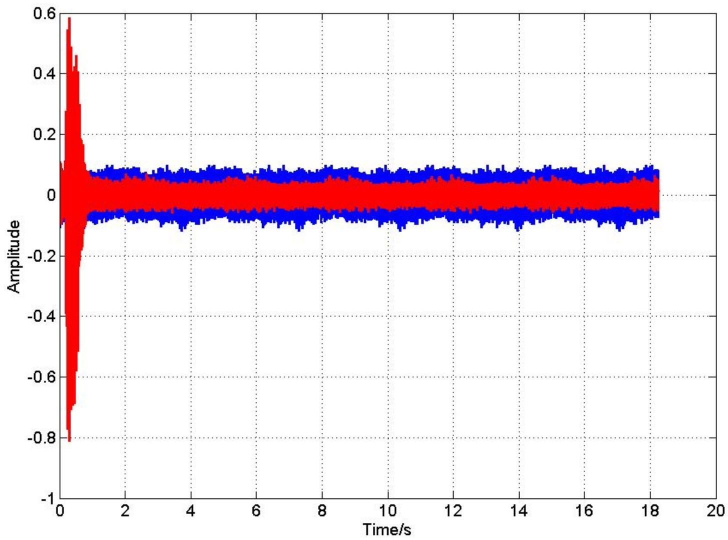

Abb. 73: Anlauf der Kompensation des Motorsignals nach einem System-Reset. $\mu = 1/128$

8 Erweiterungen des Systems

Das entwickelte System stellt einen Prototypen zur aktiven Geräuschkompensation dar. Durch Erweiterungen in der Implementierung des adaptiven Filters kann der Einsatzbereich des Systems erweitert werden:

Durchzuführende Erweiterung	Ziel	Maßnahmen
Adaptive Schrittweite μ	Kompensation von Signalen mit zeitvarianter mittlerer Eingangsleistung	Berechnung des Schrittweitenfaktors μ nach einem neuen Abtastwert $x(n)$ anhand einer Schätzung der mittleren Eingangsleistung des Referenzsignals $x(n)$ $\Rightarrow$ Normierter LMS-Algorithmus (vgl. Kap. 3.3.3)
Übertragungsfunktion $S(z)$ des Sekundärpfades durch ein zusätzliches Filter nachbilden	Störeinflüsse des Sekundärpfades beseitigen $\Rightarrow$ kompensierbaren Frequenzbereich erweitern	Identifikation der Übertragungsstrecke $S(z)$ des Sekundärpfades $\Rightarrow$ Filter zur Nachbildung der Übertragungsstrecke $S(z)$ entwerfen und implementieren
Höhere Filterordnung N	Kompensation von niederfrequenten Signalen und verbesserte Nachbildung der Übertragungsstrecke durch das adaptive Filter $\Rightarrow$ exaktere Kompensation	Speicher des adaptiven Filters vergrößern $\Rightarrow$ höherer Rechenaufwand $\Rightarrow$ Taktfrequenz erhöhen

Tabelle 3: Erweiterungsmöglichkeiten in der Implementierung des adaptiven Filters auf dem FPGA

Wie in Kapitel 3.3.2 erläutert, ist die Konvergenz des LMS-Adaptionsalgorithmus vom Schrittweitenfaktor μ abhängig. Wird die Schrittweite μ zu groß gewählt, ist der Algorithmus instabil, wodurch ein inkorrektes Kompensationssignal $y(n)$ erzeugt wird. Eine zu klein gewählte Schrittweite μ führt dazu, dass die Adaption der Filterkoeffizienten zu langsam ist und dadurch ebenfalls kein korrektes Kompensationssignal $y(n)$ erzeugt wird.

Die Obergrenze μ_{max} für die Schrittweite μ wird durch die mittlere Eingangsleistung des Referenzsignals $x(n)$ bestimmt (vgl. Kap. 3.3.2), die sich mit der Zeit verändern kann. Um in einem solchen Fall zu verhindern, dass die Schrittweite μ die Obergrenze μ_{max} überschreitet, oder sich zu weit von ihr entfernt, wird die Schrittweite μ ständig an das Referenzsignal $x(n)$ angepasst.

Dazu wird mit jedem neuen Abtastwert $x(n)$ eine Schätzung der mittleren Eingangsleistung berechnet

$$\mathbf{x}^t(n)\mathbf{x}(n) = \mathbf{x}^t(n-1)\mathbf{x}(n-1) + x^2(n) - x^2(n-N+1) \tag{70}$$

Der Schrittweitenfakor μ berechnet sich dann durch

$$\mu(n) = \frac{\beta}{\gamma + \mathbf{x}^t(n)\mathbf{x}(n)} \qquad 0 < \beta < 2,\ 0 < \gamma < 1 \tag{71}$$

Für die Schätzung der mittleren Eingangsleistung werden zwei zusätzliche Multiplizierer und zur Quadrierung des aktuellsten $x(n)$ sowie des ältesten Abtastwertes $x(n-N+1)$ ein Addierer und ein Subtrahierer in der FPGA-Implementierung benötigt (vgl. Abb. 74). Da die Umsetzung von Gleichung 71 eine Division erfordert, und diese in den meisten Fällen nicht synthetisierbar ist [19] [23], weicht der Realisierungansatz an dieser Stelle von der mathematischen Lösung ab, indem die Schrittweite μ anhand der geschätzen mittleren Eingangsleistung durch einen Dekoder bestimmt wird.

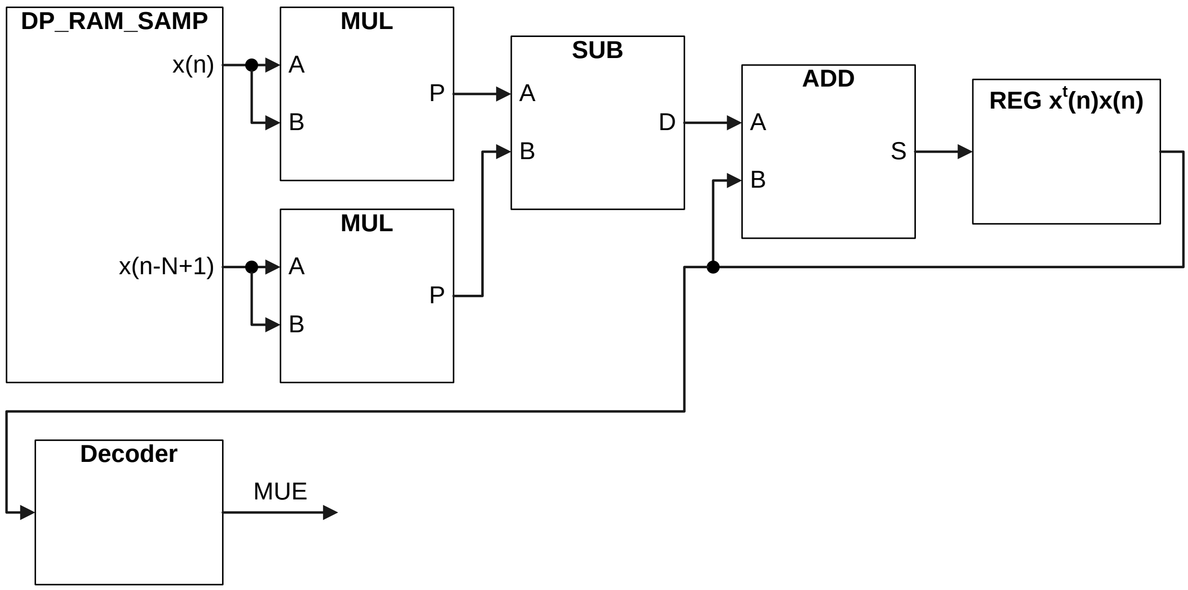

Abb. 74: HW-Realisierungsansatz zur Adaption des Schrittweitenfaktors μ

Der Sekundärpfad des Systems setzt sich aus dem Audio-Codec, dem Kopfhörer und dem In-ohrmikrofon zusammen. Dieser Signalpfad besitzt eine Übertragungsfunktion $S(z)$, die sich störend auf die Funktionalität des Systems auswirkt. Verantwortlich dafür sind zwei Eigenschaften des Pfades:

- Signallaufzeit
- Frequenzgang

In dem entwickelten ANC-System ist bisher nur die Laufzeit berücksichtigt worden, indem eine entsprechende Verzögerung des Referenzsignals $x(n)$ in den Eingangspfad des Adaptionsalgorithmus integriert wurde (vgl. Kap. 4 ff.). Um die störenden Eigenschaften des Frequenzgangs auszugleichen, muss anstelle der Verzögerung ein zusätzliches Filter in den Eingangspfad gesetzt werden. Dieses Filter hat die Aufgabe die gesamte Übertragungsfunktion $S(z)$ des Sekundärpfades nachzubilden.

Da die Übertragungsstrecke $S(z)$ des Sekundärpfades konstant ist, wird an dieser Stelle ein nicht adaptives Filter eingesetzt. Für den Entwurf eines FIR-Filters sind zwei Lösungsansätze denkbar:

- **System Identifikation** (vgl. Kap. 2.1): Nachbildung des Sekundärpfades $S(z)$ durch ein adaptives Filter. Das Filter erhält das gleiche Eingangssignal $x(n)$ wie der Sekundärpfad und aus der Subtraktion des Filter-Ausgangssignals $\widehat{y}(n)$ vom Ausgangssignal des Sekundräpfades $y(n)$ wird das Fehlersignal $e(n)$ gebildet (vgl. Abb. 75). Die Koeffizienten werden dahingehend adaptiert, dass gilt $\widehat{y}(n) = y(n)$, wodurch das Fehlersignal $e(n)$ minimiert wird. Die auf diese Weise ermittelten Koeffizienten können dann für das FIR-Filter verwendet werden, welches im Eingangspfad des Adaptionsalgorithmus auf dem FPGA implementiert ist.
- **Messung des Frequenzgangs**: Dies ist eine klassische Entwurfsmethode für FIR-Filter.

Durch die inverse Fourier-Transformation des Frequenzgangs erhält man die Impulsantwort des idealen Filters. Da diese für eine Realisierung zu lang ist, wird sie durch die Multiplikation mit einer geeigneten Fensterfunktion auf die gewünschte Filterordnung begrenzt und liefert die Koeffizienten des Filters [22].

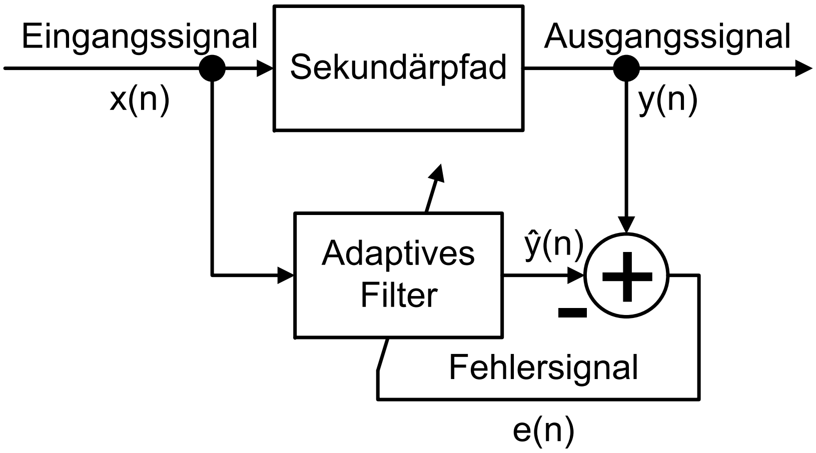

Abb. 75: System Identifikation zur Ermittlung der Koeffizienten eines Filters für die Nachbildung des Sekundärpfades

Um die frequenzabhängige Gruppenlaufzeit des Sekundärpfades nachzubilden muss jedoch statt eines FIR-Filters mit konstanter Gruppenlaufzeit ein IIR-Filter verwendet werden. Verfahren zum Entwurf eines solchen Filters finden sich in [22] und [5]. Strategien zur Modellierung eines IIR-Filters in VHDL sind [19] zu entnehmen.

Die Ordnung $N = 256$ des implementierten adaptiven FIR-Filters stellt eine Untergrenze für den kompensierbaren Frequenzbereich dar. Die korrekte Nachbildung eines Signals durch das Filter erfordert, dass eine Periode dieses Signals durch $N + 1$ Abtastwerte beschrieben werden kann. Die Frequenzuntergrenze errechnet sich daher durch

$$f_{min} = \frac{f_s}{N+1} \tag{72}$$

Bei einer Abtastfrequenz $f_s = 48\ kHz$ und der Ordnung $N = 256$ liegt diese Untergrenze bei $f_{min} = 186\ Hz$. Wie aus Gleichung 72 hervorgeht, kann diese Grenze durch das Erhöhen der Filterordnung N oder durch die Reduzierung der Abtastfrequenz f_s heruntergesetzt werden. Da die Abtastrate f_s durch den Audio-Codec bestimmt wird, bleibt als realisierbare Option eine höhere Filterordnung N. Zusätzlich lassen sich durch diese Maßnahme komplexere Übertragungsstrecken durch das Filter nachbilden.

Bei einer Taktfrequenz von $f_{clk} = 50\ Mhz$ beträgt die maximale Filterordnung des implementierten Systems $N_{max} = 437$. Um höhere Ordnungen umzusetzen, muss daher entweder die Ausnutzung der zur Verfügung stehenden Takte effizienter gestaltet werden, z.B. durch das Zurückschreiben der adaptierten Koeffizienten in den Dual-Port-RAM auf der negativen Taktflanke, oder die Taktung des Filters mit einer höheren Frequenz f_{clk} erfolgen. Die Verdopplung der Taktfrequenz auf $f_{clk} = 100\ MHz$ erfordert die Verkürzung des längsten Datenpfades durch das Einfügen eines zusätzlichen Register (vgl. Kap. 6.1), was eine Anpassung des Automaten zur Steuerung des Datenpfades nachsichzieht.

Es ist jedoch zu beachten, dass eine Abhängigkeit des mittleren quadratischen Fehlers (MSE) J, der durch das adaptive Filter minimiert wird, von der Filterordnung N besteht. Der MSE im eingeschwungenen Zustand ergibt sich aus

$$J = J_{min} + J_{ex} \tag{73}$$

wobei J_{min} den minimal erreichbaren MSE darstellt, der mit zunehmender Ordnung N abnimmt, und J_{ex} für die Ungenauigkeit des Adaptionsalgorithmus steht, die mit steigender Anzahl an Filterkoeffizienten zunimmt, da mit jedem zusätzlichen Koeffizienten ein weiterer verrauschter Parameter hinzukommt [16]. Die zu wählende Filterordnung darf daher nicht zu groß ausfallen und ist exakt den gewünschten Anforderungen des Systems anzupassen.

9 Zusammenfassung

Das in dieser Masterarbeit entwickelte ANC-System basiert auf einem adaptiven Filter, welches sich aus einem FIR-Filter mit maximaler Ordnung $N_{max} = 436$ und dem LMS-Adaptionsalgorithmus zusammensetzt. Wie die Messergebnisse zeigen, ist das System in der Lage, Frequenzen zwischen $200\ Hz$ und $4000\ Hz$, sowohl in monofrequenten als auch in breitbandigen Signalen, zu kompensieren. Der Adaptionsalgorithmus arbeitet in dieser Version mit einer konstanten Schrittweite μ, wodurch der Bereich der kompensierbaren Störgeräusche auf Signale mit gleichbleibender Leistung beschränkt ist. Die Geräuschkompensation wurde für einen Kanal demonstiert. Mit einer weiteren FPGA-Audio-Codec Plattform lässt sich das ANC-System für eine Stereo-Kompensation erweitern.

Um ein vollständiges Simulationsmodell zu erstellen, wurden die Signalpfade des Referenzsignals $x(n)$, des Primärsignals $d(n)$, des Kompensationssignals $y(n)$ und des Fehlersignals $e(n)$ analysiert. Dazu wurden mit Hilfe eines Signalgenerators und eines Oszilloskops Laufzeitmessungen für die einzelnen Pfade durchgeführt. Die Laufzeiten wurden in der Modellierung berücksichtigt, um ein realistisches Modell des Systems mit einer Nachbildung des Sekundärpfades zu erhalten. Anhand dieses Modells wurde der Aufbau des realen Systems und die Implementierung des adaptiven Filters durchgeführt.

Bei der Entwicklung und Implementierung der parallelen Hardware auf dem FPGA wurden allgemein gültige Design-Methoden angewandt. Die Entwicklung des adaptiven Filters auf dem FPGA erfolgte nach dem Top-Down Entwurfsstil, bei dem zunächst das Blockschaltbild der obersten Ebene durch die Top-Entity beschrieben wird und anschließend die Präzisierung der untergeordneten Komponenten erfolgt [19]. Der Aufbau und die Struktur des Codec-Interfaces und des adaptiven Filter entsprechen dem eines Synchron-Systems mit jeweils einem Steuer- und einem Datenpfad. Dabei werden alle Komponenten eines Systems über das selbe Signal getaktet und sämtliche Vorgänge im Datenpfad durch den Steuerpfad kontrolliert [23]. Diese Entwurfsmethodiken haben zu einer strukturierten und effizienten Nutzung der FPGA-Ressourcen geführt.

Die durchgeführten Funktionstests haben gezeigt, dass das System in einem Frequenzbereich von $200\ Hz - 4\ kHz$ zuverlässig funktioniert. Tests bei Frequenzen von 200, 400 und $1500\ Hz$ zeigten, dass eine Abschwächung von mehr als $-20\ dB$ erreicht wird. Zum unteren Frequenzbereich sind dem implementierten System durch die Filterordnung von $N = 256$ Grenzen gesetzt. Im oberen Frequenzbereich wird eine erfolgreiche Kompensation durch die Frequenzgang- und Gruppenlaufzeit-Eigenschaften des Sekundärpfades verhindert, da diese als konstant angesehen werden und lediglich eine konstante Gruppenlaufzeit des Sekundärpfades berücksichtigt wird. Ein Ventilations- und ein Motorgeräusch, in denen mehrere Frequenz- und zusätzliche Rauschanteile enthalten sind, konnten um bis zu $-8\ dB$ gedämpft werden.

Die geringe Auslastung der verfügbaren FPGA-Kapazitäten (vgl. Kap. 6.1) ermöglicht die Erweiterung der Parallelität im adaptiven Filter. Dadurch kann die Signalverarbeitung beschleunigt werden, was den Spielraum für Systemerweiterungen, in Bezug auf eine höhere Filterordnung N und einen flexibleren LMS-Adaptionsalgorithmus, vergrößert.

Abbildungsverzeichnis

1 ANC System mit den physikalisch/analogen Signalen: Störsignal $s(t)$, Nutzsignal $n(t)$, Primärsignal $d(t)$, Referenzsignal $x(t)$, Kompensationssignal $y(t)$ und Fehlersignal $e(t)$, sowie den digitalisierten Signalen $x(n)$, $y(n)$ $e(n)$ und einem FPGA als digitale Signalverarbeitungseinheit 10
2 System Identifikation: Modellierung des unbekannten Systems $G_s(z)$ durch das adaptive Filter $G_a(z)$ [10] 14
3 System Inversion: Modellierung der inversen Übertragungsfunktion des unbekannten Systems $G_s(z)$ [10] 14
4 Signal Prädiktion: Vorhersage des Eingangssignals $x(n)$ [10] 15
5 Multisensor Störsignal Kompensation: Kompensation eines Störsignals im Primärsignal [10] 16
6 Auslöschung einer Sinusschwingung 16
7 Adaptive Feedback System 17
8 Adaptive Feedforward System 17
9 Adaptives Filter in einem Adaptive Feedforward ANC-System 19
10 Adaptives FIR-Filter in Direktform mit der Ordnung $N-1$ 20
11 Fehlerfläche bei N = 2 Koeffizienten 22
12 Rekursionsverlauf des Newton-, SDA- und LMS-Algorithmus auf den Höhenlinien der Fehlerfläche 25
13 Konvergenzverhalten des SDA-Algorithmus bei einer Schrittweite $\mu = \frac{2}{\lambda_{max}}$ . . 27
14 Aufbau des Feedforward ANC-Systems mit allen verwendeten HW-Komponenten: Lautsprecher, Kondensatormikrofon, Audio-Interface, Audio-Codec, FPGA, Kopfhörer und Inohr-Mikrofon. Signalbezeichnungen mit dem Index t stellen analoge Signale dar, während der Index n ein digitales Signal bezeichnet. 29
15 Adaptives Filter mit einem zusätzlichen Filter $\widehat{S}(z)$ im Eingangspfad des Referenzsignals $x(n)$ zum Adaptionsalgorithmus, der die Übertragungsstrecke $S(z)$ des Sekundärpfades modelliert 30
16 ANC-System-Aufbau mit den gemessenen Signallaufzeiten: Störsignal $s(t)$ vom Ausgang des Signalgenerators, über Lautsprecher und Kondensatormikrofon, bis zum Ausgang des Audio-Interfaces ($t1$); Gruppenlaufzeit des Audio-Codecs ($t2$); Signalkreislaufzeit des Kompensationssignals $y(t)$ vom Eingang des Audio-Codecs, zum Ausgang des Codecs, über den Kopfhörer, das Inohr-Mikrofon und das Audio-Interface, zurück zum Eingang des Audio-Codecs ($t3$); Störsignal $s(t)$ vom Ausgang des Signalgenerators, über Lautsprecher und Inohr-Mikrofon, bis zum Ausgang des Audio-Interfaces ($t4$) 32
17 Simulink-Modell des Feedforward ANC-Systems mit den am Audio-Interface aufgenommenen Primär- und Referenzsignalen, sowie den ermittelten Signallaufzeiten 33
18 Simulink-Modell mit der Verzögerung des Referenzsignals im Eingangspfad des Adaptionsalgorithmus 34
19 Referenzsignal $x(n)$, $f = 400\ Hz$ 35
20 Primärsignal $d(n)$ aufgenommen am Inohr-Mikrofon und Kompensationssignal $y(n)$ erzeugt vom adaptiven Filter 36
21 Fehlersignal $e(n)$ nach der Überlagerung von $d(n)$ mit $y(n)$ 36
22 Adaptierte Filterkoeffizienten $w_0(n) - w_{256}(n)$ des FIR-Filters 36
23 Primärsignal $d(n)$ und exponentiell wachsendes Kompensationssignal $y(n)$ . . 37

24 Exponentiell wachsendes Fehlersignal $e(n)$ nach der Überlagerung von $d(n)$ mit $y(n)$ . . . 37
25 Exponentiell wachsende Filterkoeffizienten $w_0(n) - w_{256}(n)$ . . . 37
26 Primärsignal $d(n)$ und Kompensationssignal $y(n)$ nach der Simulation mit verzögertem $x(n)$ . . . 38
27 Fehlersignal $e(n)$ nach der Überlagerung von $d(n)$ mit $y(n)$ und verzögertem $x(n)$ . . . 38
28 Adaptierte Filterkoeffizienten $w_0(n) - w_{256}(n)$ nach der Simulation mit verzögertem $x(n)$ . . . 38
29 Register Transfer Level-Modell [19] . . . 40
30 Partitionierung eines digitalen Systems in Daten- und Steuerpfad. Die beiden Teilsysteme kommunizieren über Steuer- bzw. Statussignale. Zusätzlich wird das Verhalten des Steuerwerks von Statussignalen externer Komponenten beeinflusst und es werden Steuersignale für externe Komponenten erzeugt [19] . . 41
31 FPGA Systemaufbau bestehend aus Codec-Interface und adaptivem FIR-Filter. $CLK = 50\ MHz$, $BIT\_CLK = 12,288\ MHz$, Abtastfrequenz $f_s = 48\ kHz$ 44
32 Hierarchischer Baum der Komponenten des gesamten VHDL-System-Modells 44
33 Bidirektionaler Audio-Frame [17] . . . 45
34 Komponenten des Codec Interface. FPGA-Schnittstelle zum Audio-Codec. Datenpfad: *SHIFT_DATA_IN*, *LOAD_OUT_REGISTERS* und *SHIFT_DATA_OUT*. Steuerpfad: *CONTROL_IN_REGISTERS* und *CONTROL_OUT_REGISTERS*. . . . 46
35 VHDL-Simulation des *SLOT_COUNT*-Zählers und des *SLOT_BIT_COUNT*-Zählers. Beginn eines Datenframes mit $SYNC = 1$. $f_{BIT\_CLK} = 12,288\ MHz$ . . . 47
36 VHDL-Simualtion des *INIT_SEQ*-Zählers. Fünf Datenframes zur Konfiguration des Audio-Codecs. Bei $INIT\_SEQ = 6$ ist die Anlaufphase beendet. . . . 47
37 VHDL-Simulation des Codec-Interfaces. Aktivierung der Ein- und Ausgangsregister (Marker 1); Paralleles Laden der Ausgangsregister *ADR_REG_OUT* und *DATA_REG_OUT* (Marker 2); Erzeugung des Ready-Impulses *AUDIO_IN_READY* (Marker 3) . . . 48
39 `entity`-Baum des VHDL-Filter-Modells . . . 49
38 Sequentielles adaptives FIR-Filter *LMS_FIR* mit den Datenpfadkomponenten MAC-Einheit *MAC_FIR*, LMS-Adaptionseinheit *MAC_LMS* und Abtastwerte-RAM *RAM_S_257_18* sowie dem Steuerpfad, der aus dem Zustandsautomaten *FSM_LMS_FIR* und den zugehörigen Steuersignalen besteht. Taktsignal *CLK* mit $f_{CLK} = 50\ MHz$; $N = 256$; (vgl. Code S. 105) . . . 50
40 Multiplizierer-Akkumulatoreinheit: 18x18-Bit Multiplizierer *MUL*, 42-Bit Addierer *ADD* mit $\lceil ld(L) = 9$ Guard-Bits, 42-Bit Akkutmulationsregister *REG_Y*, Sättigungsmodul *SAT* (vgl. Code S. 110). $f_{CLK} = 50\ MHz$ . . . 53
41 VHDL-Simulation der MAC-Einheit. Abschluss der Akkumulation (Marker 1); Sättigung des Ergebnisses und Bildung des Zweierkomplements (Marker 2); sowie Rücksetzen des Akku-Registers (Marker 3) . . . 53
42 Beispielsequenz für die Kombination der Koeffizienten mit den im Ringpuffer sukzessive gespeicherten Abtastwerten. Filterordnung $N = 3$ [19] . . . 54

43 VHDL-Simulation der RAM-Einheit beim Start der Adressierungssequenz. Adresszählerstände nach $L = 257$ Schreibvorgängen, Datum der letzten Schreib-Adresse $ADR\_S = 256$ erscheint am Ausgang(Marker 1); Dekrementierung der Sample-Adresse ADR_S und Inkrementierung der Koeffizienten-Adresse ADR_CR (Marker 2); Datum der Adresse $ADR\_S = 255$ erscheint am Ausgang (Marker 3) 54
44 VHDL-Simulation der RAM-Einheit beim Abschluss der Adressierungssequenz. Adressierung des letzten Faktorenpaares $x(0)$ und c_{256} (Marker 1); Enable-Signal für den Zähler des Koeffizientenspeichers wird gesetzt, um den Überlauf auf 0 durchzuführen (Marker 2); Sample-Adresse $A(0)$ wird beibehalten zur Speicherung des nächsten Samples, Überlauf der Koeffizienten-Adresse auf 0 (Marker 3) 55
45 RAM-Einheit für die Abtastsamples des Referenzsignals bestehend aus zwei 257x18-Bit Block-RAMs *RAM_SAMP_FILT* und *RAM_SAMP_LMS*, zwei Multiplexern *MUX* zum Umschalten des Eingangsdatums bei einem Reset, einem Adresszähler *ADR_CNT_SAMP* und einem z^{-64} Verzögerungsglied, das sich aus einer 64x18-Bit Registerkette X_DELAY zusammensetzt. $f_{CLK} = 50\,MHz$ 55
46 VHDL-Simulation des Schiebevorgangs in der Verzögerungskette *X_DELAY*. Enable-Signal *EN_X_SHIFT* zum Schieben wird gesetzt (Marker 1); Die Abtastwerte in *X_DELAY(0)* und *X_DELAY(1)* werden um jeweils eine Position weitergeschoben; Ein neues Sample wird in das Register *X_DELAY(0)* geschrieben 56
47 LMS-Adaptionseinheit *MAC_LMS*: zwei 18x18-Bit Multiplizierer *MUL*, 18-Bit Addierer *ADD* mit $n = 2$ Summanden und $\lceil 3,32log(n) = 1$ Guard-Bit, 36-Bit Multiplikationsregister *REG_EMUE*, 257x19-Bit Dual-Port-Block-RAM *DP_RAM_COEF*, Multiplexer *MUX* zum Umschalten des Eingangsdatums bei einem Reset, zwei Adresszähler *ADRW_CNT_COEF*für die Schreibadresse und *ADRR_CNT_COEF* für die Leseadresse. $f_{CLK} = 50\,MHz$ 56
48 VHDL-Simulation der *MAC_LMS*-Einheit beim Adressieren des Dual-Port-RAM *DP_RAM_COEF* 58
49 Zustandsdiagramm der Sequenzsteuerung des adaptiven FIR-Filters mit MAC-Einheit und LMS-Adaption; Mealy-FSM mit Moore-Ausgängen [19] 59
50 VHDL-Simulation der *FSM_LMS_FIR* beim Durchlaufen der Pipelinestufen. Aktualisierung der Lese-Adressen *ADR_S* und *ADR_CR* (Marker 1); An den Speicherausgängen *SAMP_FILT*, *SAMP_LMS* und *COEF* erscheinen neue Werte (Marker 2); Aktualisierung der Schreibadresse *ADR_CW* (Marker 3); Akkumulation und Speicherung in *REG_Y* (Marker 4); Speicherung des adaptierten Koeffizienten $COEF\_I$ an der Adresse *ADR_CW* (Marker 5) 61
51 VHDL-Simulation der Anlaufphase nach einem System-Reset. Das Reset-Signal wechselt auf den Low-Pegel und die Anlaufphase startet mit dem Adressierungszyklus und der ersten Abtastperiode, markiert durch das *SYNC*-Signal (Marker 1); der Adressierungszyklus ist abgeschlossen und der Automat geht in den Zustand *IDLE* über (Marker 2); die erste von fünf Abtastperioden der Anlaufphase des Codec-Interfaces ist beendet (Marker 3) 61

52 Timing Simulation des Codec-Interfaces, beim parallelen Senden und Empangen eines TAGs. Freigabe des Senderegisters TAG_REG_OUT (Marker 1); erstes Bit wird auf die Ausgangsleitung *SDATA_OUT* geschoben und das Empangsregister TAG_REG_IN wird aktiviert (Marker 2); das erste Bit wird von der Eingangsleitung $SDATA\_IN$ gelesen und in das Eingangsregister TAG_REG_IN geschoben (Marker 3); das letzte Bit des ausgehenden TAGs wird gesendet (Marker 4); das letzte Bit des eingehenden TAGs wird empfangen und das Ausgangsregister TAG_REG_OUT wird deaktiviert (Marker 5); das Empfangsregister TAG_REG_IN wird deaktiviert (Marker 6) 65
53 Timing Simulation des Codec-Interfaces beim Senden und Empfangen von Audiodaten. Beginn der Übertragung eines 20-Bit Abtastwertes des rechten Audiokanals in Slot 3, Empfang über *SDATA_IN*, Senden über *SDATA_OUT* (Marker 1); Empfang des zweiten Abtastwertes (linker Kanal) in Slot 4, linker Sendekanal wird nicht genutzt (Marker 2); Ende der Audioübertragung (Marker 3); Erzeugung des Ready-Impulses für das Filter (Marker 4) 66
54 Timing Simulation des Codec-Interfaces beim Laden des Audio-Ausgangregisters $AUDIO\_REG\_OUT$ mit einem gefilterten Abtastwert YN. Empfang des Ready-Impulses $AUDIO\_OUT\_READY$ vom Filter (Marker 1); Verlängerung des Impulses $AUDIO\_OUT\_READY$ auf eine *BIT_CLK*-Periode (Marker 2); Laden des Audio-Ausgangregisters $AUDIO\_REG\_OUT$ mit dem vom Filter erhaltenen 20-Bit Sample YN (Marker 3) 66
55 Timing Simulation des Kompensationsprozesses des adatpiven FIR Filters. $N = 256$, $f_s = 48\ kHz$, $f_{st} = 400\ Hz$, $\mu = 1/512$ 67
56 Timing Simulation der Laufzeiten des Referenz- XN und Kompensationssignals YN. Verzögerung des Referenzsignals XN im Eingangspfad $625\ \mu s$; Verzögerung des Referenzsignals XN zum Adaptionsalgorithmus $1333,33\ \mu s$; Verzögerung bis zum ersten Sample des Kompensationssignals YN am Filterausgang $41,67\ \mu s$; Verzögerung des Kompensationssignal YN im Ausgangspfad $604,17\ \mu s$ 68
57 Timing Simulation der Adressierung des Dual-Port-RAMs. $f_{CLK} = 50\ MHz$, $Zustand = ffd1\ ffd2\ ffd3\ MUL\_ACC = 011\ WRITE\_C = 100$ 68
58 $f_{st} = 200\ Hz$: Zeitlicher Verlauf der Signale. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot) 71
59 $f_{st} = 200\ Hz$: Signale im Frequenzbereich. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot) 71
60 $f_{st} = 400\ Hz$: Zeitlicher Verlauf der Signale. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot) 71
61 $f_{st} = 400\ Hz$: Signale im Frequenzbereich. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot) 71
62 $f_{st} = 1500\ Hz$: Zeitlicher Verlauf der Signale. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot) 72
63 $f_{st} = 1500\ Hz$: Signale im Frequenzbereich. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot) 72
64 $f_{st} = 4000\ Hz$: Zeitlicher Verlauf der Signale. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot) 72
65 $f_{st} = 4000\ Hz$: Signale im Frequenzbereich. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot) 72
66 $f_{st} = 5000\ Hz$: Zeitlicher Verlauf der Signale. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot) 73

67 $f_{st} = 5000\ Hz$: Signale im Frequenzbereich. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot) 73
68 Ventilations-Störsignal: Zeitlicher Verlauf der Signale. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot) 74
69 Ventilations-Störsignal: Signale im Frequenzbereich. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot) 74
70 Anlauf der Kompensation des Ventilationssignals nach einem System-Reset. $\mu = 1/128$ 74
71 Motor-Störsignal: Zeitlicher Verlauf der Signale. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot) 75
72 Motor-Störsignal: Signale im Frequenzbereich. Primärsignal $d(t)$ ohne Kompensation (blau); Primärsignal $d(t) \equiv e(t)$ mit Kompensation (rot) 75
73 Anlauf der Kompensation des Motorsignals nach einem System-Reset. $\mu = 1/128$ 75
74 HW-Realisierungsansatz zur Adaption des Schrittweitenfaktors μ 77
75 System Identifikation zur Ermittlung der Koeffizienten eines Filters für die Nachbildung des Sekundärpfades 78
76 Frequenzgangmessung des Audio-Codec. Abtastfrequenz $f_s = 48\ kHz$ 91
77 Phasengangmessung des Audio-Codecs 91
78 Gruppenlaufzeitmessung des Audio-Codecs. $t_{gd} = 1,2ms$ 92
79 Frequenzgangmessung des Sekundärpfades, bestehend aus Audio-Codec, Kopfhörer und Inohr-Mikrofon. Ab einer Frequenz von $5\ kHz$ deutliche Abschwächung des Signals um bis zu $-22\ dB$ 92
80 Phasengangmessung des Sekundärpfades, bestehend aus Audio-Codec, Kopfhörer und Inohr-Mikrofon 93
81 Gruppenlaufzeitmessung des Sekundärpfades, bestehend aus Audio-Codec, Kopfhörer und Inohr-Mikrofon. Laufzeit bewegt sich im unteren und oberen Frequenzbereich zwischen $1,3\ ms$ und $1,5\ ms$. Bei einer Frequenz von $6\ kHz$ springt die Laufzeit auf $2,8\ ms$ und bei $16kHz$ sinkt sie kurzzeitig auf $0,6\ ms$ ab. 93
82 Messergebnis $\Delta t = 296\ \mu s = t1$ 94
83 Messergebnis $\Delta t = 1230\ \mu s = t2$ 94
84 Messergebnis $\Delta t = 1330\ \mu s = t3$ 95
85 Messergebnis $\Delta t = 3200\ \mu s = t4$ 95
86 Spartan-3 LC Entwicklungsplattform [14] 116
87 Spartan-3 FPGA Architektur [25] 117
88 Audio/Video Modul [2] 118
89 Das AC ’97 Interface als Verbindung zwischen UCB 1400 und FPGA [17] . . . 119
90 Bidirektionaler Audio-Frame [17] 119
91 AC Link Audio Ausgangsframe vom FPGA zum Codec [17] 120
92 Start eines Audio Ausgangsframes vom FPGA zum Codec [17] 121
93 AC Link Audio Eingangsframe vom Codec zum FPGA [17] 122
94 Start eines Audio Eingangsframes vom Codec zum FPGA [17] 122
95 SHIFT_DATA_IN Prozess schiebt anhand der Enable-Signale die seriellen Bits von der *SDATA_IN*-Leitung in die Eingangsregister. Das Schieben erfolgt auf der negativen Taktflanke (vgl. Code S. 102 Z. 280-307). 124
96 SHIFT_DATA_OUT Prozess enthält einen Multiplexer, der über die Enable-Signale der Ausgangsregister gesteuert wird, um den jeweils aktiven Ausgang eines Registers mit der seriellen Ausgangsleitung *SDATA_OUT* zu verbinden (vgl. Code S. 103 Z. 314-341). 125

98 GEN_SYNC Prozess bestehend aus einem Framzähler, der die Bits eines Frames zählt, und einem Framezähler, der die ersten vier Frames für das Anlaufverhalten des Codec-Interfaces zählt. (vgl. Code S.100) 126
97 LOAD_OUT_REGISTERS Prozess enthält einen Decoder, der anhand des Anlaufzählers *INIT_SEQ* die Ausgangsschieberegister mit den zum Anlaufverhalten passenden Daten lädt. Nach der Anlaufphase werden anhand der Enable-Signale die Schiebevorgänge der Ausgangsregister aktiviert. Über einen Pulsverlängerer [23] wird der Ready-Impuls des adaptiven Filters zum Ladeimpuls des Audio-Ausgangsregisters verlängert (vgl. Code S. 104 Z. 369-433). 127
99 COUNT_SLOTS Prozess bestehend aus einem Zähler für die Slots eines Frames *SLOT_COUNT*, einem Zähler für die Bits eines Slots *SLOT_BIT_COUNT* und einem Zähler für die Frames der Anlaufphase des Interfaces *INIT_SEQ* (vgl. Code S. 102 Z. 236-276) 129
100 CONTROL_IN_REGISTERS Prozess bestehend aus einem Decoder, der anhand der Slot-Zählerstände die Eingangsschieberegister aktiviert und einen Ready-Impuls *AUDIO_IN_READY* für das adaptive Filter erzeugt. Die Schiebefreigabe erfolgt auf der positiven Taktflanke (vgl. Code S. 100 Z. 105-163). . 131
101 CONTROL_OUT_REGISTERS Prozess bestehend aus einem Decoder, der anhand der Slot-Zählerstände die Ausgangsschieberegister aktiviert. Die Schiebefreigabe erfolgt auf der negativen Taktflanke (vgl. Code S. 101 Z. 168-232). . . 133
102 Timing Simulation des Codec-Interfaces während der Anlaufphase. Adressregister *ADR_REG_OUT* und Datenregister *DATA_REG_OUT* werden parallel geladen (Marker 1); serielle Übertragung der Registerdaten zum Codec (Marker 2); Das Senden der Konfigurationdaten ist abgeschlossen, das TAG-Register wird für die Audioübertragung neu geladen, Adress- und Datenregister werden auf ’0’ gesetzt (Marker 3); Senden des TAGs zur Audioübertragung an den Codec (Marker 4); Anlaufphase ist beendet (Marker 5) 134
103 Spartan-3 LC Entwicklungsplattform schematisch [14] 136
104 P160 Prototyp Modul [13] 136
105 P160 Prototyp Modul Block Diagramm [13] 137
106 Audio/Video Modul Block Diagramm [2] 137
107 UCB 1400 Block Diagramm [17] 138

Tabellenverzeichnis

1 *Klassifikation von Anwendungen der adaptiven Filterung* [10] 13
2 Maximaler Verzögerungspfad des Systems. $N = 256$, FSM mit Gray-Code, Spartan 3 FPGA XC3S400-5 mit je 8064 LUTs und D-FFs 64
3 Erweiterungsmöglichkeiten in der Implementierung des adaptiven Filters auf dem FPGA 76
4 Signalnamen mit Beschreibung, Äquivalenten Signalen und erstmaligem Auftreten 90
5 J3 Pin-Belegung 139
6 J4 Pin-Belegung 140
7 J5 Pin-Belegung 140
8 J6 Pin-Belegung 141
9 Pin-Verbindung zwischen P160- und AV-Board 141
10 *Reset Register* 142
11 *Reset Register* Bit-Beschreibung 142
12 *Master Volume Register* 142
13 *Master Volume Register* Bit-Beschreibung 142
14 *Record Select Register* 142
15 *Record Select Register* Bit-Beschreibung 142
16 *Record Gain Register* 143
17 *Record Gain Register* Bit-Beschreibung 143
18 *Feature Control/Status Register 1* 143
19 *Feature Control/Status Register 1* Bit-Beschreibung 143

Literatur

[1] A. Angermann, M. Beuschel, M. Rau, and U. Wohlfarth. *Matlab - Simulink - Stateflow : Grundlagen, Toolboxen, Beispiele*. Oldenbourg Wissenschaftsverlag GmbH, 2002. ISBN 3-486-25979-2.

[2] Avnet Inc. *Audio/Video Module Product Brief*, 2004.

[3] Y. Gong, Y. Song, and S. Liu. Performance analysis of the unconstrained fxlms algorithm for active noise control. Technical report, IEEE, 2003.

[4] T. Hör. *Entwicklung eines adaptiven FIR-Filters auf FPGA-Basisfür Audio-Anwendungen*. 2003.

[5] E. C. Ifeachor and B. W. Jervis. *Digital Signal Processing - A Practical Approach*. Pearson Education Limited, 2. edition, 2002. ISBN 0-201-59619-9.

[6] F. Ihlenburg. *Finite element analysis of acoustic scattering*. Springer-Verlag, 1998. ISBN 0-387-98319-8.

[7] B. Jansen. Active noise cancellation systems. Technical report, I-Car, Januar 2007.

[8] C. D. Kestell and C. H. Hansen. An overview of active noise control. *Safety Science Monitor*, 3(5):1–7, Feb. 1998.

[9] X. Kong and S. M. Kuo. Study of causality constraint on feedforward active noise control systems. Technical report, IEEE, 1999.

[10] D. G. Manolakis, V. K. Ingle, and S. M. Kogon. *Statistical and Adaptive Signal Processing*. McGraw-Hill, 2000. ISBN 0-07-040051-2.

[11] The MathWorks Inc. *Matlab Help*, 1984-2007.

[12] J. Matthies. Implementierung eines sequentiellen fir-filters auf einer fpga-audio-codec plattform. Technical report, Hochschule für Angewandte Wissenschaften Hamburg, 2007.

[13] Memec Design. *P160 Prototype Module User's Guide*, November 2002. Rev. 1.1.

[14] Memec Design. *Memec Spartan-3 LC User's Guide*, Juni 2004. Rev. 2.0.

[15] U. Meyer-Baese. *Digital Signal Processing with Field Programmable Gate Arrays*. Springer-Verlag, 2. edition, 2004. ISBN 3-540-21119-5.

[16] G. S. Moschytz and M. Hofbauer. *Adaptive Filter*. Springer-Verlag, 2000. ISBN 3-540-67651-1.

[17] Philips Semiconductors. *UCB1400 Audio codec with touch screen controller and power management monitor*, Juni 2002. Rev. 02.

[18] Philips Semiconductors. *UCB1400 Audio codec with touch screen controller and power management monitor*, Juni 2002. Rev. 02.

[19] J. Reichardt and B. Schwarz. *VHDL-Synthese : Entwurf digitaler Schaltungen und Systeme*. Oldenbourg Wissenschaftsverlag GmbH, 4. edition, 2007. ISBN 978-3-486-58192-8.

[20] Sennheiser. *Hör-/Sprechgarnituren für die Luftfahrt mit aktiver Lärmkompensation*, Mai 2002.

[21] P. Vary and U. Heute. *Digitale Sprachsignalverarbeitung*. B. G. Teubner, 1998. ISBN 3-519-06165-1.

[22] D. C. von Grünigen. *Digitale Signalverarbeitung*. Carl Hanser Verlag, 3. edition, 2004. ISBN 3-446-22861-6.

[23] J. F. Wakerly. *Digital Design Principles and Practices*. Pearson Education, Inc., 4. edition, 2006. ISBN 0-13-186389-4.

[24] B. Widrow and S. D. Stearns. *Adaptive Signal Processing*. Prentice-Hall, Inc., 1985. ISBN 0-13-004029-01.

[25] XILINX. *Spartan-3 FPGA Family: Complete Data Sheet*, November 2007. Rev. 2.3.

[26] U. Zölzer. *Digitale Audiosignalverarbeitung*. B. G. Teubner, 3. edition, 2005. ISBN 3-519-26180-4.

[27] K. Zona. Making future commercial aircraft quieter. Technical report, NASA Glenn Research Center, März 2006.

A Signalverzeichnis

Signalname	Beschreibung	Äquivalent	Erstmaliges Auftreten
$s(t)$	Störsignal in analoger / akkustischer Form		Kap. 1
$x(t)$	Referenzsignal in analoger / akkustischer Form		Kap. 1
$d(t)$	Primärsignal in analoger / akkustischer Form		Kap. 1
$y(t)$	Kompensationssignal in analoger / akkustischer Form		Kap. 1
$e(t)$	Fehlersignal in analoger / akkustischer Form		Kap. 1
$x(n)$	Referenzsignal in digitaler / diskreter Form	XN	Kap. 1
$e(n)$	Fehlersignal in digitaler / diskreter Form	EN	Kap. 1
$y(n)$[4]	Ausgangssignal eines unbekanten Systems		Kap. 2.1
$\widehat{y}(n)$	Ausgangssiganl des adaptiven Filter (Näherung von $y(n)$)		Kap. 2.1
$i(n)$	Signalinterferenzen		Kap. 2.1
$\widehat{x}(n)$	Ausgangssiganl des adaptiven Filter (Näherung von $x(n)$)		Kap. 2.3
$d(n)$	Primärsignal in digitaler / diskreter Form	DN	Kap. 2.4
$n(n)$	Nutzsignal in digitaler / diskreter Form		Kap. 2.4
$s(n)$	Störsignal in digitaler / diskreter Form		Kap. 2.4
$y(n)$	Kompensationssignal in digitaler / diskreter Form	YN	Kap. 2.4
$w(n)$	Filterkoeffizient	$COEF$	Kap. 3.1
μ	Schrittweitenfaktor des LMS-Adaptionsalgorithmus	MUE	Kap. 3.3.1
XN	Referenzsignal in digitaler / diskreter Form	$x(n)$	Kap. 5.2
EN	Fehlersignal in digitaler / diskreter Form	$e(n)$	Kap. 5.2
YN	Kompensationssignal in digitaler / diskreter Form	$y(n)$	Kap. 5.2
MUE	Schrittweitenfaktor des LMS-Adaptionsalgorithmus	μ	Kap. 5.4
$SAMP\_FILT$	Abtastwert des Referenzsignals aus dem Block-RAM RAM_SAMP_FILT		Kap. 5.4
$SAMP\_LMS$	Verzögerter Abtastwert des Referenzsignals aus dem Block-RAM RAM_SAMP_LMS		Kap. 5.4
$COEF$	Filterkoeffizient	$w(n)$ $COEF\_O$	Kap. 5.4
$COEF\_I$	Adaptierter Filterkoeffizient		Kap. 5.4
$d(t) \equiv e(t)$	Durch die Echtzeitkompensation erhaltenes Primärsignal	$e(t)$	Kap. 7

Tabelle 4: Signalnamen mit Beschreibung, Äquivalenten Signalen und erstmaligem Auftreten

[4]Diese Bezeichnung tritt mit zwei Bedeutungen auf: 1. Ausgangssignal eines unbekannten Systems, 2. Kompensationssignal

B Messtechnische Ergebnisse der Untersuchung des Sekundärpfades

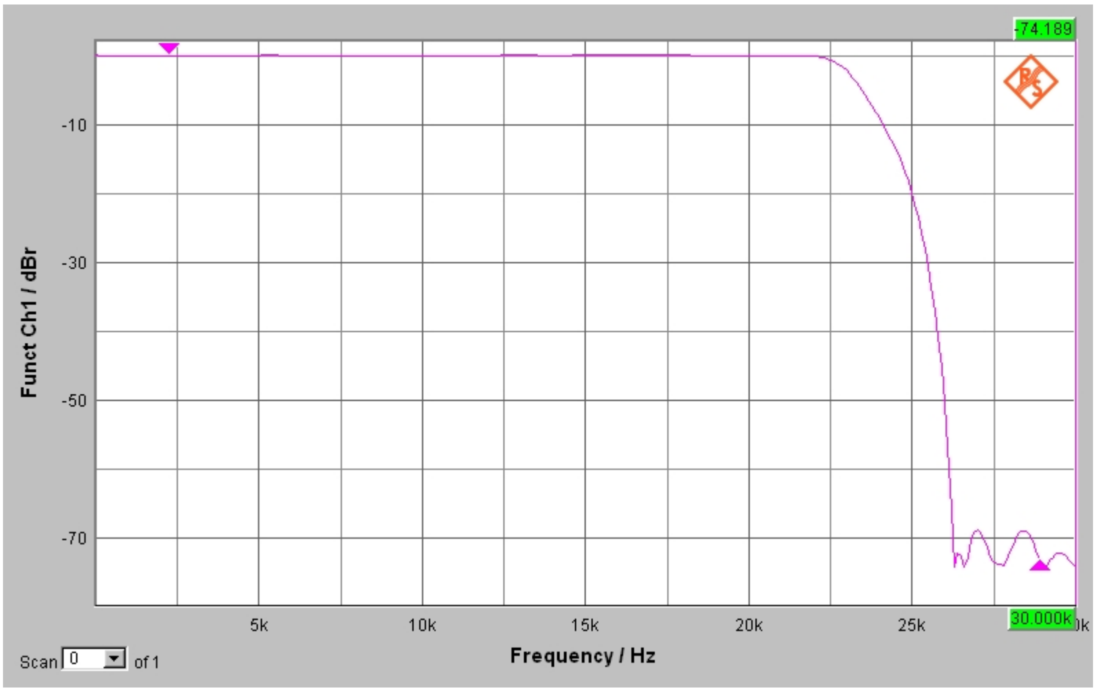

Abb. 76: Frequenzgangmessung des Audio-Codec. Abtastfrequenz $f_s = 48\ kHz$

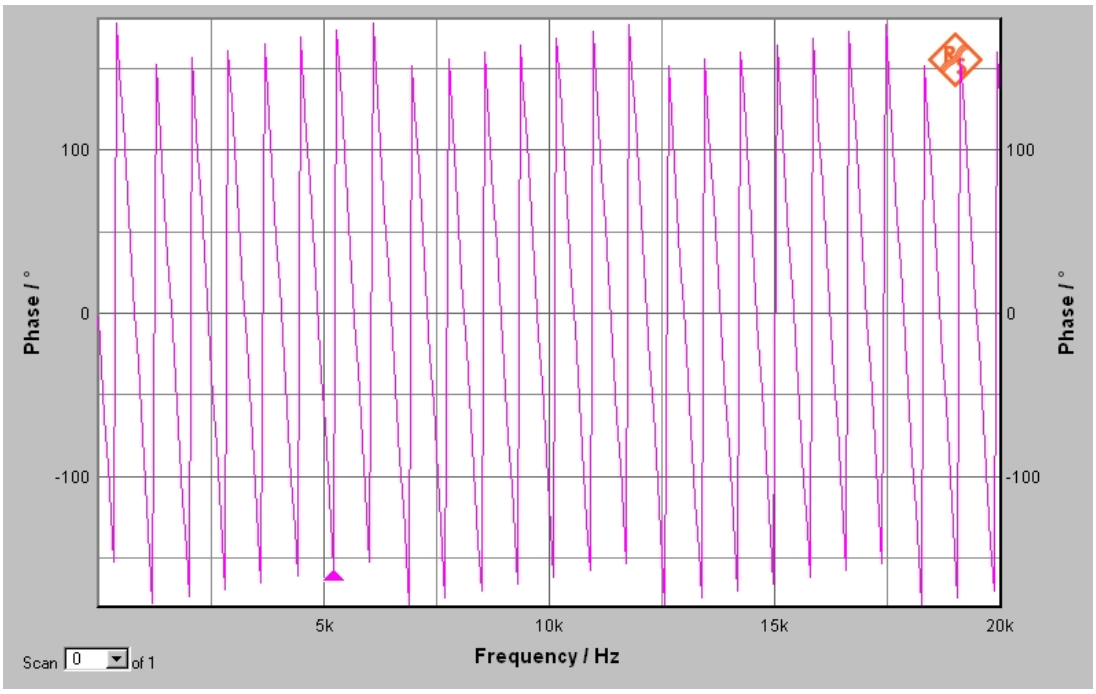

Abb. 77: Phasengangmessung des Audio-Codecs

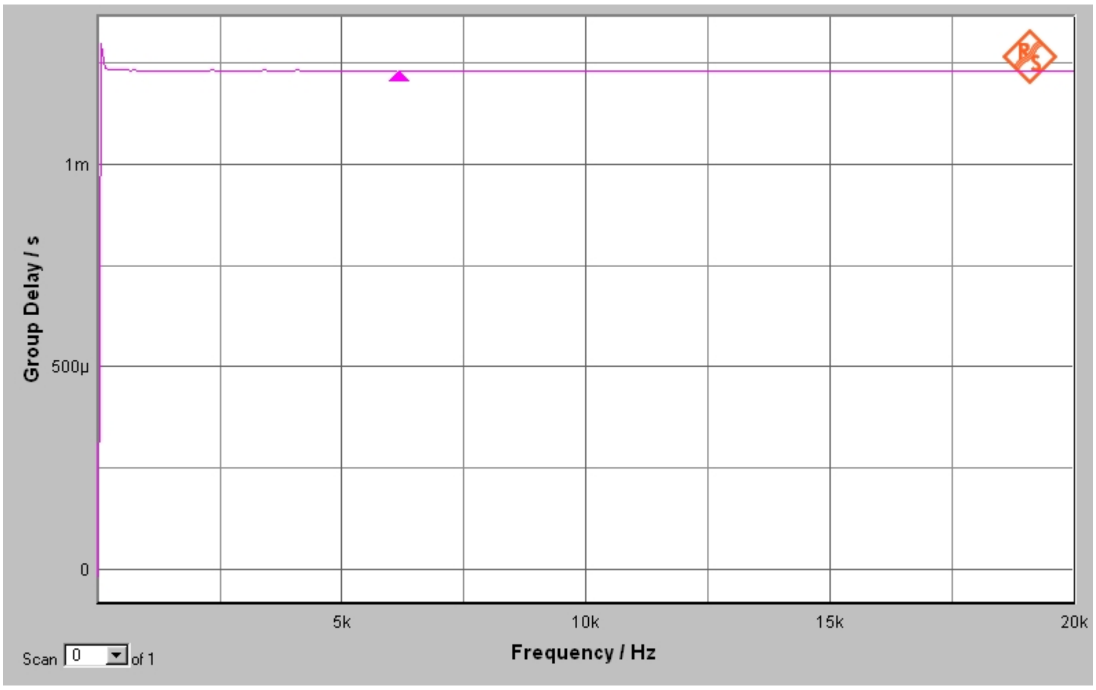

Abb. 78: Gruppenlaufzeitmessung des Audio-Codecs. $t_{gd} = 1,2ms$

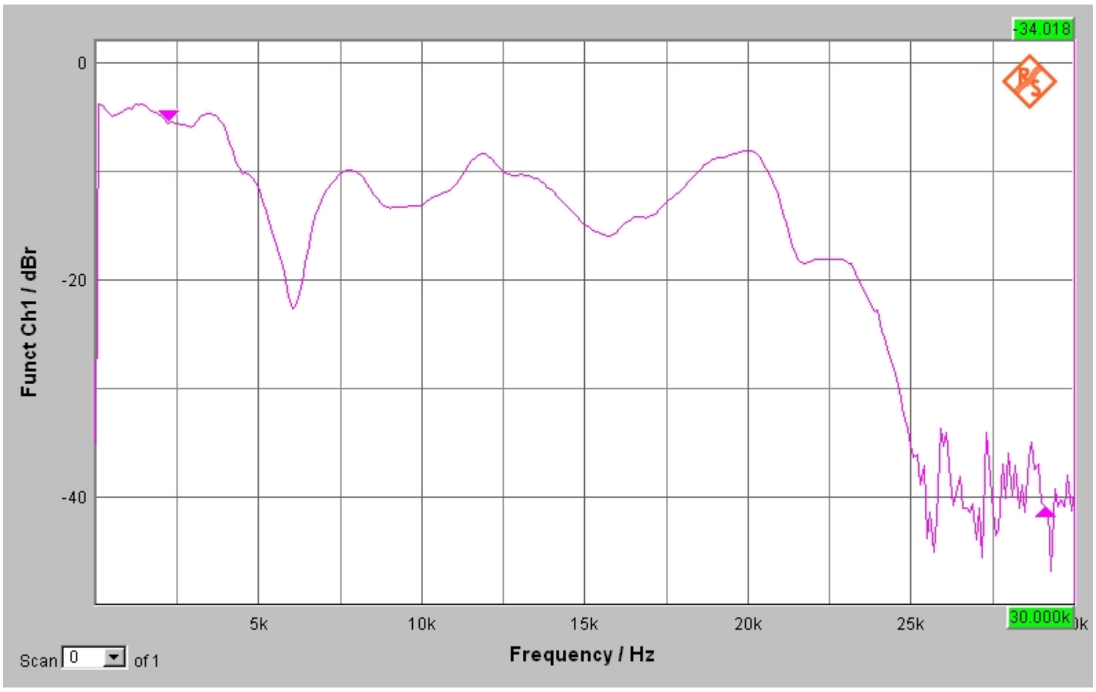

Abb. 79: Frequenzgangmessung des Sekundärpfades, bestehend aus Audio-Codec, Kopfhörer und Inohr-Mikrofon. Ab einer Frequenz von $5\ kHz$ *deutliche Abschwächung des Signals um bis zu* $-22\ dB$

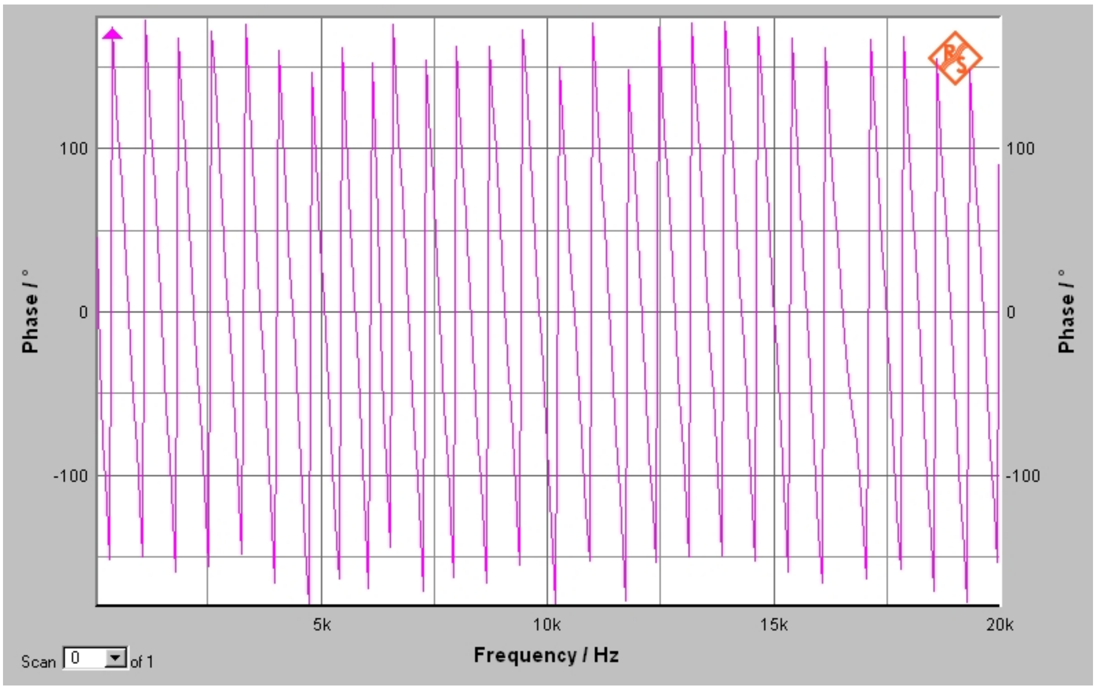

Abb. 80: Phasengangmessung des Sekundärpfades, bestehend aus Audio-Codec, Kopfhörer und Inohr-Mikrofon

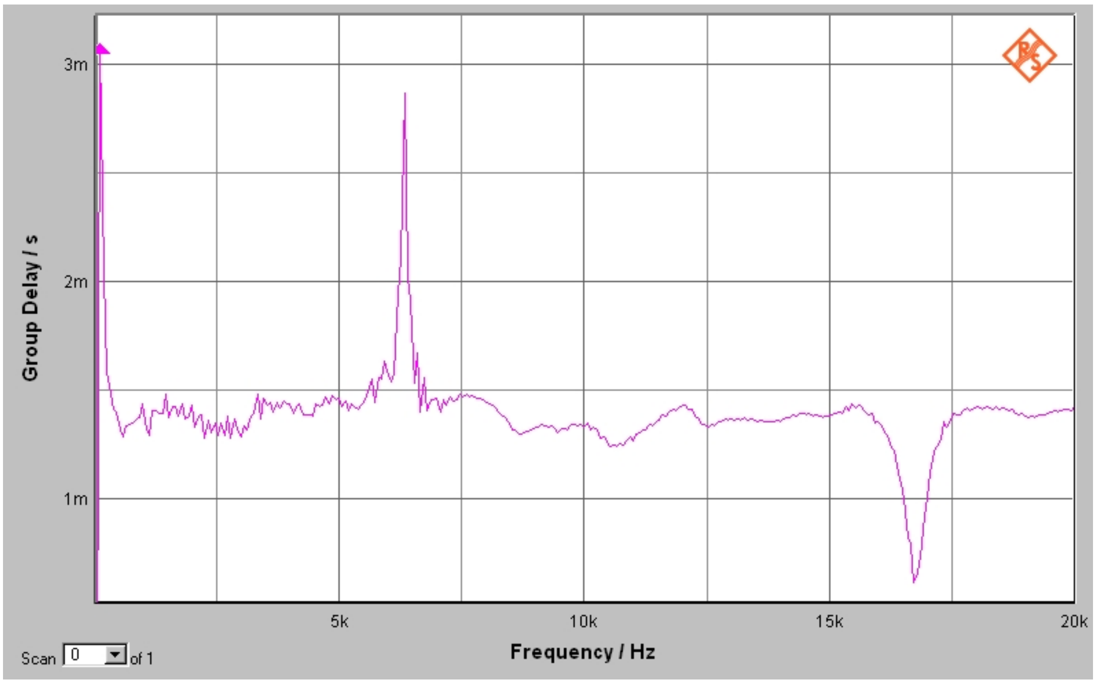

Abb. 81: Gruppenlaufzeitmessung des Sekundärpfades, bestehend aus Audio-Codec, Kopfhörer und Inohr-Mikrofon. Laufzeit bewegt sich im unteren und oberen Frequenzbereich zwischen $1,3\ ms$ und $1,5\ ms$. Bei einer Frequenz von $6\ kHz$ springt die Laufzeit auf $2,8\ ms$ und bei $16kHz$ sinkt sie kurzzeitig auf $0,6\ ms$ ab.

Die Ursache des Extremums in der Gruppenlaufzeit des Sekundärpfades bei einer Frequenz von $6\ kHz$ lässt sich durch akkustische Eigenschaften des Hohlraums bestehend aus Kopfhörer, Ohr und Gehörgang erklären. Durch die sogenannte Helmholtz-Resonanz [6] entstehen Luftwirbel, welche die Laufzeit eines Signal verlängern.
Das zweite Extremum bei einer Frequenz von $16\ kHz$ ensteht durch die Annäherung an das Ende des Übertragungsbereiches vom Inohr-Mikrofon.

C Ergebnisse der Signallaufzeitmessungen

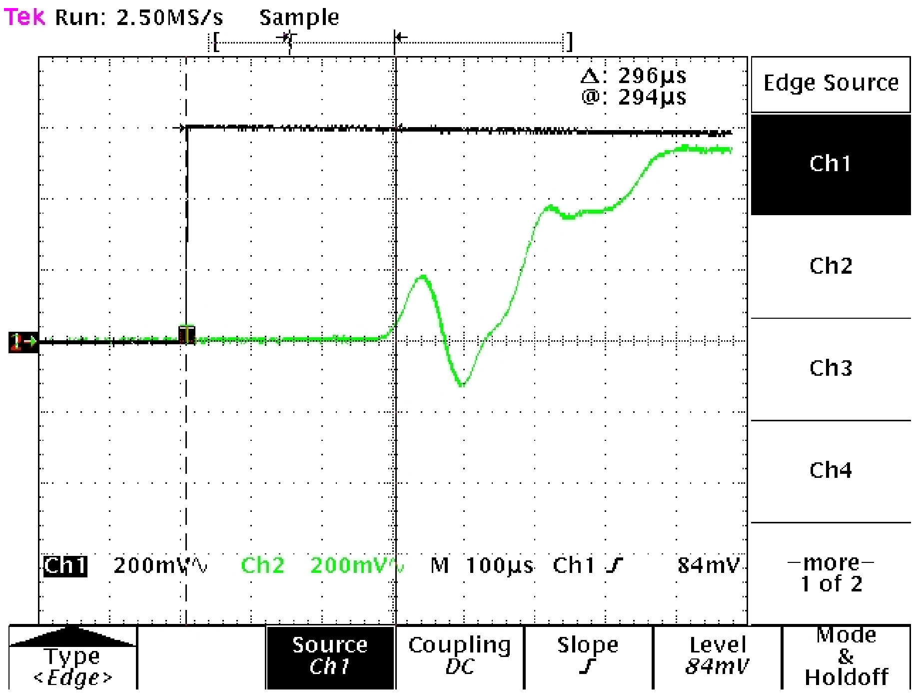

Abb. 82: Messergebnis $\Delta t = 296\ \mu s = t1$

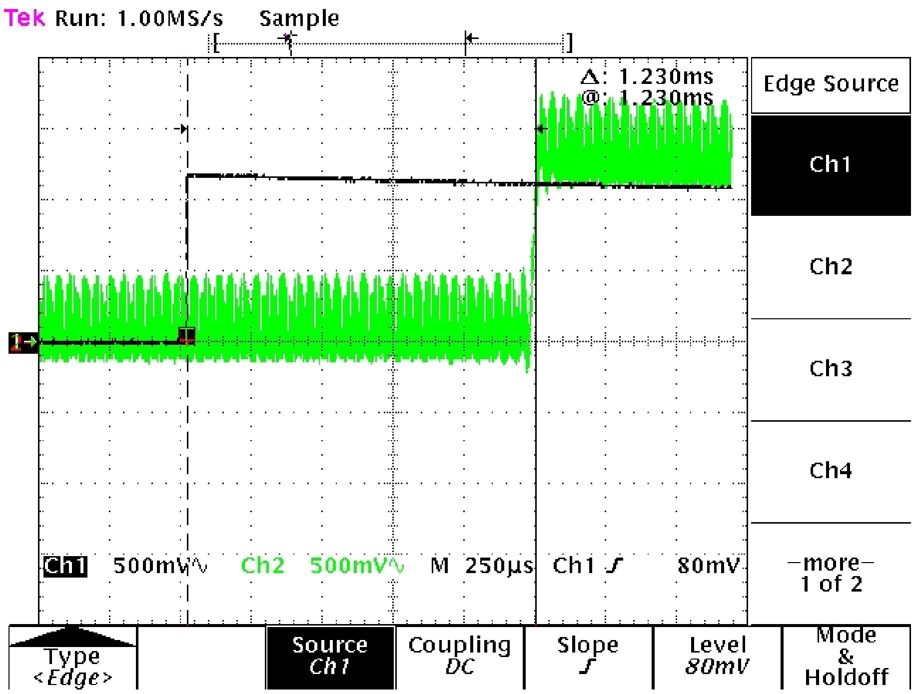

Abb. 83: Messergebnis $\Delta t = 1230\ \mu s = t2$

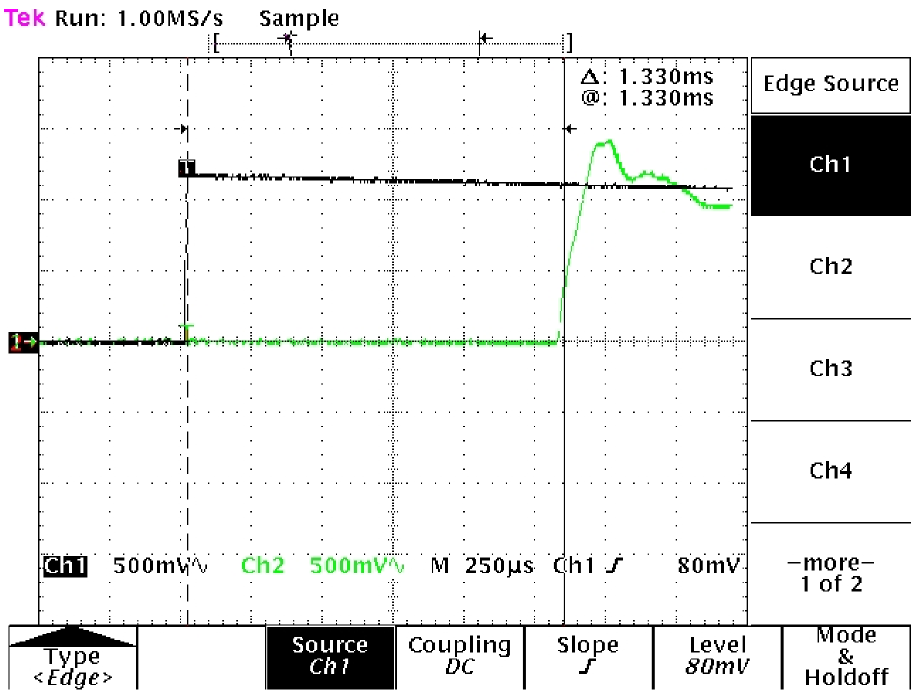

Abb. 84: Messergebnis $\Delta t = 1330\ \mu s = t3$

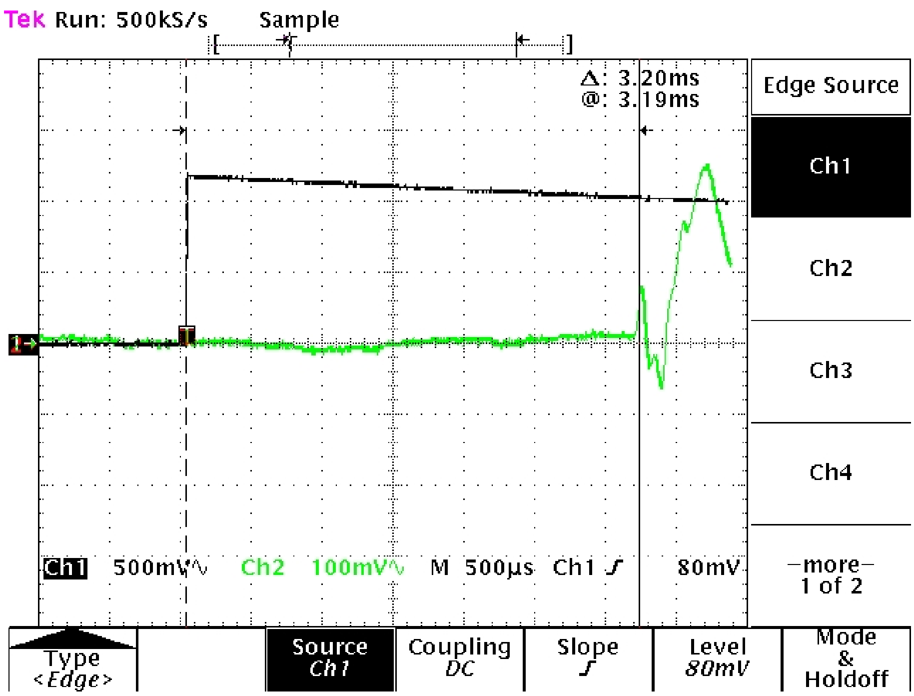

Abb. 85: Messergebnis $\Delta t = 3200\ \mu s = t4$

D Matlab Code

D.1 Level-2 S-Function Adaptives FIR-Filter

Listing 1: Level-2 m file S-Function des adaptiven FIR-Filters mit der Ordnung $N = 256$

```

function adapt_filter(block)
% Level-2 M file S-Function for an adaptive FIR-Filter
% with order N = 256
% Input1: Reference signal
% Input2: Coefficient vector
% Output: Compensation signal

  setup(block);

%endfunction

function setup(block)

  %% Regieste number of input and output ports
  block.NumInputPorts  = 2;
  block.NumOutputPorts = 1;

  %% Setup functional port properties to dynamically
  %% inherited.
  block.SetPreCompInpPortInfoToDynamic;
  block.SetPreCompOutPortInfoToDynamic;

  block.InputPort(1).Complexity   = 'Real';
  block.InputPort(1).DataTypeId   = 0;
  block.InputPort(1).SamplingMode = 'Sample';
  block.InputPort(1).Dimensions   = 1;
  block.InputPort(1).DirectFeedthrough = false;

  block.InputPort(2).Complexity   = 'Real';
  block.InputPort(2).DataTypeId   = 0;
  block.InputPort(2).SamplingMode = 'Sample';
  block.InputPort(2).Dimensions   = 257;
  block.InputPort(2).DirectFeedthrough = false;

  block.OutputPort(1).Complexity   = 'Real';
  block.OutputPort(1).DataTypeId   = 0;
  block.OutputPort(1).SamplingMode = 'Sample';
  block.OutputPort(1).Dimensions   = 1;

  %% Register methods
  block.RegBlockMethod('PostPropagationSetup',    @DoPostPropSetup);
  block.RegBlockMethod('Start',                   @Start);
  block.RegBlockMethod('Outputs',                 @Outputs);

%endfunction

function DoPostPropSetup(block)

  %% Setup Dwork
  N = 256;                          %% Filter order
  block.NumDworks = 1;
  block.Dwork(1).Name = 'X';        %% x[1],...,x[N+1]
  block.Dwork(1).Dimensions      = N+1;
  block.Dwork(1).DatatypeID      = 0;
  block.Dwork(1).Complexity      = 'Real';
  block.Dwork(1).UsedAsDiscState = true;

  %% Register all tunable parameters as runtime parameters.
  block.AutoRegRuntimePrms;

%endfunction

function Start(block)

  %% Initialize Dwork
  block.Dwork(1).Data = zeros(1, 257);

```

```
%endfunction

function Outputs(block)

  N   = 256;

  x = block.InputPort(1).Data;
  W = block.InputPort(2).Data;

  X = block.Dwork(1).Data;

  X(2:N+1)  = X(1:N);
  X(1)      = x;
  y         = X'*W;
  y         = y.*(-1);

  block.Dwork(1).Data = X;
  block.OutputPort(1).Data = y;

%endfunction
```

D.2 Level-2 S-Function LMS-Adaptionsalgorithmus

Listing 2: Level-2 m file S-Function des LMS-Adaptionsalgorithmus

```

function adapt_algorithm(block)
% Level-2 M file S-Function for a LMS-Algorithm
% to adapt the coeffients of a FIR-Filter
% Input1: Reference signal
% Input2: Error signal
% Output: Coefficient vector

  setup(block);

%endfunction

function setup(block)

  %% Register dialog parameter: LMS step size
  block.NumDialogPrms = 1;
  block.DialogPrmsTunable = {'Tunable'};

  %% Regieste number of input and output ports
  block.NumInputPorts  = 2;
  block.NumOutputPorts = 1;

  %% Setup functional port properties to dynamically
  %% inherited.
  block.SetPreCompInpPortInfoToDynamic;
  block.SetPreCompOutPortInfoToDynamic;

  block.InputPort(1).Complexity   = 'Real';
  block.InputPort(1).DataTypeId   = 0;
  block.InputPort(1).SamplingMode = 'Sample';
  block.InputPort(1).Dimensions   = 1;
  block.InputPort(1).DirectFeedthrough = false;

  block.InputPort(2).Complexity   = 'Real';
  block.InputPort(2).DataTypeId   = 0;
  block.InputPort(2).SamplingMode = 'Sample';
  block.InputPort(2).Dimensions   = 1;
  block.InputPort(2).DirectFeedthrough = false;

  block.OutputPort(1).Complexity   = 'Real';
  block.OutputPort(1).DataTypeId   = 0;
  block.OutputPort(1).SamplingMode = 'Sample';
  block.OutputPort(1).Dimensions   = 257;

  %% Register methods
  block.RegBlockMethod('CheckParameters',         @CheckPrms);
  block.RegBlockMethod('ProcessParameters',       @ProcessPrms);
  block.RegBlockMethod('PostPropagationSetup',    @DoPostPropSetup);
  block.RegBlockMethod('Start',                   @Start);
```

```
  block.RegBlockMethod('Outputs',                    @Outputs);

%endfunction

function CheckPrms(block)
  mu = block.DialogPrm(1).Data;

  if mu <= 0 || mu > 1
    error('Step size must be a scalar between 0 and 1.');
  end

%endfunction

function DoPostPropSetup(block)

  %% Setup Dwork
  N = 256;                          %% Filter order
  block.NumDworks = 2;
  block.Dwork(1).Name = 'X';        %% x[1],...,x[N+1]
  block.Dwork(1).Dimensions       = N+1;
  block.Dwork(1).DatatypeID       = 0;
  block.Dwork(1).Complexity       = 'Real';
  block.Dwork(1).UsedAsDiscState  = true;

  block.Dwork(2).Name = 'W'; %% Filter coefficients
  block.Dwork(2).Dimensions       = N+1;
  block.Dwork(2).DatatypeID       = 0;
  block.Dwork(2).Complexity       = 'Real';
  block.Dwork(2).UsedAsDiscState  = true;

  %% Register all tunable parameters as runtime parameters.
  block.AutoRegRuntimePrms;

%endfunction

function ProcessPrms(block)

  block.AutoUpdateRuntimePrms;

%endfunction

function Start(block)

  %% Initialize Dwork
  block.Dwork(1).Data = zeros(1, 257);
  block.Dwork(2).Data = zeros(1, 257);

%endfunction

function Outputs(block)

  mu = block.RuntimePrm(1).Data;
  N  = 256;

  x = block.InputPort(1).Data;
  e = block.InputPort(2).Data;

  X = block.Dwork(1).Data;
  W = block.Dwork(2).Data;

  X(2:N+1) = X(1:N);
  X(1)     = x;
  W        = W+mu*e*X;

  block.Dwork(1).Data = X;
  block.Dwork(2).Data = W;
  block.OutputPort(1).Data = W;

%endfunction
```

E VHDL-Quellcode

E.1 Entity CODEC_INTERFACE

Listing 3: Codec_Interface VHDL Code

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity CODEC_INTERFACE is
    generic (TAG_REG_WIDTH    : integer := 16;
             ADR_REG_WIDTH    : integer := 20;
             DATA_REG_WIDTH   : integer := 20;
             AUDIO_REG_WIDTH  : integer := 40);
    Port ( SYNC                 : out std_logic;
           BIT_CLK              : in  std_logic;
           SDATA_OUT            : out std_logic;
           SDATA_IN             : in  std_logic;
           RESET_IN             : in  std_logic;
           AUDIO_IN_READY       : out std_logic;  -- ready-pulse to the FIR-filter
           AUDIO_OUT_READY      : in  std_logic;  -- ready-pulse from the FIR-filter
           AUDIO_DATA_IN        : in  std_logic_vector(AUDIO_REG_WIDTH-1 downto 0);  -- audio data from the FIR-filter
           AUDIO_DATA_OUT_LEFT  : out std_logic_vector(19 downto 0);  -- audio data to the FIR-filter
           AUDIO_DATA_OUT_RIGHT : out std_logic_vector(19 downto 0);  -- audio data to the FIR-filter
           SYNC_TST             : out std_logic;  -- test output of the SYNC signal, to be measured by oscilloscope
           SDATA_OUT_TST        : out std_logic   -- test output of SDATA_OUT, to be measured by oscilloscope
         );
end CODEC_INTERFACE;

architecture BEHAVIORAL of CODEC_INTERFACE is

-- counter for the incoming BIT_CLK periods
signal SEQ                  : std_logic_vector(7 downto 0) := (others=>'0');
-- counter for the first four data frames
signal INIT_SEQ             : std_logic_vector(2 downto 0) := (others=>'0');
-- internal sync signal, connected with SYNC output, set for 16 BIT_CLK periods
signal SYNC_INT             : std_logic := '0';
-- register for the outgoing tag, send to codec
signal TAG_REG_OUT          : std_logic_vector(TAG_REG_WIDTH-1 downto 0)
                              := "1110000000000000";
-- register for the incoming tag, received from codec
signal TAG_REG_IN           : std_logic_vector(TAG_REG_WIDTH-1 downto 0)
                              := (others=>'0');
-- switches to enable the TAG_REG_IN and _OUT register
signal TAG_REG_IN_EN        : std_logic := '0';
signal TAG_REG_OUT_EN       : std_logic := '0';
-- register for the incoming status address, received from codec
signal ADR_REG_IN           : std_logic_vector(ADR_REG_WIDTH-1 downto 0)
                              := (others=>'0');
-- register for the outgoing command address, send to codec
signal ADR_REG_OUT          : std_logic_vector(ADR_REG_WIDTH-1 downto 0)
                              := "00000000000000000000";
-- switches to enable the ADR_REG_IN and _OUT register
signal ADR_REG_IN_EN        : std_logic := '0';
signal ADR_REG_OUT_EN       : std_logic := '0';
-- register for the incoming status data, received from codec
signal DATA_REG_IN          : std_logic_vector(DATA_REG_WIDTH-1 downto 0)
                              := (others=>'0');
-- register for the outgoing command data, send to codec
signal DATA_REG_OUT         : std_logic_vector(DATA_REG_WIDTH-1 downto 0)
                              := "00000000000000000000";
-- switches to enable the DATA_REG_IN and _OUT register
signal DATA_REG_IN_EN       : std_logic := '0';
signal DATA_REG_OUT_EN      : std_logic := '0';
-- register for incoming serial audio data, received from codec
signal AUDIO_REG_IN         : std_logic_vector(AUDIO_REG_WIDTH-1 downto 0)
                              := (others=>'0');
-- register for outgoing serial audio data, send to codec
```

```
signal AUDIO_REG_OUT              : std_logic_vector(AUDIO_REG_WIDTH-1 downto 0)
                                  := (others=>'0');
-- switches to enable the AUDIO_REG_IN and _OUT register
signal AUDIO_REG_IN_EN            : std_logic := '0';
signal AUDIO_REG_OUT_EN           : std_logic := '0';
-- counter for the slots of one stream (0 to 12)
signal SLOT_COUNT                 : std_logic_vector(3 downto 0) := (others=>'1');
-- counter for the bits of one slot (0 to 15 TAG-Slot / 0 to 19 all other slots)
signal SLOT_BIT_COUNT             : std_logic_vector(4 downto 0) := (others=>'1');
-- signals for the pulse-stretcher
signal Q1                         : std_logic;
signal LOAD_AUDIO                 : std_logic;

begin


-- process to generate the sync signal and determine the start-up phase
GEN_SYNC: process(BIT_CLK, RESET_IN)
    begin
        if RESET_IN = '1' then
            SEQ <= (others=>'0');
            SYNC_INT <= '0';
        elsif (BIT_CLK'event and BIT_CLK = '1') then
            -- sync high for the first 16 bits (TAG phase)
            if SEQ < x"10" then
                SYNC_INT <= '1';
            -- for the following 240 bits the sync is low
            else
                SYNC_INT <= '0';
            end if;
            -- count the bits of one frame
            SEQ <= SEQ + 1;
        end if;
    end process GEN_SYNC;

SYNC <= SYNC_INT;
SYNC_TST <= SYNC_INT;


-- process to control the enable switches for the tag- data-
-- and audio-in-registers
CONTROL_IN_REGISTERS: process(BIT_CLK, RESET_IN)
    begin
        if RESET_IN = '1' then
            TAG_REG_IN_EN <= '0';
            ADR_REG_IN_EN <= '0';
            DATA_REG_IN_EN <= '0';
            AUDIO_REG_IN_EN <= '0';
            AUDIO_IN_READY <= '0';
        -- enable signals for the in-registers are set on the rising edge of BIT_CLK
        elsif (BIT_CLK'event and BIT_CLK = '1') then
            -- while SYNC is high, enable tag-register (16 bit)
            if SYNC_INT = '1' then
                TAG_REG_IN_EN <= '1';
            else
                -- disable register for the incoming tag when sync is low
                TAG_REG_IN_EN <= '0';
            end if;

            -- enable adr-register from the end of slot 0 to the end of slot 1 (20 bit)
            if (SLOT_COUNT = x"0" and SLOT_BIT_COUNT = "01111") then
                ADR_REG_IN_EN <= '1';
            elsif (SLOT_COUNT = x"1" and SLOT_BIT_COUNT < "10011") then
                ADR_REG_IN_EN <= '1';
            else
                ADR_REG_IN_EN <= '0';
            end if;

            -- enable data-register from the end of slot 1 to the end of slot 2 (20 bit)
            if (SLOT_COUNT = x"1" and SLOT_BIT_COUNT = "10011") then
                DATA_REG_IN_EN <= '1';
            elsif (SLOT_COUNT = x"2" and SLOT_BIT_COUNT < "10011") then
                DATA_REG_IN_EN <= '1';
            else
                DATA_REG_IN_EN <= '0';
```

```
            end if;

            -- enable audio-register from the end of slot 2 to the end of slot 4 (40 bit)
            if (SLOT_COUNT = x"2" and SLOT_BIT_COUNT = "10011") then
                AUDIO_REG_IN_EN <= '1';
            elsif SLOT_COUNT = x"3" then
                AUDIO_REG_IN_EN <= '1';
            elsif (SLOT_COUNT = x"4" and SLOT_BIT_COUNT < "10011") then
                AUDIO_REG_IN_EN <= '1';
            else
                AUDIO_REG_IN_EN <= '0';
            end if;

            -- at the beginning of slot 5 and if the audio data is valid, generate ready-pulse
            if ((SLOT_COUNT = x"5" and SLOT_BIT_COUNT = x"00000")
                 and TAG_REG_IN(TAG_REG_WIDTH-1) = '1'
                 and (TAG_REG_IN(TAG_REG_WIDTH-4) = '1'
                 and TAG_REG_IN(TAG_REG_WIDTH-5) = '1')
                 and INIT_SEQ > 4) then
                AUDIO_IN_READY <= '1';
            else
                AUDIO_IN_READY <= '0';
            end if;
        end if;
    end process CONTROL_IN_REGISTERS;


-- process to control the enable switches for the tag- data-
-- and audio-out-registers
CONTROL_OUT_REGISTERS: process(BIT_CLK, RESET_IN)
    begin
        if RESET_IN = '1' then
            TAG_REG_OUT_EN <= '0';
            ADR_REG_OUT_EN <= '0';
            DATA_REG_OUT_EN <= '0';
            AUDIO_REG_OUT_EN <= '0';
        -- enable signals for the out-registers are set on the falling edge of BIT_CLK
        elsif (BIT_CLK'event and BIT_CLK = '0') then
            -- while SYNC is high, enable tag-register (16 bit)
            if SYNC_INT = '1' then
                TAG_REG_OUT_EN <= '1';
            else
                -- disable register for the outgoing tag when sync is low
                TAG_REG_OUT_EN <= '0';
            end if;

            -- all other out-registers are only enabled if the output frame contains valid data
            if TAG_REG_OUT(TAG_REG_WIDTH-1) = '1' then
                -- slot 1 contains valid data
                if TAG_REG_OUT(TAG_REG_WIDTH-2) = '1' then
                    -- enable adr-register from the end of slot 0 to the end of slot 1 (20 bit)
                    if (SLOT_COUNT = x"0" and SLOT_BIT_COUNT = "01111") then
                        ADR_REG_OUT_EN <= '1';
                    elsif (SLOT_COUNT = x"1" and SLOT_BIT_COUNT < "10011") then
                        ADR_REG_OUT_EN <= '1';
                    else
                        ADR_REG_OUT_EN <= '0';
                    end if;
                else
                    ADR_REG_OUT_EN <= '0';
                end if;

                -- slot 2 contains valid data
                if TAG_REG_OUT(TAG_REG_WIDTH-3) = '1' then
                    -- enable data-register from the end of slot 1 to the end of slot 2 (20 bit)
                    if (SLOT_COUNT = x"1" and SLOT_BIT_COUNT = "10011") then
                        DATA_REG_OUT_EN <= '1';
                    elsif (SLOT_COUNT = x"2" and SLOT_BIT_COUNT < "10011") then
                        DATA_REG_OUT_EN <= '1';
                    else
                        DATA_REG_OUT_EN <= '0';
                    end if;
```

```vhdl
                else
                    DATA_REG_OUT_EN <= '0';
                end if;

                -- slot 3 and 4 contain valid data
                if TAG_REG_OUT((TAG_REG_WIDTH-4) downto (TAG_REG_WIDTH-5)) = "11" then
                    -- enable audio-register from the end of slot 2 to the end of slot 4 (40
                        bit)
                    if (SLOT_COUNT = x"2" and SLOT_BIT_COUNT = "10011") then
                        AUDIO_REG_OUT_EN <= '1';
                    elsif SLOT_COUNT = x"3" then
                        AUDIO_REG_OUT_EN <= '1';
                    elsif (SLOT_COUNT = x"4" and SLOT_BIT_COUNT < "10011") then
                        AUDIO_REG_OUT_EN <= '1';
                    else
                        AUDIO_REG_OUT_EN <= '0';
                    end if;
                else
                    AUDIO_REG_OUT_EN <= '0';
                end if;
            end if;
        end if;
    end process CONTROL_OUT_REGISTERS;


-- process to count the slots of one frame and the bits of one slot
COUNT_SLOTS: process(BIT_CLK, RESET_IN)
    begin
        -- reset the counters to -1, because the frame starts with slot 0 bit 0
        if RESET_IN = '1' then
            SLOT_COUNT <= (others=>'1');
            SLOT_BIT_COUNT <= (others=>'1');
            INIT_SEQ <= (others=>'0');
        elsif (BIT_CLK'event and BIT_CLK = '1') then
            -- SYNC high (tag phase), always slot 0
            if SYNC_INT = '1' then
                SLOT_COUNT <= (others=>'0');   -- reset slot counter
                -- reset the slot bit counter after the last 20 bit data slot
                if SLOT_BIT_COUNT = 19 then
                    SLOT_BIT_COUNT <= (others=>'0');
                else
                    SLOT_BIT_COUNT <= SLOT_BIT_COUNT + 1;   -- count slot bits
                end if;
            -- SYNC LOW (data phase), slot 1-12
            elsif (SYNC_INT = '0' and SLOT_COUNT < 15) then
                -- reset the slot bit counter and increase the slot counter
                -- after the 16 bit tag slot
                if (SLOT_BIT_COUNT = 15 and SLOT_COUNT = 0) then
                    SLOT_BIT_COUNT <= (others=>'0');
                    SLOT_COUNT <= SLOT_COUNT + 1;
                -- reset the slot bit counter and increase the slot counter
                -- after a 20 bit data slot
                elsif SLOT_BIT_COUNT = 19 then
                    SLOT_BIT_COUNT <= (others=>'0');
                    SLOT_COUNT <= SLOT_COUNT + 1;
                else
                    SLOT_BIT_COUNT <= SLOT_BIT_COUNT + 1;   -- count slot bits
                end if;
            end if;
            -- count the first four frames after startup or reset (start-up phase)
            if (SLOT_COUNT = 5 and SLOT_BIT_COUNT = 0 and INIT_SEQ < 6) then
                INIT_SEQ <= INIT_SEQ + 1;
            else
                INIT_SEQ <= INIT_SEQ;
            end if;
        end if;
    end process COUNT_SLOTS;


-- process shift the data from SDATA_IN to the in-registers
SHIFT_DATA_IN: process(BIT_CLK, RESET_IN)
    begin
        if RESET_IN = '1' then
            TAG_REG_IN <= (others=>'0');
            ADR_REG_IN <= (others=>'0');
```

```
            DATA_REG_IN <= (others=>'0');
            AUDIO_REG_IN <= (others=>'0');
        -- bits from SDATA_IN are sampled on the falling edge of BIT_CLK
        elsif (BIT_CLK'event and BIT_CLK = '0') then
            -- shift tag in
            if TAG_REG_IN_EN = '1' then
                TAG_REG_IN <= TAG_REG_IN(TAG_REG_WIDTH-2 downto 0)
                              & SDATA_IN;
            -- shift status address in
            elsif ADR_REG_IN_EN = '1' then
                ADR_REG_IN <= ADR_REG_IN(ADR_REG_WIDTH-2 downto 0)
                              & SDATA_IN;
            -- shift status data in
            elsif DATA_REG_IN_EN = '1' then
                DATA_REG_IN <= DATA_REG_IN(DATA_REG_WIDTH-2 downto 0)
                               & SDATA_IN;
            -- shift audio data in
            elsif AUDIO_REG_IN_EN = '1' then
                AUDIO_REG_IN <= AUDIO_REG_IN(AUDIO_REG_WIDTH-2 downto 0)
                                & SDATA_IN;
            end if;
        end if;
    end process SHIFT_DATA_IN;

AUDIO_DATA_OUT_LEFT <= AUDIO_REG_IN(19 downto 0);
AUDIO_DATA_OUT_RIGHT <= AUDIO_REG_IN(39 downto 20);


-- process selects which out-register output is connected to SDATA_OUT
SHIFT_DATA_OUT: process(BIT_CLK, RESET_IN)
    begin
        if RESET_IN = '1' then
            SDATA_OUT <= '0';
            SDATA_OUT_TST <= '0';
        elsif (BIT_CLK'event and BIT_CLK = '1') then
            -- shift tag out
            if TAG_REG_OUT_EN = '1' then
                SDATA_OUT <= TAG_REG_OUT(TAG_REG_WIDTH-1);
                SDATA_OUT_TST <= TAG_REG_OUT(TAG_REG_WIDTH-1);
            -- shift command address out
            elsif ADR_REG_OUT_EN = '1' then
                SDATA_OUT <= ADR_REG_OUT(ADR_REG_WIDTH-1);
                SDATA_OUT_TST <= ADR_REG_OUT(ADR_REG_WIDTH-1);
            -- shift command data out
            elsif DATA_REG_OUT_EN = '1' then
                SDATA_OUT <= DATA_REG_OUT(DATA_REG_WIDTH-1);
                SDATA_OUT_TST <= DATA_REG_OUT(DATA_REG_WIDTH-1);
            -- shift audio data out
            elsif AUDIO_REG_OUT_EN = '1' then
                SDATA_OUT <= AUDIO_REG_OUT(AUDIO_REG_WIDTH-1);
                SDATA_OUT_TST <= AUDIO_REG_OUT(AUDIO_REG_WIDTH-1);
            else
                SDATA_OUT <= '0';
                SDATA_OUT_TST <= '0';
            end if;
        end if;
    end process SHIFT_DATA_OUT;

-- processes to synchronize and stretch the ready pulse, received from the FIR-filter
-- input pulse: AUDIO_OUT_READY, output pulse: LOAD_AUDIO
AUDIO_OUT_READY_SYNC1: process(AUDIO_OUT_READY, LOAD_AUDIO)
    begin
        -- reset Q1 if LOAD_AUDIO is set
        if LOAD_AUDIO = '1' then
            Q1 <= '0';
        -- set Q1 on the rising edge of the ready pulse
        elsif (AUDIO_OUT_READY'event and AUDIO_OUT_READY = '1') then
            Q1 <= '1';
        end if;
    end process AUDIO_OUT_READY_SYNC1;

AUDIO_OUT_READY_SYNC2: process(BIT_CLK)
    begin
        -- set LOAD_AUDIO on the falling edge of BIT_CLK to Q1
        -- the LOAD_AUDIO pulse gets the length of a BIT_CLK period
```

```
        -- and is high on the rising edge of BIT_CLK
        if (BIT_CLK'event and BIT_CLK = '0') then
            LOAD_AUDIO <= Q1;
        end if;
    end process AUDIO_OUT_READY_SYNC2;


-- process to load and shift the out-registers according to the enable signals
-- and the start-up phase
LOAD_OUT_REGISTERS: process(BIT_CLK, RESET_IN)
    begin
        if RESET_IN = '1' then
            TAG_REG_OUT(15)          <=  '1';              -- Valid data
            TAG_REG_OUT(14 downto 3) <= "110000000000";    -- Slots that contain valid data (Slot 1 and 2 valid)
            TAG_REG_OUT(2 downto 0)  <= "000";             -- Primary codec (Slot 0 Tag value: 0xE000)
            ADR_REG_OUT <= "00000000000000000000";         -- Slot 1 Reset Register address: 0x00
            DATA_REG_OUT <= "00000000000000000000";        -- Slot 2 data value: 0x0000
            AUDIO_REG_OUT <= (others=>'0');
        -- shift and load out-registers on the rising edge of BIT_CLK
        elsif (BIT_CLK'event and BIT_CLK = '1') then
            -- shift tag register, feed back the output to the input
            if TAG_REG_OUT_EN = '1' then
                TAG_REG_OUT <= TAG_REG_OUT(TAG_REG_WIDTH-2 downto 0)
                               & TAG_REG_OUT(TAG_REG_WIDTH-1);
            -- shift command address register, feed back the output to the input
            elsif ADR_REG_OUT_EN = '1' then
                ADR_REG_OUT <= ADR_REG_OUT(ADR_REG_WIDTH-2 downto 0)
                               & ADR_REG_OUT(ADR_REG_WIDTH-1);
            -- shift command data register, feed back the output to the input
            elsif DATA_REG_OUT_EN = '1' then
                DATA_REG_OUT <= DATA_REG_OUT(DATA_REG_WIDTH-2 downto 0)
                                & DATA_REG_OUT(DATA_REG_WIDTH-1);
            -- shift audio register, fill with zeros
            elsif AUDIO_REG_OUT_EN = '1' then
                AUDIO_REG_OUT <= AUDIO_REG_OUT(AUDIO_REG_WIDTH-2 downto 0)
                                 & '0';
            -- load audio register with filtered data after the ready pulse
            elsif LOAD_AUDIO = '1' then
                AUDIO_REG_OUT <= AUDIO_DATA_IN;
            -- start-up phase frame 1: load tag, command address and command data registers
            -- writing 0x0000 to the Master Volume register (address:0x02) (Master mute off)
            elsif INIT_SEQ = 1 then
                TAG_REG_OUT <= "1110000000000000";         -- Slot 0 tag value: 0xE000 (Slots 1 and 2 valid)
                ADR_REG_OUT <= "00000010000000000000";     -- Slot 1 address value: 0x02
                DATA_REG_OUT <= "00000000000000000000";    -- Slot 2 data value: 0x0000
            -- start-up phase frame 2: load tag, command address and command data registers
            -- writing 0x0404 to the Record Select register (address:0x1A) (Left record source = Line In L)
            -- (Right record source = Line In R)
            elsif INIT_SEQ = 2 then
                TAG_REG_OUT <= "1110000000000000";         -- Slot 0 tag value: 0xE000 (Slots 1 and 2 valid)
                ADR_REG_OUT <= "00011010000000000000";     -- Slot 1 address value: 0x1A
                DATA_REG_OUT <= "00000100000000010000000";    -- Slot 2 data value: 0x0404 0000010000000001000000
            -- start-up phase frame 3: load tag, command address and command data registers
            -- writing 0x0000 to the Record Gain register (address:0x1C) (Master Record mute off)
            elsif INIT_SEQ = 3 then
                TAG_REG_OUT <= "1110000000000000";         -- Slot 0 tag value: 0xE000 (Slots 1 and 2 valid)
                ADR_REG_OUT <= "00011100000000000000";     -- Slot 1 address value: 0x1C
                DATA_REG_OUT <= "00000000000000000000";    -- Slot 2 data value: 0x0000
            -- start-up phase frame 4: load tag, command address and command data registers
            -- writing 0x0040 to the Feature Control/Status Register 1 (address:0x6A) (Headphone driver on)
            elsif INIT_SEQ = 4 then
                TAG_REG_OUT <= "1110000000000000";         -- Slot 0 tag value: 0xE000 (Slots 1 and 2 valid)
                ADR_REG_OUT <= "01101010000000000000";     -- Slot 1 address value: 0x6A
                DATA_REG_OUT <= "00000000010000000000";    -- Slot 2 data value: 0x0040
```

```
            -- start-up phase frame 5: load tag, command address and command data registers
            -- set the tag to audio data output: slot 3 and 4 are valid
            -- value stays the same until next start-up or reset
            elsif INIT_SEQ = 5 then
                TAG_REG_OUT <= "1001100000000000";       -- Slot 0 tag value: 0x9800 (Slots 3 and 4 valid)
                ADR_REG_OUT <= "00000000000000000000";   -- Reset ADR_REG_OUT to 0x00000
                DATA_REG_OUT <= "00000000000000000000";  -- Reset DATA_REG_OUT to 0x00000
            end if;
        end if;
    end process LOAD_OUT_REGISTERS;

end BEHAVIORAL;
```

E.2 Adaptives FIR-Filter VHDL Code

E.2.1 Entity LMS_FIR

Listing 4: LMS_FIR VHDL Code

```

-- Sequential LMS FIR Module with MAC Unit
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity LMS_FIR is
generic(WIDTH : positive := 9;        -- counter width
        DELAY : positive := 64 );     -- number of delayed reference signal samples
port
(   CLK, RESET, RD   : in STD_LOGIC;
    XN   : in std_logic_vector(19 downto 0);     -- 20 bit reference signal
    MUE  : in std_logic_vector(17 downto 0);     -- 18 bit step size
    EN   : in std_logic_vector(19 downto 0);     -- 20 bit error signal
    YN   : out std_logic_vector(19 downto 0);    -- 20 bit compensation signal
    FILT_RDY : out std_logic );                  -- ready pulse to codec-interface
end entity LMS_FIR;

architecture HYBRID of LMS_FIR is
    component MAC_LMS
    generic( WIDTH : positive ); -- counter width
    port
    (   CLK, RESET, CLR_REG, EN_REG_EMUE, EN_CNT_CW, EN_CNT_CR, WE_R_C, EN_RESET: in std_logic ;
        X         : in std_logic_vector(2*WIDTH -1 downto 0);
        E         : in std_logic_vector(2*WIDTH -1 downto 0);
        MUE       : in std_logic_vector(2*WIDTH -1 downto 0);
        CNT_C_MAX : out std_logic;
        COEF      : out std_logic_vector(2*WIDTH -1 downto 0)
    );
    end component;

    component RAM_S_257_18
     generic( WIDTH : positive; -- counter Width
              DELAY : positive ); -- sample delay of input signal
     port
     (   RESET, CLK, EN_CNT_S, WE_R_S, EN_X_SHIFT, EN_RESET : in std_logic ;
         DIN        : in  std_logic_vector(2*WIDTH -1 downto 0);
         SAMP_FILT  : out std_logic_vector(2*WIDTH -1 downto 0);   -- sample for filter process
         SAMP_LMS   : out std_logic_vector(2*WIDTH -1 downto 0) ); -- delayed sample for adaption process
    end component;

    component MAC_FIR
    generic( WIDTH : positive );
    port
    (   CLK, CLR_REG, EN_REG_Y, EN_SAT : in std_logic;
        COEF, SAMP : in std_logic_vector(2*WIDTH -1 downto 0); -- 18x18 Bit Mult.
        YN          : out std_logic_vector(19 downto 0);        -- Filter output
        FILT_RDY   : out std_logic );
    end component;

    signal TEMP_1, TEMP_2, ADC_FULL: std_logic;                  -- Puls shorter
    signal COEF_I, SAMP_FILT_I, SAMP_LMS_I : std_logic_vector(2*WIDTH -1 downto 0);
    type ZUSTAENDE is(IDLE, Z_RESET, EMUE, MUL_ACC, WRITE_C, STOP, UPDATE);
```

```
   signal ZUSTAND, Z_PLUS : ZUSTAENDE;
   -- FSM control signal and feedback signal from the coefficient RAM modul
   signal EN_CNT_S, EN_CNT_CW, EN_CNT_CR, WE_R_S, WE_R_C, EN_X_SHIFT : std_logic;
   signal CLR_REG, EN_REG_Y, EN_REG_EMUE, EN_SAT, CNT_C_MAX, EN_RESET : std_logic;

begin
   PULSE_SHORTER: process(CLK,RESET)
   begin
     if RESET = '1' then
         TEMP_1 <= '0' after 3 ns; TEMP_2 <= '0'after 3 ns;
     elsif CLK='1' and CLK'event  then
         TEMP_1 <= RD after 3 ns;      -- ready signal from codec interface
         TEMP_2 <= TEMP_1 after 3 ns;
     end if;
   end process PULSE_SHORTER;

   ADC_FULL <= TEMP_1 and (not TEMP_2) after 3 ns; -- short pulse

   MAC_L: MAC_LMS
   generic map( WIDTH => WIDTH )
   port map
   ( CLK              => CLK,
     RESET            => RESET,
     CLR_REG      => CLR_REG,
     EN_REG_EMUE  => EN_REG_EMUE,
     EN_CNT_CW    => EN_CNT_CW,
     EN_CNT_CR    => EN_CNT_CR,
     WE_R_C       => WE_R_C,
     EN_RESET     => EN_RESET,
     X            => SAMP_LMS_I,
     E            => EN(19 downto 2),
     MUE          => MUE,
     CNT_C_MAX   => CNT_C_MAX,
     COEF         => COEF_I
   );

   RAM: RAM_S_257_18
   generic map( WIDTH => WIDTH,
                DELAY => DELAY )
   port map
   ( RESET        => RESET,
     CLK            => CLK,
     EN_CNT_S          => EN_CNT_S,
     WE_R_S       => WE_R_S,
     EN_X_SHIFT  => EN_X_SHIFT,
     EN_RESET    => EN_RESET,
     DIN          => XN(19 downto 2),
     SAMP_FILT   => SAMP_FILT_I,
     SAMP_LMS    => SAMP_LMS_I );

   MAC_F: MAC_FIR
   generic map( WIDTH => WIDTH )
   port map
   ( CLK          => CLK,
     CLR_REG   => CLR_REG,
     EN_REG_Y     => EN_REG_Y,
     EN_SAT    => EN_SAT,
     COEF         => COEF_I,
     SAMP         => SAMP_FILT_I,
     YN        => YN,
     FILT_RDY  => FILT_RDY
   );

Z_REG: process(CLK)
begin
    if (CLK = '1' and CLK'event) then
       if (RESET = '1') then ZUSTAND <= Z_RESET after 3 ns;
        else ZUSTAND <= Z_PLUS after 3 ns;    -- state update
       end if;
    end if;
end process Z_REG;

UE_AUS_SN : process(ZUSTAND, ADC_FULL, CNT_C_MAX)    -- Mealy-FSM
begin
    EN_RESET     <= '0' after 3 ns;  -- default assignments
```

```
    EN_CNT_S     <= '0' after 3 ns;
    EN_CNT_CW    <= '0' after 3 ns;
    EN_CNT_CR    <= '0' after 3 ns;
    WE_R_S       <= '0' after 3 ns;
    WE_R_C       <= '0' after 3 ns;
    EN_X_SHIFT   <= '0' after 3 ns;
    CLR_REG      <= '0' after 3 ns;
    EN_REG_Y     <= '0' after 3 ns;
    EN_REG_EMUE  <= '0' after 3 ns;
    EN_SAT       <= '0' after 3 ns;
    Z_PLUS       <= IDLE after 3 ns;
    case ZUSTAND is
        when Z_RESET => CLR_REG <= '1' after 3 ns;        -- state for RAM reset
                        EN_RESET <= '1' after 3 ns;
                        EN_CNT_CW <= '1' after 3 ns;
                        EN_CNT_CR <= '1' after 3 ns;
                        EN_CNT_S <= '1' after 3 ns;
                        WE_R_C <= '1' after 3 ns;
                        WE_R_S <= '1' after 3 ns;
                        Z_PLUS <= Z_RESET after 3 ns;
                        if (CNT_C_MAX = '1') then
                            Z_PLUS <= IDLE after 3 ns;
                        end if;
        when IDLE => CLR_REG <= '1' after 3 ns;           -- synchronus reset of MAC an EMUE
            registers
                     if (ADC_FULL = '1') then
                         WE_R_S     <= '1' after 3 ns;     -- sample RAM write enable
                         EN_X_SHIFT <= '1' after 3 ns;
                         Z_PLUS     <= EMUE after 3 ns;
                     end if;
        when EMUE => EN_REG_EMUE <= '1' after 3 ns;       -- enable EMUE register
                     Z_PLUS <= MUL_ACC after 3 ns;
        when MUL_ACC => EN_REG_Y <= '1' after 3 ns;    -- enable MAC register
                        EN_CNT_CR <= '1' after 3 ns;
                        EN_CNT_S <= '1' after 3 ns;
                        Z_PLUS    <= WRITE_C after 3 ns;
         when WRITE_C => EN_CNT_CW <= '1' after 3 ns;
                         WE_R_C <= '1' after 3 ns;         -- coefficient RAM write enable
                       Z_PLUS <= MUL_ACC after 3 ns;
                       if (CNT_C_MAX = '1') then
                           Z_PLUS <= STOP after 3 ns;
                       end if;
        when STOP => EN_REG_Y <= '1' after 3 ns;
                     EN_CNT_CR <= '1' after 3 ns;
                     Z_PLUS <= UPDATE after 3 ns;
        when UPDATE => EN_SAT    <= '1' after 3 ns;  -- saturate cycle result
                       EN_CNT_CW <= '1' after 3 ns;
                       WE_R_C <= '1' after 3 ns;
                       Z_PLUS    <= IDLE after 3 ns;
        when others => null; -- reaction in pseudo states
      end case;
  end process UE_AUS_SN;

end architecture HYBRID;
```

E.2.2 Entity RAM_S_257_18

Listing 5: RAM_S_257_18 VHDL Code

```

-- N = 256, sample RAM 257x18, 9 bit address counter
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_UNSIGNED.all;

entity RAM_S_257_18 is
generic( WIDTH : positive := 9; -- counter width
         DELAY : positive := 64 );-- sample delay of input signal
port
(  RESET, CLK, EN_CNT_S, WE_R_S, EN_X_SHIFT, EN_RESET : in std_logic ;
   DIN          : in std_logic_vector(2*WIDTH -1 downto 0);
   SAMP_FILT    : out std_logic_vector(2*WIDTH -1 downto 0);
   SAMP_LMS     : out std_logic_vector(2*WIDTH -1 downto 0)
);
```

```vhdl
end entity RAM_S_257_18;
architecture ADR_RAM_S of RAM_S_257_18 is

component RAM_SAMP                    -- CoreGenerator RAM modul 257x18 bit
    port ( addr: IN std_logic_VECTOR(8 downto 0);
           clk: IN std_logic;
           din: IN std_logic_vector(17 downto 0);
           dout: OUT std_logic_VECTOR(17 downto 0);
           sinit: IN std_logic;
           we: IN std_logic);
end component;

-- 64x18 bit register chain
type REG_ARRAY is array(DELAY-1 downto 0) of std_logic_vector(2*WIDTH-1 downto 0);

signal ADR_S : std_logic_vector(WIDTH-1 downto 0);
signal X_DELAY : REG_ARRAY := (others=>"000000000000000000");
signal DIN_I_FILT, DIN_I_LMS : std_logic_vector(17 downto 0) := (others=>'0');
constant MAX : std_logic_vector(WIDTH -1 downto 0) := "100000000"; -- 256

begin
ADR_CNT_SAMP: process(CLK)    -- address counter:decrementer
begin
 if (CLK'event and CLK = '1') then
     if (RESET = '1') then
         ADR_S <= (others => '0')after 3 ns;
     elsif (EN_CNT_S = '1') then
         if (ADR_S = 0) then ADR_S <= MAX after 3 ns;
         else ADR_S <= ADR_S - 1 after 3 ns;
         end if;
     end if;
 end if;
end process ADR_CNT_SAMP;

X_SHIFT: process(CLK)
begin
    if (CLK'event and CLK = '1') then
        if (RESET = '1') then
            X_DELAY <= (others=>"000000000000000000") after 3 ns;
        elsif (EN_X_SHIFT = '1') then
            X_DELAY <= X_DELAY(DELAY-2 downto 0) & DIN after 3 ns;  -- shift in register chain
        end if;
    end if;
end process X_SHIFT;

SWITCH_DIN : process(EN_RESET, DIN, X_DELAY)
begin
   if (EN_RESET = '1') then
      DIN_I_FILT <= (others=>'0');  -- switch RAM input for reset
      DIN_I_LMS <= (others=>'0');
   else
      DIN_I_FILT <= DIN;
      DIN_I_LMS <= X_DELAY(DELAY-1);
   end if;
end process SWITCH_DIN;

 -- RAM unit for filter process
 RAM_S_FILT : RAM_SAMP
 port map (addr => ADR_S(8 downto 0),
           clk => CLK,
           din => DIN_I_FILT,
           dout => SAMP_FILT,
           sinit => EN_RESET,
           we => WE_R_S);

 -- RAM unit for adaption process with delayed samples
 RAM_S_LMS : RAM_SAMP
 port map (addr => ADR_S(8 downto 0),
           clk => CLK,
           din => DIN_I_LMS,
           dout => SAMP_LMS,
           sinit => EN_RESET,
           we => WE_R_S);

end architecture ADR_RAM_S;
```

E.2.3 Entity MAC_LMS

Listing 6: MAC_LMS VHDL Code

```vhdl

-- LMS adaption modul, 257x19 bit dual port RAM, 9 bit address counter
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_UNSIGNED.all;

entity MAC_LMS is
generic(WIDTH : positive:= 9);  -- counter width
port
(  CLK, RESET, CLR_REG, EN_REG_EMUE, EN_CNT_CW, EN_CNT_CR, WE_R_C, EN_RESET : in std_logic;
   X          : in std_logic_vector(2*WIDTH -1 downto 0);
   E          : in std_logic_vector(2*WIDTH -1 downto 0);
   MUE        : in std_logic_vector(2*WIDTH -1 downto 0);
   CNT_C_MAX  : out std_logic;
   COEF       : out std_logic_vector(2*WIDTH -1 downto 0)
);
end entity MAC_LMS;

architecture MAC_ADAPT of MAC_LMS is

component DP_RAM_COEF                    -- CoreGenerator RAM modul 257x19 bit
    port (
    addra: IN std_logic_VECTOR(8 downto 0);
    addrb: IN std_logic_VECTOR(8 downto 0);
    clka: IN std_logic;
    clkb: IN std_logic;
    dinb: IN std_logic_VECTOR(18 downto 0);
    douta: OUT std_logic_VECTOR(18 downto 0);
    sinita: IN std_logic;
    web: IN std_logic);
end component;

signal ADR_CW : std_logic_vector(WIDTH-1 downto 0);
signal ADR_CR : std_logic_vector(WIDTH-1 downto 0);
constant MAX  : std_logic_vector(WIDTH -1 downto 0) := "100000000";   -- 256
signal COEF_I : std_logic_vector(2*WIDTH downto 0);  -- adapted coef
signal COEF_O : std_logic_vector(2*WIDTH downto 0);  -- output coef
signal EMUE   : signed(35 downto 0);   -- 36 bit Q34 format
signal XEMUE  : signed(35 downto 0);   -- 36 bit Q33 format
signal COEF_PLUS : signed(2*WIDTH downto 0);

begin

ADRW_CNT_COEF: process(CLK)              -- address counter: incrementer
begin                                    -- write address
 if (CLK'event and CLK = '1') then
     if (RESET = '1') then
         ADR_CW <= (others => '0')after 3 ns;
     elsif (EN_CNT_CW = '1') then
         if (ADR_CW = MAX) then ADR_CW <= (others => '0') after 3 ns;
         else ADR_CW <= ADR_CW + 1 after 3 ns;
         end if;
     end if;
 end if;
end process ADRW_CNT_COEF;

ADRR_CNT_COEF: process(CLK)              -- address counter: incrementer
begin                                    -- read address
 if (CLK'event and CLK = '1') then
     if (RESET = '1') then
         ADR_CR <= (others => '0')after 3 ns;
     elsif (EN_CNT_CR = '1') then
         if (ADR_CR = MAX) then ADR_CR <= (others => '0') after 3 ns;
         else ADR_CR <= ADR_CR + 1 after 3 ns;
         end if;
     end if;
 end if;
end process ADRR_CNT_COEF;

CNT_C_MAX <= '1' after 3 ns when ADR_CR = MAX else '0' after 3 ns;   -- status
```

```
 RAM_C: DP_RAM_COEF
      port map (addra => ADR_CR(8 downto 0),
                addrb => ADR_CW(8 downto 0),
                clka => CLK,
                clkb => CLK,
                dinb => COEF_I,
                douta => COEF_O,
                sinita => EN_RESET,
                web => WE_R_C);


CALC_EMUE : process(CLK)
   begin
       if (CLK = '1' and CLK'event) then
          if (CLR_REG = '1') then  EMUE <= (others => '0') after 3 ns;
          elsif (EN_REG_EMUE = '1') then
             EMUE <= signed(E) * signed(MUE) after 3 ns; -- 36 Bit product
          end if;
       end if;
end process CALC_EMUE;

SWITCH_DIN : process(EN_RESET, COEF_PLUS)
begin
   if (EN_RESET = '1') then
      COEF_I <= (others=>'0'); -- switch RAM input for reset
   else
      COEF_I <= std_logic_vector(COEF_PLUS);
   end if;
end process SWITCH_DIN;

XEMUE <= signed(EMUE(35 downto 18)) * signed(X);    -- 36 bit product
COEF_PLUS <= signed(COEF_O) + XEMUE(35 downto 17); -- adapt coef
COEF <= COEF_O(18 downto 1);

end architecture MAC_ADAPT;
```

E.2.4 Entity MAC_FIR

Listing 7: MAC_FIR VHDL Code

```

-- FIR MAC modul with saturation of the output to 20 Bit signed
-- N = 256, 257 products with max. 1 -> 9 Guard-Bits
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_std.all;

entity MAC_FIR is
generic(WIDTH : positive:= 9);
port
(  CLK, CLR_REG, EN_REG_Y, EN_SAT: in std_logic;
   COEF      : in std_logic_vector(2*WIDTH -1 downto 0); -- 18 bit coefficient
   SAMP      : in std_logic_vector(2*WIDTH -1 downto 0); -- 18 bit sample
   YN          : out std_logic_vector(19 downto 0);        -- 20 bit output signal
   FILT_RDY  : out std_logic                              -- ready pulse to codec interface
 );
end entity MAC_FIR;

architecture MAC_SAT of MAC_FIR is
   signal Y  : signed(43 downto 0);    -- 11 guard bits & 33 bit in Q32 format
begin                                  -- 2 guard bits from coef + 9 additional
MAC : process(CLK)
   begin
       if (CLK = '1' and CLK'event) then
          if (CLR_REG = '1') then  Y <= (others => '0') after 3 ns;
          elsif (EN_REG_Y = '1') then
             Y <= Y +  signed(COEF) * signed(SAMP) after 3 ns;-- 36 Bit Produkt
          end if;                      -- 9 guard bits carry N+1 max. products
       end if;
end process MAC;

SAT: process(CLK)          -- saturation, complementation and transfer register of cycle result
   begin
```

```
    if (CLK = '1' and CLK'event) then
       if (EN_SAT = '1') then
           if (CLR_REG = '1') then YN <= (others => '0') after 3 ns;
           elsif ((Y(43 downto 34) = 0) or (Y(43 downto 34) = -1)) then
              YN <= std_logic_vector(not(Y(34 downto 15)) + 1) after 3 ns;  -- 20 bit codec interface
           elsif ((Y(43) = '0' and Y(42 downto 34) /= "000000000")) then
              YN <= "10000000000000000000" after 3 ns;       -- maxium negative
           elsif ((Y(43) = '1') and (Y(42 downto 34) /= "111111111")) then
              YN <= "01111111111111111111" after 3 ns;       -- max. positive
           end if;
       end if;
       FILT_RDY <= EN_SAT;
    end if;
end process SAT;
end architecture MAC_SAT;
```

E.3 Entity SYSTEM

Listing 8: SYSTEM VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity SYSTEM is
    port ( CLK            : in  std_logic;
           PUSH1          : in  std_logic;
           DIP            : in  std_logic_vector(3 downto 0);
           LED            : out std_logic_vector(3 downto 0);
           BIT_CLK        : in  std_logic;
           SDATA_IN       : in  std_logic;
           SYNC           : out std_logic;
           SATA_OUT      : out std_logic;
           N_RESET_OUT    : out std_logic;
           DISPLAY        : out std_logic_vector(6 downto 0);
           BIT_CLK_TST    : out std_logic;
           SDATA_IN_TST   : out std_logic;
           SYNC_TST       : out std_logic;
           SDATA_OUT_TST  : out std_logic;
           USER_LED       : out std_logic
            );
end SYSTEM;

architecture BEHAVIORAL of SYSTEM is

component CODEC_INTERFACE
    port ( SYNC                   : out std_logic;
           BIT_CLK                : in  std_logic;
           SDATA_OUT              : out std_logic;
           SDATA_IN               : in  std_logic;
           RESET_IN               : in  std_logic;
           AUDIO_IN_READY         : out std_logic;
           AUDIO_OUT_READY        : in  std_logic;
           AUDIO_DATA_IN          : in  std_logic_vector(39 downto 0);
           AUDIO_DATA_OUT_LEFT    : out std_logic_vector(19 downto 0);
           AUDIO_DATA_OUT_RIGHT   : out std_logic_vector(19 downto 0);
           SYNC_TST               : out std_logic;
           SDATA_OUT_TST          : out std_logic
            );
end component ;

component LMS_FIR
    port
    ( CLK, RESET, RD     : in STD_LOGIC;
      XN      : in std_logic_vector(19 downto 0);      -- 20 Bit Codec Interface
      MUE : in std_logic_vector(17 downto 0);
      EN   : in std_logic_vector(19 downto 0);
      YN      : out std_logic_vector(19 downto 0);
      FILT_RDY : out std_logic );
end component;

signal RESET                  : std_logic := '0';
```

```
signal AUDIO_OUT_READY        : std_logic := '0';
signal AUDIO_IN_READY         : std_logic := '0';
signal AUDIO_DATA_IN          : std_logic_vector(39 downto 0) := (others=>'0');
signal AUDIO_DATA_OUT_LEFT    : std_logic_vector(19 downto 0) := (others=>'0');
signal AUDIO_DATA_OUT_RIGHT   : std_logic_vector(19 downto 0) := (others=>'0');
signal Q1                     : std_logic;
signal Q2                     : std_logic;
signal MONO                   : std_logic;
signal MUE                    : std_logic_vector(17 downto 0) := "000000000100000000";   -- MUE = 1/512

signal BIT_CLK_COUNT : std_logic_vector(23 downto 0) := (others=>'0');

begin

-- codec interface
INTERFACE : CODEC_INTERFACE
  PORT MAP(
    SYNC                   => SYNC,
    BIT_CLK                => BIT_CLK,
    SDATA_OUT              => SDATA_OUT,
    SDATA_IN               => SDATA_IN,
    RESET_IN               => RESET,
    AUDIO_IN_READY         => AUDIO_IN_READY,
    AUDIO_OUT_READY        => AUDIO_OUT_READY,
    AUDIO_DATA_IN          => AUDIO_DATA_IN,
    AUDIO_DATA_OUT_LEFT    => AUDIO_DATA_OUT_LEFT,
    AUDIO_DATA_OUT_RIGHT   => AUDIO_DATA_OUT_RIGHT,
    SYNC_TST               => SYNC_TST,
    SDATA_OUT_TST          => SDATA_OUT_TST
  );

-- adaptive LMS-FIR-filter
LMS_FIR_FILTER : LMS_FIR
   port map(
       CLK      => CLK,
       RESET    => RESET,
       RD       => AUDIO_IN_READY,
       XN       => AUDIO_DATA_OUT_LEFT,
       MUE      => MUE,
       EN       => AUDIO_DATA_OUT_RIGHT,
       YN       => AUDIO_DATA_IN(39 downto 20),
       FILT_RDY => AUDIO_OUT_READY
    );

AUDIO_DATA_IN(19 downto 0) <= (others=>'0');
BIT_CLK_TST <= BIT_CLK;
SDATA_IN_TST <= SDATA_IN;

BIT_CLK_DIV: process (BIT_CLK)
  begin
    if (BIT_CLK'event and BIT_CLK = '1') then
        BIT_CLK_COUNT <= BIT_CLK_COUNT + 1;
      end if;
  end process BIT_CLK_DIV;

USER_LED <= bit_clk_count(23);
N_RESET_OUT <= PUSH1;
RESET <= not(PUSH1);

SELECT_MUE: process (DIP)
  begin
    case DIP is
          when "0000" => MUE <= "000000000000000010";  -- MUE = 1/65536
          when "0001" => MUE <= "000000000000000100";  -- MUE = 1/32768
          when "0010" => MUE <= "000000000000001000";  -- MUE = 1/16384
          when "0011" => MUE <= "000000000000010000";  -- MUE = 1/8192
          when "0100" => MUE <= "000000000000100000";  -- MUE = 1/4096
          when "0101" => MUE <= "000000000001000000";  -- MUE = 1/2048
          when "0110" => MUE <= "000000000010000000";  -- MUE = 1/1024
          when "0111" => MUE <= "000000000100000000";  -- MUE = 1/512
          when "1000" => MUE <= "000000001000000000";  -- MUE = 1/256
          when "1001" => MUE <= "000000010000000000";  -- MUE = 1/128
          when "1010" => MUE <= "000000100000000000";  -- MUE = 1/64
          when "1011" => MUE <= "000001000000000000";  -- MUE = 1/32
```

```
            when "1100" => MUE <= "000010000000000000";   -- MUE = 1/16
            when "1101" => MUE <= "000100000000000000";   -- MUE = 1/8
            when "1110" => MUE <= "001000000000000000";   -- MUE = 1/4
            when "1111" => MUE <= "010000000000000000";   -- MUE = 1/2
            when others => MUE <= "000000000000100000";   -- MUE = 1/4096
        end case;
    end process SELECT_MUE;

LED(3) <= DIP(3);
LED(2) <= DIP(2);
LED(1) <= DIP(1);
LED(0) <= DIP(0);

DISPLAY <= "0001000";

end BEHAVIORAL;
```

E.4 Testbench VHDL Code

Listing 9: Testbench VHDL Code

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity testbench is

end;

architecture Behavioral of testbench is

component SYSTEM
    Port ( CLK : in  std_logic;
           PUSH1 : in  std_logic;
           DIP : in  std_logic_vector(3 downto 0);
           LED : out std_logic_vector(3 downto 0);
           BIT_CLK : in std_logic;
           SDATA_IN : in std_logic;
           SYNC : out std_logic;
           SDATA_OUT : out std_logic;
           N_RESET_OUT : out std_logic;
           DISPLAY : out std_logic_vector(6 downto 0);
           BIT_CLK_TST : out std_logic;
           SDATA_IN_TST : out std_logic;
           SYNC_TST : out std_logic;
           SDATA_OUT_TST : out std_logic;
           USER_LED : out std_logic
             );
end component;

-- reference signal ROM
component X_ROM
    port
    (   addr : in std_logic_vector(7 downto 0);
        clk  : in std_logic;
        dout : out std_logic_vector(19 downto 0));
end component;

-- primary signal ROM
component D_ROM
    port
    (   addr : in std_logic_vector(7 downto 0);
        clk  : in std_logic;
        dout : out std_logic_vector(19 downto 0));
end component;

-- z^-30 delay chain
type REG_ARRAY30 is array(29 downto 0) of std_logic_vector(19 downto 0);
-- z^-29 delay chain
type REG_ARRAY29 is array(28 downto 0) of std_logic_vector(19 downto 0);
-- z^-35 delay chain
type REG_ARRAY35 is array(34 downto 0) of std_logic_vector(19 downto 0);
```

```
SIGNAL sync       : std_logic := '0';
SIGNAL CLK        : std_logic := '0';
signal BIT_CLK    : std_logic := '0';
signal PUSH1      : std_logic := '1';
SIGNAL DATA_IN    : std_logic := '0';
SIGNAL DATA_OUT   : std_logic := '0';
SIGNAL reset      : std_logic := '0';
SIGNAL tag_reg    : std_logic_vector(15 downto 0);
signal BIT_CLK_OUT : std_logic;
-- 20 bit register contains data send to the codec interface
signal DATA_REG : std_logic_vector(19 downto 0);
signal START : std_logic := '0';
signal DIP : std_logic_vector(3 downto 0) := "0111"; -- DIP selects MUE = 1/512
signal ADDR_X, ADDR_D : std_logic_vector(7 downto 0) := (others=>'0');
constant MAX : std_logic_vector(7 downto 0) := "10000010"; -- Number of ROM samples = 131
signal X, D, Y : std_logic_vector(19 downto 0) := (others=>'0');
signal E : signed(20 downto 0) := (others => '0');
-- BIT_CLK counter
signal CNT : std_logic_vector(7 downto 0) := (others => '0');
-- signal CNT2 : std_logic_vector(16 downto 0) := (others => '1');

signal X_DELAY : REG_ARRAY30 := (others=>X"00000");
signal Y_DELAY : REG_ARRAY29 := (others=>X"00000");
signal E_DELAY : REG_ARRAY35 := (others=>X"00000");

CONSTANT tag_reg_width   : integer := 16;

BEGIN

   top_entity : System
   PORT MAP(
      CLK            => CLK,
      PUSH1          => PUSH1,
      DIP            => DIP,
      BIT_CLK        => BIT_CLK,
      SDATA_IN       => DATA_IN,
      SYNC           => sync,
      SDATA_OUT      => DATA_OUT
   );

   ROM_X: X_ROM
        port map(addr => ADDR_X,
                 clk => CLK,
                 dout => X);

   ROM_D: D_ROM
        port map(addr => ADDR_D,
                 clk => CLK,
                 dout => D);

   CLOCK : process
   begin
      wait for 10 ns; CLK <= not CLK;
   end process CLOCK;

   bit_clock : PROCESS
   begin
      wait for 40.7 ns; BIT_CLK  <= not BIT_CLK;
   end PROCESS bit_clock;

   stimulus : PROCESS
   begin
      PUSH1  <= '0';
      wait for 20 ns; PUSH1  <= '1';
      wait;
   end PROCESS stimulus;

   send_data : PROCESS(BIT_CLK, PUSH1)
   begin
      if PUSH1 = '0' then
         tag_reg(15)          <=  '1';              -- Codec Ready
         tag_reg(14 downto 3) <= "001100000000";    -- Only Slot 3 and 4 contain valid data
         tag_reg(2 downto 0)  <= "000";             -- Reserved, set to 0
         DATA_REG <= x"00000";
```

```
            DATA_IN <= '0';
            CNT <= (others => '0');
        elsif (BIT_CLK'event and BIT_CLK = '1') then
            CNT <= CNT + 1;
            -- first send error signal sample (right channel)
            if (CNT = 57) then
                DATA_REG <= E_DELAY(34);
            -- send reference signal sample (left channel)
            elsif (CNT = 77) then
                DATA_REG <= X_DELAY(29);
            -- shift TAG to serial line
            elsif sync = '1' then
                tag_reg <= tag_reg(tag_reg_width-2 downto 0)
                              & tag_reg(tag_reg_width-1);
                DATA_IN <= tag_reg(tag_reg_width-1) after 15 ns;
                START <= '1';
            -- shift data value to serial line
            elsif (sync = '0' and START = '1') then
                DATA_REG <= DATA_REG(18 downto 0)
                              & '0';
                DATA_IN <= DATA_REG(19) after 15 ns;
            end if;
            -- calculate error signal
            if (CNT = 80) then
                E <= signed(D(D'left)&D) + signed(Y_DELAY(28)(Y_DELAY(28)'left)&Y_DELAY(28));
            -- increment ROM address counter
            -- and shift in new sample in delay chains
            elsif (CNT = 255) then
                ADDR_X <= ADDR_X + 1;
                ADDR_D <= ADDR_D + 1;
                X_DELAY <= X_DELAY(28 downto 0) & X;
                Y_DELAY <= Y_DELAY(27 downto 0) & Y;
                E_DELAY <= E_DELAY(33 downto 0) & std_logic_vector(E(20 downto 1));
                -- address counter wrap around
                if (ADDR_X = MAX) then
                    ADDR_X <= (others=>'0');
                    ADDR_D <= (others=>'0');
                end if;
            end if;
        end if;
    end PROCESS send_data;

--PUSH : process(BIT_CLK)
--begin
--      if (BIT_CLK'event and BIT_CLK = '1') then
--          if (CNT2 > 122850 and CNT2 <= 131071) then
--              PUSH1 <= '0';
--          else
--              PUSH1 <= '1';
--          end if;
--          CNT2 <= CNT2 + 1;
--      end if;
--end process PUSH;

-- serial receive of filter output sample
RECEIVE_DATA : process(BIT_CLK)
begin
    if (BIT_CLK'event and BIT_CLK = '0') then
        if (CNT >= 59 and CNT < 79) then
            Y <= Y(18 downto 0) & DATA_OUT;
        end if;
    end if;
end process;

end Behavioral;
```

F Die Spartan-3 LC Entwicklungsplattform

Die Spartan-3 LC Platine ist eine Plattform zur Entwicklung von Anwendungen, die auf der Xilinx Spartan-3 FPGA Familie basieret. Auf der Platine befindet sich das Xilinx Spartan-3 FPGA XC3S400-4PQ208CES. Außerdem ist auf dem Board ein Oszillator integriert, der dem FPGA einen $50\ MHz$ Takt zur Verfügung stellt.
Die Platine wird durch das P160 Modul erweitert, um die Verbindung zu einer externen AV-Plattform herzustellen (vgl. Abb. 86). In Abbildung 103 sind die auf der Plattform enthaltenen Komponenten schematisch dargestellt.

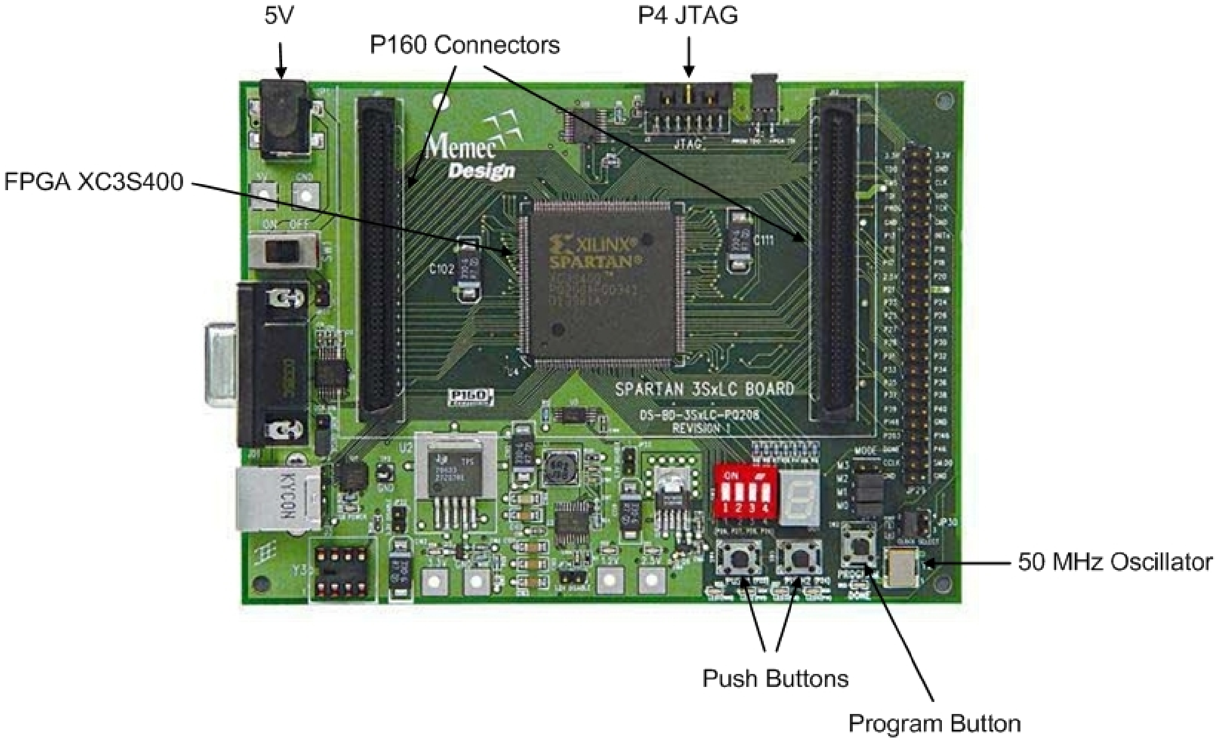

Abb. 86: Spartan-3 LC Entwicklungsplattform [14]

Das Xilinx XC3S400-4PQ208CES FPGA verfügt über folgende Ressourcen:

- 400K System Gatter
- 8064 Logik Zellen[5]
- 896 Konfigurierbare Logik Blöcke (CLBs)
- 56K Verteilte RAM Bits (K=1024)
- 288K Block RAM Bits (K=1024)
- 16 Embedded Multiplizierer
- 4 DCMs
- 141 User I/O Pins
- 62 Unterschiedliche I/O Pairs

[11]

[5] Logik Zelle = 4-input Look-Up-Table (LUT) + DFF

- CLBs enthalten RAM-basierte Look-Up-Tables um Logik- und Speicherelemente zu implementieren, die als Flip-Flops oder Latches verwendet werden können.

- Input/Output Blöcke (IOBs) kontrollieren den Datenfluss zwischen den I/O Pins und der internen Logik. Jede IOB unterstützt bidirektionalen Datenfluss und 3-State Operationen.

- Block RAM ermöglicht Datenspeicherung in Form von 18-Kbit Dual-Port-Blöcken.

- Multiplizierer Blöcke akzeptieren 18-Bit Binärzahlen als Eingangswerte und berechnen das Produkt.

- Digital Clock Manager (DCM) Blöcke unterstützen selbstkalibrierende, voll digitale Lösungen zur Verteilung, Verzögerung, Multiplikation, Teilung und Phasenverschiebung von Taktsignalen.

Ein Ring von IOBs umgibt ein Array von CLBs. Das XC3S400 besitzt zwei in dieses Array eingebettete Spalten Block-RAM. Jede Spalte besteht aus mehreren 18-Kbit RAM Blöcken, von denen jeder mit einem Multiplizierer verbunden ist. Die DCMs sind an den Enden der äußeren Block-RAM Spalten positioniert (vgl. Abb. 87).

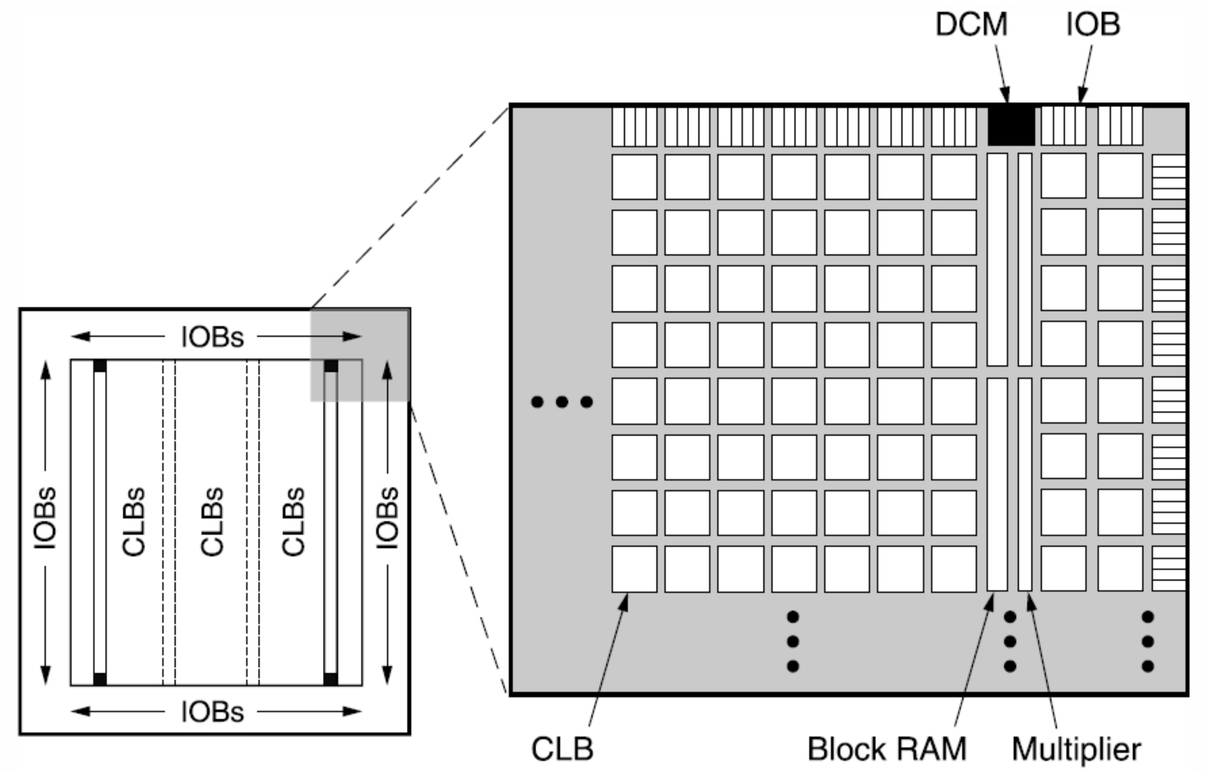

Abb. 87: Spartan-3 FPGA Architektur [25]

Das FPGA wird in diesem Projekt eingesetzt, um digitale Audiosignale zu verarbeiten. Dazu wird ein Interface zum Audio-Codec UCB 1400 (vgl. Kapitel G) sowie ein adaptives FIR-Filter (vgl. Kapitel 5.4) auf dem FPGA implementiert.
Das P160 Prototyp Modul wird über die P160 Konnektoren JX1 und JX2 (vgl. Abb. 86, 103 und 104) mit der Spartan-3 Platine verbunden. In dieser Anwendung wird das Modul genutzt, um die Verbindung zu einer externen Audio-Video-Plattform herzustellen, auf welcher der Audio-Codec UCB 1400 enthalten ist. Diese Verbindung wird über den Konnektor J6 (vgl. Abb. 104 und 105) hergestellt.

G Der Audio-Codec UCB 1400

Der Audio-Codec UCB 1400 von Philips befindet sich auf der AVNET AV-Plattform (vgl. Abb. 88). Es handelt sich um einen Stereo-Audio-Codec, der über ein AC ’97 Rev. 2.1 Interface [17] mit einem externen Prozessor, in diesem Fall dem Spartan-3 FPGA, kommunizieren kann. Der Codec verfügt über einen Stereo Line-Eingang J7 und Ausgang J8, sowie einen mono Mikrofon-Eingang J6 (vgl. Abb. 88 und 106). Die Auflösung eines digitalen Audio-Samples beträgt 20 Bit. Die Sampling-Frequenz ist variabel und wird über Konfigurationsregister eingestellt (Kapitel G.2). In der hier beschriebenen Anwendung wird mit einer Sample-Rate von 48 kHz gearbeitet.

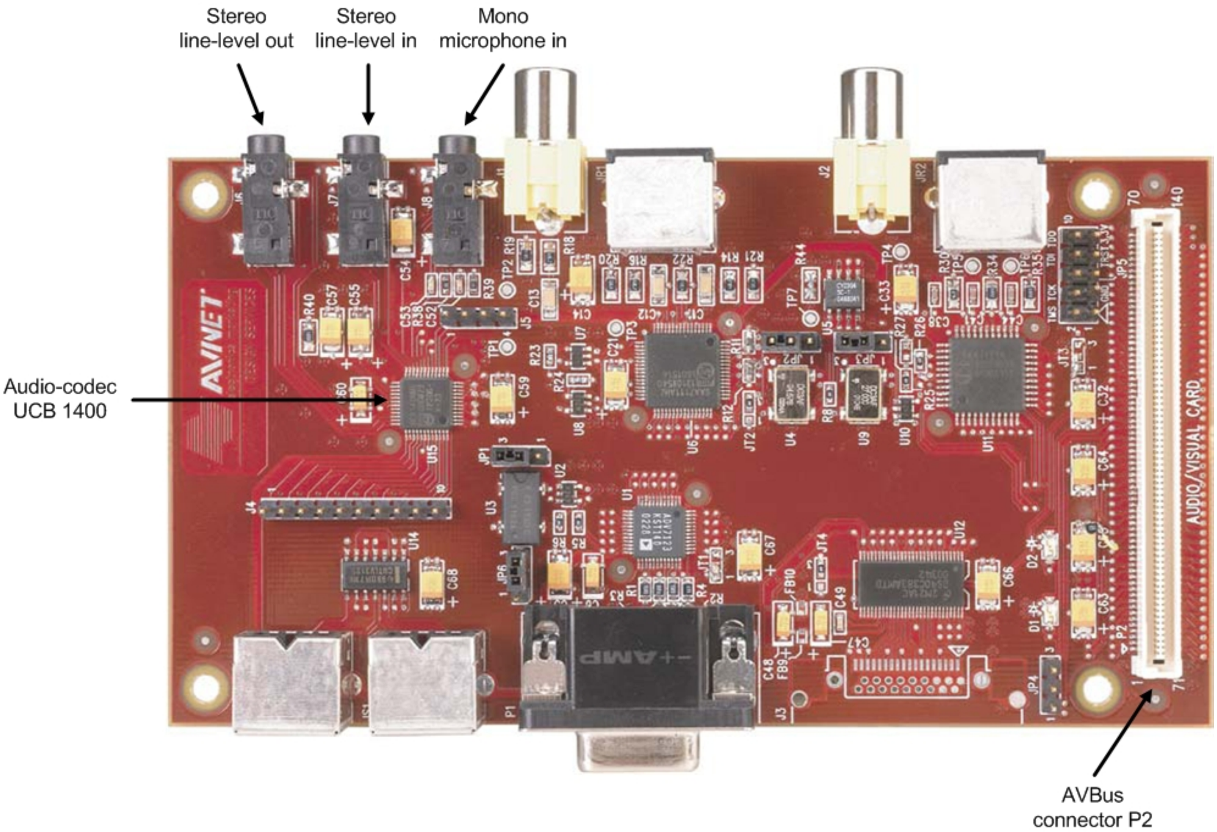

Abb. 88: Audio/Video Modul [2]

Auf der Seite des AV-Moduls wird die Verbindung zur FPGA-Platine über den AVBus Konnektor P2 (vgl. Abb. 88 und 106) hergestellt. Bei dieser Anwendung werden die Verbindungen zum AC ’97 Interface sowie der Line-Ein- und Ausgang verwendet.

G.1 Das AC ’97 Interface

Das AC ’97 Interface stellt eine Schnittstelle zur Kommunikation zwischen dem Audio-Codec und dem FPGA dar, das die Rolle des AC97-Controllers übernimmt. Die Synchronisation erfolgt dabei über die Signale *BIT_CLK* und *SYNC*. Der UCB 1400 leitet intern seinen Takt von einem extern angeschlossenen 24,576 MHz Oszillator ab und stellt den halbierten Takt von 12,288 MHz über die *BIT_CLK*-Leitung als Ausgangssignal bereit.
Der AC97-Controller, welcher durch das FPGA realisiert wird, erhält das *BIT_CLK*-Signal vom Codec und generiert aus der Anzahl der Takte das *SYCN*-Signal. Die Übertragung eines 256 Bit langen Audio-Frames wird synchronisiert über die steigende Flanke des *SYNC*-Signals. Mit jedem *BIT_CLK*-Takt wird ein Bit übertragen.
Während der ersten 16 Bits, was der Länge des TAG-Slots entspricht, ist das *SYNC*-Signal high und für die restlichen 240 Takte low (vgl. Abb. 90). Dies führt zu einem 48 kHz *SYNC*-

Signal dessen Periode einen Audio-Frame definiert. Ein Datenbit wird vom Sender mit jeder steigenden *BIT_CLK* Flanke bereitgestellt und mit jeder fallenden Flanke durch den Empfänger abgetastet.
Daten werden über die seriellen Leitungen *SDATA_OUT* und *SDATA_IN* ausgetauscht, wobei das Senden und Empfangen parallel stattfindet. Über die *SDATA_OUT*-Leitung werden serielle Telegramme vom FPGA zum Audio-Codec gesendet und über die *SDATA_IN*-Leitung in die entgegengesetzte Richtung. Um den Audio-Codec zurückzusetzen, wird die $\overline{RESET}$-Leitung verwendet (Abb. 89).

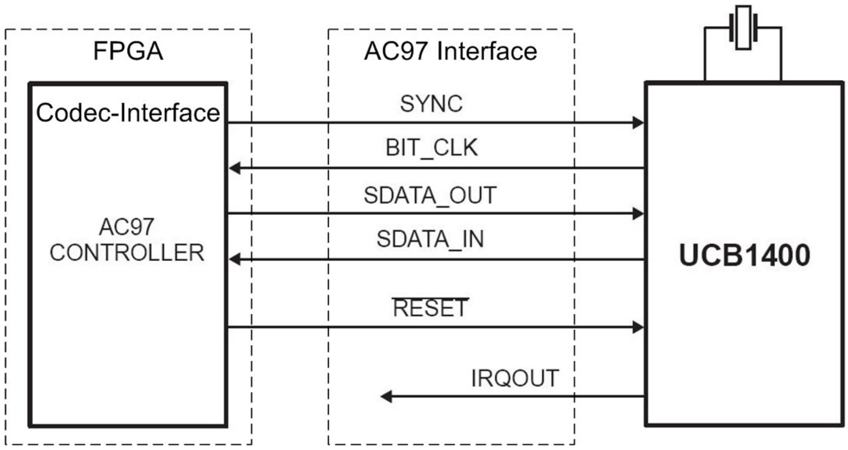

Abb. 89: Das AC '97 Interface als Verbindung zwischen UCB 1400 und FPGA [17]

Ein Audio-Frame ist unterteilt in 13 Slots. Der TAG-Slot hat eine Länge von 16 Bit, wohingegen alle anderen Slots 20 Bit lang sind, was der Gesamtlänge eines Frames von 256 Bit entspricht. Der TAG-Slot wird immer während der *SYNC*-High-Phase übertragen. Für die Dauer der Übertragung der restlichen Slots ist das *SYNC*-Signal low (vlg. Abb. 90). Hier werden nur die ersten fünf Slots erläutert, da die restlichen Slots bei dieser Anwendung keine Bedeutung haben. Die Übertragung beginnt mit dem MSB (Bit 16/Bit 19) eines Slots.

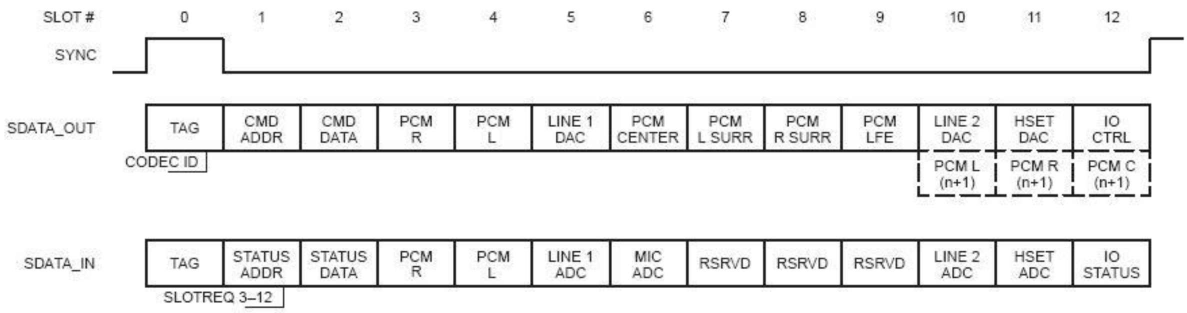

Abb. 90: Bidirektionaler Audio-Frame [17]

G.1.1 SDATA_OUT Frame

Der *SDATA_OUT* Frame wird vom FPGA zum UCB 1400 Audio-Codec übertragen und enthält Konfigurations- und Audiodaten.

- **TAG Slot 0:** Dieser Slot ist der TAG-Slot und enthält Status-Bits. Bit 15 gibt an, ob der Frame gültige Daten enthält. Wenn das Bit eine 1 ist, enthält der Frame mindestens einen Slot mit gültigen Daten. Die nächsten 12 Bits des TAG-Slots geben an, welcher der folgenden 12 Daten-Slots gültige Daten enthält. Die restlichen drei Bits sind in dieser Anwendung immer 0.
  - Bit 15: Valid Frame

  - Bit 14-3: Valid Slot Bits
  - Bit 2-0: 0

- **Command address port Slot 1:** Über den Command Port lassen sich Einstellungen am Codec vornehmen und dessen Status abfragen. Hierzu können 64 16-Bit Register beschrieben bzw. gelesen werden. Dieser Slot enthält 7 Bits mit der Adresse eines Registers, sowie ein Bit welches signalisiert, ob es sich um einen Lese- oder Schreibbefehl handelt. Die restlichen 12 Bits werden mit 0 aufgefüllt.
  - Bit 19: Lese- / Schreibbefehl (1 = lesen, 0 = schreiben)
  - Bit 18-12: Kontroll-Register Adresse
  - Bit 11-0: 0

- **Command data port Slot 2:** Dieser Slot enthält die Daten, die im Falle eines Schreibbefehls in ein Register geschrieben werden sollen. Bei einem Lesebefehl werden alle Bits des Slots mit 0 gefüllt.
  - Bit 19-4: Kontroll-Register Daten
  - Bit 3-0: 0

- **PCM playback right channel Slot 3:** Dieser Slot enthält ein digitales Audio-Sample des rechten Audio-Kanals, das vom AC97 Controller zum Audio-Codec gesendet wird. Die Auflösung des Samples beträgt standardmässig 20-Bit. Bei einer geringeren Auflösung werden die restlichen Bits mit 0 gefüllt.
  - Bit 19-0: Audio-Sample (rechter Kanal)

- **PCM playback left channel Slot 4:** Dieser Slot enthält ein digitales Audio-Sample des linken Audio-Kanals, das vom AC97 Controller zum Audio-Codec gesendet wird. Die Auflösung des Samples beträgt standardmässig 20-Bit. Bei einer geringeren Auflösung werden die restlichen Bits mit 0 gefüllt.
  - Bit 19-0: Audio-Sample (linker Kanal)

Im Falle einer Mono-Konfiguration sind die Daten in Slot 3 und 4 identisch.

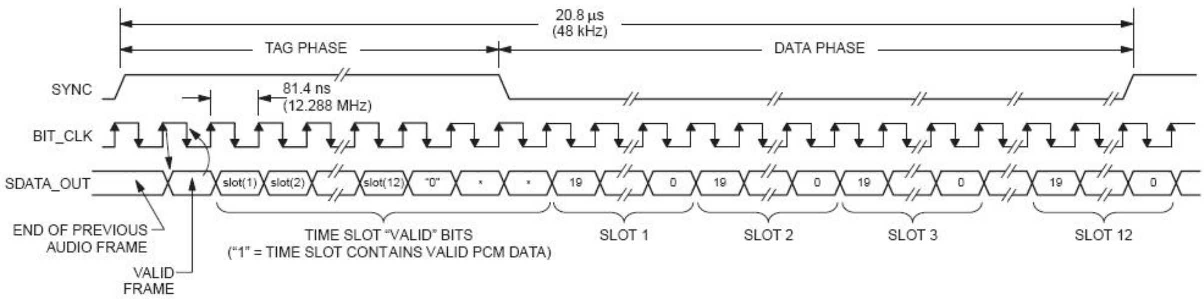

Abb. 91: AC Link Audio Ausgangsframe vom FPGA zum Codec [17]

Ein neuer Audio-Ausgangs-Frame beginnt immer mit dem *SYNC*-Wechsel von low auf high, wobei *SYNC* synchron zur steigenden Flanke von *BIT_CLK* ist. Bei der unmittelbar folgenden, fallenden Flanke von *BIT_CLK* tastet der UCB 1400 das *SYNC*-Signal ab. Dadurch wissen zu diesem Zeitpunkt beide Seiten, der AC97 Controller und der UCB 1400, dass ein neuer Audio-Frame beginnt. Danach taktet der Sender, in diesem Fall der AC97 Controller auf dem FPGA, mit jeder steigenden Flanke von *BIT_CLK* das Sende-Schieberegister und schiebt ein

Bit auf die *SDATA_OUT*-Leitung. Der Empfänger, hier der Audio-Codec, taktet das Empfangs-Schieberegister mit der folgenden, fallenden Flanke und tastet dabei ein Bit von der *SDATA_OUT*-Leitung ab (vgl. Abb. 91 und 92).

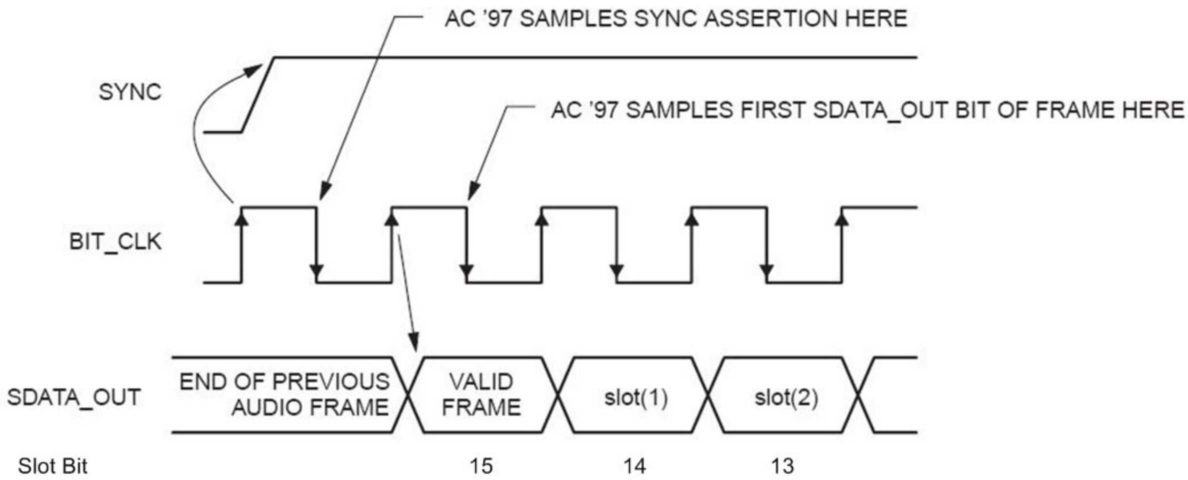

Abb. 92: Start eines Audio Ausgangsframes vom FPGA zum Codec [17]

G.1.2 SDATA_IN Frame

Der *SDATA_IN* Frame wird vom UCB 1400 Audio-Codec zum FPGA übertragen und enthält Status- und Audiodaten.

- **TAG Slot 0:** Dieser Slot ist der TAG-Slot und enthält Status-Bits. Bit 15 gibt an, ob sich der UCB 1400 im *Codec Ready*-Zustand befindet. Wenn das *Codec-Ready*-Bit 0 ist, bedeutet dies, dass der Audio-Codec sich nicht im normalen Betriebszustand befindet. Eine 1 signalisiert, dass der Codec betriebsbereit ist. Die nächsten 12 Bits des TAG-Slots geben an, welcher der folgenden 12 Daten-Slots gültige Daten enthält. Die restlichen drei Bits sind in dieser Anwendung immer 0.
  - Bit 15: Codec Ready
  - Bit 14-3: Valid Slot Bits
  - Bit 2-0: 0
- **Status address port Slot 1:** Der Status Port wird genutzt um die Konfiguration und den Status des Codecs abzufragen. In diesem Slot liefert der Codec die Adresse des Registers, welches über den Command Port vom AC97 Controller abgefragt wurde.
  - Bit 19: 0
  - Bit 18-12: Kontroll-Register Adresse
  - Bit 11-2: *SLOTREQ*-Bits (in dieser Anwendung ohne Bedeutung)
  - Bit 1-0: 0
- **Status data port Slot 2:** Dieser Slot enthält die Daten des Registers, das im Falle eines Lesebefehls vom AC97 Controller über den Command Port abgefragt wurde.
  - Bit 19-4: Kontroll-Register Daten
  - Bit 3-0: 0
- **PCM record right channel Slot 3:** Dieser Slot enthält ein digitales Audio-Sample des rechten Audio-Kanals, das vom Audio-Codec zum AC97 Controller gesendet wird. Die Auflösung des Samples beträgt standardmässig 20-Bit. Bei einer geringeren Auflösung werden die restlichen Bits mit 0 gefüllt.

- Bit 19-0: Audio-Sample (rechter Kanal)

- **PCM record left channel Slot 4:** Dieser Slot enthält ein digitales Audio-Sample des linken Audio-Kanals, das vom Audio-Codec zum AC97 Controller gesendet wird. Die Auflösung des Samples beträgt standardmässig 20-Bit. Bei einer geringeren Auflösung werden die restlichen Bits mit 0 gefüllt.
  - Bit 19-0: Audio-Ausgangs-Sample (linker Kanal)

Im Falle einer Mono-Konfiguration sind die Daten in Slot 3 und 4 identisch.

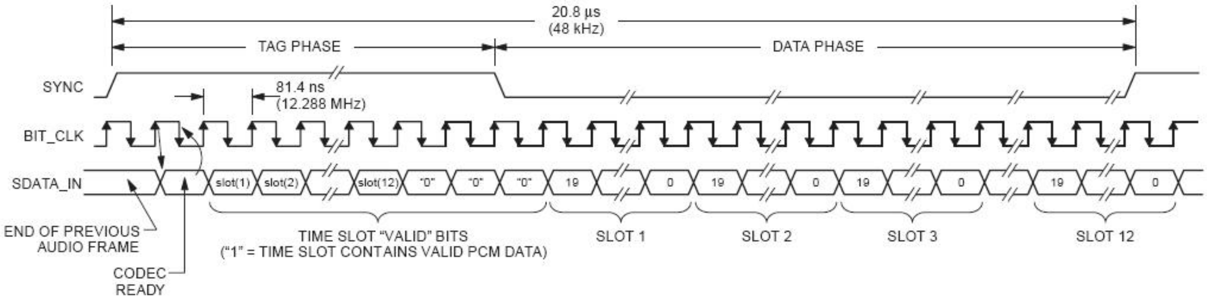

Abb. 93: AC Link Audio Eingangsframe vom Codec zum FPGA [17]

Ein Audio-Eingangs-Frame wird ebenso wie ein Audio-Ausgangs-Frame über das *BIT_CLK*- und das *SYNC*-Signal synchronisiert. Der Sender (UCB 1400) setzt mit jeder steigenden Flanke von *BIT_CLK* ein Bit auf die *SDATA_IN*-Leitung und der Empfänger (AC97 Controller) tastest das Bit mit der folgenden, fallenden Flanke ab (vgl. Abb. 93 und 94).

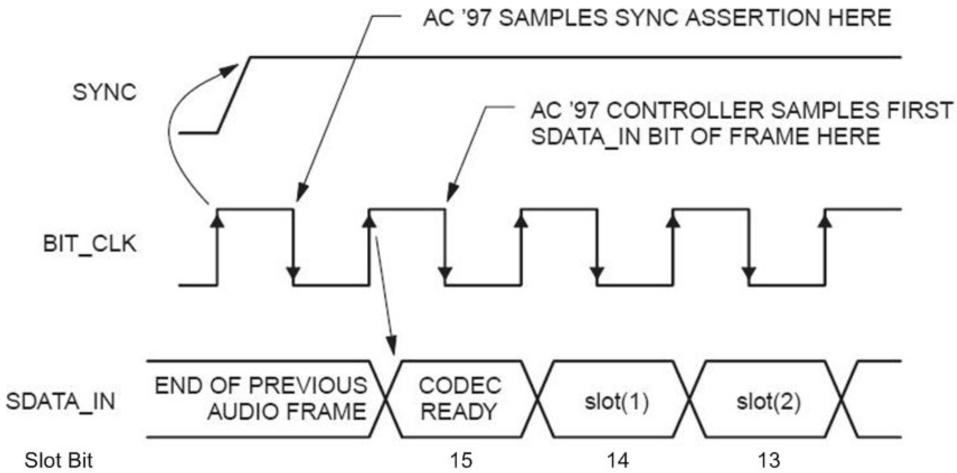

Abb. 94: Start eines Audio Eingangsframes vom Codec zum FPGA [17]

G.2 Konfigurationsregister des Audio-Codecs

Der UCB 1400 verfügt über 64 16-Bit Kontroll-Register, über die der Audio-Codec konfiguriert wird. Für diese Anwendung werden fünf dieser Register beschrieben. Das *Master Volume Register* wird beschrieben, um die *Master Mute* Option auszuschalten. Bei eingeschalteter *Master Mute* Option sind beide Audio-Ausgangskanäle deaktiviert.

Das *Record Select Register* wird beschrieben, um die Audio-Eingänge zu konfigurieren und das *Record Gain Register* um die *Master Record Mute* Option auszuschalten. Bei eingeschalteter *Master Record Mute* Option sind beide Audio-Eingangskanäle deaktiviert. Um den *Headphone Driver* einzuschalten, wird das *Feature Control/Status Register 1* beschrieben.

Der *Headphone Driver* steuert einen Referenz-Ausgang, der als virtuelle Masse dient. Dadurch kann beispielsweise ein Stereo-Kopfhörer angeschlossen werden, ohne dass externe Kondensatoren zur Sperrung des Gleichspannungs-Anteils benötigt werden. Außerdem wird im Falle eines Resets das *Reset Register* beschrieben um alle Register auf ihren Standardwert zurückzusetzen.

- **Reset Register**
  Register Adresse: 0x00; Standardwert: 0x02A0
  Das Schreiben eines beliebigen Wertes in das Register führt zu einem Register-Reset bei dem alle Register auf ihre Standardwerte zurückgesetzt werden. In dieser Anwendung wird das Register bei einem Reset mit 0x0000 beschrieben.

- **Master Volume Register**
  Register Adresse: 0x02; Standardwert: 0x8000
  Das Register wird mit 0x0000 beschrieben, um die *Master Mute* Option auszuschalten.

- **Record Select Register**
  Register Adresse: 0x1A; Standardwert: 0x0000
  Das Register wird mit 0x0404 beschrieben, um den Line-In-L-Eingang als Quelle für den linken Kanal auszuwählen und als Quelle für den rechten Kanal den Line-In-R-Eingang.

- **Record Gain Register**
  Register Adresse: 0x1C; Standardwert: 0x8000
  Das Register wird mit 0x0000 beschrieben, um die *Master Record Mute* Option auszuschalten.

- **Feature Control/Status Register 1**
  Register Adresse: 0x6A; Standardwert: 0x0000
  Das Register wird mit 0x0040 beschrieben, um den *Headphone Driver* einzuschalten.

H Prozesse des VHDL-Codec-Interfaces

H.1 Serieller Datenempfang des FPGAs vom Audio-Codec

***SHIFT_DATA_IN*:**

Der *SHIFT_DATA_IN*-Prozess tastet auf der fallenden *BIT_CLK*-Flanke die *SDATA_IN*-Leitung ab, auf der die Daten vom Audio-Codec zum FPGA seriell übertragen werden. In Abhängigkeit der Enable-Signale für die Eingangs-Register werden die Daten über den seriellen Eingang in die Register geschoben (vgl. Abb. 95 und Code S. 102 Z. 280-307).
Es gibt vier Eingangs-Schiebregister:

- *TAG_REG_IN* (16-Bit-Register für den TAG-Slot Slot 0)
- *ADR_REG_IN* (20-Bit-Register für die Kontroll-Register Adresse Slot 1)
- *DATA_REG_IN* (20-Bit-Register für die Kontroll-Register Daten Slot 2)
- *AUDIO_REG_IN* (40-Bit-Register für die Audio Daten Slot 3 und 4)

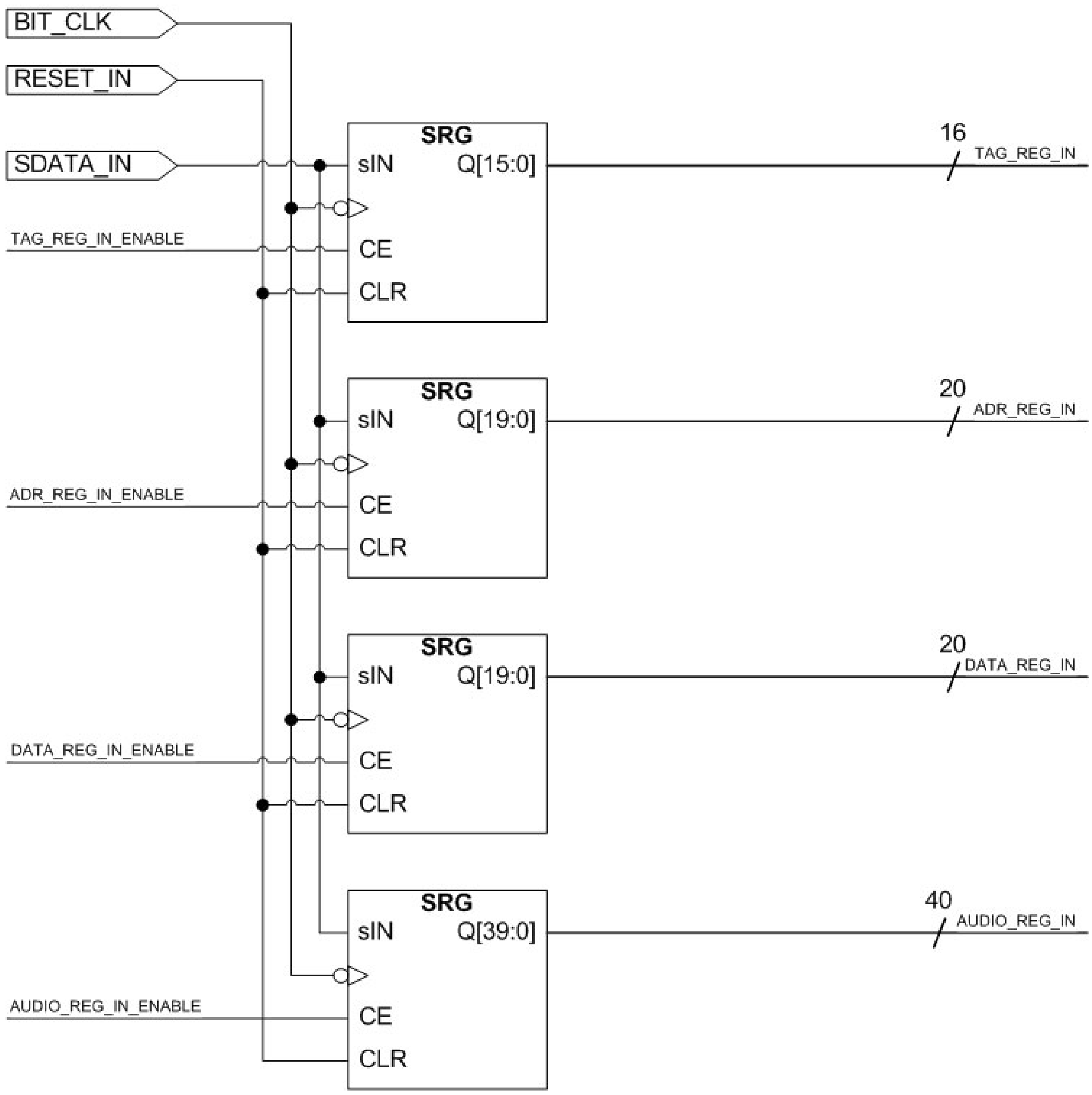

Abb. 95: SHIFT_DATA_IN Prozess schiebt anhand der Enable-Signale die seriellen Bits von der SDATA_IN-Leitung in die Eingangsregister. Das Schieben erfolgt auf der negativen Taktflanke (vgl. Code S. 102 Z. 280-307).

H.2 Serielles Senden des FPGAs zum Audio-Codec

***SHIFT_DATA_OUT*:**

Der *SHIFT_DATA_OUT*-Prozess übernimmt die Rolle eines Multiplexers, der die *SDATA_OUT*-Leitung, auf der die Daten vom FPGA zum Audio-Codec seriell übertragen werden, mit dem seriellen Ausgang des aktiven Ausgangs-Register verbindet. Wenn das Enable-Signal eines Registers gesetzt ist, wird das Bit, welches am seriellen Ausgang des Registers anliegt, mit der steigenden *BIT_CLK*-Flanke auf die *SDATA_OUT*-Leitung gesetzt. So wird sichergestellt, dass die Ausgangsdaten zum richtigen Zeitpunkt an den Audio-Codec gesendet werden (vgl. Abb. 96 und Code S. 103 Z. 314-341).

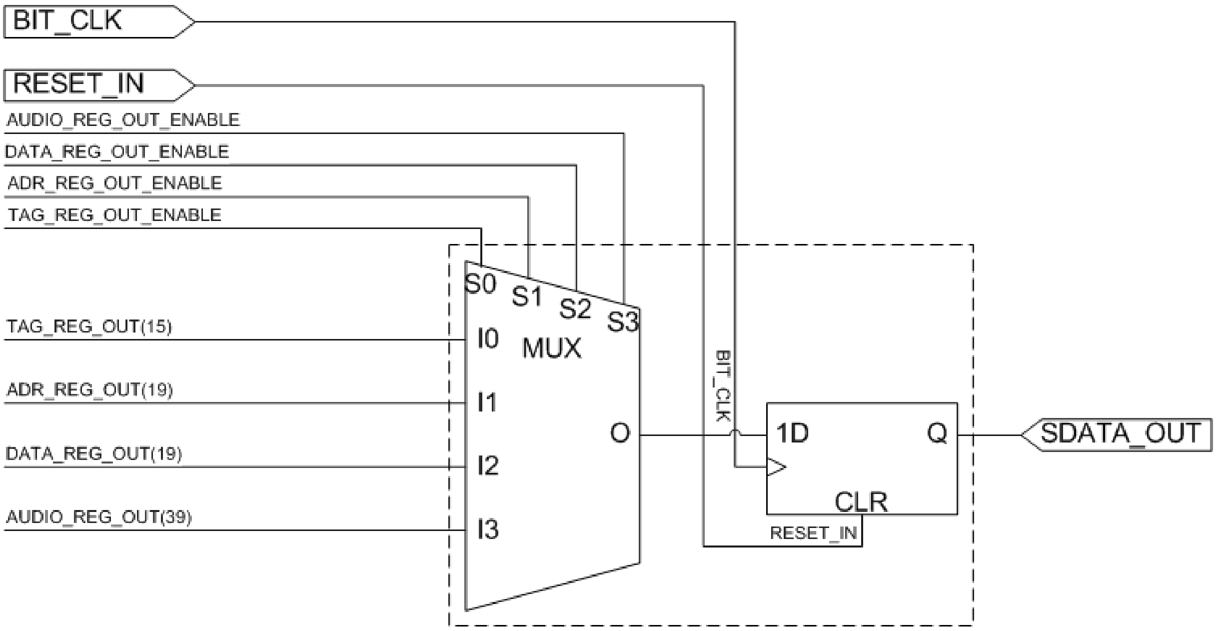

Abb. 96: SHIFT_DATA_OUT Prozess enthält einen Multiplexer, der über die Enable-Signale der Ausgangsregister gesteuert wird, um den jeweils aktiven Ausgang eines Registers mit der seriellen Ausgangsleitung SDATA_OUT zu verbinden (vgl. Code S. 103 Z. 314-341).

Analog zu den Eingangs-Registern gibt es vier Ausgangs-Register:

- TAG-Out-Register (16-Bit-Register für den TAG-Slot Slot 0)
- ADR-Out-Register (20-Bit-Register für die Kontroll-Register Adresse Slot 1)
- DATA-Out-Register (20-Bit-Register für die Kontroll-Register Daten Slot 2)
- AUDIO-Out-Register (40-Bit-Register für die Audio Daten Slot 3 und 4)

H.3 Paralleles Laden der Ausgangsschieberegister

***LOAD_OUT_REGISTERS*:**

Im *LOAD_OUT_REGISTERS*-Prozess findet das Schieben und Laden der Ausgangsregister statt. Aus dem Anlaufphasen-Zähler *INIT_SEQ* werden die Ladesignale für die Register *TAG_REG_OUT*, *ADR_REG_OUT* und *DATA_REG_OUT* dekodiert. In der Anlaufphase bestehend aus den ersten fünf Datenframes werden die drei Register geladen, um die in Anhang G.2 beschriebenen Konfigurationsregister des Codecs zu beschreiben. In den ersten vier Frames wird das *TAG_REG_OUT*-Register mit 0xE000 geladen, um dem Codec zu signalisieren, dass in in Slot 1 und 2 gültige Daten enthalten sind. Das *ADR_REG_OUT*- und das *DATA_REG_OUT*-Register werden in dieser Phase mit den in Anhang G.2 erläuterten Werten geladen, um nacheinander die gewünschten Konfigurationsregister zu beschreiben. Im fünften

Frame der Anlaufphase wird das *TAG_REG_OUT*-Register mit 0x9800 geladen. Dadurch wird dem Codec mitgeteilt, dass ab diesem Zeitpunkt nur noch Audiodaten in den Frames enthalten sind.

Das *AUDIO_REG_OUT*-Register wird durch das *AUDIO_OUT_READY*-Signal geladen. Das Signal ist ein Impuls, der vom adaptiven FIR-Filter nach der Filterung eines Samples erzeugt wird und über einen Impulsverlängerer [23] zu einem Impuls mit der Länge einer *BIT_CLK*-Periode gemacht wird. Wenn dieser Impuls registriert wird, lädt der Prozess das *AUDIO_REG_OUT*-Register mit den gefilterten Samples vom linken und vom rechten Audio-Kanal.

Das Schieben der Werte in den Registern erfolgt auf der steigenden *BIT_CLK*-Flanke, um die Bits aus den Registern auf dieser Flanke an den Codec senden zu können. Aktiviert wird der Schiebevorgang eines Registers durch das zugehörige Enable-Signal. Dies geschieht während der Übertragung des jeweiligen Slots zum Audio-Codec. Auf diese Weise werden der Tag-Slot, der Address-Slot, der Data-Slot und die beiden Audio-Slots seriell zum Codec gesendet.

Das *TAG_REG_OUT*-, *ADR_REG_OUT*- und *DATA_REG_OUT*-Register rotieren während des Schiebens. Dadurch werden über die seriellen Eingänge der Register die Ausgangswerte wieder hergestellt (vgl. Abb. 97 und Code S. 104 Z. 381-391). Die Rotation wird durchgeführt, da sich die Werte in den Registern nach der Anlaufphase nicht mehr ändern und so die Ladevorgänge der Register eingespart werden.

Der Prozess ist so realisiert, dass das Schieben Priorität hat vor dem Laden (vgl. Code S. 104 Z. 381-395). Dies ist notwendig, um das Laden der Register während des Schiebezyklus zu unterbinden. Da das *TAG_REG_OUT*-, das *ADR_REG_OUT*- und das *DATA_REG_OUT*-Register während der Anlaufphase in Abhängigkeit vom Anlaufphasen-Zähler *INIT_SEQ* geladen werden, würde andernfalls das Schieben in dieser Phase unterdrückt werden.

H.4 Erzeugung des Signals zur Synchronisation zwischen FPGA und Audio-Codec

***GEN_SYNC*:**

Der *GEN_SYNC*-Prozess erzeugt das Signal zur Synchronisierung des Datenaustauschs zwischen FPGA und Audio-Codec. Dazu zählt er über einen internen 8-Bit-Zähler die *BIT_CLK*-Takte. Solange der Zähler kleiner als 16 ist, wird das *SYNC*-Signal auf 1 gesetzt und für die restlichen Zählerstände auf 0. Nach 256 Takten gibt es einen *Wrap-Around* des Zählers, so dass dieser wieder bei 0 beginnt (vgl. Abb. 98) und Code S. 100 Z. 81-97). Dadurch wird gewährleistet, dass das *SYNC*-Signal für 16 Takte, die den TAG-Slot und damit den Anfang eines Daten-Frames bilden, 1 ist und für die restlichen 240 Takte 0.

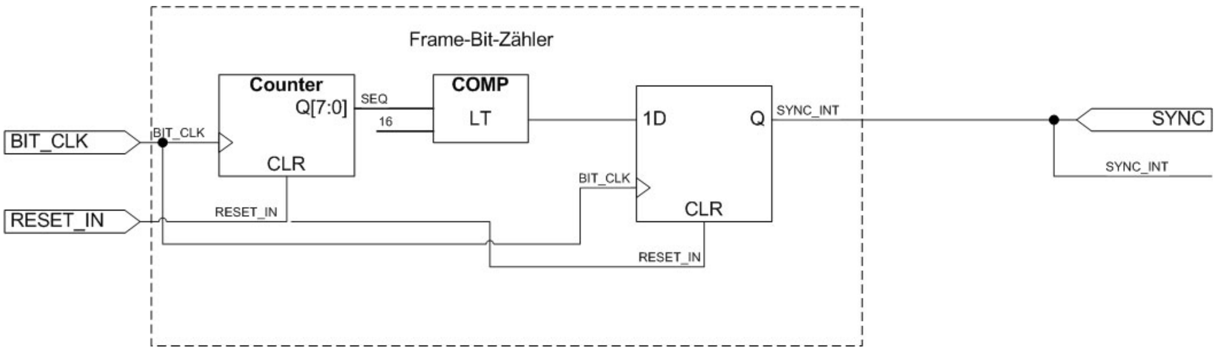

Abb. 98: GEN_SYNC Prozess bestehend aus einem Framzähler, der die Bits eines Frames zählt, und einem Framezähler, der die ersten vier Frames für das Anlaufverhalten des Codec-Interfaces zählt. (vgl. Code S.100)

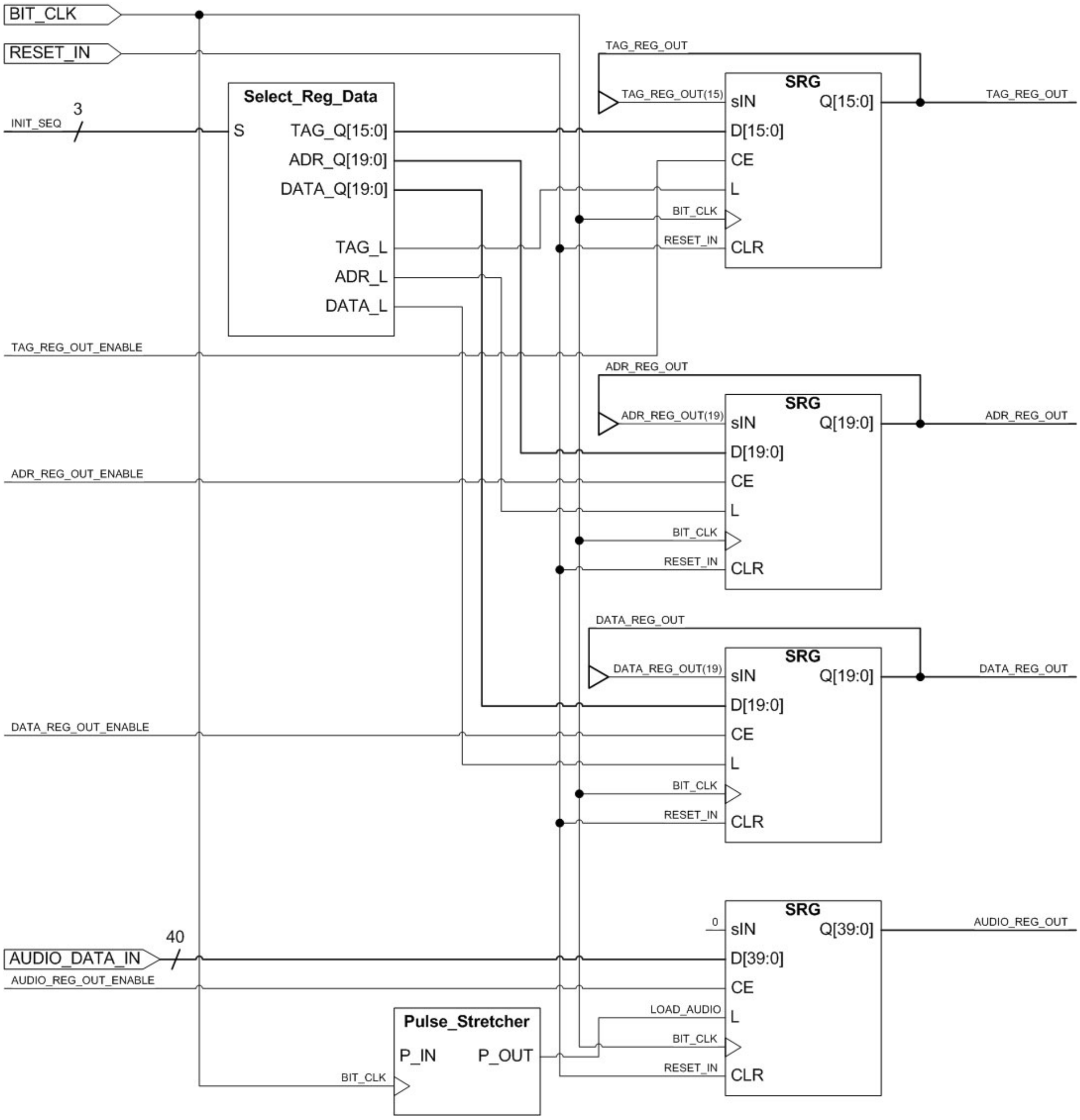

Abb. 97: LOAD_OUT_REGISTERS Prozess enthält einen Decoder, der anhand des Anlaufzählers INIT_SEQ die Ausgangsschieberegister mit den zum Anlaufverhalten passenden Daten lädt. Nach der Anlaufphase werden anhand der Enable-Signale die Schiebevorgänge der Ausgangsregister aktiviert. Über einen Pulsverlängerer [23] wird der Ready-Impuls des adaptiven Filters zum Ladeimpuls des Audio-Ausgangsregisters verlängert (vgl. Code S. 104 Z. 369-433).

H.5 Bestimmung der Position innerhalb des seriellen Telegramms zur Steuerung der Sende- und Empfangsregister

***COUNT_SLOTS*:**

Der *COUNT_SLOTS*-Prozess zählt die Slots eines Datenframes, die Bits eines Slots und die ersten sechs Frames nach einem Systemstart oder einem Reset, aus denen sich die Anlaufphase des Interfaces zusammensetzt. Der *SLOT_COUNT*-Zähler ist ein 4-Bit-Zähler, der die 13 Slots zählt, aus denen sich ein Frame zusammmen setzt. Ein Slot besteht im Fall des Tag-Slots aus 16 Bits und im Fall eines der 12 Datenslots aus 20 Bits. Diese Bits werden von dem 5-Bit-Zähler *SLOT_BIT_COUNT* gezählt. Der 3-Bit-Zähler *INIT_SEQ* zählt die Frames der Anlaufphase.

Alle Zähler starten mit dem Wert 0. Daher werden der *SLOT_COUNT*-Zähler und der *SLOT_BIT_COUNT*-Zähler mit diesem Wert geladen, wenn die jeweilige Ladebedingung erfüllt ist. Die Bedingung ist beim *SLOT_COUNT*-Zähler immer dann erfüllt, wenn das *SYNC*-Signal 1 ist. Zu diesem Zeitpunkt beginnt ein neuer Frame mit Slot 0. Während der gesamten High-Phase des *SYNC*-Signals wird dieser Slot übertragen, daher kann der Zähler während der gesamten Phase mit 0 geladen werden.

Erhöht wird der Wert des *SLOT_COUNT*-Zählers, wenn am CE-Eingang eine 1 anliegt. Dies geschieht unter zwei Bedingungen. In beiden Fällen muss das *SYNC*-Signal 0 sein, da der Zähler bei *SYNC* gleich 1 mit 0 geladen wird. Wenn der *SLOT_COUNT*-Zähler bei 0 steht und der *SLOT_BIT_COUNT*-Zähler den Wert 15 erreicht hat, wird der *SLOT_COUNT*-Zähler um 1 erhöht, da in diesem Fall das Ende des Tag-Slots erreicht hat. Im anderen Fall hat der *SLOT_BIT_COUNT*-Zähler den Wert 19 und damit das Ende eines der 12 Datenslots erreicht.

Wenn eine dieser Situationen eintritt, wird der *SLOT_BIT_COUNT*-Zähler mit 0 geladen, da das Ende eines Slots erreicht wurde. Da das Ende des letzten Slots eines Frames noch in die nächste High-Phase des *SYNC*-Signals hinein reicht (vgl. Anhang G.1.1 Abb. 91), besteht eine weitere Ladebedingung des Zählers bei *SYNC* gleich 1 und einem *SLOT_BIT_COUNT*-Zählerstand von 19, was dem Ende von Slot 12 entspricht.

Der Zähler soll jedes Bit eines Frames zählen und wird daher sowohl in der High-Phase als auch in der Low-Phase des *SYNC*-Signals aktiviert. Der Zählvorgang wird lediglich unterbrochen, wenn eine der Ladebedingungen erfüllt ist. Das Aktivierungssignal wird daher unter zwei Bedingungen gesetzt. Die erste ist erfüllt, wenn das *SYNC*-Signal 1 ist und der *SLOT_BIT_COUNT*-Zähler nicht den Wert 19 hat. Die zweite trifft zu, wenn das *SYNC*-Signal 0 ist, der *SLOT_BIT_COUNT*-Zählerstand nicht 19 ist und die Zählerstände von *SLOT_COUNT* und *SLOT_BIT_COUNT* nicht die Kombination von 0 und 15 annehmen.

Der *INIT_SEQ*-Zähler zählt sechs Frames nach einem Reset oder Systemstart und bleibt dann auf diesem Wert stehen. Der sechste Frame wird gezählt, da die Anlaufphase ab diesem Frame beendet ist und die Ausgangsregister nicht mehr geladen werden müssen. Aktiviert wird dieser Zähler immer bei Bit 0 von Slot 5, solange er selbst einen Wert hat, der kleiner als 6 ist. Da ab Slot 5 keine relevanten Daten mehr gesendet werden, wird auf diese Weise der Wert des Zählers, von dem das Laden der Ausgangsregister abhängig ist, erst dann erhöht, wenn die Phase, in der die Werte in den Ausgangsregistern geschoben werden, beendet ist (vgl. Abb. 99 und Code S. 102 Z. 236-276).

H.6 Steuerung der Eingangsschieberegister

***CONTROL_IN_REGISTERS*:**

Der *CONTROL_IN_REGISTERS*-Prozess steuert die Enable-Signale für die Eingangs-Schieberegister *TAG_REG_IN*, *ADR_REG_IN*, *DATA_REG_IN* und *AUDIO_REG_IN* anhand der Slot Zählerstände und des *SYNC*-Signals. Nach dem erfolgreichen Empfang eines Audio-Samples pro Kanal generiert er einen Startimpuls für das adaptive Filter. Alle Abfragen und Signalzuweisungen des Prozesses liegen in einem getakteten Rahmen, der auf die steigende

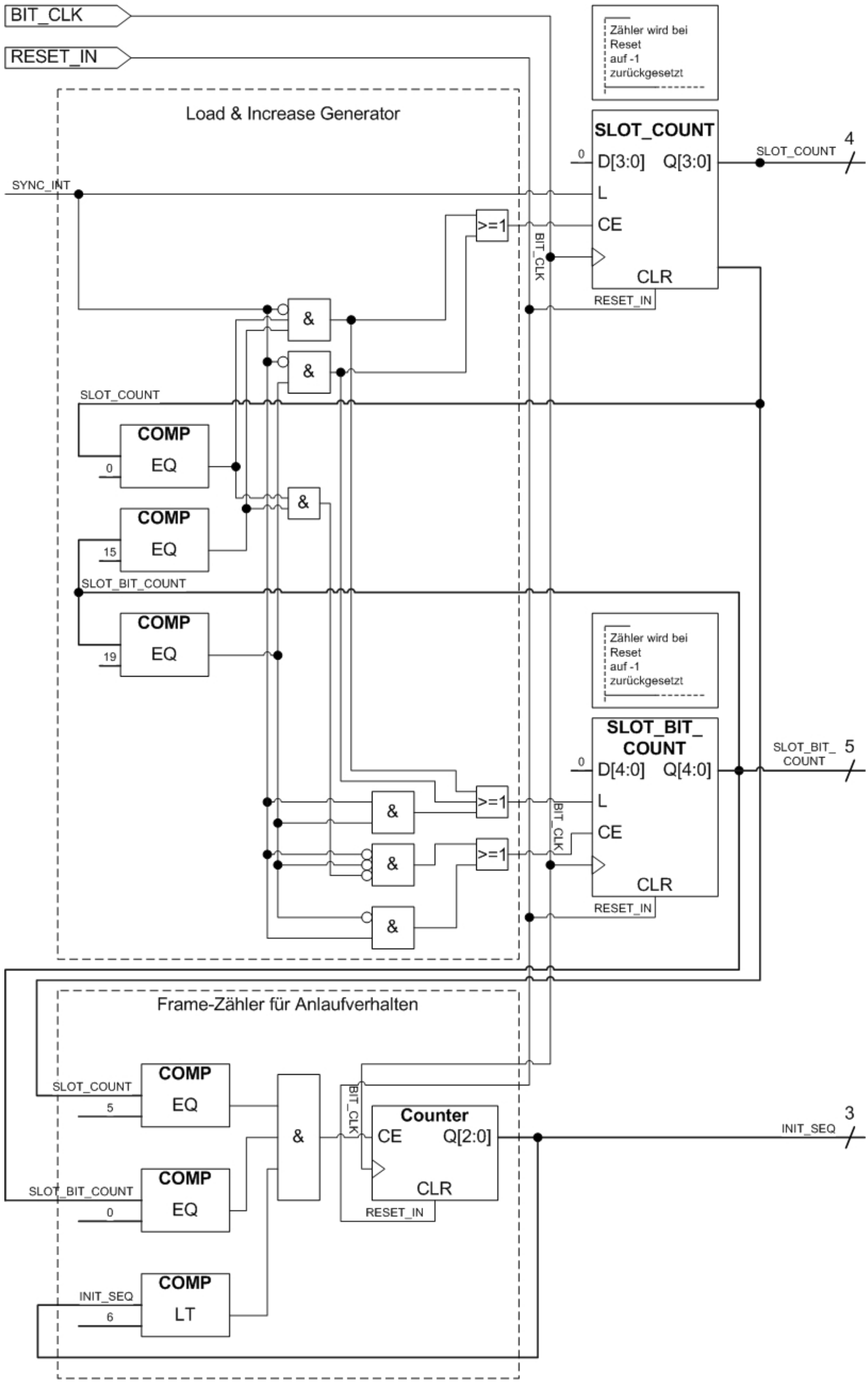

Abb. 99: COUNT_SLOTS Prozess bestehend aus einem Zähler für die Slots eines Frames SLOT_COUNT, einem Zähler für die Bits eines Slots SLOT_BIT_COUNT und einem Zähler für die Frames der Anlaufphase des Interfaces INIT_SEQ (vgl. Code S. 102 Z. 236-276)

BIT_CLK-Flanke reagiert. Dadurch werden die Eingansregister auf der steigenden Flanke freigegeben und sind somit bei der nächsten fallenden Flanke bereit, Daten aufzunehmen. Dies ist notwendig, da die Daten auf der seriellen Eingangsleitung *SDATA_IN* immer mit der fallenden *BIT_CLK*-Flanke abgetastet werden (vgl. Anhang G.1.2).
Das TAG-In-Register wird aktiviert, so lange das *SYNC*-Signal 1 ist, um die Daten des TAG-Slots aufzunehmen. Während der restlichen Zeit bleibt das Register deaktiviert.
Das Enable-Signal für das ADR-In-Register wird auf 1 gesetzt, wenn das letzte Bit des TAG-Slots gezählt wurde. Da die Signalzuweisung in einem getakteten Rahmen geschieht, liegt das gesetzte Enable-Signal erst beim nächsten Takt, und damit pünktlich zum Beginn des ADR-Slots, am Register an. Beim letzten Bit des ADR-Slots wird das Signal dann wieder auf 0 gesetzt.
Analog dazu wird das Enbale-Signal für das DATA-In-Register beim letzten Bit des ADR-Slots auf 1 und am Ende des DATA-Slots wieder auf 0 gesetzt.
Das AUDIO-In-Register muss den Inhalt von zwei Slots aufnehmen. Slot 3 enthält ein Audiosample des linken Kanals und Slot 4 enthält ein Sample des rechten Kanals. Daher wird das Enable-Signal am Ende des DATA-Slots auf 1 gesetzt und bleibt bis zum Ende von Slot 4 auf diesem Wert.
Wenn die Slot Zählerstände die Kombination von Slot 5 Bit 0 erreicht haben, wird das *AUDIO_IN_READY*-Signal für eine *BIT_CLK*-Periode gesetzt, um zu signalisieren, dass der Empfang der Audio-Daten abgeschlossen ist. Dieses Signal wird genutzt, um das Filter zu aktivieren. An dieser Stelle wird kontrolliert, ob das *Codec-Ready*-Bit (*TAG_REG_IN* Bit 15) und die *Valid-Slot-Bits* für die Audio-Slots (*TAG_REG_IN* Bit 12 und 11) gesetzt sind (vgl. Abb. 100 und Code S. 100 Z. 105-163). Wenn alle drei Bits gesetzt sind, wurden gültige Audiodaten vom Codec empfangen. Zuvor werden beim Empfang der seriellen Daten die *Valid-Slot-Bits* im TAG-In-Register nicht ausgewertet, da diese Daten erst am Ende von Slot 0 vollständig vorhanden sind. Zu diesem Zeitpunkt muss bereits das nächste Eingangs-Register aktiviert werden. Daher werden zunächst alle Eingangsdaten in die zugehörigen Register geschoben und erst nach dem Empfang aller relevanten Daten wird geprüft, ob die Audiodaten gültig sind.

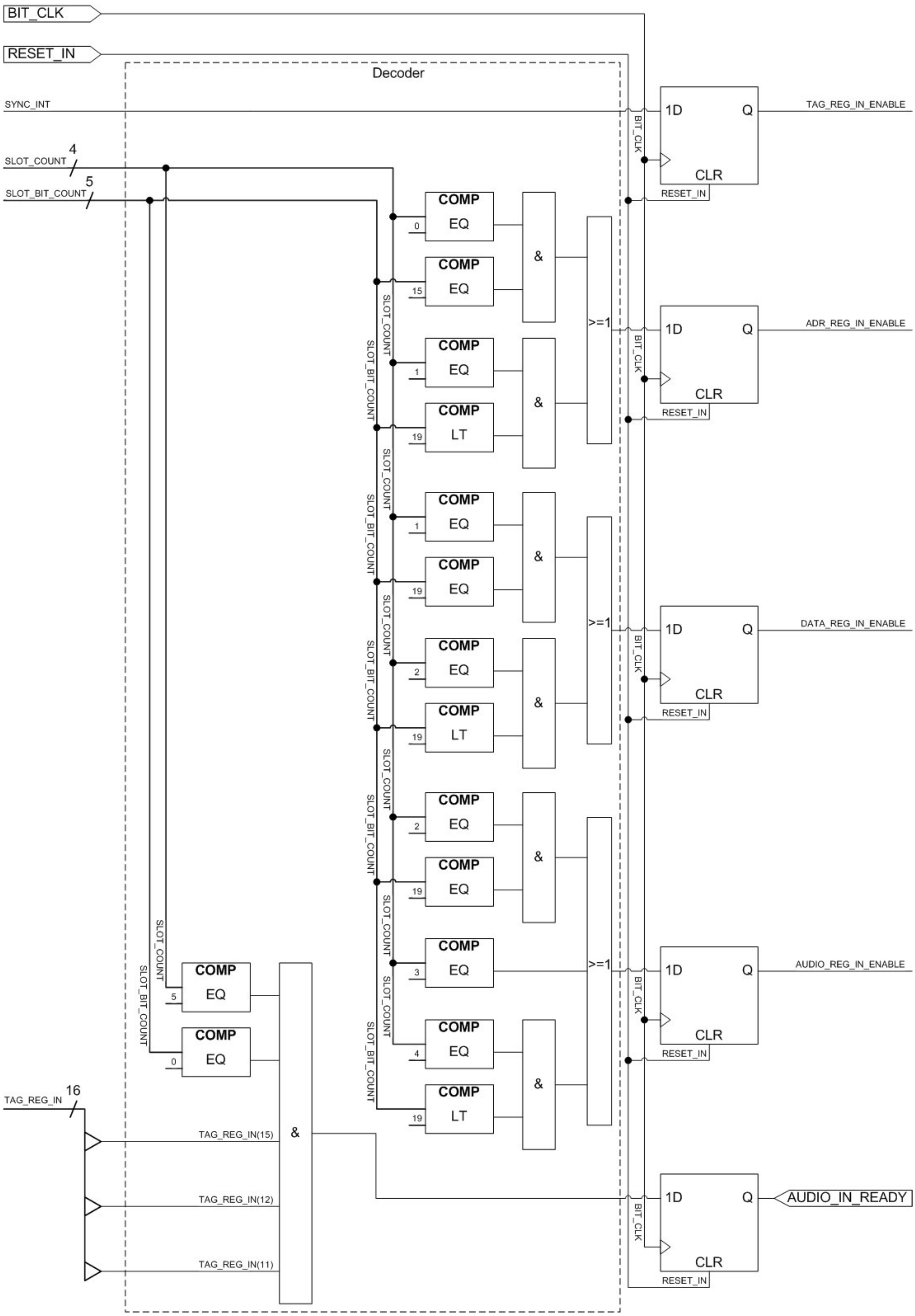

Abb. 100: CONTROL_IN_REGISTERS Prozess bestehend aus einem Decoder, der anhand der Slot-Zählerstände die Eingangsschieberegister aktiviert und einen Ready-Impuls AUDIO_IN_READY für das adaptive Filter erzeugt. Die Schiebefreigabe erfolgt auf der positiven Taktflanke (vgl. Code S. 100 Z. 105-163).

H.7 Steuerung der Ausgangsschieberegister

***CONTROL_OUT_REGISTERS*:**

Der *CONTROL_OUT_REGSITERS*-Prozess steuert die Enable-Signale für die Ausgangs-Schieberegister *TAG_REG_OUT*, *ADR_REG_OUT*, *DATA_REG_OUT* und *AUDIO_REG_OUT* ebenfalls anhand der Slot Zählerstände und des *SYNC*-Signals. Allerdings wird hier auf die fallende Flanke getriggert, da die Ausgangs-Register bei der steigenden Flanke jeweils ein Bit auf die serielle Ausgangsleitung *SDATA_OUT* setzen müssen (vgl. Anhang G.1.1).

Der Prozess aktiviert die Ausgangsregister bei den gleichen Zählerständen, bei denen auch die Eingangsregister aktiviert werden. Da die Daten, die im *TAG_REG_OUT*-Register stehen, von vornherein bekannt sind, wird in diesem Prozess vor jeder Aktivierung eines Registers geprüft, ob der Frame gültige Daten enthält und ob der jeweilige Slot des Frames relevante Daten enthält. Dies geschieht anhand des *Valid-Frame-Bit* (*TAG_REG_OUT* Bit 15) und der *Valid-Slot-Bits* (*TAG_REG_OUT* Bit 14-11). Nur wenn das *Valid-Slot-Bit* eines Slots gesetzt ist, wird das zugehörige Ausgangsregister freigegeben. Dadurch wird sichergestellt, dass keine ungültigen Daten zum Codec gesendet werden.

Das *TAG_REG_OUT*-Register wird in jedem Fall freigegeben, da es die Informationen enthält, die dem Codec mitteilen, welche Slots des aktuellen Frames gültige Daten enthalten.

Das *ADR_REG_OUT*-Register wird freigegeben, wenn das *Valid-Slot-Bit* für den ADR-Slot (*TAG_REG_OUT* Bit 14) gesetzt ist.

Für die Freigabe des *DATA_REG_OUT*-Registers wird das *Valid-Slot-Bit* für den DATA-Slot (*TAG_REG_OUT* Bit 13) überprüft.

Das *AUDIO_REG_OUT*-Register enthält Daten für den Audio-Slot des linken und des rechten Kanals. Daher werden vor der Freigabe dieses Registers das *Valid-Slot-Bit* für den PCML-Slot (*TAG_REG_OUT* Bit 12) und für den PCMR-Slot (*TAG_REG_OUT* Bit 11) abgefragt (vgl. Abb. 101 und Code S. 101 Z. 168-232).

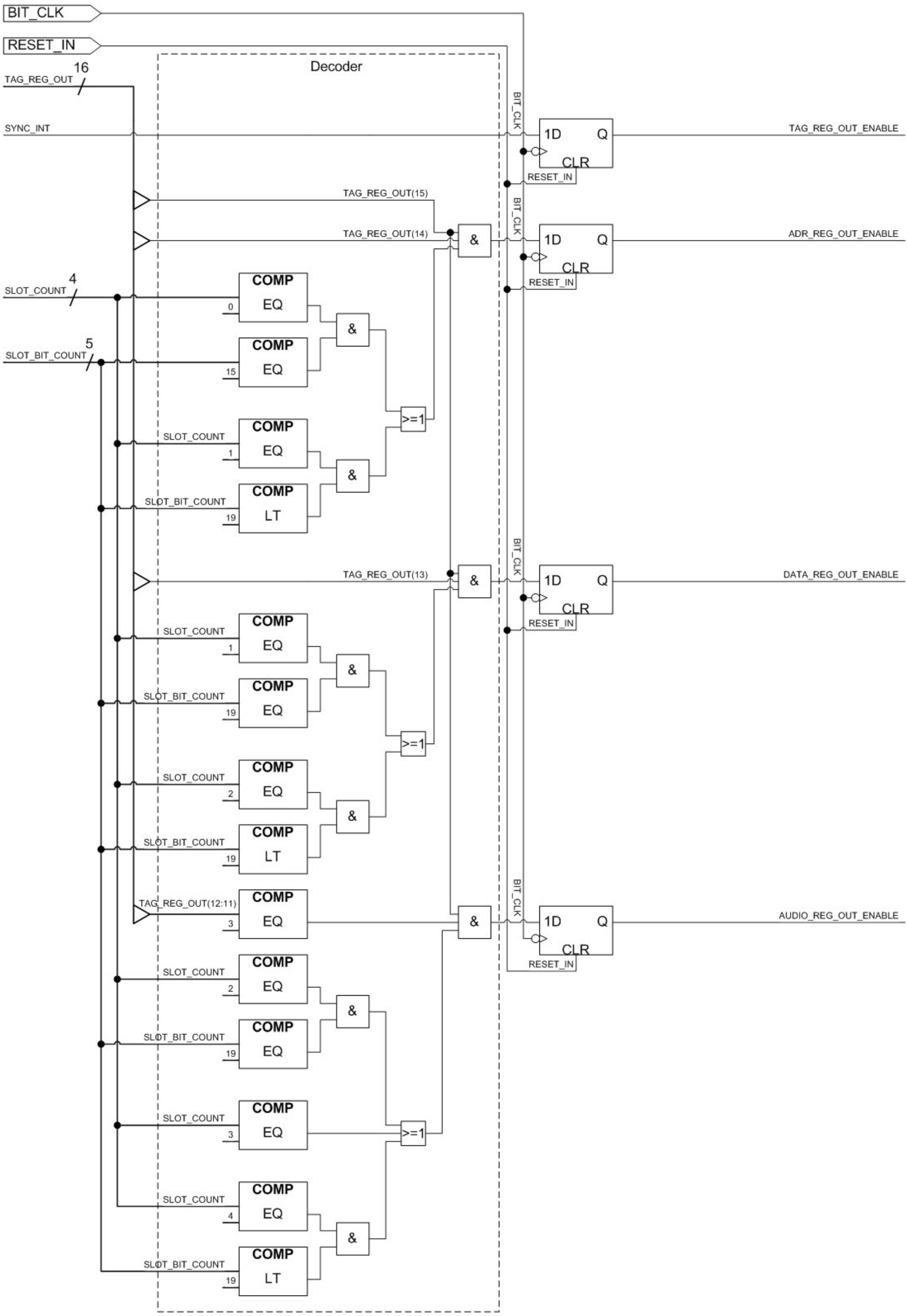

Abb. 101: CONTROL_OUT_REGISTERS Prozess bestehend aus einem Decoder, der anhand der Slot-Zählerstände die Ausgangsschieberegister aktiviert. Die Schiebefreigabe erfolgt auf der negativen Taktflanke (vgl. Code S. 101 Z. 168-232).

H.8 Anlaufphase des Codec-Interfaces

In den ersten vier Frames nach einem Systemstart oder einem Reset werden das *TAG_REG_OUT*-, das *ADR_REG_OUT*- und das *DATA_REG_OUT*-Register mit den benötigten Werten geladen, um die in Anhang G.2 erläuterten Konfigurationsregister des Audio-Codecs nacheinander zu beschreiben. Im fünften Frame wird das *TAG_REG_OUT*-Register mit dem TAG geladen, der dem Codec mitteilt, dass ab diesem Zeitpunkt nur noch Audiodaten übertragen werden. Ab Frame sechs ist die Anlaufphase beendet und die drei Ausgangsregister *TAG_REG_OUT*, *ADR_REG_OUT* und *DATA_REG_OUT* werden nicht mehr parallel geladen.

Der Zähler *INIT_SEQ*, der die Frames der Anlaufphase zählt, wird immer zu Beginn von Slot 5 inkrementiert, da zu diesem Zeitpunkt alle Schiebevorgänge der Ausgangsregister abgeschlossen sind und somit die Register parallel geladen werden können, was in Abhängigkeit dieses Zählers geschieht (vgl. Abb. 102).

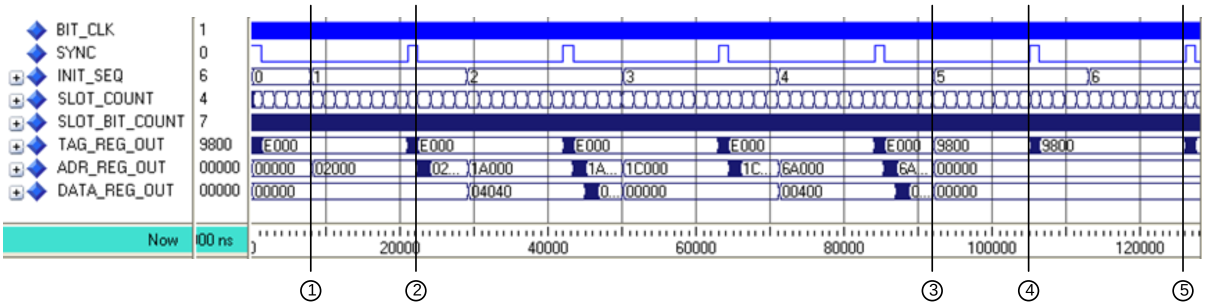

Abb. 102: Timing Simulation des Codec-Interfaces während der Anlaufphase. Adressregister ADR_REG_OUT und Datenregister DATA_REG_OUT werden parallel geladen (Marker 1); serielle Übertragung der Registerdaten zum Codec (Marker 2); Das Senden der Konfigurationdaten ist abgeschlossen, das TAG-Register wird für die Audioübertragung neu geladen, Adress- und Datenregister werden auf '0' gesetzt (Marker 3); Senden des TAGs zur Audioübertragung an den Codec (Marker 4); Anlaufphase ist beendet (Marker 5)

Im ersten Frame der Anlaufphase (*INIT_SEQ* = 1), wird das *TAG_REG_OUT*-Register mit dem Wert 0xE000 geladen. Dieser Wert gibt an, dass der Frame Daten im Command-Address-Slot und im Command-Data-Slot enthält. Das *ADR_REG_OUT*-Register wird mit dem Wert 0x02000 geladen, der die Adresse 0x02 des zu beschreibenden Registers darstellt. Das *DATA_REG_OUT*-Register wird mit dem Wert 0x00000 geladen, um die *Master Mute* Option im *Master Volume Register* zu deaktivieren. Im darauffolgenden Frame werden die Daten aus den Registern seriell zum Codec gesendet.

Diese Vorgänge wiederholen sich für das *Record Select Register* mit der Adresse 0x1A, das mit dem Wert 0x0404 beschrieben wird, um den Line-In-L-Eingang als Quelle für den linken Kanal auszuwählen und als Quelle für den rechten Kanal den Line-In-R-Eingang.

Das *Record Gain Register* mit der Adresse 0x1C wird mit dem Wert 0x0000 beschrieben, um die *Master Record Mute* Option auszuschalten und das *Feature Control/Status Register 1* mit der Adresse 0x6A wird mit 0x0040 beschrieben, um den *Headphone Driver* einzuschalten.

Mit dem fünften Frame wird die Konfiguration des Codecs abgeschlossen und das *TAG_REG_OUT*-Register wird mit dem Wert 0x9800 geladen. Bit 15 ist weiterhin gesetzt als Zeichen für einen gültigen Frame und Bit 12 und 11 signalisieren dem Codec gültige Daten in den beiden Audio-Slots. Das *ADR_REG_OUT*- und *DATA_REG_OUT*-Register werden mit den Werten 0x00000 geladen, da sie nach der Anlaufphase nicht mehr aktiviert werden.

In Frame 6 wird der neue TAG gesendet und der *INIT_SEQ*-Zähler zum letzten Mal auf 6 inkrementiert, um das Ende der Anlauphase zu markieren. Die Register werden nicht mehr parallel geladen. Das *TAG_REG_OUT*-Register wird ab diesem Zeitpunkt zyklisch durch eine Rück-

kopplung des seriellen Ausgangs auf den seriellen Eingang immer wieder mit dem gleichen Ausgangswert geladen.
Die ersten verarbeiteten Audiodaten werden daher ab dem siebten Frame vom FPGA zum Audio-Codec übertragen, wodurch die Anlaufphase eine Gesamtlänge von sechs Frames besitzt. Da ein Frame durch eine 48 kHz Periode des *SYNC*-Signals begrenzt wird, bertägt die Länge der Anlaufphase $125\ \mu s$.

I Bilder zur Entwicklungsplattform und zum AV-Board

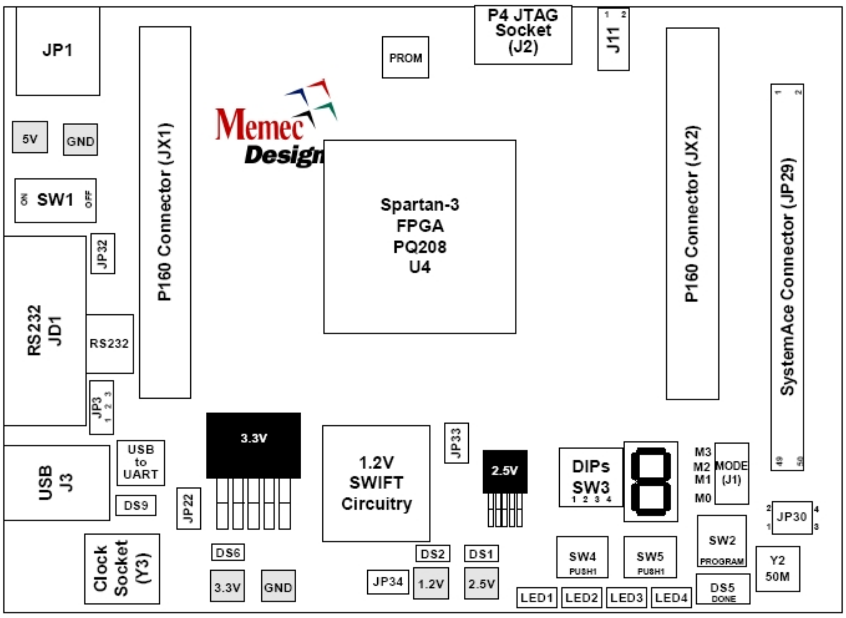

Abb. 103: Spartan-3 LC Entwicklungsplattform schematisch [14]

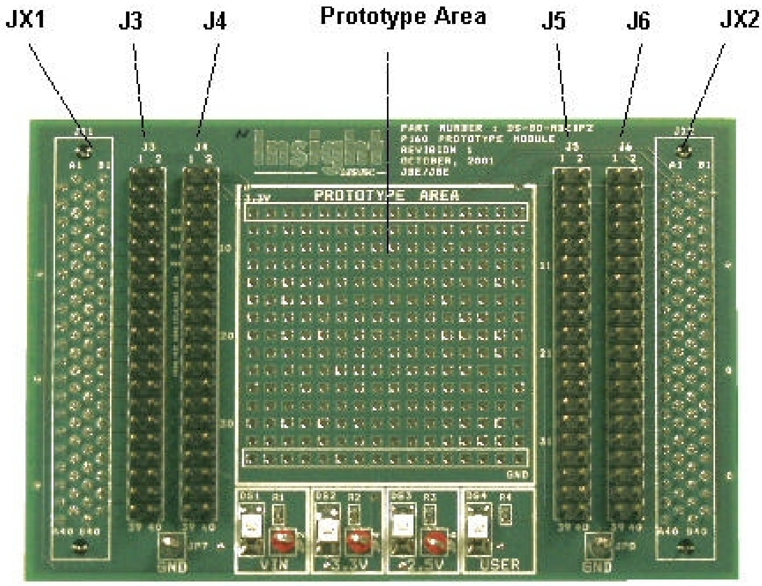

Abb. 104: P160 Prototyp Modul [13]

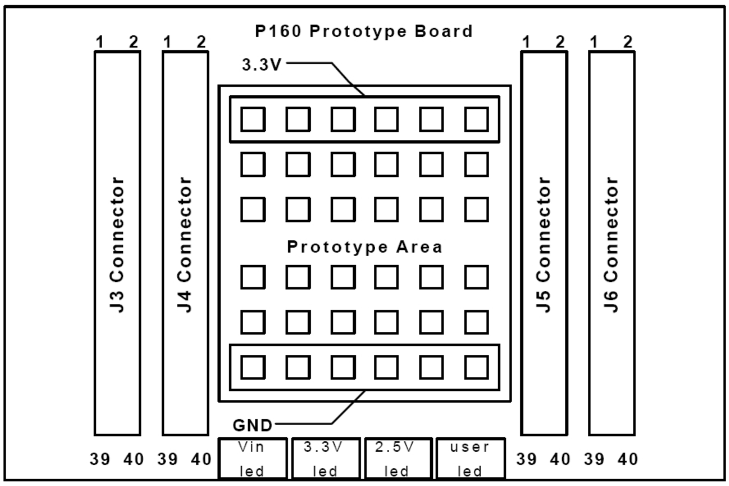

Abb. 105: P160 Prototyp Modul Block Diagramm [13]

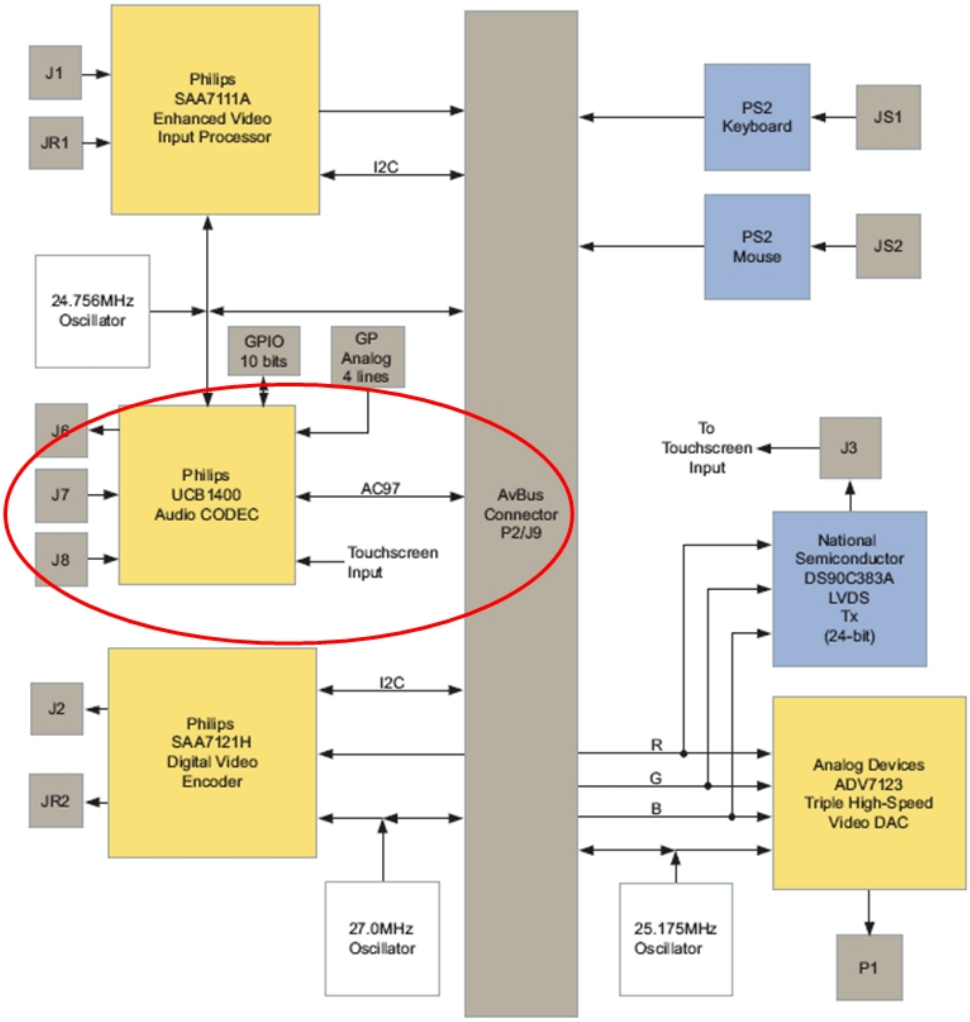

Abb. 106: Audio/Video Modul Block Diagramm [2]

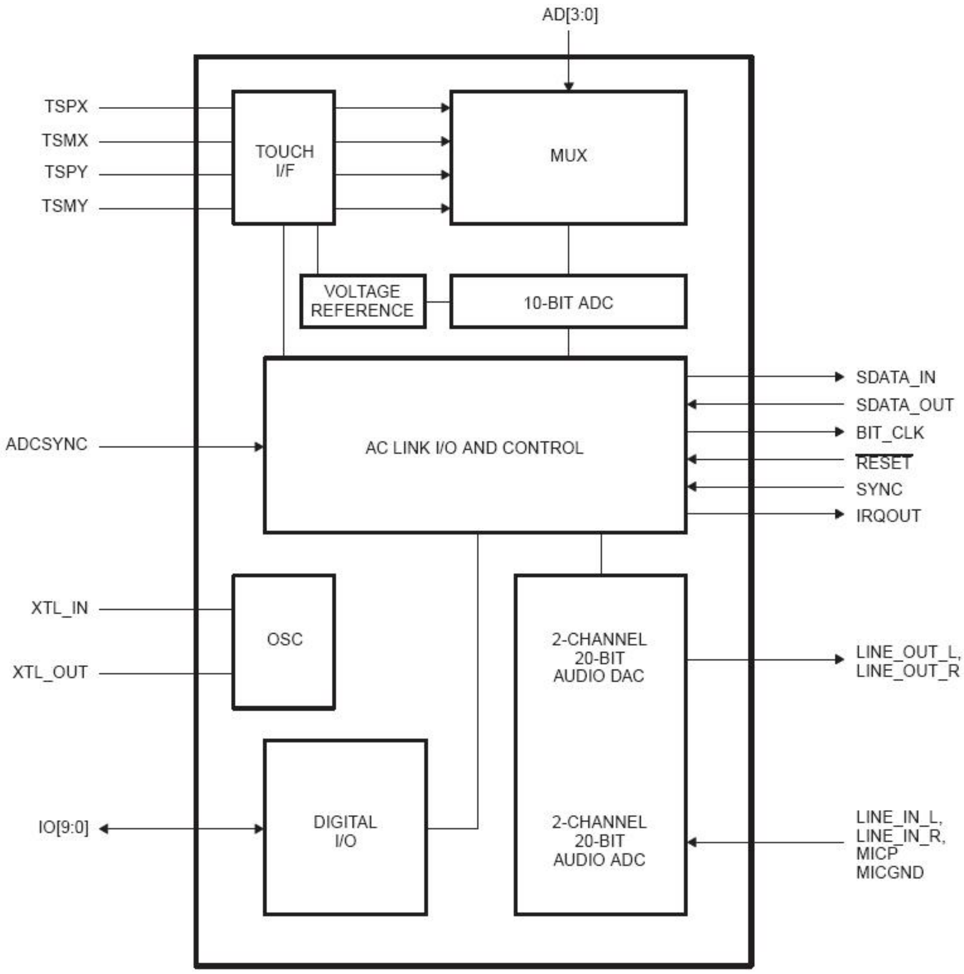

Abb. 107: UCB 1400 Block Diagramm [17]

J Tabellen zur Pinbelegung der Konnektoren des Spartan-3-Boards, AV-Boards und P160 Moduls

FPGA Pin #	JX1/2 Pin #	J3 Pin #	J3 Pin #	JX1/2 Pin #	FPGA Pin #
Vin	JX1 A4	1	2	JX1 A8	3.3V
2.5V	JX1 A12	3	4	JX1 A2	GND
P131	JX2 A1	5	6	NC	NC
P130	JX2 A2	7	8	JX1 B8	P132
P133	JX1 B9	9	10	JX1 B10	P135
P137	JX1 B11	11	12	JX1 B12	P138
P139	JX1 B13	13	14	JX1 B14	P140
P141	JX1 B15	15	16	JX1 B16	P143
P144	JX1 B17	17	18	JX1 B18	P146
P147	JX1 B19	19	20	JX1 B20	P148
P149	JX1 B21	21	22	JX1 B22	P150
P152	JX1 B23	23	24	JX1 B24	P154
P155	JX1 B25	25	26	JX1 B26	P156
P187	JX1 B27	27	28	JX1 B28	P189
P190	JX1 B29	29	30	JX1 B30	P191
P194	JX1 B31	31	32	JX1 B32	P196
P197	JX1 B33	33	34	JX1 B34	P198
P199	JX1 B35	35	36	JX1 B36	P200
P203	JX1 B37	37	38	JX1 B38	P204
P205	JX1 B39	39	40	JX1 B40	P2

Tabelle 5: J3 Pin-Belegung

FPGA Pin #	JX1/2 Pin #	J4 Pin #	J4 Pin #	JX1/2 Pin #	FPGA Pin #
Vin	JX1 A16	1	2	JX1 A8	3.3V
2.5V	JX1 A24	3	4	JX2 B2	P102
TCK	JX1 A1	5	6	JX1 A9	P161
TDO	JX1 A7	7	8	JX1 A11	P162
TDI	JX1 A5	9	10	JX1 A13	P180
TMS	JX1 A3	11	12	JX1 A15	P181
FPGA.BITSTREAM	JX1 B1	13	14	JX1 A17	P165
SM.DOUT/BUSY	JX1 B2	15	16	JX1 A19	P166
FPGA.CCLK	JX1 B3	17	18	JX1 A21	P167
DONE	JX1 B4	19	20	JX1 A23	P168
INITn	JX1 B5	21	22	JX1 A25	P169
PROGRAMn	JX1 B6	23	24	JX1 A27	P171
GND	JX1 A6	25	26	JX1 A29	P172
GND	JX1 A10	27	28	JX1 A31	P175
GND	JX1 A14	29	30	JX1 A33	P176
GND	JX1 A18	31	32	JX1 A35	P178
GND	JX1 A22	33	34	JX1 A37	P182
GND	JX1 A26	35	36	JX1 A39	P185
GND	JX1 A30	37	38	JX2 A39	P42
GND	JX1 A34	39	40	JX2 A40	P40

Tabelle 6: J4 Pin-Belegung

FPGA Pin #	JX1/2 Pin #	J5 Pin #	J5 Pin #	JX1/2 Pin #	FPGA Pin #
Vin	JX2 B3	1	2	JX2 B7	3.3V
2.5V	JX2 B11	3	4	JX2 B4	P101
NC	NC	5	6	JX2 B6	P100
JTAG_LOOPBACK		7	8	JX2 B8	P97
JTAG_LOOPBACK		9	10	JX2 B10	P96
NC	NC	11	12	JX2 B12	P95
NC	NC	13	14	JX2 B14	P94
NC	NC	15	16	JX2 B16	P93
NC	NC	17	18	JX2 B18	P90
NC	NC	19	20	JX2 B20	P87
NC	NC	21	22	JX2 B22	P86
NC	NC	23	24	JX2 B24	P85
GND	JX2 B1	25	26	JX2 B26	P81
GND	JX2 B5	27	28	JX2 B28	P80
GND	JX2 B9	29	30	JX2 B30	P79
GND	JX2 B13	31	32	JX2 B32	P78
GND	JX2 B17	33	34	JX2 B34	P77
GND	JX2 B21	35	36	JX2 B36	P76
GND	JX2 B25	37	38	JX2 B38	P74
GND	JX2 B29	39	40	JX2 B40	P72

Tabelle 7: J5 Pin-Belegung

FPGA Pin #	JX1/2 Pin #	J6 Pin #	J6 Pin #	JX1/2 Pin #	FPGA Pin #
Vin	JX2 B15	1	2	JX2 B19	3.3V
2.5V	JX2 B23	3	4	JX2 B1	GND
P126	JX2 A4	5	6	JX2 A3	P128
P124	JX2 A6	7	8	JX2 A5	P125
P122	JX2 A8	9	10	JX2 A7	P123
P119	JX2 A10	11	12	JX2 A9	P120
P116	JX2 A12	13	14	JX2 A11	P117
P114	JX2 A14	15	16	JX2 A13	P115
P111	JX2 A16	17	18	JX2 A15	P113
P108	JX2 A18	19	20	JX2 A17	P109
P106	JX2 A20	21	22	JX2 A19	P107
P68	JX2 A22	23	24	JX2 A21	P71
P65	JX2 A24	25	26	JX2 A23	P67
P63	JX2 A26	27	28	JX2 A25	P64
P61	JX2 A28	29	30	JX2 A27	P62
P57	JX2 A30	31	32	JX2 A29	P58
P51	JX2 A32	33	34	JX2 A31	P52
P48	JX2 A34	35	36	JX2 A33	P50
P45	JX2 A36	37	38	JX2 A35	P46
P43	JX2 A38	39	40	JX2 A37	P44

Tabelle 8: J6 Pin-Belegung

Signal Name	FPGA Pin #	P2 Pin #	J6 Pin #	J6 Pin #	P2 Pin #	FPGA Pin #	Signal Name
AC97_SDATA_OUT	P126	P101	5	6	P109	P128	IRQ_OUT
AC97_SDATA_IN	P124	P103	7	9	P110	P122	ADC_SYNC
AC97_BIT_CLK	P119	P104	11	13	P106	P116	AC97_SYNC
AC97_RESET	P114	P107	15				

Tabelle 9: Pin-Verbindung zwischen P160- und AV-Board

K Tabellen zur Definition der Konfigurationsregister-Bits

Bit	D15	D14	D13	D12	D11	D10	D9	D8
Symbol	X	X	X	X	X	X	ID9	X
Bit	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
Symbol	ID7	X	ID5	X	X	X	X	X

Tabelle 10: Reset Register

Bit	Symbol	Typ	Beschreibung
D15-D10	X	R	Reserviert
D9	ID9	R	Immer ’1’ (20-Bit ADC Auflösung wird unterstützt)
D8	X	R	Reserviert
D7	ID7	R	Immer ’1’ (20-Bit DAC Auflösung wird unterstützt)
D6	X	R	Reserviert
D5	ID5	R	Immer ’1’ (Bass Boost wird unterstützt)
D4-D0	X	R	Reserviert

Tabelle 11: Reset Register Bit-Beschreibung

Bit	D15	D14	D13	D12	D11	D10	D9	D8
Symbol	MM	X	ML5	ML4	ML3	ML2	ML1	ML0
Bit	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
Symbol	X	X	MR5	MR4	MR3	MR2	MR1	MR0

Tabelle 12: Master Volume Register

Bit	Symbol	Typ	Beschreibung
D15	MM	RW	Master Mute
D14	X	R	Reserviert
D13-D8	ML5-ML0	RW	Schallschwächung des linken Kanals in 1,5 dB Schritten (000000 = 0 dB; 111111 = -94,5 dB)
D7-D6	X	R	Reserviert
D5-D0	MR5-DR0	RW	Schallschwächung des rechten Kanals in 1,5 dB Schritten (000000 = 0 dB; 111111 = -94,5 dB)

Tabelle 13: Master Volume Register Bit-Beschreibung

Bit	D15	D14	D13	D12	D11	D10	D9	D8
Symbol	X	X	X	X	X	SL2	SL1	SL0
Bit	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
Symbol	X	X	X	X	X	SR2	SR1	SR0

Tabelle 14: Record Select Register

Bit	Symbol	Typ	Beschreibung
D15-D11	X	R	Reserviert
D10-D8	SL2-SL0	RW	Eingangsquelle für den linken Kanal (000 = MIC; 100 = Line In L; andere Werte reserviert)
D7-D3	X	R	Reserviert
D2-D0	SR2-SR0	RW	Eingangsquelle für den rechten Kanal (000 = Copy Left; 100 = Line In R; andere Werte reserviert)

Tabelle 15: Record Select Register Bit-Beschreibung

Bit	**D15**	**D14**	**D13**	**D12**	**D11**	**D10**	**D9**	**D8**
Symbol	RM	X	X	X	GL3	GL2	GL1	GL0
Bit	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
Symbol	X	X	X	X	GR3	GR2	GR1	GR0

Tabelle 16: Record Gain Register

Bit	**Symbol**	**Typ**	**Beschreibung**
D15	RM	RW	Master Record Mute
D14-D12	X	R	Reserviert
D11-D8	GL3-GL0	RW	Verstärkung des linken Aufnahme-Kanals in 1,5 dB Schritten (0000 = 0 dB; 1111 = 22,5 dB)
D7-D4	X	R	Reserviert
D3-D0	GR3-GR0	RW	Verstärkung des rechten Aufnahme-Kanals in 1,5 dB Schritten (0000 = 0 dB; 1111 = 22,5 dB)

Tabelle 17: Record Gain Register Bit-Beschreibung

Bit	**D15**	**D14**	**D13**	**D12**	**D11**	**D10**	**D9**	**D8**
Symbol	X	BB3	BB2	BB1	BB0	TR1	TR0	M1
Bit	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
Symbol	M0	HPEN	DE	DC	HIPS	GIEN	X	OVFL

Tabelle 18: Feature Control/Status Register 1

Bit	**Symbol**	**Typ**	**Beschreibung**
D15	X	R	Reserviert
D14-D11	BB3-BB0	RW	Bass boost
D10-D9	TR1-TR0	RW	Treble boost
D8-D7	M1-M0	RW	Mode
D6	HPEN	RW	’1’: Headphone Driver eingeschaltet
D5	DE	RW	’1’: De-Emphasis ist eingeschaltet bei Abtastraten von 48, 44,1 oder 32 kHz
D4	DC	RW	’1’: DC Filter ist eingeschaltet
D3	HIPS	R	’1’: Hochpass-Filter im Dezimator aktivieren
D2	GIEN	RW	’1’: Interrupt/wake-up Signalisierung ist eingeschaltet
D1	X	R	Reserviert
D0	OVFL	RW	ADC Overflow Status

Tabelle 19: Feature Control/Status Register 1 Bit-Beschreibung

L UCF-Datei des Spartan 3 FPGA

```
NET "CLK" TNM_NET = "CLK";
TIMESPEC "TS_CLK" = PERIOD "CLK" 20 ns HIGH 50 %;
NET "BIT_CLK" TNM_NET = "BIT_CLK";
TIMESPEC "TS_BIT_CLK" = PERIOD "BIT_CLK" 81.4 ns HIGH 50 %;
OFFSET = IN 5 ns BEFORE "CLK"  ;
OFFSET = OUT 5 ns AFTER "CLK"  ;
OFFSET = IN 10 ns BEFORE "BIT_CLK"  ;
OFFSET = OUT 15 ns AFTER "BIT_CLK"  ;
#PACE: Start of Constraints generated by PACE

#PACE: Start of PACE I/O Pin Assignments
NET "CLK"  LOC = "p184" | IOSTANDARD = LVCMOS33 ;
NET "DIP<0>"  LOC = "p26" | IOSTANDARD = LVCMOS33 ;
NET "DIP<1>"  LOC = "p27" | IOSTANDARD = LVCMOS33 ;
NET "DIP<2>"  LOC = "p28" | IOSTANDARD = LVCMOS33 ;
NET "DIP<3>"  LOC = "p29" | IOSTANDARD = LVCMOS33 ;
NET "LED<0>"  LOC = "p20" | IOSTANDARD = LVCMOS33 ;
NET "LED<1>"  LOC = "p21" | IOSTANDARD = LVCMOS33 ;
NET "LED<2>"  LOC = "p18" | IOSTANDARD = LVCMOS33 ;
NET "LED<3>"  LOC = "p19" | IOSTANDARD = LVCMOS33 ;
NET "DISPLAY<0>"  LOC = "p36" | IOSTANDARD = LVCMOS33 ;
NET "DISPLAY<1>"  LOC = "p37" | IOSTANDARD = LVCMOS33 ;
NET "DISPLAY<2>"  LOC = "p39" | IOSTANDARD = LVCMOS33 ;
NET "DISPLAY<3>"  LOC = "p33" | IOSTANDARD = LVCMOS33 ;
NET "DISPLAY<4>"  LOC = "p31" | IOSTANDARD = LVCMOS33 ;
NET "DISPLAY<5>"  LOC = "p34" | IOSTANDARD = LVCMOS33 ;
NET "DISPLAY<6>"  LOC = "p35" | IOSTANDARD = LVCMOS33 ;
NET "BIT_CLK"  LOC = "p119" | IOSTANDARD = LVCMOS33 ;
NET "N_RESET_OUT"  LOC = "p114" | IOSTANDARD = LVCMOS33 ;
NET "PUSH1"  LOC = "p22" | IOSTANDARD = LVCMOS33 ;
NET "SDATA_IN"  LOC = "p124" | IOSTANDARD = LVCMOS33 ;
NET "SDATA_OUT"  LOC = "p126" | IOSTANDARD = LVCMOS33 ;
NET "SYNC"  LOC = "p116" | IOSTANDARD = LVCMOS33 ;
NET "SDATA_IN_TST"  LOC = "p40" | IOSTANDARD = LVCMOS33 ;
NET "SYNC_TST"  LOC = "p205" | IOSTANDARD = LVCMOS33 ;
NET "SDATA_OUT_TST"  LOC = "p2" | IOSTANDARD = LVCMOS33 ;
NET "USER_LED"  LOC = "p128" | IOSTANDARD = LVCMOS33 ;
NET "BIT_CLK_TST"  LOC = "p42" | IOSTANDARD = LVCMOS33 ;


#PACE: Start of PACE Area Constraints

#PACE: Start of PACE Prohibit Constraints

#PACE: End of Constraints generated by PACE
```

Abkürzungsverzeichnis

ANC	**A**ctive **N**oise **C**ancellation
CLB	**C**onfigurable **L**ogic **B**lock
DCM	**D**igital **C**lock **M**anager
FIR	**F**inite **I**mpulse **R**esponse
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
HW	**H**ard**w**are
IIR	**I**nfinite **I**mpulse **R**esponse
IOB	**I**nput **O**utput **B**lock
LMS	**L**east **M**ean **S**quare (Error)
LPC	**L**inear **P**redictive **C**oding
LPC	**L**inear **T**ime **I**nvariant
LSB	**L**east **S**ignificant **B**it
LUT	**L**ook **U**p **T**able
MAC	**M**ultiply **Ac**cumulate
MSE	**M**ean **S**quare **E**rror
PCM	**P**ulse **C**ode **M**odulation
RAM	**F**inite **S**tate **M**achine
RAM	**R**andom **A**ccess **M**emory
RTL	**R**egister **T**ransfer **L**evel
SDA	**S**teepes **D**escent **A**lgorithm
VHDL	**V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage

Index

μ, 24, 28, 34
Dedicated Multipliers, 64
FxLMS-Algorithmus, 30
`entities`, 49
`entity`, 49

AC'97, 43, 45
Adaptive Entzerrung, 13
Adaptive Feedback System, 17
Adaptive Feedforward System, 17, 19
adaptive FIR-Filter, 19
Adaptive Regelung, 13
Adaptive Schrittweite, 76
Adaptives FIR-Filter, 67
adaptives sequentielles FIR-Filter, 40
Audio-Codec, 29, 30, 45, 60
Autokorrelationsmatrix, 21–23

Block-RAM, 41, 53, 57, 60

Codec-Interface, 41, 45, 62, 64, 67
Codec-Interfaces, 80

Datenpfad, 40, 41, 50, 58, 60, 80
Datenpfades, 58
Diagonaltmatrizen, 26
Direktform, 49
Dual-Port-Block-RAM, 48
Dual-Port-RAM, 54, 56–58, 78

Echokompensation, 13
Eigenvektor, 26
Eigenvektortransformation, 26
Einheitsmatrix, 25
Embedded 18x18-Bit Multiplizierer, 41
embedded 18x18-Bit Multiplizierer, 48, 52, 57

Feedforward System, 29
Fehlerfläche, 22, 23
Fehlerfunktion, 21, 23, 25
Finite state machine, 41
FIR-Filter, 34, 43, 45, 48, 49, 62, 64, 77, 78, 80
FIR-Filters, 59
FIR-Filterstruktur, 19
FIR-Struktur, 48
Fourier-Transformation, 78
Frequenzgang, 31, 38, 73
FSM, 41, 58, 60, 62

Gradient, 22, 23, 25
Gradienten-Verfahren, 23
Gruppenlaufzeit, 30, 33, 39, 48, 56, 73, 78, 80
Guard-Bits, 42, 52, 57

IIR-Filter, 78
IIR-Struktur, 19

Kreuzkorrelationsvektor, 21, 22
Kunstkopf-Messsystem, 70

Least Mean Square Error, 22
Level-2 S-Functions, 34
Linear Predictive Codeing, 15
Linear-Phasen-Struktur, 49
Lineare optimale Filterung, 21
LMS-Adaptionsalgorithmus, 22, 34, 40, 48, 76, 80
LPC-Analyse, 13
LTI-System, 13

MAC-Einheit, 48, 53, 59
MAC-Unit, 41, 49, 63
Mealy, 60
Mealy-Automat, 51, 59
Mealy-Charakterisitk, 59
Mealy-FSM, 59
Mean Square Error, 21
Method Of Steepest Descent, 23
mittlere Eingangsleistung, 28, 71, 76
Moore, 60
Moore-Ausgänge, 59
Moore-Charakteristik, 59
MSE, 21, 22, 78
Multisensor Störsignal Kompensation, 16
Multisensor-Störsignal-Kompensation, 13

Newton-Verfahren, 23, 24
Normierter LMS-Algorithmus, 76

P160-Modul, 43
Pipeline, 40, 51, 59
Pipelinestruktur, 51
Pipelinestrukturen, 41
Pipelinestufen, 61
Prozessorelement, 40, 49

Q-Format, 42, 52

Register Transfer Level, 40
RTL, 40
RTL-Modell, 40

Schrittenweitenfaktor, 28
Schrittweitenfaktor, 24, 34, 48, 56, 67, 71, 73, 75
Schrittweitenfaktors, 76
SDA, 24
Sekundärpfad, 30, 34, 48, 56, 70, 72–74, 76–78, 80
Signal-Prädiktion, 15
Signal-Prognose, 13
Spartan-3, 43
Spartan-3 FPGA, 40
Sparten-3 FPGA, 41
Steepest Descent Algorithm, 24
Steuerpfad, 40, 50, 58, 80
System Identifikation, 77
System-Identifikation, 13
System-Inversion, 13, 14

Top-Down Entwurfsstil, 80

Wiener-Filter, 21
Wiener-Filters, 21
Wiener-Hopf-Gleichung, 22
Wiener-Lösung, 22–24

Xilinx CORE Generator, 42, 64

Zustandsautomat, 41, 50, 58
Zweierkomplement, 53
Zweierkomplement-Darstellung, 42

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 15.02.2008