Diplomarbeit

Frederik Teichert

Digital-Down-Converter für

Field-Programmable-Gate-Arrays in VHDL

Fakultät Technik und Informatik
Department Informations- und
Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and
Electrical Engineering

Frederik Teichert

Digital-Down-Converter für Field-Programmable-Gate-Arrays in VHDL

Diplomarbeit eingereicht im Rahmen der Diplomprüfung
im Studiengang Informations- und Elektrotechnik
Studienrichtung Informationstechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr.-Ing. Karl Ragmar Riemschneider
Zweitgutachter : Prof. Dr. Franz Schubert

Abgegeben am 11. März 2009

Frederik Teichert

Thema der Diplomarbeit

Digital-Down-Converter für Field-Programmable-Gate-Arrays in VHDL

Stichworte

FPGA (Field-Programmable-Gate-Array), VHDL, DDC (Digital-Down-Converter), CIC (Cascaded-Integrator-Comb), SCIC (Sharpened-Cascaded-Integrator-Comb)

Kurzzusammenfassung

Für High-End-Anwendungen der digitalen Signalverarbeitung sind Digital-Down-Converter in integrierten Schaltungen nicht leistungsstark genug und nicht portabel, weshalb diese auf FPGAs implementiert werden. Es werden verschiedene Ansätze verfolgt, einen Digital-Down-Converter in VHDL zu implementieren, wobei die Schwerpunkte auf hardware-optimierter Dezimation und der Erzeugung harmonischer Schwingungen mit digitalen Schaltungen liegen. Am Ende der Diplomarbeit werden die Implementierungen miteinander verglichen und die Ergebnisse sowie weitere Optimierungsmöglichkeiten vergleichend dargestellt.

Frederik Teichert

Title of the paper

Digital-Down-Converter for FPGAs using VHDL

Keywords

FPGA (Field-Programmable-Gate-Array), VHDL, DDC (Digital-Down-Converter), CIC (Cascaded-Integrator-Comb), SCIC (Sharpened-Cascaded-Integrator-Comb)

Abstract

Digital-down-converters in integrated circuits are not fast enough for modern high-end-applications in digital signal processing. DDCs in field-programmable-gate-arrays can achieve much better data throughput and reach a higher clockrate and are suitable for reuse. This diploma theses puts emphasis on generating harmonic sinusodial signals with hardware optimized digital circuits and hardware optimized decimation with (S)CIC-decimators in VHDL. Different DDC approaches are shown and are compared with each other in respect of chip complexity and performance.

Danksagungen

An erster Stelle danke ich meinen betreuenden Professoren der HAW-Hamburg, Herrn Dr.-Ing. Karl-Ragmar Riemschneider, der sich in der Funktion meines betreuenden Prüfers sehr viel Zeit für mich genommen hat und mich mit konstruktiver Kritik unterstützte und weiterhin gilt mein Dank meinem Zweitgutachter, Herrn Dr. Franz Schubert, der mich ebenfalls fachlich sehr unterstützte.

Zudem gilt mein Dank meinen weiteren Betreuern Herrn Massimo Gelosa und Herrn Steffan Klimkiewicz, die sowohl mit fachlicher als auch mit persönlicher Unterstützung zum Entstehen dieser Diplomarbeit beigetragen haben.

Meiner Familie danke ich für den seelischen Beistand und die finanzielle Unterstützung, ohne die ich mein Studium nicht in der Form und Kürze hätte absolvieren können.

Ich danke dem Leiter der EDV-Servicegruppe des Zentrums für Molekulare Neurobiologie Hamburg, Herrn Dr. Hans-Martin Ziethen, und meinem ehemaligen Kommilitonen, Herrn Thorsten Eger, für deren hilfreiche Unterstützung durch Korrekturlesen.

Allen, die bisher nicht namentlich erwähnt wurden, danke ich für das Interesse an meiner Diplomarbeit und hoffe, dass der interessierte Leser einen guten Überblick über die Implementierung eines Digital-Down-Converters in VHDL erlangt.

Inhaltsverzeichnis

Glossar **1**

1. Einleitung **3**

2. Digital Down Converter **5**

3. Digitale Oszillatoren **11**
3.1. Anforderungen an den Oszillator . . . 11
3.2. Lookuptable-Oszillator . . . 11
3.2.1. Vorüberlegungen . . . 11
3.2.2. Hardwareoptimierung . . . 12
3.2.3. Vor- und Nachteile des Lookuptable-Oszillators . . . 14
3.3. IIR-Oszillator . . . 15
3.3.1. Herleitung der Schaltung . . . 15
3.3.2. Ermittlung der Pole . . . 16
3.3.3. Stabilisierung . . . 18
3.3.4. Stabilisierung durch kontinuierliches Neuinitialisieren . . . 18
3.3.5. Stabilisierung durch Neuinitialisierung bei Nulldurchgängen . . . 20
3.3.6. Stabilisierung durch Polverschiebung . . . 21
3.3.7. Genauigkeit und Frequenz . . . 23
3.3.8. Simulation . . . 24
3.3.9. Vor- und Nachteile des IIR-Oszillators . . . 24
3.4. Taylorreihen-Oszillator . . . 27
3.4.1. Herleitung der Taylorreihen für harmonische Schwingungen . . . 27
3.4.2. Genauigkeit und Iterationsanzahl . . . 29
3.4.3. Implementierbarkeit . . . 30
3.4.4. Vor- und Nachteile des Taylorreihen-Oszillators . . . 31
3.5. CORDIC-Oszillator . . . 32
3.5.1. Architekturen . . . 33
3.5.2. CORDIC als Sinus-Cosinus-Oszillator . . . 34
3.5.3. Implementierbarkeit . . . 35
3.5.4. Vor- und Nachteile des CORDIC-Oszillators . . . 35
3.6. Gegenüberstellung der Oszillatoren . . . 36

4. Finite Impulse Response Filter (FIR-Filter) **37**
4.1. Grundlagen . . . 37
4.2. Transponierung und Hardwareoptimierung . . . 37
4.3. Übertragungsfunktion und Nullstellen . . . 39
4.4. Koeffizientenberechnung . . . 40
4.5. Quantisierungseffekte . . . 41
4.6. Vor- und Nachteile des FIR-Dezimationsfilters . . . 42

5. Kaskadiertes FIR-Halfband-Dezimationsfilter **43**

5.1. Grundlagen . . . 43
5.2. Kaskadierung mehrerer Filter . . . 44
5.3. Vor- und Nachteile des FIR-Halfbandfilters . . . 45
5.4. Gegenüberstellung der FIR-Filter . . . 45

6. Cascaded Integrator Comb (CIC-Filter) 47
6.1. Herleitung der Übertragungsfunktion eines CIC-Filters . . . 47
6.2. Frequenzcharakteristik . . . 49
6.3. Registerdimensionierung . . . 50
6.4. Hardwareoptimierung . . . 50
6.5. Runden und Abschneiden . . . 51
6.6. Entwurfsbeispiel eines CIC-Dezimators . . . 54
6.7. Vor- und Nachteile des CIC-Dezimators . . . 57

7. Sharpened Cascaded Integrator Comb (SCIC-Dezimator) 59
7.1. Herleitung der Übertragungsfunktion eines SCIC-Dezimators . . . 59
7.2. Frequenzcharakteristik . . . 60
7.3. Registerdimensionierung . . . 61
7.4. Hardwareoptimierung . . . 62
7.5. Vergleich von CIC- und SCIC-Dezimatoren . . . 64
7.6. Vor- und Nachteile des SCIC-Dezimators . . . 65
7.7. Gegenüberstellung des CIC- und SCIC-Dezimators . . . 66

8. Field-Programmable-Gate-Array 71
8.1. Programmierbare Verbindung von Logikblöcken . . . 72
8.2. Design-Flow von FPGAs . . . 72
8.3. Software . . . 74

9. Implementierung der Oszillatoren in VHDL 75
9.1. Implementierung eines Lookuptable-Oszillators in VHDL . . . 75
9.2. Implementierung eines IIR-Oszillators in VHDL . . . 76
9.2.1. Zählender IIR-Oszillator . . . 76
9.2.2. Überwachender IIR-Oszillator . . . 77
9.2.3. Adaptiver IIR-Oszillator . . . 78
9.3. Gegenüberstellung der Syntheseergebnisse . . . 79

10. Implementierung eines FIR-Dezimationsfilters in VHDL 81
10.1. Realisierungen . . . 83
10.2. Gegenüberstellung der Syntheseergebnisse . . . 84

11. Implementierung eines FIR-Halfband-Dezimationsfilters in VHDL 87
11.1. Gegenüberstellung der Syntheseergebnisse . . . 89

12. Implementierung eines CIC-Dezimators in VHDL 91
12.1. Elemente . . . 92
12.2. Gesamtsystem . . . 93
12.3. Simulation und Ergebnisse . . . 93
12.4. Gegenüberstellung der Syntheseergebnisse . . . 96

13. Implementierung eines SCIC-Dezimators in VHDL 97
13.1. Gegenüberstellung der Syntheseergebnisse und Simulationsergebnissen von CIC- und SCIC-Dezimatoren . . . 100

14. Simulation des Gesamtsystems mit C-Programmen 107
14.1. Simulationsprogramme . . . 107
14.2. Funktionsnachweis des Gesamtsystems . . . 109

15. Vergleich verschiedener DDC-Systeme 111
15.1. Vergleich der Dezimationsfilter anhand der Syntheseergebnisse . . . 113
15.2. Vergleich der Dezimationsfilter anhand der Impulsantwort . . . 116
15.3. Übersicht der Chipauslastung verschiedener Digital-Down-Converter . . . 118

16. Fazit 121

A. Computer-Arithmetik 127
A.1. Multiplikation . . . 127
A.2. Booth-Algorithmus . . . 128

B. Anhang für Digitale Oszillatoren 133
B.1. Lookuptable-Oszillator . . . 133
B.1.1. Selbstfüllender Lookuptable für den Lookuptable-Oszillator . . . 133
B.1.2. Oszillator . . . 136
B.2. IIR-Oszillator . . . 139
B.2.1. Zählerbasierter IIR-Oszillator . . . 139
B.2.2. IIR-Oszillator mit Nulldurchgangsdetektion . . . 142
B.2.3. Adaptiver IIR-Oszillator . . . 145

C. VHDL-Komponenten 155
C.1. Booth-Multiplier mit 4er-Sequenz . . . 155
C.2. Serieller Multiplizierer . . . 158

D. Anhang für FIR-Dezimations-Filter 161
D.1. FIR-Filter mit Hardwaremultipliern . . . 161
D.2. FIR-Filter mit Booth-4-Multipliern . . . 164

E. Anhang für FIR-Halfband-Filter 169
E.1. FIR-Filter mit Hardware-Multiplizierern . . . 169
E.2. FIR-Filter mit seriellen Multiplizierern . . . 174

F. Anhang für Cascaded Integrator Comb 179
F.1. Herleitung des Amplitudengangs eines CIC-Filters . . . 179
F.2. Herleitung der maximalen Verstärkung eines CIC-Dezimators . . . 180
F.3. Simulation mit Scilab . . . 183
F.4. VHDL-Implementierungen . . . 184

G. Anhang für Sharpened Cascaded Integrator Comb 191
G.1. Simulation mit Scilab . . . 191
G.2. VHDL-Implementierungen . . . 193

H. Quelltexte des DDC-Simulators 199

Abbildungsverzeichnis 215

Tabellenverzeichnis 217

Quellcodeverzeichnis 219

Literaturverzeichnis **221**

Glossar

B	Breite von Registern in Bit
$Im()$	Imaginäranteil einer komplexen Zahl
$Re()$	Realanteil einer komplexen Zahl
$\lceil \ldots \rceil$	ceil() - Aufrunden eines Wertes auf die nächste Ganzzahl
$\lfloor \ldots \rfloor$	floor() - Abrunden eines Wertes auf die nächste Ganzzahl
f_c	Passbandbreite von CIC- und SCIC-Dezimationsfiltern
ADC	Analog-to-Digital-Converter, Deutsch ADU, ist ein Analog-Digital-Umsetzer
ALUT	Lookuptable in einem FPGA-Logikblock
CIC	Cascaded Integrator Comb nach [CIC]
CORDIC	Coordinate Rotation Digital Computer
DDC	Digital-Down-Converter
Dedicated Logicregisters	Register in einem FPGA
Generic	Ein *Generic* ist ein Parameter einer VHDL-Komponente, durch den ein generisches Design mit VHDL ermöglicht wird
Passbanddroop	Maximale Dämpfung eines Filters innerhalb des Passbands
QDSO	Quadrature Digital Sinusoidal Oscillator - Ein Oszillator, der sinusförmige Signale in Phasenquadratur erzeugt
SCIC	Sharpened Cascaded Integrator Comb nach [SCIC]
Taktdomäne	Ein Bereich einer digitalen Schaltung, welcher mit demselben Taktsignal versorgt wird
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLSI	Very Large Scale Integration - Ein sehr hoch integriertes System

1. Einleitung

Digital-Down-Converter finden derzeit eine weite Verbreitung in modernen digitalen Kommunikationssystemen wie in Mobiltelefonen, digitalen Radios, Satellitenempfängern, DVB-C/T/S-Empfängern, DSL-Modems, Kabelmodems und in Anwendungen der Militär- und Medizintechnik. Somit liegt auf der Hand, auf welche Annehmlichkeiten des modernen Lebens wir ohne Digital-Down-Converter verzichten müssten. Mobiltelefonie und mobiles Internet wie auch breitbandiges Internet, das in der Regel mit DSL-Modems in Wohnungen gelangt, sind heutzutage praktisch unverzichtbar. Durch digitales Fernsehen konnte die Anzahl der Fernsehkanäle, die über eine Antenne empfangen werden können, vervielfacht werden.

Digitale Schaltungen sind in der Regel wesentlich kostengünstiger als analoge Pendants und bieten die Möglichkeit, eine hohe Integrationsdichte zu erreichen, wohingegen analoge äquivalente Schaltungsaufbauten mehr Platz benötigen.

Das Ziel der Diplomarbeit ist die Untersuchung, Entwicklung und der Vergleich verschiedener Digital-Down-Converter für FPGAs. Dabei werden mehrere Realisierungsmöglichkeiten bezüglich deren Genauigkeit und Leistungsfähigkeit miteinander verglichen, um eine optimale Lösung für gegebene Rahmenbedingungen zu finden.

Im Rahmen dieser Diplomarbeit wird die Funktionsweise eines Digital-Down-Converters und dessen Komponenten erläutert, sowie näher darauf eingegangen, auf welche Weise das Ziel der Diplomarbeit erreicht wurde.

2. Digital Down Converter

Ein Digital-Down-Converter konvertiert ein reelles, digitales Bandpasssignal zu einem komplexen Tiefpasssignal (Bild 2.1) und führt gleichzeitig eine Dezimation durch.

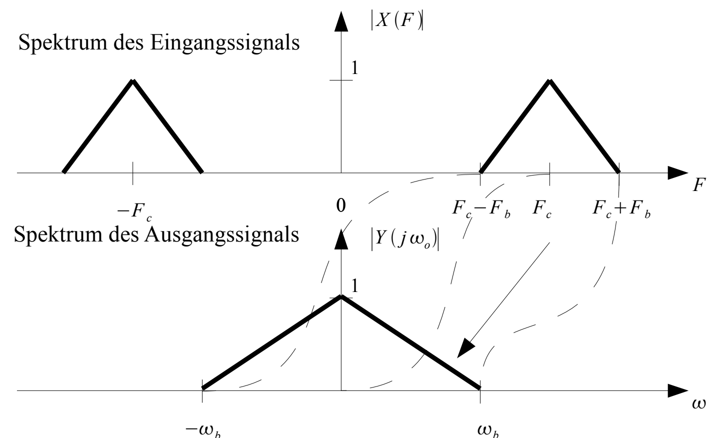


Bild 2.1.: Reelles Bandpasssignal und korrespondierendes komplexes Tiefpasssignal

Dazu wird in der Regel zunächst ein analoges, hochfrequentes Bandpasssignal mit einem analogen Mischer zu einer Zwischenfrequenz heruntergemischt und doppelte Frequenzanteile herausgefiltert. Das resultierende Bandpasssignal wird mit einem Analog-Digital-Umsetzer digitalisiert und dem Digital-Down-Converter zugeführt (Bild 2.2).

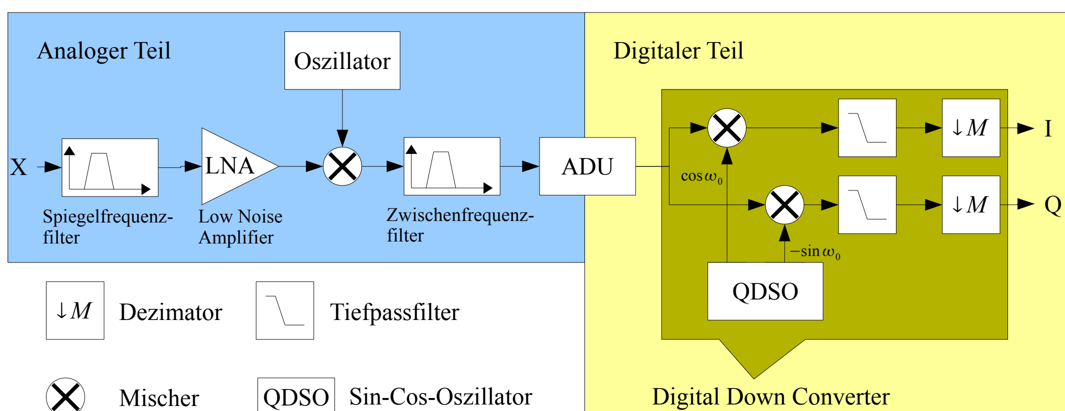


Bild 2.2.: Blockschaltbild eines Digital-Down-Converters mit analoger Vorstufe

Innerhalb des Digital-Down-Converters werden die Eingangssignale in zwei parallele Pfade aufgeteilt, mit denen anschließend eine Quadraturamplitudenmodulation durchgeführt wird, um dem Signal eine Phaseninformation aufzuprägen. Für diesen Vorgang werden zwei harmonische Schwingungen benötigt, die mit einem digitalen Oszillator (QDSO) erzeugt werden.

Die durch die Modulation entstandenen Doppelfrequenzanteile werden mit Hilfe eines Tiefpassfilters in beiden Pfaden eliminiert und das resultierende Signal dezimiert.

Die Ausgangssignale, I und Q, eines Digital-Down-Converters sind der In-Phase-Anteil (I) und der Quadratur-Anteil (Q). I und Q stellen die Komponenten des komplexen Tiefpasssignals dar.

Digital-Down-Converter können sowohl in Software als auch in Hardware realisiert werden. Realisierungen in Software können beispielsweise für digitale Signalprozessoren oder als Computerprogramme erstellt werden. Realisierungen in Hardware hingegen können als integrierte Schaltkreise[1], wie beispielsweise von der Firma *Intersil*, bezogen werden.

Softwarerealisierungen sind in der Regel weniger leistungsstark als Realisierungen in Hardware. ICs beinhalten mehrere Digital-Down-Converter, die extern zur Laufzeit konfiguriert werden müssen und weisen mittlerweile einen zu geringen Durchsatz auf, um den Anforderungen der modernen Funkpeiltechnik gerecht zu werden.

Die VHDL-Implementierung eines Digital-Down-Converters für einen FPGA erreicht einen höheren Datendurchsatz als herkömmliche ICs und ist portabel, so dass es möglich ist, einen Digital-Down-Converter direkt in den signalverarbeitenden Teil eines Gerätes einzubetten. Außerdem muss eine VHDL-Realisierung nicht zwingend zur Laufzeit konfiguriert werden, da bereits während der Kompilierung deren Verhalten festgelegt werden kann. Allerdings ist der Entwicklungsaufwand eines Digital-Down-Converters in VHDL höher als die Erstellung eines Programms, das einen fertigen Digital-Down-Converter mit einer SPI-Schnittstelle oder Ähnlichem konfiguriert.

Eine VHDL-Implementierung muss in einer möglichst schnellen Schaltung resultieren, um einen hohen Datendurchsatz zu erzielen. Neben der maximalen Taktfrequenz ist ein kompaktes Design wünschenswert. Kompaktheit und eine schnelle Taktfrequenz einer Schaltung stehen einander im Widerspruch, so dass in der Regel ein Kompromiss zwischen der resultierenden Größe einer Schaltung und deren maximaler Taktfrequenz gefunden werden muss.

Zu den Komponenten eines Digital-Down-Converters zählen, wie bereits erwähnt, Oszillatoren, Filter und Dezimatoren.

[1] Integrated circuit oder kurz IC

Theorie

Theorie

In diesem Teil der Diplomarbeit werden verschiedene Komponenten eines Digital-Down-Converters, Oszillatoren und Dezimationsfilter, theoretisch untersucht und miteinander verglichen.

Es werden vier verschiedene Möglichkeiten vorgestellt, Oszillatoren mit digitalen Schaltungen zu realisieren. Dazu zählen ein Lookuptable-Oszillator, ein IIR-Oszillator, ein Oszillator mit Taylorreihenentwicklung und der CORDIC-Prozessor.

Desweiteren werden zwei verschiedene Realisierungen von FIR-Filtern untersucht. Diese sind FIR-Dezimationsfilter für beliebige, ganzzahlige Dezimationsfaktoren und FIR-Halfbandfilter für Dezimationen um Faktor Zwei, die sich kaskadieren lassen.

Zuletzt werden CIC- und SCIC-Dezimatoren vorgestellt, mit denen sich große Dezimationsfaktoren ohne Multiplikationen realisieren lassen.

Die theoretischen Untersuchungen in diesem Teil bilden die Grundlage für die Synthese- und Simulationsergebnisse, die später miteinander verglichen werden, um einen hardwareoptimierten Digital-Down-Converter für bestimmte Rahmenbedingungen realisieren zu können.

3. Digitale Oszillatoren

Ein DDC führt eine Quadraturamplitudenmodulation des Eingangssignals durch. Dabei wird das reelle Eingangssignal in zwei Pfade aufgespaltet und mit zwei harmonischen Schwingungen moduliert, die einander um 90° phasenverschoben sind. Ein Digital-Down-Converter benötigt Cosinus- und Sinusschwingungen mit negativem Vorzeichen, um eine Down-Konvertierung zu ermöglichen. Durch die Quadraturamplitudenmodulation wird dem zunächst reellen Eingangssignal eine Phaseinformation aufgeprägt, wodurch ein komplexes Ausgangssignal resultiert.

In diesem Kapitel werden verschiedene Möglichkeiten vorgestellt, harmonische Schwingungen mit digitalen Oszillatoren zu erzeugen, mit denen die Quadraturamplitudenmodulation durchgeführt wird.

3.1. Anforderungen an den Oszillator

In einem Kommunikationssystem stellen Digital-Down-Converter in der Regel die erste Instanz des digitalen Schaltungsteils dar, weshalb eine minimale Verfälschung der im Signal enthaltenen Informationen erstrebenswert ist. Daher ist es wichtig, dass die Ausgangssignale wenig Phasenrauschen aufweisen, d.h. ein möglichst schmales Spektrum besitzen.

3.2. Lookuptable-Oszillator

Die einfachste Möglichkeit, einen Oszillator zu realisieren, ist, vorausberechnete Werte in einem Lookuptable abzulegen und der Reihe nach auszugeben. Dabei ist die Anzahl der zu speichernden Werte sowohl von der Ausgangsfrequenz als auch dem Verhältnis von Takt- und Ausgangsfrequenz abhängig.

Abgesehen von Rundungsfehlern während der Vorausberechnung der Funktionswerte, ist es mit einem Lookuptable-Oszillator möglich, optimale Ergebnisse zu erzielen, wobei der benötigte Hardwareaufwand unter Umständen sehr hoch sein kann.

3.2.1. Vorüberlegungen

Die Anzahl der benötigten Werte ist von der gewünschten Ausgangsfrequenz abhängig und ist mindestens Zwei, da die größtmögliche Ausgangsfrequenz $\pi T = \pi / f$ beträgt, wobei T die Abtastperiode darstellt.

Die Anzahl der Werte muss so geartet sein, dass sich eine Periodizität des Ausgangssignals ergibt. Sind Taktfrequenz und die gewünschte Ausgangsfrequenz in einem ganzzahligen Verhältnis, lässt sich diese Anforderung leicht erfüllen, indem

$$N = \frac{2\pi}{\omega T} \tag{3.1}$$

berechnet wird, wobei ω die gewünschte Ausgangsfrequenz im Verhältnis zur Taktfrequenz (2π) darstellt.

Sind beide Frequenzen in keinem ganzzahligen Verhältnis, müssen mehrere Perioden für die Berechnung in Betracht gezogen werden, bis N ganzzahlig ist oder sich der Fehler innerhalb der akzeptablen Toleranz befindet.

$$N + \varepsilon = k\,\frac{2\pi}{\omega T}, \tag{3.2}$$

wobei k eine natürliche Zahl größer als Eins ist. ε stellt den Fehler dar, der den Nachkommastellen der rechten Seite der Gleichung entspricht. k wird so lange vergrößert, bis der Fehler klein genug ist.

Für die Ansteuerung des Lookuptables wird ein Modulo-N-Zähler benötigt (Bild 3.1).

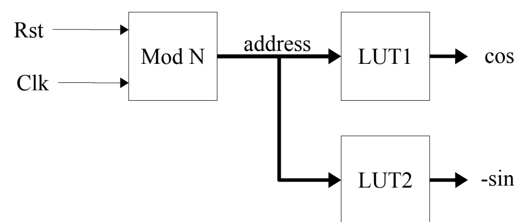


Bild 3.1.: Lookuptable-Oszillator

3.2.2. Hardwareoptimierung

Es genügt, die Änderungen der Funktionswerte zu speichern, wenn die Frequenz nicht zur Laufzeit geändert werden soll, so dass sich die Größe pro berechnetem Funktionswert besonders für hohe Ausgangsfrequenzen erheblich reduziert (Bild 3.2a). Die Bitanzahl für einen Eintrag im Lookuptable kann ermittelt werden, indem der maximale Werteunterschied um die Nullstellen herum ermittelt wird. Die Größe des Lookuptables für die nicht optimierte Realisierung berechnet sich mit

$$S = N \cdot B \text{ Bit}, \tag{3.3}$$

wohingegen sich die Größe des hardwareoptimierten LUTs mit

$$S_{opt} = N \left\lceil log_2 \left| int \left\{ sin\,(2\pi f/F)\; 2^{B-1} \right\} \right| \right\rceil \text{ Bit} \tag{3.4}$$

berechnet wird, wobei f die Ausgangs- und F die Taktfrequenz darstellen. B stellt jeweils die Auflösung der Signale in Bit dar.

Da sich die Funktionswerte von Sinus und Cosinus wiederholen, kann der Hardwareaufwand reduziert werden, indem nur ein Viertel einer vollen Periode berechnet wird $(0 \ldots \pi/2)$. Dabei müssen die Taktfrequenz und die Ausgangsfrequenz so im Verhältnis stehen, dass der Fehler klein bleibt (Bild 3.2b).

Eine Phasenverschiebung um 90° lässt sich mit einem Hilbert-Transformator (Allpass) realisieren, durch den sich ebenfalls ein Lookuptable einsparen lässt (Bild 3.2c).

In Bild 3.2d werden alle Hardwareoptimierungsmöglichkeiten vereint.

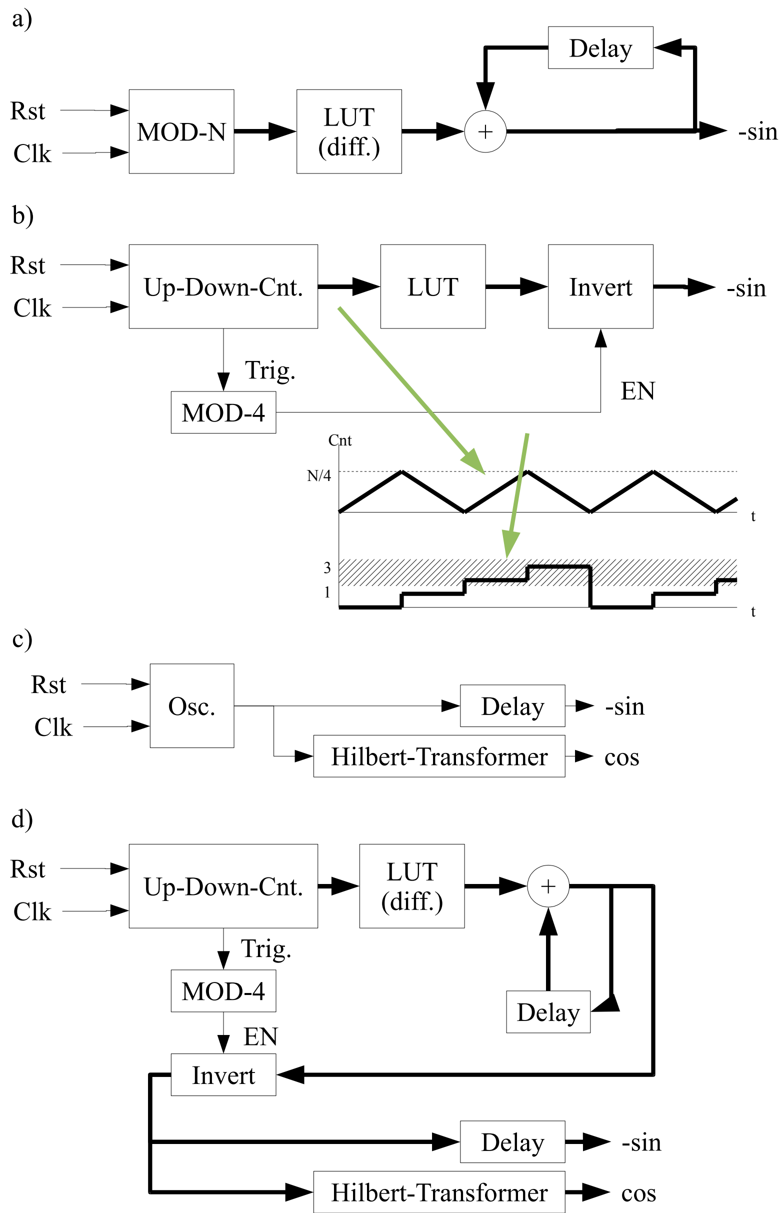


Bild 3.2.: Hardwareoptimierungen von Lookuptable-Oszillatoren

3.2.3. Vor- und Nachteile des Lookuptable-Oszillators

Der Lookuptable-Oszillator weist folgende Vor- und Nachteile auf.

Vorteile:

- Die Implementierung ist einfach und es werden wenig Logikzellen benötigt
- Der Oszillator ist stabil
- Es lassen sich Sinus- und Cosinus-Schwingungen gleichzeitig erzeugen
- Die Berechnung der Funktionswerte erfolgt nicht zur Laufzeit
- Ein Lookuptable-Oszillator ist für hohe Taktraten geeignet und bietet ein hohes Optimierungspotenzial
- Inklusive aller Optimierungen kann ggf. ein Kompromiss zwischen Hardwareaufwand und Qualität der Signale gefunden werden
- Durch Vergrößerung der Schrittbreite des Modulo-N-Zählers kann die Ausgangsfrequenz verändert werden

Nachteile:

- Die Frequenz kann zur Laufzeit nicht beliebig geändert werden
- Pro Oszillator wird mindestens ein ROM des FPGAs in Anspruch genommen
- Der Oszillator ist nicht für jede Ausgangsfrequenz gleich gut geeignet, da teilweise der benötigte Speicherplatz sehr groß ist

3.3. IIR-Oszillator

IIR-Systeme zweiter Ordnung sind in der Lage, zu oszillieren, wenn beide Pole auf dem Einheitskreis liegen [PR96]. Basierend auf dieser Tatsache, kann ein IIR-basierter Oszillator in gekoppelter Form realisiert werden, der Sinus und Cosinus parallel ausgibt, aber dabei minimalen Hardwareaufwand benötigt.

Die Berechnung der Funktionswerte beruht hierbei auf den vorherigen Funktionswerten, wodurch eine Fehlerfortpflanzung resultiert, die im Folgenden mit verschiedenen Ansätzen kompensiert wird.

3.3.1. Herleitung der Schaltung

Um kontinuierliche Ausgangssignale zu erzeugen, muss taktweise ein gleichgroßer Winkel (ω_0) auf die Argumente von Sinus und Cosinus addiert werden.

$$cos\ n\ \omega_0 = cos(\{n-1\}\omega_0 + \omega_0) \tag{3.5}$$

$$sin\ n\ \omega_0 = sin(\{n-1\}\omega_0 + \omega_0), \tag{3.6}$$

wobei $n = 0, 1, 2\dots$ ist. Die Argumente beider Winkelfunktionen lassen sich ebenfalls durch Summen ausdrücken, wodurch unter Ausnutzung der Additionstheoreme für Winkelsummen

$$cos(\alpha + \beta) = cos\ \alpha\ cos\beta - sin\ \alpha\ sin\beta \tag{3.7}$$

$$sin(\alpha + \beta) = sin\ \alpha\ cos\beta + cos\ \alpha\ sin\beta \tag{3.8}$$

folgende Differenzengleichungen resultieren, die in einer Schaltung (Bild 3.3) realisiert werden können.

$$\begin{pmatrix} y_c(n) \\ y_s(n) \end{pmatrix} = \begin{pmatrix} cos\ \omega_0 & -sin\ \omega_0 \\ sin\ \omega_0 & cos\ \omega_0 \end{pmatrix} \begin{pmatrix} y_c(n-1) \\ y_s(n-1) \end{pmatrix} \tag{3.9}$$

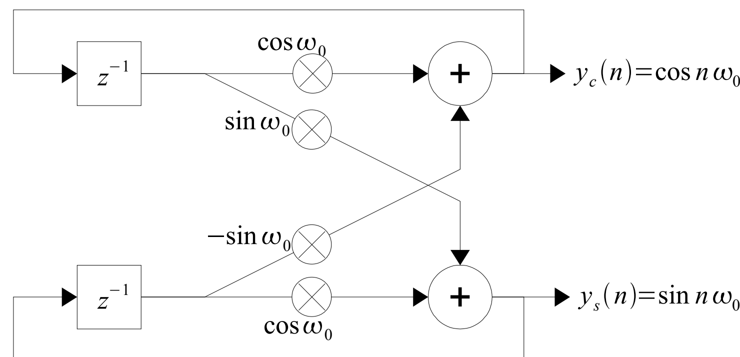


Bild 3.3.: Blockschaltbild des IIR-Oszillators

Das System besitzt keine Eingänge, muss aber initialisiert werden, um zu oszillieren. Hierzu ist $y_c(-1) = A\ cos\ \omega_0$ und $y_s(-1) = -A\ sin\ \omega_0$ einzuhalten, wobei A die maximale Amplitude des Ausgangssignals darstellt.

Die resultierende Schaltung besteht aus vier Multiplizierern, zwei Addierern und zwei Verzögerungsgliedern.

3.3.2. Ermittlung der Pole

Es kann gezeigt werden, dass der Oszillator konjugiert-komplexe Polstellen bei dessen Initialisierungswerten besitzt. Somit wirkt sich die Berechnung der Koeffizienten direkt auf die Stabilität des Oszillators aus.

Hierzu werden zunächst erneut die geometrischen Funktionen betrachtet.

$$\begin{aligned} cos(n\omega_0 + \omega_0) &= cos\ n\omega_0\ \underbrace{cos\ \omega_0}_{k_c} - sin\ n\omega_0\ \underbrace{sin\ \omega_0}_{k_s} \\ &= k_c\ cos\ n\omega_0\ - k_s\ sin\ n\omega_0 \\ sin(n\omega_0 + \omega_0) &= sin\ n\omega_0 cos\omega_0 + cos\ n\omega_0\ sin\ \omega_0 \\ &= k_c\ sin\ n\omega_0\ + k_s\ cos\ n\omega_0 \end{aligned} \tag{3.10}$$

Aus der abgeleiteten Gleichung lassen sich erneut die Differenzengleichungen des Systems bestimmen.

$$y_c(n) = k_c\ y_c(n-1) - k_s\ y_s(n-1) + S \tag{3.11}$$

$$y_s(n) = k_c\ y_s(n-1) + k_s\ y_c(n-1) \tag{3.12}$$

S dient sinnbildlich zum Anstoßen der Oszillation. Mit Hilfe der z-Transformation werden die Gleichungen in den Bildbereich transformiert.

$$Y_c = k_c\ Y_c\ z^{-1} - k_s\ Y_s\ z^{-1} + S \tag{3.13}$$

$$Y_s = k_c\ Y_s\ z^{-1} + k_s\ Y_c\ z^{-1} \tag{3.14}$$

Daraus folgt

$$Y_c = \frac{S - k_s\ Y_s\ z^{-1}}{1 - k_c\ z^{-1}} \tag{3.15}$$

$$Y_s = \frac{k_s\ Y_c\ z^{-1}}{1 - k_c\ z^{-1}}. \tag{3.16}$$

Die Gleichungen sind voneinander abhängig, also muss eine in die andere eingesetzt werden.

$$Y_c = \frac{S - k_s\ z^{-1}}{1 - k_c\ z^{-1}} \left[\frac{k_s\ z^{-1}\ Y_c}{1 - k_c\ z^{-1}} \right] = \frac{1}{1 - k_c\ z^{-1}} \left(S - k_c\ z^{-1} S - k_s^2\ z^{-2}\ Y_c \right) \tag{3.17}$$

Daraus folgt

$$Y_c = \frac{1 - k_c\ z^{-1}}{1 - 2\ k_c\ z^{-1} + z^{-2}} \cdot S \tag{3.18}$$

$$Y_s = \frac{k_s\ z^{-1}}{1 - 2\ k_c\ z^{-1} + z^{-2}}. \tag{3.19}$$

Aus einem Nenner werden die Pole bestimmt.

$$z^2 - 2\ z\ k_c + 1 = 0 \rightarrow z_p = k_c \pm \sqrt{k_c^2 - 1} \tag{3.20}$$

Da

$$k_c^2 + k_s^2 = 1 \rightarrow -k_s^2 = k_c^2 - 1 \tag{3.21}$$

folgt

$$z_p = k_c \pm \sqrt{-k_s^2} = k_c \pm j\ k_s. \tag{3.22}$$

Die Pole des Systems liegen also direkt bei $cos\ \omega_0 \pm j\ sin\ \omega_0$ und werden durch die Faktoren der Multiplikation repräsentiert. Die möglichen Lagen der Pole im ersten Quadranten der komplexen Ebene werden in Bild 3.4 dargestellt. Alle Pole befinden sich in einem linearen Raster mit dem Abstand eines LSBs zueinander. Die meisten Pole liegen nicht auf dem Einheitskreis, weshalb bei der Wahl der Pole ein Kompromiss gemacht werden muss und der Pol verwendet wird, der am wenigsten vom Einheitskreis entfernt liegt. Durch die unexakte Lage der Pole resultieren sowohl Fehler in der Amplitude als auch in der Phase eines IIR-Oszillators.

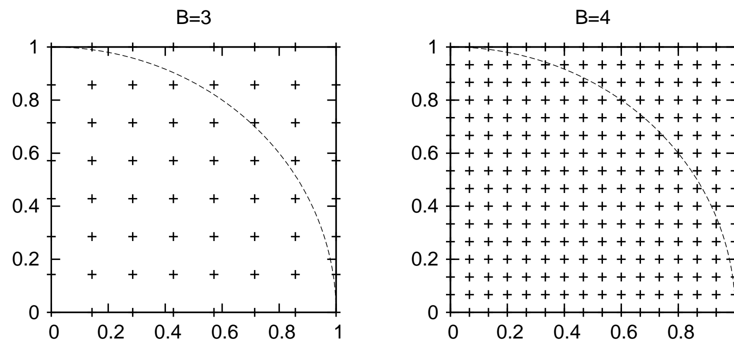


Bild 3.4.: Mögliche Lagen der Pole eines IIR-Systems in gekoppelter Form

3.3.3. Stabilisierung

In den meisten Fällen ist es unter Verwendung von Festpunktarithmetik nicht möglich, einen stabilen IIR-Oszillator zu realisieren. In Bild 3.5 werden ein optimaler Signalverlauf und zwei Signalverläufe dargestellt, die durch Fehlerfortpflanzung hervorgerufen werden. Links ist der Betrag der Pole kleiner als Eins, so dass die Amplitude schnell abklingt. Rechts ist der Betrag der Pole größer als Eins, so dass die Amplitude aufschwingt. Beide Verläufe sind unbrauchbar, wenn ein langzeitstabiler Oszillator benötigt wird. Unter Umständen kann eine abklingende Amplitude toleriert werden, wenn das Signal nur eine bestimmte Zeit benötigt wird, wie beispielsweise in der digitalen Klangsynthese [WGO].

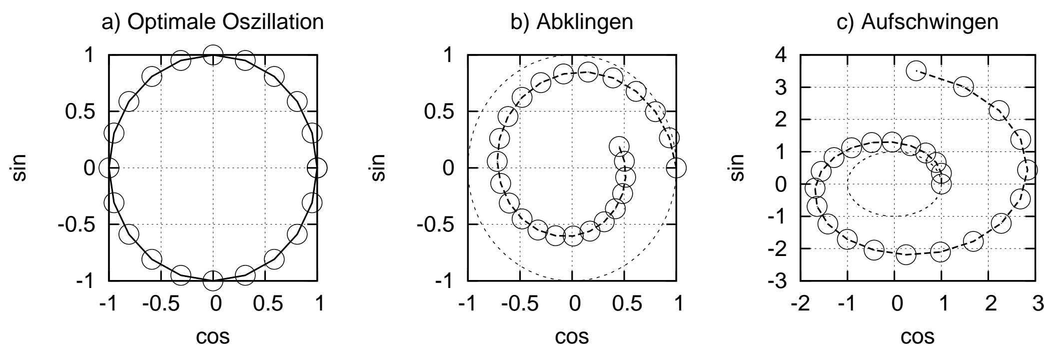


Bild 3.5.: Abklingen und Aufschwingen eines IIR-Oszillators

Um einen langzeitstabilen IIR-Oszillator zu realiseren, müssen Aufschwingen und Abklingen kompensiert werden, so dass der Verlauf der Oszillation möglichst dem optimalen Verlauf gleicht. Hierzu werden drei Ansätze verfolgt:

1. Neuinitialisieren des Oszillators nach N Schritten
2. Neuinitialisieren bei Nulldurchgängen eines Signals
3. Polverschiebung

3.3.4. Stabilisierung durch kontinuierliches Neuinitialisieren

Der IIR-Oszillator liefert langzeitstabile Signale, wenn dieser nach N Schritten auf den Ausgangspunkt zurückgesetzt wird. Dabei ist es sinnvoll, Pole zu wählen, die sich innerhalb des Einheitskreises befinden, um ein Abklingen der Amplitude während einer Periode zu erzwingen. Am Punkt der Neuinitialisierung tritt ein Sprung in der Amplitude auf, der im Spektrum Störungen hervorruft. Dieser Sprung lässt sich aber durch eine geeignete Wahl der Bit-Auflösung reduzieren. Der Verlauf der Signale wird in Bild 3.6 dargestellt, links für $N \leq f/F$ und rechts für $N > f/F$, so dass mehrere Perioden benötigt werden, um eine Periodizität zu erreichen.

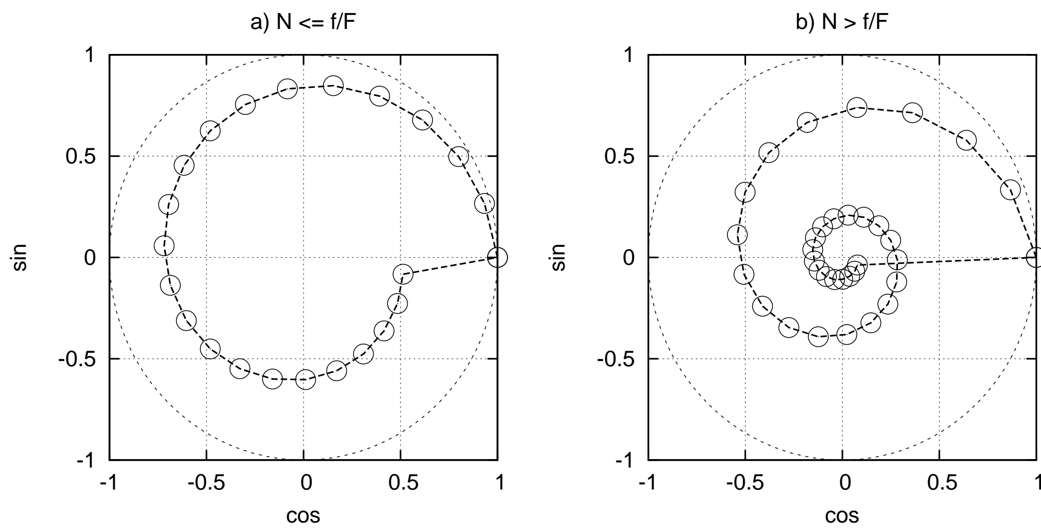


Bild 3.6.: Verlauf der Signale bei Neuinitialisierung durch Mitzählen

Vor- und Nachteile der Stabilisierungsmethode durch Mitzählen

Der IIR-Oszillator mit zählerbasierter Neuinitialisierung weist folgende Vor- und Nachteile auf.

Vorteile:

- Die Implementierung ist einfach
- Es wird neben der Grundschaltung nur ein weiterer Zähler benötigt, der als One-Hot-Zähler ausgelegt werden kann
- Der Hardwareaufwand ist moderat

Nachteile:

- Es entstehen Sprünge im Signal, die das Spektrum stören
- Der zählerbasierte IIR-Oszillator ist nicht für alle Frequenzen gleich gut geeignet

3.3.5. Stabilisierung durch Neuinitialisierung bei Nulldurchgängen

Für die Neuinitialisierung kann ebenfalls eine Nulldurchgangsüberwachung realisiert werden, so dass eine Neuinitialisierung beider Signale bei Detektion des Nulldurchgangs eines Signals durchgeführt wird (Bild 3.7). Hierbei werden zwei Sprünge der Amplitude pro Periode verursacht (Bild 3.8), wodurch ebenfalls das Spektrum gestört wird. Abhängig von der Breite des Detektionsbereichs entsteht ebenfalls ein Fehler in der Phase, wodurch Phasenrauschen hervorgerufen wird.

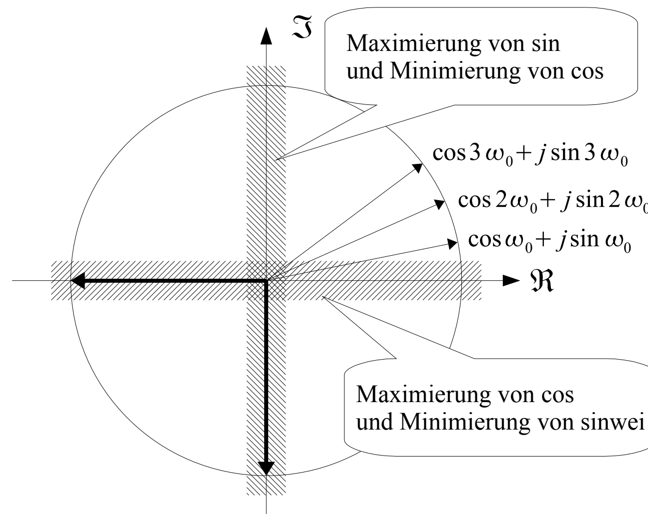


Bild 3.7.: Neuinitialisierung des IIR-Oszillators bei Nulldurchgängen der Signale

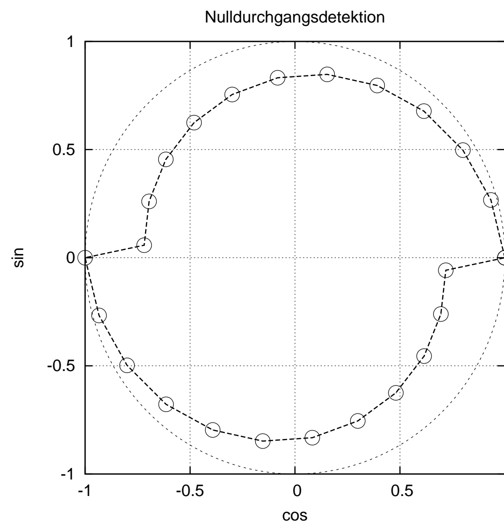


Bild 3.8.: Neuinitialisierung mit Nulldurchgangsdetektion

Vor- und Nachteile der Stabilisierungsmethode durch Nulldurchgangsdetektion

Der IIR-Oszillator mit Nulldurchgangsüberwachung weist folgende Vor- und Nachteile auf.

Vorteile:

- Es entstehen zwei Stellen zur Neuinitialisierung, so dass sich Sprünge verkleinern und sich die resultierenden Oberschwingungen im Spektrum voneinander entfernen

Nachteile:

- Es entstehen Sprünge im Ausgangssignal, die das Spektrum stören
- Bedingt duch die Definition eines Bereichs, der einen Nulldurchgang darstellt, entstehen weitere Fehler, die abhängig von der Größe des Bereichs groß werden können

3.3.6. Stabilisierung durch Polverschiebung

Es ist möglich, den Oszillator zu stabilisieren, indem während des Betriebs dessen Pole so verschoben werden, dass sich Aufschwingen und Abklingen gegenseitig ausgleichen. Als Resultat ergibt sich eine Schwankung in der Amplitude, die sich über mehrere Takte verteilt, aber nur wenig auf das Spektrum auswirkt. Ein Vorteil gegenüber den vorherigen Ansätzen ist, dass die Überwachung häufiger stattfindet, so dass ein geringeres Abklingen oder Aufschwingen in einer Amplitude zu erwarten ist. Das entstehende Spektrum der Ausgangssignale ist besser als die der anderen Ansätze.

Die idealen Pole des IIR-basierten Oszillators sind

$$p_{1,2} = cos\ \omega_0 \pm j\ sin\ \omega_0. \tag{3.23}$$

Die quantisierten Pole sind

$$\overline{p}_{1,2} = \overline{cos}\ \omega_0 \pm j\ \overline{sin}\ \omega_0, \tag{3.24}$$

wobei die Quantisierung durch den Strich über den Variablen angezeigt wird.

Der Betrag des quantisierten Pols ist somit

$$A = \sqrt{\overline{cos}^2 \omega_0 + \overline{sin}^2 \omega_0}. \tag{3.25}$$

Die Koeffizienten liegen hierbei bereits in quantisierter Form vor, wobei A kleiner als Eins ist, wodurch die Amplitude im Laufe der Zeit abklingt. Um das Abklingen auszugleichen, werden zwei weitere Pole erzeugt, deren Betrag $1/A$ ist.

$$\overline{p}'_{1,2} = \frac{1}{A}\{\overline{cos}\ \omega_0 \pm j\ \overline{sin}\ \omega_0\}. \tag{3.26}$$

Diese Pole liegen außerhalb des Einheitskreises und erzeugen eine aufschwingende Amplitude. Die Lage der Pole wird in Bild 3.9 abgebildet.

Während des Betriebs wird regelmäßig das Quadrat der Amplitude errechnet und mit dem Maximalwert der Quantisierung verglichen. Während der Berechnung schwingt die Amplitude entweder auf oder klingt ab. Die Dauer der Berechnung wirkt sich also auf den Fehler der Amplitude direkt aus. In der VHDL-Implementierung wird ein serieller Multiplizierer verwendet, der für die Multiplikation zweier 16-Bit-Werte 17 Takte benötigt. Mit dieser Anordnung ließen sich gute Ergebnisse erzielen.

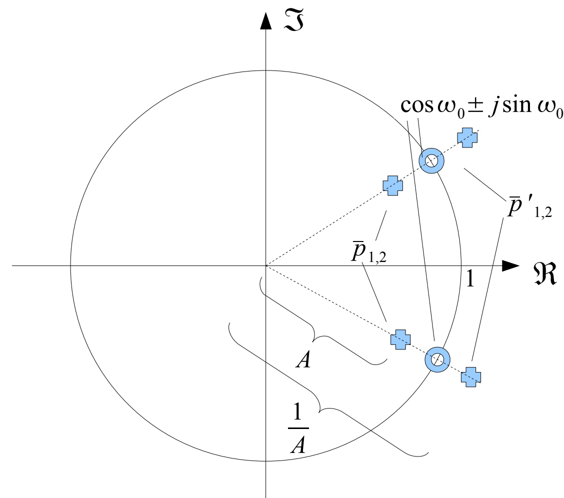


Bild 3.9.: Lage der Pole eines IIR-basierten adaptiven Oszillators

Abhängig von der Lage des komplexen Zeigers bezogen auf den Einheitskreis wird zwischen den Polpaaren umgeschaltet, so dass die Amplitude abklingt, wenn sie größer als Eins ist und aufschwingt, wenn sie kleiner als Eins ist. Bild 3.10 zeigt den Signalverlauf.

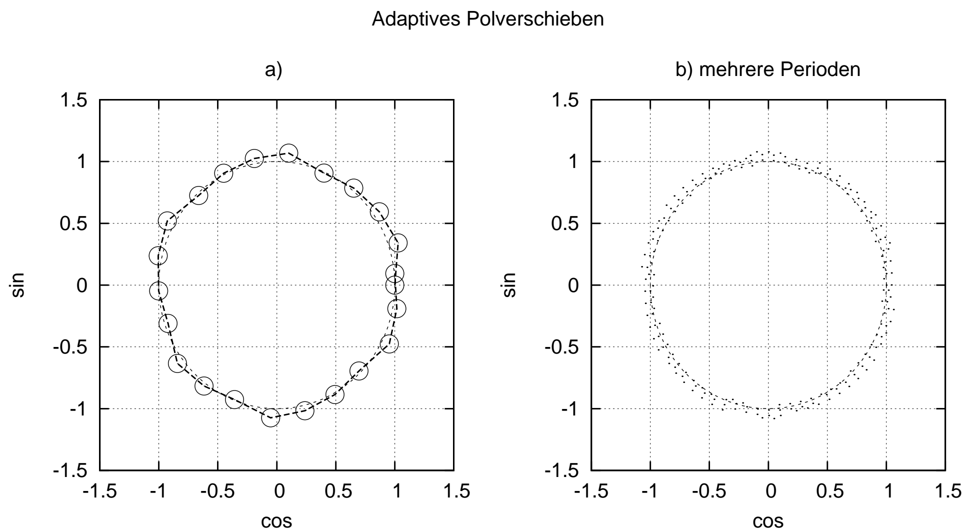


Bild 3.10.: IIR-Oszillator mit Polverschiebung

Vor- und Nachteile der Stabilisierungsmethode durch adaptives Polverschieben

Der adaptive IIR-Oszillator, der Auf- und Abschwingen der Amplitude durch Verschiebung der Pole zur Laufzeit ausgleicht, weist folgende Vor- und Nachteile auf.

Vorteile:

- Sprünge der Ausgangssignale werden vermieden, da sich Auf- und Abschwingen gegenseitig abwechseln
- Das Spektrum ist weniger verfälscht als bei anderen Stabilisierungsansätzen
- Der Hardwareaufwand ist moderat

Nachteile:

- Die Ausgangsamplituden sind teilweise größer als Eins und schwanken
- Für die Berechnung der Amplitude wird ein Multiplizierer benötigt, der unter Umständen aber seriell realisiert werden kann

3.3.7. Genauigkeit und Frequenz

Wird der IIR-Oszillator mit endlicher Genauigkeit realisiert, ist die Genauigkeit der Berechnungen von der Ausgangsfrequenz abhängig, weil die Koeffizienten zur Berechnung eines Folgewertes von der Ausgangsfrequenz abhängen.

$sin\ \omega_0$ ist im Bereich 0 bis π klein und $cos\ \omega_0$ in der Nähe von $\pi/2$, so dass an diesen Stellen eine geringere Ausnutzung der Registerdynamik resultiert, worunter die Genauigkeit der Multiplikationen leidet. Grundsätzlich kann ein IIR-Oszillator gar nicht für die Frequenzen $w_i = k \cdot \pi/2$ verwendet werden, da jeweils einer der Koeffizienten Null ist.

Die erreichbare Genauigkeit in Bit für normierte Multiplikationen wird in Bild 3.11 dargestellt. Optimale Ausgangsfrequenzen für einen IIR-Oszillator befinden sich somit in der Nähe von $\pi/4$ und $3\pi/4$.

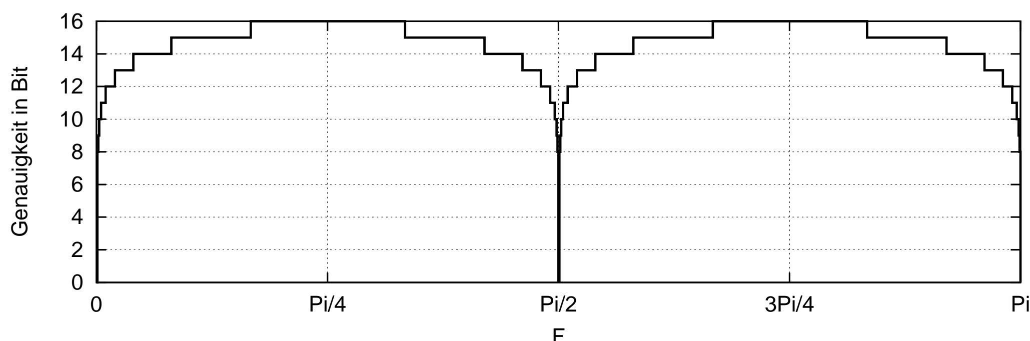


Bild 3.11.: Genauigkeit des IIR-Oszillators in Abhängigkeit der Ausgangsfrequenz

3.3.8. Simulation

Es wurden alle bisher vorgestellten Oszillatoren für verschiedene Ausgangsfrequenzen simuliert und die Ergebnisse vergleichend in den Bildern 3.12 und 3.13 dargestellt. Es lässt sich erkennen, dass der Verlauf der Amplitude im Vergleich zu dem zählenden bzw. dem überwachten Oszillator insbesondere für kleine Frequenzen weit überlegen ist.

3.3.9. Vor- und Nachteile des IIR-Oszillators

Die gemeinsamen Vor- und Nachteile der in dieser Diplomarbeit vorgestellten IIR-Oszillatoren können zusammengefasst werden.

Vorteile:

- Der Hardwareaufwand aller IIR-Oszillatoren ist gering
- Sinus und Cosinus werden gleichzeitig mit einer Schaltung erzeugt
- Die adaptive Realisierung erzeugt ein sauberes Spektrum

Nachteile:

- Rechenfehler pflanzen sich fort und müssen durch regelmäßige Neuinitialisierung kompensiert werden
- Die Nulldurchgangsüberwachung benötigt lange kombinatorische Pfade und reduziert die maximale Taktrate (s. Synthese)
- Neuinitialisierungen erzeugen Störungen im Spektrum
- Die Genauigkeit des Ausgangssignals ist von der Ausgangsfrequenz abhängig
- Der IIR-Oszillator ist weniger universell einsetzbar als der Lookuptable-Oszillator

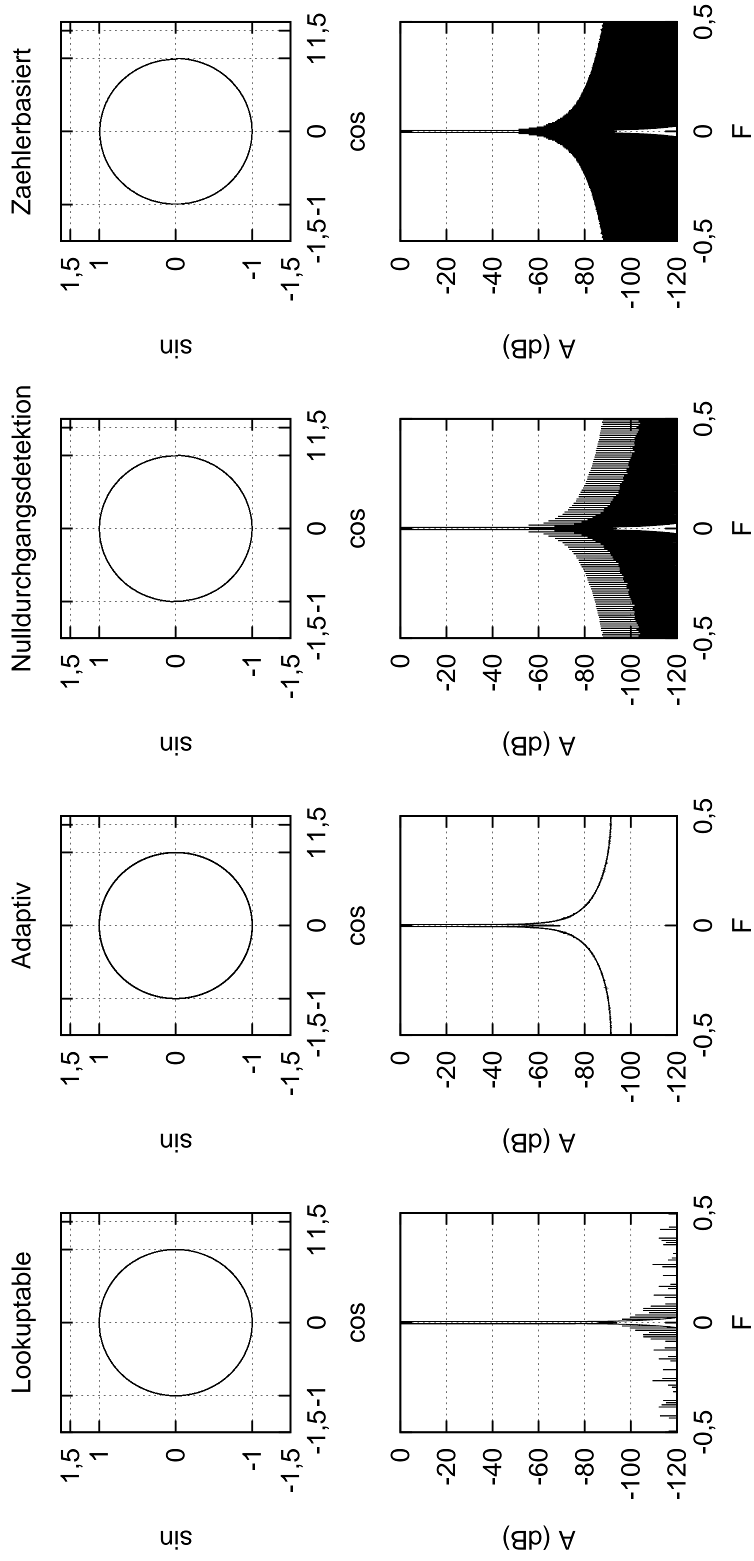


Bild 3.12.: Ausgangssignale und Spektren verschiedener Oszillatoren für die Frequenz $\pi/100$ bei 16 Bit Auflösung

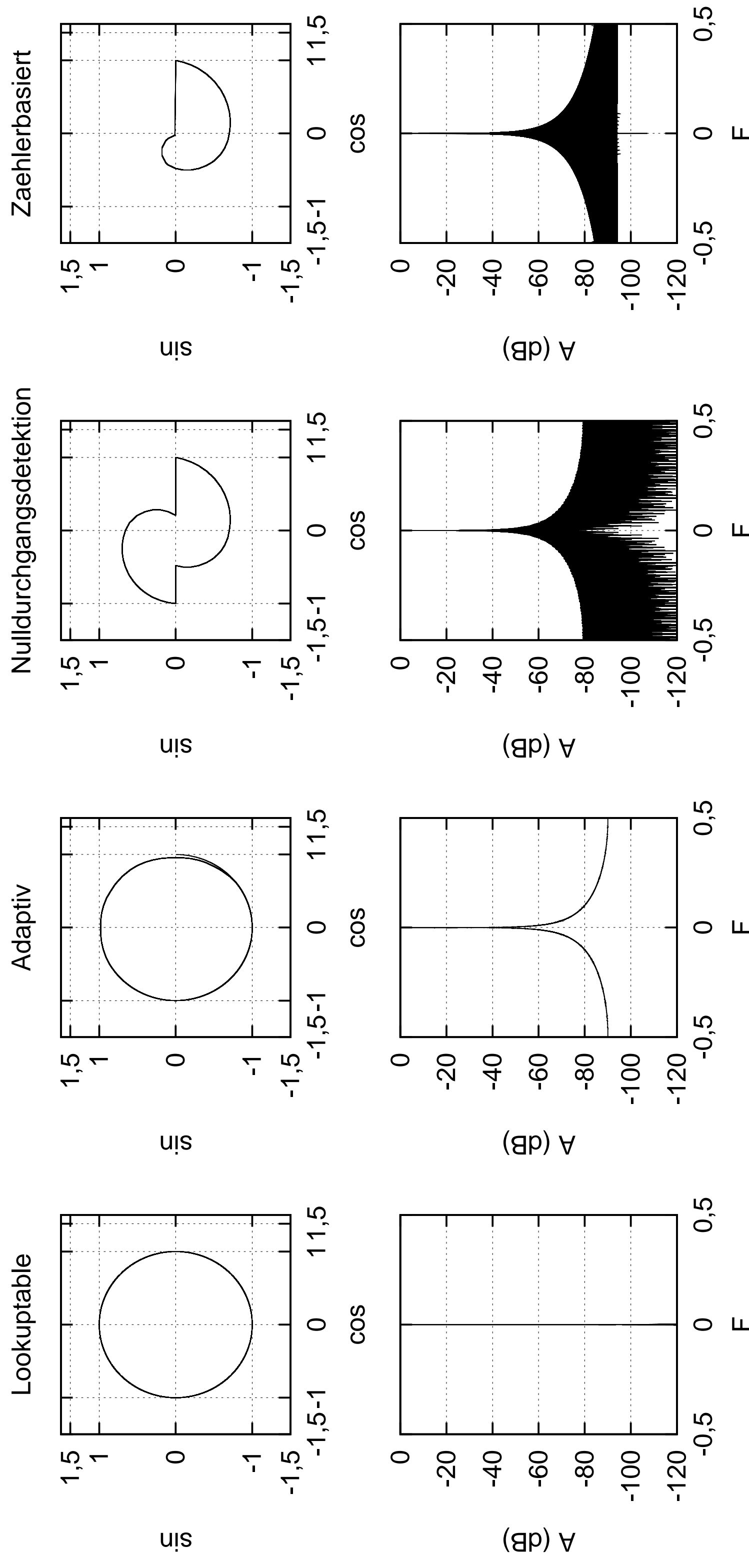


Bild 3.13.: Ausgangssignale und Spektren verschiedener Oszillatoren für die Frequenz $\pi/10000$ bei 16 Bit Auflösung

3.4. Taylorreihen-Oszillator

Mit Hilfe der Taylorreihenentwicklung ist es möglich, jede beliebig oft ableitbare mathematische Funktion durch eine Potenzreihe anzunähern. Darunter fallen auch Sinus und Cosinus, weshalb ein Taylorreihen-Oszillator entwickelt werden kann.

Taylorreihen sind in gängiger Fachliteratur wie [PAPFS] zu finden, aber werden der Vollständigkeit halber hier hergeleitet.

3.4.1. Herleitung der Taylorreihen für harmonische Schwingungen

Allgemein lautet eine Taylorreihe

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n, \tag{3.27}$$

wobei $f^{(n)}$ für die n-te Ableitung der Funktion $f(x)$ steht. Wenn f eine analytische Funktion ist, konvergiert die Reihe am Punkt $f(x)$, wodurch es möglich ist, den Funktionswert zu berechnen. a ist der sogenannte Entwicklungspunkt der Reihe. Häufig ist es hinreichend, den Wert 0 zu verwenden, wobei die Reihe dann Mac Laurinsche Reihe genannt wird.

Enwicklung von Cosinus und Sinus in eine Taylorreihe:

Zunächst müssen die Ableitungen von Sinus und Cosinus aufgelistet werden.

$$\begin{aligned}
\frac{d\ sin\ \omega}{d\ \omega} &= cos\ \omega \\
\frac{d^2\ sin\ \omega}{d\ \omega^2} &= \frac{d\ cos\ \omega}{d\ \omega} = -sin\ \omega \\
\frac{d^3\ sin\ \omega}{d\ \omega^3} &= \frac{d\ (-sin\ \omega)}{d\ \omega} = -cos\ \omega \\
\frac{d^4\ sin\ \omega}{d\ \omega^4} &= \frac{d\ (-cos\ \omega)}{d\ \omega} = sin\ \omega \\
\frac{d^5\ sin\ \omega}{d\ \omega^5} &= \frac{d\ sin\ \omega}{d\ \omega} = cos\ \omega \\
&\dots
\end{aligned} \tag{3.28}$$

Es ist zu erkennen, dass die vierte Ableitung der Sinusfunktion dessen Urspungsfunktion entspricht.

Wird die Cosinus-Funktion mehrmals abgeleitet, ergeben sich ähnliche Ergebnisse.

$$\begin{aligned}
\frac{d\;cos\;\omega}{d\;\omega} &= -sin\;\omega \\
\frac{d^2\;cos\;\omega}{d\;\omega^2} &= \frac{d\;(-sin\;\omega)}{d\;\omega} = -cos\;\omega \\
\frac{d^3\;cos\;\omega}{d\;\omega^3} &= \frac{d\;(-cos\;\omega)}{d\;\omega} = sin\;\omega \\
\frac{d^4 cos\;\omega}{d\;\omega^4} &= \frac{d\;sin\;\omega}{d\;\omega} = cos\;\omega \\
\frac{d^5\;cos\;\omega}{d\;\omega^5} &= \frac{d\;cos\;\omega}{d\;\omega} = -sin\;\omega \\
&\ldots
\end{aligned} \tag{3.29}$$

Für die Taylorreihenentwicklung ist es nötig, die Funktionswerte der n-ten Ableitungen an dem Entwicklungspunkt zu berechnen. Der Enwicklungspunkt a wird zu 0 gesetzt, um die Berechnungen zu vereinfachen.

Für die folgenden Berechnungen soll eine vereinfachte Schreibweise gelten

$$\frac{d^{(n)} cos\;\omega}{d\;\omega^n} = cos^{(n)}\omega. \tag{3.30}$$

Das Ergebnis repräsentiert jeweils die n-te Ableitung der jeweiligen Funktion.

$$\begin{aligned}
\omega &= a = 0 \\
sin^{(0)}(0) &= 0 \\
sin^{(1)}(0) &= cos(0) = 1 \\
sin^{(2)}(0) &= -sin(0) = 0 \\
sin^{(3)}(0) &= -cos(0) = -1 \\
sin^{(4)}(0) &= sin^{(0)}(0) = 0 \\
&\ldots
\end{aligned} \tag{3.31}$$

Jede zweite Ableitung von Sinus ist 0 und die übrigen Funktionswerte wechseln zwischen -1 und 1. Die Ergebnisse für die Ableitungen der Cosinus-Funktion an dem Entwicklungspunkt $a = 0$ lauten

$$\begin{aligned}
\omega &= a = 0 \\
cos^{(0)}(0) &= 1 \\
cos^{(1)}(0) &= -sin(0) = 0 \\
cos^{(2)}(0) &= -cos(0) = -1 \\
cos^{(3)}(0) &= sin(0) = 0 \\
cos^{(4)}(0) &= cos^{(0)}(0) = 1 \\
&\ldots.
\end{aligned} \tag{3.32}$$

Da sowohl für Sinus als auch für Cosinus jede zweite Ableitung am Entwicklungspunkt 0 ist, vereinfachen sich die zu entwickelnden Reihen. Für Sinus müssen alle (2n+1)-ten Reihenglieder berücksichtigt werden und für Cosinus alle 2n-ten Glieder, wobei $n = 0, 1, 2, 3 \ldots \infty$. Die Reihenentwicklung für

Cosinus lautet

$$cos(x) = \sum_{n=0}^{\infty} \frac{cos^{(n)}(0)}{(2n)!} x^{2n} = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}. \quad (3.33)$$

Das sich wechselnde Vorzeichen der 2n-ten Ableitungen der Cosinus-Funktion kann durch $(-1)^n$ dargestellt werden und vereinfacht den Ausdruck erneut. Für die Entwicklung der Sinus-Reihe müssen alle (2n+1)-ten Glieder betrachtet werden. Daher wird hier das n der ursprünglichen Reihe durch (2n+1) ersetzt. Die Reihe für die Sinus-Funktion lautet

$$sin(x) = \sum_{n=0}^{\infty} \frac{sin^{(n)}(0)}{(2n+1)!} x^{2n} = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}. \quad (3.34)$$

Wieder lässt sich der Ausdruck weiter vereinfachen, indem das ebenfalls wechselnde Vorzeichen der Ableitungen der Sinus-Funktion durch $(-1)^n$ dargestellt wird. Die ersten ausgeschriebenen Glieder der jeweiligen Reihen lauten

$$cos(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!} =$$
$$\frac{x^0}{0!} - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots = 1 - \frac{x^2}{2} + \frac{x^4}{24} \ldots \quad (3.35)$$

und

$$sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} =$$
$$\frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots = x - \frac{x^3}{6} + \frac{x^5}{120} \ldots. \quad (3.36)$$

Wenn die unendliche Reihe in eine endliche Reihe überführt wird, indem die obere Grenze durch (N-1) ersetzt wird, kann das Ergebnis angenähert werden. Es wird im folgenden Teil gezeigt, dass für endliche Genauigkeit für typische Anwendungen eine relativ geringe Iterationsanzahl nötig ist.

3.4.2. Genauigkeit und Iterationsanzahl

Die Genauigkeit der Berechnung einer Taylorreihe ist nur von der Iterationsanzahl abhängig. Mit beliebiger Iterationanzahl kann somit eine beliebig hohe Genauigkeit erreicht werden. Tabelle 3.1 beinhaltet den Verlauf der Fehler für die Reihen von Sinus und Cosinus in Abhängigkeit der Schrittanzahl. Der Fehler ist für die Sinusreihe geringer, da dessen erstes Glied x ist, und die Sinusfunktion im Bereich um Null linear verläuft, während das erste Glied der Cosinusreihe 1 ist.

Eine gegebene Genauigkeit (B) in Bit wird dann erreicht, wenn das N-te Glied eine geringere Änderung im Ergebnis als ein halbes LSB hervorruft.

$$LSB = \frac{1}{2^B} \quad (3.37)$$

Bild 3.14 beinhaltet den Verlauf der Iterationsanzahl in Abhängigkeit der zu erreichenden Genauigkeit der Taylorsinusreihe für die Frequenzen 0 bis $\{\pi/2$, π bzw. $2\pi\}$. In Tabelle 3.2 werden die Werte tabellarisch aufgelistet.

Ein optimales Ergebnis für die Berechnung von Sinus und Cosinus bei gegebener Genauigkeit kann aus der Sinus-Reihe gewonnen werden, wobei der Wertebereich auf $(0 \ldots \pi/2)$ beschränkt wird und alle anderen Werte durch Umrechnung gewonnen werden.

N	Err(cos)	Err(sin)	N	Err(cos)	Err(sin)
1	1.000000E0	5.707963E-1	6	4.647660E-7	5.625895E-8
2	2.337006E-1	7.516777E-2	7	6.321469E-9	6.627803E-10
3	1.996896E-2	4.524856E-3	8	6.513363E-11	6.023182E-12
4	8.945230E-4	1.568986E-4	9	5.260020E-13	4.374279E-14
5	2.473728E-5	3.542584E-6	10	3.438047E-15	1.110223E-16

Tabelle 3.1.: Fehler der Taylorreihe für Sinus und Cosinus in Abhängigkeit der Iterationsanzahl N

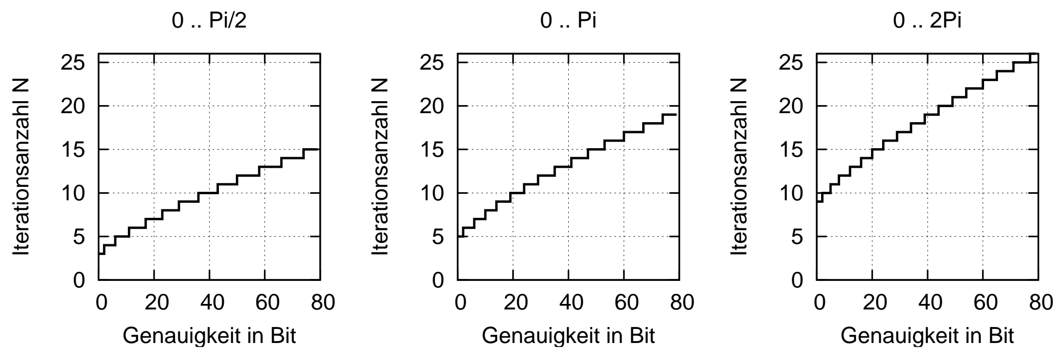


Bild 3.14.: Benötigte Iterationsanzahl N in Abhängigkeit der benötigten Genauigkeit in Bit für die Frequenzen 0 bis $\{\pi/2, \pi$ und $2\pi\}$

3.4.3. Implementierbarkeit

Um Sinus- bzw. Cosinus-Signale zu erzeugen, ist eine Hardwareimplementierung mit Taylorreihenentwicklung weniger geeignet, da komplizierte Berechnungen (Potenzieren, Fakultät, etc.) benötigt werden. Es lassen sich wesentlich einfachere Wege finden, um dies zu tun. Sollen andere Funktionen berechnet werden, ist die Taylorreihenentwicklung möglicherweise die beste Wahl. Die Implementierung erweist sich als kompliziert und sehr speziell.

Ein Ziel der Diplomarbeit stellt die Suche nach hardware-optimalen Komponenten dar, mit denen ein DDC realisiert werden kann. Da aus oben genannten Gründen nicht davon auszugehen ist, dass der Taylorreihen-Oszillator diese Bedingungen erfüllt, wird im Rahmen dieser Diplomarbeit von einer Umsetzung in VHDL abgesehen.

N	$\pi/2$	π	2π	N	$\pi/2$	π	2π
1	3	3	3	7	37	30	25
2	7	7	6	8	44	36	30
3	12	11	9	9	51	42	35
4	18	15	13	10	59	48	40
5	24	20	17	11	67	54	45
6	30	25	21	12	75	61	50

Tabelle 3.2.: Erreichbare Genauigkeit in Bit der Taylorreihe für Sinus in Abhängigkeit der Schrittanzahl N und der maximalen Frequenz

3.4.4. Vor- und Nachteile des Taylorreihen-Oszillators

Ein Oszillator mit Taylorreihenentwicklung weist folgende Vor- und Nachteile auf.

Vorteile:

- Der Berechnungsfehler hängt nur von dem verwendeten Wertebereich und der Iterationsanzahl ab, er pflanzt sich aber nicht fort
- Es kann beliebig hohe Genauigkeit erreicht werden
- Die Taylorreihenentwicklung ist auch für andere Funktionen geeignet
- Eine Pipelinestruktur kann implementiert werden, um taktweise Funktionswerte zu berechnen
- Die Frequenz des Ausgangssignals ist zur Laufzeit änderbar und abhängig von dem Eingang x

Nachteile:

- Die Hardwareimplementierung ist für jede Taylorreihe kompliziert
- Um eine Berechnung der Fakultäten zu implementieren, müssen die internen Register sehr groß sein
- Es muss neben Multiplikation auch Division implementiert werden
- Sollen hohe Datenraten erreicht werden, wird die Implementierung bzgl. Chipauslastung sehr groß

3.5. CORDIC-Oszillator

Der CORDIC-Algorithmus ist unter anderem in der Lage, verschiedene trigonometrische Funktionen mit relativ geringem Aufwand zu berechnen. Sinus und Cosinus können mit einem CORDIC gleichzeitig berechnet werden, wobei keine Fehlerfortpflanzung stattfindet und eine beliebige Genauigkeit erreichbar ist.

Der Begriff CORDIC steht für *Coordinate-Rotation-Digital-Computer* [UMB1].

Der CORDIC-Algorithmus berechnet ohne Multiplizierer eine Koordinatentransformation zwischen kartesischen (x, y) und polaren Koordinaten (R, Θ) basierend auf Vektorrotation. Es sind drei verschiedene Betriebsmodi zu unterscheiden: zirkulär, linear und hyperbolische, die durch m festgelegt werden. Weitere Informationen zu den Betriebsmodi befinden sich in Tabelle 3.3

Modus	Winkel Θ_k	Schiebesequenz	Radiusfaktor
zirkulär $m = 1$	$\tan^{-1} 2^{-k}$	$0, 1, 2, \ldots$	$K_1 = 1.65$
linear $m = 0$	2^{-k}	$1, 2, \ldots$	$K_0 = 1.0$
hyperbolisch $m = -1$	$\tanh^{-1} 2^{-k}$	$1, 2, 3, 4, 4 \ldots$	$K_{-1} = 0.8$

Tabelle 3.3.: CORDIC-Modi

Neben der Einstellmöglichkeit von m gibt es zwei grundsätzlich zu unterscheidende Funktionsweisen, den *Vektormodus* und den *Rotationsmodus*. Zur Berechnung der gewünschten Funktion wird entweder der Vektor (X_0, Y_0) so rotiert, dass Y sukzessiv kleiner wird, bis der Vektor schließlich auf der Abszisse liegt (*Vektormodus*) oder die Rotation so durchgeführt wird, dass das Winkelregister Z allmählich Null wird (*Rotationsmodus*).

Die Rotationswinkel werden so gewählt, dass sich die Multiplikationen auf einfache Links-Rechts-Schiebeoperationen (mit Potenzen von Zwei) reduzieren, wodurch keine Multiplizierer benötigt werden. Die Genauigkeit der Berechnungen eines CORDIC-Prozessors steigt mit der Iterationsanzahl und es findet keine Fehlerfortpflanzung statt. Allgemein wird der CORDIC-Algorithmus wie folgt definiert.

$$\begin{bmatrix} X_{k+1} \\ Y_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & -m\ \delta_k\ 2^{-k} \\ \delta_k\ 2^{-k} & 1 \end{bmatrix} \begin{bmatrix} X_k \\ Y_k \end{bmatrix} \tag{3.38}$$

$$Z_{k+1} = Z_k + \delta_k\ \Theta_k\ , \tag{3.39}$$

wobei der Winkel Θ_k sich aus Tabelle 3.3 ergibt, $\delta_k = \pm 1$ ist, und die Funktionsweisen $Y_k \to 0$ den Vektormodus und $Z_k \to 0$ den Rotationsmodus darstellen [UMB1].

Es sind also sechs verschiedene Berechnungen möglich, $X \cdot Y$, Y/X, $sin(Z)$, $cos(Z)$ $tan^{-1}(Z)$, $sinh(Z)$, $cosh(Z)$ und $tanh(Z)$. Durch Kombination verschiedener Funktionen sind somit weitere Funktionen berechenbar.

$$tan(Z) = \frac{sin(Z)}{cos(Z)} \qquad m = 1, 0 \tag{3.40}$$

$$tanh(Z) = \frac{sinh(Z)}{cosh(Z)} \qquad m = -1, 0 \tag{3.41}$$

$$exp(Z) = sinh(Z) + cosh(Z) \qquad m = -1; \quad X = Y = 1 \tag{3.42}$$

$$log_e(W) = 2\ tan^{-1}(Y/X) \qquad m = -1 \text{ mit } X = W + 1,\ Y = W - 1 \tag{3.43}$$

$$\sqrt{W} = \sqrt{X^2 - Y^2} \qquad m = 1 \text{ mit } X = W + \frac{1}{4},\ Y = W - \frac{1}{4} \tag{3.44}$$

Um sicherzustellen, dass der CORDIC-Algorithmus konvergiert, muss für den linearen und zirkulären Modus die Summe aller Rotationswinkel größer als der Startwinkel Z sein. Tabelle 3.4 beinhaltet eine Darstellung der internen Register aus denen sich die Ergebnisse ableiten lassen.

m	Rotationsmodus ($Z_k \to 0$)	Vektormodus ($Y_k \to 0$)
zirkulär 1	$X_k = K_1(X_0\, cos\, Z_0 - Y_0\, sin\, Z_0)$ $Y_k = K_1(X_0\, cos\, Z_0 + Y_0\, sin\, Z_0)$	$X_k = K_1\sqrt{X_0^2 + Y_0^2}$ $Z_k = Z_0 + arctan(Y_0/X_0)$
linear 0	$X_k = X_0$ $Y_k = Y_0 + X_0 \cdot Z_0$	$X_k = X_0$ $Z_k = Z_0 + Y_0/X_0$
hyperbolisch -1	$X_k = K_{-1}(X_0\, cosh\, Z_0 - Y_0\, sinh\, Z_0)$ $Y_k = K_{-1}(X_0\, cosh\, Z_0 + Y_0\, sinh\, Z_0)$	$X_k = K_{-1}\sqrt{X_0^2 + Y_0^2}$ $Z_k = Z_0 + tanh^{-1}(Y_0/X_0)$

Tabelle 3.4.: Ergebnisdarstellung in Abhängigkeit von m, Z_k und Y_k für den CORDIC-Algorithmus

3.5.1. Architekturen

Der CORDIC-Algorithmus ist rekursiv, d.h. es wird mehrfach dieselbe Operation nacheinander durchgeführt, wobei das Ergebnis mit jeder Iteration genauer wird. Für die Hardwareimplementierung eines rekursiven CORDICs bietet sich eine Statemachine (Bild 3.15) an, weil so der Hardwareaufwand gering bleibt.

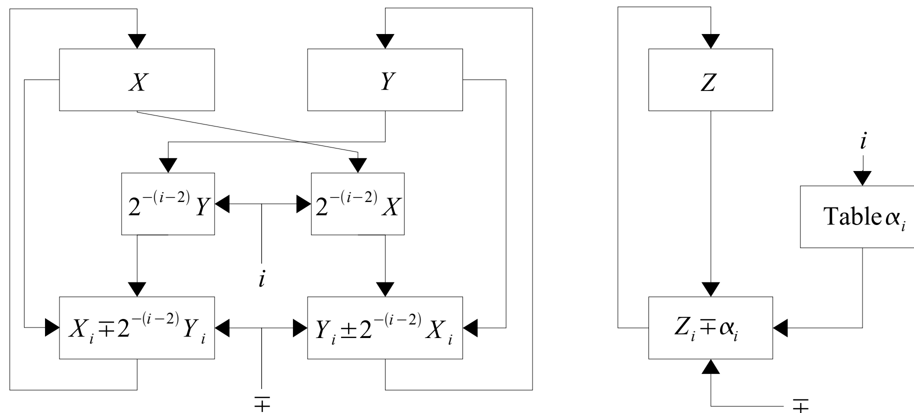


Bild 3.15.: CORDIC-Statemachine

Wenn pro Takt ein Ergebnis benötigt wird, muss der CORDIC in Pipeline-Struktur (Bild 3.16) implementiert werden. Hierbei wird dieselbe Operation vervielfacht und die Berechnung bewegt sich sukzessiv in Richtung Ausgang. Der Hardwareaufwand multipliziert sich dabei in etwa um die Anzahl der Iterationen verglichen mit der Realisierung mit einer Statemachine.

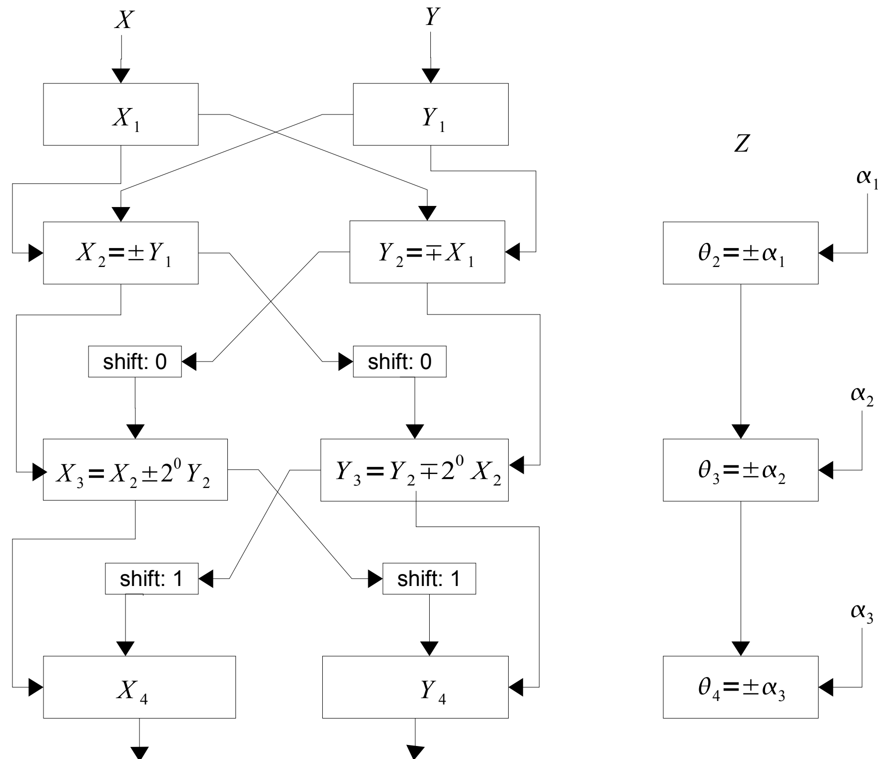


Bild 3.16.: CORDIC-Pipeline-Prozessor

3.5.2. CORDIC als Sinus-Cosinus-Oszillator

Um den CORDIC als Oszillator zu verwenden, wird der zirkuläre Modus mit Vektorrotation verwendet

$$X_k = K_1(X_0\ cos\ Z_0 - Y_0\ sin\ Z_0) \tag{3.45}$$

$$Y_k = K_1(X_0\ cos\ Z_0 + Y_0\ sin\ Z_0), \tag{3.46}$$

wobei $K_1 = 1,65$ ist. Sinus und Cosinus lassen sich aus

$$K_1\ cos(Z_0) = \frac{X_k + Y_k}{2} \tag{3.47}$$

$$K_1\ sin(Z_0) = \frac{X_k - Y_k}{2} \tag{3.48}$$

errechnen.

3.5.3. Implementierbarkeit

Der CORDIC ist ein gut implementierbares Rechenwerk mit mittlerem Hardwareaufwand für normale Anwendungen. Soll der CORDIC für High-End-Anwendungen implementiert werden, ergibt sich ein relativ großer Hardwareaufwand, weshalb im Rahmen der Diplomarbeit von einer VHDL-Implementierung abgesehen wird.

3.5.4. Vor- und Nachteile des CORDIC-Oszillators

Ein CORDIC-Oszillator weist folgende Vor- und Nachteile auf.

Vorteile:

- Die Berechnungsungenauigkeit pflanzt sich nicht fort und ist lediglich von der Iterationsanzahl abhängig
- Die Frequenz der Ausgangssignale ist zur Laufzeit änderbar
- Es können zur Laufzeit ggf. auch andere Funktionen berechnet werden

Nachteile:

- Sollen hohe Datenraten erzielt werden, muss der CORDIC in Pipelinestruktur implementiert werden, wodurch eine große Schaltung resultiert
- Der Skalierungsfaktor der Ausgangssignale sorgt für eine nicht optimale Ausnutzung weiterer Signalverarbeitung

3.6. Gegenüberstellung der Oszillatoren

Es wurden vier Möglichkeiten digitaler Oszillatoren vorgestellt, die sich unterschiedlich gut für die Verwendung in einem Digital-Down-Converter eignen. In Tabelle 3.5 werden die Bewertungen der Eigenschaften der vorgestellten Oszillatoren vergleichend dargestellt.

Osz.-Typ	Komplexität	Genauigkeit	Frequenzabhängigkeit	Implementierbarkeit
LUT-Osz.	⊕ ⊕ ⊕	⊕ ⊕ ⊕	⊕ ⊕ ⊕	⊕
Zähl. IIR.	⊕ ⊕ ⊕	⊖⊖	⊖ ⊖ ⊖	⊕
Überwacht. IIR.	⊕⊕	⊖	⊖ ⊖ ⊖	⊖
Adapt. IIR.	⊕	⊕	⊖	⊕⊕
Taylor-Osz.	⊖ ⊖ ⊖	⊕ ⊕ ⊕	⊕ ⊕ ⊕	⊖ ⊖ ⊖
CORDIC-Osz.	⊖⊖	⊕ ⊕ ⊕	⊕ ⊕ ⊕	⊖

Tabelle 3.5.: Bewertung der Oszillatoren

Der Lookuptable-Oszillator erreicht die höchste Genauigkeit der Oszillatoren und bietet ein großes Spektrum an Hardwareoptimierungen. Für die Quadraturamplitudenmodulation in einem DDC wird häufig eine Modulationsfrequenz zwischen $(0 \ldots \pi/2)$ benötigt, wodurch die Wahrscheinlichkeit groß ist, eine geringe Anzahl von Funktionswerte vorausberechnen zu müssen.

Der IIR-Oszillator kann mit relativ geringem Hardwareaufwand realisiert werden und bietet mit der Stabilisierungsmethode des Polverschiebens eine gute Genauigkeit. Allerdings eignet sich der IIR-Oszillator nicht für alle Frequenzen gleich gut, weshalb eine gründliche Prüfung dessen Einsatzmöglichkeit vorgenommen werden muss.

Für die VHDL-Implementierungen der verschiedenen DDCs werden der Lookuptable-Oszillator und der IIR-Oszillator verwendet, da mit anderen Ansätzen keine optimalen Ergebnisse bezüglich der Chipauslastung zu erwarten sind.

4. Finite Impulse Response Filter (FIR-Filter)

Ein FIR-Filter ist ein einfaches digitales Filter, das einen linearen Phasengang aufweisen kann. Im Gegensatz zu einem IIR-Filter (Infinite Impulse Response Filter) besitzt ein FIR-Filter keine Rückkopplung, woraus eine endliche Impulsantwort resultiert und es aufgrund der fehlenden Pole der Übertragungsfunktion stabil ist.

FIR-Filter werden benötigt, um die Filterung und Dezimation innerhalb des Digital-Down-Converters vorzunehmen.

Das zu entwickelnde Filter soll mit minimalem Aufwand, exakte Ergebnisse liefern. Dabei wird versucht, die Anzahl der benötigten Multiplizierer zu reduzieren und spezielle Eigenschaften der Koeffizienten auszunutzen. Es wird vorrausgesetzt, dass beliebige, ganzzahlige Dezimationsfaktoren mit dem Filter realisierbar sind.

4.1. Grundlagen

Die allgemeine Differenzengleichung eines FIR-Filters lautet

$$y(n) = \sum_{k=0}^{N-1} h(k)\, x(n-k). \tag{4.1}$$

Die z-Transformierte der Differenzengleichung ist

$$H(z) = \sum_{k=0}^{N-1} h(k)\, z^{-k}, \tag{4.2}$$

wobei $h(k)$ die Koeffizienten des FIR-Filters darstellen.

Ein linearer Phasengang wird erreicht, wenn die Koeffizienten symmetrisch sind, also wenn folgende Bedingung bezüglich der Koeffizienten erfüllt ist.

$$h(n) = \pm h(N-1-n), \quad n = 0, 1, \dots N-1 \tag{4.3}$$

Gleichung (4.1) führt zu der Realisierung in Direktform, die in Bild 4.1 abgebildet wird.

4.2. Transponierung und Hardwareoptimierung

Laut der Transponierungs-Theorie kann ein digitales System transponiert (umgestellt) werden, indem Ein- und Ausgang miteinander vertauscht werden und Knoten durch Addierer und umgekehrt ersetzt werden [DSP1]. Die Übertragungsfunktion ändert sich dabei nicht.

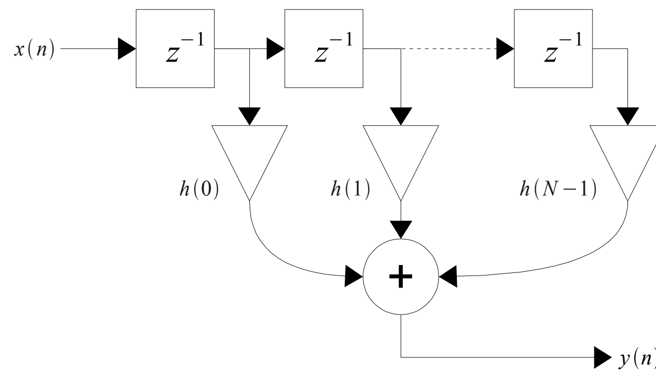


Bild 4.1.: FIR-Filter in Direktform

Wird ein FIR-Filter transponiert, ergeben sich zwei implementierbare Strukturen, die Direktform und die transponierte Form (Bild 4.2), die verschiedene Vor- und Nachteile haben.

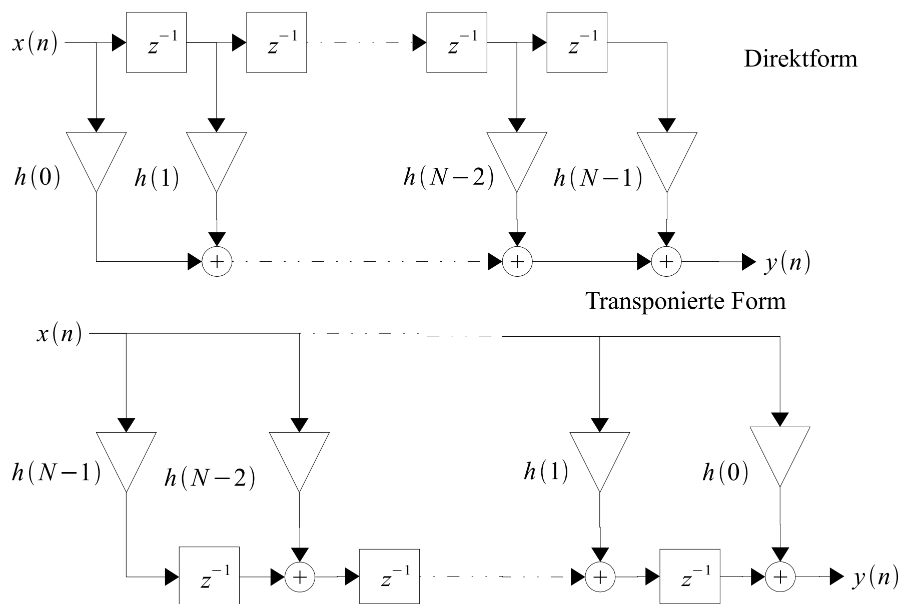


Bild 4.2.: Beispiel der Transponierung eines Filters

In der Direktform wird das Eingangssignal N-mal verzögert. Nach jeder Verzögerung findet eine Multiplikation mit einem Koeffizienten statt. Alle Produkte werden gleichzeitig addiert und ergeben einen Ausgangswert. Wegen der gleichzeitigen Addition aller Produkte ergibt sich eine komplizierte Schaltung für die Addition, die lange kombinatorische Pfade erzeugt. Allerdings ist es möglich, eine Dezimation vor der Multiplikation vorzunehmen, so dass Multiplikationen und Additionen mehr Takte pro Berechnung zur Verfügung haben.

Die transponierte Form speist die Multiplizierer mit dem aktuellen Eingangswert. Die Produkte werden anschließend verzögert und miteinander aufsummiert, so dass schnelle Additionen möglich sind. Die transponierte Form eignet sich somit für hohe Taktraten.

4.3. Übertragungsfunktion und Nullstellen

In diesem Abschnitt werden die Korrespondenz zwischen Koeffizienten eines FIR-Filters und dessen Übertragungsfunktion erläutert.

Hierzu ist zunächst ein kleiner Exkurs zur z-Transformation nötig [DSP1, SUS].

Ein LTI-System[1] kann durch Terme der z-Transformierten der Impulsantwort beschrieben werden, wobei $x(n)$, $y(n)$ und $h(n)$ die Eingangs- und Ausgangswerte bzw. die die Impulsantwort (oder Koeffizienten eines FIR-Filters) darstellen. $X(z)$, $Y(z)$ und $H(z)$ sind die jeweiligen z-Transformierten.

Ein digitales Filter „faltet“ das Eingangssignal mit der Impulsantwort, um das Ausgangssignal im Zeitbereich zu erzeugen. Eine Faltung im Zeitbereich entspricht einer Multiplikation im Frequenzbereich.

$$y(n) = x(n) * h(n) \tag{4.4}$$

$$Y(z) = X(z) \cdot H(z) \tag{4.5}$$

Die Übertragungsfunktion eines digitalen Systems lautet allgemein

$$H(z) = \frac{\sum_{r=0}^{N-1} b_r \; z^{-r}}{\sum_{k=0}^{M-1} a_k \; z^{-k}}. \tag{4.6}$$

Das entspricht einer gebrochen rationalen Funktion mit N Nullstellen und M Polen. Mit Hilfe des Fundamentalsatzes der Algebra lässt sich die Übertragungsfunktion auch folgendermaßen darstellen.

$$H(z) = A \; \frac{\prod_{i=1}^{N}(z - z_{0i})}{\prod_{j=1}^{M}(z - z_{\infty i})}, \tag{4.7}$$

wobei z_{0i} die Nullstellen und $z_{\infty i}$ die Polstellen der Übertragungsfunktion darstellen und A ein Skalierungsfaktor ist. Die Null- und Polstellen sind, bedingt durch die reellen Koeffizienten eines Systems, reell oder konjugiert-komplex.

Da ein FIR-Filter ein nicht-rekursives System ist und keine Pole besitzt, vereinfacht sich dessen Übertragungsfunktion.

$$H_{FIR}(z) = A \; \prod_{i=1}^{N}(z - z_{0i}) \tag{4.8}$$

$$H_{FIR}(e^{j\omega}) = A \; \prod_{i=1}^{N}(e^{j\omega} - z_{0i}) = H_{FIR}(z)|_{z=e^{j\omega}}. \tag{4.9}$$

$$|H_{FIR}(e^{j\omega})| = |A| \; \prod_{i=1}^{N}|e^{j\omega} - z_{0i}| \tag{4.10}$$

$$arg(H_{FIR}(e^{j\omega})) = arg(A) + \sum_{i=0}^{N-1} arg(e^{j\omega} - z_{0i}) \tag{4.11}$$

[1] LTI bedeutet Linear-Time-Invariant

Die Nullstellen der Übertragungsfunktion, Gl. (4.9), können in einem Pol-Nullstellen-Diagramm dargestellt werden, wie in Bild 4.3a dargestellt wird.

Der Amplitudengang kann aus dem Pol-Nullstellen-Diagramm ermittelt werden, indem alle Beträge der Nullstellen zu dem Punkt auf dem Einheitskreis einer Frequenz ($e^{j\omega}$) miteinander multipliziert (Bild 4.3b) werden (Gl. (4.10)).

Der Phasengang ergibt sich, indem alle Differenzwinkel der Nullstellen zu der Phase (Bild 4.3b) aufsummiert werden (Gl. (4.11)). Der Skalierungsfaktor A darf dabei nicht vergessen werden.

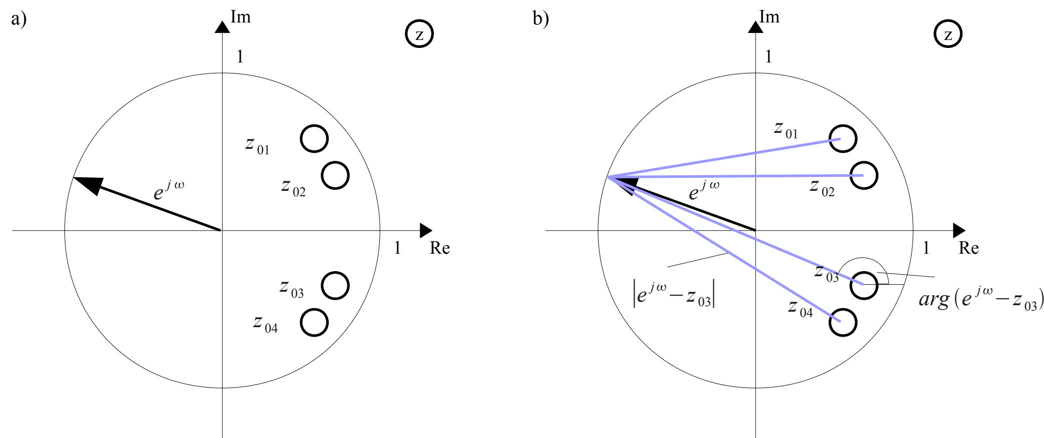


Bild 4.3.: Pol-Nullstellen-Diagramm in der z-Ebene

4.4. Koeffizientenberechnung

Die Berechnung von FIR-Filter-Koeffizienten ist kompliziert und wird in der Regel mit Programmen automatisch vorgenommen. Im Rahmen der Diplomarbeit wird von der Theorie der Koeffizientenberechnung abgesehen und auf die gängige Fachliteratur verwiesen, wie z.B. [PR96, SUS].

Die in dieser Arbeit verwendeten Filterkoeffzizienten werden mit *Scilab 5.0.3* berechnet. Scilab stellt verschiedene Funktionen bereit:

- **eqfir**: Minimax-Approximation, Multi-Band, lineare Phase
- **ffilt**: allgemeines FIR-Filter
- **fsfirlin**: Frequenzabtastmethode, lineare Phase
- **remezb**: Minimax-Approximation im Frequenzbereich
- **wfir**: Fenstermethode mit linearer Phase

Für weitere Informationen bezüglich der Verwendung der o.g. Funktionen, kann die Hilfefunktion von Scilab verwendet werden, indem in der Konsole *help* <*Funktion*> eingegeben wird.

4.5. Quantisierungseffekte

Wenn die Koeffizienten eines FIR-Filters quantisiert werden, verschieben sich dessen Nullstellen und der Amplituden- und Phasengang werden verfälscht. Bild 4.4 stellt das Nullstellendiagramm und den Amplitudengang für ideale und 8-Bit-quantisierte Koeffizienten eines FIR-Tiefpasses vergleichend dar. Es ist zu erkennen, dass einige Nullstellen erheblich von den ursprünglichen Nullstellen abweichen, wodurch sich der Amplitudengang dahingehend verfälscht, dass die Dämpfung abnimmt und das Passband größere Schwankungen aufweist. Dadurch, dass die Abweichung reeller Nullstellen von der Ursprungslage am größten ist, kann darauf geschlossen werden, dass reelle Nullstellen besonders stark von Quantisierungseffekten beeinflusst werden.

Es konnte in Simulationen festgestellt werden, dass Quantisierungseffekte mit steigender Koeffizientenanzahl zunehmen, so dass ein FIR-Filter hohen Grades mit höherer Genauigkeit rechnen muss als ein FIR-Filter mit weniger Koeffizienten. In [PR96] wird versucht, FIR-Filter aus Kaskaden zweiten Grades in Polyphasenstruktur zu realisieren, da die Quantisierungseffekte auf diese Weise minimiert werden.

Grundsätzlich sind Quantisierungseffekte in FIR-Filtern ein weites Gebiet, weshalb an dieser Stelle auf die Fachliteratur wie [PR96, SUS] verwiesen wird.

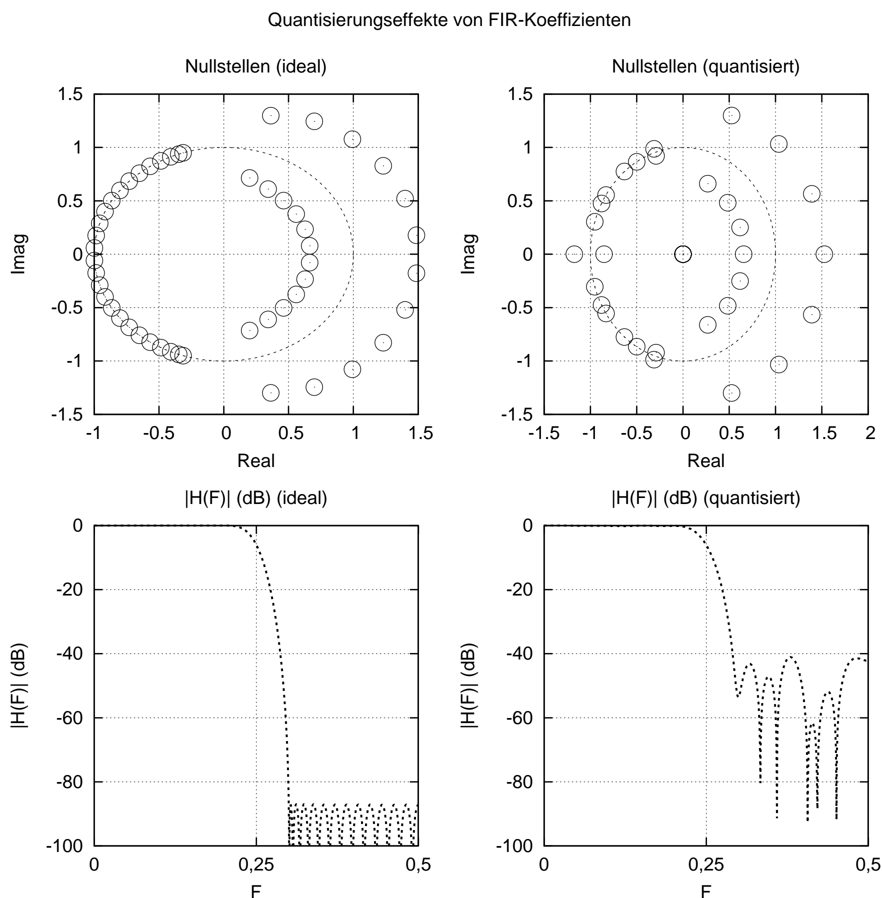


Bild 4.4.: Quantisierungseffekte eines Equiripple-FIR-Filters 51sten Grades für 8 Bit Auflösung

4.6. Vor- und Nachteile des FIR-Dezimationsfilters

Das vorgestellte universelle FIR-Dezimationsfilter weist die folgenden Vor- und Nachteile auf.

Vorteile:

- Ein FIR-Filter kann einen linearen Phasengang aufweisen
- Es ist einfach zu implementieren
- Für die Koeffizientenberechnung existieren viele Programme
- Durch Transponierung kann die Hardware für spezielle Anwendungen optimiert werden
- Ein FIR-Filter kann als Dezimationsfilter verwendet werden

Nachteile:

- Ein FIR-Filter hat gegenüber einem vergleichbaren IIR-Filter einen größeren Hardwareaufwand
- Mit steigender Koeffizientenanzahl wird höhere Genauigkeit aller Berechnungen benötigt, da die Dynamik der Register aufgrund der kleiner werdenden Koeffizienten weniger ausgenutzt wird

5. Kaskadiertes FIR-Halfband-Dezimationsfilter

Das zu entwickelnde Filter soll mit minimalem Aufwand exakte Ergebnisse liefern. Es wird ein Dezimationsfaktor von Zwei vorgesehen, so dass das Filter die speziellen Eigenschaften von Halfbandfilterkoeffizienten ausnutzen kann.

5.1. Grundlagen

FIR-Halfbandfilter besitzen einen symmetrischen Koeffizientensatz, bei denen - abgesehen von dem Mittenkoeffizienten - jeder zweite Koeffizient Null ist. Diese Eigenschaft führt direkt zu einer Hardwarereduzierung, weil Multiplikationen der fehlenden Koeffizienten nicht berücksichtigt werden müssen.

Wie der Name bereits vermuten lässt, ist ein Halfbandfilter ein Tiefpassfilter, das die obere Hälfte des Eingangsspektrums dämpft, so dass eine anschließende Dezimation um Faktor Zwei möglich wird. Das FIR-Filter in Direktform bietet, wie bereits erwähnt, die Möglichkeit, eine Dezimation in das Filter zu integrieren, so dass ein Multiplizierer für zwei Multiplikationen verwendet werden kann (Bild 5.1).

Wenn zusätzlich die Symmetrie der Koeffizienten ausgenutzt wird, werden nur

$$M = \frac{\lceil N/4 + 1 \rceil}{2} \tag{5.1}$$

Multiplizierer benötigt, wodurch die Anzahl der Multiplizierer gegenüber der Implementierung in Direktform wesentlich geringer ausfällt.

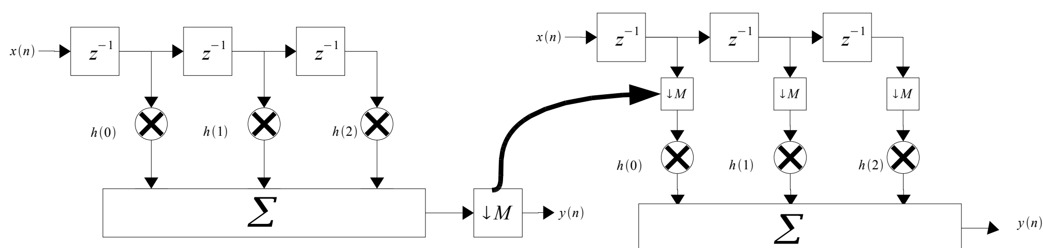


Bild 5.1.: Hardwarereduzierung eines FIR-Halfband-Filters

5.2. Kaskadierung mehrerer Filter

Durch eine Kaskadierung mehrerer FIR-Halfbandfilter lassen sich Dezimationsfaktoren zur Basis Zwei realisieren (Bild 5.2).

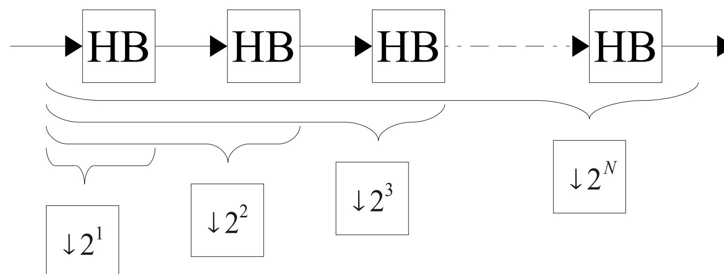


Bild 5.2.: Kaskade mehrerer FIR-Halfbandfilter

Innerhalb der Kaskade hat jede Stufe doppelt so viele Takte pro Sample zur Verfügung wie die vorherige Stufe, so dass in der Regel nach einigen Stufen eine serielle Multiplikation realisiert werden kann. Außerdem muss sich die Breite des Transitionsbands von Stufe zu Stufe halbieren, so dass die ersten Filter in der Kaskade, die eine hohe Datenrate verarbeiten, ein relativ breites Transitionsband und eine geringe Koeffizientenanzahl besitzen können, so dass sich ebenfalls eine optimale Nutzung der Hardware ergibt.

Beispiel

Um eine Dezimation um einen Faktor Acht zu erreichen, wird eine Kaskade aus 3 FIR-Halfbandfiltern verwendet. Das Ausgangssignal darf maximal 90% der resultierenden Bandbreite verwenden, damit die Filterordnungen klein bleiben. Die Dämpfung in jeder Stufe soll mindestens 60 dB betragen.

Das Transitionsband des dritten Filters ist das steilste (20%) und muss zwischen 0,225 und 0,275 liegen (jeweils normierte Frequenzen). Das zweite Filter darf ein doppelt so breites Transitionsband

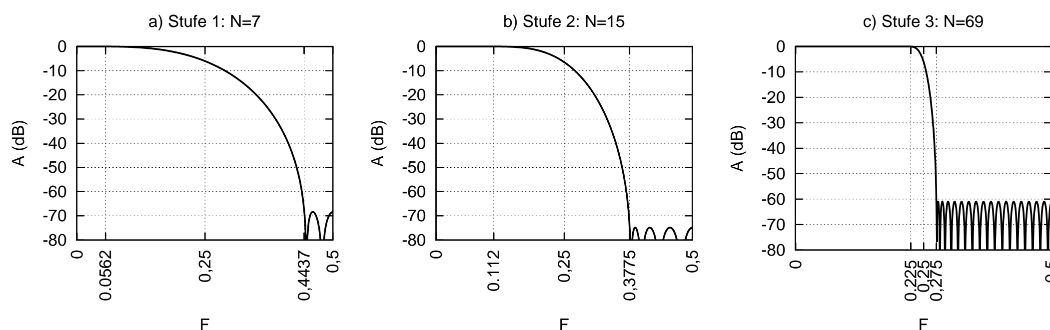


Bild 5.3.: Amplitudengänge einer FIR-Halfband-Kaskade mit drei Stufen

(40%) aufweisen, das zwischen 0,1125 und 0,3775 liegt. Die Breite des Transitionsbands verdoppelt sich erneut für das erste Filter (80%), das somit zwischen 0,05625 und 0,44375 liegt. Die Amplitudengänge der Kaskade werden in Bild 5.3 dargestellt.

Ein vergleichbares einstufiges Filter, das eine Dezimation um Faktor Acht durchführt und dessen Transitionsband zwischen 0,05625 und 0,0625 liegt, benötigt etwa 550 Koeffizienten, die jeweils in der hohen Datenrate arbeiten, um eine Ausnutzung von maximal 90% des resultierenden Ausgangssignals und eine Dämpfung von mindestens 60 dB zu erreichen.

5.3. Vor- und Nachteile des FIR-Halfbandfilters

Das FIR-Halfbandfilter weist die folgenden Vor- und Nachteile auf.

Vorteile:

- Ein FIR-Halfbandfilter bietet großes Optimierungspotential durch Ausnutzung fehlender und symmetrischer Koeffizienten
- Halfbandfilter sind kaskadierbar
- Aufgrund der Implementierung in Direktform kann die Dezimation bereits vor der Multiplikation stattfinden, so dass etwa die Hälfte der Koeffizienten gegenüber der transponierten Form eingespart werden kann
- Die Koeffizienten sind immer symmetrisch

Nachteile:

- Die Summation der Produkte ist komplizierter als in der Implementierung in transponierter Form, da Verzögerung und Addition voneinander getrennt sind

5.4. Gegenüberstellung der FIR-Filter

Es wurden zwei verschiedene Realisierungen für FIR-Dezimationsfilter theoretisch untersucht, ein Dezimationsfilter für beliebige, ganzzahlige Dezimationsfaktoren und ein Halfbandfilter, das kaskadiert Dezimationsfaktoren mit Potenz von Zwei realisieren kann.

Das Dezimationsfilter wird in transponierter Form realisiert, wodurch eine schnelle Schaltung mit wenig Registern für die Addition resultiert. Dahingegen wird das Halfbandfilter in Direktform realisiert, so dass die Dezimation bereits vor der Multiplikation geschieht. Dies wirkt sich negativ auf die Addition aus, die mit einem Addierbaum realisiert wird, um eine hohe Taktrate zu erzielen.

In Tabelle 5.1 werden die Eigenschaften der FIR-Filter bewertet und vergleichend dargestellt.

Filtertyp	Komplexität	Koeffizientenanzahl	Multipliziereranzahl
FIR-Dezimationsfilter	⊕	⊕	⊕
Halfbandfilter	⊕⊕	⊕⊕	⊕ ⊕ ⊕

Tabelle 5.1.: Bewertung der FIR-Filter

6. Cascaded Integrator Comb (CIC-Filter)

CIC-Filter sind Dezimations- und Interpolationsfilter, die aus einer Kaskade von Integratoren und Differentiatoren sowie einem Dezimator bzw. Interpolator bestehen. Durch den Einsatz von CIC-Filtern können Dezimation und Interpolation ohne die Verwendung von Multiplizierern realisiert werden, weshalb sich CIC-Filter für hochintegrierte Hardwaredesigns bestens eignen.

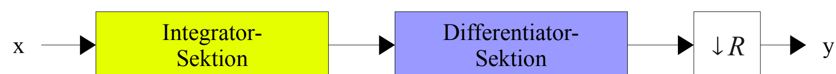
Dieses Kapitel basiert auf den Literaturangaben [CIC, MPD1, MB1, MB2].

Der CIC-Dezimator soll ohne Multiplizierer eine Dezimation um einen hohen Dezimationsfaktor durchführen können. Dabei soll eine möglichst ideale Übertragungscharakteristik erreicht werden und der benötigte Hardwareaufwand minimiert werden.

6.1. Herleitung der Übertragungsfunktion eines CIC-Filters

CIC-Filter besitzen eine einfache Struktur, die aus einer Integratorsektion, einer Differentiatorensektion und einem Dezimator bzw. Interpolator besteht. Die Funktionsweise des CIC-Filters wird durch die Reihenfolge der verschiedenen Sektionen bestimmt. Wird die Integratorsektion von der Differentiatorensektion gefolgt, entsteht ein CIC-Dezimator (6.1a). Anderenfalls entsteht ein CIC-Interpolator (6.1b). Die Dezimation erfolgt nach der Differentiation und die Interpolation erfolgt vor der Differentiation.

a) CIC-Dezimator



b) CIC-Interpolator

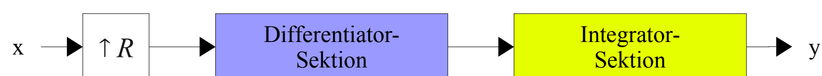


Bild 6.1.: Struktur von CIC-Dezimator und -Interpolator

Der Dezimations- bzw. Interpolationsfaktor R stellt einen der drei Parameter eines CIC-Filters dar. Die Sektionen, die in Bild 6.1 verwendet werden, bestehen jeweils aus gleich vielen Integratoren bzw. Differentiatoren, deren Anzahl durch den zweiten Parameter N festgelegt wird. Bild 6.2 veranschaulicht den Aufbau der CIC-Sektionen.

Die Schaltbilder und die jeweilige Übertragungsfunktion von Integratoren und Differentiatoren werden in Bild 6.3 dargestellt, wobei besonderes Augenmerk auf die Anzahl der Verzögerungsglieder

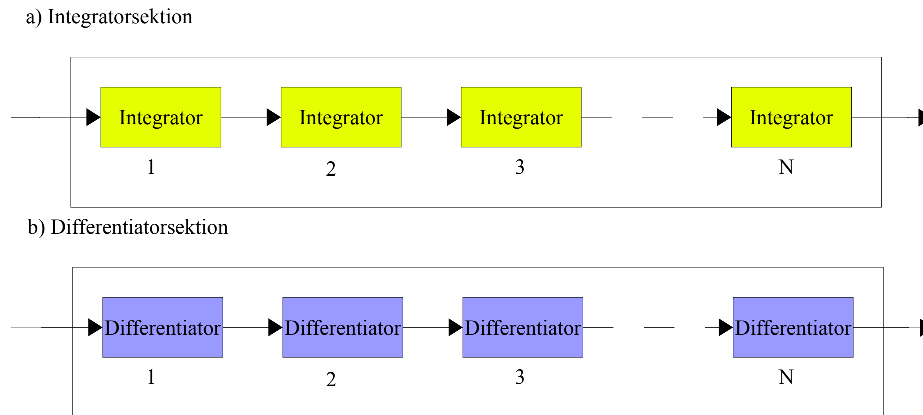


Bild 6.2.: Struktur der Differentiatoren- bzw. Integratorsektion eines CIC-Filters

(RM) eines Differentiators gelegt werden muss. M stellt den dritten Parameter von CIC-Filtern dar.

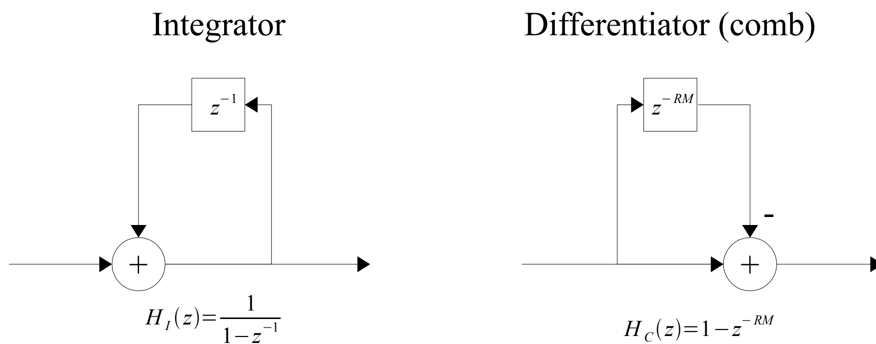


Bild 6.3.: Bausteine von Cascaded-Integrator-Combs

Nun können die Übertragungsfunktionen von den beiden Sektionen ermittelt werden.

$$H_{Intsec} = H_I^N = \left(\frac{1}{1 - z^{-1}}\right)^N \tag{6.1}$$

$$H_{Combsec} = H_C^N = \left(1 - z^{-RM}\right)^N \tag{6.2}$$

Die gesamte Übertragungsfunktion eines CIC-Filters ergibt sich aus der Multiplikation beider Übertragungsfunktionen der Sektionen und ist für CIC-Dezimator und -Interpolatoren identisch.

$$H(z) = H_I^N(z)\ H_C^N(z) = \frac{(1 - z^{-RM})^N}{(1 - z^{-1})^N} = \left\{\sum_{k=0}^{RM-1} z^{-k}\right\}^N \tag{6.3}$$

Ein CIC-Filter basiert auf Pol-Nullstellen-Kompensation, wodurch dessen Übertragungsfunktion der eines FIR-Filters entspricht, das einen linearen Phasenverlauf aufweist. Aufgrund dieser Eigenschaft eröffnet sich ein weites Anwendungsspektrum für CIC-Filter in der digitalen Signalverarbeitung.

Es muss bemerkt werden, dass jede Integratorstufe eine Rückkopplung mit dem Faktor 1 besitzt, was Registerüberläufe zur Folge hat. Diese Tatsache hat keine Konsequenzen, wenn das CIC-Filter in Zweierkomplement-Arithmetik implementiert ist und wenn die Register so ausgelegt werden, dass deren Wertebereich größer oder gleich dem erwarteten maximalen Wert am Ende der Integratorstufen darstellt (siehe Abschnitt 6.3 Register-Dimensionierung).

6.2. Frequenzcharakteristik

Mit der im Anhang F.1 angegebenen Herleitung ergibt sich für den Amplitudengang eines CIC-Filters der Zusammenhang

$$|H(f)| = \left| \frac{sin \pi M f}{sin \frac{\pi f}{R}} \right|^N . \tag{6.4}$$

Für große Dezimationsraten kann der Amplitudengang für einen beschränkten Frequenzbereich angenähert werden.

$$|H(f)| = \left| RM \frac{sin \pi M f}{\pi M f} \right|^N \qquad \text{für} \quad 0 \leq f \leq \frac{1}{M} \tag{6.5}$$

Ein Beispiel eines Amplitudengangs eines 4-Stufen-CIC-Filters ($N = 4$) befindet sich in Bild 6.4. Die

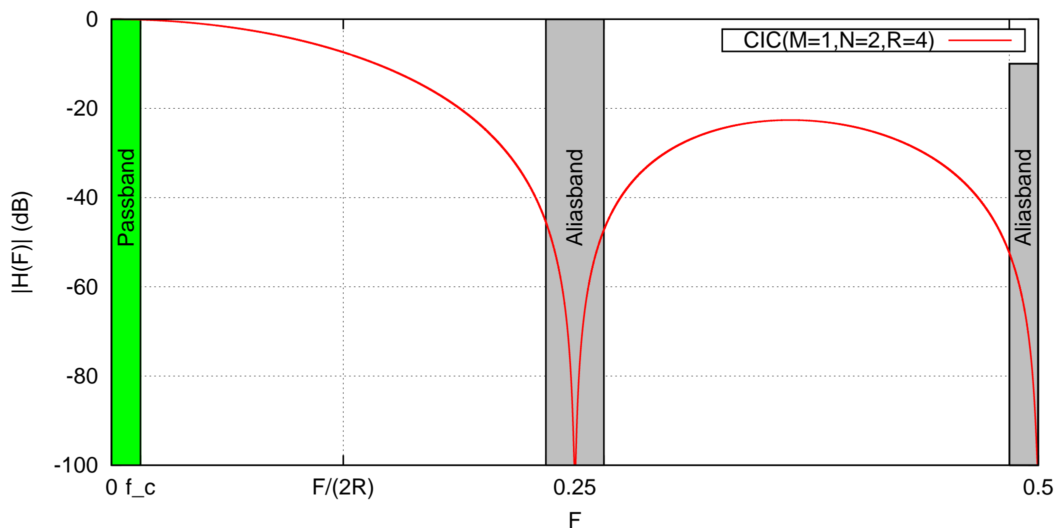


Bild 6.4.: Beispielhafter CIC-Amplitudengang für ein zweistufiges CIC-Filter mit $M = 1$ und $R = 4$

Verzögerung in den Differentiations-Stufen ist $M = 1$ und der Dezimationsfaktor $R = 7$.

Die Übertragungsfunktionen (6.4) und (6.5) haben Nullstellen bei Vielfachen von $f = 1/M$. Daher kann die Anzahl der Verzögerungen in der Differentiatorenstufe dazu genutzt werden, Nulldurchgänge

in der Übertragungsfunktion zu realisieren. Da das Transitionsband relativ breit und flach ist, wird ein CIC-Filter in der Regel von einem weiteren FIR-Filter oder mehreren FIR-Halfband-Filtern gefolgt, um eine Schärfung des Transitionsbands und ggf. gleichzeitiger weiterer Dezimation vorzunehmen.

Da CIC-Interpolatoren für den Einsatz in einem DDC keine Verwendung finden, wird im Folgenden nur der CIC-Dezimator behandelt.

Wenn M gleich 1 ist, wird die Dezimation eines CIC-Dezimators so durchgeführt, dass die resultierende Abtastfrequenz an der Stelle der ersten Nullstelle liegt. Aliasing-Effekte treten dabei auf, weil alle Bereiche um die M-ten Nullstellen in das Passband hineingefaltet werden. Aliasing-Effekte besitzen die doppelte Breite der Passbandbreite und treten an den Frequenzen

$$\frac{i}{RM} - f_c \leq f_{Ai} \leq \frac{i}{RM} + f_c \tag{6.6}$$

auf.

Um die Aliasing- bzw. Imaging-Fehler zu charakterisieren, wird in der Praxis lediglich der größte Anteil dieses Fehlers f_{A1} betrachtet (Bild 6.4).

Durch den Parameter M werden Nullstellen in die Übertragungsfunktion im Bereich von 0 bis $1/R$ gelegt, so dass das Spektrum des Ausgangssignals $M/2$ Nullstellen aufweist.

6.3. Registerdimensionierung

Die Register eines CIC-Dezimators müssen so dimensioniert werden, dass die durch die Integratoren hervorgerufene Verstärkung der Eingangssinale toleriert wird.

Die maximale Verstärkung des CIC-Filters ist

$$G_{max} = (RM)^N. \tag{6.7}$$

Somit müssen die Register des CIC-Dezimators B_{max} Bit breit sein.

$$B_{max} = \lceil N\ log_2 RM + B_{in} \rceil \text{ Bit} \tag{6.8}$$

Eine ausführliche Herleitung befindet sich im Anhang F.2 auf Seite 180.

Beispiel

Die Register eines CIC-Filters für R=12, M=2 und N=4 müssen für 16 Bit breite Eingangsdaten

$$B_{max} = \lceil 4\ log_2(12 \cdot 2) + 16 \rceil \text{ Bit} = 35 \text{ Bit} \tag{6.9}$$

breit sein.

6.4. Hardwareoptimierung

Durch Umstellen des Differentiators bzw. des Interpolators kann der Hardwareaufwand eines CIC-Filters erheblich reduziert werden. Indem der Dezimator vor die Differnentiatorsektion (Bild 6.5a)

bzw. der Interpolator hinter die Differentiatorensektion (Bild 6.5b) verlegt wird, verringert sich die Anzahl der Verzögerungsglieder in den Differentiatoren jeweils um den Faktor R. Die Differentiatorensektion muss anschließend mit dem Takt der niedrigen Taktdomäne betrieben werden.

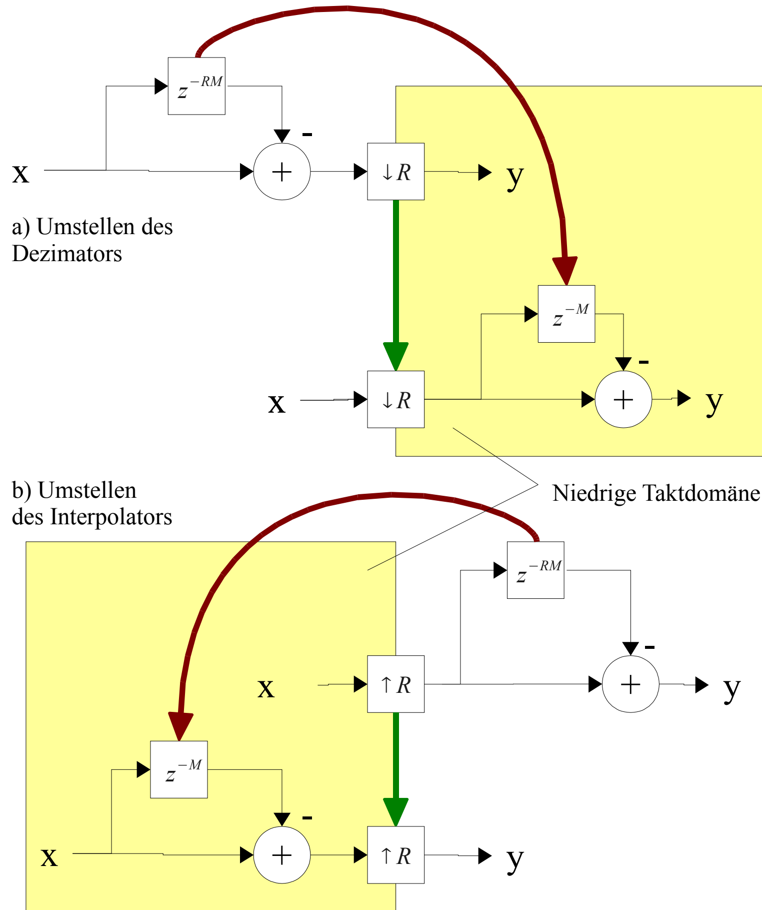


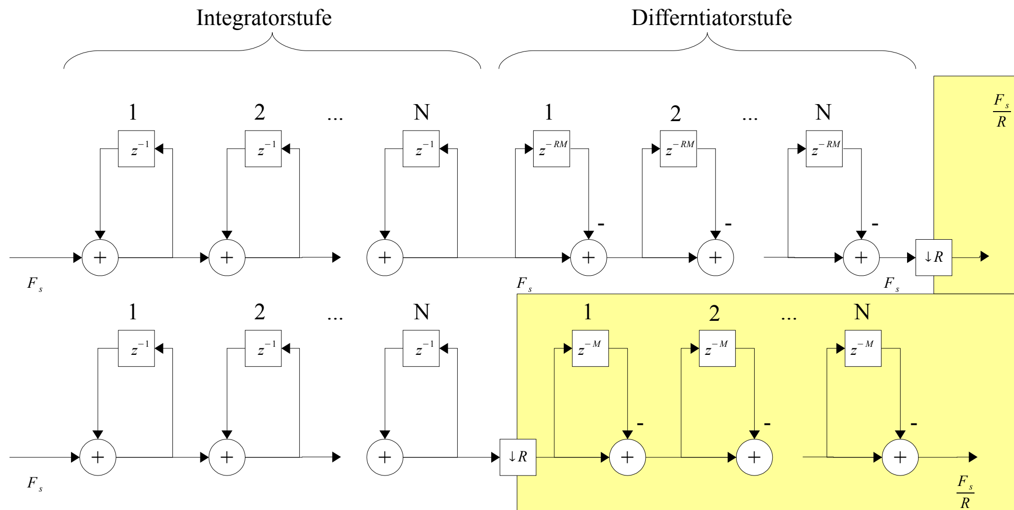
Bild 6.5.: Umstellung von Dezimator und Interpolator eines CIC-Filters zwecks Hardwareoptimierung

In Bild 6.5 wird der Vorgang schematisch erläutert. Gelb eingerahmte Teile der Schaltung befinden sich in der niedrigen Taktdomäne. In Bild 6.6 befindet sich ein weiteres Beispiel für einen CIC-Dezimator mit und ohne Optimierung der Struktur und einem optimierten CIC-Interpolator.

6.5. Runden und Abschneiden

Es ist möglich, den Hardwareaufwand eines CIC-Dezimators weiter zu verringern, indem die Register in Richtung des Signalflusses verkleinert werden (Bild 6.7). Wenn der resultierende Fehler innerhalb tolerierbarer Grenzen ist. Die schraffierten Bereiche in Bild 6.7 stellen den einsparbaren Bereich dar. Im Folgenden wird eine mathematische Analyse der Effekte von Runden und Abschneiden von Bits innerhalb eines CIC-Dezimators vorgestellt, die aus [CIC] entnommen wurde.

a) CIC-Dezimator



b) CIC-Interpolator

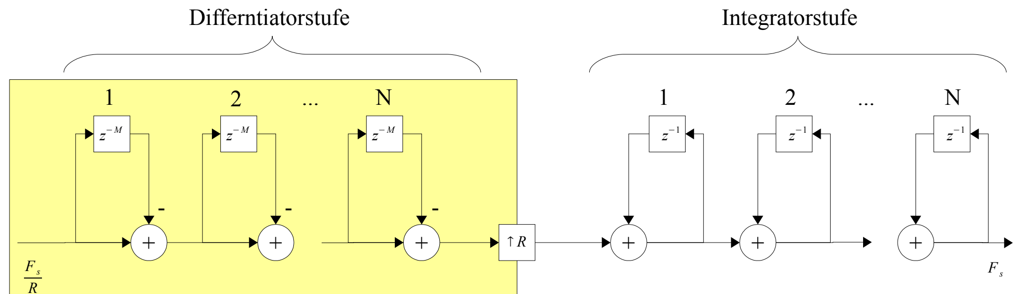


Bild 6.6.: Aufbau von CIC-Dezimatoren und -Interpolatoren

Um den entstehenden Gesamtfehler für Runden bzw. Abschneiden zu berechnen, müssen die Mittelwerte und die Standardabweichungen jeder einzelnen Fehlerquellen bezüglich des Ausgangs ermittelt und anschließend aufsummiert werden.

Es gibt $2N + 1$ Fehlerquellen. $2N$ Fehlerquellen für N Integratoren bzw. Differentiatoren und eine weitere Fehlerquelle am Ausgang.

In der Regel wird angenommen, dass Runden sich günstiger auf einen Rechenfehler auswirkt als Abschneiden von Bits. Bei einem CIC-Dezimator ist der Fehler, abgesehen von der ersten und letzten Fehlerquelle, gleichgroß, unabhängig davon, ob gerundet oder abgeschnitten wird. Es wird angenommen, dass jede Fehlerquelle weißes Rauschen verursacht, das nicht von den Eingangssignalen abhängt.

Die Eintrittswahrscheinlichkeit des Fehlers an der j-ten Stufe ist

$$E_j = 2^{B_j}, \tag{6.10}$$

wobei B_j die Anzahl der abgeschnittenen bzw. für die Rundung betrachteten LSBs darstellt.

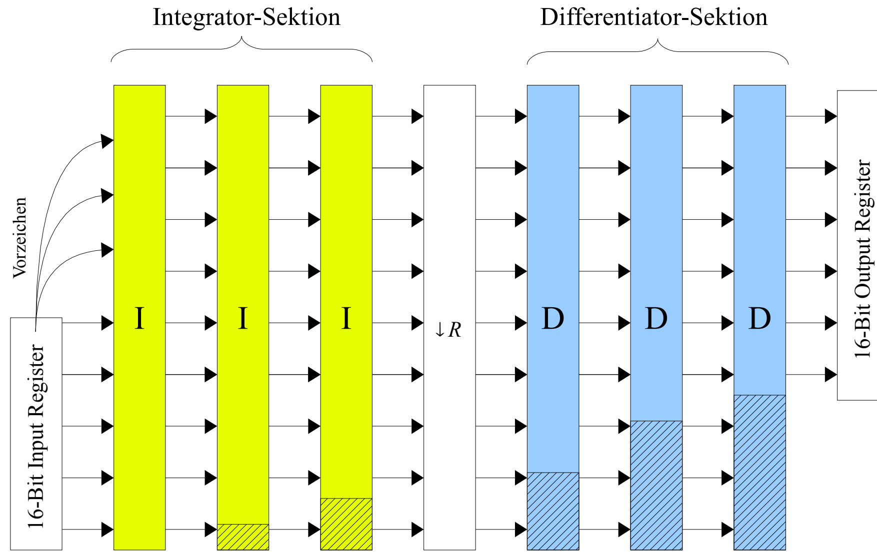


Bild 6.7.: Schematische Darstellung der Verringerung der Register in einem CIC-Dezimator

Der statistische Mittelwert des Fehlers ist

$$\mu_j = \begin{cases} \frac{1}{2}E_j, & \text{für Abschneiden} \\ 0, & \text{sonst} \end{cases} \tag{6.11}$$

und die Standardabweichung des Fehlers ergibt sich aus

$$\sigma_j^2 = \frac{1}{12}E_j^2. \tag{6.12}$$

Es muss der statistische Fehler, der von der j-ten Fehlerquelle herrührt, bestimmt werden. Der Mittelwert und die Standardabweichung, erzeugt von der k-ten Quellen, sind $\mu_j h_j(k)$ und $\sigma_j^2 h_j^2(k)$. Da die entstehenden Fehler unabhängig von k sind, bilden sich der statistische Fehler an der j-ten Stufe aus der Summe aller einzelner Fehler der Impulsantwortkoeffizienten.

Daher ist der Mittelwert des Fehlers insgesamt

$$\mu_{T_j} = \mu_j D_j, \tag{6.13}$$

und

$$D_j = \begin{cases} \sum_k h_j(k), & j = 1, 2, \dots, 2N \\ 1, & j = 2N + 1 \end{cases}. \tag{6.14}$$

Die Standardabweichung wird ähnlich gebildet,

$$\sigma_{T_j}^2 = \sigma_j^2 F_j^2 \tag{6.15}$$

und

$$F_j^2 = \begin{cases} \sum_k h_j^2(k), & j = 1, 2, \ldots, 2N \\ 1, & j = 2N+1 \end{cases} . \tag{6.16}$$

Beide erhaltenen Fehler werden verwendet, um den Einfluss der Fehlerquelle auf den Ausgang des Filters zu berechnen. Es kann gezeigt werden, dass D_j gleich 0 für alle außer der ersten und letzten Fehlerquelle ist [CIC]. Daher vereinfacht sich der Ausdruck zu

$$D_j = \begin{cases} (RM)^N, & j = 1 \\ 0, & j = 2, 3, \ldots, 2N \\ 1, & j = 2N+1 \end{cases} . \tag{6.17}$$

Da die Standardabweichung von einer Fehlerquelle für Runden oder Abschneiden gleich ist, und nur die erste und letzte Fehlerquelle den Ausgang des Filters beeinflussen, wirkt sich die Wahl zwischen Runden und Abschneiden, abgesehen von der ersten und letzten Fehlerquelle, nicht auf den Ausgang aus.

Der Mittelwert des Fehlers und die Standardabweichung sind

$$\mu_T = \sum_{j=1}^{2N+1} \mu_{T_j} = +\mu_{T_1} + \mu_{T_{2N+1}} \tag{6.18}$$

und

$$\sigma_T^2 = \sum_{j=1}^{2N+1} \sigma_{T_j}^2 . \tag{6.19}$$

Anschließend kann bestimmt werden, wie viele LSBs pro Stufe gestrichen werden können, um den Fehler in akzeptablen Grenzen zu halten.

Es wird angenommen, dass die Anzahl der Bits am Ausgang B_{out} ist. Daher ist die Anzahl der Bits, die gestrichen werden können

$$B_{2N+1} = B_{max} - B_{out} + 1 \text{ Bit}. \tag{6.20}$$

Die resultierende Fehlervarianz $\sigma_{T_{2N+1}}^2$ errechnet sich aus den Gleichungen (6.15) und (6.16).

Es ist sinnvoll, die Fehlervarianz der ersten $2N$ Fehlerquellen kleiner oder gleich der letzten Fehlerquelle (Ausgang) zu machen.

Mit der folgenden Gleichung kann somit bestimmt werden, wieviele Bits pro Stufe weggestrichen werden dürfen.

$$B_j = \left\lfloor -log_2\, F_j + log_2\, \sigma_{T_{2N+1}} + \frac{1}{2}\, log_2\, \frac{6}{N} \right\rfloor \text{ Bit}, \tag{6.21}$$

und $j = 1, 2, \ldots, 2N$.

6.6. Entwurfsbeispiel eines CIC-Dezimators

Es soll ein Dezimationsfilter entworfen werden, um die Abtastrate von 6 MHz auf 240 kHz zu reduzieren, mit einem Passband von 30 kHz ($R = 25$ und f_c relativ zur niedrigen Abtastfrequenz ist

rel. Bandbreite f_c mult. M	Passbanddroop bei f_c in Abhängigkeit von N					
	N=1	2	3	4	5	6
1/256	3.76E-4	7.53E-4	1.13E-3	1.51E-3	1.88E-3	2.26E-3
1/128	1.51E-3	3.01E-3	4.52E-3	6.02E-3	7.53E-3	9.04E-3
1/64	6.02E-3	1.20E-2	1.81E-2	2.41E-2	3.01E-2	3.61E-2
1/32	2.41E-2	4.82E-2	7.23E-2	9.64E-2	1.21E-1	1.45E-1
1/16	9.65E-2	1.93E-1	2.90E-1	3.86E-1	4.83E-1	5.79E-1
1/8	3.88E-1	7.76E-1	1.16E+0	1.55E+0	1.94E+0	2.33E+0
1/4	1.58E+0	3.17E+0	4.75E+0	6.33E+0	7.92E+0	9.50E+0

Tabelle 6.1.: Passbanddroop eines CIC-Dezimators in Abhängigkeit von N

M	rel. Bandbreite f_c	Aliasrejection in dB bei f_{AI} in Abhängigkeit von N					
		N=1	2	3	4	5	6
1	1/256	1.02E+2	2.04E+2	3.06E+2	4.07E+2	5.09E+2	6.11E+2
1	1/128	8.80E+1	1.76E+2	2.64E+2	3.52E+2	4.40E+2	5.28E+2
1	1/64	7.41E+1	1.48E+2	2.22E+2	2.97E+2	3.71E+2	4.45E+2
1	1/32	6.03E+1	1.21E+2	1.81E+2	2.41E+2	3.01E+2	3.62E+2
1	1/16	4.65E+1	9.29E+1	1.39E+2	1.86E+2	2.32E+2	2.79E+2
1	1/8	3.27E+1	6.54E+1	9.81E+1	1.31E+2	1.63E+2	1.96E+2
1	1/4	1.92E+1	3.84E+1	5.76E+1	7.68E+1	9.61E+1	1.15E+2
2	1/256	9.48E+1	1.90E+2	2.84E+2	3.79E+2	4.74E+2	5.69E+2
2	1/128	8.08E+1	1.62E+2	2.43E+2	3.23E+2	4.04E+2	4.85E+2
2	1/64	6.67E+1	1.33E+2	2.00E+2	2.67E+2	3.34E+2	4.00E+2
2	1/32	5.24E+1	1.05E+2	1.57E+2	2.10E+2	2.62E+2	3.15E+2
2	1/16	3.78E+1	7.57E+1	1.14E+2	1.51E+2	1.89E+2	2.27E+2
2	1/8	2.29E+1	4.58E+1	6.87E+1	9.16E+1	1.15E+2	1.37E+2
2	1/4	8.51E+0	1.70E+1	2.55E+1	3.41E+1	4.26E+1	5.11E+1

Tabelle 6.2.: Aliasrejection eines CIC-Dezimators in Abhängigkeit von N

$1/8$). Alle Alias-Bänder sollen um mehr als $100\ dB$ gedämpft werden, und das Signal innerhalb des Passbands um maximal $1,5\ dB$ gedämpft werden.

Eingangssignale sollen mit einer Auflösung von $16\ Bit$ eingespeist werden: $B_{in} = B_{out} = 16$.

Aus den Tabellen 6.1 und 6.2 lässt sich entnehmen, dass ein CIC-Filter mit $N = 4$ Stufen in jeder Sektion und einer Verzögerung von $M = 1$ eine Aliasing-Dämpfung von $131\ dB$ und eine Dämpfung im Passband von $1,3\ dB$ liefert.

Um den Entwurf zu vereinfachen, werden Bits abgeschnitten. Die Größe der Register für den CIC-Dezimator ergeben sich zu

$$B_{max} = \lceil 4log_2(25) + 16 \rceil \text{ Bit} = 35 \text{ Bit}, \tag{6.22}$$

woraus

$$B_{2N+1} = B_{max} - B_{out} + 1 \text{ Bit} = 19 \text{ Bit} \tag{6.23}$$

resultiert.

Aus (6.21) werden die abschneidbaren Bits berechnet, die sich zu $\{1, 6, 9, 13, 14, 15, 16, 16\}$ ergeben.

Bild 6.8 enthält ein Blockschaltbild für ein Implementierungsbeispiel eines CIC-Dezimators mit M = 1, N = 4 und R = 25.

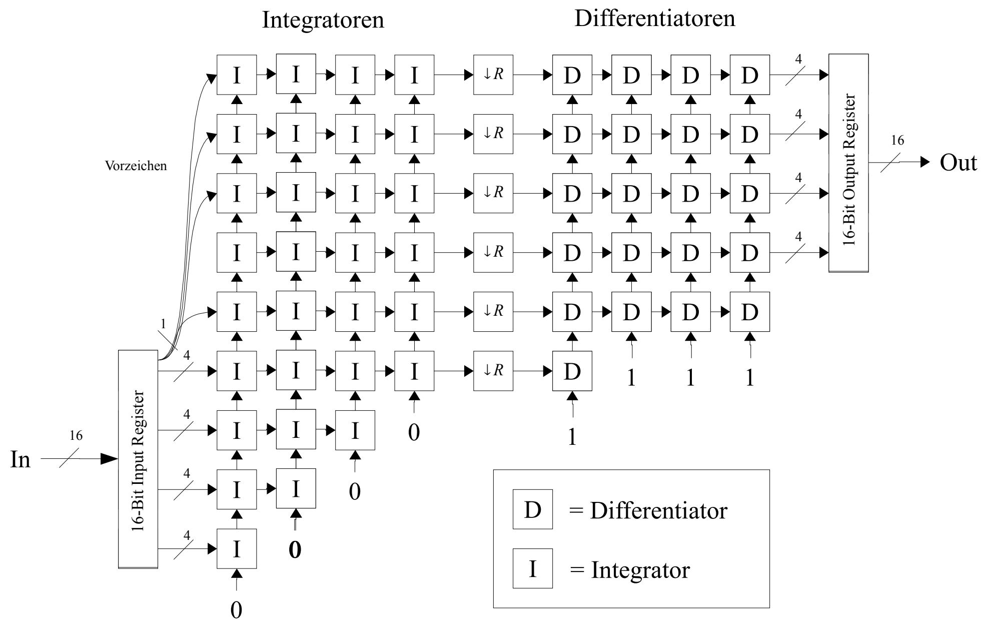


Bild 6.8.: Blockschaltbild eines optimierten CIC-Dezimators mit $M = 1$, $N = 4$ und $R = 25$ nach [CIC]

6.7. Vor- und Nachteile des CIC-Dezimators

CIC-Filter gelten als besonders ökonomisch bezüglich Hardware- und Kostenaufwand. Dies wird maßgeblich durch sechs Eigenschaften bestimmt.

Vorteile:

- Ein CIC-Dezimator benötigt keine Multiplizierer
- Es wird kein Speicherplatz für Filterkoeffizienten benötigt, da sämtliche Koeffzienten 1 sind
- Dadurch, dass die Differentiation in einer niedrigeren Frequenz stattfindet, verringert sich der benötigte Speicherplatz für die Verzögerungsglieder um den Faktor R
- Ein CIC-Filter wird aus zwei sich wiederholenden Blöcken erstellt, wodurch sich die Implementierung vereinfacht
- Es wird relativ wenig externe Logik benötigt, um die Dezimation zu realisieren, z.B. ein One-Hot-Zähler
- Ein Filter-Entwurf kann für eine Vielzahl verschiedener Dezimations-Faktoren verwendet werden, da lediglich das Zeitverhalten der Differentiationsstufen angepasst werden muss

Nachteile:

- Sowohl die Registerbreiten als auch das Register eines One-Hot-Zählers werden für große Dezimationsfaktoren sehr groß
- Das Übertragungsverhalten eines CIC-Dezimators wird nur durch drei Parameter (R,M und N) bestimmt, wodurch der Freiheitsgrad gering ist
- Eine optimale Nutzung der Hardware ergibt sich nur für Dezimationsfaktoren, die eine Potenz von Zwei sind [MPD1]

7. Sharpened Cascaded Integrator Comb (SCIC-Dezimator)

[SCIC] stellt eine Möglichkeit vor, die Eigenschaften eines CIC-Dezimators mit Hilfe von Filter-Schärfung nach Hamming und Kaiser [HK77] so zu optimieren, dass die Dämpfung des Passbands reduziert wird, das Transitionsband steiler wird und die damit verbundenen Aliasing-Effekte verringert werden.

SCIC-Dezimatoren sind CIC-Dezimatoren mit einer „geschärften“ Übertragungfunktion. Mithilfe der Methode des Filterschärfens von Kaiser und Hamming [HK77] lässt sich die Übertragungsfunktion jedes symmetrischen FIR-Filters durch die Verwendung mehrerer gleicher Filter verbessern, so dass die Dämpfung im Stoppband vergrößert wird und die Übertragungsfunktion im Bereich des Passbands steiler wird. Die Anwendung dieser Methode auf CIC-Dezimatoren führt direkt zu SCIC-Dezimatoren, die eine ähnliche Übertragungsfunktion aufweisen wie CIC-Dezimatoren, nur mit einem flacheren Verlauf im Bereich des Passbands und einer höheren Alias-Unterdrückung auf Kosten größeren Hardwareaufwands. Ein SCIC-Dezimator besteht aus drei CIC-Dezimatoren, dessen Aufbau im Folgenden beschrieben wird.

Obwohl der vergrößerte Hardwareaufwand zunächst die Frage aufwirft, welchen Vorteil SCIC-Dezimatoren im Vergleich zu CIC-Dezimatoren in Hinsicht auf VLSI-Design haben, kann durch den Einsatz eines SCICs insgesamt Hardwareaufwand eingespart werden, vor allem dann, wenn der Dezimationsfaktor eines Systems zur Laufzeit änderbar ist.

Wie bereits erwähnt, werden CIC-Dezimatoren von weiteren Dezimationsfiltern gefolgt. Das Filter muss so geartet sein, dass der vom CIC hervorgerufene Passbanddroop möglichst ausgeglichen wird. In der Regel wird hierzu ein FIR-Equalizer verwendet. Wenn der Dezimationsfaktor des CICs während der Laufzeit änderbar ist, müssen die Koeffizienten des FIR-Equalizers zur Laufzeit programmierbar sein, was einen großen Hardwareaufwand bewirkt.

Aufgrund der verbesserten Frequenzcharakteristik des SCICs kann der FIR-Equalizer durch ein nicht-programmierbares FIR-Filter ersetzt werden, wodurch der gesamte Hardwareaufwand kleiner ist als für die herkömmliche Variante mit einem CIC-Dezimator.

Im Gegensatz zu CIC-Dezimatoren wird der Freiheitsgrad bezüglich des Parameters M nicht gegeben, da sich optimale Ergebnisse ohnehin für $M = 1$ ergeben und - wie sich in langer Kleinarbeit heraustellte - ein SCIC-Dezimator mit dem Parameter M nicht handhabbar ist.

7.1. Herleitung der Übertragungsfunktion eines SCIC-Dezimators

Um ein FIR-Filter zu „schärfen“, wird laut Kaiser und Hamming [HK77] eine neue Übertragungsfunktion gebildet durch

$$H_{new}(z) = H^2(z)[3 - 2\,H(z)], \tag{7.1}$$

wobei $H_{new}(z)$ die neue, verbesserte Übertragungsfunktion darstellt und $H(z)$ die des ursprünglichen symmetrischen FIR-Filters.

Der Amplitudengang des geschärften Filters ist

$$\left|H(e^{j\omega})\right| = \left|H^2(e^{j\omega})[3 - 2H(e^{j\omega})]\right|. \tag{7.2}$$

Es ist möglich, diese Methode auch auf einen CIC-Dezimator anzuwenden. Wird $H(z)$ zu

$$H_{CIC}(z) = \left\{\frac{1 - z^{-R}}{1 - z^{-1}}\right\}^N \tag{7.3}$$

$$H_{CIC}\left(e^{j\omega}\right) = \left\{\frac{sin\frac{\omega R}{}}{sin\frac{\omega}{2}} e^{-j\omega[(R-1)/2]}\right\}^N, \tag{7.4}$$

ergibt sich folgender Amplitudengang für einen SCIC-Dezimator für gerade N

$$\left|H_{SCIC}(e^{j\omega})\right| = \left|\left[3\left(\frac{sin\ \pi Rf}{R\ sin\ \pi f}\right)^{2N}\right] - \left[2\left(\frac{sin\ \pi Rf}{R\ sin\ \pi f}\right)^{3N}\right]\right|, \tag{7.5}$$

wobei innerhalb der CIC-Dezimatoren nicht dezimiert wird, aber die Anzahl der Verzögerungen der Differentiatoren in diesem Fall dem Dezimationsfaktor entspricht. N muss gerade sein, da der zweite Anteil sonst wegen des ungeraden Exponenten negative Anteile in die Berechnung einbringt, die das Übertragungsverhalten verfälschen.

Bild 7.1 zeigt das Blockschaltbild eines SCIC-Dezimators, der nicht normalisierte CIC-Dezimatoren verwendet, weshalb eine Multiplikation mit R^N in der unteren Verzweigung vorgenommen wird, um die Verstärkung des CIC-Dezimators in der oberen Verzweigung zu kompensieren. Die Verzögerung im unteren Zweig dient dazu, die Gruppenverzögerung eines CIC-Filters auszugleichen, die Gleichung (7.4) entnommen werden kann und $(R-1)$ Takte beträgt. Im Gegensatz zu der herkömmlichen CIC-Implementierung wird bei den Bauteilen eines SCIC-Dezimators in den CICs keine Dezimation vorgenommen, aber eine Verzögerung in den Differentiatoren um R durchgeführt.

7.2. Frequenzcharakteristik

Die Übertragungsfunktion des SCIC-Dezimators wird in Bild 7.2 dargestellt. Das Pass- und Aliasing-Band liegt an denselben Stellen wie die des CIC-Dezimators.

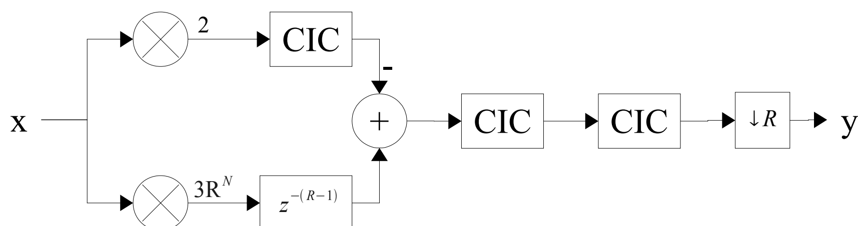


Bild 7.1.: Blockschaltbild eines SCIC-Dezimators

Der Amplitudengang ist im Bereich des Passbands wesentlich flacher und der Passbanddroop ist geringer als der eines CIC-Dezimators, während gleichzeitig eine höhere Dämpfung und ein schmaleres Transitionsband auffallen. Ein SCIC-Dezimator ermöglicht somit auf der einen Seite eine Verwendung eines nachfolgenden FIR-Dezimationfilters mit festen Koeffizienten anstelle eines FIR Equalizers und es ist möglich, das Passband zu verbreitern. Durch das flachere Passband kann der Dezimationsfaktor R des SCIC-Dezimators gegenüber dem Dezimationsfaktor des CIC-Dezimators größer gewählt werden.

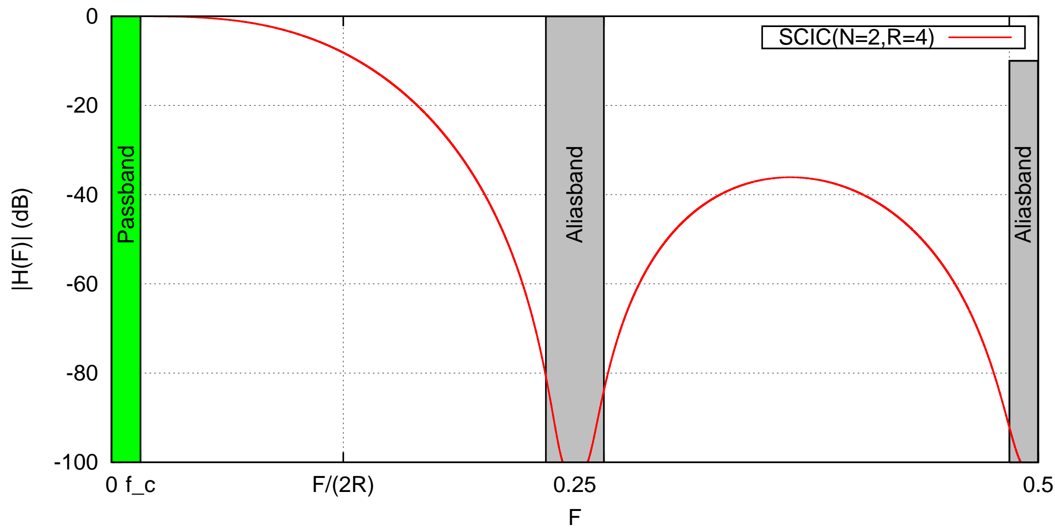


Bild 7.2.: Amplitudengang eines SCIC-Dezimators für R=4 und N=2

7.3. Registerdimensionierung

Ähnlich wie für die CIC-Dezimatoren muss auch für SCIC-Dezimatoren die Dimensionierung der Register besonders berücksichtigt werden. Um die maximale Verstärkung eines SCIC-Dezimators zu ermitteln, wird dessen Übertragungsfunktion bei Gleichspannung berechnet. Hierbei kann auf die Ergebnisse der Verstärkung eines CIC-Dezimators zurückgegriffen werden.

Die Übertragungsfunktion eines SCIC-Dezimators unter Berücksichtigung der Verstärkung in dem unteren Zweig lautet

$$H_{SCIC}(z) = H^2(z)[3R^N - 2H(z)], \tag{7.6}$$

wobei

$$H(z) = \left\{\sum_{k=0}^{R-1} z^{-k}\right\}^N . \tag{7.7}$$

Eingesetzt ergibt sich

$$H_S(z) = \left[\sum_{k=0}^{R-1} z^{-k}\right]^{2N} \cdot \left\{3R^N - 2\left[\sum_{k=0}^{R-1} z^{-k}\right]^N\right\}. \tag{7.8}$$

Wird nun $z = 1$ eingesetzt, ergibt sich Gleichspannungsverstärkung für einen gesamten SCIC-Dezimator

$$G_{max} = R^{2N} \left\{ 3R^N - 2R^N \right\} = R^{3N}. \tag{7.9}$$

Die interne Registerbreite muss so groß sein, dass das Eingangssignal multipliziert mit G_{max} darin Platz hat. Innerhalb der Verzweigung müssen die Register unter Berücksichtigung der weiteren Multiplikationen mit 3 respektive -2 dimensioniert werden, so dass für den oberen Zweig ein weiteres Bit und für den unteren Zweig zwei weitere Bits für die Register vorgesehen werden müssen.

$$B_{max} = \lceil log_2\, G_{max} + B_{in} + 2 \rceil = \lceil 3N\, log_2\, R + B_{in} + 2 \rceil \text{ Bit}, \tag{7.10}$$

wobei B_{in} die Registerbreite der Eingangssignale darstellt.

Im Vergleich zu einem CIC-Dezimator sind die Register also dreimal so groß. Allerdings kann die Stufenanzahl N für einen SCIC-Dezimator gegenüber der des CIC-Dezimators halbiert werden, um vergleichbare Ergebnisse zu erzielen, wodurch sich ein um 50% erhöhter Hardwarebedarf für einen SCIC- verglichen mit einem CIC-Dezimator ergibt.

Besonders für programmierbare Dezimationsfilter eignet sich dennoch ein SCIC-Dezimator besser, da dieser nicht wie ein CIC-Dezimator von einem ebenfalls programmierbaren FIR-Equalizier gefolgt werden muss, um das Passbanddroop auszugleichen. Somit ist der gesamte Hardwareaufwand bei Verwendung eines SCIC-Dezimators trotz des zunächst höheren Hardwareaufwands der Predezimation geringer.

7.4. Hardwareoptimierung

Es wurde gezeigt, dass der Hardwareaufwand eines CIC-Dezimators durch Umstellung der Bauelemente reduziert werden kann, indem die Dezimation vor der Differentiatiation durchgeführt wird. Derselbe Ansatz lässt sich auch auf SCIC-Dezimatoren anwenden.

Zunächst wird ein SCIC-Dezimator aus einzelnen Bauelementen der CIC-Dezimatoren aufgebaut (Bild 7.3a). Es ist möglich, die Dezimation vor der Differentiation durchzuführen, wenn alle Integratoren von CIC#2 und CIC#3 vor die Verzweigung verlegt werden, um eine Reduzierung der Verzögerer in den Differentiatoren zu erzielen. Die Dezimation geschieht innerhalb der Verzweigung, ähnlich wie bei dem CIC-Dezimator zwischen Integratoren und Differentiatoren und im unteren Zweig hinter dem Verzögerer, dessen Größe sich dabei von $z^{-(R-1)}$ auf z^{-1} verringert (Bild 7.3b).

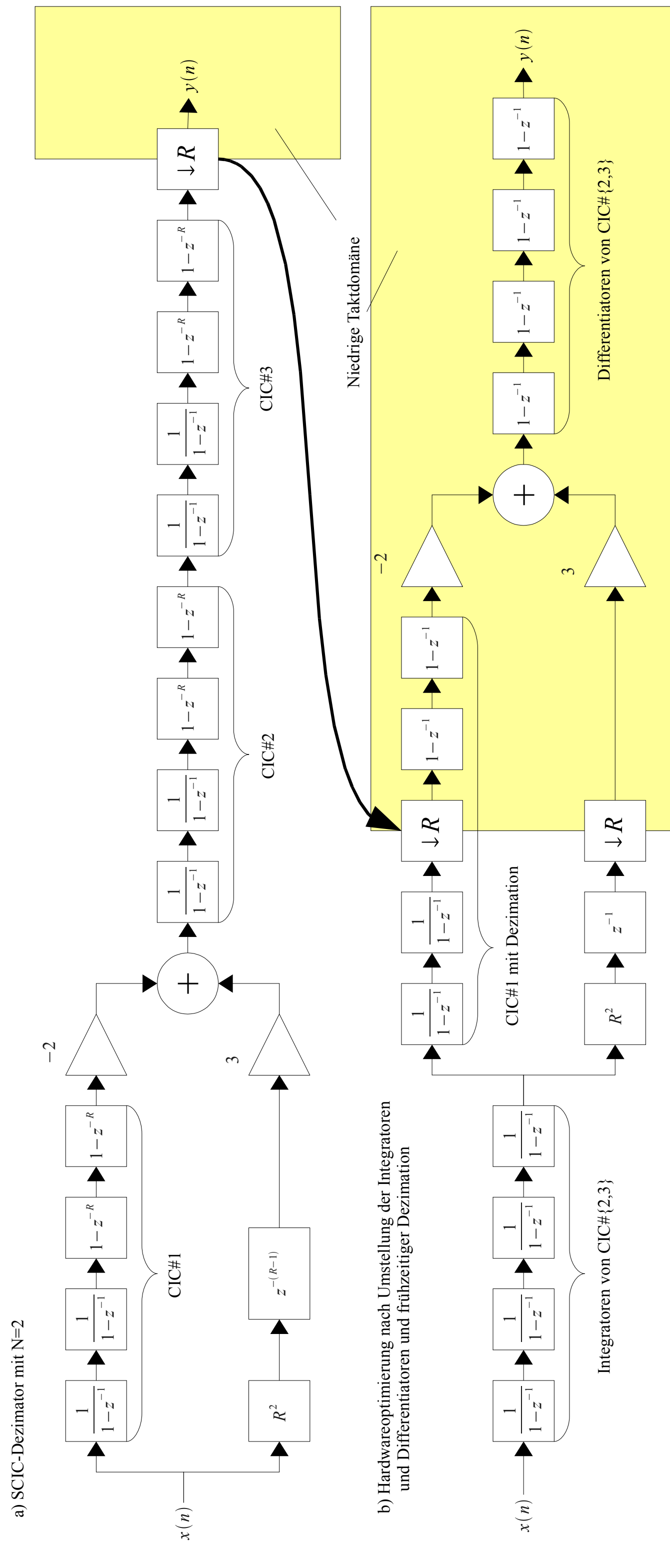


Bild 7.3.: Hardwareoptimierung eines SCIC-Dezimators

7.5. Vergleich von CIC- und SCIC-Dezimatoren

CIC- und SCIC-Dezimatoren können auch anhand der Merkmale Passbanddroop und Aliasrejection miteinander verglichen werden. Passbanddroop ist die Dämpfung an der höchsten Frequenz des Passbands und Aliasrejection ist die Dämpfung an der Stelle, an der das maximale Aliasing auftritt f_{A1} (s. Bild 7.2 und Kap. 6).

Bild 7.4a beinhaltet die Übertragungsfunktionen eines CIC- und einen SCIC-Dezimators mit jeweils N=2 und R=4. Der Parameter M des CIC-Dezimators ist 1. Bild 7.4b zeigt den vergrößerten Bereich des Passbands der Übertragungsfunktion.

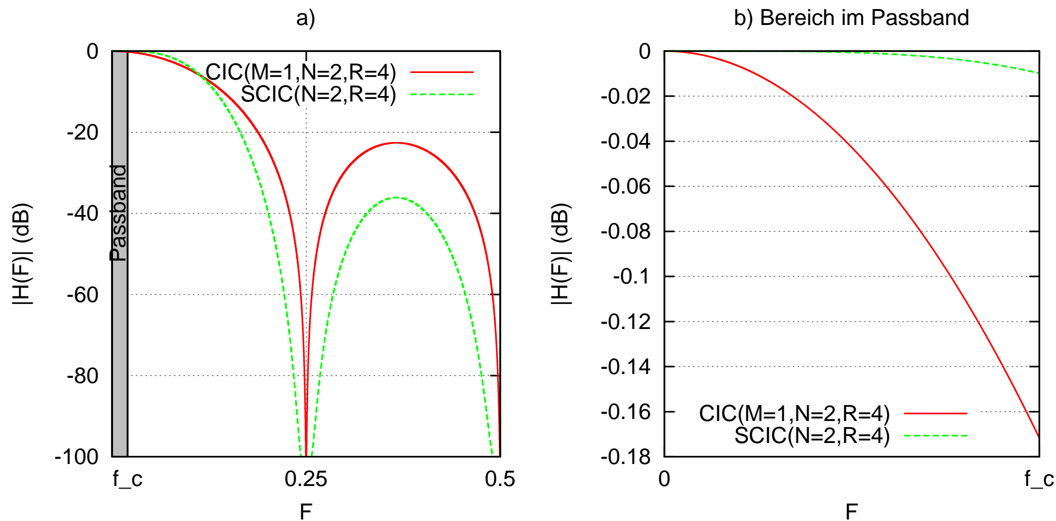


Bild 7.4.: Vergleich der Übertragungsfunktionen von SCIC- und CIC-Dezimatoren. M=1, N=4 und R=8

In [SCIC] wird ein ausführlicher Vergleich beider Dezimationsfilter vorgenommen. Ein Vergleich in tabellarischer Form wurde im Rahmen der Diplomarbeit durchgeführt und kann in Anahng 13.1 eingesehen werden.

Der SCIC-Dezimator ist dem CIC-Dezimator bezüglich der Dämpfung im Passbandbereich weitaus überlegen. Die Aliasrejection ist bei gleichem N etwa doppelt so groß, weshalb es genügt, N zu halbieren, wenn ein CIC- durch einen SCIC-Dezimator ersetzt werden soll. Der Passbanddroop ist für große Dezimationsfaktoren ($R \geq 10$) etwa konstant [SCIC].

7.6. Vor- und Nachteile des SCIC-Dezimators

Ein SCIC-Dezimator weist folgende Vor- und Nachteile auf.

Vorteile:

- Ein SCIC-Dezimator benötigt wie ein CIC-Dezimator keine Multiplizierer
- Es wird kein Speicherplatz für Filterkoeffizienten benötigt, da sämtliche Koeffzienten 1 sind
- Dadurch, dass die Differentiation in einer niedrigeren Frequenz stattfindet, verringert sich der benötigte Speicherplatz für die Verzögerungsglieder um den Faktor R
- Ein SCIC-Dezimator wird aus zwei sich wiederholenden Blöcken erstellt, wodurch sich die Implementierung vereinfacht
- Es wird relativ wenig externe Logik benötigt, um die Dezimation zu realisieren, z.B. ein One-Hot-Zähler
- Ein Filter-Entwurf kann für eine Vielzahl verschiedener Dezimations-Faktoren verwendet werden, da lediglich das Zeitverhalten der Differentiationsstufen angepasst werden muss
- Die Frequenzcharakteristik hat bessere Eigenschaften als die eines CIC-Dezimators mit geringerem Passbanddroop und höherer Aliasrejection
- Aufgrund der geringen Dämpfung im Passband (Passbanddroop) ist es möglich, auf einen programmierbaren FIR-Equalizer zu verzichten, wodurch sich der Hardwareaufwand insgesamt reduziert
- Wenn besonders große Dezimationsfaktoren realisiert werden sollen, kann ein SCIC-Dezimator einem CIC-Dezimator folgen
- Ein SCIC-Dezimator kann in einer schnellen Schaltung realisiert werden

Nachteile:

- Aufwändigere Implementierung und Handhabung gegenüber dem CIC-Dezimator
- Die Register sind etwa um die Hälfte größer als die eines vergleichbaren CIC-Dezimators
- Der Freiheitsgrad bezüglich des Parameters M ist gegenüber dem CIC-Dezimator nicht vorhanden (aber i.d.R. auch nicht notwendig)

7.7. Gegenüberstellung des CIC- und SCIC-Dezimators

Es wurden zwei verschiedene Dezimatoren vorgestellt, die unterschiedliche Eigenschaften bezüglich der Komplexität, Flexibilität, Genauigkeit und der Handhabbarkeit aufweisen.

Der CIC-Dezimator ist gut handhabbar und lässt sich mit geringem Aufwand in einer schnellen Schaltung realisieren. Bezüglich der Qualität der Übertragungscharakteristik fällt das relativ hohe Passbanddroop negativ ins Gewicht.

Der SCIC-Dezimator weist ein erheblich geringeres Passbanddroop verglichen mit dem des CIC-Dezimators auf. Allerdings resultiert dieser in einer weitaus komplexeren Schaltung, die schlechter handhabbar ist und aufgrund des fehlenden Parameters M eine geringere Flexibilität aufweist.

In Tabelle 7.1 werden die Bewertungen der Eigenschaften der vorgestellten Dezimatoren vergleichend dargestellt.

Dezimatortyp	Komplexität	Qualität	Handhabbarkeit
CIC-Dezimator	$\oplus\oplus\oplus$	$\ominus$	$\oplus\oplus\oplus$
SCIC-Dezimator	$\oplus$	$\oplus\oplus\oplus$	$\ominus\ominus\ominus$

Tabelle 7.1.: Bewertung der Dezimatoren

Implementierung

Implementierung

In diesem Teil der Diplomarbeit werden ein FPGA erklärt und die im vorherigen Teil vorgestellten Komponenten in VHDL implementiert, wobei der Taylorreihen- und der CORDIC-Oszillator hierbei außer Acht gelassen werden, da sich diese, wie im vorherigen Teil festgestellt wurde, aufgrund der hohen Komplexität wenig für eine Hardwareimplementierung in VHDL für FPGAs eignen.

Es wird die Realisierung eines Digital-Down-Converters mit Hilfe von VHDL und FPGAs dargestellt, indem die zuvor beschriebenen Komponenten miteinander kombiniert werden. Dabei wird versucht, unter Berücksichtigung von Kosten, Aufwand und den oben genannten Aspekten eine optimale technische Lösung zu finden.

Die Grundlagen für die VHDL-Implementierungen bezüglich der verwendeten Computer-Arithmetik und der Architektur von FPGAs befinden sich in den Anhängen A bzw. 8.

Die VHDL-Implementierungen der Komponenten sind notwendig, um im späteren Teil deren Syntheseergebnisse miteinander vergleichen zu können.

8. Field-Programmable-Gate-Array

Der Begriff Field-Programmable-Gate-Array oder kurz FPGA steht für einen elektronischen Logikbaustein, der rekonfigurierbar ist. *Field-Programmable* bedeutet hier *außerhalb der Fabrik progammierbar*. Ein FPGA ist ein integrierter Schaltkreis, der aus verschiedenen Teilen besteht, die durch spezifische Zusammenschaltung verschiedene digitale Schaltungen realisieren können. Diese Teile sind:

1. Ein- und Ausgabeblöcke
2. Logikblöcke
3. programmierbare Verbindungen

Bild 8.1 beinhaltet den schematischen Aufbau eines FPGAs. Aktuell besteht ein typischer FPGA-Logikblock (beispielsweise von Altera [ALT1]) (Bild 8.2) aus einem Look-Up-Table mit vier Eingängen und einem Ausgang, einem taktsychronen Flipflop und einem Multiplexer. Mit Hilfe des Multiplexers wird entschieden, ob das Ausgangssignal taktsynchron oder direkt weitergegeben wird.

Im Vergleich zu einem CPLD (Complex-Programmable-Logic-Device) besitzt ein FPGA eine wesentlich größere Anzahl von Logikblöcken, wodurch sich ein FPGA für komplexere Anwendungen eignet und wesentlich höhere Leistungen erzielt, auf Kosten einer komplizierteren Vorhersagbarkeit des Zeitverhaltens (Timings) und höheren Kosten.

Aktuelle FPGAs besitzen integrierte Funktionsblöcke wie Multiplizierer, Addierer, RAM, ROM etc. und teilweise sogar integrierte Mikroprozessoren.

Die Anwendungsgebiete eines FPGAs sind:

- Digitale Signalverarbeitung
- Datenspeicherung und -prozessierung
- ASIC-Prototypen
- Spracherkennung
- Bilderkennung
- Kryptographie
- Medizintechnik
- Verteidigungstechnik
- Bioinformatik
- Computer-Hardware-Emulation
- Reconfigurable-Computing

Weitere Informationen können [DiEl1] entnommen werden.

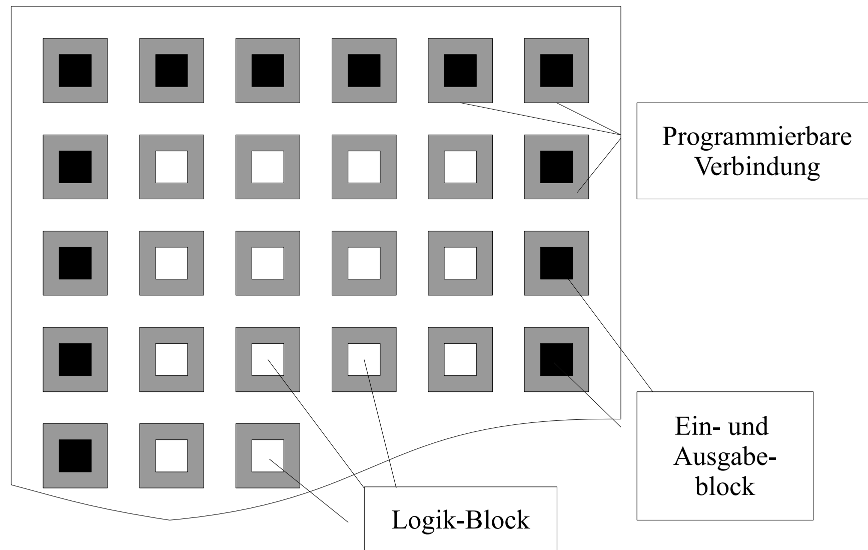


Bild 8.1.: Schema eines FPGAs

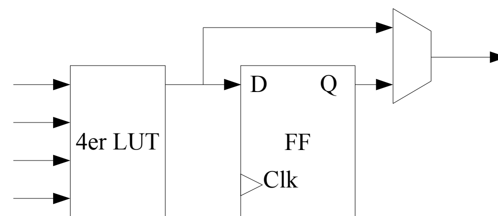


Bild 8.2.: Logikblock eines FPGAs

8.1. Programmierbare Verbindung von Logikblöcken

In einem FPGA geschieht die Verbindung bzw. Verschaltung von Logikblöcken mit Hilfe von SRAM-Zellen. SRAM oder *Statischer RAM* ist ein flüchtiger Halbleiterspeicher, d.h. jede SRAM-Zelle speichert eine Information solange die Versorgungsspannung anliegt. Die Konfiguration der SRAM-Zellen wird nach einem Reset des FPGAs in der Regel aus einem EEPROM geladen. Die Ausgänge der SRAMs steuern CMOS-Transistoren an, die die sich kreuzenden Verbindungsleitungen miteinander verbinden können wie in Bild 8.3 schematisch dargestellt wird.

8.2. Design-Flow von FPGAs

Die Programmierung eines FPGAs beginnt mit der Erstellung von Quelltexten in einer Hardware-Beschreibungssprache wie z.B. *VHDL* oder *Verilog*. Dieser Schritt wird als *Design-Entry* bezeichnet. Der

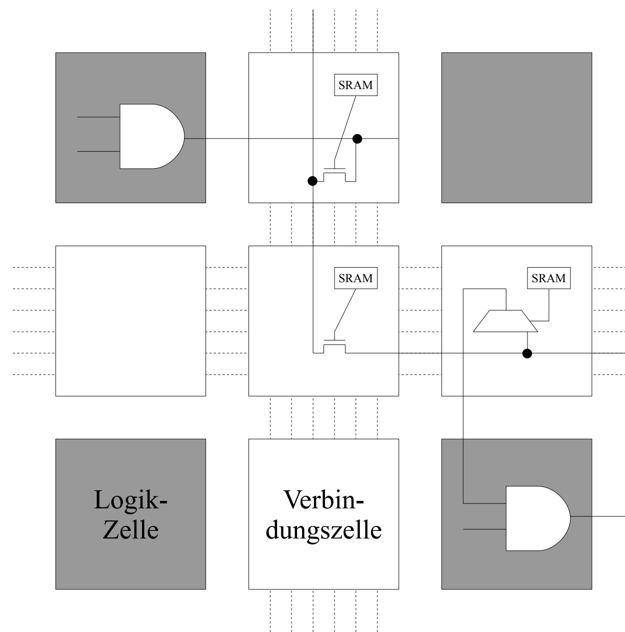


Bild 8.3.: Schematische Darstellung der Verbindung zweier Logikzellen in einem FPGA

grundsätzliche Unterschied zwischen Hard- und Softwaredesign ist, dass beschriebene Hardware *parallel* funktioniert, während Software in der Regel sequentielle Befehle enthält. Alle Eingangssignale einer in VHDL (oder Verilog) beschriebenen Logikschaltung durchlaufen eine Reihe von Logikzellen und Verbindungszellen gleichzeitig in Richtung deren Ziel. Informationen, die von Macrozelle zu Macrozelle wandern, werden in der Regel durch getaktete Register synchronisiert.

Als zweiter Schritt folgt die *Simulation*. Die Simulation führt einen Design-Entwurf mit vorgegebenen Eingangssignalen aus und zeigt Ausgangssignale bzw. interne Signale an. Die Simulation dient dem Programmierer, die korrekte Funktionsweise des Entwurfs zu validieren. Schlägt die Simulation fehl (Funktion ist nicht korrekt), werden Schritt 1 und 2 solange wiederholt, bis das Ergebnis korrekt ist.

An dritter Stelle in dem Entwicklungsprozess einer FPGA-Hardware steht die *Synthese*. Die Synthese erstellt eine *Netzliste* mit einer Hardware-Bibliothek aus der vorgegebenen Hardwarebeschreibung. Diese Netzliste ist geräteunabhängig und von keinen Parametern der Zielhardware abhängig. Anschließend folgt *Place-And-Route*. Hierbei wird die logische Struktur der Netzliste so umgewandelt, dass diese durch Ein- und Ausgabeblöcke, Verbindungsblöcke und Logikblöcke realisiert wird. Dieser Schritt erzeugt eine Bit-Kette in Form einer Binärdatei, mit der der FPGA initialisiert werden kann. Die Schritte Synthese und Place-And-Route bilden die sog. *Hardwarecompilierung*. Es sollte bemerkt werden, dass das Ergebnis von Place-And-Route ebenfalls mit allen Zeitabhängigkeiten simuliert werden kann.

Als letzter Schritt erfolgt der *Download*, durch den die im vorigen Schritt erzeugte Binärdatei in das EEPROM des FPGAs heruntergeladen wird. Der FPGA ist anschließend einsatzbereit.

Bild 8.4 illustriert die Schritte des Entwicklungsprozesses in Form eines Flussdiagramms.

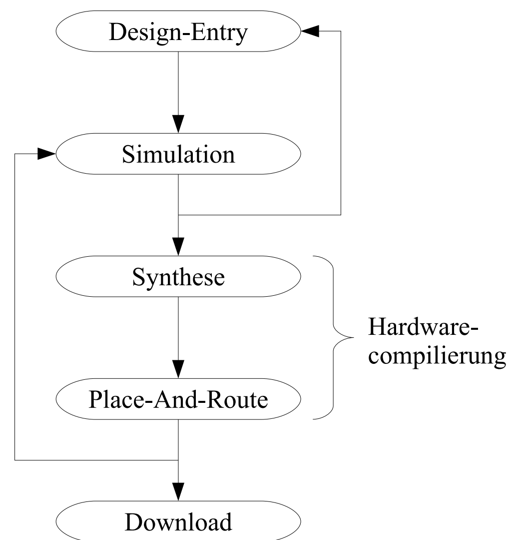


Bild 8.4.: Ablauf von Design und Entwicklung einer FPGA-Hardware

8.3. Software

Im Rahmen der Diplomarbeit wird für den Schritt *Design-Entry* das Programm *HDL-Author v6.1* der Firma *Mentor Graphics*[1] verwendet, für *Simulation* das Programm *Modelsim SE Plus 6.2i* derselben Firma eingesetzt.

Für die *Hardwarecompilierung* und den *Download* wird *Quartus v7.2* der Firma *Altera* verwendet, welches den Einsatz auf Altera-FPGAs beschränkt.

[1] http://www.mentor.com

9. Implementierung der Oszillatoren in VHDL

9.1. Implementierung eines Lookuptable-Oszillators in VHDL

Die VHDL-Implementierung des Lookuptable-Oszillators realisiert einen selbstfüllenden Lookuptable, der eine gesamte Periode von Sinus und Cosinus beinhaltet und einen Modulo-N-Zähler (Bild 9.1). Der LUT wird durch eine eigenständige Komponente gebildet, die die Takt- und Ausgangsfrequenz als Parameter erhält. Während der Compilierung wird intern ein Array mit vorausberechneten Werten befüllt. Die Ansteuerung und die Prozesse innerhalb der LUT-Komponente entsprechen den Vorschlägen der Hilfefunktion von Altera-Quartus, so dass automatisch ein LUT synthetisiert wird. Diese Realisierung bietet den großen Vorteil, herstellerunabhängig und damit beliebig portierbar zu sein. Es ist nicht notwendig, herstellerspezifische *Megafunctions*[1] für den LUT zu verwenden, wodurch eine Initialisierung mit MIF- oder HEX-Dateien[2] entfällt. Die Quelltexte beider Komponenten befinden sich in Anhang B.2 auf Seite 136ff.

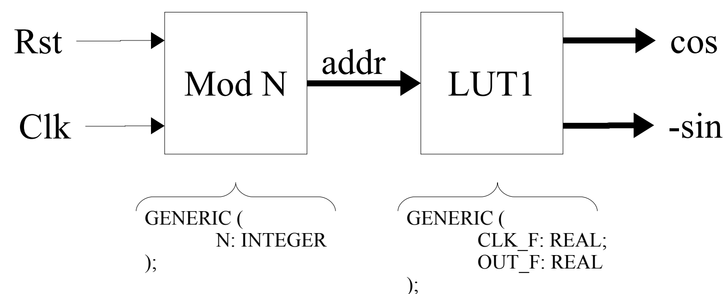


Bild 9.1.: Implementierung des Lookuptable-Oszillators

[1]Eine Megafunction ist ein Funktionsblock für FPGAs der Firma Altera.
[2]MIF- bzw. HEX-Dateien sind Textdateien, die zur Initialisierung von RAM und ROM eines FPGAs dienen.

9.2. Implementierung eines IIR-Oszillators in VHDL

9.2.1. Zählender IIR-Oszillator

Es wird ein IIR-Oszillator implementiert, der durch einen Zähler periodisch neuinitialisiert wird. Für die Neuinitialisierung werden zwei Multiplexer benötigt, die zwischen den vorherigen Werten und den Initialisierungswerten umschalten. Beide Multiplexer werden durch einen One-Hot-Zähler gesteuert. Durch die eingefügten Multiplexer verlängert sich der kritische Pfad. Es müssen pro Takt ein Multiplizierer, ein Addierer und der Multiplexer durchlaufen werden, was zu einer niedrigeren Taktfrequenz führt. Der schematische Aufbau der Implementierung (Listing B.3 auf Seite 139) befindet sich in Bild 9.2.

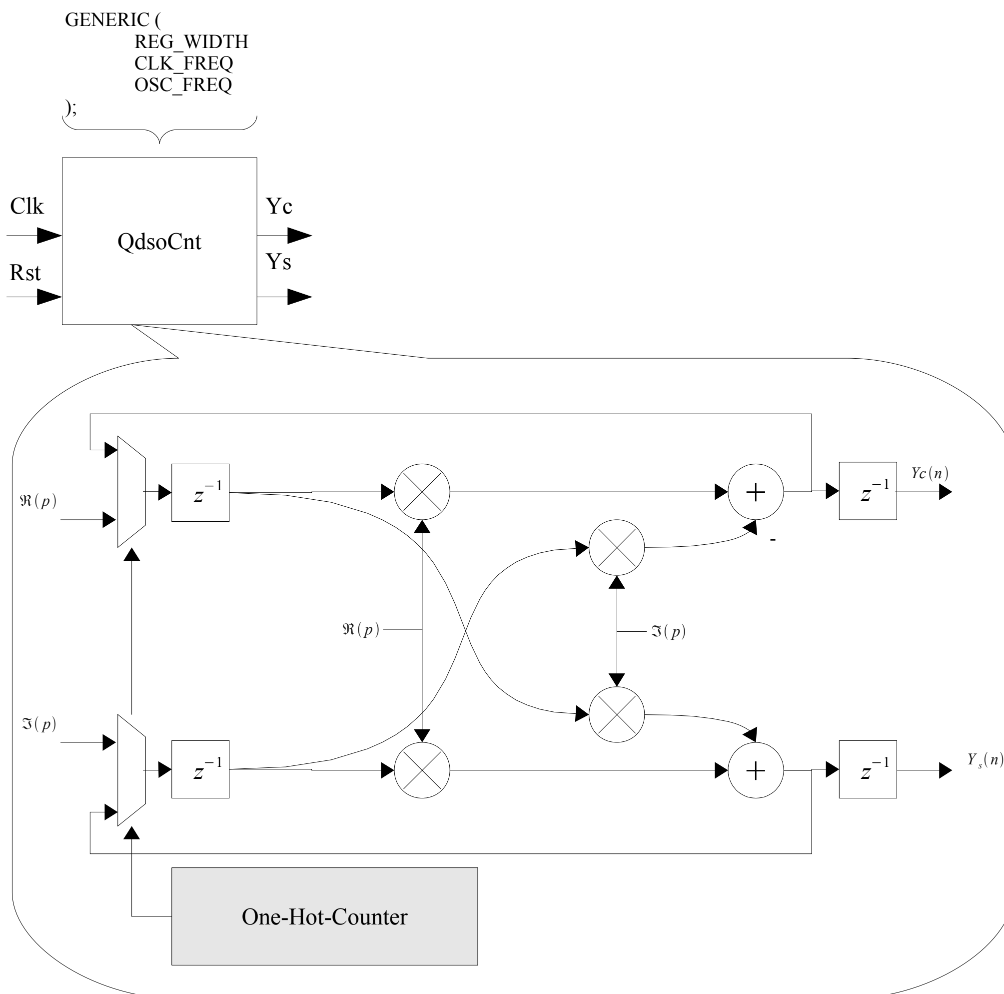


Bild 9.2.: Implementierung des zählenden IIR-Oszillators

9.2.2. Überwachender IIR-Oszillator

Die Implementierung (Bild 9.3) des überwachten IIR-Oszillators unterscheidet sich von der vorherigen Implementierung nur darin, dass anstelle des Zählers ein Block verwendet wird, der überprüft, ob sich das Ausgangssignal innerhalb des definierten Bereichs für die Nulldurchgangserkennung befindet und ggf. eine Neuinitialisierung veranlasst. Der Quelltext der Implementierung befindet sich in Listing B.4 auf Seite 142.

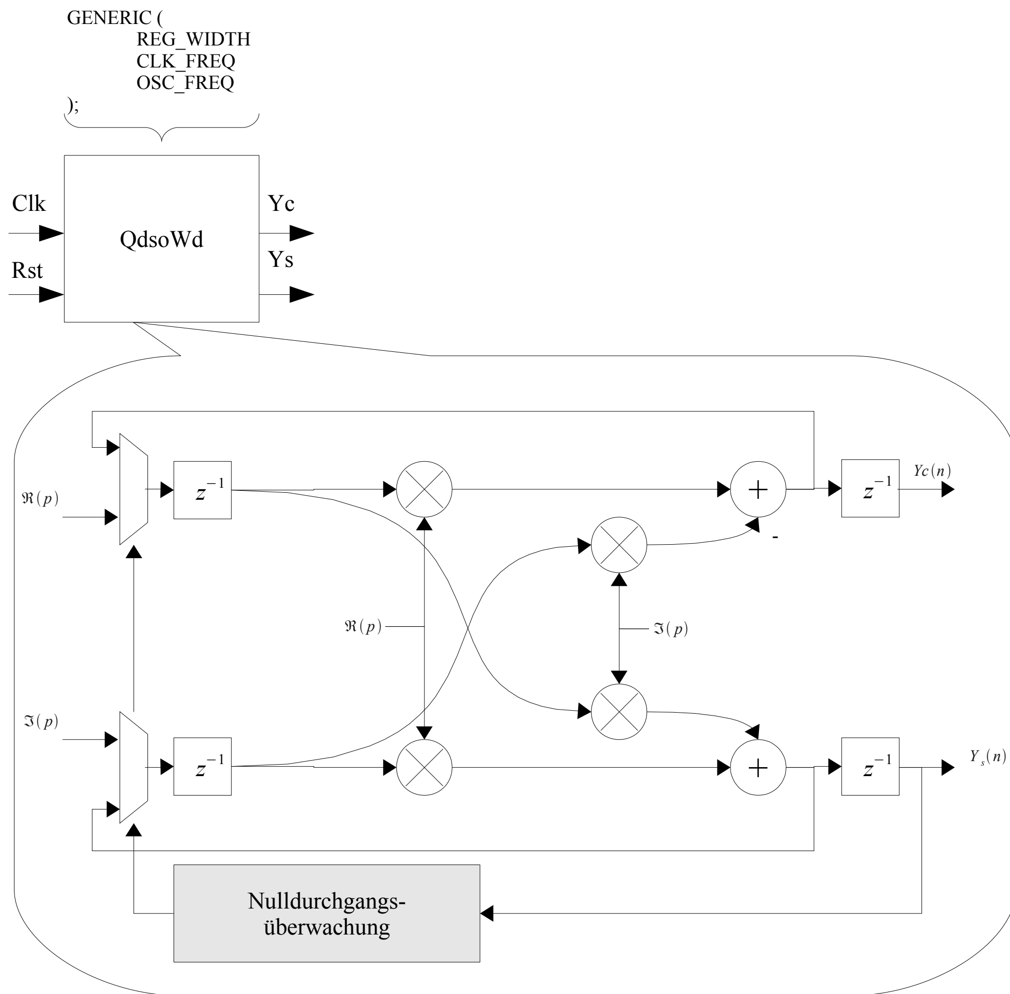


Bild 9.3.: Implementierung des überwachten IIR-Oszillators

9.2.3. Adaptiver IIR-Oszillator

Der adaptive IIR-Oszillator benötigt einen Multiplizierer, der permanent das Quadrat der Amplitude errechnet. Dieser Block wird in einer eigenständigen Komponente realisiert, die eine serielle Multiplikation durchführt. Der Quellcode befindet sich in Listing B.6 auf Seite 148. Die Schaltung, die die Oszillation realisiert, unterscheidet sich von den vorherigen Schaltungen darin, dass zwischen verschiedenen Koeffizienten umgeschaltet werden kann, so dass die Multiplexer nicht vor den Registern liegen, sondern die Multiplizierer mit Koeffizienten speisen. Der Quelltext befindet sich in Listing B.5 auf Seite 145. Beide Komponenten werden durch den Toplevel-Block (Bild 9.4) zusammengeschaltet, dessen Quelltext sich in Listing B.7 auf Seite 151 befindet.

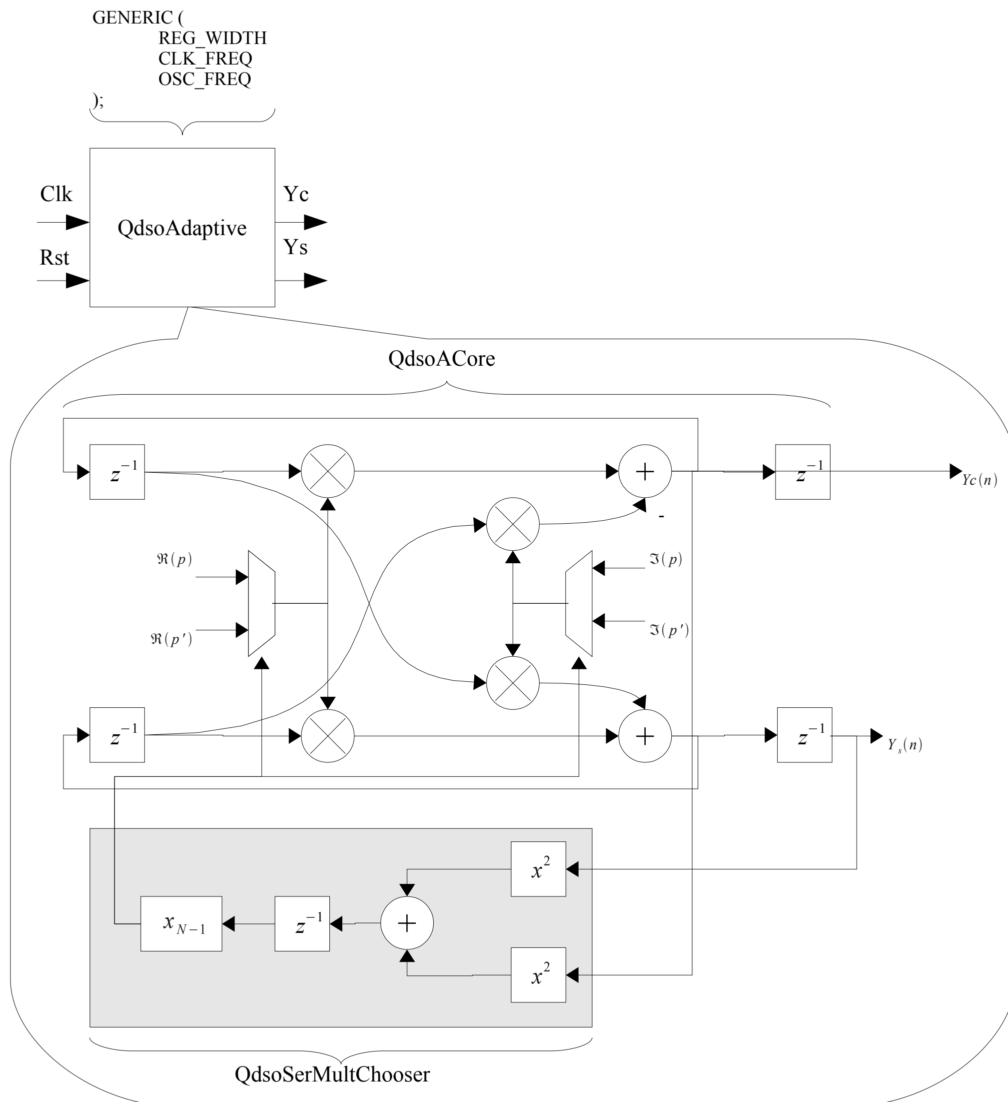


Bild 9.4.: Implementierung des adaptiven IIR-Oszillators

9.3. Gegenüberstellung der Syntheseergebnisse

In Tabelle 9.1 werden einige Syntheseergebnisse des LUT-Oszillators und den IIR-Oszillatoren gegenübergestellt, die mit Quartus 7.2 für einen Stratix III EP3SE50F484C2 FPGA erstellt wurden.

Die Ergebnisse sind die Anzahl der benötigten „Combinatorial ALUTs“, die Anzahl der „Dedicated Logicregisters“, die maximale Taktfrequenz und die Anzahl benötigter Hardwaremultiplizierer bzw. Speicherbedarf.

Typ	Comb. Aluts	Register	F_{max}/MHz	Multiplizierer	Memblocks
LUT	7	12	480,08	0	2048
IIR-Adaptive	891	182	192,20	4	0
IIR-Counting	459	42	204,62	4	0
IIR-Überwacht	465	32	158,53	4	0

Tabelle 9.1.: Syntheseergebnisse verschiedener Oszillatoren für die Ausgangsfrequenz $\pi/20$ und 16 Bit Genauigkeit von Quartus 7.2 für einen Stratix III FPGA

10. Implementierung eines FIR-Dezimationsfilters in VHDL

Die VHDL-Implementierungen der FIR-Dezimations-Filter werden in transponierter Form und unter Ausnutzung symmetrischer Koeffizienten vorgenommen.

Der Vollständigkeit halber werden alle Möglichkeiten der Addition und Verzögerung eines FIR-Filters in transponierter Form vorgestellt:

1. nichtsymmetrisch, (un)gerade Koeffizientenanzahl (Bild 10.1a)
2. symmetrisch, gerade Koeffizientenanzahl (Bild 10.1b)
3. symmetrisch, ungerade Koeffizientenanzahl (Bild 10.1c)

Da für die FIR-Filter stets ein linearer Phasengang benötigt wird, nutzen alle Implementierungen die Symmetrie der Koeffizienten aus. Um FIR-Filter für verschiedene Datenraten zur Verfügung zu stellen, wurden je eine Implementierung mit Hardwaremultiplizierern und Booth-4-Multiplizierern (C.1,

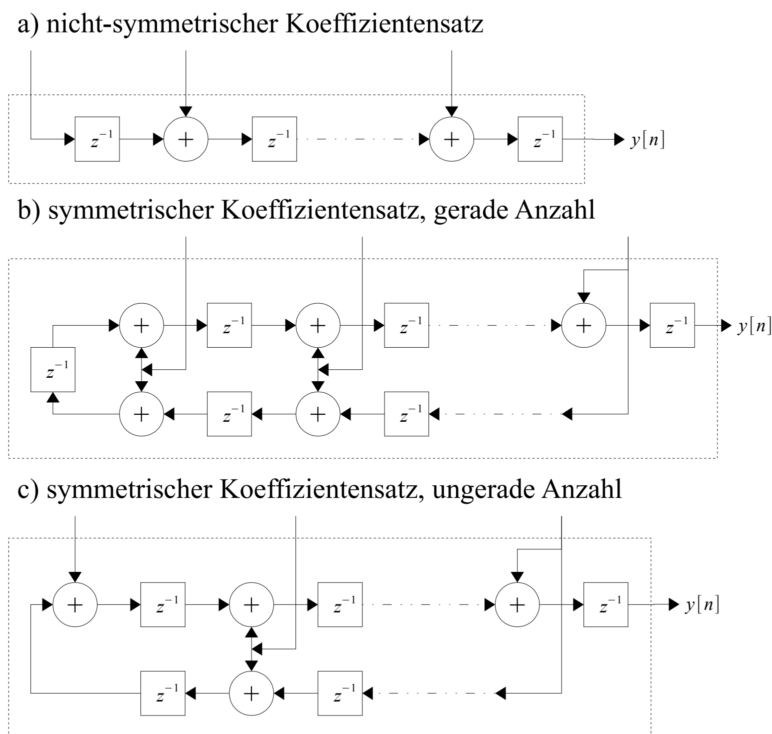


Bild 10.1.: FIR-Addier-Blöcke

Seite 155) erstellt. Die Booth-Multiplizierer können ohne großen Aufwand bei Bedarf auch durch serielle Multiplizierer (C.2, Seite 158) ersetzt werden, um eine dritte Implementierung für niedrigere Datenraten zu erstellen.

Die VHDL-Komponenten sind generisch, so dass der Benutzer das Verhalten nur durch Verändern der Generics bestimmen kann. Die Koeffizienten der Filter können mit Hilfe eines VHDL-Packages gespeichert werden. Das VHDL-Package kann mehrere Konstanten enthalten, die jeweils einem Koeffizienten-Array entsprechen. Per Generic kann dem Filter während der Kompilierung so eine Konstante übergeben werden. Die Filterkoeffizienten werden als Fließkommazahlen gespeichert (Listing 10.1) und können ebenfalls zur Kompilierung intern umgerechnet werden.

Listing 10.1: Beispiel für Koeffizientenübergabe für FIR-Filter in VHDL

```
1  CONSTANT pcMAX_COEFFS : natural := 1024;
2  type T_Coeff is array (natural range <>) of real;
3
4  CONSTANT pcFIR_COEFFS_23 : T_Coeff (0 to pcMAX_COEFFS-1) :=
5  (
6      -0.0153162841603391,
7      0.0,
8      0.0233100755782965,
9      0.0,
10     -0.035236221972497,
11     0.0,
12     0.0553358536029083,
13     0.0,
14     -0.0992955153387857,
15     0.0,
16     0.308859092852497,
17     0.487336810134295,
18     0.308859092852497,
19     0.0,
20     -0.0992955153387857,
21     0.0,
22     0.0553358536029083,
23     0.0,
24     -0.035236221972497,
25     0.0,
26     0.0233100755782965,
27     0.0,
28     -0.0153162841603391,
29     others => 0.0
30 );
```

10.1. Realisierungen

Beide Implementierungen unterscheiden sich nur in den Multiplizierern. Bild 10.2 beinhaltet die Implementierung des FIR-Dezimationsfilters mit Hardwaremultiplizierern (Quellcode: Listing D.1, Seite 161) und Bild 10.3 die Implementierung des FIR-Dezimationsfilters mit vierschrittigen Booth-Multiplizierern (Quellcode: Listing D.2, Seite 164).

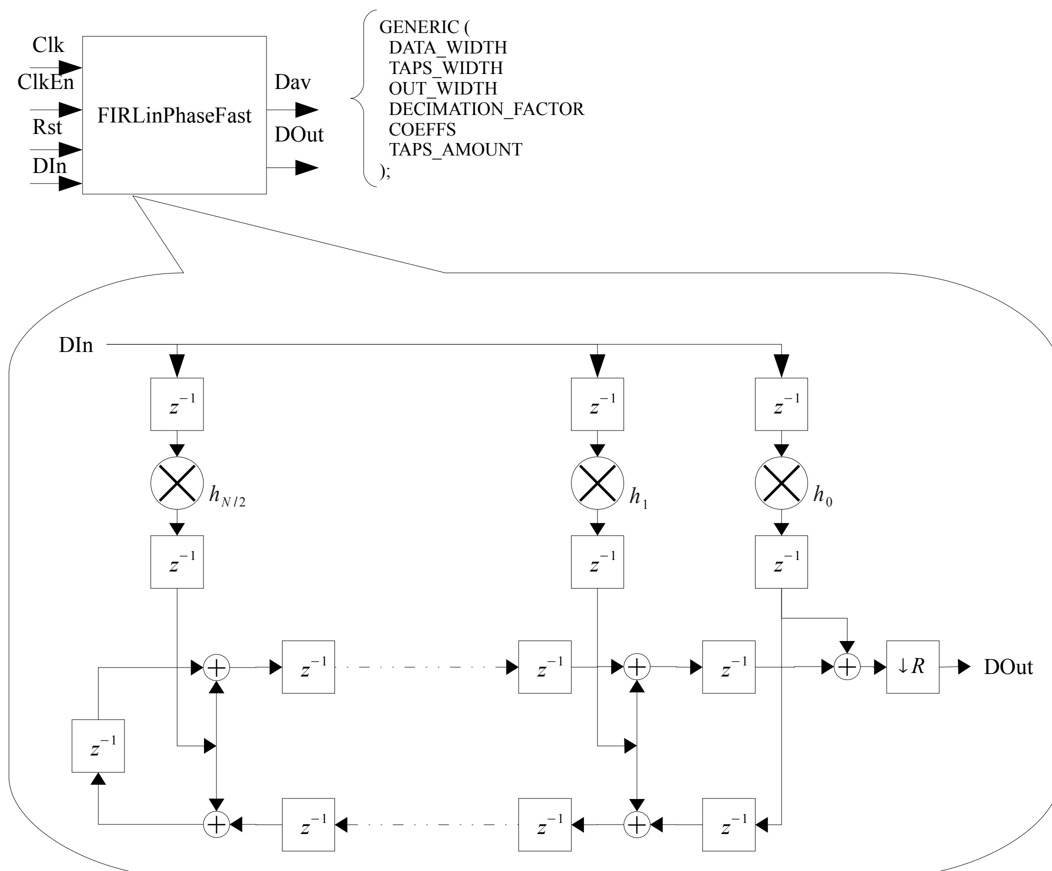


Bild 10.2.: Implementierung des FIR-Dezimationsfilters mit Hardwaremultiplizierern

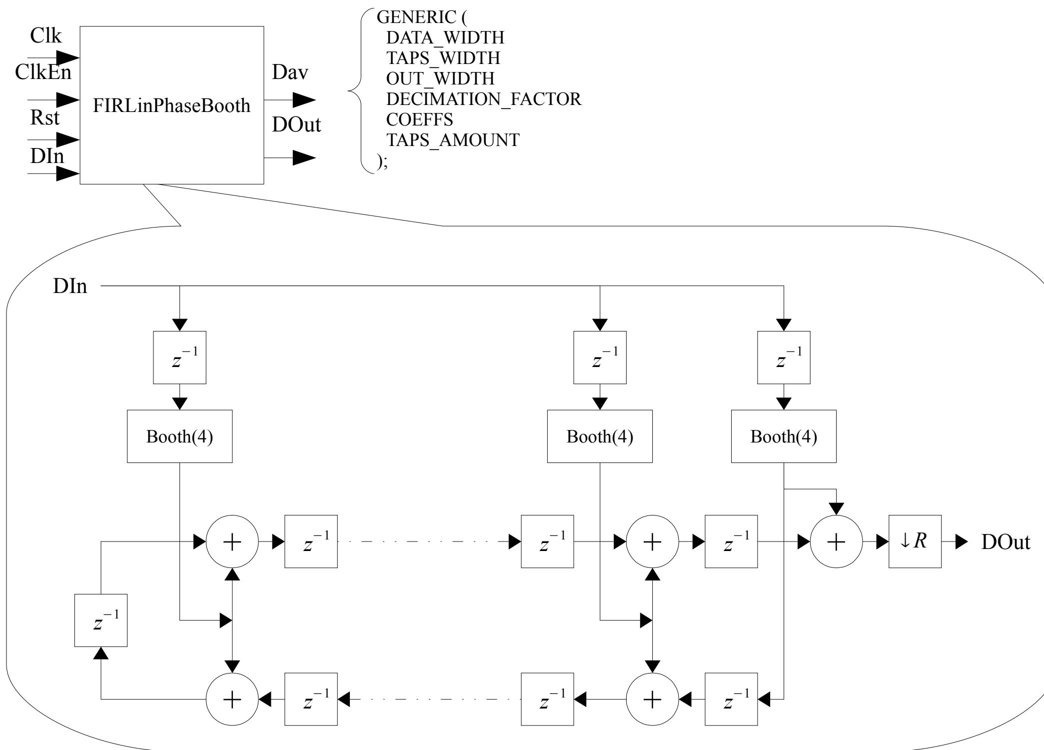


Bild 10.3.: Implementierung des FIR-Dezimationsfilters mit vierschrittigen Booth-Multiplizierern

10.2. Gegenüberstellung der Syntheseergebnisse

In Tabelle 10.1 werden einige Syntheseergebnisse für FIR-Dezimationsfilter gegenübergestellt, die mit Quartus 7.2 für einen Stratix III EP3SE50F484C2 FPGA erstellt wurden.

Zum einen wurden FIR-Dezimationsfilter synthetisiert, die mit Halfbandfilter-Koeffizienten verwendet wurden. Es wurden 7, 15 und 83 Koeffizienten verwendet. Desweiteren wurde ein FIR-Dezimationsfilter für Dezimationsfaktoren größer als 2 synthetisiert, wobei die Synthese für Dezimationsfaktoren größer als 8 aufgrund von Platzmangel des FPGAs unmöglich war. Die Koeffizienten sind so gewählt, dass jeweils 90% des Ausgangsspektrum nicht mit Aliasing-Effekten belastet sind. Es werden jeweils Syntheseergebnisse für FIR-Filter mit Hardware-Multiplizierern, Booth-Multiplizierern und seriellen Multiplierern vorgestellt.

Die Ergebnisse sind die Anzahl der benötigten „Combinatorial ALUTs“, die Anzahl der „Dedicated Logicregisters“ und die maximale Taktfrequenz.

Halfbandfilterkoeffizienten					
Bit	Koeffizientenanz.	Comb. Aluts	Register	F_{max}/MHz	Multiplizierer
FIR mit Hardware-Multiplizierern					
16	7	78	172	387,90	4
	15	301	314	309,12	16
	83	3432	6353	220,75	36
20	7	94	212	317,86	8
	15	357	378	306,56	32
	83	6488	7103	194,63	84
FIR mit Booth-Multiplizierern					
16	7	718	303	248,31	0
	15	1965	719	228,21	0
	83	5123	7529	225,48	0
20	7	758	373	234,36	0
	15	1983	876	224,72	0
	83	8962	8777	208,77	0
FIR mit seriellen Multiplizierern					
16	7	240	319	424,71	0
	15	716	737	343,05	0
	83	4364	7553	227,27	0
20	7	295	394	399,68	0
	15	832	905	334,68	0
	83	7888	8669	204,54	0
Halfbandfilterkoeffizienten und 20 Bit Auflösung					
R	Koeffizientenanz.	Comb. Aluts	Register	F_{max}/MHz	Multiplizierer
2	75	6410	6787	206,36	140
4	147	17098	233336	136,54	204
8	291	77204	888846	-	436
16	581	-	-	-	-
32	1161	-	-	-	-

Tabelle 10.1.: Syntheseergebnisse verschiedener FIR-Filter für Halfbandkoeffizienten und Koeffizienten für direkte Dezimation

11. Implementierung eines FIR-Halfband-Dezimationsfilters in VHDL

Die Implementierungen des FIR-Halfband-Dezimationsfilters werden in Direktform vorgenommen, so dass bereits vor der Multiplikation eine Dezimation der Eingangssignale um Faktor Zwei stattfinden kann, um die Anzahl der Multiplizierer zu halbieren. Die Implementierung in Direktform wirft das Problem der Addition der Multiplikationsergebnisse vor allem in hohen Datenraten auf, da alle Teilergebnisse gleichzeitig aufsummiert werden müssen, um ein Ergebnis zu errechnen. Um die langen kombinatorischen Pfade aufzuspalten, kann eine sukzessive Addition in Baumstruktur realisiert werden. Hierdurch ist ein größerer Hardwareaufwand zu erwarten, wodurch allerdings erheblich höhere Taktraten erreicht werden können. Implementiert wurde ein Baum mit zweifachen Verzweigungen. Moderne FPGAs besitzen allerdings Addierer, die drei Eingänge besitzen, weshalb an dieser Stelle darauf hingewiesen werden soll, dass eine Baumstruktur mit dreifachen Verzweigungen ggf. eine Hardwareersparnis mit sich bringt, wenn der FPGA die Voraussetzungen erfüllt.

Um ein hardwareoptimiertes System realisieren zu können, wurden drei verschiedene Implementierungen realisiert, die für verschiedene Datenraten ausgelegt sind:

- Mit Hardwaremultiplizierern für die volle Datenrate
- Mit Booth-Multiplizierern für eine reduzierte Datenrate
- Mit seriellen Multiplizierern für niedrige Datenraten

Da sich alle Implementierungen größtenteils gleichen, wird an dieser Stelle auf die grafische Darstellung aller Implementierungen verzichtet und nur die Implementierung mit Hardwaremultiplizierern vorgestellt (Bild 11.1).

Die Implementierungen nutzen dieselben VHDL-Komponenten der anderen FIR-Filter, die bereits erwähnt wurden. Die Quelltexte befinden sich für die jeweiligen Implementierungen in:

- Hardwaremultiplizierer: Listing E.1, Seite 169
- Booth-Multiplizierer: befindet sich auf der CD
- Serielle Multiplizierer: Listing E.2, Seite 174

Um optimierte VHDL-Kompontenten zu realisieren, die in sehr niedrigen Datenraten arbeiten (d.h. es stehen viele Takte für die Berechnung eines Samples zur Verfügung), können die vorhandenen Komponenten durch verteilte Arithmetik verändert werden, so dass beispielsweise nur ein Booth-Multiplizierer für alle Multiplikationen verwendet wird.

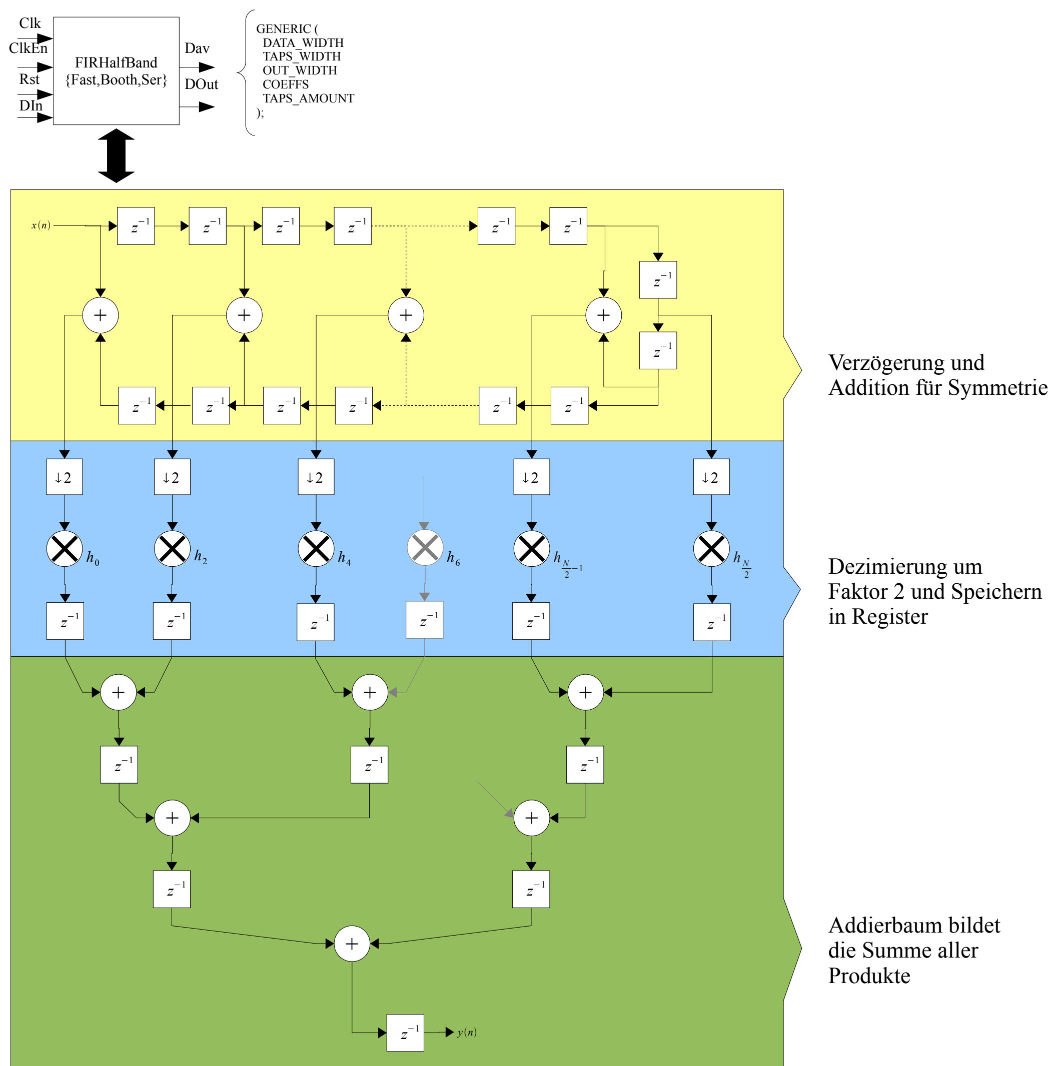


Bild 11.1.: Implementierung des FIR-Halfband-Dezimationsfilters

11.1. Gegenüberstellung der Syntheseergebnisse

In Tabelle 11.1 werden einige Syntheseergebnisse gegenübergestellt, die mit Quartus 7.2 für einen Stratix III EP3SE50F484C2 FPGA erstellt wurden.

Es wurden FIR-Halfbandfilter synthetisiert für die kaskadierbaren Koeffizientenanzahlen 7, 15 und 83. Es werden jeweils Syntheseergebnisse für FIR-Filter mit Hardware-Multiplizierern, Booth-Multiplizierern und seriellen Multiplizierern vorgestellt.

Die Ergebnisse sind die Anzahl der benötigten „Combinatorial ALUTs“, die Anzahl der „Dedicated Logicregisters“ und die maximale Taktfrequenz.

Bit	Koeffizientenanz.	Comb. Aluts	Register	F_{max}/MHz	Multiplizierer
FIR-HB mit Hardware-Multiplizierern					
16	7	114	331	351,74	4
	15	186	418	352,49	8
	83	1204	2812	266,10	36
20	7	138	310	287,52	8
	15	226	373	287,85	16
	83	1369	3490	228,41	84
FIR-HB mit Booth-Multiplizierern					
16	7	777	486	246,06	0
	15	1345	868	246,37	0
	83	2656	3926	226,76	0
20	7	838	606	251,38	0
	15	1544	1091	238,95	0
	83	3629	4857	202,92	0
FIR-HB mit seriellen Multiplizierern					
16	7	281	499	419,99	0
	15	482	886	402,25	0
	83	2107	4005	266,45	0
20	7	354	615	348,55	0
	15	594	1094	378,07	0
	83	2571	4939	238,95	0

Tabelle 11.1.: Syntheseergebnisse verschiedener FIR-Halfbandfilter mit unterschiedlichen Multiplikationsmethoden

12. Implementierung eines CIC-Dezimators in VHDL

Es wird ein statischer CIC-Dezimator in VHDL implementiert, dessen Parameter als *Generics* übergeben werden. Diese sind: M, N, R und die Wortbreite der Eingangs- und Ausgangsdaten.

Um dem Benutzer der CIC-Dezimator-Komponente Arbeit zu ersparen, werden alle Berechnungen für die Registerdimensionierung und die Auswahl der Bits für das Ausgangssignal während der Kompilierung automatisch vorgenommen. Somit genügt es, diesen Block zu parametrisieren, um eine funktionsfähige CIC-Dezimator-Komponente in VHDL nutzen zu können. Berechnungen während der Kompilierung können mit Hilfe von Konstanten bzw. Funktionen vorgenommen werden.

Die Komponente benötigt ein Taktsignal, das mindestens so schnell ist wie das der Abtastrate der Eingangsdaten. Intern wird ein One-Hot-Zähler instanziiert, der von 1 bis R zählt, um die Dezimation vorzunehmen. Der Vorteil der Verwendung eines One-Hot-Zählers ist der, dass in dem Zählregister jeweils ein Bit gesetzt ist und somit keine Dekodierlogik für die Abfrage des Zählerstands benötigt wird, wodurch sich höhere Taktraten erzielen lassen. Außerdem kann ein Bit eines Registers direkt als Enable-Signal eines Logikblocks in einem FPGA verwendet werden. Die Datenpfade aller weiteren

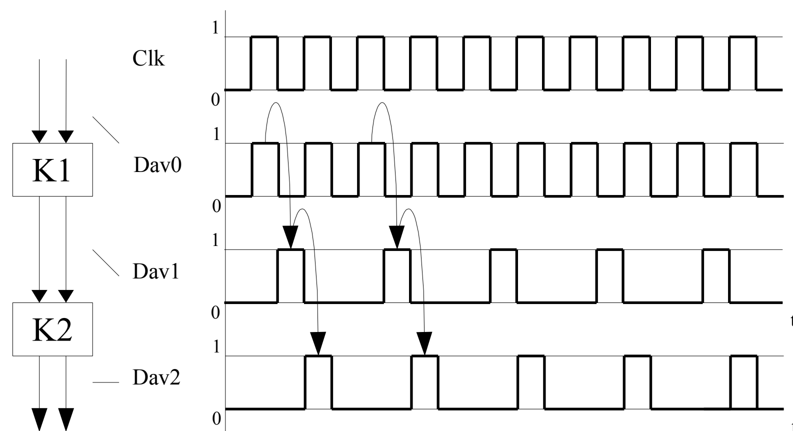


Bild 12.1.: Datenfluss zweier VHDL-Komponenten

Komponenten, die in dieser Diplomarbeit vorgestellt werden, signalisieren das Anliegen valider Daten mit Hilfe eines Data-Valid-Pulses, der in den nachfolgenden Komponenten als Enable-Signal für den Clock verwendet werden kann. Somit wird keine komplizierte Zählerlogik benötigt und keinerlei Synchronisation der Einzelkomponenten des DDCs. Bild 12.1 dient als Beispiel des Datenflusses.

Wie bereits im theoretischen Teil des CIC-Dezimators beschrieben wurde, bildet sich die Struktur dieses Dezimators aus simplen Einzelteilen (Integratoren und Differentiatoren), die ebenfalls einzeln in VHDL abgebildet werden können. Die CIC-Komponenten müssen bei der Instanziierung der Inte-

gratoren bzw. Differentiatoren lediglich für deren korrekte Parametrisierung und Zusammenschaltung Sorge tragen und die Ein- und Ausgangssignale richtig zuweisen. Der Zähler und die Integratoren werden mit dem Enable-Signal des Eingangs angesteuert. Um die Dezimation zu realisieren, wird der Enable-Eingang aller Differentiatoren mit einem Bit des One-Hot-Zählers angesteuert, welches auch als Data-Valid-Signal des Ausgangs verwendet wird.

12.1. Elemente

Implementierung eines Integrators

Ein Integrator besitzt die Differenzengleichung

$$y(n) = x(n) + y(n-1). \tag{12.1}$$

Bild 12.2 beinhaltet ein Blockschaltbild in Pipelinestruktur, das in VHDL implementiert werden kann.

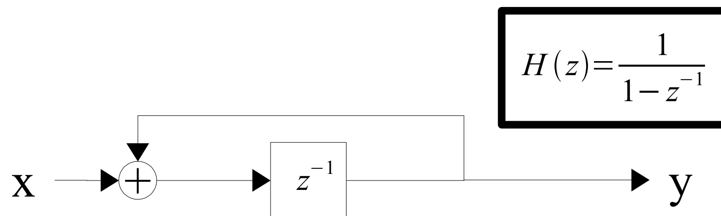


Bild 12.2.: Implementierung eines Integrators in Pipelinestruktur.
Diese Struktur ist für hohe Taktfrequenzen geeignet und erzeugt lediglich eine konstante Verzögerung der Signale.

Die VHDL-Implementierung erfolgt in einem einzigen taktsynchronen Prozess mit asynchronem Reset. Der Quellcode befindet sich in Listing F.2 im Anhang auf Seite 184.

Implementierung eines Differentiators

Ein Differentiator besitzt die Differenzengleichung

$$y(n) = x(n) - x(n-M). \tag{12.2}$$

Bild 12.3 beinhaltet ein Blockschaltbild in Pipelinestruktur, das in VHDL implementiert werden kann.

In einem taktsychronen Prozess mit asychronem Reset werden bei steigender Flanke des Clk-Signals das aktuelle Eingangssignal mit dem um M Takte verzögerten Eingangssignal addiert. Um schnelle Taktfrequenzen zu ermöglichen, wird das Ergebnis der Addition direkt in ein Register geschrieben, um kombinatorische Pfade kurz zu halten. Der Quellcode befindet sich in Listing F.3 im Anhang auf Seite 186.

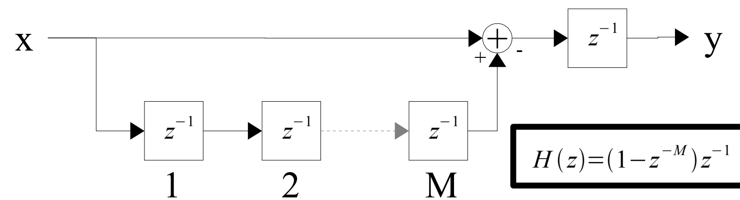


Bild 12.3.: Implementierung eines Differentiators in Pipelinestruktur.
Diese Struktur ist für hohe Taktfrequenzen geeignet. Das Eingangssignal wird um M Takte verzögert. Das Ergebnis wird jeweils in ein Ausgangsregister geschrieben.

12.2. Gesamtsystem

Bild 12.4 beinhaltet die schematische Darstellung der Implementierung des CIC-Filters.

Der Block *Resize* vergrößert das Einganssignal auf die interne Registerbreite. Dazu wird die gleichnamige VHDL-Funktion `resize()` verwendet. Der Block aus N Integratoren ist rot dargestellt, da dieser in der hohen Taktdomäne arbeitet. Der Differentiatorenblock ist grün dargestellt, da pro Berechnung mehr Takte zur Verfügung stehen als für die Integration. Die Differentiatoren arbeiten in einer um R reduzierten Taktdomäne. Der Quelltext der VHDL-Implementierung des Gesamtsystems befindet sich im Anhang in Listing F.4 auf Seite 188.

Aufgrund der mit hohem Aufwand verbundenen Berechnung der abschneidbaren Bits in einem CIC-Dezimator wird im Rahmen dieser Diplomarbeit auf eine Implementierung unter Berücksichtigung der Hardwareoptimierung verzichtet.

12.3. Simulation und Ergebnisse

Bild 12.5 beinhaltet vier Impulsantworten des dezimierten Ausgangssignal eines CIC-Dezimators.

Da anhand von Impulsantworten keine Rückschlüsse auf die korrekte Funktionsweise bzw. Aliasing-Anteile gezogen werden können, wurde anstelle eines Impulses ein Kamm-Signal verwendet.

Bild 12.6 zeigt die Kontur des Ausgangssignals eines CIC-Dezimators von Übertragungsfunktion und Aliaseffekten. Die Aliaseffekte werden durch die untere Kontur dargestellt und die Übertragungsfunktion von der Oberen. Anhand der Grafik wird deutlich, aus welchem Grund M in der Regel 1 oder 2 ist, da der Abfall der Amplitude im Bereich des Passbandes so wenig abnimmt bei etwa gleichbleibenden Aliasanteilen. Die Grafiken wurden erzeugt, indem ein Spektralkamm mit ausreichend vielen Linien als Eingangssignal des CIC-Dezimators verwendet wurde. Die Linienanzahl wurde so gewählt, dass keine Überlappungen derer im Ausgangssignal auftreten. Angezeigt werden anschließend nur die Spitzen der Linen, wodurch sich eine Kontur erkennen lässt.

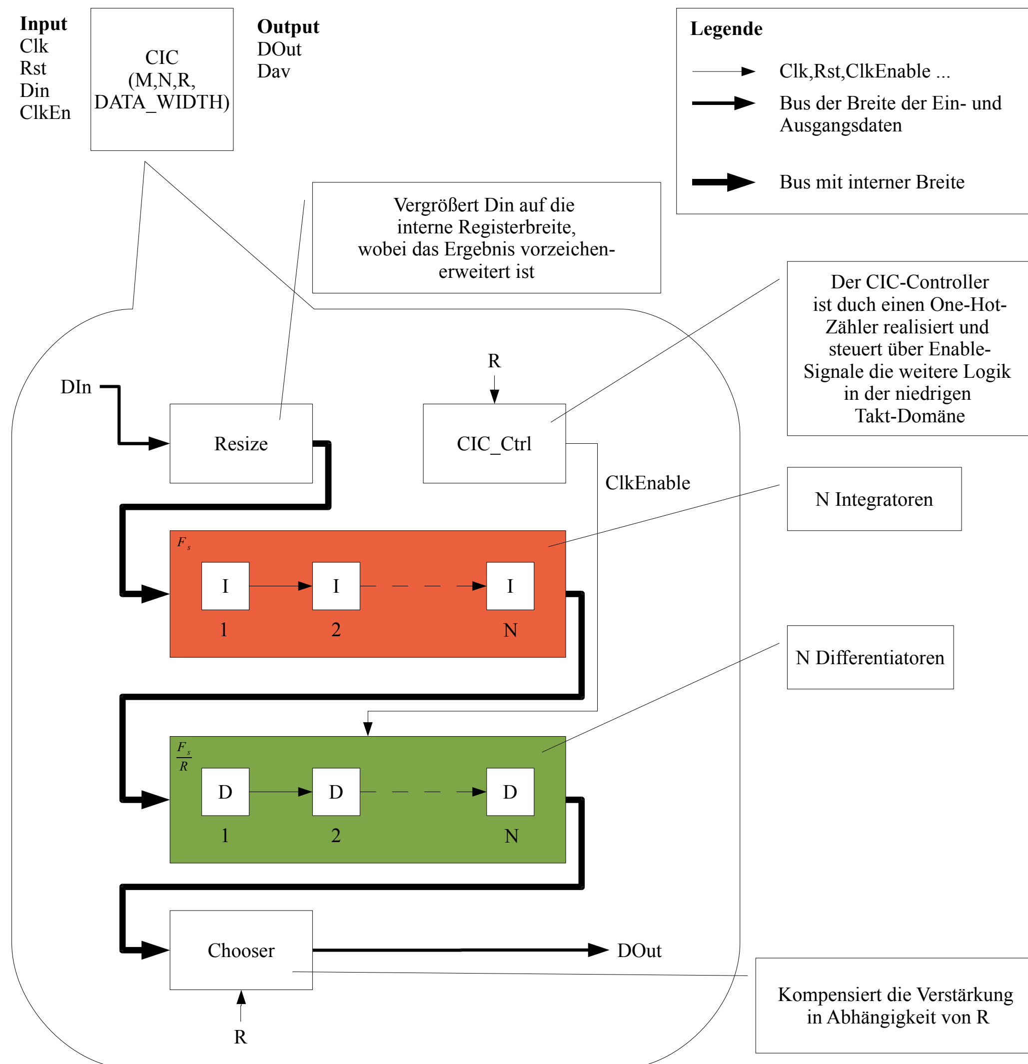


Bild 12.4.: Darstellung der Implementierung des CIC-Dezimators

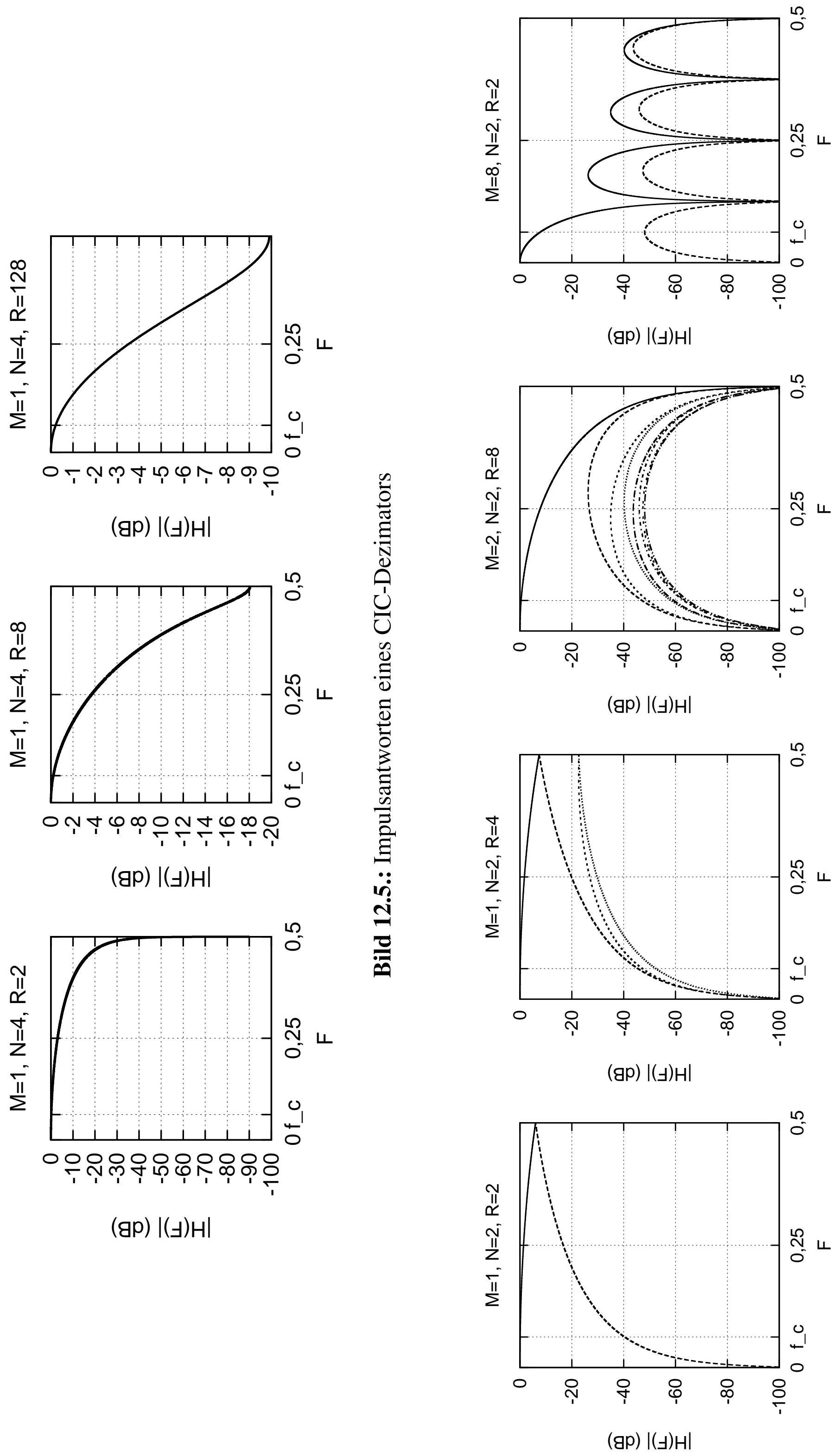


Bild 12.5.: Impulsantworten eines CIC-Dezimators

Bild 12.6.: Passbanddroop und Aliasing von CIC-Dezimatoren

12.4. Gegenüberstellung der Syntheseergebnisse

In Tabelle 12.1 werden einige Syntheseergebnisse gegenübergestellt, die mit Quartus 7.2 für einen Stratix III EP3SE50F484C2 FPGA erstellt wurden. Neben den drei Parametern M, N und R wirkt sich ebenfalls die Eingangs- und Ausgangsregisterbreite B auf die Synthese aus. Die Ergebnisse sind die Anzahl der benötigten „Combinatorial ALUTs“, die Anzahl der „Dedicated Logicregisters“ und die maximale Taktfrequenz.

R	Comb. Aluts	Register	F_{max}/MHz
16	276	427	373,27
128	372	671	338,98
256	404	843	305,16
1024	485	1699	293,77
8192	581	8999	260,82
65536	678	66475	- nicht synthetisierbar -

Tabelle 12.1.: Syntheseergebnisse eines CIC-Dezimators mit $N = 4$, $M = 1$ und 18 Bit Eingangsdaten von Quartus 7.2 für einen Stratix III FPGA

13. Implementierung eines SCIC-Dezimators in VHDL

Es wird ein statischer SCIC-Dezimator in VHDL implementiert, der als Pendant zu dem CIC-Dezimator in VHDL dient. Abgesehen von dem fehlenden Parameter M gleicht die Komponente dem CIC-Dezimator aus Sicht des Anwenders. Da ein SCIC-Dezimator auf denselben Elementen basiert wie der CIC-Dezimator, werden diese an dieser Stelle weiter verwendet.

Bild 13.1 zeigt das Blockschaltbild einer hardwareoptimierten SCIC-Implementierung.

Die Multiplikation mit dem Faktor Zwei im oberen Zweig wird durch eine Hartverdrahtung mit einem Bitshift um ein Bit nach links realisiert. Die Multiplikation im unteren Zweig mit dem Faktor $3R^N$ erfordert eine kompliziertere Schaltung. Der Dezimationsfaktor muss eine Potenz von Zwei sein, um die Multiplikation ebenfalls durch Bitshifts realisieren zu können. Um für die Multiplikation mit Drei keinen Multiplizierer zu verwenden, wird der Multiplikand mit sich selbst um ein Bit nach links geschoben addiert. Die Multiplikation kann in einem Takt realisiert werden und erzeugt keinen langen kombinatorischen Pfad. Der VHDL-Quelltext befindet sich in Anhang G in Listing G.2 auf Seite 193.

Bild 13.2 beinhaltet vier Impulsantworten des dezimierten Ausgangssignals eines SCIC-Dezimators.

Es ist zu erkennen, dass im Vergleich zu einem CIC-Dezimator das Passbanddroop deutlich geringer ausfällt und die Aliasrejection etwas höher ist.

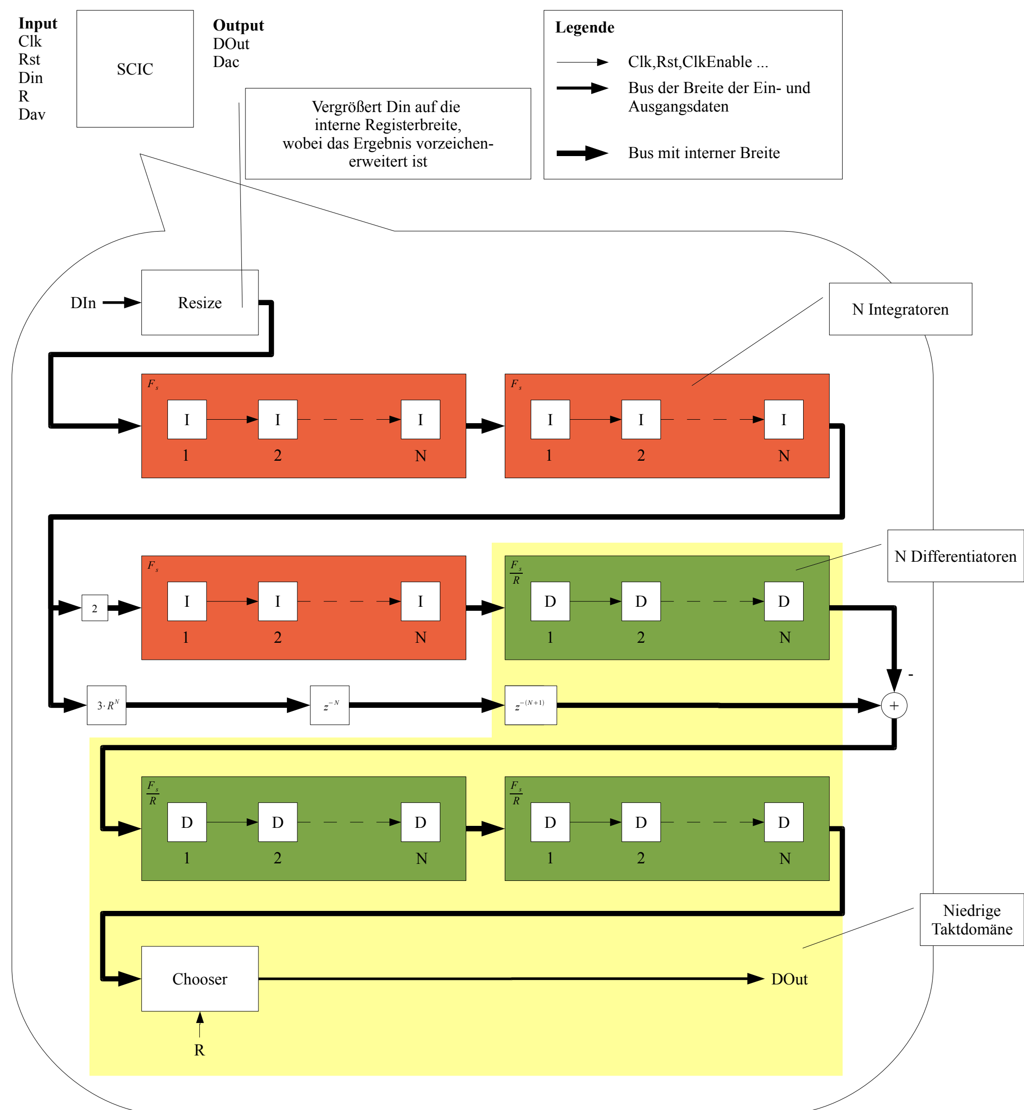


Bild 13.1.: Hardwarereduzierte SCIC-Implementierung

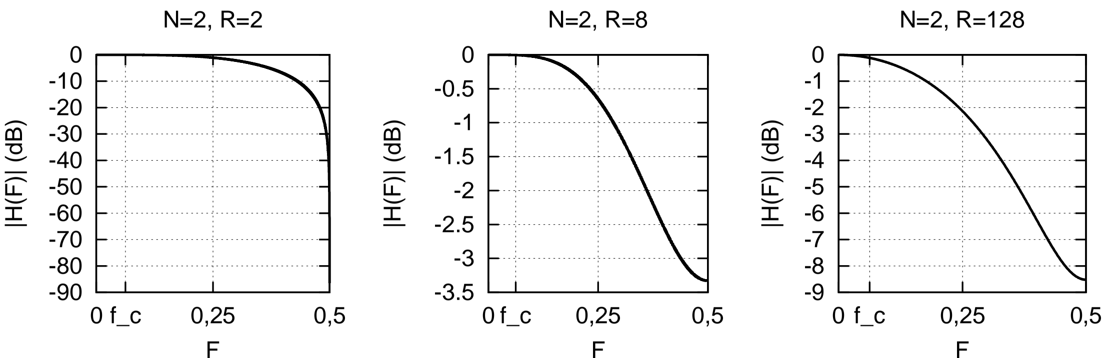


Bild 13.2.: Impulsantworten eine SCIC-Dezimators

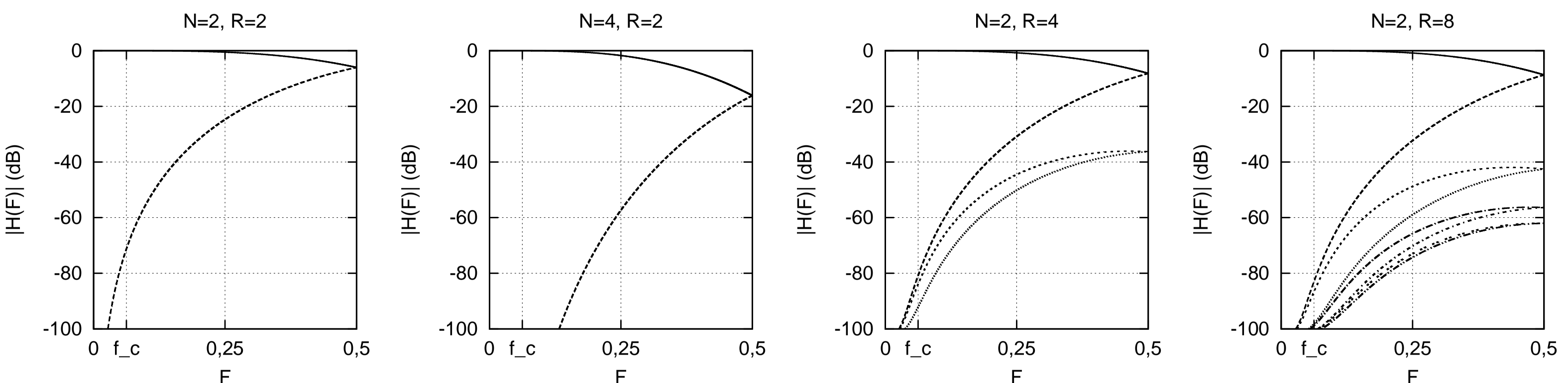


Bild 13.3.: Passbanddroop und Aliasing von SCIC-Dezimatoren in Abhängigkeit von R und N

13.1. Gegenüberstellung der Syntheseergebnisse und Simulationsergebnissen von CIC- und SCIC-Dezimatoren

In Tabelle 13.1 werden einige Syntheseergebnisse gegenübergestellt, die mit Quartus 7.2 für einen Stratix III EP3SE50F484C2 FPGA erstellt wurden. Neben den zwei Parametern N und R wirkt sich ebenfalls die Eingangs- und Ausgangsregisterbreite B auf die Synthese aus. Die Ergebnisse sind die Anzahl der benötigten „Combinatorial ALUTs“, die Anzahl der „Dedicated Logicregisters“ und die maximale Taktfrequenz.

R	Comb. Aluts	Register	F_{max}/MHz
16	582	1102	325,52
128	846	1646	270,12
256	928	1918	266,17
1024	1092	2974	236,41
8192	1338	10574	225,94
65536	1584	68350	- nicht synthetisierbar -

Tabelle 13.1.: Syntheseergebnisse eines SCIC-Dezimators mit $N = 4$ und 18 Bit Eingangsdaten von Quartus 7.2 für einen Stratix III FPGA

Im Folgenden werden CIC- und SCIC-Dezimatoren miteinander bezüglich Passbanddroop und Aliasrejection verglichen. Die Ergebnisse wurden mit Scilab errechnet und finden sich in tabellarischer Form in den Tabellen 13.2 und 13.3.

f_C = 1/8	N		
R	2	4	6
Passbanddroop (dB)			
CIC			
2	5.23e-03	1.57e-02	2.62e-02
3	6.20e-03	1.86e-02	3.10e-02
4	6.54e-03	1.96e-02	3.27e-02
5	6.70e-03	2.01e-02	3.35e-02
6	6.78e-03	2.03e-02	3.39e-02
7	6.83e-03	2.05e-02	3.42e-02
8	6.87e-03	2.06e-02	3.43e-02
9	6.89e-03	2.07e-02	3.45e-02
10	6.91e-03	2.07e-02	3.45e-02
SCIC			
2	2.39e-03	2.09e-02	5.62e-02
3	3.35e-03	2.90e-02	7.79e-02
4	3.72e-03	3.22e-02	8.62e-02
5	3.90e-03	3.37e-02	9.02e-02
6	4.00e-03	3.46e-02	9.24e-02
7	4.06e-03	3.51e-02	9.38e-02
8	4.09e-03	3.54e-02	9.46e-02
9	4.12e-03	3.56e-02	9.52e-02
10	4.14e-03	3.58e-02	9.57e-02
Aliasrejection (dB)			
CIC			
2	4.03e+01	1.21e+02	2.02e+02
3	4.43e+01	1.33e+02	2.21e+02
4	4.56e+01	1.37e+02	2.28e+02
5	4.61e+01	1.38e+02	2.31e+02
6	4.65e+01	1.39e+02	2.32e+02
7	4.66e+01	1.40e+02	2.33e+02
8	4.68e+01	1.40e+02	2.34e+02
9	4.68e+01	1.41e+02	2.34e+02
10	4.69e+01	1.41e+02	2.35e+02
SCIC			
2	7.12e+01	2.33e+02	3.94e+02
3	7.90e+01	2.56e+02	4.33e+02
4	8.16e+01	2.64e+02	4.46e+02
5	8.28e+01	2.67e+02	4.52e+02
6	8.34e+01	2.69e+02	4.55e+02
7	8.38e+01	2.70e+02	4.57e+02
8	8.40e+01	2.71e+02	4.58e+02
9	8.42e+01	2.72e+02	4.59e+02
10	8.43e+01	2.72e+02	4.59e+02

Tabelle 13.2.: Passbanddroop und Aliasrejection in dB eines CIC- bzw. SCIC-Dezimators in Abhängigkeit von R, M bei f_C = 1/8

f_C = 1/32	N		
R	2	4	6
Passbanddroop (dB)			
CIC			
2	5.23e-03	1.57e-02	2.62e-02
3	6.20e-03	1.86e-02	3.10e-02
4	6.54e-03	1.96e-02	3.27e-02
5	6.70e-03	2.01e-02	3.35e-02
6	6.78e-03	2.03e-02	3.39e-02
7	6.83e-03	2.05e-02	3.42e-02
8	6.87e-03	2.06e-02	3.43e-02
9	6.89e-03	2.07e-02	3.45e-02
10	6.91e-03	2.07e-02	3.45e-02
SCIC			
2	9.45e-06	8.49e-05	2.35e-04
3	1.33e-05	1.19e-04	3.30e-04
4	1.48e-05	1.32e-04	3.67e-04
5	1.55e-05	1.39e-04	3.85e-04
6	1.59e-05	1.42e-04	3.95e-04
7	1.61e-05	1.45e-04	4.01e-04
8	1.63e-05	1.46e-04	4.05e-04
9	1.64e-05	1.47e-04	4.07e-04
10	1.65e-05	1.48e-04	4.09e-04
Aliasrejection (dB)			
CIC			
2	6.44e+01	1.93e+02	3.22e+02
3	6.88e+01	2.06e+02	3.44e+02
4	7.02e+01	2.11e+02	3.51e+02
5	7.09e+01	2.13e+02	3.54e+02
6	7.12e+01	2.14e+02	3.56e+02
7	7.14e+01	2.14e+02	3.57e+02
8	7.15e+01	2.15e+02	3.58e+02
9	7.16e+01	2.15e+02	3.58e+02
10	7.17e+01	2.15e+02	3.59e+02
SCIC			
2	1.19e+02	3.77e+02	6.34e+02
3	1.28e+02	4.03e+02	6.78e+02
4	1.31e+02	4.12e+02	6.93e+02
5	1.32e+02	4.16e+02	6.99e+02
6	1.33e+02	4.18e+02	7.03e+02
7	1.33e+02	4.19e+02	7.05e+02
8	1.34e+02	4.20e+02	7.06e+02
9	1.34e+02	4.20e+02	7.07e+02
10	1.34e+02	4.21e+02	7.07e+02

Tabelle 13.3.: Passbanddroop und Aliasrejection in dB eines CIC- bzw. SCIC-Dezimators in Abhängigkeit von R, M bei f_C = 1/32

Auswertung

Auswertung

In den folgenden Kapiteln werden Simulationsprogramme für die implementierten Komponenten vorgestellt.

Anschließend werden sowohl die Simulationsergebnisse als auch die Syntheseergebnisse verschiedener Kombinationen der Komponenten miteinander vergleichend dargestellt, um eine Designentscheidung treffen zu können, auf welche Weise ein Digital-Down-Converter für gegebene Rahmenbedingungen optimal realisiert werden kann.

14. Simulation des Gesamtsystems mit C-Programmen

Um eine Simulation des Digital-Down-Converters zu ermöglichen, wurde ein modulares Systems mit verschiedenen C-Programmen erstellt. Jedes Programm übernimmt eine Teilfunktion eines Digital-Down-Converters, so dass es möglich ist, verschiedene DDC-Konstellationen zu simulieren und miteinander zu vergleichen.

Als Ein- und Ausgangsdaten können sowohl Text- als auch Wavedateien verwendet werden. Durch Letztere ist es möglich, Ergebnisse direkt zu „hören", um eine erste Aussage bezüglich der Qualität zu machen.

Die Programme sind Konsolenprogramme für das Linux-Betriebssystem. Sie können ebenfalls in Windows zusammen mit Cygwin verwendet werden, wobei mit großen Performance-Einbußen zu rechnen ist. Die Daten zwischen mehreren Programmen werden über Unix-Pipes übergeben. Somit ist gewährleistet, dass Mehrprozessorsysteme voll ausgenutzt werden, ohne dass viel Aufwand in Threadprogrammierung investiert wird. Auch die Handhabbarkeit wird dadurch erheblich gesteigert, weil Simulationen direkt aus der Konsole aufgerufen werden können.

14.1. Simulationsprogramme

Zwei Programme (*w2t*, *t2w*) stellen die Schnittstelle zwischen Wave- und Textdateien her. Das Handling der Wavedateien wird mit Hilfe von der frei verfügbaren C-Bibliothek *libaudiofile*[1] vorgenommen. Mit diesen Programmen ist es möglich, Wavedateien in Textdateien umzuwandeln und umgekehrt.

Alle weiteren Programme verarbeiten Textdateien bzw. Binärdateien- und -datenströme. Textdateien enthalten jeweils zwei Samples pro Zeile, die im `long long int`-Format gespeichert werden, so dass pro Zeile zwei Kanäle enthalten sind (Listing 14.1).

Der Oszillator wird durch das Programm *oscillator* realisiert. Es ist in der Lage, alle vorgestellten Oszillatoren zu simulieren und moduliert die Eingangsdaten auf die Signale auf, die anschließend ausgegeben werden. Das Programm wird mit der Modulationsfrequenz und dem Oszillatortyp parametrisiert.

Als Vordezimationsfilter wurden CIC- und SCIC-Dezimatoren in C realisiert (*cic*, *scic*). Deren Parametrisierung wird mit M (nur CIC), N und R vorgenommen, die den gleichnamigen Parametern der VHDL-Implementierungen entsprechen.

[1] http://www.68k.org/~michael/audiofile/

Listing 14.1: Beispiel einer Textdatei zur Speicherung von Signalen

```
0 0
512 512
2048 2048
5120 5120
10240 10240
15872 15872
20480 20480
22528 22528
20480 20480
15872 15872
10240 10240
5120 5120
2048 2048
512 512
0 0
```

Die Ausgabe der Vordezimation wird an ein weiteres Programm (*fir*) überreicht, das ein FIR-Dezimations-Filter simuliert, dessen Koeffizienten aus einer Textdatei gelesen werden können und dessen Dezimationsfaktor mit Hilfe des Parameters R übergeben wird. Die FIR-Koeffizienten werden in reellen Zahlen zeilenweise gespeichert. Die resultierenden Daten können anschließend mit anderer Software weiterverarbeitet werden oder zurück in eine Wavedatei umgewandelt werden.

Desweiteren wurden ein Interpolator, eine Statusanzeige, ein Programm zum Normalisieren und ein Programm zur Verstärkung der Signale um einen reellen Faktor realisiert.

Zum Beispiel kann ein DDC mit der Modulationsfrequenz 0.1 eines LUT-Oszillators, einem Dezimationsfaktor von 1024 und einer vierstufigen FIR-Halfbandkaskade mit dem Befehl in Listing 14.2 simuliert werden.

Listing 14.2: Unix-Kommandozeilenbefehl zur Simulation eines DDCs

```
cat impulse.txt                 \
| oscillator -w0.1 -trom -o     \
| scic -N2 -R 64 -io            \
| fir -c hb.coeff -R2 -io       \
| fir -c hb.coeff -R2 -io       \
| fir -c hb.coeff -R2 -io       \
| fir -c hb.coeff -R2 -i        \
> ddc.txt
```

Das Flag `-i` bzw. `-o` gibt dem jeweiligen Programm an, dass Eingangs- bzw. Ausgangsdaten in binärer Form verwendet werden.

Sämtliche Quelltexte der C-Programme inklusive dem Makefile und aller Headerdateien befinden sich auf der CD.

14.2. Funktionsnachweis des Gesamtsystems

Um zu beweisen, dass der im Rahmen dieser Diplomarbeit entwickelte Digital-Down-Converter funktioniert, wird ein Szenario erstellt. Die Simulation wird mit den, zu Beginn dieses Kapitels beschriebenen, Simulationsprogrammen durchgeführt. Dies ist zulässig, da sich die Entwicklung der VHDL-Komponenten an den Ergebnissen der jeweiligen Simulationsprogramme orientiert, so dass angenommen wird, dass die Ergebnisse der Hardwarerealisierung und der Simulation gleich sind.

Mit Scilab wurde ein Signal erzeugt, das aus den Frequenzen 400 Hz, 800 Hz und 1200 Hz und den jeweiligen Amplituden 1, 0,75 und 0,5 besteht. Dieses Signal wird auf einen Träger der Frequenz 40 MHz aufmoduliert, indem zu den jeweiligen Frequenzen 40 MHz addiert wird, aus denen das Eingangssignal erzeugt wird. Die Abtastfrequenz beträgt 96 MHz.

Das Signal soll um 40 MHz herunter konvertiert werden und gleichzeitig um den Faktor R=65536 dezimiert werden, so dass die Abtastrate des Ausgangssignal 2,9297 kHz beträgt.

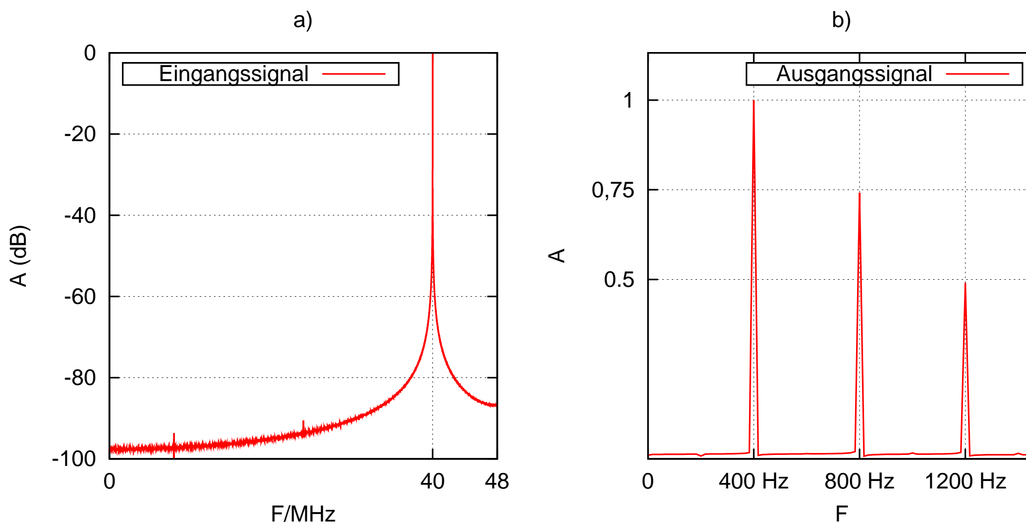


Bild 14.1.: Funktionsbeweis des Digital-Down-Converters

Die Spektren der Ein- und Ausgangsdaten werden in Bild 14.1 abgebildet. Da die Amplituden denen des Eingangsspektrums entsprechen (s.o.) und das Verhältnis beider Abtastraten richtig ist, ist die korrekte Funktion des erstellten Digital-Down-Converters hiermit bewiesen. Es muss noch angemerkt werden, dass in Bild 14.1a nur die positiven Frequenzen des reellen Eingangsspektrums mit logarithmischem Maßstab dargestellt werden. In Bild 14.1b wird das gesamte komplexe Spektrum mit linearem Maßstab des Ausgangssignals abgebildet.

15. Vergleich verschiedener DDC-Systeme

Ein DDC lässt sich in zwei Teile aufspalten, den Modulations- und den Filter-und-Dezimations-Teil. Der Modulationsteil besteht aus dem Oszillator, wobei für den Vergleich der Lookuptable-Oszillator und der adaptive IIR-Oszillator in Betracht gezogen werden, weil die erwarteten Ergebnisse der anderen Oszillatoren nicht den Anforderungen genügen.

Der Filter-und-Dezimations-Teil filtert das modulierte Signal des Modulationsteils und führt eine Dezimation durch. Die Filterung und Dezimation kann auf verschiedene Weisen realisiert werden und bietet das größte Einsparungspotenzial bezüglich des Hardwareaufwands (Bild 15.1).

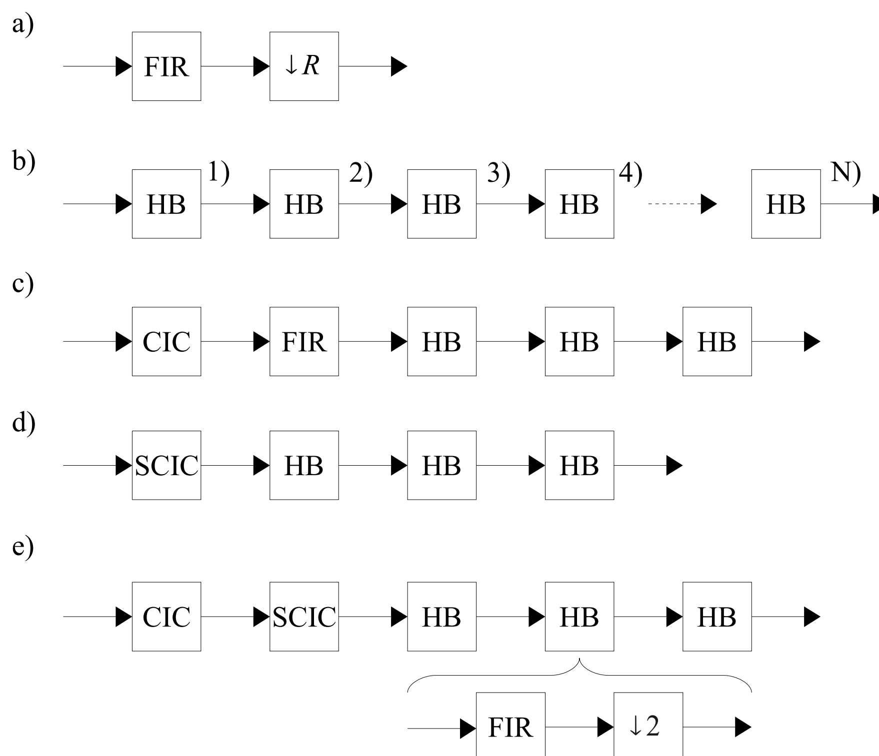


Bild 15.1.: Realisierungsmöglichkeiten der Filterung und Dezimation eines DDCs

Eine naheliegende Lösung, eine Tiefpassfilterung mit anschließender Dezimation zu realisieren, ist ein FIR-Filter, das von einem Dezimator gefolgt wird (Bild 15.1a). Das FIR-Filter arbeitet mit einer hohen Datenrate, muss ein sehr schmales Passband und ein steiles Transitionsband aufweisen, wodurch eine große Koeffizientenanzahl resultiert. Der Vorteil ist, dass sich so beliebige, ganzzahlige Dezimationsfaktoren realisieren lassen.

Eine bessere Möglichkeit bietet die Verwendung einer FIR-Halfband-Kaskade (Bild 15.1b). Halfbandfilter weisen optimale Eigenschaften bezüglich Koeffizientenanzahl und Steilheit des Transitionsbereichs auf und führen eine stufenweise Dezimation um den Faktor Zwei durch, womit sich der Hardwareaufwand durch serielle Arithmetik stark verringern lässt. Aus einer Halfband-Kaskade lassen sich Dezimationsfaktoren, die eine Potenz von Zwei sind, realisieren - die Anzahl der Stufen ist $log_2(R)$. Um andere Dezimationsfaktoren zu realisieren, kann die Kaskade von einem „normalen“ FIR-Filter mit anschließender Dezimation gefolgt werden. Dieses Filter arbeitet in einer reduzierten Datenrate und ist ökonomisch sinnvoll. Weitere Dezimationsfaktoren können grundsätzlich in Primfaktoren zerlegt werden, so dass für jeden Faktor ein einzelnes FIR-Filter verwendet wird. Es ist sinnvoll, in einer Filterkaskade zunächst die größeren Faktoren zu realisieren, um die Datenrate möglichst früh zu verringern.

Der Hardwareaufwand lässt sich weiter reduzieren, indem der erste Teil der Halfband-Kaskade durch ein CIC- bzw. SCIC-Dezimator ersetzt wird (Bild 15.1c,d). Im Gegensatz zu der Halfband-Kaskade wächst der Hardwareaufwand dann kaum mit dem Dezimationsfaktor, da stets dieselbe Anzahl von Halfbandfiltern nach einem CIC- oder SCIC-Dezimator benötigt wird und sich der Dezimationsfaktor nur auf die Register beider Predezimationsfilter auswirkt. In der Regel wird ein CIC-Dezimator von drei bis vier Halfbandfiltern (oder mehr) gefolgt, wobei ggf. ein FIR-Equalizer verwendet werden muss, um den Passbanddroop auszugleichen. Aufgrund der verbesserten Eigenschaften des SCICs, genügen für diesen Fall in der Regel drei Halfbandfilter und auf den FIR-Equalizer kann verzichtet werden, da der Passbanddroop vergleichsweise gering ausfällt.

Da zu erwarten ist, dass die Register sowohl in CIC- als auch in SCIC-Dezimatoren für hohe Dezimationsfaktoren sehr groß werden, können ein CIC- und ein SCIC-Dezimator kombiniert werden (Bild 15.1e).

Um aus den Syntheseergebnissen eines Oszillators und einer Realisierung des Filter-und-Dezimationsteils auf den Hardwarewaufwand eines DDCs zu schließen, müssen der doppelte Hardwareaufwand der Filter und der des Oszillators addiert werden.

15.1. Vergleich der Dezimationsfilter anhand der Syntheseergebnisse

Den größten Anteil bezüglich der Chipauslastung eines Digital-Down-Converters bilden die Dezimationsfilter. Darum ist es sinnvoll, deren Hardwareaufwand für verschiedene Dezimationsfaktoren miteinander zu vergleichen. Die Syntheseergebnisse eines Stratix III FPGAs sind die Anzahl der benötigten Lookuptables (Combinatorial ALUTs) und die Anzahl der benötigten Register (Dedicated Logicregisters). Weitere Informationen bezüglich der FPGA-Architektur befinden sich in [ALT1].

Die dargestellten Ergebnisse basieren auf den folgenden Komponenten:

- Halfbandfilterkaskaden: (N-2)-mal 7, 15 und 83 Koeffizienten, wobei N=log2(R) darstellt. Die Rechengenauigkeit beträgt 18 Bit
- CIC-Dezimator: M=1, N=4 und 18 Bit Eingangsdaten gefolgt von 4 Halfbandfiltern
- SCIC-Dezimator: N=2 und 18 Bit Eingangsdaten gefolgt von 3 Halfbandfiltern

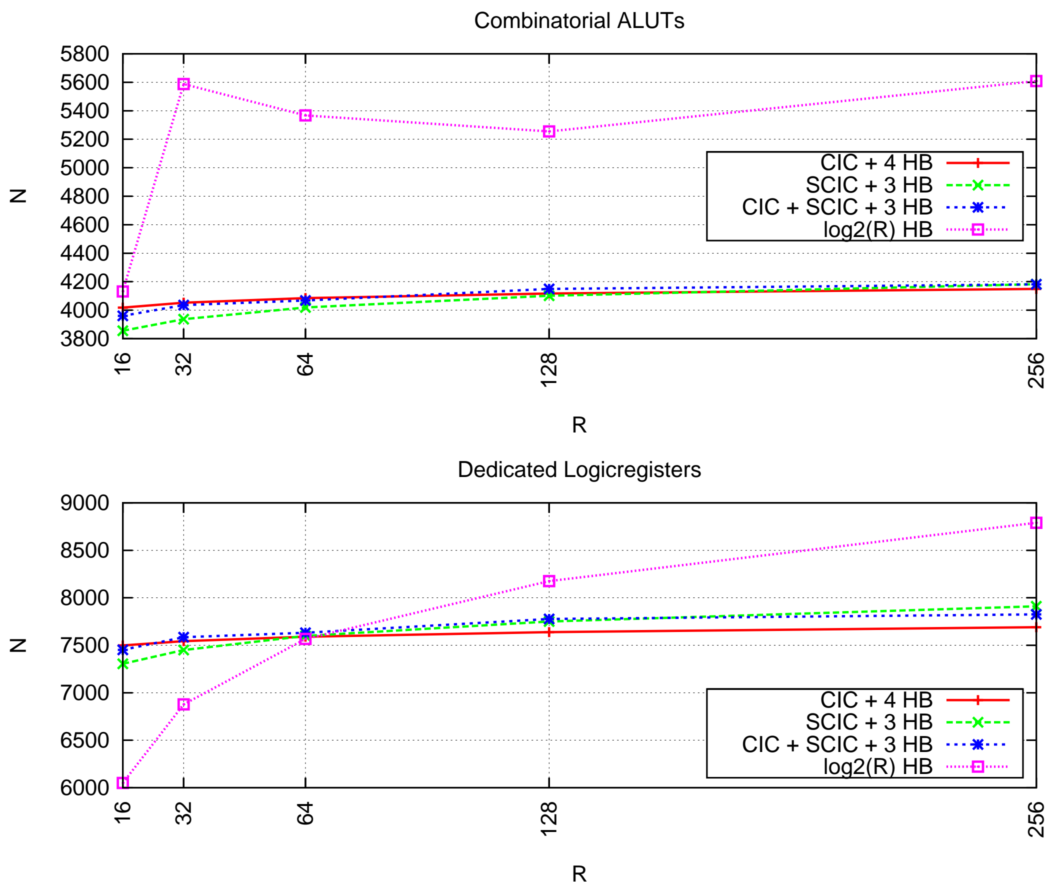


Bild 15.2.: Vergleich von Dezimationsfiltern für R=16 . . . 256

Im Bild 15.2 wird der Bereich von R=16 . . . 256 dargestellt. Die Anzahl der Combinatorial ALUTs und Dedicated Logicregisters ist für alle Dezimationsfilter außer der Halfbandfilterkaskade etwa gleich

groß. Für Dezimationsfaktoren R=16 ... 128 ist ein SCIC mit drei Halfbandfiltern optimal, während bei R=256 bereits ein CIC-Dezimator, der von vier Halfbandfiltern gefolgt wird, den geringsten Hardwareaufwand aller Dezimationsfilter aufweist. Für Dezimationsfaktoren unter 16 stellt eine Kaskade aus Halfbandfiltern die optimale Lösung dar. Der Verlauf der Comb. ALUTs der Halfbandfilterkaskade, ergibt sich dadurch, dass ein Filter mit Booth-Multiplizierern einen mit Hardwaremultiplizierern ablöst, so dass der Hardwareaufwand zunächst ansteigt. In den Grafiken wird die Nutzung von Hardwaremultiplizierern nicht berücksichtigt.

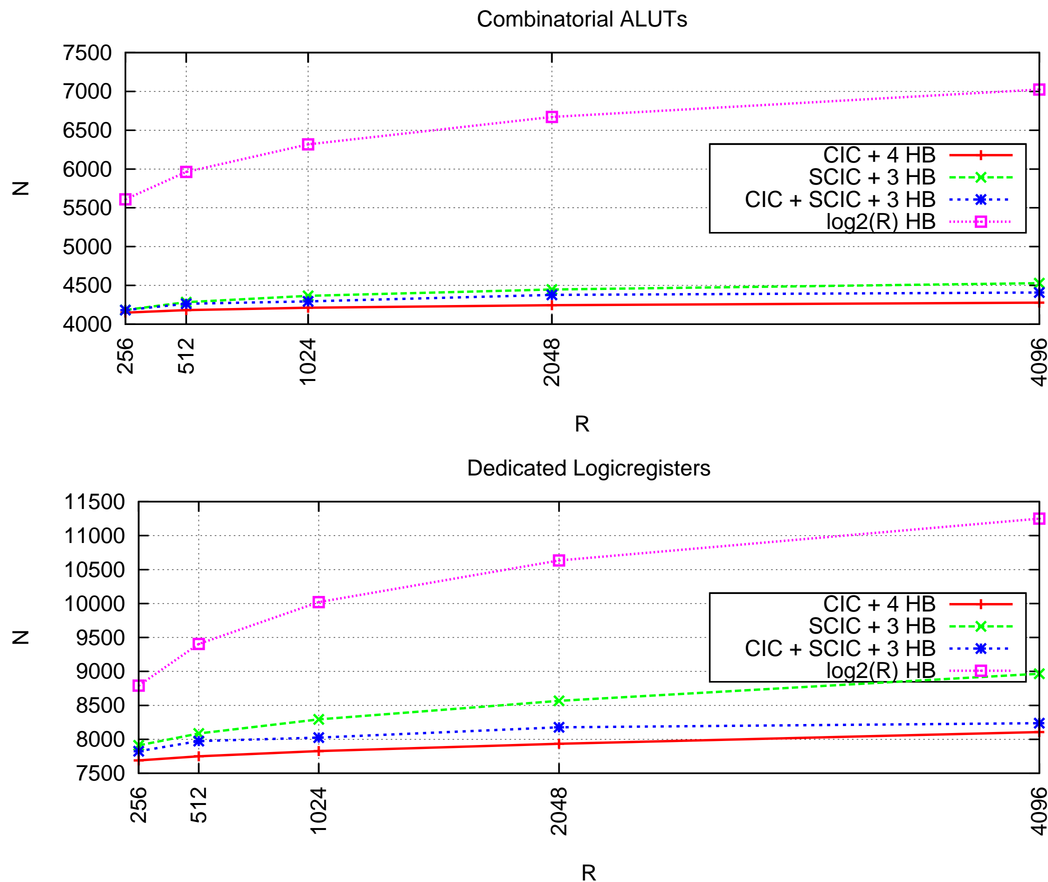


Bild 15.3.: Vergleich von Dezimationsfiltern für R=256 ... 4096

Für alle Dezimationsfaktoren zwischen 256 und 4096 (Bild 15.3) ist ein CIC-Dezimator mit vier Halfbandfiltern die optimale Lösung. Der Hardwareaufwand bezüglich Combinatorial ALUTs der Halfbandfilterkaskade ist etwa um die Hälfte größer als der der anderen Dezimationsfilter.

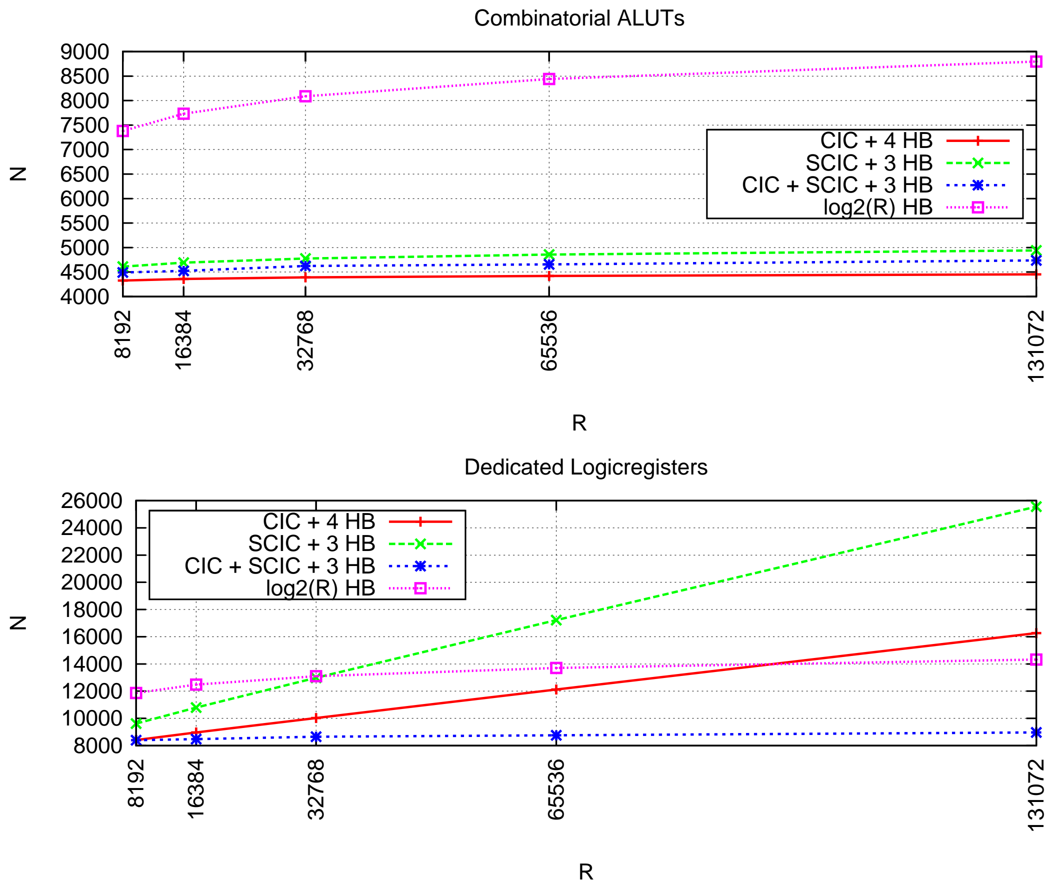


Bild 15.4.: Vergleich von Dezimationsfiltern für R=8192 . . . 131072

In dem Bereich der Dezimationsfaktoren 8192 . . . 131072, die in Bild 15.4 dargestellt werden, ist eine Kaskade aus einem CIC-Dezimator, einem SCIC-Dezimator und drei Halfbandfiltern optimal einzusetzen. Der Bedarf an Combinatorial ALUTs der Halfbandfilterkaskade ist in dem Bereich etwa doppelt so hoch als der der übrigen Ansätze.

Die Anzahl der benötigten Register von CIC- und SCIC-Dezimatoren steigen im Bereich der Dezimationsfaktoren etwa linear an. Der Verlauf ist durch die Verwendung des One-Hot-Zählers zu erklären, der für große Dezimationsfaktoren bereits R Register benötigt, um lediglich die Dezimation zu steuern. Für große Dezimationsfaktoren muss somit eine andere Implementierung für CIC- und SCIC-Dezimatoren verwendet werden, so dass der One-Hot-Zähler durch einen binären Zähler mit Komparator ersetzt wird, wodurch die maximale Taktfrequenz sinkt.

15.2. Vergleich der Dezimationsfilter anhand der Impulsantwort

Im Folgenden werden die Impulsantworten der Dezimationsfilter für verschiedene Dezimationsfaktoren miteinander verglichen. Es werden jeweils die Impulsantworten der Dezimationsfilter für einen Bereich von Dezimationsfaktoren dargestellt, die optimal bezüglich der Syntheseergebnisse sind. Für alle Übertragungsfunktionen gilt die Vorschrift, dass das Passband mindestens 90% der Bandbreite des Ausgangssignals ausnutzen muss.

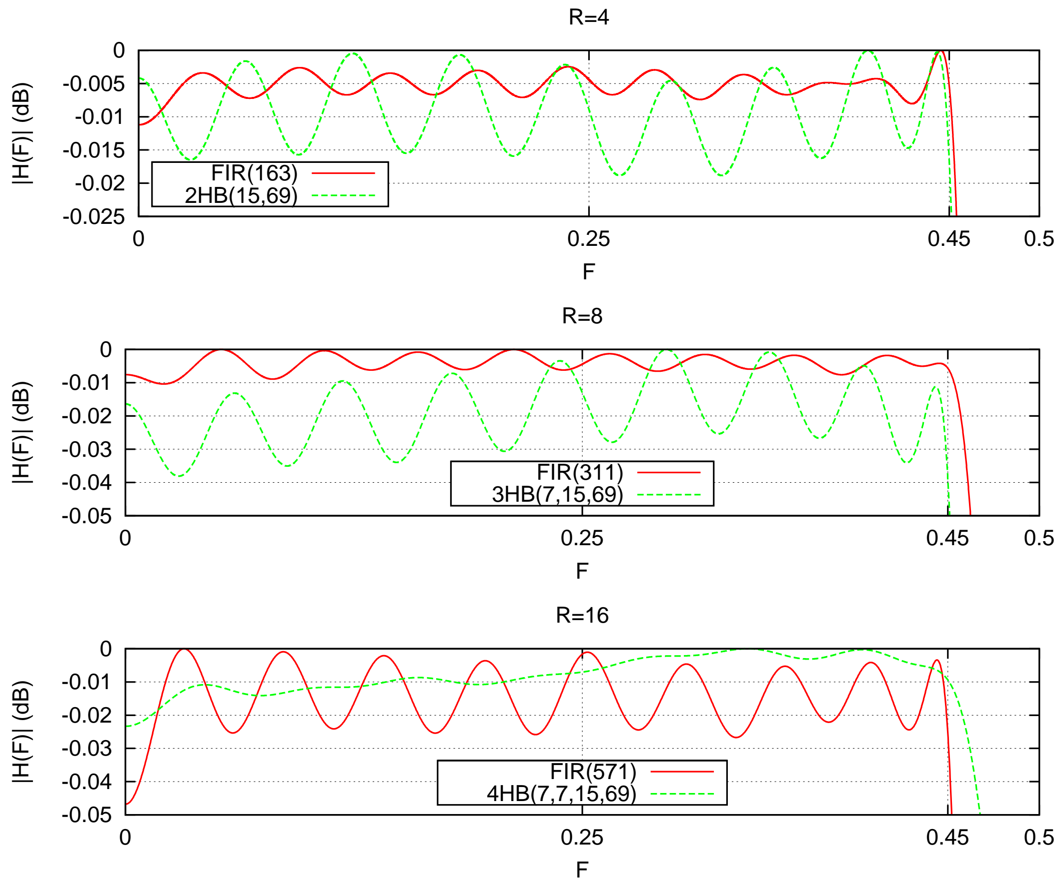


Bild 15.5.: Vergleich der Impulsantworten für FIR-Dezimationsfilter und Halfbandfilter-Kaskaden für kleine Dezimationsfaktoren

In Bild 15.5 werden die Impulsantworten von FIR-Dezimationsfiltern und FIR-Halfbandfilterkaskaden dargestellt. Die Koeffizientenanzahl des FIR-Dezimationsfilters für die Dezimationsfaktoren 4, 8 und 16 sind 163, 311 bzw. 571. Bereits durch die lediglich Betrachtung der Koeffizientenanzahl wird deutlich, welchen Vorteil die Kaskadierung mehrerer Filter mit sich bringt. Die Dezimation um Faktor 16 wird in dem Beispiel mit insgesamt 108 Koeffizienten realisiert, die sich auf vier Halfbandfilter aufteilen.

Für den Dezimationsfaktor 4 ist das FIR-Dezimationsfilter als Kaskade aus zwei Halfbandfiltern bezüglich des Ripples im Passband leicht überlegen, wobei die Koeffizientenanzahl des FIR-Filters mit 163 gegenüber 84 (Summe der Koeffizientenanzahl beider Halfbandfilter) beinahe doppelt so groß ist.

Für R=8 gilt dieselbe Aussage wie zuvor, wobei ein Ripple im Passband von 0.035 dB in der Regel innerhalb der Toleranz liegt. Ebenfalls ist die Koeffizientenanzahl des FIR-Filters mit 311 gegenüber 101 dreimal so groß.

Für den Dezimationsfaktor 16 ist das Ripple des FIR-Filters größer als das der Filterkaskade. Dies ist darauf zurückzuführen, dass 571 Koeffizienten in jeder Multiplikation innerhalb des FIR-Filters eine sehr geringe Ausnutzung der Registerdynamik zurfolge haben, so dass Quantisierungseffekte einen größeren Einfluss auf die Übertragungsfunktion haben als bei geringeren Koeffizientenanzahlen.

Die Anzahl der Koeffizienten des FIR-Filters ist mehr als viermal so groß wie die Summe der Koeffizienten der Halfbandfilterkaskade.

Grundsätzlich kann das Ripple einer Halfbandfilterkaskade verringert werden, indem die Koeffizientenanzahlen der einzelnen Filter vergrößert werden. Das Ripple, das in allen Bildern am meisten auffällt, wird durch das letzte Filter hervorgerufen. Durch eine Erhöhung der Koeffizientenanzahl der letzten Stufe, kann eine verbesserte Übertragungsfunktion bezüglich des Ripples erzielt werden.

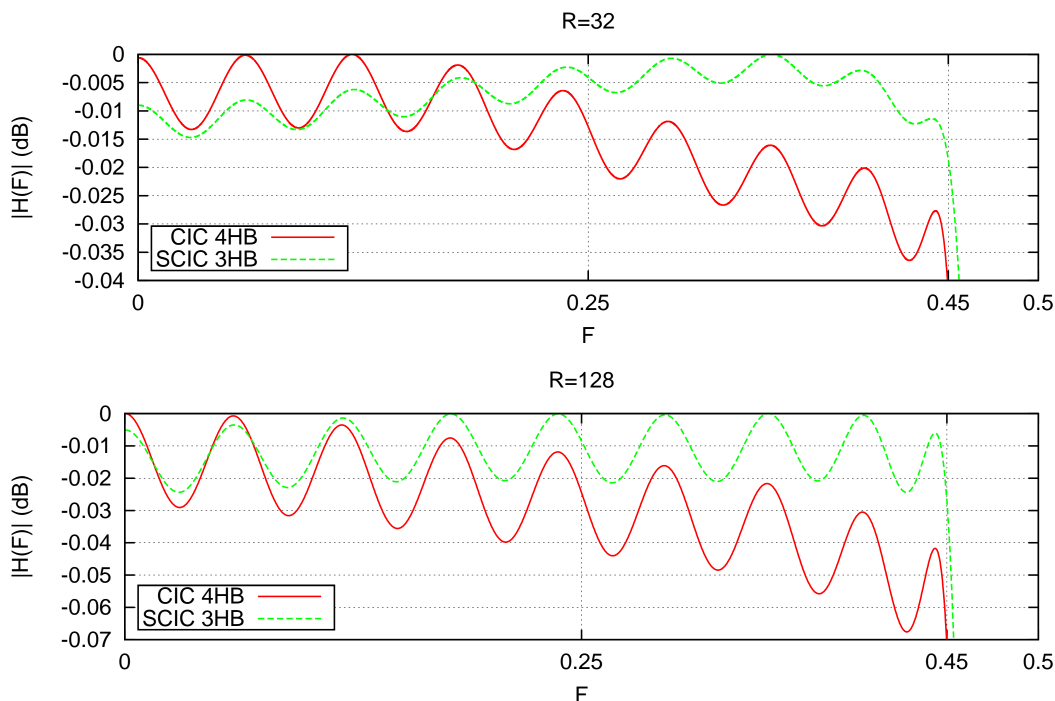


Bild 15.6.: Vergleich der Impulsantworten für Dezimationsfilter mit SCIC- und CIC-Dezimatoren und Halfbandfiltern für mittlere Dezimationsfaktoren

Für Dezimationsfaktoren zwischen 32 und 4096 hat sich ergeben, dass Dezimationsfilter, bestehend aus einem CIC- oder SCIC-Dezimator und vier bzw. drei Halfbandfiltern optimale Ergebnisse liefern. In Bild 15.6 werden Impulsantworten dieser Dezimationsfilter für die Dezimationsfaktoren 32 und 128 miteinander verglichen.

Wie erwartet ist, obwohl das Passband des CIC-Dezimators gegenüber dem Passband des SCIC-Dezimators halb so groß ist, der Passbanddroop des Dezimationsfilters mit SCIC-Dezimator erheblich geringer als das des anderen Filters. Ebenfalls fällt auf, dass die Übertragungsfunktion unter Verwendung eines CIC-Dezimators und vier Halfbandfiltern ein größeres Ripple aufweist. Der Verlauf wird

durch die vergrößerte Anzahl an Filterstufen hervorgerufen.

Obwohl eine Dezimation mit einem SCIC-Dezimator einen größeren Hardwareaufwand für den Dezimationsfaktor 128 hervorruft, ist dessen Übertragungsfunktion besser als die des Dezimationsfilters mit einem CIC-Dezimator.

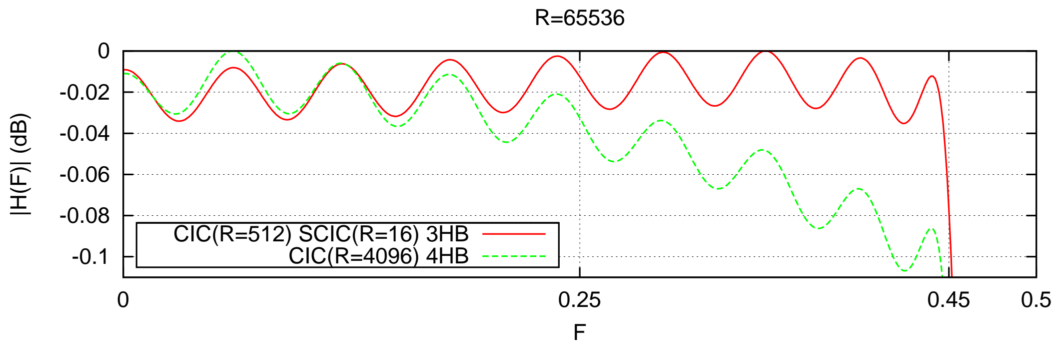


Bild 15.7.: Impulsantworten für Dezimationsfilter aus CIC- und SCIC-Dezimatoren und Halfbandfiltern

In Bild 15.7 werden die Impulsantworten eines CIC-Dezimators, der von vier Halfbandfiltern gefolgt wird, und eines CIC-Dezimators, der von einem SCIC-Dezimator und drei Halfbandfiltern gefolgt wird, für den Dezimationsfaktor 4096 miteinander verglichen. Der Dezimationsfaktor des CIC-Dezimators beträgt 512 und der des SCIC-Dezimators beträgt 16 für das zuletzt genannte Dezimationsfilter.

Der Passbanddroop des CIC-Dezimators mit Halfbandfiltern beträgt innerhalb des Passbands bis zu 0.11 dB, während der des anderen Dezimationsfilters lediglich 0.035 dB beträgt.

Zusammenfassend lässt sich feststellen, dass für große Dezimationsfaktoren eine Kaskade von CIC und SCIC und folgenden Halfbandfiltern sowohl bezüglich des Hardwarebedarfs als auch für die Qualität der Übertragungsfunktion bessere Ergebnisse liefert.

15.3. Übersicht der Chipauslastung verschiedener Digital-Down-Converter

Mit den gewonnenen Erkenntnissen können gesamte Digital-Down-Converter bezüglich der Chipauslastung miteinander verglichen werden. Eine Übersicht der Ergebnisse wird in Bild 15.8 dargestellt.

Abgesehen von dem Digital-Down-Converter, der ein FIR-Filter verwendet, das um den Faktor Vier dezimiert, passen alle verwendeten Digital-Down-Converter etwa zweimal in den kleinsten verfügbaren Stratix-III-FPGA hinein. Der größte verfügbare FPGA der Stratix-III-Familie der Firma Altera kann ungefähr zehn Digital-Down-Converter ohne zusätzliche Optimierungen beinhalten. Ein FIR-Filter, das um den Faktor Vier dezimiert, füllt bereits den größten Stratix-III alleine aus, so dass es nicht möglich ist, einen Digital-Down-Converter mit einem solchen FIR-Filter zu realisieren. Der Digital-Down-Converter mit zwei Halfbandfiltern erreicht eine Reduktion der Chipauslastung um 96,8% gegenüber der erstgenannten Variante. Da der Aufwand eines einzelnen FIR-Filters für höhere Dezimationsfaktoren exponentiell steigt, ist die Reduktion der Chipauslastung für höhere Dezimationsfaktoren entsprechend höher.

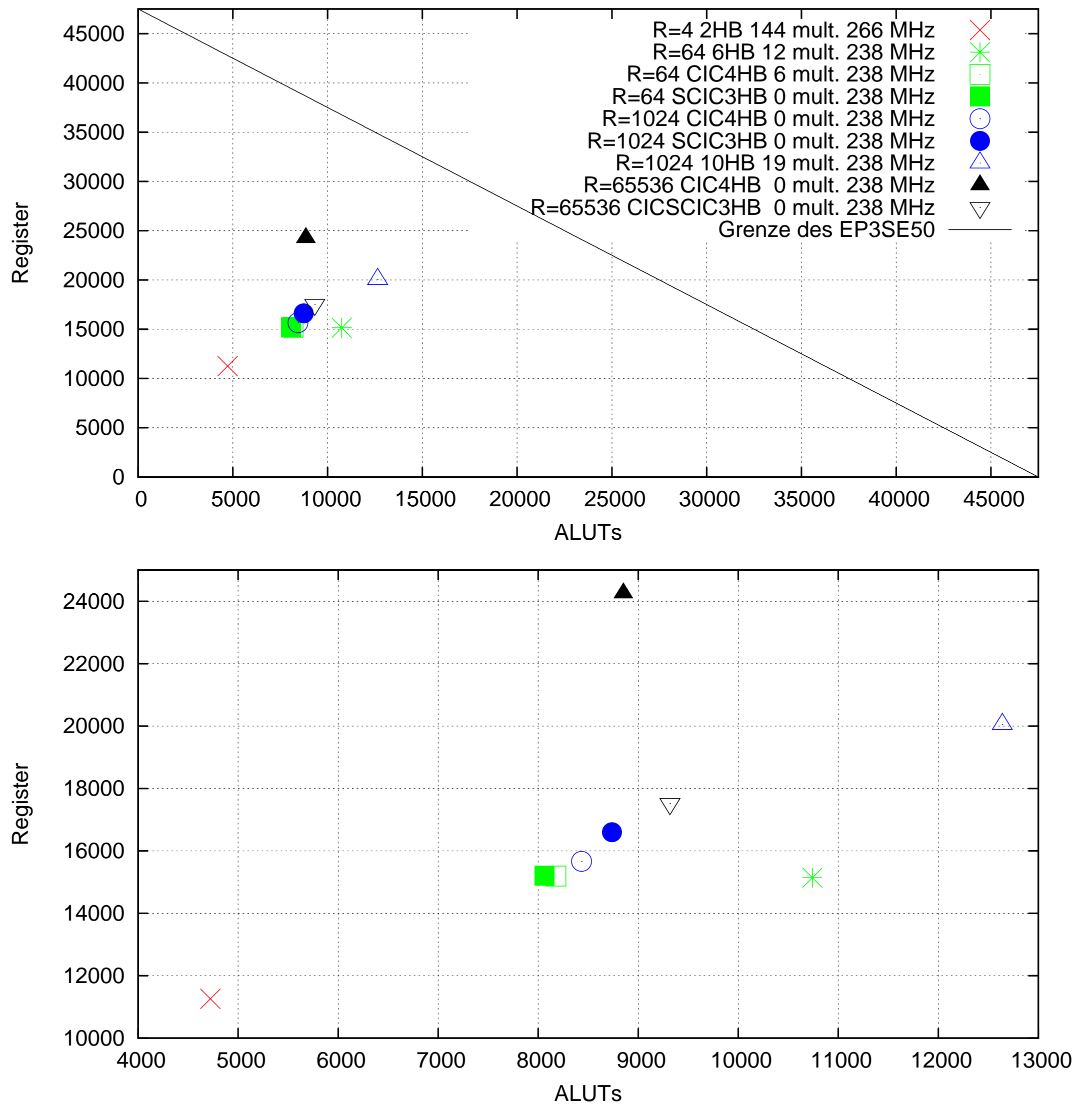


Bild 15.8.: Vergleich der Chipauslastung verschiedener Digital-Down-Converter

Die Ergebnisse machen deutlich, dass das Ziel der Diplomarbeit, einen hardwareoptimierten Digital-Down-Converter zu entwickeln, erfüllt werden.

16. Fazit

Zusammenfassung

Ein Digital-Down-Converter besteht aus drei Teilen: Oszillatoren, Tiefpassfilter und Dezimatoren. Diese Teile können jeweils auf verschiedene Weisen realisiert und miteinander verglichen werden. Wie gezeigt wurde, ist auch die Kombination von Filterung und Dezimation möglich und sinnvoll.

Zu Beginn der Diplomarbeit werden vier digitale Oszillatoren vorgestellt, die harmonische Schwingungen erzeugen.

Der Lookuptable-Oszillator ist einfach zu implementieren und kann in einer schnellen Schaltung realisiert werden. Für niedrige Frequenzen eignet sich dieser Oszillator weniger gut, da viele Funktionswerte gespeichert werden müssen. Trotz allem kann der Speicherplatz durch viele Optimierungsmöglichkeiten, die der Lookuptable-Oszillator aufweist, deutlich verringert werden, so dass dessen Vorteile wie Genauigkeit und Stabilität ggf. in einem Digital-Down-Converter genutzt werden können.

Der IIR-Oszillator ist ein rekursives digitales IIR-System in gekoppelter Form, das Sinus- und Cosinusschwingungen gleichzeitig erzeugt. Die Berechnung eines Funktionswerts beruht dabei jeweils auf den vorherigen Funktionswerten, so dass in einer Realisierung in endlicher Genauigkeit eine Fehlerfortpflanzung stattfindet, durch die der Oszillator instabil ist. Da für einen Digital-Down-Converter ein langzeitstabiler Oszillator benötigt wird, wird der IIR-Oszillator stabilisiert, indem eine regelmäßige Neuinitialisierung realisiert wird, so dass das Abklingen oder Aufschwingen der Amplituden kompensiert wird.

Eine Stabilisierung durch Mitzählen erzeugt einen Sprung in der Amplitude, der Oberschwingungen im Spektrum der Ausgangssignale erzeugt. Die Möglichkeit, eine Nulldurchgangsüberwachung der Ausgangssignale zu realisieren, erzeugt zwei Sprünge im Spektrum, die weniger ins Gewicht fallen und doppelt so häufig auftreten, so dass sich die Oberschwingungen im Spektrum voneinander entfernen.

Eine weitere Stabilisierungsmethode ist die Adaption zwischen verschiedenen Koeffizienten, so dass sich Aufschwingen und Abklingen der Amplituden gegenseitig ausgleichen. Durch das Ausgleichen werden Sprünge in den Schwingungen durch eine schwankende Amplitude ersetzt, wodurch ein hochwertigeres Spektrum verglichen mit den Spektren der anderen Stabilisierungsmethoden resultiert, das weniger Phasenrauschen und eine höher Dämpfung aufweist.

Grundsätzlich eignet sich der IIR-Oszillator nicht für alle Frequenzen gleich gut, da die Berechnungsgenauigkeit unter Anderem von der Frequenz abhängt. Um den Fehler aller drei Ansätze zu reduzieren, muss die Rechengenauigkeit des IIR-Oszillators erhöht werden.

Ein weiterer Oszillator, der auf der Taylorreihenentwicklung basiert, wird ebenfalls bezüglich dessen Implementierbarkeit untersucht. Ein Taylorreihen-Oszillator ist nicht in der Lage, zwei harmonische Schwingungen gleichzeitig zu erzeugen und benötigt eine sehr hohe Rechengenauigkeit. Auf der anderen Seite ist es möglich, mit einem Taylorreihen-Oszillator eine beliebig hohe Genauigkeit der Berechnungen zu erreichen.

Der CORDIC-Oszillator basiert auf Vektorrotation und erzeugt Sinus- und Cosinusschwingungen gleichzeitig. Er ist in der Lage, beliebig hohe Genauigkeit zu erzielen und ermöglicht eine Änderung der Ausgangsfrequenz zur Laufzeit, weshalb dessen Einsatzmöglichkeit in Erwägung gezogen wurde.

Miteinander verglichen, harmonieren der IIR-Oszillator und der Lookuptable-Oszillator mehr mit dem Ziel, ein hardwareoptimiertes System zu realisieren. Deshalb wird im Rahmen der Diplomarbeit auf die VHDL-Implementierungen des Taylorreihen-Oszillators und des CORDIC-Oszillators verzichtet.

Die Filterung und Dezimation wird zu FIR-Dezimationsfiltern zusammengefasst, die in zwei unterschiedlichen Realisierungen implementiert werden.

Die Realisierung der FIR-Dezimationsfilter erfolgt in der transponierten Form, so dass eine schnelle Schaltung realisiert wird. Die FIR-Dezimationsfilter nutzen die Symmetrie der Koeffizienten aus und sind für ganzzahlige Dezimationsfaktoren verwendbar.

Die FIR-Halfbandfilter werden in der Direktform realisiert, so dass eine Dezimation bereits vor der Multiplikation erfolgen kann. Dadurch kann eine Hälfte der Multiplizierer eingespart werden. Durch Ausnutzung symmetrischer und fehlender Koeffizienten, werden weitere Multiplizierer eingespart. Somit ergibt sich ein sehr kompaktes Filter, das eine Dezimation um Faktor Zwei vornimmt.

Durch die Kaskadierung mehrerer Halfbandfilter werden Dezimationsfaktoren realisiert, die einer Potenz von Zwei entsprechen. Innerhalb einer Filterkaskade verdoppelt sich die Anzahl der Takte pro Sample mit jeder Stufe, so dass ggf. eine Realisierung der Filter mit Booth-Multiplizierern oder seriellen Multiplizierern umsetzbar ist. Durch die Einsparung von Hardwaremultiplizierern wird eine sehr kompakte Schaltung realisiert, die den Ansprüchen einer VLSI-Implementierung genügt.

Bezüglich des Hardwareaufwands ist es günstiger, einen Dezimationsfaktor in Primfaktoren zu zerlegen, für jeden Faktor ein Filter zu realisieren und diese zu kaskadieren. Innerhalb einer Filterkaskade besitzen die ersten Filter in der Regel wesentlich weniger Koeffizienten als die letzten Filter. Mit dem bereits erwähnten Vorteil der seriellen Arithmetik ergibt sich im Vergleich zu einem FIR-Dezimationsfilter, das eine direkte Dezimation vornimmt, ein erheblich geringerer Hardwareaufwand. Zusätzlich kann die Berechnung für Filter mit kleinerer Koeffizientenanzahl mit geringerer Genauigkeit erfolgen als für Filter mit größerer Koeffizientenanzahl.

Für große Dezimationsfaktoren werden viele Filter mit geringem Dezimationsfaktor, wie zum Beispiel einer Kaskade aus Halfbandfiltern, oder eine geringe Filteranzahl mit jeweils hoher Koeffizientenanzahl und großem Dezimationsfaktor benötigt. Das führt in jedem Fall zu einem hohen Hardwarebedarf, der durch die Verwendung von Vordezimationsfiltern reduziert wird.

CIC- bzw. SCIC-Dezimatoren sind Dezimationsfilter, die keine Multiplikationen benötigen und mit äußerst geringem Hardwareaufwand hohe Dezimationsfaktoren erzielen. Da sowohl CIC- als auch SCIC-Dezimatoren eine Übertragungsfunktion aufweisen, aus der lediglich ein schmaler Teil weiterverwendet werden kann, müssen diese durch weitere Filter gefolgt werden. Der SCIC-Dezimator stellt eine Verbesserung des CIC-Dezimators bezüglich der Breite des nutzbaren Bereichs in der Übertragungsfunktion dar, wodurch ein Halfbandfilter gegenüber der Verwendung eines CIC-Dezimators eingespart werden kann. In der Regel wird ein SCIC-Dezimator von drei und ein CIC-Dezimator von vier Halfbandfiltern gefolgt.

Durch den Einsatz von CIC- bzw. SCIC-Dezimatoren ist der Hardwareaufwand eines gesamten Systems nur noch in geringem Maße von dem Dezimationsfaktor abhängig, da die Anzahl der Filter, die dem CIC- bzw. SCIC-Dezimator folgen, konstant ist und der Hardwareaufwand beider Vordezimationsfilter nur in geringem Maße von dem Dezimationsfaktor abhängt.

Die Diplomarbeit endet mit dem Vergleich verschiedener Dezimationsfilter bezüglich des Hardware-

aufwands und der Übertragungsfunktion für verschiedene Dezimationsfaktoren. Für Dezimationsfaktoren zwischen Zwei und 16 ist es sinnvoll, Halfbandfilterkaskaden zu verwenden. Dadurch resultiert eine optimale Nutzung der Hardware und es wird eine gute Übertragungscharakteristik erzielt.

Für größere Dezimationsfaktoren unter 8192 sollten CIC- bzw. SCIC-Dezimatoren mit Halfbandfilterkaskaden verwendet werden. Hierbei muss unterschieden werden, ob eine Optimierung bezüglich des Hardwareaufwands oder der Qualität der Übertragungsfunktion erfolgen soll. Die Kombination eines CIC-Dezimators und vier Halfbandfiltern erwirkt einen geringeren Hardwareaufwand als ein SCIC-Dezimator mit drei Halfbandfiltern, während die Übertragungsfunktion letzterer Kombination bessere Eigenschaften bezüglich des Ripples und des Passbanddroops aufweist.

Für sehr große Dezimationsfaktoren ist es sinnvoll, einen CIC-Dezimator einzusetzen, der von einem SCIC-Dezimator mit drei Halfbandfiltern gefolgt wird. Das System aus CIC-Dezimator und vier Halfbandfiltern weist für gleichgroße Dezimationsfaktoren sowohl einen erheblich höheren Hardwareaufwand als auch eine deutlich schlechtere Übertragungsfunktion auf.

Um den Hardwareaufwand zu minimieren, sollte ein Digital-Down-Converter aus einem adaptiven IIR-Oszillator und einem CIC-Dezimator mit vier Halfbandfiltern pro Verzweigung realisiert werden. Wenn primär eine qualitativ hochwertige Übertragungscharakteristik des Digital-Down-Converters erzielt werden soll, sollte dieser aus einem Lookuptable-Oszillator und einem SCIC-Dezimator mit drei Halfbandfiltern pro Verzweigung realisiert werden, wobei der Hardwareaufwand nur geringfügig höher ausfällt als bei der erstgenannten Variante.

Alle Implementierungen sind hersteller- und hardwareunabhängig, so dass sich viele Einsatzgebiete für den entwickelten Digital-Down-Converter ergeben.

Eine Übersicht, welcher Digital-Down-Converter für welche Dezimationsfaktoren zu einer optimalen Realisierung führt, wird in Tabelle 16.1 dargestellt.

Dezimationsfaktor	Optimale Realisierung eines Digital-Down-Converters
$2 \ldots 8$	LUT-Oszillator und Kaskade aus Halfband- und/oder FIR-Filtern
$8 \ldots 128$	LUT-Oszillator und SCIC-Dezimator mit drei Halfbandfiltern
$128 \ldots 8192$	LUT-Oszillator mit CIC-Dezimator und vier Halfbandfiltern
≥ 8192	LUT-Oszillator mit CIC-Dezimtor gefolgt von einem SCIC-Dezimator mit drei Halfbandfiltern

Tabelle 16.1.: Optimale Realisierung eines Digital-Down-Converters für einen gegebenen Dezimationsfaktor

Bewertung

Die im Rahmen dieser Diplomarbeit entwickelten und untersuchten Komponenten lassen den Vergleich verschiedener Realisierungsmöglichkeiten für Digital-Down-Converter zu. Anhand der Gegenüberstellung verschiedener Systeme wird deutlich, an welchen Stellen auf welche Weise der Hardwareaufwand reduziert werden kann, um einen hardwareoptimierten Digital-Down-Converter zu entwickeln, der trotzdem eine qualitativ hochwertige Übertragungscharakteristik aufweist.

Abgesehen von einigen weiteren Optimierungsmöglichkeiten, die im folgenden Abschnitt beschrieben werden, wurde das Ziel der Diplomarbeit erreicht. Mit Hilfe der Ergebnisse kann eine Designentscheidung getroffen werden, auf welche Weise ein Digital-Down-Converter für gegebene Bedingungen optimal realisiert werden kann und welche Übertragungscharakteristik in etwa zu erwarten ist.

Neben den Implementierungen generischer VHDL-Komponenten, werden diese durch Simulationsprogramme für das Linux-Betriebssystem realisiert, so dass erheblich kürzere Simulationszeiten im Vergleich zu einer Simulation der Hardware zu erwarten sind.

Ausblick

Da das DDC-System bisher nur in einzelnen Komponenten für eine Zielhardware synthetisiert wurde, muss die Synthese des Gesamtsystems vor einem Einsatz in einem produktiven System erfolgen. Dabei sind Probleme bei der Übergabe der Generics zwischen den Filter-Komponenten zu erwarten, da diese aus dem Datentyp `real` bestehen, der als problematisch bezüglich der VHDL-Synthese gilt. Abhilfe könnte ggf. die Übergabe eines Indexes auf ein Array aus Filterkoeffizientenarrays schaffen.

Zudem müsste anhand von weiteren aufwändigen Simulationen herausgefunden werden, mit welchen Bitbreiten die einzelnen Filter für eine Anwendung rechnen müssen, um optimale Ergebnisse zu erzielen.

Eine Reihe weiterer Optimierungsmöglichkeiten weisen die Filter auf. Diese könnten für besonders niedrige Datenraten seriell mit verteilter Arithmetik für die Multiplikation und die Addition realisiert werden. Dadurch würde sich der Hardwareaufwand weiter verringern. Grundsätzlich ist es ebenfalls möglich, eine Multiplikation durch mehrere Additionen und Schiebeoperationen zu realisieren. Hierzu müssten die Filterkoeffizienten analysiert und ggf. optimiert werden, um diese enorme Hardwareoptimierung zu erzielen. Da im Rahmen der Diplomarbeit aber nur generische Komponenten erstellt wurden, wurde diese Möglichkeit nicht weiter untersucht. Für die Addition der Filter in der Realisierung in Direktform, kann der Addierbaum ggf. mit dreifachen Verzweigungen realisiert werden, falls die Zielhardware dreifache Additionen innerhalb eines FPGA-Blocks unterstützt.

Grundsätzlich wurden im Rahmen der Diplomarbeit nur statische Filter realisiert. Für spezielle Anwendungen ist es eventuell notwendig, den Dezimationsfaktor zur Laufzeit zu ändern. Dazu müssten die CIC- und SCIC-Dezimatoren so ausgelegt werden, dass die Dezimation programmierbar ist. Sollen FIR-Filter programmierbar sein, müssen sowohl deren Koeffizienten als auch deren Dezimationsfaktor zur Laufzeit änderbar sein. Der Entwicklungsaufwand ist besonders für programmierbare FIR-Filter hoch, weshalb in der Diplomarbeit von einer Implementierung abgesehen wurde.

Wie bereits angesprochen, bietet der CIC-Dezimator die Optimierungsmöglichkeit, die Register sukzessive zu verkleinern. Dies wurde aufgrund der aufwändigen Berechnungen nicht in VHDL umgesetzt. Ein ähnlicher Ansatz lässt sich vermutlich auch für SCIC-Dezimatoren finden, konnte aber in keiner angegebenen Literatur gefunden werden. Schätzungsweise dürfte der nötige Aufwand hoch sein, weshalb ebenfalls davon abgesehen wurde.

Anhang

A. Computer-Arithmetik

A.1. Multiplikation

Zwei Binärzahlen lassen sich durch Additions- und Schiebeoperationen in mehreren Takten multiplizieren.

Wie bereits erwähnt, kann eine Zahl als Summe dargestellt werden. Für eine Multiplikation sind ausschließlich die gesetzten Bits dieser Zahl von Bedeutung.

Sollen beispielsweise die Zahlen 3_{10} und 9_{10} miteinander multipliziert werden, muss zunächst die Binärschreibweise beider Zahlen ermittelt werden.

$$3_{10} \rightarrow 0011_2 \tag{A.1}$$

$$9_{10} \rightarrow 1001_2 \tag{A.2}$$

Eine der beiden Zahlen wird als Summe geschrieben.

$$9_{10} = \sum_{n=0}^{3} x_n \cdot 2^n = 2^3 + 2^0 \tag{A.3}$$

Die Multiplikation von 3_{10} und der Summendarstellung von 9 ergibt

$$3_{10} \cdot \sum_{n=0}^{3} x_n \cdot 2^n = 3(2^3 + 2^0) = 27. \tag{A.4}$$

Alle Potenzen von Zwei lassen sich durch einfache Schiebeoperationen realisieren. Das Ergebnis ergibt sich, wie die Formel zeigt, aus der Summe von 3 << 3 und 3 << 0.

Die Anzahl der Additionen hängt von der Anzahl gesetzter Bits der Zahl ab, die als Summe betrachtet wird. Eine $N \cdot N$-Multiplikation benötigt N Berechnungsschritte; eine $N \cdot M$-Multiplikation M Berechnungsschritte, wenn $M < N$, sonst N Schritte.

Diese Methode wird auch die *„Papier-und-Bleistift-Methode“* genannt. Auf Papier werden alle Zahlen, die aufsummiert werden sollen, untereinander geschrieben, wobei die Schiebeoperationen durch Einrücken von rechts geschehen. Anschließend werden die Zahlen addiert, das Ergebnis ist das der Multiplikation.

Beispiel ohne Vorzeichen:

$$\begin{array}{cccccccccr}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & & \\
\hline
 & & & 0 & 1 & 0 & 1 & 1 & \times & (11) \\
 & & & & 0 & 1 & 1 & 0 & & (6) \\
\hline
 & & & 0 & 0 & 0 & 0 & 0 & & \\
 & & 0 & 1 & 0 & 1 & 1 & & & \\
 & 0 & 1 & 0 & 1 & 1 & & & & \\
0 & 0 & 0 & 0 & 0 & & & & & \\
\hline
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & & (66)
\end{array} \tag{A.5}$$

Für vorzeichenbehaftete Zahlen muss das Vorzeichenbit während der Berechnung bis zum MSB wiederholt werden. Beispiel für vorzeichenbehaftete Zahlen:

$$
\begin{array}{cccccccccc}
7&6&5&4&3&2&1&0&&\\
\hline
&&&&1&0&1&1&\times&(-5)\\
&&&&0&1&1&0&&(6)\\
\hline
&&&&0&0&0&0&&\\
1&1&1&1&0&1&1&&&\\
1&1&1&0&1&1&&&&\\
0&0&0&0&0&&&&&\\
\hline
1&1&1&0&0&0&1&0&&(-30)
\end{array}
\tag{A.6}
$$

Die beschriebene Multiplikationsmethode ist einfach, aber relativ langsam. Wenn mit normierten Integerzahlen gerechnet wird, wird nach einer Multiplikation in der Regel die untere Hälfte der Ergebnisbits abgeschnitten. Dieser Vorgang lässt sich in die serielle Multiplikationsmethode für nicht-vorzeichenbehaftete Zahlen integrieren, indem nach jeder Addition das LSB abgeschnitten wird. Das Ergebnis gleicht dem einer normalen Multiplikation, wobei sich die Registeranzahl verringert. Diese Methode findet Verwendung bei der Amplitudenüberwachung des adaptiven IIR-Oszillators.

A.2. Booth-Algorithmus

Mit Hilfe des Booth-Algorithmus ist es möglich, in relativ wenigen Takten eine Multiplikation zweier Binärzahlen in Zweierkomplementdarstellung durchzuführen. Gegenüber der herkömmlichen *„Papier-und-Bleistift-Methode“* ist der Booth-Algorithmus schneller.

Der Booth-Algorithmus basiert auf der Idee einer Zahlendarstellung, die sich durch Bit-Sequenzen ergibt.

Die Zahl X sei

$$
X = \overbrace{\underbrace{0}_{n}\ 0 \ldots 0\ \underbrace{0}_{u}}^{Seq.0}\ \overbrace{\underbrace{1}_{u-1}\ 1 \ldots 1\ \underbrace{1}_{v}}^{Seq.1}\ \overbrace{\underbrace{0}_{v-1}\ 0 \ldots 0\ \underbrace{0}_{0}}^{Seq.0}. \tag{A.7}
$$

Sie besteht aus drei Bit-Sequenzen. Der Zahlenwert von X ergibt sich aus der Summe und Gewichtungen der Bits, die Eins sind, also

$$
X = \sum_{i=v}^{u-1} 2^i. \tag{A.8}
$$

Alternativ lässt sich der Wert aber auch aus

$$
X = 2^u - 2^v \tag{A.9}
$$

berechnen. u und v sind jeweils die ersten Bits nach einer Bitänderung aus Sicht vom LSB. Die zweite Möglichkeit ergibt eine Berechnung aus zwei Schiebeoperationen und einer Subtraktion, während die erste Operation $u - v$ Additionen benötigt.

Diese Zahlendarstellung kann auch auf eine Multiplikation übertragen werden. Hierzu sei B eine Binärzahl in Zweierkomplementdarstellung

$$
B \cdot X = \sum_{i=v}^{u-1} B \cdot 2^i \tag{A.10}
$$

oder

$$B \cdot X = B \cdot 2^u - B \cdot 2^v. \tag{A.11}$$

Für eine Multiplikation müssen Bitfolgenänderungen detektiert werden, um mit Hilfe der o.g. Zahlendarstellung eine schnelle Berechnung durchführen zu können.

Beispielsweise ist die Darstellung der Zahl $A = 10_{10} = 01010_2$, wenn jeweils 2 Bit verglichen werden

$$\begin{array}{cccccccc}
4 & 3 & 2 & 1 & 0 & & & \\
\hline
0 & 1 & 0 & 1 & 0 & (0) & & \\
\hline
 & & & & 0 & (0) & & \rightarrow a_0 = 0 \\
 & & & 1 & 0 & & & \rightarrow a_1 = -2^1 \\
 & & 0 & 1 & & & & \rightarrow a_2 = 2^2 \\
 & 1 & 0 & & & & & \rightarrow a_3 = -2^3 \\
1 & 0 & & & & & & \rightarrow a_3 = 2^4
\end{array} . \tag{A.12}$$

Die Zahl A lässt sich somit darstellen als

$$A = 2^4 - 2^3 + 2^2 - 2^1 = 2^3 + 2^1 \tag{A.13}$$

$$B \cdot A = B \cdot 2^3 + B \cdot 2^1. \tag{A.14}$$

Beispiel für Sequenzen von 2 Bit: $A = -6_{10} = 111010_2$ und $B = 5_{10} = 000101$ bzw. $-B = 111011$.

Die Sequenzen von A sind

$$\begin{array}{cccccccc}
5 & 4 & 3 & 2 & 1 & 0 & & \\
\hline
1 & 1 & 1 & 0 & 1 & 0 & (0) & \\
\hline
 & & & & & 0 & (0) & \rightarrow 0 \\
 & & & & 1 & 0 & & \rightarrow -2^1 \\
 & & & 0 & 1 & & & \rightarrow 2^2 \\
 & & 1 & 0 & & & & \rightarrow -2^3 \\
 & 1 & 1 & & & & & \rightarrow 0 \\
1 & 1 & & & & & & \rightarrow 0
\end{array} \tag{A.15}$$

Die Berechnung ist dann

$$\begin{array}{cccccccccccc}
10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & \\
\hline
 & & & & & 0 & 0 & 0 & 0 & 0 & 0 & \\
(1) & (1) & (1) & (1) & 1 & 1 & 1 & 0 & 1 & 1 & & -B \cdot 2^1 \\
 & & & 0 & 0 & 0 & 1 & 0 & 1 & & & B \cdot 2^2 \\
(1) & (1) & 1 & 1 & 1 & 0 & 1 & 1 & & & & -B \cdot 2^3 \\
 & 0 & 0 & 0 & 1 & 0 & 1 & & & & & \\
0 & 0 & 0 & 1 & 0 & 1 & & & & & & \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & (-30)
\end{array} \tag{A.16}$$

Die Berechnung benötigt genauso viele partielle Summen wie die klassische Methode, berechnet aber automatisch Ergebnisse im Zweierkomplement.

Die Anzahl der partiellen Summen lässt sich verringern, wenn sich überlappende m-Bit-Sequenzen herangezogen werden, wobei $m > 2$ ist.

Beispiel für $m = 3$:

5	4	3	2	1	0		
1	1	1	0	1	0	(0)	
				1	0	(0)	$a_0 = -2^1$
		1	0	1			$a_1 = 2^1 - 2^3 = -2^2$
1	1	1					$a_2 = 0$

(A.17)

$$A = -2^3 + 2^2 - 2^1 \tag{A.18}$$

$$B \cdot A = -2^3 \cdot B + 2^2 \cdot B - 2^1 \cdot B \tag{A.19}$$

$A = -6_{10} = 111010_2$ und $B = 5_{10} = 000101$ bzw. $-B = 111011$.

10	9	8	7	6	5	4	3	2	1	0	
(1)	(1)	(1)	(1)	1	1	1	0	1	1		$a_0 \cdot B$
(1)	(1)	(1)	1	1	1	0	1	1			$a_1 \cdot B$
0	0	0	0	0	0	0					$a_2 \cdot B$
1	1	1	1	1	1	0	0	0	1	0	(−30)

(A.20)

Im Vergleich zu einer Zweiersequenz halbiert sich die Anzahl der Summanden bei der Dreiersequenz.

Allgemein lässt sich sagen, dass die Multiplikation einer N-Bit-Zahl A so viele Schritte benötigt, wie sich überlappende Sequenzen in N passen.

$$R = \left\lceil \frac{N}{m-1} \right\rceil, \tag{A.21}$$

wobei m die Länge der Sequenzen darstellt.

Aus den Tabellen A.1, A.2 und A.3, lassen sich die Terme entnehmen, die aus den jeweiligen Bitfolgen resultieren.

Werden vier Bit verglichen, ergeben sich Faktoren, die nicht mehr als Potenzen von Zwei darstellbar sind.

A_i	A_{i-1}	$a_i/2^i$
0	0	0
0	1	B
1	0	$-B$
1	1	0

Tabelle A.1.: Booth-Faktoren für 2-Bit-Sequenzen

A_i	A_{i-1}	A_{i-2}	$a_i/2^i$
0	0	0	0
0	0	1	B
0	1	0	B
0	1	1	$2B$
1	0	0	$-2B$
1	0	1	$-B$
1	1	0	$-B$
1	1	1	0

Tabelle A.2.: Booth-Faktoren für 3-Bit-Sequenzen

A_i	A_{i-1}	A_{i-2}	A_{i-3}	$a_i/2^i$
0	0	0	0	0
0	0	0	1	B
0	0	1	0	B
0	0	1	1	$2B$
0	1	0	0	$2B$
0	1	0	1	$3B$
0	1	1	0	$3B$
0	1	1	1	$4B$
1	0	0	0	$-4B$
1	0	0	1	$-3B$
1	0	1	0	$-3B$
1	0	1	1	$-2B$
1	1	0	0	$-2B$
1	1	0	1	$-B$
1	1	1	0	$-B$
1	1	1	1	0

Tabelle A.3.: Booth-Faktoren für 4-Bit-Sequenzen

B. Anhang für Digitale Oszillatoren

B.1. Lookuptable-Oszillator

B.1.1. Selbstfüllender Lookuptable für den Lookuptable-Oszillator

Es wurde eine Lookuptable-Komponente implementiert, die sich während der Kompilierung selber mit Werten füllt (Bild B.1). Die Komponente lehnt an der vorgeschlagenen Implementierung eines ROMs in VHDL an, das aus der Hilfe der Altera-Software entnommen werden kann.

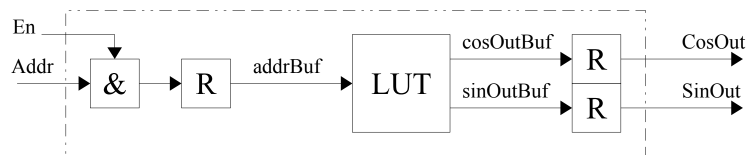


Bild B.1.: Blockschaltbild des selbstfüllenden Lookuptables

Um eine schnelle Schaltung zu realisieren, werden alle Ein- und Ausgänge in das Register geschrieben. Register für Ein- und Ausgänge sind in modernen FPGAs in jedem Logikblock vorhanden und erzeugen keinen zusätzlichen Hardwareaufwand.

Listing B.1: VHDL-Code des selbstberechnenden Lookuptables

```
1  --
2  -- SinCosRom_Archi.vhd
3  --
4  -- Frederik Teichert
5  --
6  -- Instanziiert einen Lookuptable für die
7  -- Verwendung in einem Sinus-Cosinus-Oszillator
8  -- mit vorgegebener Frequenz.
9  -- Die Werte werden zur Kompilierung
10 -- automatisch errechnet.
11 --
12
13 LIBRARY ieee;
14 USE ieee.std_logic_1164.all;
15 USE ieee.numeric_std.all;
16 USE ieee.math_real.all;
17
18 ENTITY SinCosRom IS
```

```
19   GENERIC(
20    REG_WIDTH   : natural := 16;
21    ROM_LENGTH  : natural := 2000;
22    ADDR_LEN    : natural := 16;
23    W0          : real    := 0.00314159
24   );
25   PORT(
26    Clk         : IN     std_logic;
27    En          : IN     std_logic;
28    Addr        : IN     unsigned(ADDR_LEN-1 downto 0);
29    CosOut      : OUT    signed ( REG_WIDTH-1 DOWNTO 0 );
30    Sinout      : OUT    signed ( REG_WIDTH-1 DOWNTO 0 )
31   );
32  END ENTITY SinCosRom;
33
34  ARCHITECTURE SinCosRom_Archi OF SinCosRom IS
35   -- Register für Ein- und Ausgänge
36   SIGNAL addrBuf    : unsigned(ADDR_LEN-1 downto 0);
37   SIGNAL cosOutBuf : signed(REG_WIDTH-1 downto 0);
38   SIGNAL sinOutBuf : signed(REG_WIDTH-1 downto 0);
39
40   TYPE T_varray is array (0 to ROM_LENGTH-1) of signed(↲
41                              REG_WIDTH-1 downto 0);
42
43   -- Die Funktionen getCos() und getSin()
44   -- erzeugen während der Kompilierung
45   -- ein Array aus Funktionswerten
46   -- gleichnamiger Funktionen.
47   function getCos return T_varray is
48    variable vCos : T_Varray;
49   begin
50    for i in 0 to ROM_LENGTH-1 loop
51     vCos(i) := to_signed(integer(cos(real(i)*W0)*real(2**(↲
52          REG_WIDTH-1)-1)), vCos(0)'length);
53    end loop;
54    return vCos;
55   end getCos;
56
57   function getSin return T_varray is
58    variable vSin : T_Varray;
59   begin
60    for i in 0 to ROM_LENGTH-1 loop
61     vSin(i) := to_signed(integer(sin(real(i)*W0)*real(2**(↲
62          REG_WIDTH-1)-1)), vSin(0)'length);
63    end loop;
64    return vSin;
65   end getSin;
66
67   -- Folgende Konstanten werden
68   -- zu einem ROM instanziiert:
69   CONSTANT cCos : T_varray := getCos;
```

```
70  CONSTANT cSin : T_varray := getSin;
71
72 BEGIN
73
74 -- Ein- und Ausgänge in Register schreiben
75 -- und den LUT ansteuern:
76 RomSelect : PROCESS (Clk) is
77 BEGIN
78  if Clk'event and Clk='1' then
79   if En='1' then
80    addrBuf   <= Addr;
81    cosOutBuf <= cCos(to_integer(addrBuf));
82    sinOutBuf <= cSin(to_integer(addrBuf));
83    CosOut    <= cosOutBuf;
84    SinOut    <= sinOutBuf;
85   end if;
86  end if;
87 END PROCESS RomSelect;
88
89 END ARCHITECTURE SinCosRom_Archi;
```

B.1.2. Oszillator

Die Implementierung des Lookuptable-Oszillators in VHDL beinhaltet einen Modulo-N-Zähler, der den selbstfüllenden Lookuptable ansteuert (Bild B.2).

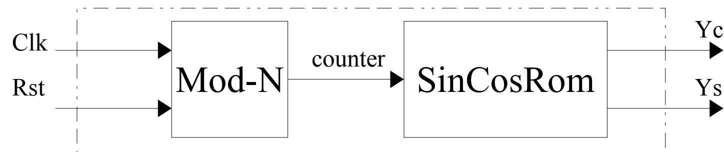


Bild B.2.: Blockschaltbild des Lookuptable-Oszillators

Listing B.2: VHDL-Code des Lookuptable-Oszillators

```
1  --
2  -- QdsoRom_Archi.vhd
3  --
4  -- Frederik Teichert
5  --
6  -- Sinus-Cosinus-Oszillator
7  -- mit Lookuptable mit einstellbarer
8  -- und nicht änderbarer Frequenz.
9  --
10
11 LIBRARY ieee;
12 USE ieee.std_logic_1164.all;
13 USE ieee.numeric_std.all;
14 USE ieee.math_real.all;
15
16 ENTITY QdsoRom IS
17  GENERIC(
18   REG_WIDTH    : natural := 16;
19   MAX_ERR      : natural := 7;
20   CLK_FREQ     : real    := 1.0;
21   OSC_FREQ     : real    := 0.0027
22  );
23  PORT(
24   Clk          : IN     std_logic;
25   Rst          : IN     std_logic;
26   Yc           : OUT    signed ( REG_WIDTH-1 DOWNTO 0 );
27   Ys           : OUT    signed ( REG_WIDTH-1 DOWNTO 0 )
28  );
29 END ENTITY QdsoRom;
30
31 ARCHITECTURE QdsoRom_Archi OF QdsoRom IS
32  -- Berechnung der Anzahl zu speichernder Werte
33  -- in Abhängigkeit der Registerbreite, des
34  -- akzeptablen Fehlers und der Takt- bzw.
```

```
35  -- Ausgangsfrequenz
36  function getRomLen(maxperiods : integer) return integer is
37   variable P : real;
38   variable w : real := 2.0*MATH_PI*OSC_FREQ/CLK_FREQ;
39   variable M : real := real(2**(REG_WIDTH-1)-1);
40   variable ecos, esin : integer;
41   variable N : integer;
42  begin
43   P := abs(CLK_FREQ/OSC_FREQ);
44   for i in 1 to maxperiods loop
45    ecos := integer(M*cos(0.0)) - integer(M*cos(round(real(i) ⤸
46            *P)*w));
47    esin := integer(M*cos(0.0)) - integer(M*cos(round(real(i) ⤸
48            *P)*w));
49    if ecos < MAX_ERR and esin < MAX_ERR then
50     N := integer(round(real(i)*P));
51     report "----------------- ROM besteht aus "
52       & natural'image(N)
53       & " Werten"
54     severity note;
55     return N;
56    end if;
57   end loop;
58   report "----------------- FEHLER! kann kein Rom mit " & ⤸
59   natural'image(maxperiods) & " Perioden erstellen und dabei ⤸
60   den Fehler akzeptable klein halten -----------------"
61   severity error;
62   assert false report " ------- ENDE!!! ------"
63   severity failure;
64   return 23;
65  end getRomLen;
66
67  -- Anzahl der zu speichernden Werte:
68  CONSTANT cROM_LENGTH : integer := getRomLen(100);
69  -- Berechnung der Frequenz für den LUT:
70  CONSTANT cW0         : real  := 2.0*MATH_PI*OSC_FREQ/CLK_FREQ;
71
72  -- Maximaler Zählerstand:
73  CONSTANT cCounterLen : natural := natural(ceil(log2(real( ⤸
74                                    cROM_LENGTH))));
75  -- Signal für den Modulo-N-Zähler:
76  SIGNAL   counter     : unsigned(cCounterLen-1 downto 0);
77
78  -- Komponentendeklaration des LUTs
79  COMPONENT SinCosRom IS
80  GENERIC(
81   REG_WIDTH   : natural := 16;
82   ROM_LENGTH  : natural := 2000;
83   ADDR_LEN    : natural := 16;
84   W0          : real    := 0.00314159
85  );
```

```
 86  PORT(
 87   Clk      : IN     std_logic;
 88   En       : IN     std_logic;
 89   Addr     : IN     unsigned ( ADDR_LEN-1 downto 0 );
 90   CosOut   : OUT    signed ( REG_WIDTH-1 DOWNTO 0 );
 91   Sinout   : OUT    signed ( REG_WIDTH-1 DOWNTO 0 )
 92  );
 93  END COMPONENT;
 94
 95  -- Ein Signal für dauerhaftes Enable-Signal
 96  -- des LUTs:
 97  SIGNAL shigh : std_logic;
 98 BEGIN
 99 shigh <= '1';
100
101 -- Instanziierung der Komponente für den LUT:
102 SinCosLut : SinCosRom
103    GENERIC MAP (
104     REG_WIDTH,
105     cROM_LENGTH,
106     cCounterLen,
107     cW0
108    )
109    PORT MAP (
110     Clk => Clk,
111     En   => shigh,
112     Addr => counter,
113     CosOut => Yc,
114     SinOut => Ys
115    );
116
117 -- Der Prozess AddressCounter
118 -- realisiert einen Modulo-N-Zähler, der
119 -- den LUT ansteuert.
120 AddressCounter : PROCESS (Clk, Rst) is
121 begin
122  if Rst='0' then
123   counter <= to_unsigned(0, counter'length);
124  elsif Clk'event and Clk='1' then
125   -- Zählen:
126   counter <= counter+1;
127   if counter >= to_unsigned(cROM_LENGTH-1, counter'length) ⤸
128                 then
129    counter <= to_unsigned(0, counter'length);
130   end if;
131  end if;
132 end PROCESS QdsoRomCalc;
133
134 END ARCHITECTURE QdsoRom_Archi;
```

B.2. IIR-Oszillator

B.2.1. Zählerbasierter IIR-Oszillator

Die Implementierung des zählerbasierten IIR-Oszillators in VHDL beinhaltet einen Modulo-N-Zähler, der für die Neuinitialisierung verwendet wird.

Listing B.3: VHDL-Code des zählerbasierten IIR-Oszillators

```
 1 --
 2 -- QdsoCnt_Archi.vhd
 3 --
 4 -- Frederik Teichert
 5 --
 6 -- Ein IIR-Oszillator mit
 7 -- zählerbasierter Neuinitialisierung.
 8 --
 9 LIBRARY ieee;
10 USE ieee.std_logic_1164.all;
11 USE ieee.numeric_std.all;
12
13 USE ieee.numeric_std.all;
14 USE ieee.math_real.all;
15
16 ENTITY QdsoCnt IS
17  GENERIC(
18   REG_WIDTH   : natural := 18;
19   OUT_WIDTH   : natural := 18;
20   MAX_COUNT   : natural := 64000;
21   CLK_FREQ    : real    := 1.0;
22   OSC_FREQ    : real    := 0.0027
23  );
24
25  PORT(
26   Clk     : IN     std_logic;
27   Rst     : IN     std_logic;
28   CosOut  : OUT    signed ( OUT_WIDTH-1 DOWNTO 0 );
29   SinOut  : OUT    signed ( OUT_WIDTH-1 DOWNTO 0 );
30   Restart : IN     std_logic
31  );
32
33 -- Declarations
34
35 END QdsoCnt ;
36
37 --
38 ARCHITECTURE QdsoCnt_Archi OF QdsoCnt IS
39  -- Ausgangsfrequenz errechnen:
40  CONSTANT cW0 : real := 2.0*MATH_PI*OSC_FREQ/CLK_FREQ;
41
```

```
42  -- Register für die fortlaufende Berechnung:
43  SIGNAL yC, yS : signed (REG_WIDTH-1 downto 0);
44
45  -- Zähler für die Neuinitialisierung:
46  CONSTANT cCounterWidth : integer := integer(ceil(log2(real( ↩
47                                      MAX_COUNT))));
48  SIGNAL   cnt : unsigned (cCounterWidth-1 downto 0);
49
50  -- Konstante Faktoren:
51  CONSTANT cCosW0 : signed(REG_WIDTH-1 downto 0) := to_signed( ↩
52                           integer(real(2**(REG_WIDTH-1)-1) ↩
53                           *cos(cW0)), cCosW0'length);
54  CONSTANT cSinW0 : signed(REG_WIDTH-1 downto 0) := to_signed( ↩
55                           integer(real(2**(REG_WIDTH-1)-1) ↩
56                           *sin(cW0)), cSinW0'length);
57 BEGIN
58
59 -- Ausgänge:
60 CosOut <= resize(shift_right(yC, REG_WIDTH-OUT_WIDTH), ↩
61           OUT_WIDTH);
62 SinOut <= resize(shift_right(yS, REG_WIDTH-OUT_WIDTH), ↩
63           OUT_WIDTH);
64
65 -- Der Prozess QDSO errechnet jeweils den Folgewert
66 -- der Funktionen und kann eine
67 -- Neuinitialisierung durchführen.
68 QDSO : PROCESS (Clk, Rst, Restart) is
69 begin
70  if Rst='0' then
71   -- reset:
72   yC     <= to_signed(0, REG_WIDTH);
73   yS     <= to_signed(0, REG_WIDTH);
74   cnt    <= to_unsigned(0, cCounterWidth);
75  elsif Clk'event and Clk='1' then
76   if Restart='1' then
77    yC     <= CosW0;
78    yS     <= SinW0;
79    cnt    <= to_unsigned(0, cCounterWidth);
80   else
81    -- Folgewert berechnen:
82    -- Cosinus:
83    yC <= resize(shift_right(cCosW0*yC, REG_WIDTH-1), ↩
84          REG_WIDTH)
85      -resize(shift_right(cSinW0*yS, REG_WIDTH-1), REG_WIDTH);
86    -- Sinus:
87    yS <= resize(shift_right(cSinW0*yC, REG_WIDTH-1), ↩
88          REG_WIDTH)
89      +resize(shift_right(cCosW0*yS, REG_WIDTH-1), REG_WIDTH);
90
91    -- Zählen:
92    cnt <= cnt + 1;
```

```
 93    if cnt >= MAX_COUNT-1 then
 94     cnt <= to_unsigned(0, cCounterWidth);
 95     -- Neuinitialisierung:
 96     yC  <= cCosW0;
 97     yS  <= cSinW0;
 98    end if;
 99   end if;
100  end if;
101 end PROCESS;
102
103 END ARCHITECTURE QdsoCnt_Archi;
```

B.2.2. IIR-Oszillator mit Nulldurchgangsdetektion

Die Implementierung des IIR-Oszillators mit Nulldurchgangsdetektion in VHDL benötigt Komparatoren, die für die Neuinitialisierung verwendet werden.

Listing B.4: VHDL-Code des überwachten IIR-Oszillators

```
 1 --
 2 -- QdsoWd_Archi.vhd
 3 --
 4 -- Frederik Teichert
 5 --
 6 -- Ein IIR-Oszillator mit
 7 -- Nulldurchgangsdetektion.
 8 --
 9 LIBRARY ieee;
10 USE ieee.std_logic_1164.all;
11 USE ieee.numeric_std.all;
12 USE ieee.math_real.all;
13
14 ENTITY QdsoWd IS
15  GENERIC(
16   REG_WIDTH   : natural := 18;
17   OUT_WIDTH   : natural := 18;
18   CLK_FREQ    : real    := 1.0;
19   OSC_FREQ    : real    := 0.0027;
20   WD_EXP      : integer := 1;
21   WD_STYLE    : string  := "WD_SIN"
22  );
23  PORT(
24   Clk     : IN     std_logic;
25   Rst     : IN     std_logic;
26   CosOut  : OUT    signed ( OUT_WIDTH-1 DOWNTO 0 );
27   SinOut  : OUT    signed ( OUT_WIDTH-1 DOWNTO 0 );
28   Restart : IN     std_logic
29  );
30 END ENTITY QdsoWd;
31
32 --
33 ARCHITECTURE QdsoWd_Archi OF QdsoWd IS
34  -- Ausgangsfrequenz errechnen:
35  CONSTANT cW : real := 2.0*MATH_PI*OSC_FREQ/CLK_FREQ;
36
37  -- Register für die fortlaufende Berechnung:
38  SIGNAL yC, yS : signed (REG_WIDTH-1 downto 0);
39
40  -- Konstante Faktoren:
41  CONSTANT cCosW0   : signed(REG_WIDTH-1 downto 0)
42      := to_signed(integer(real(2**(REG_WIDTH-1)-1)*cos(cW0)), ⤸
43         cCosW0'length);
44  CONSTANT cSinW0   : signed(REG_WIDTH-1 downto 0)
```

```
45      := to_signed(integer(real(2**(REG_WIDTH-1)-1)*sin(cW0)), ⤸
46         cSinW0'length);
47
48  CONSTANT cMaxVal : signed (REG_WIDTH-1 downto 0)
49      := to_signed(2**(REG_WIDTH-1)-1, REG_WIDTH);
50  CONSTANT cMinVal : signed (REG_WIDTH-1 downto 0)
51      := to_signed(-1*(2**(REG_WIDTH-1)-1), REG_WIDTH);
52 BEGIN
53
54 -- Ausgänge:
55 CosOut <= resize(shift_right(yC, REG_WIDTH-OUT_WIDTH), ⤸
56           OUT_WIDTH);
57 SinOut <= resize(shift_right(yS, REG_WIDTH-OUT_WIDTH), ⤸
58           OUT_WIDTH);
59
60 -- Der Prozess QDSO errechnet jeweils den Folgewert
61 -- der Funktionen und kann eine
62 -- Neuinitialisierung durchführen.
63 QDSO : PROCESS (Clk, Rst, Restart, CosW0, SinW0) is
64  variable vyC, vyS : signed(REG_WIDTH-1 downto 0);
65 begin
66  if Rst='0' then
67   -- reset:
68   yC             <= cCosW0;
69   yS             <= cSinW0;
70  elsif Clk'event and Clk='1' then
71   if Restart='1' then
72    yC             <= cCosW0;
73    yS             <= cSinW0;
74   else
75
76    -- calculate cosine
77    vyC := resize(shift_right(cCosW0*yC, REG_WIDTH-1), ⤸
78           REG_WIDTH)
79       -resize(shift_right(cSinW0*yS, REG_WIDTH-1), REG_WIDTH);
80    -- calculate sine
81    vyS := resize(shift_right(cSinW0*yC, REG_WIDTH-1), ⤸
82           REG_WIDTH)
83       +resize(shift_right(cCosW0*yS, REG_WIDTH-1), REG_WIDTH);
84
85    -- Nulldurchgangsdetektion
86    -- wenn einer der Werte innerhalb des definierten ⤸
87    Nulldurchgangsbereichs ist,
88    -- wird das flag gesetzt, um die nächste Berechnung mit
89    -- den Maximal- bzw. Minimalwerten durchzuführen:
90    --default-value:
91
92    if WD_STYLE="WD_COS" then
93     if  (vyC < to_signed(0, REG_WIDTH) and vyC > shift_right(- ⤸
94          1*cCosW0, WD_EXP))
95     then
```

```
 96        vyC := to_signed(0, REG_WIDTH);
 97        if vyS < to_signed(0, REG_WIDTH) then
 98         vyS := cMinVal;
 99        else
100         vyS := cMaxVal;
101        end if;
102       end if;
103      elsif WD_STYLE="WD_SIN" then
104       if  (vyS < to_signed(0, REG_WIDTH) and vyS > shift_right(-
105            1*cSinW0, WD_EXP))
106       then
107        yS <= to_signed(0, REG_WIDTH);
108        if vyC < to_signed(0, REG_WIDTH) then
109         vyC := cMinVal;
110        else
111         vyC := cMaxVal;
112        end if;
113       end if;
114      end if;
115
116      yC <= vyC;
117      yS <= vyS;
118     end if;
119    end if;
120   end PROCESS;
121
122   END ARCHITECTURE QdsoWd_Archi;
```

B.2.3. Adaptiver IIR-Oszillator

Die Implementierung des adaptiven IIR-Oszillators in VHDL beinhaltet zwei Komponenten, einen IIR-Oszillator, der eine Polverschiebung durchführen kann und einen seriellen Multiplizierer, der die Amplitude überprüft und die Polverschiebung auslöst (Bild B.3).

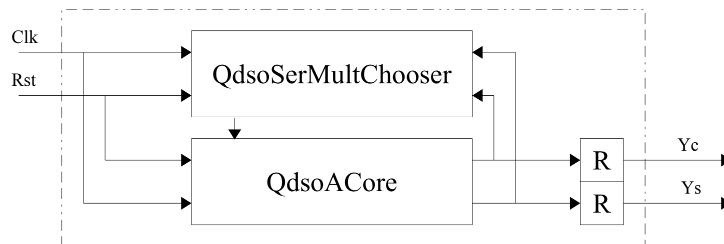


Bild B.3.: Blockschaltbild des adaptiven IIR-Oszillators

Listing B.5: VHDL-Code des Kerns des adaptiven IIR-Oszillators

```
 1 --
 2 -- QdsoACore_Archi.vhd
 3 --
 4 -- Frederik Teichert
 5 --
 6 -- Sinus-Cosinus-Oszillator
 7 -- mit IIR und einstellbaren
 8 -- Koeffizientensätzen.
 9 --
10 LIBRARY ieee;
11 USE ieee.std_logic_1164.all;
12 USE ieee.numeric_std.all;
13 USE ieee.math_real.all;
14
15 ENTITY QdsoACore IS
16     GENERIC(
17         REG_WIDTH : natural := 18;
18         OUT_WIDTH : natural := 18;
19         W0        : real    := 0.3
20     );
21     PORT(
22         Clk      : IN     std_logic;
23         Rst      : IN     std_logic;
24         CoeffSet : IN     std_logic;
25         Yc       : OUT    signed (REG_WIDTH-1 DOWNTO 0);
26         Ys       : OUT    signed (REG_WIDTH-1 DOWNTO 0)
27     );
28 END QdsoACore ;
29
30 ARCHITECTURE QdsoACore_Archi OF QdsoACore IS
```

```
31  -- Signale zum Speichern der letzten Werte:
32  SIGNAL syC, syS : signed (REG_WIDTH-1 downto 0);
33
34  -- Konstanten:
35  -- Maximalamplitude:
36  CONSTANT cMax : natural := 2**(REG_WIDTH-2)-1;
37
38  -- Berechnung von zwei Koeffizientensätzen:
39  CONSTANT cCos_min : signed(REG_WIDTH-1 downto 0)  :=
40                             to_signed(integer(cos(W0)*real(
41                             cMax)), REG_WIDTH);
42  CONSTANT cSin_min : signed(REG_WIDTH-1 downto 0)  :=
43                             to_signed(integer(sin(W0)*real(
44                             cMax)), REG_WIDTH);
45  CONSTANT cAmp     : signed(REG_WIDTH-1 downto 0)  :=
46                             to_signed(to_integer(
47          (cCos_min*cCos_min + cSin_min*cSin_min)
48          /integer(cMax)), REG_WIDTH);
49  CONSTANT cCos_max : signed(REG_WIDTH-1 downto 0)  :=
50          to_signed(
51           to_integer(cCos_min*cMax/cAmp+2)
52           , REG_WIDTH);
53  CONSTANT cSin_max : signed(REG_WIDTH-1 downto 0)  :=
54                             to_signed(
55           to_integer(cSin_min*cMax/cAmp+2)
56           , REG_WIDTH);
57 BEGIN
58
59 -- Ausgänge treiben:
60 yC <= syC;
61 yS <= syS;
62
63
64 -- Der Prozess QDSO führt die Berechnung
65 -- der Folgewerte von Sinus und Cosinus
66 -- abhängig von dem Koeffizientensatz durch:
67 QDSO : PROCESS (Clk, Rst) is
68 begin
69  if Rst='0' then
70   -- reset:
71   syC     <= to_signed(cMax, REG_WIDTH);
72   syS     <= to_signed(0, REG_WIDTH);
73  elsif Clk'event and Clk='1' then
74   -- Auswahl des zu verwendenden Koeffizientensatzes:
75   if (CoeffSet = '1') then
76    syC <= resize(shift_right(cCos_max*syC, REG_WIDTH-2),
77           REG_WIDTH)
78     -resize( shift_right( cSin_max*syS, REG_WIDTH-2),
79              REG_WIDTH);
80    -- calculate sine
81    syS <= resize(shift_right(cSin_max*syC, REG_WIDTH-2),
```

```
82            REG_WIDTH)
83      +resize( shift_right( cCos_max*syS, REG_WIDTH-2),
84               REG_WIDTH);
85    else
86      -- calculate cosine
87     syC <= resize(shift_right(cCos_min*syC, REG_WIDTH-2),
88            REG_WIDTH)
89      -resize( shift_right( cSin_min*syS, REG_WIDTH-2),
90               REG_WIDTH);
91     -- calculate sine
92     syS <= resize(shift_right(cSin_min*syC, REG_WIDTH-2),
93            REG_WIDTH)
94      +resize( shift_right( cCos_min*syS, REG_WIDTH-2),
95               REG_WIDTH);
96    end if;
97   end if;
98  end PROCESS;
99  END ARCHITECTURE QdsoACore_Archi;
```

Listing B.6: VHDL-Code des seriellen Multiplizierers des adaptiven IIR-Oszillators

```
 1 --
 2 -- QdsoSerMultChooser_Archi.vhd
 3 --
 4 -- Frederik Teichert
 5 --
 6 -- Serieller Multiplier
 7 -- für die Überwachung der Amplitude
 8 -- der Oszillation und zum
 9 -- Umschalten zwischen
10 -- zwei Koeffizientensätzen.
11 --
12 LIBRARY ieee;
13 USE ieee.std_logic_1164.all;
14 USE ieee.numeric_std.all;
15
16 ENTITY QdsoSerMultChooser IS
17  GENERIC(
18   REG_WIDTH : natural := 18;
19   OUT_WIDTH : natural := 18
20  );
21  PORT(
22   Clk        : IN     std_logic;
23   Cos        : IN     signed (REG_WIDTH-1 DOWNTO 0);
24   Rst        : IN     std_logic;
25   CoeffSet   : OUT    std_logic;
26   Sin        : IN     signed (REG_WIDTH-1 DOWNTO 0)
27  );
28 END QdsoSerMultChooser ;
29 ARCHITECTURE QdsoSerMultChooser_Archi OF QdsoSerMultChooser
30 IS
31  CONSTANT cMax : natural := 2**(REG_WIDTH-2)-1;
32  CONSTANT cCntMax : integer := REG_WIDTH;
33  -- one-hot-kodiertes signal, um schnelle designs zu
34  ermöglichen!
35  SIGNAL sCnt    : std_logic_vector(cCntMax downto 0) := (
36                                   others => '0');
37  SIGNAL a, b    : signed(REG_WIDTH-1 downto 0); --
38                           multiplikator:
39  SIGNAL a_std   : std_logic_vector(REG_WIDTH-1 downto 0);
40  SIGNAL b_std   : std_logic_vector(REG_WIDTH-1 downto 0);
41  SIGNAL ap, bp  : signed(REG_WIDTH downto 0);
42  SIGNAL enMult  : bit;
43 BEGIN
44
45 -- Diese Komponente berechnet in mehreren Takten die
46 -- Amplitude von Sinus und Cosinus, wobei die
47 Multiplikationen
48 -- seriell geschehen. Während der Multiplikation wird
```

```
49 jeweils
50 -- durch Zwei dividiert, um die Addiererpfade kurz zu halten
51 -- und hohe Geschwindigkeiten zu garantieren. Nach einer ↩
52 Multiplikation
53 -- werden beide Quadrate aufaddiert und mit dem Maximalwert ↩
54 verglichen.
55 -- Ist die Summe kleiner als der Maximalwert, wird der ↩
56 Ausgang HIGH,
57 -- ansonsten LOW.
58 -- Diese Komponente entscheidet, welche Koeffizienten im
59 -- Oszillator verwendet werden.
60
61 Magnitude : PROCESS (Clk, Rst, Sin, Cos) is
62  variable vM : unsigned(REG_WIDTH+1 downto 0);
63  variable vTmp : std_logic_vector(REG_WIDTH-1 downto 0);
64 begin
65  if Rst='0' then
66   sCnt <= (others => '0');
67   sCnt <= sCnt(sCnt'high downto 1) & '1';
68   enMult <= '0';
69  elsif Clk'event and Clk='1' then
70   -- "zählen" .. one-hot-kodiert:
71   sCnt <= sCnt(sCnt'high-1 downto 0) & sCnt(sCnt'high);  -- ↩
72          ringförmig
73
74   -- 0-ter takt:
75   -- Rücksetzen und Laden:
76   if sCnt(0) = '1' then
77    a <= signed(abs(Cos));
78    a_std <= std_logic_vector(signed(abs(Cos)));
79    ap <= to_signed(0, REG_WIDTH+1);
80    b <= signed(abs(Sin));
81    b_std <= std_logic_vector(signed(abs(Sin)));
82    bp <= to_signed(0, REG_WIDTH+1);
83    enMult <= '1'; -- Multiplikation anstoßen!
84   end if;
85
86   if sCnt(REG_WIDTH-2) = '1' then
87    enMult <= '0'; -- multiplikation anhalten!
88   end if;
89
90   -- ergebnis berechnen:
91   if sCnt(REG_WIDTH-1) = '1' then
92    vM := unsigned(resize(shift_right(ap+bp, 0), REG_WIDTH+2));
93   end if;
94
95   -- ausgang setzen:
96   if sCnt(REG_WIDTH) = '1' then
97    if vM > cMax then
98     CoeffSet <= '0'; -- abklingen
99    else
```

```
100      CoeffSet <= '1'; -- aufklingen
101     end if;
102    end if;
103
104    -- rechnen:
105    if enMult = '1' then
106     -- cos^2() berechnen:
107     if a_std(0) = '1' then
108      ap <= shift_right(a+ap, 1);
109     else
110      ap <= shift_right(ap, 1);
111     end if;
112
113     -- sin^2() berechnen:
114     if b_std(0) = '1' then
115      bp <= shift_right(b+bp, 1);
116     else
117      bp <= shift_right(bp, 1);
118     end if;
119
120     -- a_std und b_std schieben:
121     a_std <= '0' & a_std(a_std'high downto 1);
122     b_std <= '0' & b_std(b_std'high downto 1);
123    end if;
124   end if;
125  end process;
126  END ARCHITECTURE QdsoSerMultChooser_Archi;
```

Listing B.7: VHDL-Code adaptiven IIR-Oszillators (Toplevel)

```
 1 --
 2 -- QdsoAdaptive_Archi.vhd
 3 --
 4 -- Frederik Teichert
 5 --
 6 -- Sinus-Cosinus-Oszillator
 7 -- mit IIR (Toplevel).
 8 --
 9 LIBRARY ieee;
10 USE ieee.std_logic_1164.all;
11 USE ieee.numeric_std.all;
12 USe ieee.math_real.all;
13
14 LIBRARY Oszillatoren_lib;
15 USE Oszillatoren_lib.all;
16
17 ENTITY QdsoAdaptive IS
18  GENERIC(
19   REG_WIDTH    : natural := 18;
20   OUT_WIDTH    : natural := 18;
21   CLK_FREQ     : real    := 1.0;
22   OSC_FREQ     : real    := 0.0027
23  );
24  PORT(
25   Clk : IN     std_logic;
26   Rst : IN     std_logic;
27   Yc  : OUT    signed ( REG_WIDTH-1 DOWNTO 0 );
28   Ys  : OUT    signed ( REG_WIDTH-1 DOWNTO 0 )
29  );
30 END ENTITY QdsoAdaptive;
31
32 ARCHITECTURE QdsoAdaptive_Archi OF QdsoAdaptive IS
33  -- Neuer Datentyp:
34  SUBTYPE T_data is signed(REG_WIDTH-1 downto 0);
35  TYPE T_buf is array  (0 to 2) of T_data;
36  -- Regsiter für die Ausgabe:
37  SIGNAL CS    : std_logic;
38  SIGNAL Yc1   : signed(REG_WIDTH-1 DOWNTO 0);
39  SIGNAL Ys1   : signed(REG_WIDTH-1 DOWNTO 0);
40  SIGNAL bufC : T_buf;
41  SIGNAL bufS : T_buf;
42
43  -- Ausgangsfrequenz berechnen:
44  CONSTANT cW : real := 2.0*MATH_PI*OSC_FREQ/CLK_FREQ;
45
46
47  -- Komponentendeklarationen:
48  COMPONENT QdsoACore
```

```
49  GENERIC (
50   REG_WIDTH : natural := 18;
51   OUT_WIDTH : natural := 18;
52   W0        : real    := 0.01
53  );
54  PORT (
55   Clk      : IN     std_logic;
56   Rst      : IN     std_logic;
57   CoeffSet : IN     std_logic;
58   Yc       : OUT    signed (REG_WIDTH-1 DOWNTO 0);
59   Ys       : OUT    signed (REG_WIDTH-1 DOWNTO 0)
60  );
61  END COMPONENT;
62
63  COMPONENT QdsoSerMultChooser
64  GENERIC (
65   REG_WIDTH : natural := 18;
66   OUT_WIDTH : natural := 16
67  );
68  PORT (
69   Clk      : IN     std_logic ;
70   Cos      : IN     signed (REG_WIDTH-1 DOWNTO 0);
71   Rst      : IN     std_logic ;
72   CoeffSet : OUT    std_logic ;
73   Sin      : IN     signed (REG_WIDTH-1 DOWNTO 0)
74  );
75  END COMPONENT;
76
77  -- pragma synthesis_off
78  FOR ALL : QdsoACore USE ENTITY Oszillatoren_lib.QdsoACore;
79  FOR ALL : QdsoSerMultChooser USE ENTITY Oszillatoren_lib. ⤸
80  QdsoSerMultChooser;
81  -- pragma synthesis_on
82 BEGIN
83
84 -- Taktsynchrone Ausgabe der Ausgangssignale:
85 DriveOutput : PROCESS (Clk, Rst) is
86 begin
87  if Rst='0' then
88   Yc <= to_signed(0, Yc'length);
89   Ys <= to_signed(0, Ys'length);
90  elsif Clk'event and Clk='1' then
91   Yc <= Yc1;
92   Ys <= Ys1;
93  end if;
94 end PROCESS DriveOutput;
95
96 -- Instanziierung der Komponenten:
97 QdsoACore_GEN : QdsoACore
98  GENERIC MAP (
99   REG_WIDTH => REG_WIDTH,
```

```
100    OUT_WIDTH => OUT_WIDTH,
101    W0        => cW0
102   )
103   PORT MAP (
104    Clk      => Clk,
105    Rst      => Rst,
106    CoeffSet => CS,
107    Yc       => Yc1,
108    Ys       => Ys1
109   );
110  QdsoSerMultChooser_GEN : QdsoSerMultChooser
111   GENERIC MAP (
112    REG_WIDTH => REG_WIDTH-1,
113    OUT_WIDTH => REG_WIDTH
114   )
115   PORT MAP (
116    Clk      => Clk,
117    Cos      => Yc1,
118    Rst      => Rst,
119    CoeffSet => CS,
120    Sin      => Ys1
121   );
122  END ARCHITECTURE QdsoAdaptive_Archi;
```

C. VHDL-Komponenten

C.1. Booth-Multiplier mit 4er-Sequenz

In Listing C.1 wird ein Booth-4-Multiplier implementiert. Der Multiplizierer wird sowohl in FIR- als auch in FIR-Halfbandfiltern verwendet. Die Funktionsweise eines Booth-Multiplizierers wird in Anhang A.2 erläutert.

Listing C.1: VHDL-Implementierung eines normalisierten 4-schrittigen Booth-Multipliers

```
 1 --
 2 -- Booth4MultNormalized_Archi.vhd
 3 --
 4 -- Frederik Teichert
 5 --
 6 -- Implementiert einen generischen
 7 -- vierschrittigen, normalisierten Booth-Multiplier.
 8 --
 9 LIBRARY ieee;
10 USE ieee.std_logic_1164.all;
11 USE ieee.numeric_std.all;
12 USE ieee.math_real.all;
13
14 LIBRARY FIR_lib;
15 USE FIR_lib.FIR_Def_pkg.all;
16
17 ENTITY Booth4MultNormalized IS
18  GENERIC(
19   GAIN_BITS   : natural := 0;
20   DATA_WIDTH : natural := pcDATA_WIDTH
21  );
22  PORT(
23   Clk    : IN      std_logic;
24   A      : IN      signed ( DATA_WIDTH-1 DOWNTO 0 );
25   Prod   : OUT     signed ( DATA_WIDTH-1 DOWNTO 0 );
26   Rst    : IN      std_logic;
27   B      : IN      signed ( DATA_WIDTH-1 DOWNTO 0 );
28   Start : IN      std_logic;
29   Stop   : IN      std_logic
30  );
31  GENERIC(
32         DATA_WIDTH : natural := 18
33     );
34 END ENTITY Booth4MultNormalized;
```

```
35
36 ARCHITECTURE Booth4MultNormalized_Archi OF ⤸
37 Booth4MultNormalized IS
38  -- Länge des Registers, um die Blöcke zu detektieren:
39  CONSTANT cBoothLen : natural := natural(ceil(real(DATA_WIDTH) ⤸
40                                  /3.0))*3+1; -- für 16 -> 0 . ⤸
41                                  . 18
42  -- Register:
43  SIGNAL   boothReg   : std_logic_vector(cBoothLen downto 0) := ⤸
44                                        (others => '0');
45  SIGNAL   akku       : signed(DATA_WIDTH+2 downto 0);
46  SIGNAL   run        : bit;
47  SIGNAL   bMult      : signed(DATA_WIDTH-1 downto 0);
48
49  -- Datentyp für die Art der Multiplikation für die ⤸
50  Validierung
51  -- in einer Simulation:
52  type t_mults is (B0, B1, B2, B3, B4 , M4, M3, M2 , M1, ⤸
53                  M_NONE);
54 BEGIN
55
56 BoothMultNormalized : PROCESS (Clk, Rst) is
57  variable vakku, vbmult, vakkushift : signed(akku'range);
58  variable multtype : t_mults := M_NONE;
59 begin
60  if Rst='0' then
61   Prod <= to_signed(0, Prod'length);
62   run  <= '0';
63   boothReg <= (others => '0');
64   akku <= to_signed(0, akku'length);
65   bMult <= to_signed(0, bMult'length);
66  elsif Clk'event and Clk='1' then
67   if Start = '1' then
68    run <= '1';
69    -- Werte speichern.
70    boothReg <= std_logic_vector(shift_left(resize(A, ⤸
71                boothReg'length-1), cBoothLen-DATA_WIDTH+ ⤸
72                GAIN_BITS)) & '0'; -- xxxxxx(0)
73    akku <= to_signed(0, akku'length);
74    bMult <= resize(B, bMult'length);
75   end if;
76
77   if Stop = '1' then
78    run <= '0';
79    Prod <= resize(akku, Prod'length);
80   end if;
81
82   -- Multiplizieren:
83   if run='1' then
84    vakku := akku;
85    case boothReg(3 downto 0) is
```

```
 86        when "0001" | "0010" => -- +B
 87         vbmult := resize(bMult, vbmult'length);
 88         akku <= shift_right(vakku+vbMult, 3);
 89         multtype := B1;
 90        when "0011" | "0100" => -- +2B
 91         vbmult := shift_left(resize(bMult, vbMult'length), 1);
 92         akku <= shift_right(vakku+vbMult, 3);
 93         multtype := B2;
 94        when "0101" | "0110" => -- +3B
 95         vbmult := shift_left(resize(bMult, vbMult'length), 1) +
 96                    bMult;
 97         akku <= shift_right(vakku+vbMult, 3);
 98         multtype := B3;
 99        when "0111" =>          -- +4B
100         vbmult := shift_left(resize(bMult, vbMult'length), 2);
101         akku <= shift_right(vakku+vbMult, 3);
102         multtype := B4;
103        when "1000" =>          -- -4B
104         vbmult := shift_left(resize(bMult, vbMult'length), 2);
105         akku <= shift_right(vakku - vbMult, 3);
106         multtype := M4;
107        when "1001" | "1010" => -- -3B
108         vbmult := shift_left(resize(bMult, vbMult'length), 1) +
109                    bMult;
110         akku <= shift_right(vakku - vbMult, 3);
111         multtype := M3;
112        when "1011" | "1100" => -- -2B
113         vbmult := shift_left(resize(bMult, vbMult'length), 1);
114         akku <= shift_right(vakku - vbMult, 3);
115         multtype := M2;
116        when "1101" | "1110" => -- -B
117         vbmult := resize(bMult, vbmult'length);
118         akku <= shift_right(vakku - vbMult, 3);
119         multtype := M1;
120        when others => akku <= shift_right(akku, 3);
121         multtype := M_NONE;
122       end case;
123
124       -- Schiebeoperation:
125       boothReg <=   boothReg(boothReg'high)
126          & boothReg(boothReg'high)
127          & boothReg(boothReg'high)
128          & boothReg(boothReg'high downto 3);
129      end if;
130     end if;
131   end process BoothMultNormalized;
132   END ARCHITECTURE Booth4MultNormalized_Archi;
```

C.2. Serieller Multiplizierer

In Listing C.2 wird ein serieller Multiplizierer implementiert, der bei jedem Multiplikationsschritt eine Normalisierung durchführt, wodurch eine sehr kompakte Schaltung resultiert. Die Funktionsweise eines seriellen Multiplizierers wird in Anhang A.1 erläutert.

Listing C.2: VHDL-Implementierung eines normalisierten seriellen Multipliers

```
 1 --
 2 -- SerialMultiplierNormalized_Archi.vhd
 3 --
 4 -- Frederik Teichert
 5 --
 6 -- Implementiert einen generischen
 7 -- seriellen, normalisierten Multiplizierer.
 8 --
 9 LIBRARY ieee;
10 USE ieee.std_logic_1164.all;
11 USE ieee.numeric_std.all;
12
13 LIBRARY FIR_lib;
14 USE FIR_lib.FIR_Def_pkg.all;
15
16 ENTITY SerialMultiplierNormalized IS
17  GENERIC(
18   DATA_WIDTH : natural := pcDATA_WIDTH
19  );
20  PORT(
21   Clk   : IN     std_logic;
22   A     : IN     signed ( DATA_WIDTH-1 DOWNTO 0 );
23   Prod  : OUT    signed ( DATA_WIDTH-1 DOWNTO 0 );
24   Rst   : IN     std_logic;
25   B     : IN     signed ( DATA_WIDTH-1 DOWNTO 0 );
26   Start : IN     std_logic;
27   Stop  : IN     std_logic
28  );
29 END SerialMultiplierNormalized ;
30
31 ARCHITECTURE SerialMultiplierNormalized_Archi OF ⤸
32 SerialMultiplierNormalized IS
33  -- Signale:
34  SIGNAL a_std : std_logic_vector(DATA_WIDTH-1 downto 0);
35  SIGNAL b_abs : signed(DATA_WIDTH-1 downto 0);
36  SIGNAL akku  : signed(DATA_WIDTH downto 0);
37  SIGNAL run : bit;
38  -- Vorzeichen des Ausgangssignals:
39  SIGNAL sign    : std_logic; -- 1 -> negativ, sonst positiv
40 BEGIN
41
42 -- Nach Empfang von Start, werden a und b in DATA_WIDTH+1 ⤸
```

```
43 Takten seriell multipliziert
44 -- Stop beendet die Multiplikation und das Ergebnis wird an
45 Prod angelegt.
46
47 SerMult : PROCESS (Clk, Rst) is
48  variable va, vb : std_logic_vector(DATA_WIDTH-1 downto 0);
49 begin
50  if Rst='0' then
51   Prod <= to_signed(0, DATA_WIDTH);
52   run  <= '0';
53   akku <= to_signed(0, DATA_WIDTH+1);
54   sign <= '0';
55   a_std <= (others => '0');
56   b_abs <= (others => '0');
57  elsif Clk'event and Clk='1' then
58   if Start = '1' then
59    run <= '1';
60    a_std <= std_logic_vector(abs(A));
61    b_abs <= abs(B);
62    akku <= to_signed(0, DATA_WIDTH+1);
63    -- Vorzeichen des Ausgangssignals errechnen:
64    va := std_logic_vector(A);
65    vb := std_logic_vector(B);
66    sign <= va(va'high) xor vb(vb'high);
67   end if;
68
69   if Stop = '1' then
70    run <= '0';
71    -- Ergebnis ausgeben:
72    if sign = '1' then
73     -- Wert negieren:
74     Prod <= resize(not akku, Prod'length) + 1;
75    else
76     Prod <= resize(akku, Prod'length);
77    end if;
78   end if;
79   -- Multiplizieren (normalisiert):
80   if run='1' then
81    if a_std(0) = '1' then
82     akku <= shift_right(resize(b_abs, akku'length)+akku, 1);
83    else
84     akku <= shift_right(akku, 1);
85    end if;
86    -- shift a_std:
87    a_std <= '0' & a_std(a_std'high downto 1);
88   end if;
89  end if;
90 end process SerMult;
91 END ARCHITECTURE SerialMultiplierNormalized_Archi;
```

D. Anhang für FIR-Dezimations-Filter

D.1. FIR-Filter mit Hardwaremultipliern

In Listing D.1 wird ein FIR-Filter mit Hardwaremultipliern implementiert.

Der Prozess *InputAndMultiplication* liest Eingangsdaten und speichert diese in Register. Anschließend wird eine Multiplikation mit den jeweiligen Koeffizienten durchgeführt und das Ergebnis wiederum in ein Register geschrieben. Der Prozess *DelayAndAdd* führt die Verzögerung der Multiplikationsergebnisse und die Addition unter Berücksichtigung gerade oder ungerade Koeffizientenanzahl durch. Der Prozess *DecimationAndOutput* gibt die Ergebnisse taktsychron aus und sorgt für die Dezimation des Ausgangssignals.

Listing D.1: VHDL-Implementierung eines FIR-Filters mit Hardwaremultipliern

```
 1 --
 2 -- FIRLinPhaseFast_Archi.vhd
 3 --
 4 -- Frederik Teichert
 5 --
 6 -- Implementiert ein FIR-Filter mit linearer
 7 -- Phase und Hardwaremultipliern.
 8 --
 9 LIBRARY ieee;
10 USE ieee.std_logic_1164.all;
11 USE ieee.numeric_std.all;
12 USE ieee.math_real.all;
13 LIBRARY FIR_lib;
14 USE FIR_lib.FIR_Def_pkg.all;
15 USE FIR_lib.FIR_Coeff_def.all;
16
17 ENTITY FIRLinPhaseFast IS
18  GENERIC(
19   DATA_WIDTH        : natural := 20;
20   TAPS_WIDTH        : natural := 20;
21   OUT_WIDTH         : natural := 20;
22   DECIMATION_FACTOR : natural := 2;
23   COEFFS     : T_Coeff (0 to pcMAX_COEFFS_AMOUNT-1) :=
24                         pcFIR_COEFFS_291;
25   TAPS_AMOUNT       : natural := 291
26  );
27  PORT(
28   Clk    : IN     std_logic;
29   Din    : IN     signed ( DATA_WIDTH DOWNTO 0 );
```

```
30   DOut  : OUT    signed ( DATA_WIDTH DOWNTO 0 );
31   Rst   : IN     std_logic;
32   Dav   : OUT    std_logic;
33   ClkEn : IN     std_logic
34  );
35 END ENTITY FIRLinPhaseFast;
36
37 --
38 ARCHITECTURE FIRLinPhaseFast_Archi OF FIRLinPhaseFast IS
39  --delays:
40  CONSTANT cMultiplierCount : natural := (TAPS_AMOUNT+1)/2;
41  TYPE   T_delays is array(natural range <>) of signed(
42                             DATA_WIDTH downto 0);
43  -- adddelays: Pro Addtion ein Bit mehr und ein weiteres Bit
44  für den Überlauf
45  -- einer Sprungantwort!
46  TYPE   T_adddelays is array(natural range <>) of signed(
47                                DATA_WIDTH+TAPS_AMOUNT downto 0);
48
49  SIGNAL inputdelays : T_delays(0 to cMultiplierCount-1);
50  SIGNAL multdelays  : T_delays(0 to cMultiplierCount-1);
51  SIGNAL adddelays   : T_adddelays(0 to TAPS_AMOUNT-1);
52  -- counter:
53  SIGNAL counter : std_logic_vector(DECIMATION_FACTOR-1 downto
54                                      0);
55 BEGIN
56
57 InputAndMultiplication : PROCESS (Clk, Rst, ClkEn) is
58 begin
59  if Rst='0' then
60   inputdelays <= (others =>
61                    to_signed(0, inputdelays(0)'length));
62   multdelays <= (others =>
63                   to_signed(0, multdelays(0)'length));
64  elsif Clk'event and Clk='1' then
65   if ClkEn='1' then
66    -- Eingang lesen:
67    inputdelays <= (others => DIn);
68
69    -- Multiplikation:
70    for i in inputdelays'range loop
71     multdelays(i) <= resize(
72          shift_right(
73           inputdelays(i) *
74            to_signed(integer(COEFFS(multdelays'high-i)*real(
75                      2**(TAPS_WIDTH-1)-1)), TAPS_WIDTH)
76          , TAPS_WIDTH-1)
77         , multdelays(0)'length);
78    end loop;
79   end if;
80  end if;
```

```
 81 end PROCESS InputAndMultiplication;
 82
 83 DelayAndAdd : PROCESS (Clk, Rst, ClkEn) is
 84 begin
 85  if Rst='0' then
 86   adddelays <= (others => to_signed(0, adddelays(0)'length));
 87  elsif Clk'event and Clk='1' then
 88   if ClkEn='1' then
 89    -- Unterer Zweig:
 90    adddelays(0) <= resize(multdelays(multdelays'high),
 91                 adddelays(0)'length);
 92    for i in 1 to cMultiplierCount-2 loop
 93     adddelays(i) <= multdelays(multdelays'high-i) + adddelays(
 94                  i-1);
 95    end loop;
 96
 97    -- Oberer Zweig
 98    for i in 0 to cMultiplierCount-2 loop
 99     adddelays(adddelays'high-i) <= adddelays(adddelays'high-i-
100                  1) + multdelays(multdelays'high-i);
101    end loop;
102
103    adddelays(cMultiplierCount-1) <= multdelays(0) + adddelays(
104                 (cMultiplierCount-1)-1);
105   end if;
106  end if;
107 end PROCESS DelayAndAdd;
108
109 DecimationAndOutput : PROCESS (Clk, Rst, ClkEn) is
110 begin
111  if Rst='0' then
112   counter     <= (others => '0');
113   counter(0)  <= '1';
114   Dav         <= '0';
115   Dout        <= to_signed(0, Dout'length);
116  elsif Clk'event and Clk='1' then
117   Dav <= '0';
118   if ClkEn='1' then
119    counter <= counter(counter'high-1 downto 0) & counter(
120                counter'high);
121    if counter(0) = '1' then
122     Dav <= '1';
123     Dout <= resize(adddelays(adddelays'high), Dout'length);
124    end if;
125   end if;
126  end if;
127 end PROCESS DecimationAndOutput;
128
129 END ARCHITECTURE FIRLinPhaseFast_Archi;
```

D.2. FIR-Filter mit Booth-4-Multipliern

In Listing D.2 wird ein FIR-Filter mit Booth-4-Multipliern implementiert.

Die Implementierung mit Booth-Multiplizierern unterscheidet sich in dem Prozess *InputAndMultiplicationCounter*. Dort wird ein One-Hot-Counter gesteuert, der das Timing der Booth-Multiplizierer steuert.

Da sich die Implementierung mit seriellen Multiplizierern nur geringfügig von der Implementierung mit Booth-4-Multipliern unterscheidet, wird von einem Listing abgesehen und auf die CD im Anhang verwiesen.

Listing D.2: VHDL-Implementierung eines FIR-Filters mit Booth-Multipliern

```
1  --
2  -- FIRLinPhaseBooth_Archi.vhd
3  --
4  -- Frederik Teichert
5  --
6  -- Implementiert ein FIR-Filter mit linearer
7  -- Phase und Booth-4-Multipliern.
8  --
9  LIBRARY ieee;
10 USE ieee.std_logic_1164.all;
11 USE ieee.numeric_std.all;
12 USE ieee.math_real.all;
13
14 LIBRARY FIR_lib;
15 USE FIR_lib.FIR_Def_pkg.all;
16 USE FIR_lib.FIR_Coeff_def.all;
17
18 ENTITY FIRLinPhaseBooth IS
19  GENERIC(
20   DATA_WIDTH        : natural := 20; -- pcDATA_WIDTH;
21   TAPS_WIDTH        : natural := 20; --pcTAPS_WIDTH;
22   OUT_WIDTH         : natural := 20; --pcOUT_WIDTH;
23   DECIMATION_FACTOR : natural := 2;
24   COEFFS      : T_Coeff (0 to pcMAX_COEFFS_AMOUNT-1) :=
25                          pcFIR_COEFFS_291;
26   TAPS_AMOUNT       : natural := 291 -- pcMAX_TAPS_AMOUNT
27  );
28  PORT(
29   Clk    : IN     std_logic;
30   Din    : IN     signed ( DATA_WIDTH DOWNTO 0 );
31   DOut   : OUT    signed ( DATA_WIDTH DOWNTO 0 );
32   Rst    : IN     std_logic;
33   Dav    : OUT    std_logic;
34   ClkEn  : IN     std_logic
35  );
36 END ENTITY FIRLinPhaseBooth;
37
```

```
38 --
39 ARCHITECTURE FIRLinPhaseBooth_Archi OF FIRLinPhaseBooth IS
40  --delays:
41  CONSTANT cMultiplierCount : natural := (TAPS_AMOUNT+1)/2;
42  TYPE    T_delays is array(natural range <>) of signed(⤸
43                            DATA_WIDTH downto 0);
44  -- adddelays: Pro Addtion ein Bit mehr und ein weiteres Bit⤸
45  für den Überlauf
46  -- einer Sprungantwort!
47  TYPE    T_adddelays is array(natural range <>) of signed(⤸
48                               DATA_WIDTH+TAPS_AMOUNT downto 0);
49  --SIGNAL inputdelays : T_delays(0 to TAPS_AMOUNT-1);
50  TYPE    T_multdelays is array(natural range <>) of signed(⤸
51                                DATA_WIDTH downto 0);
52  SIGNAL multdelays  : T_delays(0 to cMultiplierCount-1);
53  SIGNAL adddelays   : T_adddelays(0 to TAPS_AMOUNT-1);
54  -- counter:
55  SIGNAL counter : std_logic_vector(DECIMATION_FACTOR-1 downto⤸
56                                    0);
57
58  COMPONENT Booth4MultNormalized
59   GENERIC(
60    GAIN_BITS  : natural := 0;
61    DATA_WIDTH : natural := pcDATA_WIDTH
62   );
63   PORT(
64    Clk   : IN     std_logic;
65    A     : IN     signed ( DATA_WIDTH-1 DOWNTO 0 );
66    Prod  : OUT    signed ( DATA_WIDTH-1 DOWNTO 0 );
67    Rst   : IN     std_logic;
68    B     : IN     signed ( DATA_WIDTH-1 DOWNTO 0 );
69    Start : IN     std_logic;
70    Stop  : IN     std_logic
71   );
72  end COMPONENT;
73
74  -- counter:
75  CONSTANT cBoothClks : natural := natural(ceil(real(⤸
76                                   DATA_WIDTH+2)/3.0))+1;
77  SIGNAL boothcounter : std_logic_vector(cBoothClks+3 downto 0)⤸
78                                       ;
79 BEGIN
80
81 InputAndMultiplicationCounter : PROCESS (Clk, Rst, ClkEn) is
82 begin
83  if Rst='0' then
84   boothcounter <= (others => '0');
85  elsif Clk'event and Clk='1' then
86   -- count the one-hot-counter:
87   boothcounter <= boothcounter(boothcounter'high-1 downto 0)⤸
88                 & '0';
```

```
89    if ClkEn='1' then
90     boothcounter <= (others => '0');
91     boothcounter(0) <='1';
92    end if;
93   end if;
94  end PROCESS InputAndMultiplicationCounter;
95
96  DelayAndAdd : PROCESS (Clk, Rst, ClkEn) is
97  begin
98   if Rst='0' then
99    adddelays <= (others => to_signed(0, adddelays(0)'length));
100  elsif Clk'event and Clk='1' then
101   if ClkEn='1' then
102    -- Unterer Zweig:
103    adddelays(0) <= resize(multdelays(multdelays'high),
104              adddelays(0)'length);
105    for i in 1 to cMultiplierCount-2 loop
106     adddelays(i) <= multdelays(multdelays'high-i) + adddelays(
107               i-1);
108    end loop;
109
110    -- Oberer Zweig
111    for i in 0 to cMultiplierCount-2 loop
112     adddelays(adddelays'high-i) <= adddelays(adddelays'high-i-
113               1) + multdelays(multdelays'high-i);
114    end loop;
115
116    adddelays(cMultiplierCount-1) <= multdelays(0) + adddelays(
117              (cMultiplierCount-1)-1);
118   end if;
119  end if;
120 end PROCESS DelayAndAdd;
121
122 DecimationAndOutput : PROCESS (Clk, Rst, ClkEn) is
123 begin
124  if Rst='0' then
125   counter    <= (others => '0');
126   counter(0) <= '1';
127   Dav        <= '0';
128   Dout       <= to_signed(0, Dout'length);
129  elsif Clk'event and Clk='1' then
130   Dav <= '0';
131   if ClkEn='1' then
132    counter <= counter(counter'high-1 downto 0) & counter(
133              counter'high);
134    if counter(0) = '1' then
135     Dav <= '1';
136     Dout <= resize(adddelays(adddelays'high), Dout'length);
137    end if;
138   end if;
139  end if;
```

```
140 end PROCESS DecimationAndOutput;
141
142 Multiplier: for i in multdelays'range generate
143  Mult: Booth4MultNormalized  generic map (1, DATA_WIDTH+1) --
144                                                      gain 2^2!
145            port map (Clk => Clk, Rst => Rst,
146             A => to_signed(integer(COEFFS(multdelays'high-i)
147              *real(2**(DATA_WIDTH-1)-2)), DATA_WIDTH+1),
148             B=> Din,
149             Prod => multdelays(i),
150             Start => boothcounter(0), Stop => boothcounter(
151                    cBoothClks));
152  end generate;
153 END ARCHITECTURE FIRLinPhaseBooth_Archi;
```

E. Anhang für FIR-Halfband-Filter

Die folgenden Implementierungen von FIR-Halfbandfiltern nutzen die selben Komponenten wie die vorherigen FIR-Filter, um eine Multiplikation durchzuführen.

E.1. FIR-Filter mit Hardware-Multiplizierern

In Listing E.1 wird eine Implementierung eines FIR-Halfbandfilters in Direktform mit Hardwaremultipliern vorgestellt.

Der Prozess *DelayAndCount* führt die Verzögerung der Eingangssignale durch und realisiert einen One-Hot-Counter. Der Prozess *Symmetry* addiert die verzögerten Eingangssignale unter Ausnutzung der Symmetrie der Koeffizienten. *Multiply* führt eine Multiplikation mit den Koeffizienten durch, wobei die Symmetrie und die nicht-vorhandenen Koeffizienten berücksichtigt werden. Der Prozess *AdderTree* addiert sukzessiv alle Multiplikationsergebnisse mit einer Baumstruktur.

Listing E.1: VHDL-Implementierung eines FIR-Halfband-Filters mit Hardware-Multipliern

```
 1 --
 2 -- FIRHalfBandFast_Archi.vhd
 3 --
 4 -- Frederik Teichert
 5 --
 6 -- Implementiert ein FIR-Halfband-Filter
 7 -- mit linearer Phase mit Hardware-Multipliern.
 8 --
 9 LIBRARY ieee;
10 USE ieee.std_logic_1164.all;
11 USE ieee.numeric_std.all;
12 USE ieee.math_real.all;
13
14 LIBRARY FIR_lib;
15 USE FIR_lib.FIR_Def_pkg.all;
16 USE FIR_lib.FIR_Coeff_def.all;
17
18 ENTITY FIRHalfBandFast IS
19  GENERIC(
20   DATA_WIDTH         : natural := pcDATA_WIDTH;
21   TAPS_WIDTH         : natural := pcTAPS_WIDTH;
22   OUT_WIDTH          : natural := pcOUT_WIDTH;
23   COEFFS             : T_Coeff (0 to pcMAX_COEFFS_AMOUNT-1)
24                                := pcFIR_COEFFS_27;
25   TAPS_AMOUNT        : natural := pcMAX_TAPS_AMOUNT
```

```
26  );
27  PORT(
28   Clk    : IN     std_logic;
29   Din    : IN     signed ( DATA_WIDTH DOWNTO 0 );
30   DOut   : OUT    signed ( DATA_WIDTH DOWNTO 0 );
31   Rst    : IN     std_logic;
32   Dav    : OUT    std_logic;
33   ClkEn : IN     std_logic );
34 END ENTITY FIRHalfBandFast;
35
36 ARCHITECTURE FIRHalfBandFast_Archi OF FIRHalfBandFast IS
37  -- Delays:
38  TYPE   T_delays is array(natural range <>) of signed(⤸
39                           DATA_WIDTH downto 0);
40  SIGNAL delays : T_delays(0 to TAPS_AMOUNT-1);
41  SIGNAL syst   : bit;
42
43  CONSTANT cMultiplierCount : natural := (TAPS_AMOUNT+1)/4;
44
45  TYPE   T_products is array(natural range <>) of signed(⤸
46                             DATA_WIDTH+1 downto 0);
47  -- Ein Bit mehr als die Eingänge:
48
49  SIGNAL products   : T_products(0 to cMultiplierCount);
50
51  -- Synthesis directives
52  attribute multstyle : string;
53  attribute multstyle of products : signal is "dsp";
54
55  -- Signale:
56  SIGNAL symbuffer : T_products(0 to cMultiplierCount);
57  SIGNAL multin     : T_products(0 to (cMultipliercount+1)/2);
58  SIGNAL multout     : T_products(0 to (cMultipliercount+1)/2);
59
60  -- One-Hot-Zähler:
61  SIGNAL counter : std_logic_vector(1 downto 0);
62
63  -- Die Funktion errechnet die Anzahl der
64  -- Register für den Addiererbaum:
65  function getTreeAdderCount (N : natural) return natural is
66   variable ret : natural := 0;
67  begin
68   -- \sum_{i=0}^ceil(ld(N)) 2^i
69   for i in 0 to integer(ceil(log2(real(N)))) loop
70    ret := ret + 2**i;
71   end loop;
72   return ret;
73  end getTreeAdderCount;
74
75  -- Addierbaum:
76  CONSTANT cAdderCount : natural := getTreeAdderCount((⤸
```

```
77                                      cMultiplierCount+1)/2);
78   CONSTANT cAdderSteps : natural := natural(ceil(log2(real(⤸
79                                      cMultiplierCount+1))));
80
81   TYPE T_adderbuf is array(natural range <>) of signed(⤸
82                              DATA_WIDTH+cAdderSteps downto 0);
83   SIGNAL adderbuf : T_adderbuf(cAdderCount-1 downto 0);
84  BEGIN
85
86  DelayAndCount : PROCESS (Clk, Rst, ClkEn) is
87  begin
88   if Rst='0' then
89    delays <= (others => to_signed(0, delays(0)'length));
90    syst   <= '0';
91    counter  <= (others => '0');
92   elsif Clk'event and Clk='1' then
93    -- count:
94    counter <= counter(counter'high-1 downto 0) & '0';
95    if ClkEn='1' then
96     -- clock the delays:
97     delays <= Din & delays(delays'low to delays'high-1);
98     syst <= not syst; -- systolic ..
99     if syst = '0' then
100     -- restart counter:
101     counter <= (others => '0');
102     counter(0) <= '1';
103    end if;
104   end if;
105  end if;
106 end process SOP;
107
108 Symmetry : PROCESS (Clk, Rst, counter) is
109 begin
110  -- Addiert die symmetrischen Werte und speichert sie in⤸
111  symbuffer:
112  if Rst='0' then
113   symbuffer <= (others => to_signed(0, symbuffer(0)'length));
114  elsif Clk'event and Clk='1' then
115    if counter(counter'high-1) = '1' then
116     for i in 0 to cMultiplierCount-1 loop
117      symbuffer(i) <= resize(delays(2*i), symbuffer(0)'length)
118                   + resize(delays(delays'high-2*i),⤸
119                             symbuffer(0)'length);
120     end loop;
121     -- Mittleren Wert zwischenspeichern:
122     symbuffer(symbuffer'high) <= resize(delays(delays'high/2),
123                                  symbuffer(0)'length);
124    end if;
125  end if;
126 end PROCESS Symmetry;
127
```

```
128 Multiply : PROCESS (Clk, Rst, symbuffer, counter) is
129  VARIABLE vTmp     : signed(DATA_WIDTH+2 downto 0);
130  VARIABLE vTmpLong : signed(2*DATA_WIDTH+1 downto 0);
131 begin
132  if Rst='0' then
133   products <= (others => to_signed(0, products(0)'length));
134  elsif Clk'event and Clk='1' then
135   if counter(counter'high-1) = '1' then
136    for i in 0 to (products'high-1)/2 loop
137     products(2*i) <= resize(shift_right(
138        symbuffer(2*i)*to_signed(integer(round(COEFFS(4*i)
139         *real(2**(DATA_WIDTH-1)-1))), DATA_WIDTH+1),
140         DATA_WIDTH-1), products(0)'length);
141    end loop;
142   elsif counter(counter'high) = '1' then
143    for i in 0 to (products'high-1)/2 loop
144     products(2*i+1) <= resize(shift_right(
145        symbuffer(2*i+1)*to_signed(integer(round(COEFFS(4*i+2)
146         *real(2**(DATA_WIDTH-1)-1))), DATA_WIDTH+1),
147         DATA_WIDTH-1), products(0)'length);
148    end loop;
149    -- mittelkoeffizienent:
150     products(products'high) <= resize(shift_right(
151         symbuffer(symbuffer'high)
152          *to_signed(integer(round(COEFFS(2*products'high-1)
153           *real(2**(DATA_WIDTH-1)-1))), DATA_WIDTH+1),
154         DATA_WIDTH-1), products(0)'length);
155   end if;
156  end if;
157 end process Multiply;
158
159 AdderTree : PROCESS (Clk, Rst, counter) is
160  variable vIndex : integer := 0;
161 begin
162  if Rst='0' then
163   adderbuf <= (others => to_signed(0, adderbuf(0)'length));
164   Dout <= to_signed(0, Dout'length);
165  elsif Clk'event and Clk='1' then
166   -- addieren:
167   for i in 0 to cAdderSteps-2 loop
168    for k in 0 to 2**i-1 loop
169     if syst='0' then
170      adderbuf(2**i+k-1) <= adderbuf(2**(i+1)+2*k-1) +⤸
171               adderbuf(2**(i+1)+2*k);
172     end if;
173    end loop;
174   end loop;
175
176   -- produkte laden:
177   for k in 0 to 2**(cAdderSteps-1)-1 loop
178    if counter(counter'high) = '1' then
```

```
179     if 2*k+1 <= (products'high) then -- Alle Eingänge werden ↩
180                 verbunden.
181      adderbuf(2**(cAdderSteps-1)+k-1) <= resize(products(2*k), ↩
182               adderbuf(0)'length)
183                + resize(products(2*k+1), adderbuf(0)'length);
184     elsif 2*k = (products'high) then -- Linken verbinden:
185      adderbuf(2**(cAdderSteps-1)+k-1) <= resize(products(2*k), ↩
186               adderbuf(0)'length);
187     else
188      adderbuf(2**(cAdderSteps-1)+k-1) <= to_signed(0,
189       adderbuf(0)'length);
190     end if;
191    end if;
192   end loop;
193  end if;
194
195  Dav <= counter(counter'high);
196  Dout <= resize(adderbuf(0), Dout'length);
197 end process AdderTree;
198 END ARCHITECTURE FIRHalfBandFast_Archi;
```

E.2. FIR-Filter mit seriellen Multiplizierern

Anstelle der Implementierung mit Booth-4-Multipliern wird an dieser Stelle die Implementierung eines FIR-Halfbandfilters mit seriellen Multiplizierern vorgestellt. Die Implementierung der Filter mit Booth-4-Multipliern kann auf der CD eingesehen werden.

Listing E.2: VHDL-Implementierung eines FIR-Halfband-Filters mit seriellen Multipliern

```
 1 --
 2 -- FIRHalfBandBooth_Archi.vhd
 3 --
 4 -- Frederik Teichert
 5 --
 6 -- Implementiert ein FIR-Halfband-Filter
 7 -- mit linearer Phase mit seriellen Multipliern.
 8 --
 9 LIBRARY ieee;
10 USE ieee.std_logic_1164.all;
11 USE ieee.numeric_std.all;
12 USE ieee.math_real.all;
13
14 LIBRARY FIR_lib;
15 USE FIR_lib.FIR_Def_pkg.all;
16 USE FIR_lib.FIR_Coeff_def.all;
17
18 ENTITY FIRHalfBandSer IS
19  GENERIC(
20   DATA_WIDTH        : natural := pcDATA_WIDTH;
21   TAPS_WIDTH        : natural := pcTAPS_WIDTH;
22   OUT_WIDTH         : natural := pcOUT_WIDTH;
23   COEFFS            : T_Coeff (0 to pcMAX_COEFFS_AMOUNT-1)
24                               := pcFIR_COEFFS_27;
25   TAPS_AMOUNT       : natural := pcMAX_TAPS_AMOUNT
26  );
27  PORT(
28   Clk   : IN     std_logic;
29   Din   : IN     signed ( DATA_WIDTH DOWNTO 0 );
30   DOut  : OUT    signed ( DATA_WIDTH DOWNTO 0 );
31   Rst   : IN     std_logic;
32   Dav   : OUT    std_logic;
33   ClkEn : IN     std_logic );
34 END FIRHalfBandSer ;
35
36 --
37 ARCHITECTURE FIRHalfBandSer OF FIRHalfBandSer IS
38  -- Delays:
39  TYPE    T_delays is array(natural range <>) of signed(⤸
40                           DATA_WIDTH downto 0);
41  SIGNAL delays : T_delays(0 to TAPS_AMOUNT-1);
42  SIGNAL syst   : bit;
```

```
43
44  CONSTANT cMultiplierCount : natural := (TAPS_AMOUNT+1)/4;
45
46  TYPE   T_products is array(natural range <>) of signed(↩
47                              DATA_WIDTH+1 downto 0);
48  -- Ein Bit mehr als die Eingänge:
49
50  SIGNAL products   : T_products(0 to cMultiplierCount);
51
52  -- Synthesis directives
53  attribute multstyle : string;
54  attribute multstyle of products : signal is "dsp";
55
56  -- Signale:
57  SIGNAL symbuffer : T_products(0 to cMultiplierCount);
58  SIGNAL multin     : T_products(0 to (cMultipliercount+1)/2);
59  SIGNAL multout     : T_products(0 to (cMultipliercount+1)/2);
60
61  -- One-Hot-Zähler:
62  SIGNAL counter : std_logic_vector(1 downto 0);
63
64  -- Die Funktion errechnet die Anzahl der
65  -- Register für den Addiererbaum:
66  function getTreeAdderCount (N : natural) return natural is
67   variable ret : natural := 0;
68  begin
69   -- \sum_{i=0}^ceil(ld(N)) 2^i
70   for i in 0 to integer(ceil(log2(real(N)))) loop
71    ret := ret + 2**i;
72   end loop;
73   return ret;
74  end getTreeAdderCount;
75
76  -- Addierbaum:
77  CONSTANT cAdderCount : natural := getTreeAdderCount((↩
78                                    cMultiplierCount+1)/2);
79  CONSTANT cAdderSteps : natural := natural(ceil(log2(real(↩
80                                    cMultiplierCount+1))));
81
82  TYPE T_adderbuf is array(natural range <>) of signed(↩
83                            DATA_WIDTH+cAdderSteps downto 0);
84  SIGNAL adderbuf : T_adderbuf(cAdderCount-1 downto 0);
85
86  COMPONENT SerialMultiplierNormalized
87   GENERIC(
88    DATA_WIDTH : natural := pcDATA_WIDTH
89   );
90   PORT(
91    Clk    : IN     std_logic;
92    A      : IN     signed ( DATA_WIDTH-1 DOWNTO 0 );
93    Prod   : OUT    signed ( DATA_WIDTH-1 DOWNTO 0 );
```

```
 94    Rst   : IN     std_logic;
 95    B     : IN     signed ( DATA_WIDTH-1 DOWNTO 0 );
 96    Start : IN     std_logic;
 97    Stop  : IN     std_logic
 98  );
 99  end COMPONENT;
100 BEGIN
101
102 DelayAndCount : PROCESS (Clk, Rst, ClkEn) is
103 begin
104  if Rst='0' then
105   delays <= (others => to_signed(0, delays(0)'length));
106   syst    <= '0';
107   counter  <= (others => '0');
108  elsif Clk'event and Clk='1' then
109   -- count:
110   counter <= counter(counter'high-1 downto 0) & '0';
111
112   if ClkEn='1' then
113    -- clock the delays:
114    delays <= Din & delays(delays'low to delays'high-1);
115    syst <= not syst; -- systolic ..
116    if syst = '0' then
117     -- restart counter:
118     counter <= (others => '0');
119     counter(0) <= '1';
120    end if;
121   end if;
122  end if;
123 end process SOP;
124
125 Symmetry : PROCESS (Clk, Rst, counter, ClkEn) is
126 begin
127  -- Addiert die symmetrischen Werte und speichert sie in
128  symbuffer:
129  if Rst='0' then
130   symbuffer <= (others => to_signed(0, symbuffer(0)'length));
131  elsif Clk'event and Clk='1' then
132   if ClkEn='1' then
133    if syst = '0' then
134     for i in 0 to cMultiplierCount-1 loop
135      symbuffer(i) <= resize(delays(2*i), symbuffer(0)'length)
136                   + resize(delays(delays'high-2*i),
137                              symbuffer(0)'length);
138     end loop;
139     -- Mittleren Wert zwischenspeichern:
140     symbuffer(symbuffer'high) <= resize(delays(delays'high/2),
141                                     symbuffer(0)'length);
142    end if;
143   end if;
144  end if;
```

```
145 end PROCESS Symmetry;
146
147 AdderTree : PROCESS (Clk, Rst, counter) is
148  variable vIndex : integer := 0;
149 begin
150  if Rst='0' then
151   adderbuf <= (others => to_signed(0, adderbuf(0)'length));
152   Dout <= to_signed(0, Dout'length);
153  elsif Clk'event and Clk='1' then
154   -- addieren:
155   for i in 0 to cAdderSteps-2 loop
156    for k in 0 to 2**i-1 loop
157     if counter(1) = '1' then
158      adderbuf(2**i+k-1) <= adderbuf(2**(i+1)+2*k-1) +
159               adderbuf(2**(i+1)+2*k);
160     end if;
161    end loop;
162   end loop;
163
164   -- Produkte laden:
165   for k in 0 to 2**(cAdderSteps-1)-1 loop
166    if counter(counter'high-1) = '1' then
167     if 2*k+1 <= (products'high) then -- Alle Eingänge werden
168                 verbunden.
169      adderbuf(2**(cAdderSteps-1)+k-1) <= resize(products(2*k),
170               adderbuf(0)'length)
171                + resize(products(2*k+1), adderbuf(0)'length);
172     elsif 2*k = (products'high) then -- Linken verbinden:
173      adderbuf(2**(cAdderSteps-1)+k-1) <= resize(products(2*k),
174               adderbuf(0)'length);
175     else
176      adderbuf(2**(cAdderSteps-1)+k-1) <= to_signed(0,
177       adderbuf(0)'length);
178     end if;
179    end if;
180   end loop;
181  end if;
182
183  Dav <= counter(counter'high);
184  Dout <= resize(adderbuf(0), Dout'length);
185 end process AdderTree;
186
187 -- Multiplizierer:
188 Multiplier: for i in 0 to products'high-1 generate
189  Mult: SerialMultiplierNormalizedgeneric
190  generic map (2, DATA_WIDTH+2) -- gain 2^1!
191  port map (
192   Clk    => Clk,
193   Rst    => Rst,
194   A      => to_signed(integer(COEFFS(2*i)*real(2**(DATA_WIDTH-
195          1)-1)), DATA_WIDTH+2),
```

```
196    B      => symbuffer(i),
197    Prod   => products(i),
198    Start => counter(0),
199    Stop   => counter(cBoothClks)
200   );
201   end generate;
202  MiddleMult: SerialMultiplierNormalized
203   generic map (2, DATA_WIDTH+2) -- gain 2^1!
204   port map (
205    Clk    => Clk,
206    Rst    => Rst,
207    A      => to_signed(integer(COEFFS(2*products'high-1)*real(
208            2**(DATA_WIDTH-1)-1)),DATA_WIDTH+2),
209    B      => symbuffer(symbuffer'high),
210    Prod   => products(products'high),
211    Start => counter(0),
212    Stop   => counter(cBoothClks)
213   );
214  END ARCHITECTURE FIRHalfBandSer;
```

F. Anhang für Cascaded Integrator Comb

F.1. Herleitung des Amplitudengangs eines CIC-Filters

Da die Herleitung des Amplitudengangs in [CIC] fehlt, wird diese der Vollständigkeit halber an dieser Stelle vorgenommen.

Der Amplitudengang eines CIC-Filters ergibt sich aus Gleichung (6.3), wenn

$$z = e^{j(2\pi f/R)} \tag{F.1}$$

gesetzt wird und der Betrag gebildet wird.

$$H(z) = \frac{(1 - z^{-RM})^N}{(1 - z^{-1})^N} \tag{F.2}$$

$$\left|H(e^{j(2\pi f/R)})\right| = \frac{\left|\left(1 - e^{j(2\pi fM)}\right)^N\right|}{\left|\left(1 - e^{j(2\pi f/R)}\right)^N\right|} \tag{F.3}$$

$$= \frac{\left|1 - e^{j(2\pi fM)}\right|^N}{\left|1 - e^{j(2\pi f/R)}\right|^N} \tag{F.4}$$

$$= \frac{\left|1 - (cos\ 2\pi fM - jsin\ 2\pi fM)\right|^N}{\left|1 - (cos\ 2\pi f/R - jsin\ 2\pi f/R)\right|^N} \tag{F.5}$$

$$= \frac{\left|\sqrt{(1 - cos\ 2\pi fM)^2 + sin^2 2\pi fM}\right|^N}{\left|\sqrt{(1 - cos\ 2\pi f/R)^2 + sin^2 2\pi f/R}\right|^N} \tag{F.6}$$

$$= \frac{\left|\sqrt{1 - 2cos\ 2\pi fM + cos^2 2\pi fM + sin^2 2\pi fM}\right|^N}{\left|\sqrt{1 - 2cos\ 2\pi f/R + cos^2 2\pi f/R + sin^2 2\pi f/R}\right|^N} \tag{F.7}$$

Mit

$$sin^2 x + cos^2 x = 1 \tag{F.8}$$

und

$$2\ sin^2 \frac{x}{2} = 1 - cos(x) \tag{F.9}$$

vereinfacht sich der Ausdruck erheblich zu

$$= \frac{\left|\sqrt{2(1 - cos(2\pi f M))}\right|^N}{\left|\sqrt{2(1 - cos(2\pi f/R))}\right|^N} \tag{F.10}$$

$$= \frac{\left|\sqrt{2(2sin^2(\pi f M))}\right|^N}{\left|\sqrt{2(2sin^2(\pi f/R))}\right|^N} \tag{F.11}$$

$$= \frac{\left|\sqrt{sin^2(\pi f M)}\right|^N}{\left|\sqrt{sin^2(\pi f/R)}\right|^N} \tag{F.12}$$

$$= \frac{|sin(\pi f M)|^N}{|sin(\pi f/R)|^N}. \tag{F.13}$$

Der Amplitudengang ergibt sich zu

$$|H(f)| = \left|\frac{sin \pi M f}{sin \frac{\pi f}{R}}\right|^N. \tag{F.14}$$

Für große Dezimationsfaktoren kann der Amplitudengang für einen beschränkten Frequenzbereich angenähert werden.

$$|H(f)| = \left|RM \frac{sin \pi M f}{\pi M f}\right|^N \qquad \text{für} \quad 0 \leq f \leq \frac{1}{M} \tag{F.15}$$

F.2. Herleitung der maximalen Verstärkung eines CIC-Dezimators

Die Registerdimensionierung stellt ein zentrales Problem für die Implementierung eines CIC-Dezimators dar, da bedingt durch die Integratoren eine Verstärkung stattfindet.

Um die Verstärkung zu berechnen, wird zunächst die Übertragungsfunktion von der j-ten bis einschließlich $2N$-ten Stufe gebildet. Hierzu ist zu unterscheiden, ob es sich um die Integrator- bzw. Differentiator-Stufe handelt. Die Herleitung wurde aus [CIC] entnommen und ausführlicher dargestellt.

Zunächst wird Fall 1 gelöst, Integratoren und Differentiatoren.

$$H_j(z) = H_I^{N-j+1} H_C^N, \qquad j = 1, 2, \ldots, N \tag{F.16}$$

Eingesetzt ergibt sich

$$H_j(z) = \frac{(1 - z^{-RM})^N}{(1 - z^{-1})^{N-j+1}}. \tag{F.17}$$

Wenn anschließend der Zähler mit 1 so erweitert wird, dass ein Faktor entsteht, der denselben Exponenten wie der Nenner hat, ergibt sich

$$H_j(z) = \frac{(1 - z^{-RM})^{N-j+1}(1 - z^{-RM})^{j-1}}{(1 - z^{-1})^{N-j+1}}. \tag{F.18}$$

Zusammengefasst ergibt sich dann

$$H_j(z) = \frac{(1 - z^{-RM})^{j-1}}{\left(\frac{1 - z^{-1}}{1 - z^{-RM}}\right)^{-(N-j+1)}}. \tag{F.19}$$

In einer anderen Schreibweise des entstandenen Nenners ist der Ausdruck dann

$$H_j(z) = (1 - z^{-RM})^{j-1} \left[\sum_{k=0}^{RM-1} z^{-k}\right]^{N-j+1} = \sum_{k=0}^{(RM-1)N+j-1} h_j(k) z^{-k}, \tag{F.20}$$

wobei die $h_j(k)$ die Polynomkoeffizienten der Impulsantworten darstellen.

Eine alternative Darstellung von (F.16) ist

$$H_j(z) = \left[\sum_{l=0}^{N} (-1)^l \binom{N}{l} z^{-RMl}\right] \cdot \left[\sum_{v=0}^{\infty} \binom{N-j+v}{v} z^{-v}\right] \tag{F.21}$$

$$= \sum_{l=0}^{N} \sum_{v=0}^{\infty} (-1)^l \binom{N}{l} \binom{N-j+v}{v} z^{-RMl+v.} \tag{F.22}$$

Werden $k = RMl + v$ und der Wertebereich für $l = 0, 1, \ldots \lfloor k/RM \rfloor$ gesetzt, ergibt sich

$$H_j(z) = \sum_{k \geq 0} \left[\sum_{l=0}^{\lfloor k/RM \rfloor} (-1)^l \binom{N}{l} \binom{N-j+k-RMl}{k-RMl}\right] z^{-k}, \qquad j = 1, 2, \ldots, N. \tag{F.23}$$

Die Darstellung ähnelt der Darstellung in (F.20). Anschließend wird der zweite, einfachere Fall gelöst, nur Differentiatoren.

$$H_j(z) = H_C^{2N+1-j} \tag{F.24}$$

$$= (1 - z^{-RM})^{2N+1-j}, \qquad N+1, N+2, \ldots, 2N. \tag{F.25}$$

Wird wieder die binominale Erweiterung vorgenommen, resultiert das in

$$H_j(z) = \begin{cases} H_I^{N-j+1} + H_C^N = \sum_{k=0}^{(RM-1)N+1-1} h_j(k) z^{-k}, & j = 1, 2, \ldots, N \\ H_C^{j-N} = \sum_{k=0}^{1N+1-j} h_j(k) z^{-kRM}, & j = N+1, N+2, \ldots, 2N \end{cases}, \tag{F.26}$$

wobei $h_j(k)$ die Koeffizienten der Impulsantwort darstellen [CIC].

$$h_j(k) = \begin{cases} \sum_{l=0}^{\lfloor k/RM \rfloor} (-1)^l \binom{N}{l} \binom{N-j+k-RMl}{k-RMl}, & j = 1, 2, \ldots, N \\ (-1)^k \binom{2N+1-j}{k}, & j = N+1, N+2, \ldots, 2N \end{cases} \tag{F.27}$$

Das maximale Registerwachsen G_{max} wird durch die maximale Ausgangsamplitude gebildet, für das ungünstigste Eingangssignal und wird relativ zu der Registerbreite des Eingangssinals angegeben. Dazu wird die gesamte Übertragungsfunktion verwendet, daher ist j=1 in Gleichung (F.26) zu verwenden.

$$G_{max} = \sum_{k=0}^{(RM-1)N} |h_1(k)|. \tag{F.28}$$

Dieser Ausdruck lässt sich weiter vereinfachen.

Wenn (F.26) mit $j = 1$ und (F.20) kombiniert werden, ergibt sich

$$H_1(z) = \sum_{k=0}^{(RM-1)N} h_1(k) z^{-k} = \left[\sum_{k=0}^{RM-1} z^{-k} \right]^N . \tag{F.29}$$

Wird nun $z = 1$ (also $f = 0$) gesetzt, ergibt sich

$$H_1(1) = \sum_{k=0}^{(RM-1)N} h_1(k) = (RM)^N = G_{max}. \tag{F.30}$$

Ist nun B_{in} die Anzahl der Bits der Eingangssignale, errechnet sich die Anzahl der Bits am Ausgang mit Hilfe von G_{max} nach der Formel

$$B_{max} = \lceil N \, log_2 RM + B_{in} \rceil \text{ Bit.} \tag{F.31}$$

F.3. Simulation mit Scilab

Mit dem Scilab-Programm in Listing F.1 kann ein CIC-Dezimator simuliert werden.

Listing F.1: Simulator eines CIC-Dezimators in Scilab

```
1  // ++
2  // Die Funktion cicsim() simuliert einen CIC-Dezimator mit
3  // den Parametern M, N und R für einen Eingangsvektor x
4  // und gibt einen Ausgangsvektor y zurück.
5  //  --
6  function [y]=cicsim(x, M, N, R)
7   ints=zeros(1,N);
8   comb=zeros(N,M+2)
9   l = 1;
10  // Iteration über alle Samples:
11  for i=1:size(x,'*')
12   // Integrieren:
13   ints = [x(i) ints(1:$-1)] + ints;
14   // Dezimieren:
15   if (modulo(i,R) == 0) then
16    // Differenzieren:
17    comb = [[ints($); comb(1:$-1, $)] comb(:, 1:$-1)];
18    comb(:, $) = comb(:, 1) - comb(:, $-1);
19    y(l) = comb($,$);       l=l+1;
20   end
21  end
22  // Verstärkung kompensieren:
23  y = y/int((M*R)^N);
24 endfunction
25
26 // ++
27 // Die Funktion cicresponse() simuliert die Impulsantwort
28 // eines CIC-Dezimators.
29 //  --
30 function [y]=cicresponse(M, N, R)
31  x = [0 0 1 zeros(1, 1024*2^5+1)];
32  y = cicsim(x, M, N, R);
33  Y = fftw(y);
34  [db,phi] = dbphi(Y/max(abs(Y)));
35  scf;
36  F = (1:size(db,'*'))/size(db,'*');
37  plot2d(F(1:size(F,'*')/2), db(1:size(db,'*')/2));
38  t=sprintf("CIC-Dezimator mit M=%i, N=%i und R=%i", M, N, R);
39  xtitle(t, "F_s/R", "|H(f)| [dB]");
40 endfunction
```

F.4. VHDL-Implementierungen

Der statische CIC-Dezimator besteht aus Integratoren und Differentiatoren, die als Unterkomponenten verwendet werden, und einem One-Hot-Zähler, durch den die Dezimation realisiert wird. Ein Bit des Zählers dient als Clk-Enable der Differentiatoren, wodurch Dekodierlogik für den Zählerstand entfällt und eine schnelle Schaltung resultiert.

Der Integrator integriert das Eingangssignal, indem das aktuelle Eingangssignal in ein Register gespeichert wird und mit dem vorherigen Wert des Registers addiert wird. Der Integrator besitzt einen Enable-Eingang.

Listing F.2: VHDL-Implementierung des Integrators

```
1 --
2 -- SCIC_Integrator_Archi.vhd
3 --
4 -- Frederik Teichert
5 --
6 -- Dieser Block wird von CIC- und SCIC
7 -- Dezimatioren verwendet und bildet
8 -- die Integratoren.
9 --
10 LIBRARY ieee;
11 USE ieee.std_logic_1164.all;
12 USE ieee.numeric_std.all;
13
14 LIBRARY SCIC_lib;
15 USE SCIC_lib.SCIC_Def.all;
16
17 ENTITY SCIC_Integrator IS
18  GENERIC(
19   REG_WIDTH   : natural := 18;
20   ATTENUATION : natural := 0
21  );
22  PORT(
23   Clk         : IN     std_logic;
24   ClkEnable   : IN     std_logic;
25   Rst         : IN     std_logic;
26   DataIn      : IN     signed ( REG_WIDTH-1 DOWNTO 0 );
27   DataOut     : OUT    signed ( REG_WIDTH-1 DOWNTO 0 )
28  );
29 END SCIC_Integrator;
30
31 ARCHITECTURE SCIC_Integrator_Archi OF SCIC_Integrator IS
32  signal delay : signed (REG_WIDTH-1 downto 0);
33 BEGIN
34 -- Ausgang:
35 DataOut <= shift_right(delay, ATTENUATION);
36
37 Integrieren : PROCESS (Clk, Rst) is
38 begin
```

```
39  if Rst='0' then
40   -- reset:
41   delay <= to_signed(0, delay'length);
42  elsif Clk'event and Clk='1' then
43   if ClkEnable='1' then
44    -- Integrieren:
45    delay <= delay + DataIn;
46   end if;
47  end if;
48 end process;
49 END ARCHITECTURE SCIC_Integrator_Archi;
```

Um die Differentiation durchzuführen, werden jeweils M Eingangssignale in Register gespeichert und vom aktuellen Eingangssignal subtrahiert. Das Ergebnis wird in ein Register geschrieben, das anschließend zum Ausgang geführt wird, um eine schnelle Schaltung zu erzeugen. Um eine Dezimation durchzuführen, wird der Enable-Eingang des Differentiators durch ein Bit des One-Hot-Zählers gespeist.

Listing F.3: VHDL-Implementierung des Differentiators

```
1 --
2 -- SCIC_M_Comb_Archi.vhd
3 --
4 -- Frederik Teichert
5 --
6 -- Dieser Block wird von CIC- und SCIC
7 -- Dezimatioren verwendet und bildet
8 -- die Differentiaton über M Werte.
9 --
10 LIBRARY ieee;
11 USE ieee.std_logic_1164.all;
12 USE ieee.numeric_std.all;
13 LIBRARY SCIC_lib;
14 USE SCIC_lib.SCIC_Def.all;
15
16 ENTITY SCIC_M_Comb IS
17  GENERIC(
18   REG_WIDTH : natural   := 18;
19   M         : natural   := 1
20  );
21  PORT(
22   Clk       : IN     std_logic;
23   Rst       : IN     std_logic;
24   DataIn    : IN     signed ( REG_WIDTH-1 DOWNTO 0 );
25   DataOut   : OUT    signed ( REG_WIDTH-1 DOWNTO 0 );
26   ClkEnable : IN     std_logic
27  );
28 END SCIC_M_Comb;
29
30 ARCHITECTURE SCIC_M_Comb_Archi OF SCIC_M_Comb IS
31  type T_MCombDelay is array ( 0 to M ) of signed ( REG_WIDTH-↩
32                               1 downto 0 );
33  signal delays : T_MCombDelay;
34 BEGIN
35 --   ,------------------------.
36 --   |                        V +
37 -- --o->[z^-1] .. -->[z^-1]->(+)->[z^-1]-->
38 --        (1)           (M)     -
39
40 -- Instanziiert einen Differentiator
41 -- mit M Verzögerungsgliedern und
42 -- einem Pipelineregister am Ausgang.
43
```

```
44 -- Ausgang:
45 DataOut <= Delays(Delays'high);
46
47 Differenzieren : PROCESS (Rst, Clk, ClkEnable) is
48  variable i : integer;
49 BEGIN
50  if Rst='0' then
51   -- Reset:
52   for i in 0 to M loop
53    delays(i) <= to_signed(0, delays(0)'length);
54   end loop;
55  elsif Clk'event and Clk='1' then
56   if ClkEnable='1' then
57    -- Aktuellen Wert speichern:
58    delays(0) <= DataIn;
59    -- Delays takten:
60    for i in 1 to M-1 loop
61     delays(i) <= delays(i-1);
62    end loop;
63    -- Ausgang:
64    delays(M) <= DataIn - delays(M-1);
65   end if;
66  end if;
67 END PROCESS;
68
69 END ARCHITECTURE SCIC_M_Comb_Archi;
```

Der CIC-Dezimator instanziiert jeweils N der oben genannten Komponenten, die miteinander verbunden werden.

Der Prozess *DecimationAndOutput* realisiert einen One-Hot-Zähler, mit Hilfe dessen die Dezimation und die Datenausgabe durchgeführt wird. *IntGen* und *CombGen* instanziieren N Integratoren bzw. Differentiatoren.

Listing F.4: VHDL-Implementierung des statischen CIC-Dezimators

```
1 --
2 -- CICDecimator_Static_Archi.vhd
3 --
4 -- Frederik Teichert
5 --
6 -- Ein statischer CIC-Dezimator,
7 -- der durch M,N,R und DATA_WIDTH
8 -- parametrisierbar ist.
9 --
10 LIBRARY ieee;
11 USE ieee.std_logic_1164.all;
12 USE ieee.numeric_std.all;
13 USE ieee.math_real.all;
14
15 LIBRARY SCIC_lib;
16 USE SCIC_lib.SCIC_Def.all;
17
18 ENTITY CICDecimatorStatic IS
19  GENERIC(
20   N           : natural := 4;
21   M           : natural := 1;
22   R           : natural := 16;
23   DATA_WIDTH : natural := 18
24  );
25  PORT(
26   Clk    : IN    std_logic;
27   ClkEn : IN    std_logic;
28   Rst    : IN    std_logic;
29   Dav    : OUT   std_logic;
30   DIn    : IN    signed ( DATA_WIDTH-1 DOWNTO 0 );
31   DOut   : OUT   signed ( DATA_WIDTH-1 DOWNTO 0 )
32  );
33 END ENTITY CICDecimatorStatic;
34
35 ARCHITECTURE CICDecimatorStatic_Archi OF CICDecimatorStatic ↩
36 IS
37  -- Konstanten:
38  CONSTANT cRegGrowth : natural :=   natural(ceil(real(N)
39                                    * log2(real(R)*real(M))));
40  CONSTANT cRegWidth  : natural := DATA_WIDTH+cRegGrowth;
41
42  -- One-Hot-Zähler für die Dezimation:
```

```
43  SIGNAL counter : std_logic_vector(R-1 downto 0);
44  SIGNAL sDav    : std_logic;
45
46  -- Verbindungen.
47  type T_Conn is array (natural range <>) of signed(cRegWidth-
48                        1 downto 0);
49  SIGNAL int  : T_Conn(N downto 0);
50  SIGNAL comb : T_Conn(N downto 0);
51
52  -- Deklarationen der Komponenten:
53  COMPONENT SCIC_Integrator IS
54   GENERIC(
55    REG_WIDTH   : natural := pcREG_WIDTH;
56    ATTENUATION : natural := 0
57    );
58   PORT(
59    Clk       : IN    std_logic;
60    ClkEnable : IN  std_logic;
61    Rst       : IN    std_logic;
62    DataIn    : IN    signed ( cRegWidth-1 DOWNTO 0 );
63    DataOut   : OUT   signed ( cRegWidth-1 DOWNTO 0 )
64   );
65  END COMPONENT;
66
67  COMPONENT SCIC_M_Comb IS
68   GENERIC(
69    REG_WIDTH        : natural   := 18;
70    M                : natural   := 1     --
71   );
72   PORT(
73    Clk       : IN    std_logic;
74    Rst       : IN    std_logic;
75    DataIn    : IN    signed ( cRegWidth-1 DOWNTO 0 );
76    DataOut   : OUT   signed ( cRegWidth-1 DOWNTO 0 );
77    ClkEnable : IN    std_logic
78   );
79  END COMPONENT;
80 BEGIN
81
82 -- Data-Valid ausgeben:
83 Dav <= sDav;
84
85 DecimationAndOutput : PROCESS (Clk, Rst, ClkEn) is
86 begin
87  if Rst='0' then
88   counter <= (others => '0');
89   counter(0) <= '1';
90   sDav <= '0';
91  elsif Clk'event and Clk='1' then
92   sDav <= '0';
93   if ClkEn='1' then
```

```
 94      counter <= counter(counter'high-1 downto 0) & counter(
 95                  counter'high);
 96      sDav <= counter(0); -- oder so..
 97
 98      if counter(0) = '1' then
 99       DOut <= resize(shift_right(comb(comb'high), cRegGrowth),
100               DOut'length);
101      end if;
102    end if;
103  end if;
104 end Process Counting;
105
106 -- Verkabelung:
107 int(0)  <= resize(DIn, int(0)'length);
108 comb(0) <= int(int'high);
109
110 -- Instanziierung der Komponenten:
111 IntGen : for i in 1 to N generate
112  CICIntegrator : SCIC_Integrator
113   generic map (cRegWidth, 0)
114   port map (
115    Clk       => Clk,
116    ClkEnable => ClkEn,
117    Rst       => Rst,
118    DataIn    => int(i-1),
119    DataOut   => int(i)
120   );
121 end generate;
122
123 CombGen : for i in 1 to N generate
124  CICComb : SCIC_M_Comb
125   generic map (cRegWidth, M)
126   port map (
127    Clk       => Clk,
128    ClkEnable => sDav,
129    Rst       => Rst,
130    DataIn    => comb(i-1),
131    DataOut   => comb(i)
132   );
133 end generate;
134 END ARCHITECTURE CICDecimatorStatic_Archi;
```

G. Anhang für Sharpened Cascaded Integrator Comb

G.1. Simulation mit Scilab

Mit dem Scilab-Programm in Listing G.1 kann ein SCIC-Dezimator simuliert werden.

Listing G.1: Simulation eines SCIC-Dezimators in hardwareoptimierter Form mit Scilab

```
1 // scic_ng.sci
2 // Simuliert einen SCIC-Dezimator
3 // in hardwareoptimierter Form.
4 //
5 //              ,->(-2)--->[int^N]-->[/R]->[comb^N]-.
6 //             /                                     \
7 //x->[int^2N]-{                     y<-[comb^2N]<-(+)
8 //             \                                     /
9 //              `->(3R^N)->[delay1]->[/R]->[delay2]-'
10 /////////////////////////////////////////////////////////
11
12 // ++
13 // Die Funktion scicsim() simuliert einen SCIC-Dezimator mit
14 // den Parametern N und R für einen Eingangsvektor x
15 // und gibt einen Ausgangsvektor y zurück.
16 //  --
17 function [y]=scicsim(x, N, R)
18  int1   = zeros(1, 2*N);
19  int2   = zeros(1, N);
20  comb1  = zeros(N, 1+2); // = M+2
21  comb2  = zeros(2*N, 1+2);
22  delay1 = zeros(1, N-1);
23  delay2 = zeros(1, N+1);
24  l      = 1;
25
26  // Iteration über alle Samples:
27  for i=1:size(x, '*')
28  // int1 integrieren:
29  int1 = [x(i) int1(1:$-1)] + int1;
30  // Pfad aufteilen und multiplizieren:
31  // Integrieren:
32  int2 = [2*int1($,$) int2(1:$-1)] + int2;
33  // Verzögerung:
34  delay1 = [R^2*3*int1($,$) delay1(1:$-1)];
```

```
35  // Dezimation:
36  if (modulo(i, R) == 0) then
37   // Combs im oberen zweig:
38   comb1 = [[int2($); comb1(1:$-1, $)] comb1(:, 1:$-1)];
39   comb1(:, $) = comb1(:, 1) - comb1(:, $-1);
40   // Delays im unteren Zweig:
41   delay2 = [delay1($) delay2(1:$-1)];
42
43   // Addition und Differentiation:
44   comb2 = [[delay2($)-comb1($,$); comb2(1:$-1, $)] comb2(:, 1:
45           $-1)];
46   comb2(:, $) = comb2(:, 1) - comb2(:, $-1);
47   // Zuweisung des Ausgangssignals:
48   y(l) = comb2($,$)/int(R^(3*N));
49   l = l+1;
50   end
51  end
52 endfunction
53
54 // ++
55 // Die Funktion scicresponse() simuliert die Impulsantwort
56 // eines SCIC-Dezimators.
57 //  --
58 function [y]=scicresponse(N, R)
59  x=[0 1 zeros(1, 1024*2^4)];
60  y = scicsim(x, N, R);
61  Y = fftw(y);
62  [db,phi] = dbphi(Y/max(abs(Y)));
63  scf;
64  F = (1:size(db,'*'))/size(db,'*');
65  plot2d(F(1:size(F,'*')/2), db(1:size(db,'*')/2));
66  t = sprintf("SCIC-Dezimator mit N=%i und R=%i", N, R);
67  xtitle(t, "F_s/R", "|H(f)| [dB]");
68 endfunction
```

G.2. VHDL-Implementierungen

Der statische SCIC-Dezimator weist eine kompliziertere Realisierung auf als der statische CIC-Dezimator. Eine große Schwierigkeit stellt die Synchronisation beider Pfade dar, da das Übertragungsverhalten des SCICs bereits bei geringsten Unterschieden verfälscht wird.

Der Prozess *DecimationAndOutput* realisiert die Dezimation und gibt die Daten aus. Der Prozess *AddAndDelay* verzögert die Daten im unteren Pfad und führt beide Pfade wieder zusammen. *Int1Gen*, *Int2Gen*, *Comb2Gen* und *Comb3Gen* instanziieren Integratoren bzw. Differentiatoren des CIC-Dezimators.

Listing G.2: VHDL-Implementierung des statischen SCIC-Dezimators

```
 1 --
 2 -- SCICDecimatorStatic_Archi.vhd
 3 --
 4 -- Frederik Teichert
 5 --
 6 -- Ein statischer SCIC-Dezimator, der durch
 7 -- N, R und DATA_WIDTH parametrisierbar ist.
 8 --
 9 LIBRARY ieee;
10 USE ieee.std_logic_1164.all;
11 USE ieee.numeric_std.all;
12 USE ieee.math_real.all;
13
14 LIBRARY SCIC_lib;
15 USE SCIC_lib.SCIC_Def.all;
16
17 ENTITY SCICDecimatorStatic IS
18  GENERIC(
19   N            : natural := 2;
20   R            : natural := 2;
21   DATA_WIDTH   : natural := 18
22  );
23  PORT(
24   Clk          : IN    std_logic;
25   ClkEn        : IN std_logic;
26   Rst          : IN    std_logic;
27   Dav          : OUT   std_logic;
28   DIn          : IN    signed ( DATA_WIDTH-1 DOWNTO 0 );
29   DOut         : OUT   signed ( DATA_WIDTH-1 DOWNTO 0 )
30  );
31 END ENTITY SCICDecimatorStatic;
32
33 ARCHITECTURE SCICDecimatorStatic_Archi OF SCICDecimatorStatic ⤸
34 IS
35  -- Konstanten:
36  CONSTANT cGMAX       : integer := integer(R**(3*N));
37  CONSTANT cRegGrowth : natural := natural(ceil(log2(real( ⤸
38                                   cGMAX))))+1;
```

```
39  CONSTANT cRegWidth   : natural := DATA_WIDTH+cRegGrowth;
40
41  CONSTANT cLog2R      : natural := natural(ceil(log2(real(R))))
42                                     ;
43
44  -- Normierung der Gleichspannungsverstärkung:
45  -- Anteil in Potenz-von-2
46  CONSTANT cGainPow2 : natural := natural(floor(log2(real(
47                                  cGMAX))));
48  -- counter:
49  SIGNAL counter : std_logic_vector(R-1 downto 0);
50  SIGNAL sDav    : std_logic;
51
52  -- Verbindungen der Komponenten:
53  type T_Conn is array (natural range <>) of signed(cRegWidth-
54                        1 downto 0);
55  -- die ersten Integratorn vor der Verzweigung:
56  SIGNAL int1   : T_Conn(2*N downto 0);
57  -- die N Integratoren innerhalb der Verzweigung:
58  SIGNAL int2   : T_Conn(N downto 0);
59
60  -------------------------------------------
61  -- Verzögerung im unteren Zweig in der hohen Taktdomäne
62  --SIGNAL delay1 : T_Conn(N+1 downto 0); -- anzahl muss
63                           anders ermittelt werden!
64  SIGNAL delay1 : T_Conn(N-1 downto 0);
65  -- Verzögerung im unteren Zweig in der niedrigen
66  Taktdomäne
67  SIGNAL delay2 : T_Conn(N+1 downto 0); -- N+1 für combs + 1,
68                         um das Gruppendelay zu
69                         kompensieren!
70
71  -- die N Combs innerhalb der Verzweigung:
72  SIGNAL comb2 : T_Conn(N downto 0);
73  -- die letzten Combs nach der Verzweigung:
74  SIGNAL comb3 : T_Conn(2*N downto 0);
75
76  -- Deklaration der Komponenten:
77  COMPONENT SCIC_Integrator IS
78   GENERIC(
79    REG_WIDTH : natural    := pcREG_WIDTH;
80    ATTENUATION : natural := 0
81   );
82   PORT(
83    Clk        : IN     std_logic;
84    ClkEnable  : IN     std_logic;
85    Rst        : IN     std_logic;
86    DataIn     : IN     signed ( cRegWidth-1 DOWNTO 0 );
87    DataOut    : OUT    signed ( cRegWidth-1 DOWNTO 0 )
88   );
89  END COMPONENT;
```

```
 90
 91  COMPONENT SCIC_M_Comb IS
 92   GENERIC(
 93    REG_WIDTH : natural    := 18;
 94    M         : natural    := 1
 95    );
 96   PORT(
 97    Clk       : IN     std_logic;
 98    Rst       : IN     std_logic;
 99    DataIn    : IN     signed ( cRegWidth-1 DOWNTO 0 );
100    DataOut   : OUT    signed ( cRegWidth-1 DOWNTO 0 );
101    ClkEnable : IN     std_logic
102   );
103  END COMPONENT;
104
105 BEGIN
106 -- Ausgang:
107 Dav <= sDav;
108 -- Verkabelung:
109 int1(0)  <= resize(DIn, int1(0)'length);
110
111 DecimationAndOutput : PROCESS (Clk, Rst, ClkEn) is
112  VARIABLE vComb3SR : signed(cRegWidth-cGainPow2 downto 0);
113  VARIABLE vGAIN    : integer;
114 begin
115  if Rst='0' then
116   counter <= (others => '0');
117   counter(0) <= '1';
118   sDav <= '0';
119   DOut <= to_signed(0, DOut'length);
120  elsif Clk'event and Clk='1' then
121   sDav <= '0';
122   if ClkEn='1' then
123    -- count:
124    counter <= counter(counter'high-1 downto 0) & counter(
125               counter'high);
126    sDav <= counter(0); -- oder so..
127
128    if counter(0) = '1' then
129     -- right shift the result:
130     vComb3SR := resize(shift_right(comb3(comb3'high),
131                 cGainPow2), vComb3SR'length);
132     DOut <= resize(vComb3SR, DOut'length);
133     vGAIN := to_integer(abs(comb3(comb3'high)))/(2**(
134              DATA_WIDTH-1)-1);
135    end if;
136   end if;
137  end if;
138 end Process Counting;
139
140 AddAndDelay : PROCESS (clk, Rst, ClkEn, sDav) is
```

```
141 begin
142  if Rst='0' then
143   delay1 <= (others => to_signed(0, delay1(0)'length));
144   delay2 <= (others => to_signed(0, delay2(0)'length));
145   comb2(0) <= to_signed(0, comb2(0)'length);
146   comb3(0) <= to_signed(0, comb3(0)'length);
147   int2(0)  <= to_signed(0, int2(0)'length);
148  elsif Clk'event and Clk='1' then
149   if ClkEn='1' then
150    -- Verzweigung in die Integratoren
151    int2(0)   <= shift_left(int1(int1'high), 1); --1);
152    delay1(0) <= shift_left(shift_left(int1(int1'high), 1) +
153           int1(int1'high), N*cLog2R);
154    -- delay1 takten:
155    delay1(delay1'high downto 1) <= delay1(delay1'high-1
156           downto 0);
157   end if;
158   if sDav='1' then
159    delay2(0) <= delay1(delay1'high);
160    -- delay1 takten:
161    delay2(delay2'high downto 1) <= delay2(delay2'high-1
162           downto 0);
163
164    comb2(0) <= int2(int2'high);
165    -- Addition beider Zweige:
166    comb3(0) <= delay2(delay2'high) - comb2(comb2'high);
167   end if;
168  end if;
169 end Process AddAndDelay;
170
171 -- Instanziierung der Komponenten:
172 -- die ersten Integratoren vor der Verzweigung:
173 Int1Gen : for i in 1 to int1'high generate
174  CICIntegrator : SCIC_Integrator
175   generic map (cRegWidth, 0) --cLog2R)
176   port map (
177    Clk    => Clk,
178    ClkEnable  => ClkEn,
179    Rst    => Rst,
180    DataIn    => int1(i-1),
181    DataOut   => int1(i)
182   );
183 end generate;
184
185 -- die Integratoren in der Verzweigung:
186 Int2Gen : for i in 1 to int2'high generate
187  CICIntegrator : SCIC_Integrator
188   generic map (cRegWidth, 0) --cLog2R)
189   port map (
190    Clk    => Clk,
191    ClkEnable  => ClkEn,
```

```
192     Rst    => Rst,
193     DataIn    => int2(i-1),
194     DataOut   => int2(i)
195    );
196  end generate;
197
198  -- die Combs in der Verzweigung:
199  Comb2Gen : for i in 1 to comb2'high generate
200   CICComb : SCIC_M_Comb
201    generic map (cRegWidth, 1) --M)
202    port map (
203     Clk    => Clk,
204     ClkEnable  => sDav,
205     Rst    => Rst,
206     DataIn    => comb2(i-1),
207     DataOut   => comb2(i)
208    );
209  end generate;
210
211  -- die Combs nach der Verzweigung:
212  Comb3Gen : for i in 1 to comb3'high generate
213   CICComb : SCIC_M_Comb
214    generic map (cRegWidth, 1) --M)
215    port map (
216     Clk    => Clk,
217     ClkEnable  => sDav,
218     Rst    => Rst,
219     DataIn    => comb3(i-1),
220     DataOut   => comb3(i)
221    );
222  end generate;
223  END ARCHITECTURE SCICDecimatorStatic_Archi;
```

H. Quelltexte des DDC-Simulators

Listing H.1: CIC-Simulator in C

```c
1 /*
2  cic.c
3  Frederik Teichert
4  3.11.2008
5
6  Dieses Programm simuliert einen CIC-Dezimator, der
7  die Daten auf STDIN liest und auf STDOUT wieder
8  ausgibt. Das Datenformat ist dasselbe wie das
9  Ausgabedatenformat
10 des Oszillators.
11
12  Parameter:
13  -R <Dezimationsfaktor>, ist eine potenz von 2!
14  -M
15  -N
16  -n <Anzahl der Werte>
17 */
18
19 #include <stdio.h>
20 #include <math.h>
21 #include <stdlib.h>
22 #include <unistd.h>
23 #include "config.h"
24 #include <string.h>
25
26 #define MAXIMUM ((1<<(BIT_WIDTH-1))-1)
27
28 typedef long long int CIC_Int_Reg[CIC_N];
29 typedef long long int CIC_Diff_Reg[CIC_N][CIC_M+1];
30 typedef struct CIC_s {
31  CIC_Int_Reg I;
32  CIC_Diff_Reg D;
33  int R;
34  long long int y;
35  int M;
36  int N;
37 } CIC_t;
38
39 int cic_calc(CIC_t *cic, long long int *x, int R);
40 void cic_init(CIC_t *cic);
```

```
41 void usage(char *progname);
42
43 void cic_init(CIC_t *cic) {
44  int i, j;
45  for (i=0; i<CIC_N; i++)
46   cic->I[i] = 0LL;
47
48  for (i=0; i<CIC_N; i++)
49   for (j=0; j<(CIC_M+1); j++)
50    cic->D[i][j] = 0LL;
51
52  cic->R = 1LL;
53  cic->y = 0LL;
54  cic->M = 0;
55  cic->N = 0;
56 }
57
58 int cic_calc(CIC_t *cic, long long int *x, int R) {
59  int i, j;
60  cic->I[0] = cic->I[0]+*x;
61  for (i=1; i<cic->N; i++) {
62   cic->I[i] = cic->I[i] + cic->I[i-1];
63
64  }
65
66 #if DEBUGLEVEL>=1
67  //// dump:
68  puts("dump:");
69  for (i=0; i<cic->N; i++)
70   printf("I[%i]=%lld ",i, cic->I[i]);
71  puts("");
72 #endif
73
74  // Differentiation
75  cic->R++;
76  if (cic->R % R == 0) {
77   cic->R = 0;
78   // Delays takten:
79   for (i=0; i<cic->N; i++)
80    for (j=cic->M; j>0; j--)
81     cic->D[i][j] = cic->D[i][j-1];
82
83   // Wert übernehmen:
84   cic->D[0][0] = cic->I[cic->N-1];
85
86   // Weiterrechnen:
87   for (i=1; i<cic->N; i++)
88    cic->D[i][0] = cic->D[i-1][0] - cic->D[i-1][cic->M];
89
90
91 #if DEBUGLEVEL>=1
```

```
92   for (i=0; i<cic->N; i++) {
93    printf("{");
94    for (j=0; j<cic->M+1; j++) {
95     printf("D[%i][%i]=%lld ", i, j, cic->D[i][j]);
96    }
97    puts("} ");
98   }
99
100 #endif
101
102   cic->y = cic->D[cic->N-1][0] - cic->D[cic->N-1][cic->M];
103 #if DEBUGLEVEL>=1
104   printf("y=%lld\n", cic->y);
105 #endif
106   return 1;
107  }
108
109  return 0;
110 }
111
112 int main (int argc, char *argv[]) {
113  CIC_t cic_c, cic_s;
114  int R=1;
115  long long int n=0LL;
116  long long int i;
117  int ch;
118  int M=2;
119  int N=4;
120
121  long long int xc, xs;
122  int cut;
123  char iflag=0, oflag=0; // Binäre Datenein- bzw. Ausgabe
124
125
126  cic_init(&cic_c);
127  cic_init(&cic_s);
128
129  while ((ch = getopt(argc, argv, "n:R:M:N:io")) != -1) {
130   switch (ch) {
131    case 'n': // Anzahl der zu berechnenden Werte lesen:
132     n = atoll(optarg);
133     break;
134    case 'R':
135     R = atoi(optarg);
136     break;
137    case 'M':
138     M = atoi(optarg);
139     if (M > CIC_M) {
140      perror("M is too big, recompile with bigger CIC_M in ⤸
141              config.h! -> exit\n");
142      exit(1);
```

```
143         }
144         break;
145       case 'N':
146         N = atoi(optarg);
147         if (N > CIC_N) {
148           perror("N is too big, recompile with bigger CIC_N in ↩
149                  config.h! -> exit\n");
150           exit(1);
151         }
152         break;
153       case 'i':
154         iflag = 1;
155         break;
156       case 'o':
157         oflag = 1;
158         break;
159       default:
160         fprintf(stderr, "Unknown option %c\nRTFM\n", ch);
161         usage(argv[0]);
162         exit(1);
163     }
164   }
165   argc -= optind;
166   argv += optind;
167
168   cic_c.M = M;
169   cic_s.M = M;
170
171   cic_c.N = N;
172   cic_s.N = N;
173
174   cut = (int)ceil((double)N*log2((double)R*(double)M));
175   fprintf(stderr, "cut=%d\n", cut);
176
177   for (i=0LL; !feof(stdin); i++) {
178     if (iflag) {
179       fread(&xc, sizeof(xc), 1, stdin);
180       fread(&xs, sizeof(xs), 1, stdin);
181     } else {
182       scanf("%lld %lld\n", &xc, &xs);
183     }
184
185     if (cic_calc(&cic_c, &xc, R) | cic_calc(&cic_s, &xs, R)) {
186       // Verstärkung dämpfen:
187       cic_c.y = cic_c.y>>cut;
188       cic_s.y = cic_s.y>>cut;
189
190       if (oflag) {
191         fwrite(&cic_c.y, sizeof(xc), 1, stdout);
192         fwrite(&cic_s.y, sizeof(xs), 1, stdout);
193       } else {
```

```
194     printf("%lld %lld\n", cic_c.y, cic_s.y);
195    }
196   }
197  }
198
199  return 0;
200 }
201
202 void usage(char *progname) {
203  fprintf(stderr, "usage: %s [-oi -R <decimation-factor> -M <M> ⤸
204          -N <N>]\n\t-o\tbinary output\n\t-i\tbinary input\n\t- ⤸
205          R\tdecimation-factor\n\t-M\t<M>\n\t-N\t<N>\n", ⤸
206          progname);
207 }
```

Listing H.2: SCIC-Simulator in C

```
 1 /*
 2  scic.c
 3  Frederik Teichert
 4  10.11.2008
 5 
 6  Dieses Programm simuliert einen SCIC-Dezimator, der
 7  die Daten auf STDIN liest und auf STDOUT wieder
 8  ausgibt. Das Datenformat ist dasselbe wie das ⤸
 9  Ausgabedatenformat
10  des Oszillators.
11 
12  Parameter:
13  -R <Dezimationsfaktor>, ist eine potenz von 2!
14  -N
15  -d (nicht dezimieren)
16  -n <Anzahl der Werte>
17 
18 /////////////////////////////////////////////////////////////
19 //
20 //                  (x1)
21 //          ,->[CIC]---->(-2)--.
22 //         /                    \ (x2)     (x3)       |
23 //  xin o-{                     (+)-->[CIC]-->[CIC]->[VM]->y
24 //         \                    /
25 //          `->[z^-(M-1)]->(3)-'
26 //
27 ////////////////////////////////////////////////////////////
28 */
29 
30 #include <stdio.h>
31 #include <math.h>
32 #include <stdlib.h>
33 #include <unistd.h>
34 #include "config.h"
35 #include <string.h>
36 
37 #define MAXIMUM ((1<<(BIT_WIDTH-1))-1)
38 
39 #define DO_NORMALIZE 1
40 
41 typedef long long int CIC_Int_Reg[CIC_N];
42 typedef long long int CIC_Diff_Reg[CIC_N][CIC_M+1];
43 typedef struct CIC_s {
44  CIC_Int_Reg I;
45  CIC_Diff_Reg D;
46  int R;
47  long long int y;
48  int M;
```

```
49  int N;
50 } CIC_t;
51
52 int cic_calc(CIC_t *cic, long long int *x, int R);
53 void cic_init(CIC_t *cic);
54 void usage(char *progname);
55
56 void cic_init(CIC_t *cic) {
57  int i, j;
58  for (i=0; i<CIC_N; i++)
59   cic->I[i] = 0LL;
60
61  for (i=0; i<CIC_N; i++)
62   for (j=0; j<(CIC_M+1); j++)
63    cic->D[i][j] = 0LL;
64
65  cic->R = 1LL;
66  cic->y = 0LL;
67  cic->M = 0;
68  cic->N = 0;
69 }
70
71 int cic_calc(CIC_t *cic, long long int *x, int R) {
72  int i, j;
73  cic->I[0] = cic->I[0]+*x;
74
75  for (i=1; i<cic->N; i++) {
76   cic->I[i] = cic->I[i] + cic->I[i-1];
77
78  }
79
80 #if DEBUGLEVEL>=1
81  //// Debug:
82  puts("dump:");
83  for (i=0; i<cic->N; i++)
84   printf("I[%i]=%lld ",i, cic->I[i]);
85  puts("");
86 #endif
87
88  // Differenzieren:
89  cic->R++;
90  if (cic->R % R == 0) {
91   cic->R = 0;
92   // Delays takten:
93   for (i=0; i<cic->N; i++)
94    for (j=cic->M; j>0; j--)
95     cic->D[i][j] = cic->D[i][j-1];
96
97   // Wert übernehmen:
98   cic->D[0][0] = cic->I[cic->N-1];
99
```

```
100   // Weiterrechnen:
101   for (i=1; i<cic->N; i++)
102    cic->D[i][0] = cic->D[i-1][0] - cic->D[i-1][cic->M];
103
104 #if DEBUGLEVEL>=1
105   for (i=0; i<cic->N; i++) {
106    printf("{");
107    for (j=0; j<cic->M+1; j++) {
108     printf("D[%i][%i]=%lld ", i, j, cic->D[i][j]);
109    }
110    puts("} ");
111   }
112
113 #endif
114
115   cic->y = cic->D[cic->N-1][0] - cic->D[cic->N-1][cic->M];
116 #if DEBUGLEVEL>=1
117   printf("y=%lld\n", cic->y);
118 #endif
119   return 1;
120  }
121
122  return 0;
123 }
124
125 int main (int argc, char *argv[]) {
126  CIC_t cic_c1, cic_c2, cic_c3, cic_s1, cic_s2, cic_s3;
127  int R=2;
128  long long int n=0LL;
129  long long int i;
130  int ch;
131  int N=2;
132  long long int xc, xs;
133  int cut;
134  char iflag=0, oflag=0; // Binäre Datenein- bzw. Ausgabe
135
136  long long int delay_c[SCIC_DELAY_MAX], delay_s[⤸
137  SCIC_DELAY_MAX];
138  int delay_clks = 0;
139  long long int sum1_c, sum1_s;
140  int nodecimation_flag = 0;
141
142  cic_init(&cic_c1);
143  cic_init(&cic_c3);
144  cic_init(&cic_c3);
145  cic_init(&cic_s1);
146  cic_init(&cic_s2);
147  cic_init(&cic_s3);
148
149  while ((ch = getopt(argc, argv, "n:R:N:iod")) != -1) {
150   switch (ch) {
```

```
151     case 'n': // Anzahl der zu berechnenden Werte lesen:
152      n = atoll(optarg);
153      break;
154     case 'R':
155      R = atoi(optarg);
156      break;
157     case 'N':
158      N = atoi(optarg);
159      if (N > CIC_N) {
160       perror("N is too big, recompile with bigger CIC_N in
161              config.h! -> exit\n");
162       exit(1);
163      }
164      break;
165     case 'i':
166      iflag = 1;
167      break;
168     case 'o':
169      oflag = 1;
170      break;
171     case 'd':
172      nodecimation_flag = 1;
173      break;
174     default:
175      fprintf(stderr, "Unknown option %c\nRTFM\n", ch);
176      usage(argv[0]);
177      exit(1);
178   }
179  }
180  argc -= optind;
181  argv += optind;
182
183  // M=R!!!
184  cic_c1.M = R;
185  cic_c2.M = R;
186  cic_c3.M = R;
187  cic_s1.M = R;
188  cic_s2.M = R;
189  cic_s3.M = R;
190
191  cic_c1.N = N;
192  cic_c2.N = N;
193  cic_c3.N = N;
194  cic_s1.N = N;
195  cic_s2.N = N;
196  cic_s3.N = N;
197
198  delay_clks = N/2*(R-1);
199  fprintf(stderr, "Delay Clocks: %i\n", delay_clks);
200
201  // Die Delays initialisieren.
```

```
202  for (i=0; i<SCIC_DELAY_MAX; i++) {
203   delay_c[i] = delay_s[i] = 0LL;
204  }
205
206  // Normierungsfaktor in Bit pro CIC-Filter:
207 #if DO_NORMALIZE>0
208  cut = (int) ceil(N*log2((double)R));
209 #else
210  cut = (int) ceil(3.0*N*log2((double)R));
211 #endif
212
213  fprintf(stderr, "cut=%d\n", cut);
214
215  for (i=0LL; !feof(stdin); i++) {
216   if (iflag) {
217     fread(&xc, sizeof(xc), 1, stdin);
218     fread(&xs, sizeof(xs), 1, stdin);
219   } else {
220     scanf("%lld %lld\n", &xc, &xs);
221   }
222
223   // Oberer Zweig
224   cic_calc(&cic_c1, &xc, 1);
225   cic_calc(&cic_s1, &xs, 1);
226
227   // Normieren:
228 #if DO_NORMALIZE>0
229   cic_c1.y >>= cut;
230   cic_s1.y >>= cut;
231 #endif
232
233   // Unterer Zweig, Delay:
234   // z^-(R-1) .
235   for (ch=delay_clks; ch>0; ch--) {
236     delay_c[ch] = delay_c[ch-1];
237     delay_s[ch] = delay_s[ch-1];
238   }
239
240 #if DO_NORMALIZE>0
241   delay_c[0] = xc;
242   delay_s[0] = xs;
243 #else
244   delay_c[0] = xc >> ((int) ceil(N*log2((double)R)));
245   delay_s[0] = xs >> ((int) ceil(N*log2((double)R)));
246 #endif
247
248   // Zusammenführung beider Zweige:
249   sum1_c = -2*cic_c1.y + 3*delay_c[delay_clks];
250   sum1_s = -2*cic_s1.y + 3*delay_s[delay_clks];
251
252   cic_calc(&cic_c2, &sum1_c, 1);
```

```
253   cic_calc(&cic_s2, &sum1_s, 1);
254 
255   // Normieren:
256 #if DO_NORMALIZE>0
257   cic_c2.y >>= cut;
258   cic_s2.y >>= cut;
259 #endif
260 
261   sum1_c = cic_c2.y;
262   sum1_s = cic_s2.y;
263 
264   //printf("gelesen: %lld %lld\n", xc, xs);
265   cic_calc(&cic_c3, &sum1_c, 1);
266   cic_calc(&cic_s3, &sum1_s, 1);
267 
268   // Normieren:
269 #if DO_NORMALIZE>0
270   cic_c3.y >>= cut;
271   cic_s3.y >>= cut;
272 #endif
273 
274   if (nodecimation_flag ||( (i%R) == 0LL)) {
275 #if DO_NORMALIZE>0
276    sum1_c = cic_c3.y;
277    sum1_s = cic_s3.y;
278 #else
279    sum1_c = cic_c3.y >> (cut+1);
280    sum1_s = cic_s3.y >> (cut+1);
281 #endif
282 
283    if (oflag) {
284     fwrite(&sum1_c, sizeof(xc), 1, stdout);
285     fwrite(&sum1_s, sizeof(xs), 1, stdout);
286    } else {
287     printf("%lld %lld\n", sum1_c, sum1_s);
288    }
289   }
290  }
291 
292  return 0;
293 }
294 
295 void usage(char *progname) {
296  fprintf(stderr, "usage: %s [-oi -R <decimation-factor> -M <M> ⤸
297         -N <N>]\n\t-o\tbinary output\n\t-i\tbinary input\n\t- ⤸
298         R\tdecimation-factor\n\t-M\t<M>\n\t-N\t<N>\n", ⤸
299         progname);
300 }
```

Listing H.3: FIR-Simulator in C

```
1  /*
2   fir.c
3   Frederik Teichert
4   3.11.2008
5
6   Dieses Programm simuliert ein FIR-Filter mit Koeffizienten,
7   die aus einer externen Textdatei gelesen werden.
8
9   Parameter:
10  -c <coeff-file>
11  -n <Anzahl der Werte>
12  -R <Dezimationsfaktor>
13
14  Das Eingabeformat über STDIN ist wie bei den anderen
15  Programm des DDC-Simulators.
16 */
17
18 #include <stdio.h>
19 #include <math.h>
20 #include <stdlib.h>
21 #include <unistd.h>
22 #include "config.h"
23 #include <string.h>
24
25 #define M ((1<<(BIT_WIDTH-1))-1)
26
27 typedef long long int Fir_Taps[FIR_TAPS];
28 typedef struct FIR_s {
29  Fir_Taps Taps;
30  Fir_Taps Coeffs;
31  long long int y;
32  int tapcount;
33 } FIR_t;
34
35 int fir_init(FIR_t *fir);
36 int fir_calc(FIR_t *fir, long long int x);
37
38 int fir_init(FIR_t *fir) {
39  int i;
40  for (i=0; i<FIR_TAPS; i++) {
41   fir->Taps[i] = 0LL;
42   fir->Coeffs[i] = 0LL;
43  }
44  fir->y = 0LL;
45  fir->tapcount = 0;
46  return 0;
47 }
48
```

```
49 int fir_calc(FIR_t *fir, long long int x) {
50  int i;
51
52  // Delays weiterrücken:
53  for (i=fir->tapcount-1; i>0; i--) {
54   fir->Taps[i] = fir->Taps[i-1];
55  }
56  fir->Taps[0] = x;
57
58 #if DEBUGLEVEL>=1
59  printf("{");
60  for (i=0; i<fir->tapcount; i++) {
61   printf("(%i):%lld ", i, fir->Taps[i]);
62  }
63  puts("}");
64 #endif
65
66  fir->y = 0LL;
67  for (i=0; i<fir->tapcount; i++) {
68   fir->y += (fir->Taps[i] * fir->Coeffs[i]);
69  }
70
71  //fir->y = fir->y >> (DATA_WIDTH-1);
72  /*
73   Cutting im Zweierkomplement rundet immer zur nächsten
74   _kleineren_ Zahl! D.h. der Betrag negativer Zahlen wird
75   größer und der Betrag positiver Zahlen wird kleiner.
76  */
77  //fir->y = fir->y / M;
78
79  //fir->y = fir->y >> (DATA_WIDTH-1);
80
81  // runden:
82  fir->y = fir->y >> (DATA_WIDTH-2);
83  if (fir->y < 0LL) {
84   fir->y = (fir->y+1LL) >> 1;
85  } else {
86   fir->y = (fir->y) >> 1;
87  }
88
89  // Runden mit Zweierkomplement:
90  // (gut)
91  //v = (fir->y < 0LL) ? -1LL : 1LL;
92
93  //fir->y = ((v*fir->y) >> (DATA_WIDTH-1))*v;
94  /*last = fir->y;
95
96  fir->y = fir->y >> (DATA_WIDTH-2);
97  if ((fir->y < 0LL) && (fir->y & 1LL)) {
98   //fprintf(stderr, "%lld\n", (fir->y & 1LL));
99   //fprintf(stderr, "rounding..\n");
```

```
100      fir->y += 2LL;
101     }
102     fir->y = fir->y >> 1;
103     */
104     //fprintf(stderr, "%lld - %lld\n", last/M, fir->y);
105
106  #if DEBUGLEVEL>=1
107     printf("y=%lld\n", fir->y);
108  #endif
109
110     return 1;
111  }
112
113  void usage(char *progname);
114
115  int main (int argc, char *argv[]) {
116     FIR_t fir_c, fir_s;
117     FILE *fp;
118     int ch;
119     char *filename="";
120     double coeff;
121     int i=0;
122     long long int n=0, xc, xs;
123     int R=1, r=0;
124     char iflag=0, oflag=0; // if set, binary input or ouput is
125                used!
126
127
128     fir_init(&fir_c);
129     fir_init(&fir_s);
130
131     while ((ch = getopt(argc, argv, "n:c:R:io")) != -1) {
132      switch (ch) {
133       case 'n': // Anzahl der zu berechnenden Werte lesen:
134        n = atoll(optarg);
135        break;
136       case 'c':
137        filename = optarg;
138        break;
139       case 'R':
140        R = atoi(optarg);
141        break;
142       case 'i':
143        iflag = 1;
144        break;
145       case 'o':
146        oflag = 1;
147        break;
148       default:
149        fprintf(stderr, "Unknown option %c\nRTFM\n", ch);
150        usage(argv[0]);
```

```
151      exit(1);
152    }
153  }
154  argc -= optind;
155  argv += optind;
156
157
158  if ((fp = fopen(filename, "r")) == NULL) {
159   perror("Could not open file");
160   exit(1);
161  }
162
163  while((!feof(fp)) && (i < FIR_TAPS)) {
164   fscanf(fp, "%lf\n", &coeff);
165   fir_c.Coeffs[i] = round(1.0*M*coeff);
166   fir_s.Coeffs[i] = fir_c.Coeffs[i];
167 #if DEBUGLEVEL >= 1
168   printf("Coeff(%i) = %lld\n", i, fir_c.Coeffs[i]);
169 #endif
170   i++;
171  }
172
173  fir_c.tapcount = fir_s.tapcount = i;
174  fprintf(stderr, "tapcount: %d\n", i);
175
176  r=0;
177  // Berechnung der Werte:
178  for (i=0LL; !feof(stdin); i++) {
179   if (iflag) {
180    fread(&xc, sizeof(xc), 1, stdin);
181    fread(&xs, sizeof(xs), 1, stdin);
182   } else {
183    scanf("%lld %lld\n", &xc, &xs);
184   }
185
186   if (fir_calc(&fir_c, xc) | fir_calc(&fir_s, xs)) {
187    r++;
188    if (r==R) {
189     // Ausgabe und Dezimation:
190     if (oflag) {
191      fwrite(&fir_c.y, sizeof(fir_c.y), 1, stdout);
192      fwrite(&fir_s.y, sizeof(fir_s.y), 1, stdout);
193     } else {
194      printf("%lld %lld\n", fir_c.y, fir_s.y);
195     }
196     r=0;
197    }
198   }
199  }
200
201  return 0;
```

```
202 }
203
204 void usage(char *progname) {
205  fprintf(stderr, "usage: %s [-oi -c <coeff-file> -R <
206          decimation-factor>]\n\t-o\tbinary output\n\t-
207          i\tbinary input\n\t-R\tgain-factor\n\t-c\tcoeff-file,
208          a text-file that contains the normalized
209          coefficients\n",
210  progname);
211 }
```

Abbildungsverzeichnis

2.1. Reelles Bandpasssignal und korrespondierendes komplexes Tiefpasssignal 5
2.2. Blockschaltbild eines DDC inklusive der analogen Vorstufen 5

3.1. Lookuptable-Oszillator 12
3.2. Hardwareoptimierungen von Lookuptable-Oszillatoren 13
3.3. Blockschaltbild des IIR-Oszillators 15
3.4. Mögliche Lagen der Pole eines IIR-Systems in gekoppelter Form 17
3.5. Abklingen und Aufschwingen eines IIR-Oszillators 18
3.6. Verlauf der Signale bei Neuinitialisierung durch Mitzählen 19
3.7. Neuinitialisierung des IIR-Oszillators bei Nulldurchgängen der Signale 20
3.8. Neuinitialisierung mit Nulldurchgangsdetektion 20
3.9. Lage der Pole eines IIR-basierten adaptiven Oszillators 22
3.10. IIR-Oszillator mit Polverschiebung 22
3.11. Genauigkeit des IIR-Oszillators in Abhängigkeit der Ausgangsfrequenz 23
3.12. Ausgangssignale und Spektren verschiedener Oszillatoren für die Frequenz $\pi/100$ bei 16 Bit Auflösung 25
3.13. Ausgangssignale und Spektren verschiedener Oszillatoren für die Frequenz $\pi/10000$ bei 16 Bit Auflösung 26
3.14. Benötigte Iterationsanzahl N in Abhängigkeit der benötigten Genauigkeit in Bit für die Frequenzen 0 bis $\{\pi/2, \pi$ und $2\pi\}$ 30
3.15. CORDIC-Statemachine 33
3.16. CORDIC-Pipeline-Prozessor 34

4.1. FIR-Filter in Direktform 38
4.2. Beispiel der Transponierung eines Filters 38
4.3. Pol-Nullstellen-Diagramm in der z-Ebene 40
4.4. Quantisierungseffekte eines Equiripple-FIR-Filters 51sten Grades für 8 Bit Auflösung . 41

5.1. FIR-Dezimationsfilter 43
5.2. Kaskade mehrerer FIR-Halfbandfilter 44
5.3. Amplitudengänge einer FIR-Halfband-Kaskade mit drei Stufen 44

6.1. Struktur von CIC-Dezimator und -Interpolator 47
6.2. Struktur der Differentiatoren- bzw. Integratorsektion eines CIC-Filters 48
6.3. Bausteine von Cascaded-Integrator-Combs 48
6.4. CIC-Amplitudengang 49
6.5. Umstellung von Dezimator und Interpolator eines CIC-Filters zwecks Hardwareoptimierung 51
6.6. Aufbau von CIC-Dezimatoren und -Interpolatoren 52
6.7. Schematische Darstellung der Verringerung der Register in einem CIC-Dezimator . . . 53
6.8. Blockschaltbild eines optimierten CIC-Dezimators mit $M = 1$, $N = 4$ und $R = 25$ nach [CIC] 56

7.1. Blockschaltbild eines SCIC-Dezimators . . . 60
7.2. Amplitudengang eines SCIC-Dezimators für R=4 und N=2 . . . 61
7.3. Hardwareoptimierung eines SCIC-Dezimators . . . 63
7.4. Vergleich der Übertragungsfunktionen von SCIC- und CIC-Filtern . . . 64

8.1. Schema eines FPGAs . . . 72
8.2. Logikblock eines FPGAs . . . 72
8.3. Schematische Darstellung der Verbindung zweier Logikzellen in einem FPGA . . . 73
8.4. Ablauf von Design und Entwicklung einer FPGA-Hardware . . . 74

9.1. Implementierung des Lookuptable-Oszillators . . . 75
9.2. Implementierung des zählenden IIR-Oszillators . . . 76
9.3. Implementierung des überwachten IIR-Oszillators . . . 77
9.4. Implementierung des adaptiven IIR-Oszillators . . . 78

10.1. FIR-Addier-Blöcke . . . 81
10.2. Implementierung des FIR-Dezimationsfilters mit Hardwaremultiplizierern . . . 83
10.3. Implementierung des FIR-Dezimationsfilters mit vierschrittigen Booth-Multiplizierern 84

11.1. Implementierung des FIR-Halfband-Dezimationsfilters . . . 88

12.1. Datenfluss zweier VHDL-Komponenten . . . 91
12.2. Implementierung eines Integrators in Pipelinestruktur . . . 92
12.3. Implementierung eines Differentiators in Pipelinestruktur . . . 93
12.4. Darstellung der Implementierung des CIC-Dezimators . . . 94
12.5. Impulsantworten eines CIC-Dezimators . . . 95
12.6. Passbanddroop und Aliasing von CIC-Dezimatoren . . . 95

13.1. Hardwarereduzierte SCIC-Implementierung . . . 98
13.2. Impulsantworten eine SCIC-Dezimators . . . 99
13.3. Passbanddroop und Aliasing von SCIC-Dezimatoren in Abhängigkeit von R und N . . 99

14.1. Funktionsbeweis des Digital-Down-Converters . . . 109

15.1. Realisierungsmöglichkeiten der Filterung und Dezimation eines DDCs . . . 111
15.2. Vergleich von Dezimationsfiltern für R=16 . . . 256 . . . 113
15.3. Vergleich von Dezimationsfiltern für R=256 . . . 4096 . . . 114
15.4. Vergleich von Dezimationsfiltern für R=8192 . . . 131072 . . . 115
15.5. Vergleich der Impulsantworten für FIR-Dezimationsfilter und Halfbandfilter-Kaskaden für kleine Dezimationsfaktoren . . . 116
15.6. Vergleich der Impulsantworten für Dezimationsfilter mit SCIC- und CIC-Dezimatoren und Halfbandfiltern für mittlere Dezimationsfaktoren . . . 117
15.7. Impulsantworten für Dezimationsfilter aus CIC- und SCIC-Dezimatoren und Halfbandfiltern . . . 118
15.8. Vergleich der Chipauslastung verschiedener Digital-Down-Converter . . . 119

B.1. Blockschaltbild des selbstfüllenden Lookuptables . . . 133
B.2. Blockschaltbild des Lookuptable-Oszillators . . . 136
B.3. Blockschaltbild des adaptiven IIR-Oszillators . . . 145

Tabellenverzeichnis

3.1. Fehler der Taylorreihe für Sinus und Cosinus in Abhängigkeit der Iterationsanzahl N . 30
3.2. Erreichbare Genauigkeit in Bit der Taylorreihe für Sinus in Abhängigkeit der Schrittanzahl N und der maximalen Frequenz 30
3.3. CORDIC-Modi 32
3.4. Ergebnisdarstellung in Abhängigkeit von m, Z_k und Y_k für den CORDIC-Algorithmus 33
3.5. Bewertung der Oszillatoren 36

5.1. Bewertung der FIR-Filter 45

6.1. Passbanddroop eines CIC-Dezimators in Abhängigkeit von N 55
6.2. Aliasrejection eines CIC-Dezimators in Abhängigkeit von N 55

7.1. Bewertung der Dezimatoren 66

9.1. Syntheseergebnisse verschiedener Oszillatoren für die Ausgangsfrequenz $\pi/20$ und 16 Bit Genauigkeit von Quartus 7.2 für einen Stratix III FPGA 79

10.1. Syntheseergebnisse verschiedener FIR-Filter für Halfbandkoeffizienten und Koeffizienten für direkte Dezimation 85

11.1. Syntheseergebnisse verschiedener FIR-Halfbandfilter mit unterschiedlichen Multiplikationsmethoden 89

12.1. Syntheseergebnisse eines CIC-Dezimators mit $N = 4$, $M = 1$ und 18 Bit Eingangsdaten von Quartus 7.2 für einen Stratix III FPGA 96

13.1. Syntheseergebnisse eines SCIC-Dezimators mit $N = 4$ und 18 Bit Eingangsdaten von Quartus 7.2 für einen Stratix III FPGA 100
13.2. Passbanddroop und Aliasrejection in dB eines CIC- bzw. SCIC-Dezimators in Abhängigkeit von R, M bei $f_C = 1/8$ 101
13.3. Passbanddroop und Aliasrejection in dB eines CIC- bzw. SCIC-Dezimators in Abhängigkeit von R, M bei $f_C = 1/32$ 102

16.1. Optimale Realisierung eines Digital-Down-Converters für einen gegebenen Dezimationsfaktor 123

A.1. Booth-Faktoren für 2-Bit-Sequenzen 131
A.2. Booth-Faktoren für 3-Bit-Sequenzen 131
A.3. Booth-Faktoren für 4-Bit-Sequenzen 131

Quellcodeverzeichnis

10.1. Beispiel für Koeffizientenübergabe für FIR-Filter in VHDL 82
14.1. Beispiel einer Textdatei zur Speicherung von Signalen 108
14.2. Unix-Kommandozeilenbefehl zur Simulation eines DDCs 108
B.1. VHDL-Code des selbstberechnenden Lookuptables 133
B.2. VHDL-Code des Lookuptable-Oszillators 136
B.3. VHDL-Code des zählerbasierten IIR-Oszillators 139
B.4. VHDL-Code des überwachten IIR-Oszillators 142
B.5. VHDL-Code des Kerns des adaptiven IIR-Oszillators 145
B.6. VHDL-Code des seriellen Multiplizierers des adaptiven IIR-Oszillators 148
B.7. VHDL-Code adaptiven IIR-Oszillators (Toplevel) 151
C.1. VHDL-Implementierung eines normalisierten 4-schrittigen Booth-Multipliers 155
C.2. VHDL-Implementierung eines normalisierten seriellen Multipliers 158
D.1. VHDL-Implementierung eines FIR-Filters mit Hardwaremultipliern 161
D.2. VHDL-Implementierung eines FIR-Filters mit Booth-Multipliern 164
E.1. VHDL-Implementierung eines FIR-Halfband-Filters mit Hardware-Multipliern 169
E.2. VHDL-Implementierung eines FIR-Halfband-Filters mit seriellen Multipliern 174
F.1. Simulator eines CIC-Dezimators in Scilab 183
F.2. VHDL-Implementierung des Integrators 184
F.3. VHDL-Implementierung des Differentiators 186
F.4. VHDL-Implementierung des statischen CIC-Dezimators 188
G.1. Simulation eines SCIC-Dezimators in hardwareoptimierter Form mit Scilab 191
G.2. VHDL-Implementierung des statischen SCIC-Dezimators 193
H.1. CIC-Simulator in C 199
H.2. SCIC-Simulator in C 204
H.3. FIR-Simulator in C 210

Literaturverzeichnis

[PR96] Proakis, John G.; Manolakis, Dimitris G.: Digital Signal Processing (Third Edition). Prentice-Hall 1996, ISBN 0-13-373762-4.

[MW08] Werner, Martin: Digitale Signalverarbeitung mit MATLAB-Praktikum. Vieweg 2008, ISBN 978-8348-0393-1.

[DSP1] Oppenheim, Alan v.; Schafer, Roland W.: Digital Signal Processing. Prentice-Hall International Editions 1975, ISBN 0-13-214107-8 01.

[UMB1] Meyer Bäse, U.: Digital Signal Processing with Field Programmable Gate Arrays (Second Edition). Springer 2004, ISBN 3-540-21119-5.

[SUS] Scheithauer, Rainer: Signale und Systeme. B. G. Teubner Stuttgart, ISBN 3-519-06425-1.

[MPD1] Donadio, Mattew P. : CIC Filter Introduction. Iowegian 2000.

[MB1] Brambilla, Marco; Liberali, Valentino : Efficient Implementation of Multiplier-Free Decimation Filters for $\Sigma\Delta$ A/D Conversion. Department of Electronics, University of Pavia.

[MB2] Brambilla, Marco; Guidi, Daniele: High Speed FIR Filters for Digital Decimation.

[CIC] Hogenauer, Eugene B.: An Economical Class of Digital Filters for Decimation and Interpolation. IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-29, NO. 2, April 1981.

[HK77] Kaiser, J.; Hamming, R.: Sharpening the Response of a Symmetric Nonrecursive Filter by Multiple Use of the Same Filter. IEEE Transactions on Acoustics, Speech and Signal Processing, VOL. ASSP-25, NO. 5, October 1977.

[SCIC] Kwentus, Alan Y.; Jiang, Zhongnong; Willson, Alan N.: Application of Filter Sharpening to Cascaded Integrator-Comb Decimation Filters. IEEE Transactions on Signal Processing, VOL 45. NO. 2, Feburary 1997.

[NASA] New Horizons REX Instrument CDR, Johns Hopkins University, Applied Physics Lab, 25. July 2003.

[RCH1] Reichardt, J.; Schwarz, B.: VHDL-Synthese - Entwurf digitaler Schaltungen und Systeme. 3. Auflage, ISBN 3-486-27384-1.

[SCI1] Campbell, S.; Chancielier, J.-P.; Nikoukhah, R.: Modeling and Simulation in Scilab/Sicos. 2005, ISBN 0-387-27802-8.

[ALT1] ALTERA: Stratix III Device Handbook. http://www.altera.com/literature/hb/stx3/stratix-3_handbook.pdf, May 2008.

[DiEl1] Maini, Anil K.: Digital Electronics Principles Devices and Applications. ISBN 978-0-470-03214-5 (HB).

[PAPFS] Papula, Lothar: Mathematische Formelsammlung. 8. Auflage, Vieweg Verlag, ISBN 3-528-74442-1, 2003.

[WGO] Smith III, Julius O.; Cook, Perry R.: The Second-Order Digital Waveguide Oscillator. Center of Computer Research in Music and Acoustics (CCRMA), Department of Music, Standford University, Oct. 1992.

[LaTeX] Niedermair, Elke; Niedermair, Michael: LaTeX das Praxisbuch. Franzis 2006, ISBN 3-7723-6930-8.

Eidesstattliche Versicherung

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §25(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Hamburg, den 11. März 2009

Frederik Teichert